CS223 UNIX/LINUX LAB MANUAL

Tony Sako

Description: UNIX is *the* Operating System to learn. This section provides some background into UNIX and it's history, and compares UNIX and Linux. Also during this section you will be given an account on the school UNIX computer, and learn how to log in and change your password.

Objectives: At the end of this section the student will be able to:

1. Describe the similarities and differences between UNIX and Linux.
2. Define an Operating System and list it's functions.
3. Provide a description of the UNIX OS.
4. Compare and contrast UNIX to other operating systems.
5. Compare the UNIX file system to the DOS FAT file system and NTFS.
6. Log on and off the CBC UNIX system.

Study Guide: To pass the test on this topic, you will need to know the following terms and concepts:

1. **Operating Systems**
   - What is an Operating System
   - What does an Operating System do for you
   - Command line interface vs. GUI
2. **History of UNIX**
   - Flavors of UNIX
     - SYSTEM V (AT&T/Bell Labs Version)
     - BSD Berkeley (UC Berkeley Version)
     - Linux (Slackware, Redhat, Ubuntu, Debian, etc.)
   - UNIX and the Internet
3. **Main Features of UNIX**
   - Multi-user & multi-tasking
   - Hierarchical directory structure
   - Portability (90% of source code in C)
   - Very powerful (lots of utilities available)
   - GUI - X Windows
4. **The UNIX Operating System Consists of**
   - Kernel
   - Shell(s)
   - Utilities
5. **The UNIX File System**
   - Hierarchical structure
   - A single tree (root directory)
   - Inode table
6. **Logging In**
   - *telnet (or putty)*
   - Changing passwords
   - Getting help
   - Exiting

Assignments

1.1. Before doing these assignments you will need to get three pieces of information from the instructor so that you can access your account.

(1) **Username** - This is your individual unique identification for UNIX server. The UNIX server is case sensitive, so make sure that you understand exactly which letters are upper and lower case.

(2) **Password** - You will be given an *initial* password that you must change once you login. When you change your password the UNIX server will force you to choose a strong password, which means: at least 8 characters long, a mix of alphanumeric and non-alphanumeric characters, a mix of upper and lower case alpha characters, and no dictionary words.

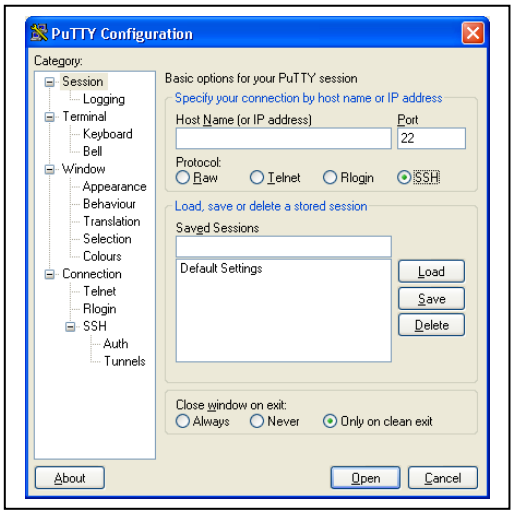
(3) **UNIX Server IP Address** - The UNIX server is not registered in DNS, so you will have to use it's IP address to access it.

Do **NOT** forget your user name or password or you will be unable to access the UNIX server.

Username: _____
Password: _____
IP Address: _____

1.2. You will use a program called *putty* to connect to the UNIX host in the CBC lab. To install and run *putty*, do the following:

A. If you are running Vista or Windows 7, you *may* need to add a rule to your firewall to allow putty access to the network. (See the sidebar below).
B. Start Windows (if it's not already started, login as the default login).
C. Go to the Start Menu and choose My Computer
D. Open the folder O:\cs223. If you are not on campus, you can download Putty from the Internet. (Search for "putty download".)
E. Copy the file putty.exe from the network drive to My Documents (or some other location) on your local computer.
F. Double-click the file putty.exe
G. The Putty Configuration window will appear. This allows you to configure the current putty session, which determines which remote computer to connect to and some specifics about making the connection.



Make sure that **Session** is selected in the **Category** pane.

- In the hostname box enter **134.39.191.100** (This is the IP address for the CBC UNIX server).
- In the protocol section choose SSH. This will automatically set the port number to 22.
- Make sure the **Always** button is checked in the **Close Window on Exit** checkbox section.

Optionally you may select the **Window/Colors** item in the **Category** pane.

- Set your preferred colors for background, foreground and cursor

Return to the **Session** section in the **Category** pane.

- You can save this configuration by selecting the Save button and supplying a name. The next time you use putty you can retrieve the settings you have just entered by selecting the load button, or double-clicking your saved session name.

H. When you have finished the configuration hit the Open button (it's in the lower right corner.) This will start a telnet window and open a connection to the specified server. The first time you connect to a host, putty will warn you that the host does not have a stored key. This is part of the encryption process. Click the Yes or Accept button. (There might be a delay between the window opening and the display of the login prompt as the encryption keys are exchanged and processed.) You will now see a "login prompt" which looks something like the following:

login as:

- Enter your user name. **Remember**: UNIX is case sensitive, make sure the caps lock is off, make sure you use lower case for everything unless you are absolutely sure you want upper case. You will now see a "password prompt" which looks something like the following:

username@134.39.191.100's password:

- Enter your password. **Remember**: UNIX is case sensitive. If you enter the password correctly you will then see a message about when you last logged in, followed by a command prompt.

I. Once you have successfully logged in to a system, the first thing you should do is change your password. This is done via the ***passwd*** command. You will be asked for your old password, then twice for your new password to ensure you typed it in correctly.

```
cs123> passwd
Changing password for cs123
Enter old password:
Enter new password:
Re-type new password:
Password changed.
```

The system does some basic checking of the new password you type in and will prompt you if your password doesn't meet some minimum parameters, such as at least 8 characters long, etc. I will check your password to make sure you changed it.

J. Setting a rule in the Windows Firewall.

- Copy the putty program to your computer, and leave it somewhere where you can find it (such as desktop or documents.
- Start the Firewall by choosing Start > Run and typing Firewall
- Add a new rule (upper right)
- Choose program
- Browse and find the putty program and select it
- Say OK to all of the other Firewall prompts

1.3. The last thing you need to do is log out. This is done by typing the command: `exit`

This will end your UNIX session, and close your Putty window.

1.4. As you start to use other UNIX commands you may notice that some characters sent by the UNIX system are displayed "funny" by putty. To fix this, login to the UNIX system and type: `setenv LANG "en_US"`

(You will have to do this every time you log in, until you learn how to add it to one of your start-up files later in the quarter.)

1.5. Who owns UNIX ? (You will probably have to do some searching on the Internet to find the answer).

1.6. Do some research and find information about at least three different flavors of Linux. How are they different from UNIX? How are they different from each other? How are they similar?

1.7. Do some research and find out how much it costs to purchase a version of UNIX, and how much it costs to purchase a version of Linux.

1.8. Find at least three web sites that have UNIX/LINUX basic tutorials.

1.9. See if you can discover what version of UNIX/LINUX is being run on the CBC server.

Review Questions

1. List the major tasks provided by any Operating System.

2. Name two other Operating Systems, besides UNIX.
3. Why is UNIX important ? That is, why should you bother learning it?
4. Name some strengths and weaknesses of UNIX.
5. Would you rather have OS source code, like Linux, or get it from vendor, like Windows 98. Name the advantages and disadvantages of each.
6. When you run the *telnet* program, it's code/instructions are loaded into memory on a computer. Is this computer the Windows machine you're sitting at, or the UNIX computer ?
7. When you log on to the UNIX computer, and run a program such as *passwd*, where is this program run, on the Windows machine you're sitting at, or the UNIX computer ?
8. An Intel (or compatible) processor can only run a single instruction at one time. How then does a multi-user, multi-tasking operating system run more than one program at the same time? Describe the process.
9. What are some of the features of a strong password?
10. Is it easier to copy a file by drag and drop, or by typing a command? That is, would you rather use a GUI interface or a command line interface?
11. What is the difference between built in commands and utility programs
12. On a UNIX system, are there any programs running besides user programs?
13. Do all programs have to be run by someone (started, stopped).
14. How big is the UNIX OS vs Windows? What part makes it big?
15. Who owns UNIX?
16. Where was UNIX invented?
17. What is relationship between Linux and UNIX. Who owns Linux ?
18. Name the advantages and disadvantages to running Linux vs. running a commercial version of UNIX.
19. How do your commands get from the PC to the UNIX box ?
20. Is there a program running on the UNIX host to handle your incoming commands?
21. What are new concerns appear when sharing system between many users (as opposed to Win 95, as opposed to many users on one machine)?
22. Should there be some method to guarantee one user doesn't "hog" the entire disk on a UNIX system?
23. Can two users have the same account name? password?

Description: Just like Windows based computers have Documents and Settings, Program Files and the Windows or WINNT folders, UNIX has some standard directories. This section introduces you to the layout of the UNIX file system, and the commands for listing the contents of a directory, and finding out other information about the system and what is happening on the system.

Objectives: At the end of this section the student will be able to:
1. Draw a picture of the UNIX file system and identify key directories.
2. Explain the concept of a home directory, and identify their own home directory.
3. Move around the file system to any given location.
4. Identify their current location in the file system at any time.
5. List directory contents, and provide other information about the file system.
6. Provide information about their account.
7. Identify who is logged on the system and what they are doing.

Study Guide: To pass the test on this topic, you will need to know the following terms and concepts:

1. **The UNIX hierarchical file system**
- Main directories and files
- Home directory
- *cd* and *pwd*
- Absolute and Relative Pathnames

2. Directory information
- *ls* (and options)
- hidden files
- . and ..
- *du* and *df*

3. Information about you and your account
- *who am i* and *whoami*
- *env*
- *ps*
- *finger* and *chfn*

Exercises

Basic Command Structure

In this section you will learn how to use several commands. To execute a command or run a program you simply type the command name. In general the things you type on the command line will have this structure:

```
command  -options  arguments
```

You will always type the command, but some commands may also have *options* and *arguments*. Options are typically designated by a "-", and modify the things the command does. You can also provide arguments to some commands. The arguments are not the disagreement type of argument; it's a programming term for things that you supply to the command. The arguments are typically things you want the command to work on. For example, a file that you want to delete would be an argument.

The UNIX file system, `ls`, `cd` *and* `pwd`

One of the first things to do is become familiar with the layout of the files and directories in the UNIX file system. In DOS you would use the `CD` and `DIR` commands to move around the file system and list the contents of the various directories. In UNIX, you use the `cd`, `ls` and `pwd` commands.

2.1 *Listing the contents of a directory*

`ls` - list or display the files and other content of a directory. The `ls` command is similar to `DIR` in DOS; it displays a list of most of the files in the directory. Type the command: `ls`

Since new users don't have many files in their home directory you may not see much. Or, if your system administrator created some files for you, their names will be displayed.

If `ls` doesn't list any files it doesn't necessarily mean you don't have any files, since you probably have some hidden files. On UNIX systems, any file that has a name that starts with a period or dot is considered hidden and will not be displayed by running ls. If you want ls to display all of the files, including the hidden files you need to give it the `-a` (for all) option. Type: `ls -a`

Note – the hidden files are hidden for a reason, they're usually configuration files. This means you don't want to mess with them unless you have an idea of what it is you're doing. UNIX has other ways to protect files besides just hiding them, unlike DOS where the Hidden Attribute was often set to try and protect files.

To see more detailed information about each file, use the `-l` (for long listing) option. Type: `ls -al`

Can you decipher what the extra information means for each file? Compare this to what you see when you run the `DIR` command under DOS.

2.2 `cd` (change directory) This command is used to change the current directory and works much the way the DOS CD command does. For example, to move to the /usr directory you would type: `cd /usr`

You can either supply an absolute or a relative path as an argument to the `cd` command. Any path that starts with a slash "/" is an absolute path, and instructs the OS to start at the top level directory and move you from there to the folder you specified. For example `cd /usr/bin` will move you to the /usr/bin directory regardless of which folder you're in when you type the command.

Any path that starts with (nearly) any other character will be a relative path, which means you are providing direction to start moving from where you're at to somewhere else. For example `cd junk` will move you to a subdirectory named junk.

There are two important directories that are used in relative paths, "." and ".." The single dot stands for the current directory, and two dots means the parent directory or the directory above the one you're currently in. So to move up one directory you would type `cd ..` If you typed `cd .` you would move to the same directory you're already in. This may seem useless, but you'll use it later.

Another important symbol that you can use with `cd` is the tilde "~" (on the top row of keys on the keyboard, to the left of the "1" key). This represents your home directory. If you type `cd ~` you will be moved to your home directory. This saves you from having to type something like `cd /home/my_account`

For each of the following commands, enter the current working directory after the command is executed. (If it's not obvious, you can use the pwd command to have the system tell you where you are.)

```
cd /usr        _____________________
cd /           _____________________
cd /etc        _____________________
cd /usr/bin    _____________________
```

Assume that you want to move the folders in the list below. What command would you use to change to each directory.

```
/usr/bin                                    _____________________
The directory above the current directory    _____________________
The subdirectory named web_stuff            _____________________
The subdirectory rc.d in the /etc folder    _____________________
```

2.3 `pwd` -print working directory (display current directory). When a user is logged onto a UNIX system, they are always somewhere in the file system. Unlike Windows, you can't just "see" where you're at. You either remember,

or you have to use the `pwd` command to figure out where you are. It displays the pathname of the current directory you are in.

Type `pwd` and then write down which directory you're in: _____

Use the `cd` command to change to the following directories, and then use `pwd` to identify which directory you're in. Let me know if `pwd` ever reports that you're in a different directory than the one you changed to, because this indicates you've found a rip in the time-space continuum.

```
cd /usr         _____________________
cd /            _____________________
cd ..           _____________________
cd /etc         _____________________
cd ~            _____________________
cd /usr/bin     _____________________
cd              _____________________
```

The UNIX File System

The UNIX file system is similar in many ways to FAT and NTFS, the file systems used by Windows; or the file and folder system used on the MAC. Like FAT and NTFS, the UNIX file system uses directories to organize files, and these directories can contain their own sub-directories. Of course there are technical differences in the way the file systems are implemented, but there are also a couple of significant differences that users will immediately see.

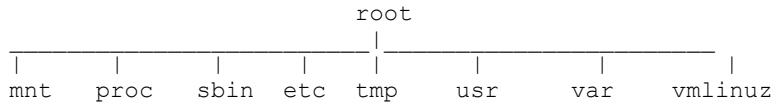
The first difference is that the directory names are not the same. Although UNIX has directories that are very similar in function to Program Files, WINNT or Windows, and Documents and Settings or Users, the names are different.

The second difference is that UNIX has a single top directory "/" which is called root, and any other filesystems are mounted as subdirectories under root, instead of as separate drive letters.

If you were to do an `ls -al` on the root (top level) directory of a Linux system you would see something like the following. (NOTE – your results will be similar, but will probably not be exactly the same.)

```
drwxr-xr-x  18 root     root         1024 Apr 18 14:24 ./
drwxr-xr-x  18 root     root         1024 Apr 18 14:24 ../
drwxr-xr-x   2 root     root         1024 Sep  4  1995 mnt/
dr-xr-xr-x   5 root     root            0 Apr 18 04:05 proc/
drwxr-x--x   3 root     root         1024 Apr 16 09:44 root/
drwxr-xr-x   2 root     bin          2048 Apr 16 14:36 sbin/
drwxrwxrwt   3 root     root         1024 Apr 18 16:15 etc/
drwxrwxrwt   3 root     root         1024 Apr 18 20:15 tmp/
drwxr-xr-x  18 root     root         1024 Apr 18 14:27 usr/
drwxr-xr-x  14 root     root         1024 Apr 16 14:36 var/
-rw-r--r--   1 root     root       360819 Apr 18 08:51 vmlinuz
```

Those entries which are (sub) directories are preceded with a 'd' character. If this were to be drawn or envisioned graphically it would look like



2.4 Using the `cd` and `ls` commands, move around the UNIX file system and explore the various files and folders. See if you can find the equivalents to the following standard Windows folders. You may have to do some Internet research or reading to figure out the UNIX names for these directories. A good place to start is by using typing `man`

`hier`. This runs the `man` command, which is used to display the online documentation which is known as the manual pages. It displays the `hier` page, which many Linux systems use to explain the various folders/directories.

Documents and Settings _____
Program Files _____
WINNT (or WINDOWS) _____

/bin This directory contains executable programs which are needed in single user mode and to bring the system up or repair it.

/sbin Like /bin, this directory holds commands needed to boot the system, but which are usually not executed by normal users.

/usr/bin This is the primary directory for executable programs. Most programs executed by normal users which are not needed for booting or for repairing the system and which are not installed locally should be placed in this directory.

/usr/sbin This directory contains program binaries for system administration which are not essential for the boot process, for mounting /usr, or for system repair.

If you want to know more about the differences between the different bin and sbin directories there's a good article at: http://forum.codecall.net/topic/48290-difference-between-bin-sbin-usrbin-usrsbin/#ixzz2CbOirxr2

2.5 By default the `ls` command will show you the contents of the directory you are currently in. This is great if this is what you want to see, but quite often you want to see the contents of a different directory. You can always use the `cd` command to change to that directory, but you can also just tell `ls` which directory you want it to list. This is done by typing `ls`, followed by any options such as `-al` you may want to use, followed by the path of the directory you want listed.

For example, if you want to the contents of the /etc directory type: `ls -al /etc`

Practice this by ensuring you are in your home directory, by typing `cd ~`, and then using `ls` to display the contents of the following directories.

/
/etc
/usr
/bin
/home

One of the problems with the UNIX file system is that it's not obvious where the various drives or partitions have been mounted in the file system. Or for that matter, finding the removable drives and devices such as CD/DVD drives, USB drives, or floppy drives. Of course, there are a couple of commands that you can use to find out this information. The `mount` command shows you where different devices and partitions have been mounted. The `df` (disk free) command also shows you some of this info, but it also reports how much disk space is available on the various partitions and drives.

2.6 Use the `df` or `mount` command to see the partitions on this system. Write down the results.

Will every UNIX system have these same partitions?

2.7 What's going on? The UNIX equivalent to task manager

As you continue snooping around a computer system one of the next things you might want to figure out is what programs or processes are running. On Windows systems you can run Task Manager and find out this information in a couple different levels of detail. On UNIX systems the `ps` (process status) command and the top command are used to show what's currently running.

Type `ps` and write down the results. (We will talk about what these processes are later in the quarter.)

Running `ps` with no arguments shows you the processes that you are currently running, but to see all of the processes being run by all users you must run `ps` with the `-a` option. Try running `ps -a` but don't try to write down the results, just inspect them.

Notice that you get a lot of information, more than will fit on one screen. When this happens you can take the results from the ps command and hand them to another command called more. This is done by using the "|" character which is called a pipe in UNIX. For example, if type `ps -a | more` (You'll learn more about piping later.) The `more` command displays a screen of information, and then waits for you to tell it what to do. Hitting the spacebar tells more to display the next page.

Next try the top command by typing `top`. Try and decipher what this command is showing you. Does it look like the process information for all users or just for you?

2.8 Who are you?

When a user is given an account on a UNIX system there are several pieces of information about that person that are created. The `who`, `whoami` and `finger` commands can be used to find out information about yourself, your account, and what you're doing. Some of these commands can also be used to find out information about other users on the system.

Type the commands:

```
whoami      _____
who am i    _____
w           _____
who         _____
who -u      _____
finger      _____
env         _____
```

2.9 Fun with the `finger` command

In the early days of UNIX and the Internet, before the web, users wanted a way to share their contact information, schedules, and possibly let other people know where they were at a given time. The `finger` command was designed to allow you see this information for other users. By default, the `finger` command will show you your finger information. If you haven't changed it yet, there won't be much to see, but you will see the various categories that can be displayed.

You can use the finger to find out information about other users as well. Actually, this is what it was originally designed for, since you hopefully already know your own finger info. To give another user the finger (this is really what it's called) type the command `finger user`

Try this for some of the other students in the class. You can get their user names by running the `w` or `who` commands. You can also use the `-l` option for `finger` to see more information.

It's theoretically possible to finger users on other UNIX systems if you know their usernames and the hostname. The syntax for this is `finger user@hostname` However, this causes some security problems, so most UNIX administrators have disabled remote finger access.

2.10 Changing your finger information

You can change your own finger information by using the `chfn` command. It will prompt you for your password, then prompt you for your contact information. You can also change your plan by creating or editing a file named `.plan` that must be located in your home directory. (You will learn how to transfer files to your UNIX account in the next section, and how to use the editor later in the quarter.)

Type the command `chfn`

After changing your finger information, verify that it has changed by typing `finger`

2.11 Getting Help

There are a couple of ways for getting help on UNIX systems. The first is a series of files called the man pages that contain information about all the various commands, programming libraries and data files on the system. UNIX documentation used to come in a series of manuals, and the online man pages are simply the electronic versions of the hardcopy manual pages. The man pages are in a special format, not plain text, so you need to use a special utility program called `man`, to look at them.

To use the `man` command type man followed by the name of the command you want help with. For example, to see the man pages for the `ls` command type `man ls`

The `man` command will display the manual page(s) for the specified command (or specified item). It displays the information a page (screen) at a time, along with a prompt. The "`:`" at the lower left of the screen represents the prompt for the `man` command. The `man` commands make it possible to do things like display the next page, go back, search for a string, etc.

At a minimum you need to know that pressing `<space>` (the space bar) will display the next man page, and to quit from `man`, you have to type `q`. To see a list of all the `man` commands you would think that you should type: `man man`
At least this is what I always thought. However, as it turns out the man page for the `man` command doesn't show the different keys to hit; it shows you other information, but not the command keys. It also turns out that the `man` command uses a UNIX utility named `less` to control the scrolling and display of the man page. The `less` utility is similar to the `more` utility, and the name is another UNIX insider pun/joke. So to see the keys you would hit to control the scrolling in the `man` utility you should type: `man less`

In the space provided list at least 5 commands that you can use while in the `man` utility (or `less`), along with a brief description of the behavior

Key	Description
<space>	display the next screen's worth of information

2.12 Help finding the right command with `apropos`

The problem with the `man` command is that it requires you to know the name of the command you want to use. So if you know you want to do something but don't know the command name then `man` is no help. Luckily most UNIX systems also contain a database of the available commands and keywords associated with the commands. There are two commands you can use to get help from the command database.

The first command is `apropos`, which will search the keywords and find any commands that match. To use apropos you type the command name followed by a word or set of words that describe what it is you would like to do. For example, to find commands that would edit files type `apropos edit` The `apropos` command searches through its database and returns all the commands that have the word "edit" in them. When `apropos` searches the database it just matches anything that has the words you gave it. Using the same example of `apropos edit`, it would also match editor, editing; anything with "edit" in the keyword list.

If you give `apropos` more than one keyword it will find any commands that match any of the words in your list. There is no way to make `apropos` match all the words in a phrase, but later on you will learn how to combine commands to make this happen. As a side note, if you use the `man` command with the `-k` option (for keyword) it will act just like `apropos`.

A. Type the command: `apropos edit`

What can you say

B. Type the command: `apropos edit`

`apropos delete`

2.13 Is this the right command? using `whatis`

The second utility that uses the command database is `whatis`, which is like the `man` command but returns a single line result instead of the entire man page. It does this by searching the command database and then returning the single line description for the command. This can be helpful when you think you know what a command does, but may not be sure. You can use `whatis` to check your memory, without having to read the entire man page. The only problem, is that just like the `man` command, `whatis` requires that you remember a valid command name to even start.

Type the commands:

```
whatis ls       _____
whatis cd       _____
```

2.14 What Version of Linux/UNIX is running

Just in case you want to know which specific release of Linux is being run there is one command, `uname`, and one file, /etc/issue that will display this information. There's a second file that may exist on some systems, but is not universally available. On redhat systems it's named /etc/redhat-release

Type the commands:

```
uname -r                _____
cat /etc/issue          _____
cat /etc/*-release      _____
```

Review Questions

1. Will the files and directories on every UNIX system be the same? Similar?
2. Vmlinuz is the kernel, but you can see it. What is it? How big is it? Is it all of the OS? Can you name the Windows equivalent?
3. UNIX supports multiple disks and/or partitions. How does UNIX identify each partition, as a drive letter, or as a subdirectory? Is this better or worse than drive letters? List some advantages and disadvantages.
4. How do you know if a subdirectory is on a different disk or partition? Does it matter?
5. Can you see what others are doing ? Can they see what you are doing ? Can you find out any personal information?
6. What are some useful options for the *ls* command? How do you find a complete list of options?
7. Can you use *cd* or *ls* on every directory. Why or why not ?
8. UNIX supports multiple disks and/or partitions. How does UNIX identify each partition, as a drive letter, or as a subdirectory?
9. Use *ls -al* on your home directory. What files do you see? What are . and .. ?
10. Can you identify the other files in your home directory? Why don't they show up under *ls*?
11. How do you get help on UNIX commands?

Objectives: At the end of this section the student will be able to:

1. Use FTP or some variant to move files between UNIX and the PC.
2. Use the commands for creating and removing files and directories.
3. Use the various commands for viewing the contents of a file.
4. Use the various commands for copying and moving files.
5. Read permissions, and set a file or directory to a given set of permissions.
6. Use *umask* to set the default permissions for new files and directories.

Study Guide: To pass the test on this topic, you will need to know the following terms and concepts:

1. **Creating and deleting files**
   - `touch` - creating a file
   - `rm` - deleting a file
   - `cat` – displaying the content of a file
2. **Working with directories**
   - `mkdir` - creating a directory
   - `rmdir` - deleting a directory
   - `rm -r` - deleting a directory and all sub-directories
3. **Viewing the contents of a file**
   - `file` - view type of file
   - `more, less` or `pg`- a page at a time
   - `head` and `tail` - see start or end of file
4. **Copying and moving**
   - `cp` - copy a file
   - `mv` - move or rename a file
   - `ln` - create a link between files
5. **Permissions**
   - Understanding file and directory permissions
   - `chmod` - changing permissions
   - `umask` - setting default permissions

Exercises

Viewing files.

There are several commands you can use to see the contents of a file. These include `cat`, `more`, `less`, `pg`, `head` and `tail`.

3.1 Viewing files with `cat`. This exercise is an introduction to the cat command. `cat` stands for concatentate and it's the basic utility used to display files. The name may seem a little strange, why not call it `type` like the DOS command? But as you will see in a later section, `cat` can also be used to combine files, or append content to existing files.

Use the `cd` command to change to the /etc directory. Type the following command to view the contents of the file named fstab.

```
cd /etc
cat fstab
```

(fstab contains the filesystem configuration information, that is, which disks/partitions/devices to mount, and where to mount them in the filesystem tree. This isn't really important; what's important is that you learn how to use the `cat` command.) Now use the `cat` command to display the following files in the /etc directory:

passwd

group
aliases

(Once again, you don't need to worry about what's in these files for now. You're just getting practice with the `cat` command.) You can also use `cat` to display files that are in a different directory. To do this, give `cat` the path to the file along with the file name. For example, make sure you are back in your home directory, then display the file /etc/passwd

```
cd ~
cat /etc/passwd
```

In the space below, write what you would type to display the designated files. Assume that you are in your home directory and assume that all files exist

/etc/Muttrc _____
/etc/init.d/sshd _____
/usr/lib/rpm/macros _____
File in current directory named junk _____
File in your bin directory named fix.sh _____

3.2 Viewing Files with `more, less` and `pg`. The `cat` command works fine on small files, that is files that are small enough to fit all of their lines on the screen. But as you've seen cat just dumps the entire file to the screen, so you only see the end of longer files. The solution to this is to use one of the utility programs that displays a single page or screen of information at a time.

The first we'll look at is called `more`. To use more to view a file type `more filename` For example to view the file /etc/passwd type `more /etc/passwd` As you can see can also pass `more` a path to file, just as long as it ends in a file. If you try to run `more` on a directory it will complain.

After `more` has displayed a page of text, it pauses and waits for you to tell it what to do next. It displays a prompt at the bottom of the screen along with what percent of the file has been displayed, to give you an idea of where you're at in the file. For example it may display `-- More -- (19%)` At this point you need to type in one of the more commands to let it know what to do. There are several commands you can use (do a `man more` to see them all), but here are a couple of the more commonly used ones.

`<space>` - display the next screen or page of text
`<enter>` - display the next line
`/string` - search for the next occurrence of the text string
`q` - quit

Use `more` on some of the same files you did in the `cat` exercise.

/etc/passwd
/etc/group
/etc/Muttrc
/etc/init.d/sshd
/usr/lib/rpm/macros

Which command makes it easier to read the file contents, `cat` or `more`?

The `less` command, which is used by the `man` command, is pretty much just like more, except that it allows you to back up and you have to quit at the end. It uses the same commands as `more`, and adds the `b` command to scroll back one screen and the `q` command to quit. The `pg` command is also just like `more` and `less`, however, it is the SYS V version, and has a few differences. It hasn't made it onto all Linux distributions yet, and isn't on the one we currently use at CBC. You should get used to just one of these commands.

3.3 The `more` command is very useful because you can also use it to display the output from other commands. This is done by doing something called piping (which we will talk in detail about later). Piping the output from one command to the more command is done typing the command you want to run, then the "|" character, then more. For example, if you list the contents of the /usr/bin directory it all scrolls by to quickly to see. But it you type:

```
ls /usr/bin | more
```

then the output from the ls command will handed to more, which will display it one screen at a time.

What would you type to see the contents of the following directories, one screen at a time?

/etc _____
/usr/sbin _____
/dev _____
/usr/lib _____

3.4 Viewing Files with `head` and `tail`. Sometimes you just want to see the first few lines in a file, or the last few. You can use more to see the first few, and then quit. But this requires typing the "q" command to quit, and that apparently is too much work, or at least someone thought that typing that extra character was too much effort so they wrote a command called `head` which displays the first 10 lines of a file and then exits automatically.

Try the `head` command by typing `head /etc/passwd`

The `tail` command is just like `head`, except that it displays the last 10 lines. The `tail` command is a little more useful, especially when it comes to looking at long files. You don't have to wait for `cat` to display the entire file, or scroll manually with `more` or `less`.

Try the `tail` command by typing `tail /etc/passwd`

By default both `head` and `tail` display 10 lines, but they also both have an option that allows you to set the number of lines you want to see. For example `head -20 /etc/passwd` would display the first 20 lines.

3.5 Now apply what you've learned about `cat`, `more`, `less`, `head` and `tail`. In the space provided write what you type to accomplish the designated task, using the appropriate command. Note that there may be more than one correct way. Assume that you are in your home directory and assume that all files exist.

1st 15 lines of /etc/Muttrc _____
Last 5 lines of /etc/init.d/sshd _____
The /usr/lib/rpm/macros, one page at a time _____
Last 20 lines of the file in current directory named junk _____
A file in your bin directory named fix.sh, one page at a time _____
The contents of the /bin directory, one page at a time _____

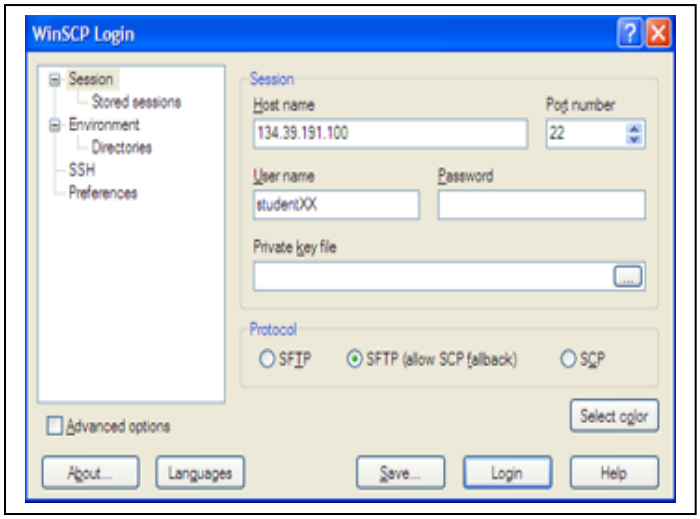
Creating Files, Deleting Files, & Moving Files Between a PC and A UNIX Computer

The next thing to learn is about making files and directories, and moving and deleting files. Since you'll need files to copy you need some ways to get files onto the UNIX system. The easiest way is to create them in an editor, but since you won't learn how to use the editor until a little later we'll look at two other ways of creating files on the UNIX system, and show you how to move files from the PC to the UNIX system.

3.6 **Using FTP** - FTP stands for **F**ile **T**ransfer **P**rotocol, and it is the program you use to move files between the PC and the UNIX host. However, FTP sends all of it's information "in the clear", which means that it is not encrypted. This includes your user name and password, which makes it relatively easy for naughty people to get illegal access to your account and onto our UNIX computer. To alleviate this situation, we will use a version of FTP that performs encryption. The client we will use is called WinSCP (SCP stands for Secure CoPy.) You can find a copy of WinSCP on O:\CS223, but you can also download a copy from the Internet.

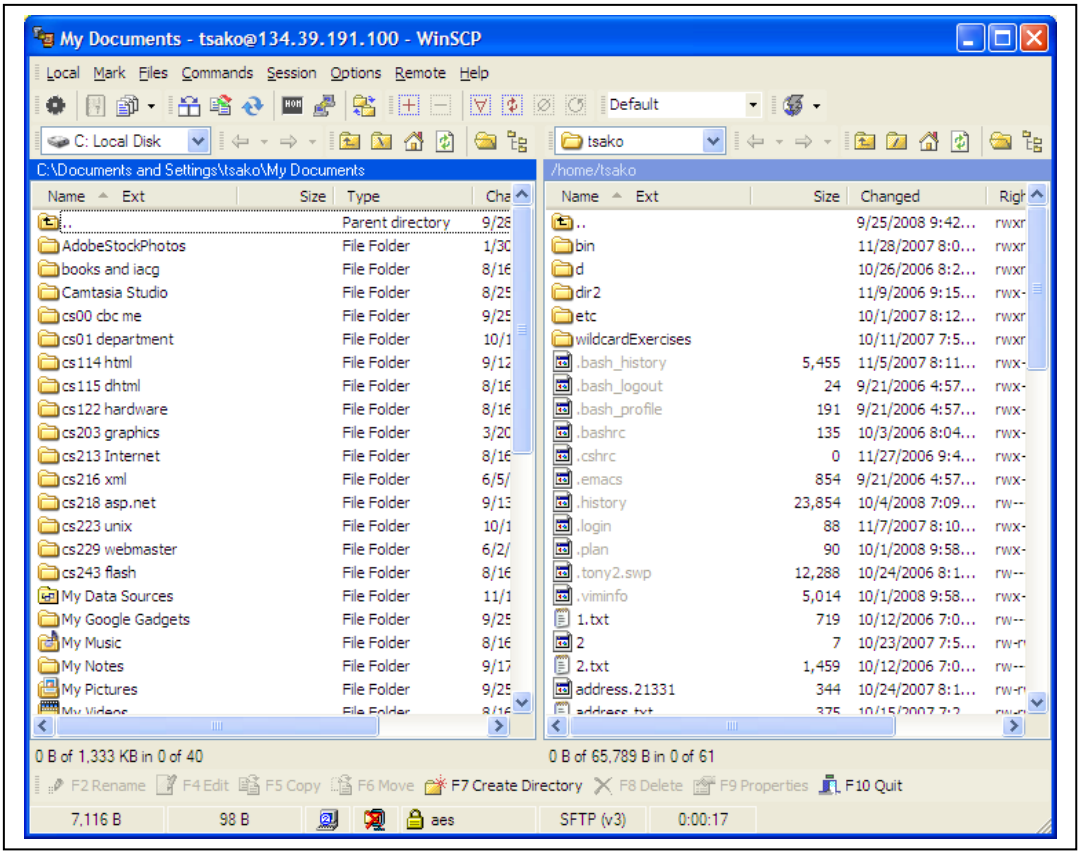
The other thing you'll need before starting is a test file to move back and forth. Use the word processor of your choice on the PC and create a text file on the PC. Make sure and save it as plain text.

Next start WinSCP, you should see a screen like this.



The two key pieces of information on this screen are the **Host Name**, and **User Name**. The Host Name must be set to the IP address of the UNIX host in our lab which is 134.39.191.100. The User Name must be set to ***your* user name**, the one that was assigned to you in lab. You can optionally add your Password, but only do this on a PC that is NOT in a public place, like your home computer or personal laptop. If you add it in one of the computers in the lab it's possible for other users to pull up the settings for your session which will include your password. Leave the Protocol option on **SFTP (allow SCP fallback)**

When you are finished, click the **Login** button, and FTP will connect you to the UNIX computer. If you didn't already enter your password you will be prompted to type it in. Once connected, you will see something similar to the following:



The window on the left represents the files on your PC, the window on the right shows the files on your UNIX account.

You can move files from the PC to the UNIX host by left-clicking on the PC file and then clicking the arrow which points to the UNIX window, or by simply double-clicking the file in the PC window. You can move multiple files by using the <ctrl> or <shift> keys to select multiple files, then clicking the right arrow.

Moving files from the UNIX host to the PC is done following the same process, but in reverse.

Remember that UNIX is case sensitive, while the PC is not.

The rest of the buttons should be self-explanatory. When you are done, click the **Exit** button in the lower right hand corner. If you're working in a public place make sure and quit or close the window so that others cannot access your account.

3.7 Another way to create files on a UNIX computer is to use the `touch` command. To use `touch`, type the command followed by the name of the file you want to create. For example `touch filename` Touch is kind of like a magic wand, you wave it and "ba-wang!" it creates a file.

Using `touch` creates blank files, which may seem like a weird thing to do. It may be useful when you're learning how to copy files and playing around, but is there any practical use for the `touch` command?

3.8 You can also create files and put some text in them by using the `cat` command. (Yes, this is the same `cat` command that you just used to display files.) By default, the `cat` command sends its output, the file you specified, to the screen. But we can also use the `cat` command and something called redirection, which we will study in detail later, to take whatever you type and send it into a file. Redirection is done using the > greater than character. For example: `cat > temp` will take whatever you type and put it in a file called temp. This will continue until you hit <CTRL-D> which means hit the <CTRL> and D keys at the same time.)

You can't really do much editing with this method. You can backspace on the current line, but once you hit <ENTER> the current line is sent to the file and you can't change it.

A. Change to your home directory by typing `cd` or `cd ~`
B. Put the following text in a file named truth.txt by typing `cat > truth.txt` (Ignore any typing mistakes, and do not use the cursor arrow keys):

```
echo Tony is the coolest guy I know
echo I believe everything he says,
echo I will give him all my money
echo UNIX rules !
<CTRL D>
```

C. Use the `cat` or `more` command to display the file truth.txt on the screen and check your input.

3.9 Now that you know how to create files you should learn how to delete them. The UNIX command for getting rid of files is called `rm`, which is short for remove. To remove a file type `rm filename`

Practice what you've learned. Create three files named file1, file2 and file3. You can make empty files or add text to them if you wish. Do an `ls` to ensure that the files exist. Once you're sure the files are there delete them. Do another `ls` to ensure they are gone.

We will look at the `rm` command in the next set of exercises, and revisit it again when you learn about using wildcards.

Filenames and File Extensions

Now that you know how to create files it's time for a quick discussion about file names. The main rule for file names is that you can use alphanumeric characters and a few of the special characters such as "." dot and "_" underscore. You should avoid all the other characters such as "*" and even spaces as they have special meaning.

Most UNIX systems support file names up to 255 characters in length, so you should make them long enough to be descriptive but short enough to work with. If you want to use multiple words in the file name you can use dots or underscores between words, or use the camelcase method where you capitalize the first letter of each new word. An example of camelcase would be recyclingMemoDraft.

Another rule is that UNIX/Linux is case sensitive. So it would see FILE1, File1 and file1 as three different files.

If you start a file name with a dot or period, then it is considered a hidden file. For example, the file .login would be hidden, but the file myProject would not be. It's not really hidden, it just won't be displayed if you use the `ls` command without the `-a` option. It's just a quick, built-in method for making the `ls` display a little cleaner.

If you've grown up on a Windows or DOS based computer then you're probably used to the way that files are named, and can tell what type a file is just by looking at the 3 character extension. For example, on a Windows based computer the file program1.exe would be an executable program, the file report.txt would be a text file, and the file song.mp3 would be an audio file. These file extensions came from DOS, which uses what is called the 8.3 naming convention. The first 8 characters are the file name and the last 3 are called the extension, which describes the type of content in the file.

In the UNIX/Linux universe files don't use the extension to describe what they are. You can do this, but you'll run into many files that don't have an extension. And you will also see many files that have more than one dot in the filename, or a one or two character extension. There are certain de-facto conventions used in UNIX, such as the .sh extension indicating a shell script or .c indicating a C program source file, but these conventions are not strictly enforced or universally used like they are in the Windows environment.

3.10 Since you don't always know what kind or type of data is in a file by looking at the name there's a command called `file` that will help. Type in the following to see a demonstration of the `file` command. See if you can decode what the `file` command is telling you about the various file types.

```
file /etc/passwd
file /usr/bin/zip
file /usr/include/crack.h
file /usr/share/man/man1/kill.1.gz
```

Creating and Deleting Directories

Hopefully you know that you realize the value of using directories to keep your files organized. (If not, I'll have you try to help one of my relatives with his computer. He's one of those who keeps all of his files in the same directory, all of them. Trying to help him is always a real treat.) This section shows you how to create directories using the `mkdir` command, and delete directories using the `rm` and `rmdir` commands.

3.11 The command to create a directory is `mkdir`. The most common way to use `mkdir` is to give it a new directory name. It will then try to create a sub directory with the name you provided, in the current directory. For example `mkdir temp` would make a subdirectory named temp in the current directory.

The second way is to give `mkdir` an absolute path that ends in a new directory (not a file). In this case it would try to create the new directory exactly where you told it to. For example `mkdir /home/studentXX/temp`
In the space provided write what you type to create the specified directory. Assume that you are in your home directory.

bin _____
jokes _____
jokes/knock-knock _____
jokes/puns _____
project1 _____
project2 _____

You should actually make all of the directories as you will need them in the next exercise.

3.12 The command to remove a directory is `rmdir`. Like `mkdir`, you can give it a directory name or a path and it will delete the specified directory. For example `rmdir temp` There's just one thing wrong with `rmdir`, it only works if the directory you want to delete is empty. If there are any files or sub directories in the directory you are trying to delete `rmdir` stops and displays a message like `rmdir temp: Directory not empty`

Another way to to remove a directory is to use the `rm` command with the `-r` option. This option stands for recursive, and it removes the directory and everything inside of it. This is much easier than using `rmdir` if you have a directory with a bunch of files or sub directories inside of it.

Try `rmdir` on the three directories you made in the previous exercise. If `rmdir` does not work, then try `rm -r`

Copying and Moving Files

3.13 The command for copying files or directories is `cp`. In it's most basic form you give the `cp` command the name of the file you want to copy and the name for the duplicate file. These are generically called the source and the destination. For example `cp /etc/passwd passwd.mine` would make a copy of /etc/passwd and put it in a new file named passwd.mine.

The cp command can also be used to copy an entire directory, or to copy a file to a different directory. To copy an entire directory use: `cp -r sourceDirectory destinationDirectory` The `cp` command is smart enough to realize that the thing you want to copy is a directory, and then give a new directory with the name you specified. The new directory will be in the current directory, unless you specify a path.

To copy a file to a different directory use: `cp sourceFile destinationDirectory` The `cp` command is smart enough to realize that the thing you want to copy is a file, and since the destination is a directory it will put the new file in the directory..

On DOS/Windows systems, the `COPY` command is smart enough to make some assumptions about the source and destination files and directories. For example if you type `COPY SOURCE` and leave off the destination filename or directory, the `COPY` command will assume that you want to copy the `SOURCE` to the current directory. However this doesn't work on UNIX systems. Try to copy the password file to your directory using : `cp /etc/passwd` You should get an error message like `cp: missing destination file` The way to fix this is to use "." which means the current directory. Using the same example it would be `cp /etc/passwd .`

In the space provided write what you would type to copy the specified file to a new file in the current directory. The new file should have the same name as the original file

/etc/passwd _____
/var/log/wtmp _____
/var/www/error/README _____

In the space provided write what you type to copy the specified file to a new file in the current directory. The new file should have the name delete.me

/etc/passwd _____
/var/log/wtmp _____
/var/www/error/README _____

3.14 The command for moving or renaming files is `mv`. It's very similar to the `cp` command in that you can move file or directories, and that you must always specify a destination. However, moving files takes a little more technical awareness than copying files. Moving files is actually done by going into the filesystem table and changing entry for the filename. This means that you can't use the `mv` utility to move files if they aren't on the same partition because this requires making a new entry in the destination filesystem table and then deleting the entry in

the source table. If you need to move files between partitions you can accomplish it by doing it in two steps. First do a copy and then delete the original file(s).

Start this exercise by doing a little setup. First ensure that you are in your home directory. If you're not sure remember you can type `pwd` to see which directory you're in, or type `cd` to change to your home directory. Next create a file named temp1 using the `touch` command. In the space provided write what you type to rename the temp1 so that the new file name is delete.me

_____

For the following exercises first determine if you can move the specified file to your home directory using `mv`, or if it's in a different partition. (Hint, remember that you can use the `df` command to see how the disk partitions are mapped into the file system.) In the space provided write what you would type to move the specified file to a file in the current directory with the same name.

/etc/passwd _____

/var/log/wtmp _____

Security & File Permissions

UNIX has a security system which allows it assign permissions to every file (which includes directories and devices because they are treated as files) that it knows about. The set of permissions is stored with every file, and can be seen by using the `ls -al` command. When you execute the command `ls -al`, it returns a listing with information similar to the following:

```
drwx------   2  tsako    ctc          1024 Apr 16 14:43 .
drwxr-xr-x  124 root     sys          3072 May 10 18:18 ..
-rw-r-----   1  tsako    ctc             0 Jan 22 13:14 .newsrc
-rw-------   1  cs111    student       526 Apr 15 11:03 .login
```

The first part reveals the permission settings of the files (`-rw-r--r--`). These settings have the following meanings:

specifies whether a directory (d), or normal file (-)
permissions for the owner of the file, read(r), write(w), execute (x)
permissions for group members
permissions for others

```
-     rw-   ---   ---
```

Changing access permissions

To change the access permissions for a file or directory use the command `chmod mode filename` The "mode" consists of three parts: who the permissions apply to; how the permissions are set; and the permissions to be set.

Who the permissions apply to is specified as one of:

u	user	- the owner of the file
g	group	- the group to which the owner belongs
o	other	- everyone else, also known as the world
a	all	- u, g and o (the world)

How the permissions are to be set is given as one of:

+	add the specified permission
-	subtract the specified permission
=	assign the specified permission, ignoring whatever may have been set before.

The permissions to be set are specified by one or more from:

r - read **w** - write **x** - execute

Changing access permissions using numbers

There is a shorthand way of setting permissions by using octal numbers. Read permission is given the value 4, write permission the value 2 and execute permission 1. This is how the system actually stores the permissions, as a binary number.

r	w	x	r	w	x
4	2	1	2^2	2^1	2^0

These values are added together for any one permission category:

UNIX	Binary	Conversion step 1	Decimal	
- - -	0 0 0	0 0 0	0	no permissions
- - x	0 0 1	0 0 1	1	execute only
- w -	0 1 0	0 2 0	2	write only
- w x	0 1 1	0 2 1	3	write and execute (1+2)
r - -	1 0 0	4 0 0	4	read only
r - x	1 0 1	4 0 1	5	read and execute (4+1)
r w -	1 1 0	4 2 0	6	read and write (4+2)
r w x	1 1 1	4 2 1	7	read and write and execute (4+2+1)

So access permissions for all three categories; user, group, world; can be expressed as three digits. For example:

	user	**group**	**others**
`chmod 640 file1`	rw-	r--	---
`chmod 754 file1`	rwx	r-x	r--
`chmod 664 file1`	rw-	rw-	r--

3.15 Decode each of the following sets of permissions. Specify what the settings are for the owner, group and world

	Owner	Group	World
`-rwxrw-r--`	_____	_____	_____
`-r-xr-xr-x`	_____	_____	_____
`-rwxr-xr-x`	_____	_____	_____
`-rw-r-----`	_____	_____	_____

3.16 Write down the number you would provide `chmod` for each of the following sets of permissions.

	Owner	Group	World
`-rwxrwxr-x`	_____	_____	_____
`-rwxr-xr-x`	_____	_____	_____
`-r-xr-xr--`	_____	_____	_____
`-rw-r--r--`	_____	_____	_____

Directory Permissions

Permissions have a little different meaning for directories than they do for regular files. The same set of letters and symbols are used, and they still apply to the directory's owner, group and the world; but they have different meanings.

r – Allows the directory listing to be seen. Can do `ls`, but not `cd` or `cat/more` on files
w – Allows you to create/rename/delete files
x – Allows you to `cd` into a directory, and read files with `cat/more` (but you will also need read permissions on each specific file)

Changing default access permissions

(You will need to understand how to set file access permissions numerically before attempting this section.) Every time you create a file or a directory, its access permissions are set to predetermined value. This value is defined by the file-creation mode mask. To display the value of this mask use the `umask` command. `umask` will display something like:

022

The OS uses the mask by subtracting it from 2 other numbers, one for files and one for directories, to determine the default values for anything you create. Unless you're an experienced programmer this probably seems like a convoluted way to do things, but remember UNIX was written by some programmers. The numbers that are used are 666 for files and 777 for directories. The values in the mask are subtracted from these values to give a default value for access permissions for the files and directories created by you. For example:

```
  777   (system value for directories)
- 077   (value of the umask)
  700   (default access permission of  rwx------)
```

Some people see the 666 and regard it as a sign that UNIX is the devil's choice for an OS. This may or may not be true, but it wasn't always 666. Originally both files and directories used 777, but this allowed users to give away execute permission by default. This ended up causing security holes, so sometime in the 1990's a decision was made to change the file number to 666.

To change your default access permissions you use the command: `umask nnn` Where each "n" is number from 0 to 7 Here are some examples of using the `umask` command

A. To give yourself full permissions for both files and directories and prevent the group and other users from having access: `umask 077` This subtracts 077 from the system defaults for files and directories: 666 and 777. The results are default access permissions for your files of 600 (`rw-------`) and for directories of 700 (`rwx------`).

B. To give all access permissions to the group and allow other users read and execute permission: `umask 002` This subtracts 002 from the sytem defaults to give a default access permission for your files of 664 (`rw-rw-r--`) and for your directories of 775 (`rwxrwxr-x`).

C. To give the group and other users all access except write access: `umask 022` This subtracts 022 from the system defaults to give a default access permission for your files of 644 (`rw-r--r--`) and for your directories of 755 (`rwxr-xr-x`).

3.17 For each of the following numbers, write down what default permissions would result.

	files	directories
`umask 066`	_____	_____
`umask 222`	_____	_____
`umask 024`	_____	_____

3.18 Write down the number you would provide `umask` if you wanted to have the following sets of default file permissions.

`-rw-rw-r--`	_____
`-rw-r--r--`	_____
`-r--r-----`	_____

3.19 What are your default file permissions? _____

3.20 What are your default directory permissions? _____

Advanced – how some versions of umask deny execute permissions

(NOTE – this is an advanced topic, and you can skip it or ignore it if you wish. You will NOT be expected to know this for tests or exams.) On some Linux distributions the `umask` command will never allow you to have execute permissions by default. This was done for security purposes, but it really makes `umask` more confusing and hard to use. This doesn't happen on very Linux distribution, but it has been implemented on several.

The way this functions is that umask checks the number you give it; and if it will result in a default execute permission it subtracts one. Subtracting one from the umask number is like adding one to the actual permissions. For example, umask

555 should give you default permissions of 111, or --x --x --x . However, since umask can't possibly allow you to have execute permission by default, it subtracts one from each 5 on the mask value. This results in a umask value of 444 and default permissions of 222 or -r- -r- -r-.

The following table shows how umask changes the mask value for any combination that may result in default execute permissions.

umask value	Results in permission	Symbolic Permission	umask changes this to	Results in permission	Symbolic Permission
5	1	--x	4	2	-w-
3	3	-wx	2	4	r--
1	5	r-x	0	6	rw-

Running Programs and Scripts

Every time you type a command you are running a program. These programs may be compiled executables, or they may be something called shell scripts, which are like DOS batch files. Either way, you just type the name of the command to run it. If you want to make your own shell scripts you can do this by:

A. Add the commands you want to a file. You can use the UNIX editor, create the file on the PC and FTP it over to your UNIX account, or use the `cat` command and the method shown below.
B. Make sure you have execute permission on the file. You will probably have to give yourself execute permission.
C. Tell the UNIX OS that you want to run the file by typing the file name. (You will have to preface the filename with `./` to indicate that it's the file in the current directory.

3.21 Create and execute a shell script by doing the following

a. Ensure that you are in your home directory by typing: `cd ~`
b. Create a file named hello.sh by typing the following:

```
cat > hello.sh
echo "hello world"
echo "the current date and time is: ";  date
<ctrl-d>
```

c. Try and run the file by typing: `./hello.sh`
   You should get an error message

d. Give yourself execute permission by typing: `chmod 700 hello.sh`
e. Try and run the file by typing: `./hello.sh`
   This time it should run successfully

Testing File & Directory Permissions

This exercise provides a way to test the file and directory permissions. You will create a directory, change the permissions; and then try to do several things like copy files into the directory, list the directory contents or execute the script. You will then repeat this process for a different set of directory permissions, until you've tried all 8 possibilities.

3.22 Using the `mkdir` command, create a subdirectory in your home directory named `ptest`. For this exercise, do the `chmod` command using the numbers indicated, **try** and run the commands, and write down the results or error messages. Think carefully about what you are doing, for example, if you cannot copy the file `hello.sh` into the directory ptest, or `cd` into ptest, you may get results anyway!

`cd`	[Get back to your home directory]
`mkdir ptest`	[Make a new subdirectory to do some testing with]
`chmod 000 ptest`	[Set the permissions on the test directory to 000]
`cp hello.sh  ptest/hcopy.sh`	[Try and copy the shell script you made above into the test directory]
`ls ptest`	[Try and list the contents of the test directory]
`cat ptest/hcopy.sh`	[Try and view the contents of the file you copied into the test directory. This won't work if you were not able to copy the file above.]
`ptest/hcopy.sh`	[Try and run the shell script]
`cd ptest`	[Change into the test directory]
`ls`	[List the directory contents. If the `cd` above failed, you will see your home directory]
`cd`	[Get back to your home directory. This command is superfluous if you never left.]
`rm ptest/hcopy.sh`	[Get rid of hcopy.sh. If you were not able to make the copy to start with, this won't work.]

```
chmod 100 ptest
cp hello.sh ptest/hcopy.sh
ls ptest
cat ptest/hcopy.sh
ptest/hcopy.sh
rm ptest/hcopy.sh
cd ptest
ls
cd
```

Keep repeating this set of instructions, but use 200, 300, 400, 500, 600, and 700 for the permissions on ptest.

Review Questions

1. What is the command to view the first 10 lines of the file /etc/passwd?
2. If you use the head command how many lines will you get if you do not specify a number? That is, what is the default?
3. What is the command to view the last 10 lines of the file /etc/passwd?
4. What is the command you would type to view the file /etc/inetd.conf ?
5. Write the command and switch you would use to remove files, but have the utility ask you before deleting the file.
6. What do you have to do to tell the *cp* command to copy to the current directory?

Description: UNIX uses many of the special characters on the keyboard to add power to your commands. UNIX provides a set of wildcards that can be used when working with files, a set of filters and pipes which allow you to take the output from one command and "hand it" to another command, and uses the three different quote characters to confuse new users.

Objectives: At the end of this section the student will be able to:

1. Use wildcards in filenames.
2. Use the various types of quotes correctly.
3. Describe stdin and stdout, and how to use filters and redirection.
4. Explain `noclobber`, and how to set or unset it.
5. Use regular expressions to match strings inside files

1. Command Interpretation and Execution
What happens when a command is executed

2. Metacharacter Expansion
Filename Substitution
globbing
Special characters ? * [ ] { } ~

3. Quoting
Command Substitution w/ `command` (grave quotes or backticks)
Expansion w/ "*.txt" (double quotes)
Normal quote of literal string w/ 'file*' (single quotes)

4. Stringing together multiple commands
Why do more than one command at a time?
Sequential commands with ;
Continuing long lines

5. Redirection
stdin and stdout
redirect symbols - > >> >>!
noclobber

6. Pipelines
pipe symbols - | && ||
grouping commands with ()

7. Pipeline processing commands
grep and *sed*

8. Regular expression wildcards
Matching a single character
Occurrence Modifiers
Positional Modifiers

Exercises

Command Line Basics

Up until this point, UNIX probably seems a lot like DOS, or any other OS that allows you type commands. The commands we've used so far have been nothing special, and you're probably thinking why even bother with the command line when it's so much easier to copy or delete files from a windows GUI.

At this point we're going to start looking at things you can type on the UNIX command line in addition to the basic commands. You'll learn how to do things from the command line that you normally have to write a C/C++ or VB (or other higher level language) to accomplish. The things you'll learn make the command line very powerful, but it can also make it very confusing to the uninitiated.

In this section we're going to look at using wildcards to specify command arguments, and connecting multiple commands with special symbols. This can be a little confusing if the concepts are new, but to make things even more confusing some of the symbols will change meaning depending on where they are used. So you will have to pay attention to the context of each symbol in addition to just learning what the symbol means. And to make things even more interesting, the symbols will behave differently in the different UNIX shells. In most cases these exercises require that you be in the bourne shell (sh) or bourne again shell (bash). They will not work properly in the c-shell (csh) or tcsh. You can temporarily change to bash by typing `bash` at the command prompt. This will last until you type `exit` or close putty. Or you can change your login shell so that each time you login you will automatically be in bash. This is done by typing `chsh`, and then specifying `/bin/bash`.

Before we start making things complicated, let's look at some very simple concepts. These may seem painfully basic, but they will be key later.

The first basic concept is that the simplest thing that you'll type on the command line is a command. The shell will execute this command. When the command has completed the shell will present a prompt and you can type another command. The command may have options, which are usually typed on the command line right after the command itself. For example in `ls -al` the `-al` tells the `ls` command that you want to use the `a` and `l` options. Every command has it's own set of options. A command may also have one or more arguments. For example, in `ls /etc` the directory `/etc` is the argument. Each command has it's own argument list, and specifications for whether the arguments are required or not.

The second basic concept is that if the command needs some input, then the input will come from the keyboard. As an example, think of a text editing program. Once the editor has started, anything you type on the keyboard will be sent to the editing program.

The third basic concept is that output from commands is usually displayed on the screen or monitor. This includes the results of the command, but it also includes error messages.

The last concept is that when a command is done it tells the shell whether it completed successfully or whether it had problems and errors. It does this by returning what is called a status code, which is 0 if it is successful and a different number if it had any errors.

Shell Wildcards & Expansion

4.1 Shell Wildcards & Expansion. When you type a command that requires a filename as an argument, you can use wildcards to match filenames. This is helpful if the filename is long and you don't want to type it all. Or it's also helpful when there are several files you want to process with the command; by using wildcards you can have the command process them all instead of typing in all of the filenames.

In UNIX certain characters, `?, *, [],` and `{}` can be used in filenames that are passed to commands. If they are used, the shell **expands** them before executing the command. This expansion is known as **globbing**. If you're used to the DOS wildcards then some of the UNIX wildcards such as `*` and `?` will look familiar, and their behavior will also be familiar. This set of exercises will provide you some experience in dealing with the wildcards used for matching filenames.

?	csh, sh	Match one character
*	csh, sh	Match any number of characters, with the exception of a leading .
~	csh	Home Directory
~user	csh	*User*'s Home Directory
[123abc]	csh, sh	Match any **single** character in the list
[1-9]	csh, sh	Match any **single** character in the range
{123,abc}	csh, sh	Match any pattern. Patterns are comma delimited

You will need a set of data files for the following exercises. Copy the directory /home/wildcardExercises to a directory under your home directory by typing:

```
cp –r /home/wildcardExercises   ~/wildcardExercises
```

As you go through each exercise remember that the point is for you to see what filenames will be matched by each wildcard.

A. Ensure that you are in your wildcardExercises directory when working on the following exercises. You can use wildcards to help change directories. First move to your home directory by typing: `cd ~`

B. Next, move to the wildcardExercises subdirectory by typing: `cd wildcardExercises`

C. Type: `ls *`

   a. Which files are displayed? Which files are not displayed?

D. Type: `ls .*`

   a. Which files are displayed? Which files are not displayed?

E. Type: `ls .* *`

   a. Which files are displayed? Which files are not displayed?

F. Type: `ls *.*`

   a. Which files are displayed? Which files are not displayed?

G. Type: `ls prog*`

   a. Which files are displayed?

H. Type: `ls prog?`

   a. Which files are displayed?

I. Type: `ls resume[123].*`

a. Which files are displayed?

J. Type: `ls resume[1-4].*`

a. Which files are displayed?

K. Type: `ls resume1.{dat,bak}`

a. Which files are displayed?

L. Type: `ls resume[12].{dat,bak}`

a. Which files are displayed?

M. Type: `ls ~`

a. Which files are displayed? Which directory is this?

N. Type: `ls ~/.*`

a. Which files are displayed?

O. Move to the `/etc` folder by typing: `cd /etc`

Now type: `cd ~`

Which directory do you move to? (Use the `pwd` command to print the current working directory).

4.2 Use the data from the previous exercises to complete the following table. Put a check mark in the csh and sh columns to indicate whether or not that shell supports the wildcard.

Wildcard	csh (tcsh)	sh (bash)	Expands to match
?			
*			
~			
~user			
[123abc]			
[1-9]			
{123,abc}			

Quotes & Backslashes for Protecting Special Characters

A problem with wildcards is that sometimes you may want to use the wildcard characters in a file name or as text, and not have the shell expand them. Plus you will learn about other characters or character sequences that have special meaning, such as `\t` which may be converted to a tab character and `\n` which may be converted to a newline character. This section looks at methods of protecting characters, or telling the shell that sometimes a `*` is just a `*`.

4.3 Quotes & Protecting Wildcards. When you type a command that requires a filename as an argument, you can use wildcards to match filenames. As you saw in the previous exercises this is helpful if the filename is long and you don't want to type it all or when there are several files you want to process with the command. But there may be occasions when you want to prevent the wildcards from being expanded.

Say for example that you have a file named "`topSecret*`". Note that the file name has the "`*`" character in it. (Yes, it is possible to create a file with a "`*`" in its name; it's not easy but it is possible. And yes, it is a very bad idea to have a "`*`" in your file names.) If you wanted to delete this file, and used the command

```
rm topSecret*
```

you would delete all your files that started with `topSecret`. In this case, you would want to tell the shell that you do NOT want it to expand the `*`. This can be done in a couple of ways, either by using single quotes, double quotes or the backslash character `\`. The exercises in this section demonstrate how to protect the wildcards.

First we'll look at how the different quotes affect expansion of wildcards for filename arguments. You will need a set of data files for the following exercises. Copy the directory /home/quoteExercises to a directory under your home directory by typing:

```
cp –r /home/quoteExercises   ~/quoteExercises
```

This set of exercises will demonstrate whether or not the single quote or double quote characters protect the various wildcards from expanding or not. In each exercise, you just need to determine whether the wildcard was expanded or not. If it is expanded you will see a list of files that match the filename and wildcard. If it is not expanded you will see either an error message saying that there is no matching files, or a list of files that match the exact characters you have specified.

A. Ensure that you are in your quoteExercises directory when working on the following exercises. You can use wildcards to help change directories. First move to your home directory by typing: `cd ~`

   Next, move to the quoteExercises subdirectory by typing: `cd quo*`

B. Next we'll check how quotes affect wildcard expansion for matching filenames. First try the wildcard without any quotes. Type: `ls *`

   Which files are displayed?

C. Type: `ls '*'` ← These are single quotes

   Which files are displayed? Is there any difference between using double quotes and using no quotes at all?

D. Type: `ls "*"` ← These are double quotes

   Which files are displayed? Is there any difference between using double quotes and using no quotes at all?

E. Type: `ls j*`

   Which files are displayed?

F. Type: `ls 'j*'` ← These are single quotes

   Which files are displayed?

G. Type: `ls "j*"` ← These are double quotes

   Which files are displayed?

H. Type: `ls "junk?"` ← These are double quotes

   Which files are displayed?

I. Type: `ls 'junk?'` ← These are single quotes

Which files are displayed?

4.4 Quotes and \t \n (print control sequences) You probably noticed that there isn't any difference between using single quotes or double quotes when it comes to wildcard expansion. That is, "junk*" and 'junk*' expand to the same thing. So why have two different sets of quotes? The reason is that there are some other characters that have special meaning to the shell, and the quotes tell the shell whether or not they should have special treatment or whether they're just another text character.

The first special character we'll look at is the "\" back slash character, which is used in a couple of different ways. The first usage is as part of a 2 character sequence that's used in strings to control the display and add tabs, newlines or other characters. If you want to add a tab to a string, add `\t` which the shell will convert to a tab character. For a newline add `\n` which the shell will convert to two characters, the carriage return and line-feed characters.

In this set of exercises you will gain some experience with using `\t` and `\n` to print tabs and newlines; and with using the various quotes to protect the shell from expanding these characters. In each exercise you will see that either `\n` and `\t` are replaced by newlines and tabs, or that they are protected and the characters \n and \t are printed on the screen.

A. Remember that the `echo` command prints strings. (On some versions of UNIX/LINUX and in the `bash` shell the `echo` command requires the `-e` option as a switch to decide whether to expand these sequences or print them literally. If you are running `tcsh`, you will most likely not need the `-e` and can omit it.) Type:

```
echo -e Hello World!
```

What is displayed? Now add some tabs by typing:

```
echo -e \tHello \tWorld!
```

What is displayed? Do you see the `\t` characters, or did they cause something else to happen? Next, add a newline by typing:

```
echo -e \tHello \n\tWorld!
```

What is displayed? Do you see the `\t` and `\n` characters, or did they cause something else to happen?

B. Now test how the slash sequences are affected by double quotes. Type:

```
echo -e "\tHello \n\tWorld!"
```

What is displayed? Are the `\t` and `\n` characters displayed or expanded?

C. Test how the slash sequences are affected by single quotes. Type:

```
echo -e '\tHello \n\tWorld!'
```

What is displayed? Are the `\t` and `\n` characters displayed or expanded?

D. Apply what you've just learned by creating the following display using a *single* `echo` command. Hint, use tab characters to get the columns to line up, and newlines to do multiple lines with a single command.

College	Phone Number
CBC	547-0511
WSU	372-7250

4.5 Quotes and shell variables Another character that has special meaning to the shell is the dollar sign "$", which is used to indicate that something is a shell variable. Those of you who are programmers know what variables are. But for those that don't, a variable is just a place to store information. It's like the different memory buttons on a calculator, but instead of pressing a button to store and retrieve information you use shell commands.

There are several variables that are created for you by the shell. Remember how a process contains both instructions and data. Anytime a user logs in to the UNIX system, the OS starts a shell process for that user. This shell process contains the shell instructions, as well as the data for that specific user. The data is stored in variables called shell variables. All of the shell variables can be seen by typing the `set` command. We'll look at shell variables in greater detail later, for now we'll just look at a couple so you can see how they're affected by the different quotes.

Some of the variables have a fixed, unchanging value; while others are constantly changing. For example, `$user` holds your user or login name, and typically does not change. The variable `$cwd` holds your current working directory, and changes each time you change directories.

The user can access the value of any individual variable by using the `echo` command. For example, to see the value of `$cwd`, type: `echo $cwd` The shell expands `$cwd` to the name of the directory you are currently in, and then hands this text to the `echo` command. So shell variables are another thing the shell will expand, and the $ is your signal to the shell that an item is a shell variable.

A. Test this out by typing: `echo $user $cwd`

What is displayed?

B. Now do some testing to see if you can use the different quote characters to protect the $ and prevent the shell from expanding it. Test the effect of double quotes by typing: `echo "$user $user"` ← These are double quotes

What is displayed? Is there any difference between using double quotes and using no quotes at all?

C. Now test the effect of single quotes by typing: `echo '$user'` ← These are single quotes

What is displayed? Is there any difference between using single quotes and using no quotes at all?

4.6 Using \ to protect characters with special meaning As you have seen, there are many characters, `* ? " ` ,` that have special meaning to the shell or to the various commands such as `echo`. But there may be occasions when you simply want to print one of these characters, and not have it signal the shell to do anything special. For example, you may want to use the echo command to print a "*" or even a double quote character ".

A. One of the ways is to enclose the entire string in single quotes, but what if the character you want to print is a single quote itself? Type:

```
echo 'This is a quote' '
```

Does `echo` print what you expected? Another way to tell the shell to leave your characters alone is to use the slash "\" character. The slash tells the shell to leave the next character alone. In UNIX talk it is said that the slash *protects* the next character. Type:

```
echo 'This is a quote\' '
```

B. So what do you do if you have a section of arguments that you want expanded, but another section that you want left alone. You can mix different quoted sections, or use the "\" to protect single characters. Type:

```
echo -e "\tThis is expanded " '\tThis is not'
```

Does `echo` print what you expected?

C. The use of quotes and what expands or not can be confusing. The confusion can be compounded by the fact that the `echo` command is one of those that behaves differently on different versions of UNIX/LINUX. Luckily, a quick bit of testing or trial can show you exactly how the quotes and the `echo` command behave on a specific UNIX or LINUX system.

For each of the following strings, try them with the `echo` command with: (1) no quotes, (2) single quotes and then (3) double quotes.

```
echo * (star) ~ (tilde) " (quote) \t (tab) \\n (slash n)
echo '* (star) ~ (tilde) " (quote) \t (tab) \\n (slash n)'
echo "* (star) ~ (tilde) " (quote) \t (tab) \\n (slash n)"

echo –e * (star) ~ (tilde) " (quote) \t (tab) \\n (slash n)
echo –e '* (star) ~ (tilde) " (quote) \t (tab) \\n (slash n)'
echo –e "* (star) ~ (tilde) " (quote) \t (tab) \\n (slash n)"
```

4.7 Using the results from your tests, fill in the following chart and show whether or not the characters are protected. For example, if the ~ is protected by a backslash, write yes in the appropriate box. If it is not protected write no.

	* ? [ ] { }	~	$var	!!	\t \n	'	"	\
\ (backslash)								
' (tick)								
" (double quote)								

Back Quotes (grave quotes) – expanding commands

4.8 Back ticks The back ticks instruct the shell to act in a much different way than the other quotes. Instead of expanding wildcards to match filenames, the back ticks tell the shell to run a command, and then use the results of the command as arguments. You will be using the `date` command, which returns information about the current date.

A. Type the following to see what `date` returns:

```
date
```

The `date` command can be told to format it's output using a set of variables. Each variable starts with the percent "%" symbol. For example, %m says to print two numbers for the month. For a complete list of the formatting variables check the man pages for date. All of the format variables must be preceded by a plus "+" character. Check the formatting by typing:

```
date "+%m%d%y"
```
← These are double quotes

B. Now for the back ticks. Run the following command:

```
ls * > junk.`date "+%m%d%y"`
```
← wrap the `date` command in back ticks

Now do an `ls`. Do you see what happened ? (The back ticks told the shell to run the `date` command, then use what `date` returned as part of the filename.)

Variables for the date command	
%%	a literal %
%a	locale's abbreviated weekday name (Sun..Sat)
%A	locale's full weekday name, variable length (Sunday..Saturday)

%b	locale's abbreviated month name (Jan..Dec)
%B	locale's full month name, variable length (January..December)
%c	locale's date and time (Sat Nov 04 12:02:33 EST 1989)
%C	century (year divided by 100 and truncated to an integer) [00'99]
%d	day of month (01..31)
%D	date (mm/dd/yy)
%e	day of month, blank padded (1..31)
%F	same as %Y'%m'%d
%g	the 2'digit year corresponding to the %V week number
%G	the 4'digit year corresponding to the %V week number
%h	same as %b
%H	hour (00..23)
%I	hour (01..12)
%j	day of year (001..366)
%k	hour (0..23)
%l	hour (1..12)
%m	month (01..12)
%M	minute (00..59)
%n	a newline
%N	nanoseconds (000000000..999999999)
%p	locale's upper case AM or PM indicator
%P	locale's lower case am or pm indicator
%r	time, 12'hour (hh:mm:ss [AP]M)
%R	time, 24'hour (hh:mm)
%s	seconds since '00:00:00 1970'01'01 UTC' (a GNU extension)
%S	second (00..60)
%t	a horizontal tab
%T	time, 24'hour (hh:mm:ss)
%u	day of week (1..7); 1 represents Monday
%U	week number of year with Sunday as first day of week (00..53)
%V	week number of year with Monday as first day of week (01..53)
%w	day of week (0..6); 0 represents Sunday
%W	week number of year with Monday as first day of week (00..53)
%x	locale's date representation (mm/dd/yy)
%X	locale's time representation (%H:%M:%S)
%y	last two digits of year (00..99)
%Y	year (1970...)
%z	RFC'822 style numeric timezone ('0500) (a nonstandard extension)
%Z	time zone (e.g., EDT), or nothing if no time zone is deter'minable

Multiple commands

4.9 Multiple commands on the same line with ";" Next we'll look at running more than one command at a time. UNIX makes this very easy and allows you to do some pretty complex operations with just a little bit of typing. But before we start we'll look at why you may want to do more than one command at a time?

Why do you run UNIX commands at all? In general it's either for entertainment or to accomplish some task. If you're trying to get some work done it usually takes more than one command to achieve. For example, you may change directories to see what files exist, and then look at the contents of a specific file. If there's a task that you have to do frequently, and it takes the same series of commands, you can type all of them on the same command line. While this may seem useful, by itself it doesn't really save any time or make you more productive.

However, running multiple commands is a key piece to building UNIX command pipelines. (We will learn the details of pipelining later in this chapter.) UNIX pipelines are ways to do things from the command line that you would have to write a program in a language like C++ or Visual Basic to do in any other OS.

So we'll start simple, executing a couple of commands; then move on to stringing them together to build pipelines.

To run multiple commands type the first command, then use a semi-colon ";" to separate it from the second command. In a generic form this would be: `command1 ; command 2`

When the UNIX shell sees this is knows to run the first command then the second. There's no real connection between commands. You can add as many commands as you want by separating them with semi-colons.

A. Display the contents of the /etc directory and the password file by typing:

```
ls –al /etc ; cat /etc/passwd
```

B. Print the string "Hello World" and your name to the screen by typing:

```
echo "Hello World"; echo "Put your name here"
```

Line Continuation

4.10 Line Continuation Sometimes the things you type on the command line can get long. If you were typing a paper and the line was getting too long you could hit the <enter> key and start a new line. But if you are typing a command and hit the <enter> key then the UNIX shell will take this as the signal that you're done typing and it should start interpreting the command(s) you typed.

However, if you type the slash character "\" (called a backwhack in UNIX speak) and then hit the <enter> key, you can continue typing on the next line. The UNIX shell will display a different prompt on continued lines to provide a visual reminder about what's going on. And the UNIX shell will not begin actual command processing until you hit the <enter> key without a "\" for continuation.

You don't use this a lot for simple UNIX commands, but it used frequently when building more complex pipeline command sequences.

A. Type:

```
echo "Hello \
 World" <enter>
```

Is the `echo` command executed before you enter the second line?
Does the shell display anything different after you type the "\" to let you know that you are continuing the command line? What character is it
What is displayed when the command is executed?

B. Practice using "\" to continue a command across lines by typing:

```
echo "my name is \
put your name here" <enter>
```

Redirection

4.11 stdin and stdout When a command is executed it produces some output which is typically displayed on the screen. If the command needs some input, it typically gets it from the user, who types it in at the keyboard. In UNIX we give the standard input and standard output the names ***stdin*** and ***stdout*** respectively. There's another device named ***stderr***, that refers to error messages. These are typically sent to the same place as normal output, that is the

screen. The unique thing that we can do with UNIX commands is tell them to send their output somewhere other than the screen, and get their input somewhere other than the keyboard.

Device	Name	File Descriptor	Normally associated with
stdin	standard input	0	Keyboard
stdout	standard output	1	console (monitor)
stderr	standard error	2	console (monitor)

4.12 Redirection. UNIX allows you to string together separate commands by taking the output from one command (stdout) and instead of displaying it on the screen putting it in a file or handing it another command as input (stdin). This allows you to build fairly sophisticated commands without having to write and compile code.

This set of exercises look at redirection, which is sending output from a command to a file. Redirection gets its name because technically speaking it is redirecting stdout away from monitor to a file. It is accomplished by typing a ">" after a command, followed by the name of a file. Generically it's: `command > outputFileName`

UNIX Redirects	
>	redirects stdout to a file creates if file does not exist (if write permission to directory) **overwrites** if the file exists
>>	redirects stdout to a file creates if file does not exist (if write permission to directory) **appends** if the file exists
>!	Shell tracks variable called noclobber If set, normal redirect will not overwrite an existing file, gives error message instead >! Can be used to force overwrite of existing file
2>	redirects stderr
>& >>&	Redirects stdout and stderr to file (>& overwrite >>& append)
>&! >>&!	Redirects stdout and stderr to file even if noclobber set and file exists (>&! Overwrite >>&! Append)
<< word	Reads from stdin up until word is found
<	Reads input from file Not so useful by itself, but has real power when combined with other shell commands Way to say read line by line from a file until a certain character is read Will cover in shell programming section

A. Display the contents of the system password file by typing: `cat /etc/passwd`

In this case, what is stdout? That is, where if the output from the `cat` command displayed?

B. Now, use the redirect to send the output from the `cat` command to a file. Make sure that you are in your home directory (or some folder where you have write permission). Type: `cat /etc/passwd > testFile`

Is the content of the password file displayed on the screen? Use the `cat` or `more` command to check the contents of the file passwd.txt. What does this file contain?

C. This portion of the exercise demonstrates how the shell can protect you from overwriting existing files. However, this feature is not supported by all shells. It is supported by csh and tcsh. It is **not** supported by sh or bash. Before starting this exercise you can ensure that you are in the tcsh shell by typing:

```
ps
```

If you see some shell other shell than tcsh listed, such as bash or sh, then switch to tcsh by typing:

```
/bin/tcsh
```

Execute the `date` command, and redirect it's output to `testFile` by typing the following. (Make sure that you redirect the output to the same file as in the previous step.)

```
date > testFile
```

Inspect the contents of the file. Was the output of the date command appended to the end of the file or did it overwrite the existing content? The contents of the file should have been overwritten, so the only thing in the file should be the current date.

If you get the error message: "`testFile: file exists`" while running this command, it means the shell is protecting you from overwriting files because a shell variable named `noclobber` has been set. The `noclobber` variable instructs the csh/tcsh shell to prevent you from overwriting files that already exist. (Shell variables are explained in greater detail in a later section.) It's ok if you get the error message, because it's showing what we want to demonstrate. If you didn't get the error message, then you still need to set the `noclobber` variable. This is done by typing:

```
set noclobber
```

You can test this again by trying to redirect the output from another command to the file `testFile`. Type the following:

```
ls -al > testFile
```

Do you get an error message, or is the content of the file `testFile` overwritten? You should see an error message similar to "`testFile: file exists`" which is the shell's way of telling you that it will not overwrite an existing file.

To remove the protection against accidental overwriting you must remove the noclobber variable. This is done by typing:

```
unset noclobber
```

Test this by trying to redirect the output from another command to the file `testFile`. Type the following:

```
ps -alx > testFile
```

Do you get an error message, or is the content of the file `testFile` overwritten?

D. Another way to tell the shell that you want to overwrite a file is to use `>!` instead of `>`. This temporarily overrides the noclobber protection, but only for this command. When the shell has finished this command the noclobber variable will still be set. To demonstrate this make sure the tcsh shell is set to prevent accidental overwriting by typing: `set noclobber` Execute the `date` command, and redirect it's output to a file named `testFile2` by typing:

```
date > testFile2
```

Do you see the output from the `date` command on the screen? Inspect the contents of the file `testFile2`. Where did the output from the `date` command go?

Next, run the `ls -al` command and send the output to the file `testFile2` by typing:

```
ls -al > testFile2
```

Do you get an error message? If so what is it and what does it mean? Now type:

```
ls -al >! testFile2
```

Inspect the contents of the file. Was the output of the `ls` command written to the file or did it overwrite the existing content? Did the shell prevent the overwriting?

E. This exercise demonstrates redirecting error messages, or stderr, from the screen to a file. WARNING – this does not work in the tcsh shell. If you are using tcsh, switch to the Bourne shell by typing `/bin/bash`

To demonstrate displaying an error message ask the `ls` command to display a file that does not exist. For example type: `ls noSuchFile`

What error message does `ls` display? And where is it displayed, or in this case what is stderr?

Now type: `ls noSuchFile 2> err.dat`

Is the error message displayed on the screen? Check the contents of the file `err.dat`. What does it contain?

F. Both stderr and stdout can be redirected from the same command. (Once again this is only supported by the Bourne shell.) Type the following:

```
ls noSuchFile /etc > etcFiles 2> etcErr
```

Inspect the contents of the file `etcFiles` What does it contain? Inspect the contents of the file `etcErr` What does it contain? Which file contains the error message(s)?

G. The double redirect >> is used to append output to a file. Type:

```
date > files.dat
```

Inspect the contents of the file. Now type:

```
ls >> files.dat
```

Use the `more` command to inspect the contents of the file. Check to see if the output from the `date` command is still at the beginning of the file. Was the output of the `ls` command appended to the end of the file? Or did it overwrite the existing content?

H. Standard input, which usually comes from the keyboard, can also be redirected using <. For example `more < /etc/passwd` tells the shell to get the input to the more command from the file /etc/passwd. This is the same as typing `more /etc/passwd` which is more straightforward and usually makes more sense, but there may be occasions when `command < file` is used.

Demonstrate the redirecting of input by typing

```
more < /etc/hosts
```

I. Apply what you have learned about redirection. For each of the following write what you type on the command line to accomplish the specified task:

i. Write the command to create a file named `aFiles` that contains a list of all the files in the directory `/usr/lib` that start with the letter `a`

```
ls /usr/lib/a* > aFiles
```

ii. Write the command to create a file named `libFiles` that contains a list of all the files in the directory `/usr/lib` that start with the letters `lib`

iii. Write the command to create a file named `soFiles` that contains a list of all the files in the directory `/usr/lib` that end with the letters `so`

_____

iv. Write the command to create a file named `currentUsers` that contains the current date followed by a list of all the users currently logged into the system. (Hint – use the `who` command.)

_____

J. What would be the result of the following?

```
date > files.dat; ls -al >> files.dat
```

That is, when all the commands have been executed what do you expect to see in the file `files.dat`?

i. Only the output from the `date` command
ii. Only the output from the `ls -al` command
iii. The output from the `date` command followed by the output from the `ls-al` command
iv. The output from the `ls -al` command followed by the output from the `date` command
v. Nothing

Processing Data in Text Files with Cut, Paste and Join

After a certain point, programmers see that in general what they do with a program is manipulate data. Regardless of the specific language, they need to understand how the data is structured, and then use the specific programming commands to read, process and or write the data.

UNIX has several commands such as `cut`, `paste`, `join` and `awk` that let the user manipulate the data in a file from the command line. The cut, paste and join commands work with plain text files that have the data organized into tables. Each line of the data file represents a table row, or database record. Each line is subdivided into columns or fields by placing a delimiter character between the text. (A delimiter is a character that is used to separate things.) For example, the following lines each have the fields "name" and "e-mail", and use the character "`:`" as the delimiter.:

```
Joe User : joe@aol.com
Ben Dover : bendover@aol.com
Mike Keister : mikekeister@aol.com
```

And this example shows text that represents 4 records. Each line is broken into 4 data fields, with a <tab> character acting as the delimiter between each field. The first field is the record number, the second field holds the first name or given name, the third field holds the last name or surname, and the last field holds the title.

```
1  <tab> Ben    <tab> Steinway      <tab> Mr.
2  <tab> Sam    <tab> Fader         <tab> Mr.
3  <tab> Sue    <tab> McCauley      <tab> Dr.
4  <tab> Everett <tab> Simpson      <tab> Mr.
```

The `cut` command allows you to grab specific fields from a file, while the `paste` and `join` commands combine fields from different files into a single file. When you combine these commands with redirection, you can do things like create reports using data from different files, or rearrange the fields of data in a single file. The huge difference in UNIX is that this can all be done from the command line; it doesn't take a special database program or even a higher level programming language like would be required under Windows.

`Cut`	Prints selected fields from a file
`Paste`	Combines lines from two files
`Awk`	Performs an editing action on all matching records/fields in a file
`Join`	Combines lines from two *sorted* files that contain the same value in a specified field

Before you start this exercise you will need all of the files shown below. You can copy them from the directory `/home/cutPractice`. Or you can create them yourself by either using `vi`, or creating the files on the PC and using FTP to move them to your UNIX account. To copy the files to your home directory use the following commands:

```
cd  ~
cp –r /home/cutPractice .
```

(The last character in the `cp` command is a "." or dot. Remember that the dot stands for the current directory; so this command will copy the `cutPractice` folder from `/home` to whatever directory you are currently in.)

names.txt

```
1  <tab> Ben     <tab> Steinway      <tab> Mr.
2  <tab> Sam     <tab> Fader         <tab> Mr.
3  <tab> Sue     <tab> McCauley      <tab> Dr.
4  <tab> Everett <tab> Simpson       <tab> Mr.
5  <tab> Renee   <tab> Ports         <tab> Mrs.
6  <tab> Franklin<tab> Smith         <tab> Mr.
7  <tab> Reid    <tab> Hall          <tab> Mr.
8  <tab> Pat     <tab> Benson        <tab> Mrs.
9  <tab> Lucy    <tab> Nascent       <tab> Dr.
10 <tab> George  <tab> Carter        <tab> Mr.
```

address.txt

```
1  <tab> 122 Wicker      <tab> Pasco   <tab> WA   <tab> 99345
2  <tab> 67 Blue Ford St <tab> Haley   <tab> ND   <tab> 77547
3  <tab> 888 N 45th      <tab> Seattle <tab> WA   <tab> 87888
4  <tab> 1856 Cedar Rd   <tab> Paris   <tab> TX   <tab> 43444
5  <tab> 997 N Verde     <tab> Trevor  <tab> OR   <tab> 97997
6  <tab> 56  Crab Creek  <tab> Snyder  <tab> AK   <tab> 68521
7  <tab> 749 Naeflus Ct  <tab> Perry   <tab> FL   <tab> 76765
8  <tab> 933 Hendrix Av  <tab> Havre   <tab> MI   <tab> 25422
9  <tab> 76 98th S       <tab> Ashland <tab> FL   <tab> 23242
10 <tab> 7444 Hewe Ct    <tab> Hattrick<tab> MI   <tab> 20092
```

phone.txt

```
1  <tab> (355)  <tab> 344-0909
2  <tab> (324)  <tab> 399-9654
3  <tab> (222)  <tab> 665-9987
4  <tab> (212)  <tab> 878-2176
5  <tab> (891)  <tab> 231-2187
6  <tab> (737)  <tab> 865-6664
7  <tab> (389)  <tab> 893-5543
8  <tab> (405)  <tab> 267-9876
9  <tab> (834)  <tab> 854-4432
10 <tab> (388)  <tab> 329-4422
```

points.txt

8. 765
2. 763
1. 755
3. 740
9. 732
5. 730
4. 717
6. 716
7. 702
10. 650

guitars.txt

Gibson
Fender
Taylor
Gibson
Rickenbacker
Fender
Ovation
Gibson
National
Fender
Taylor
PRS
Epiphone
Gretsch
Fender

4.13 The `cut` command The `cut` command finds a set of specified fields in a file.

A. The `f` option tells the cut command which field(s). By default, the `cut` command uses the tab character as the delimiter between fields. Use `cut` to extract the third field from the file names.txt:

```
cut -f3 names.txt
```

B. Multiple fields can be specified. If the fields in the `-f` option are separated by commas, cut will extract the individual fields. Use cut to extract the second and third fields from the name file

```
cut -f2,3 names.txt
```

C. Cut with field range. If the fields in the `-f` option are separated by a "-", then cut will extract all of the fields in the range between the two fields numbers. Use cut to extract fields 2 through 4 from the name file

```
cut -f2-4 names.txt
```

D. Cut by columns. You can also the `-c` option tell the `cut` command to just cut certain characters from a file by specifying the number of characters from the beginning of the line. This is also called the column count. If you use `-c`, then don't use `-f`. There's also no need to specify a delimiter. Cut characters 1 through 5 from phone.txt

```
cut -c1-5  phone.txt
```

E. Cut with specified delimiter. This exercise demonstrates how to tell the `cut` command to use a different delimiter. Try to cut the first field, the username, from the /etc/passwd file using the default delimiter.

```
cut -f1 /etc/passwd
```

Do you get what you expected? The `cut` command returns the entire line instead of the first field because there were no tabs between the fields. The /etc/passwd file uses the colon "`:`" character as the delimiter between fields. Now try the same command but use the `-d` option

```
cut –f1 –d:  /etc/passwd
```

4.14 The `paste` command The `paste` command is used to paste columns or fields from different files back together. In its most basic form `paste` is used this way: `paste file1 file2`

The `paste` command is a little brain dead, in that it just pastes corresponding lines together. That is, line1 from file 1 will be pasted together with line 1 from file 2. It's up to you to ensure that the lines actually have information that goes together.

Also, notice that the output is sent to stdout. So if you want to capture it in a file you should use a redirect. For example: `paste file1 file2 > file3`

A. Paste columns back together. The `paste` command is the not quite opposite of `cut`. It's used to join columns from two different files into a single file. When it combines the files it just goes line by line starting at the first lines.

```
paste names.txt  phone.txt
```

Notice what character is placed between the entries from the different files, that is, which character is used as a delimiter.

B. Use the `paste` command to merge the data in the files names.txt and phone.txt by typing:

```
paste names.txt phone.txt > nameAndPhone.txt
```

Look at the contents of nameAndPhone.txt. Notice that each line of the two files were pasted together. Also notice that each new line contains all of the fields from the original file. Can you think of a way to do this so that the field containing the numbers 1-10 of phone.txt are not added? (Hint, do some cutting before pasting the two files.)

C. You can tell `paste` to use a different delimiter by using the `–d` option. For example:

```
paste –d:  names.txt  phone.txt
```

Look at the results. Notice that `paste` only puts the delimiter between the fields from different files. It doesn't replace the existing delimiter inside the lines from any of the files it's pasting together. If you wanted to use the "`:`" as a delimiter between every field you would have to split every field into a separate file and then paste them all back together. (Or you can learn how to use the `sed` command later in this section.)

4.15 Practice with `cut` and `paste` Assume that you want to use the data from names.txt and phone.txt files to create a file with the names (first and last) along with the phone numbers.

i. Use the cut command to get the first and last names from the names.txt file

```
cut –f2,3  names.txt > name2
```

ii. Use the cut command to get the phone numbers from phones.txt

```
cut –f2,3  phone.txt > phone2
```

iii. Use the paste command to create the file with the names and phone numbers.

```
paste name2  phone2 > namePhone.txt
```

Take a look at the contents of this file to make that it has the names and phone numbers.

iv. Clean up the temporary files

```
rm name2 phone2
```

4.16 Practice with `cut` and `paste` Assume that you want to use the data from names.txt and address.txt files to create a file with the names (first and last) along with the address.

i. Use the cut command to get the first and last names from the names.txt file

```
cut –f2,3  names.txt > name2
```

ii. Use the cut command to get the address from adress.txt

```
cut –f2,3  address.txt > address2
```

iii. Use the paste command to create the file with the names and addresses.

```
paste name2  phone2 > nameAddress.txt
```

Take a look at the contents of this file to make that it has the names and addresses.

iv. Clean up the temporary files

```
rm name2 address2
```

4.17 More practice with `cut` and `paste` Assume that you want to use the data from names.txt but you want it in a different order. You would like to have the title first, followed by the first and last names.

i. Use the cut command to get the first and last names from the names.txt file

```
cut –f2,3  names.txt > name2
```

ii. Use the cut command to get the title from names.txt

```
cut –f4  names.txt > name3
```

iii. Use the paste command to create the file with the titles and names.

```
paste name3  name2 > titleNames.txt
```

Take a look at the contents of this file to make that it has the titles and names.

iv. Clean up the temporary files

```
rm name2 name3
```

4.18 More practice with `cut` and `paste` Assume that you want to combine some of the data from names.txt, phone.txt and address.txt into a single file. You would like to have the title first, followed by the first and last names; followed by the phone number, followed by the address. Use what you have learned about `cut` and `paste` to accomplish this.

4.19 The `join` command. The `join` command is like a smart paste. It combines lines from two files, but instead of blindly combining the lines like `paste`, you can specify a field that will be used as a key which join will use to determine which lines to combine. That is, any lines with the same value in the specified column will be combined. It requires that the files be set up with some type of delimiter to separate the columns, and will only join columns that have one field in common (like a database key field).

For example, given the following files:

names.txt

```
1   <tab>  Ben     <tab>  Steinway
2   <tab>  Sam     <tab>  Fader
4   <tab>  Everett <tab>  Simpson
5   <tab>  Renee   <tab>  Ports
6   <tab>  Franklin<tab>  Smith
```

address.txt

```
1   <tab>  122 Wicker      <tab>  Pasco
2   <tab>  67 Blue Ford St <tab>  Haley
3   <tab>  888 N 45th      <tab>  Seattle
4   <tab>  1856 Cedar Rd   <tab>  Paris
5   <tab>  997 N Verde     <tab>  Trevor
```

The command `join names.txt address.txt` would combine the two files for any lines that have the same number in the first field. This would result in:

```
1   <tab>  Ben     <tab>  Steinway       <tab>  122 Wicker      <tab>  Pasco
2   <tab>  Sam     <tab>  Fader          <tab>  67 Blue Ford St <tab>  Haley
4   <tab>  Everett <tab>  Simpson       <tab>  1856 Cedar Rd   <tab>  Paris
5   <tab>  Renee   <tab>  Ports          <tab>  997 N Verde     <tab>  Trevor
```

Notice that line 6 from names.txt is not included because there's no corresponding line 6 in address.txt. The same things happens for line 3 in address.txt; it's omitted because there's no line 3 in names.txt.

By default `join` looks at the first column for matches (the primary key), but you can specify a different field if you want. For example `join -1 3 -2 1 file1.txt file2.txt` The `-1 3` says to use the 3rd column or field of file1.txt. The `-2 1` says to use the 1st column or field of file2.txt.

If you want to include all the rows from a file, even if there are no matches in the second file you can use the `-a` option. For example `join -a1 file1.txt file2.txt` says to include all the lines from the first file or file1.txt in this example.

A. **Practice with the `join` command**. Before you start this exercise you will need the three files you used in the `cut` and `paste` exercises. Copy them from the directory /home/cutPractice to a new subdirectory using

```
cd ~
cp –r /home/cutPractice joinPractice
cd joinPractice
```

Now assume that you want to create a single file that contains names and addresses. Use what you have learned about `join` to accomplish this.

B. **Practice with the `join` command** Assume that you want to create a single file that contains names and phone numbers. Use what you have learned about `join` to accomplish this.

Basic Piping Using |

4.20 Pipeline basics. In the previous set of exercises you redirected stdout from the monitor to a file. In this set of exercises you will learn how to pipe stdout from one command to stdin on another command. It's called piping because it's just like sticking a piece of irrigation pipe to allow characters output from the first command to flow into the second command as input. Piping is done by typing the "`|`" between two commands. The `|` character, which is on the same key as the `\`, is just above the <Enter> key on most keyboards.

Remember that generally speaking *redirection* looks like `command > filename` With piping, the general syntax is `command1 | command2` Because instead of handing stdout to a file (or sending it to the monitor) it's being passed to another command. The output from command1 looks like a stream of text to command2.

A. Type: `ls -al /etc`

What is stdout for the `ls` command?
Where does it go?

B. Type: `ls -al /etc | more`

What is stdout for the `ls` command?
Where does it go?
What is stdin for the `more` command?
Where does it come from?

C. Type: `cat /etc/passwd`

What is stdout for the `cat` command?
Where does it go?

D. Type: `cat /etc/passwd | more`

What is stdout for the `cat` command?
Where does it go?
What is stdin for the `more` command?
Where does it come from?

E. As it's name suggests, the `sort` command sorts output. Type:

```
cat /etc/passwd | sort | more
```

What is stdout for the `cat` command?
Where does it go?
What is stdin for the `sort` command?
Where does it come from?
What is stdout for the `sort` command?
Where does it go?
What is stdin for the `more` command?
Where does it come from?

F. Type:

```
cat /etc/passwd > junk1; sort junk1 > junk2; more junk2
```

Where does stdout for the `cat` command go?
What file does the `sort` command read?
Where does stdout for the `sort` command go?
What file does the `more` command read?
What intermediate files are created during this process?
What happens to these intermediate files when the command is finished?
How would you remove these intermediate files?
Is it easier to use pipes (as in the previous exercise), or use this method?

G. Type:

```
ps –aux
```

What is stdout for the `ps` command?
Where does it go?

H. Type:

```
ps –aux | sort
```

What is stdout for the `ps` command?
Where does it go?
What is stdin for the `sort` command?
Where does it come from?
What is stdout for the `sort` command?
Where does it go?

I. Type:

```
ps –aux | sort | more
```

What is stdout for the `ps` command?
Where does it go?
What is stdin for the `sort` command?
Where does it come from?
What is stdout for the `sort` command?
Where does it go?
What is stdin for the `more` command?
Where does it come from?
Where does it go?

4.21 **The tee**. There are times that you want to pass stdout to a file but you also want to see the output passed on to another command in the pipeline. In other words, you want to output to go in two different directions at the same time. If you were connecting water pipes you would use a connector called a tee-connector to split the water and send it in two directions. In UNIX this same thing is done by typing `|tee filename |` between two commands. A generic example would be: `command1 | tee filename | command2`

The result of the generic example would be that the output from command1 would be sent to filename, while a duplicate copy of the output from command1 would be sent to command2.

A. Type: `ls –al /etc | tee ls.out | more`

Does the output from the `ls` command appear on the screen?
Check the contents of the file ls.out. What does it contain?

B. Type: `ps –aux | tee ps.out | more`

Does the output from the `ps` command appear on the screen?
Check the contents of the file ps.out. What does it contain?

4.22 **Conditional Piping**. Sometimes you only want the pipeline processing to continue if things are going ok, or maybe you only want to do more processing if there was an error. (For those of you who know how to program this is usually handled by an `if` statement.) In the UNIX shell this is handled simply and elegantly by using either double pipes `||` for only piping if the previous command was not completed successfully, or double ampersands `&&` to only continue if the previous command was successful (no errors).

A. Type: `ls –al ~/noSuchDir`

Is what you see on the screen an error message or output from the directory?

B. Type: `ls –al ~/noSuchDir || echo "command failed"`

Is what you see on the screen an error message, output from the directory or the output from the `echo` command?

C. Type: `ls –al ~/noSuchDir && echo "command ran successfully"`

Is what you see on the screen an error message, output from the directory or the output from the `echo` command?

D. Type: `ls –al ~/noSuchDir || mkdir ~/noSuchDir`

Is the directory is created? Why or why not

E. Type: `ls –al ~/noSuchDir && mkdir ~/noSuchDir`

Is the directory is created? Why or why not

4.23 Grouping Multiple Commands Using Parentheses All of the above methods for redirection and piping pass the output from a single command through the pipe. But there occasions when you want to group output from multiple commands before sending them to a file or through pipe. Say for example that you want to print today's date and the contents of the /etc directory to a file by typing:

```
date; ls –al /etc > etc.out
```

However, when this is executed the output from the `date` command is displayed on the screen and the only thing in the file is the output from the `ls` command. If you look at this from a certain angle it's kind of like the math issue where you have an equation like 6*4+2 Should you do the 6*4 first, so it would be (6*4) + 2 which equals 26? Or should you do the 4+2 first, so it would be 6 * (4+2) which equals 36?

The way to fix the UNIX command line is the same way you fix the math problem. By using parentheses you can tell the UNIX shell that you want to group the output from the `date` and `ls` commands before doing the redirection.

A. Assume that you want to add the date and some text to a file. You type:

```
date; echo "Hello World" > hello.out
```

Where does the output of the `date` command go?
Where does the output of the `echo` command go?
Check the contents of the file hello.out. What does it contain?

Now type:

```
(date; echo "Hello World") > hello.out
```

Where does the output of the `date` command go?
Where does the output of the `echo` command go?
Check the contents of the file hello.out. What does it contain?

B. Type: `echo "Tony"; echo "Mike"; echo "Beth"; echo "Abe" | sort`

What order are the names listed in the output?
Where does the output of each `echo` command go?
What is the `sort` command getting for input and does it do anything?

Now type:

```
(echo "Tony";echo "Mike";echo "Beth"; echo "Abe") | sort
```

What order are the names listed in the output?
Where does the output of each `echo` command go?
What is the `sort` command getting for input and does it do anything?

Data Processing with Pipes & Pipelines.

Pipes provide the ability to chain commands together. Pipes get their name from the way they function, taking the output (stdout) from one command and passing it on to the next (as stdin). This may not seem like much, but it allows you to do things from the command line that usually can only be accomplished by writing a program in a language like VB, C, JAVA, C++ etc.

In this section of exercises you will learn about commands that provide extra power to your data processing abilities. With `cut` and `paste` you were able to rearrange data in a file, or create new files that contained combinations of fields from different files. The problem with `cut` and `paste` is that they don't provide any way to select specific records, that is you get the data from all lines or records in a file. And you are also limited to copying the data, there's no way to change it with cut and paste.

In this section you will learn about grep, which is a command that allow you select specific records, so you can copy just certain lines of text. You will also learn about sed and tr which allow you to change the data. These commands were all developed to take advantage of the ability to pipe information between commands.

`grep`	Find lines that match a pattern
`sed`	Edit lines of text by adding, replacing or deleting
`tr`	Translates characters from one set to another (usually uppercase to lowercase, etc.)
`sort`	Sorts the contents of a file
`uniq`	Removes duplicate adjacent lines from a file
`comm.`	Show the common and different lines between two files
`join`	Combines lines from two *sorted* files that contain the same value in a specified field

4.24 The `sort` command. The `sort` command is used to sort data. As an example, take a look at the file names.txt. If you `cut` out the first names and displayed them they would be displayed in the order they appear in the file. But if you pipe the output from cut through the sort command the names will appear in alphabetic order.

```
cut –f2  names.txt | sort
```

The `sort` command actually works by looking at the ASCII numbers for each character, so sometimes the order may seem a little odd, particularly if there are non-alphanumeric characters involved.

Dec	Char
0	NUL
1	SOH
2	STX
3	ETX
4	EOT
5	ENQ
6	ACK
7	BEL
8	BS
9	TAB
10	LF
11	VT
12	FF
13	CR
14	SO
15	SI
16	DLE
17	DC1
18	DC2
19	DC3
20	DC4
21	NAK
22	SYN
23	ETB
24	CAN
25	EM

Dec	Char
26	SUB
27	ESC
28	FS
29	GS
30	RS
31	US
32	
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3

Dec	Char
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M

Dec	Char
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g

Dec	Char
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~
127	DEL

A. **Sort in a reverse order**. To sort in reverse or descending order use the `-r` option.

```
cut -f2  names.txt | sort -r
```

B. **Sort on a different field**. You can also tell `sort` to use any field to determine the sort order. To do this you must supply two field numbers which will be the start and stop field. E.g. `+3 -4` would mean start with field 3, and stop with 4, or essentially 4. You can also supply multiple pairs `+3 -4 +2 -3` would mean use field 3 for main sort, and field 2 for tie breaker. Try sorting the file `names.txt`, but using field 3 to determine the order. (Note that there must be a space between the `+3` and the `-4`.)

```
cat  names.txt | sort +3 -4
```

C. Practice with the `sort` command. Assume that you want to get the names from the names.txt file. You would like to have the last and first names in that order, sorted by last name. Use what you have learned about `cut, paste` and `sort` to accomplish this.

D. So far we've looked at piping data to the sort command, but it can also be used to simply sort the data in a file. The trick here is that typically you can't read and write to the same file. However this can be accomplished by using the `-o` option. (The `-o` stands for overwrite). Before starting the sort, take a look at the contents of the file points.txt

```
cat points.txt
```

Now sort the file using:

```
sort points.txt -o points.txt
```

Examine the file again and see if it has been sorted as expected.

4.25 The `uniq` command. The `uniq` command is used to find and optionally remove duplicate lines. The `uniq` command has one limitation, it can only find duplicate lines if they are adjacent in the file. Before starting these exercises, look at the file guitars.txt and familiarize yourself with the content.

A. The first use for the `uniq` command is to find out if a file has any duplicate lines. This is done by using the `-d` option. Find any lines with duplicates in guitars.txt by using the command:

```
uniq -d guitars.txt
```

Did you get the result that you expected? Remember that uniq only works when duplicate lines are next to each other in the file. To do this with guitars.txt try:

```
sort guitars.txt | uniq -d
```

B. The uniq command will also count how many times a line occurs if it's given the `-c` option. Find how many times each line in guitars.txt occurs by using:

```
sort guitars.txt | uniq -c
```

C. If you want to only have one occurrence of a line, the uniq command can be used to eliminate all duplicates. To do this, the uniq command is used with no options.

```
sort guitars.txt | uniq
```

4.26 grep The `grep` command is used for finding strings of text within files. (As opposed to using wildcards to find files with certain strings in their names.) It gets it's name from the way it looked in the ancient UNIX `ed` editor, which was `g/re/p`. The `g` stands for global, the `re` for regular expression, and the `p` stands for print.

So on a practical level `grep` is the command you use to find lines in a file that contain a specified string, but on another level, `grep` is good test of your love of UNIX. Once you start to understand and love `grep` and regular expressions you know that you're on your way to earning your "Jr. UNIX Geek" badge.

`grep` is so popular and powerful that, kind of like the Indiana Jones or Rocky Movies, it's spawned it's own set of sequels. In this case there are also commands named `egrep` and `fgrep` on almost every UNIX distribution.

In its simplest form you type `grep`, followed by the string you want to find, followed by the file(s) you want to search. For example: `grep yellow *.htm` would search all the files ending with `.htm` for the string `yellow`. Each time `grep` found a match it would print the entire line containing the string to stdout.

A. Practice with the `grep` command. Before you start this exercise you will need the three files in the directory /home/grepsedPractice. Copy them from the directory /home/grepsedPractice to a new subdirectory under your home directory.

Assume that you want to find all of the lines in the three lyric files that contain the word "`one`". Use what you have learned about `grep` to accomplish this.

B. Use the `grep` command to find the line in /etc/passwd that contains your account information by searching for your username.

4.27 sed. The `sed` command stands for **s**tream **ed**itor, and it's used for making edits to a stream of text passing through a set of pipeline commands. So in addition to cutting and pasting parts of different files together you can also make changes to them on the fly.

To used the `sed` command you type `sed` followed by a single letter to indicate what type of edit you want to perform: `s` for substitution or `d` for deletion.

If you want to do a substitution, you next type the string you want to change followed by the string you want to change it to. The s command and the two strings are separated by a delimiter character, which is usually the slash "`/`". In the general form this would be:

```
sed  s/oldString/newString/
```

For example, to change occurrences of the string "tony" to "Tony" the command would be:

```
sed  s/tony/Tony/
```

If either the old string or the new string has a slash in them you need to use a different character as the delimiter. So even though "`/`" is the typical delimiter, you can use any character you wish. The `sed` command assumes that the character immediately following the `s` command is the delimiter. So if you wanted to use "`@`" instead the general form becomes;

```
sed  s@oldString@newString@
```

If there were more than one occurrence of the string in a line the form of `sed` used so far would only change the first one. To change them all you would add another delimiter after the newString, followed by a `g` which stands for global. The general form would be:

```
sed  s/oldString/newString/g
```

The sed command needs a stream of data to process, so it's typically used as part of a pipeline. For example, the cat command can be used to send every line in a file through a pipe to sed. In general this would look like:

```
cat filename | sed  s/oldString/newString/
```

The `sed` command can also be used to delete text. To this omit the value for `newString`. You must still include the delimiter character, just don't put anything inside. The general form for a deletion is:

```
sed  s/oldString//
```

A. Practice with the `sed` command. Before you start this exercise you will need the three files in the directory /home/grepsedPractice. Copy them from the directory /home/ grepsedPractice to a new subdirectory under your home directory.

Assume that you want to change all occurrences of the word "one" in the file threeDog.lyrics to "three" and write the results to the file threeDogNew.lyrics. Use what you have learned about `sed` to accomplish this.

B. Assume that you want to change all occurrences of the word "one" in the file u2.lyrics to "geeks on scooters" and write the results to the file u2New.lyrics. Use what you have learned about `sed` to accomplish this.

C. Practice with the `sed` command. Assume that you want to change all occurrences of the word "loneliness" in the file police.lyrics to "peanutbutter" and write the results to the file policeNew.lyrics. Use what you have learned about `sed` to accomplish this.

D. Practice with the `sed` command. Assume that you want to change all occurrences of the word "bottle" in the file police.lyrics to "twitter tweet" and write the results to the file policeNew.lyrics. Use what you have learned about `sed` to accomplish this.

4.28 Regular expressions. The `sed` and `grep` commands take strings as arguments. In addition to typing simple strings, there are also wildcards that can be used. These wildcards and the expressions that can be built with them are referred to as regular expression patterns. You may also see regular expression abbreviated as regexp. When most people learn about regular expressions they go through a brief period of confusion. Regular expression are confusing because some of the symbols, such as *, look the same as the wildcards used for expanding filenames, but they have a different meaning in regular expressions. The good news is that a lot of different programs are using regular expressions. There are programs on Windows systems as well as UNIX systems, so once you learn about regular expressions you may be able to use them quite often.

4.29 Simple Expressions The simplest regular expressions are character strings. For example, is you wanted to find the word “tony” the regular expression would be `tony`.

There are a few problems just searching for strings. For example, if you want to find the word “red”, you will also find words with red in them, such as “Fred”, “predator”, etc. Or what if you wanted to find all the different forms of a word such as run, running, runner, etc.?

Regular expressions allow you to build patterns to that can narrow down your search to find just what you want. Regular expression patterns have three parts: Character sets, Anchors, and Modifiers.

4.30 Character Sets specify one or more characters that may or may not appear in the string. They may be as simple as the characters you want to match, but you can also use things like “.” which matches any character or `[a-z]` which would match any lower case letter. The following table shows the main ways to specify a character set:

Character Set	**Matches**
`The`	Any occurrence of "the". Matches the, but also matches **the**m, **the**re, ba**the**, etc.
`.`	Any single occurrence of any character (except newline)
`[a]`	A single occurrence of the character “a”
`[abc]`	A single occurrence either the character “a” the character “b” or the character “c”
`[a-z]`	A single occurrence of any lower case letter. (in csh/tcsh `setenv LC_ALL 'C'` or in sh/bash `export LC_ALL=C` to avoid matching uppercase alpha too.
`[A-Z]`	A single occurrence of any upper case letter
`[a-zA-Z]`	A single occurrence of any letter, either upper or lower case
`[0]`	The character "0"
`[012]`	A single occurrence either the character “0” the character “1” or the character “2”
`[0-9]`	Any numeral

`[0-9A-Za-z]`	Any numeral or any upper or lower case letter
`[^a]`	Any character that is **not** the character "a" (The ^ means not)
`[^a-z]`	Any character that is **not** a lower case letter
`[^1]`	Any character that is **not** the numeral character "1"
`[^0-9]`	Any character that is **not** a number
`[-a-z]`	Any lower case letter or a "-"
`[0-9-]`	Any number or a "-"
`[^-a-z]`	Any character except a lower case letter or a "-"
`[]0-9]`	Any number or a "]"
`]`	The character "]"
`[0-9]]`	Any number followed by the character "]"
`[0-9\]]`	Any number or the character "]"
`[0\-9]`	The character "0" or the character "-" or the character "9"
`[\^1]`	The character "^" or the character "1"

Examples

`'[0-9][0-9]'`	Any number 00-99
`'Room [1-2][0-9][0-9]'`	"`Room`" followed by 100 through 299
`'Title: ....[0-9]'`	"`Title:`" followed by any 4 characters, followed by 1-9

A. Practice with simple regular expressions. Remember that the regular expression `'Computer ID:'` will match any line with the characters '`Computer ID:`'. For each of the following, write the correct regular expression:

The string "CBC" _____

The string "cbc" _____

The string "https:" _____

B. Practice with simple regular expressions. and the '`.`' for matching any character Remember that the regular expression `'DataFile.'` will match any line with the characters '`DataFile`' followed by any single characters. For each of the following, write the correct regular expression:

The string "CBC" followed by any character _____

The string "cbc" preceded by any character _____

The string "Class" followed by any 2 characters _____

C. Practice with simple regular expressions and the `[characterlist]` for matching a single character from a specified list. Remember that the regular expression `'[tT]ony.'` will match any line with either the character '`t`' or '`T`' followed by the characters `'ony'`. For each of the following, write the correct regular expression:

The string "case" or the string "Case" _____

The string "room" or the string "Room" _____

The string "CLASS:" followed by any one of the following characters "aAeEiI" _____

The string "Key#" followed by any one of the following characters "135" _____

D. Practice with simple regular expressions and the `[start-end]` for matching a single character from a specified range of characters. Remember that the regular expression `'Docket[0-9]'` will match any line with the characters '`Docket`' followed by any single numeral, and `'Window-[A-Z]'` will match any line with the characters '`Window-`' followed by any single uppercase alpha character. For each of the following, write the correct regular expression:

The string "Key " followed by any single digit _____

The string "Locker:" followed by any uppercase or lowercase alphabetic character _____

The string "ID-" followed by any character that is **not** an uppercase alphabetic character _____

A number that is a valid college grade. This requires a 2 digit number where the first number is in the range 1-4, followed by a ".", followed by any single number. _____

4.31 Occurrence Modifiers are placed after a character to say how many times that character may appear. For example "`*`" says there may be zero or more occurrences of the ***previous*** character.

Modifier	**Meaning**
`*`	zero or more occurrences of the preceding
`\?`	zero or one occurrences of the preceding
`\+`	one or more occurrences of the preceding
`\{x\}`	x number of occurrences of the preceding
`\{x,y\}`	x to y number of occurrences of the preceding
`regexp1 | regexp2`	regular expression 1 **or** Regular expression 2

Examples

`'[0-9]*'`	Nothing or any number of digits. For example `123` or `99` or `4509`
`'.*'`	Nothing, or any number of any characters
`'[0-9]\{3\}'`	Any 3 character number `000-999`, same as `[0-9][0-9][0-9]`
`'[0-9]\{3,4\}'`	Any 3 or 4 character number `000-9999`
`'RBI | Runs Batted In'`	The string "`RBI`" or the string "`Runs Batted In`"

A. Practice with regular expressions using occurrence modifiers. Remember that the regular expression `'Docket[0-9]\{3}\'` will match any line with the characters '`Docket`' followed by any three numerals, and `'Window-[A-Z]\{2,5}\'` will match any line with the characters '`Window-`' followed by at least 2 but no more than 5 uppercase alpha characters. For each of the following, write the correct regular expression:

The string "case" or the string "Case" followed by a "-" and any three numbers. For example "case-332" _____

The string "Room" followed by a "space", any two numbers and any Uppercase letter. For example "Room 22D" _____

The string "Name:" followed by at least 2 but no more Than 10 uppercase or lowercase letters. _____

B. More practice with regular expressions using occurrence modifiers. Remember that the regular expression for a 4 digit number followed by the "-" character is `'[0-9]\{4\}'` For each of the following, write the correct regular expression:

A simple phone number, which is a 4 digit number, followed by a "-" followed by a 3 digit number. For example "332-3344" _____

A phone number with area code. This is a 3 digit number inside parentheses, followed by a space, a 4 digit number, a "-" and finishedwith a 3 digit number. For example "(509) 332-3344" _____

An IPv4 address. This is a 4 part number, with each part separated by a "." Each number part should be in the range 1 to 255, but you can make the range 0 to 999 to make it simpler. In other words, each number should be from 1 to 3 digits. For example 123.999.1.42 _____

C. More practice with regular expressions using occurrence modifiers. Modify each of the following regular expressions to use `\{n\}` to specify the number of occurrences of a character or set of characters. For example `'aaa'` would be replaced by `'a\{3]}'`

`'99999'` _____

`'777-7777'` _____

`'[a-z][a-z]--[a-z][a-z]'` _____

`'[a-zA-Z][a-zA-Z][a-zA-Z]'` _____

`'[a-z][a-z]:[0-9][0-9][0-9][0-9]'` _____

4.32 Positional Modifers or Anchors specify the string's position in the line. There are two anchors, `^` and `$`. `^` represents the start of a line, while `$` represents the end of the line. For the anchors to function as anchors they must be in the For example `^Tony` would match the string "`Tony`" only if it were the first thing on the line. `Tony$` would match the string "`Tony`" only if it were the last thing on the line.

If you use the ^ or $ anywhere besides the start or end respectively, then they will be interpreted as a normal character. For example in the regular expression "`This is a caret ^`" the ^ is just a caret and loses any special meaning since it's not at the beginning of the line.

Positional modifiers can also be used to specify that a pattern must be after the beginning of a word, or before the end of a word. A word is defined as a set of characters with a space character or newline before and after the

characters. To say that the characters "red" need to be in word by themselves the regular expression would be '\<red\>' Ideally we'd like to just write '<red>', but the < and > need to be protected so they each have to be proceeded by a slash. Note that '\<red\>' will match red at the beginning and end of a line as well, which is what makes it different than just using spaces before and after the word as in ' red '

Regular Expression	Meaning
^	Beginning of line
$	End of line
\<	Beginning of word
\>	End of word

Examples

'^Tony'	The string Tony at the beginning of a line
'Tony$'	The string Tony at the end of a line
'^[0-9][0-9]'	Any number 00-99 that are the first characters in a line
'Room [1-2][0-9][0-9]$'	"Room" followed by 100 through 299 as the last characters in a line
'^Title: ....[0-9]$'	"Title:" followed by any 4 characters, followed by 1-9 as the only characters on a line
'\<the\>'	The word "the". Does **not** match "then" or "breathe"
'^\<Tony\>'	The word "Tony" as the first word on the line

A. Practice with regular expressions using positional modifiers. Remember that '^Tony' will match any line with the characters 'Tony' at the *start* of the line, '$Tony' will match any line with the characters 'Tony' at the *end* of the line, and '\<Tony\>' will match any line with whitespace or newline characters before and after the characters 'Tony'. For each of the following, write the correct regular expression:

The string "CBC" at the beginning of a line: _____

The string "CBC" at the end of a line: _____

The string "CBC" with whitespace before and after the phone number _____

A simple phone number at the end of a line _____

A simple phone number with whitespace before and after the phone number _____

B. Practice with regular expressions using wildcards for occurrence modifiers. Remember that 'Station[0-9]\+FM' will match any line with the characters 'Station' followed by 1 or more numeric characters, and ending with the characters 'FM'. For each of the following, write the correct regular expression:

A DNS name such as company.com. The pattern is a whitespace character followed by at least one but possibly many alphanumeric characters, followed by a '.' and finished with a 3 character string. You can simplify the pattern by having the last 3 characters can be any upper or lower case alpha characters.

_____

The string "http://" followed by a DNS name.

An email address such as `user@company.com`. The pattern is a whitespace character followed by at least one but possibly many alphanumeric characters, followed by a DNS name.

C. More practice with regular expressions using wildcards for occurrence modifiers. Assume that you have a data file that uses ":" characters for delimiters. For example:

firstName : lastName : title : email : cellPhone : homePhone

If you wanted to search for lines where the lastName is "Moreno" the regular expression would be `'^.*:Moreno'` If you decipher this, the `^` means the pattern must start at the beginning of the line. The `.` means any character, but it is modified by the `*` to be 0 or more characters. The `:` will match a colon character, so putting `.*:` together means it will match anything up to the first colon, which also means that there can be anything in the first field. Having `Moreno` in the regular expression then limits this pattern to only matching lines that have `Moreno` in the second field.

Even though `'^.*:Moreno'` will find `Moreno` is the second field; it will also match `Moreno` in fields 3, 4, 5 and 6. If you inspect the regular expression closely you can see why this happens. If `Moreno` is in field 3, it will also be proceeded by any number of any characters and a `:` character. It's can be proceeded by more than one `:`, because the pattern isn't limited to just one `:` character. To remedy this and only get the second field, we will need to specify that there must be one : before and four after. The regular expression to do this is:

```
'^.*:Moreno:.*:.*:.*:'
```

Write the regular expression to find lines where the title field is "`Mrs.`"

Write the regular expression to find lines where the email address field contains any email addresses with the Domain Name of "`columbiabasin.edu`". Hint 1 – It will be the fourth field. The email address will be at least one but possibly many alphanumeric characters, followed by the characters `@columbiabasin.edu`. You can just check for all lowercase characters in `columbiabasin.edu`.

D. Practice with regular expressions using '\|' to match one pattern OR another pattern. Remember that `'CBC\|Columbia Basin College'` will match any line with the characters '`CBC`' or the characters '`Columbia Basin College`'. For each of the following, write the correct regular expression:

Write the regular expression to match either '`CS`' or '`Comp Sci`'.

Write the regular expression to match either '`Comp Sci`' or '`Computer Science`'.

Write the regular expression to match either '`CS`' followed by three numbers, or '`Comp Sci`' followed by three numbers.

_____

Write the regular expression to match a simple phone number that either contains a dash or not. That is, the numbers will be of the form '`xxx-xxxx`' or '`xxxxxxx`'

_____

4.33 PERL Extensions

PERL Extensions **NOTE** - these may not be supported or you may have to use `grep -P`		
\d	Any single number	`[0-9]`
\D	Any character that is not a number	`[^0-9]`
\w	Any word made from alphanumeric characters and "_"	`[a-zA-Z0-9_]`
\W	Any non-word character	`[^a-zA-Z0-9_]`
\s	Any whitespace character such as (space, tab, newline)	
\S	Any non-whitespace character	
\n	The newline character	
\r	The carriage return character	
\t	The tab character	
\nnn	The ASCII character with the octal value nnn	
\xnn	The ASCII character with the hex value nn	

4.34 POSIX Character Classes

POSIX Character Classes			
POSIX	PERL	Description	Classic regular expression
`[:alnum:]`		Alphanumeric characters	`[a-zA-Z0-9]`
`[:alpha:]`		Alphabetic characters	`[a-zA-Z]`
`[:ascii:]`		ASCII characters	`[\x00-\x7F]`
`[:blank:]`		Space and tab	`[ \t]`
`[:cntrl:]`		Control characters	`[\x00-\x1F\x7F]`

`[:digit:]`	`\d`	Digits	`[0-9]`
`[:graph:]`		Visible characters (i.e. anything except spaces, control characters, etc.)	`[\x21-\x7E]`
`[:lower:]`		Lowercase letters	`[a-z]`
`[:print:]`		Visible characters and spaces (i.e. anything except control characters, etc.)	`[\x20-\x7E]`
`[:punct:]`		Punctuation and symbols.	`[!"#$%&'()*+,\-./:;<=>?@[\\\]^_`{|}~]`
`[:space:]`	`\s`	All whitespace characters, including line breaks	`[ \t\r\n\v\f]`
`[:upper:]`		Uppercase letters	`[A-Z]`
`[:word:]`	`\w`	Word characters (letters, numbers and underscores)	`[A-Za-z0-9_]`
`[:xdigit:]`		Hexadecimal digits	`[A-Fa-f0-9]`

E. **Practice building command pipelines**. Assume that you want to write the command(s) to create a file containing a list of all files in your home directory starting with a period "." You could try using `ls ~/.* > filename` but this will also list the some extra things like. You can however use the `find` command, which will list files of certain types. The general syntax for the find command is `find directoryToSearch –options`

To list the files in your home directory, you can either `cd` to your home directory and then use `find .` or be in any directory and use `find ~`

One of the `find` options we will need to use is the `–type` option. Using `–type f` tells `find` to only list files, not directories. We will also use the `–maxdepth 1` option to say don't traverse into any directories. Using

```
find ~ –type f –maxdepth 1 | grep '^\.' > filename
```

will return a list of *all* the files in your home directory. But if you look at the output from this command you will see that every file starts with the characters "`./`" To remove these characters you can pipe the output from the `file` command through the following `sed` command

```
sed s@\./@@
```

Remember that the character following the `s` is the delimiter, so in this case we have `@` signs as the delimiters or `@old@new@` The old pattern, the one we want to replace, consists of the characters "`./`". But since "`.`" means any character in a regular expression we should protect it by adding a "\" in front of it. So the pattern we want to replace is "`\./`" We want to delete these characters, so we will replace them with nothing, which is why we use two `@` symbols in a row, with nothing between them.

So if we run the output from the file command through the `sed` command, we will get a list of files from our home directory. To finish this you need to find only the files that start with the character "`.`" Use the `grep` command and what you know about regular expressions to find lines that have a "`.`" as the first character in the line. Just remember that you will have to protect the "`.`" since it is normally used as a wildcard to match any character.

UNIX Pronunciation Guide

!	Exclamation Point	Bang
#	Pound Sign	Pound
$	Dollar Sign	Dollar
%	Percent Sign	Percent
&	Ampersand	And
*	Asterisk	star or splat
()	Parenthesis	Parens
(	Left Parenthesis	open paren
)	Right Parenthesis	close paren
-	Hyphen	minus or dash
.	Period	Dot
/	Slash	slash or whack
\	Backslash	Back whack or slosh
@	At Sign	At
[ ]	Brackets	square brackets
_	Underscore	Underline
‘	Single quote	Tick
“	Double quote	Double tick
`	Grave Quote	Back tick
{}	Braces	curly braces
|	Vertical Bar	Pipe
~	Tilde	Twiddle
#!	Pound Sign and Exclamation Point	sh'bang or wallop
usr		User or U.S.R.
etc		Et see or E.T.C., rarely etcetera
lib		Lib or libe, rarely library

Review

1. What is stdin?
2. Why would you want to change stdin?
3. What is stdout?
4. What is stderr?
5. Why would you want to change stdout or stderr?
6. How do you tell a command to send it's output to a file? If you do is the file automatically overwritten? How do you control this?
7. How do you tell a command to send output to end of a file? What if file does not already exist?
8. How do you tell a command to get input from a previous command?
9. How do you tell the *cat* command to take input and put it into a file? What are stdin and stdout in this situation? How do you tell *cat* you're finished?
10. How do you take output from *ls* and view it a page at a time
11. What is a regular expression?
12. What do following wildcards represent? * ? ^ $
13. What would the regular expression be to see all files that end in .txt?
14. What would the regular expression be to see all files that start with . (hidden files)? What would ls command line look like?
15. What is *grep*?
16. What is *sed*?
17. What would command be to find all accounts that start with cs in password file?
18. What would command be to find all accounts that start with cs in password file, and write them to new temp file?
19. What would command be to find all accounts that start with cs in password file, change cs to cgi, and write them to new temp file?
20. Can you sed from a file back into the same file ?
21. What does following match file.'*' ?
22. What does following match file."*" ?
23. How do I get shell to run a command inside quotes? Why would I want to do this?
24. State the three standard devices associated with shell procedures.
25. Rewrite the following command so that it executes correctly. who | users.log

DOS	*UNIX*	
CD (no args)	pwd	Print current location in directory structure
DIR /W	ls	Print list of file names
DIR	ls –al	Print detailed list of file names
CD	cd	Change directory
TYPE	cat	Print file contents
TYPE | MORE	more	Print file contents, one screen at a time
COPY	cp	Copy file
RENAME or MOVE	mv	Move file
DEL or ERASE	rm	Delete file
MKDIR or MD	mkdir	Make a directory
RMDIR	rmdir	Remove directory
HELP	Man apropos	Print help for a specific command
DATE and TIME	Date	Display current date and time
PRINT	Lpr	Print

BASIC UNIX COMMANDS

Most UNIX commands have several options, and each option for a particular command produces unique output -- this makes for fewer commands. An option is normally preceded by a "-" (dash), but multiple options usually only need one "-" (i.e., "ls -al").

cal	Print a calendar. ("cal 9 1992" prints month of September 1992)
cat	Concatenate and print files. ("cat .login" displays your login initialization file)
cd	Change working directory. ("cd /" and then "ls" will print the root directory; "cd" by itself will return you to your home directory)
chmod	Change mode (file permissions) on file(s) and/or directory(s).
cp	Copy files. ("cp filel file2" will copy the contents of file1 into file2. Note that if file2 already exists it will be overwritten with the contents of file1)
date	Print current time and date.
Df	(Disk Free) Show the current disk usage by partition
Du	Show how much disk is being used
echo	Echo arguments. (Try echo this is fun)
Env	Display the values of your environmental variables
exit	Logout and terminate session.
File	Show what type of informationa file holds
finger	Provide user information. (Try finger your_user_ name)
grep	Search for a string or regular expression. ("grep path .cshrc" will find all occurrences of the word "path" in the file .cshrc)
history	Provide a list of commands you've already executed
Kill	Stop a process
ls	List contents of a directory. ("ls -al" gives a long listing of all files in current directory, such as permissions on the file, file ownership, and the date the file was last modified)
mkdir	Make a directory. ("mkdir testdir" will create a directory called testdir)
more	(BSD) or pg (System V) Control scrolling of files. ("more filenname" or "pg filename" will scroll the file called filename on the screen. Hit the SPACE bar to scroll one screen forward)
mv	Move or rename files or directories. ("mv file1 file2" will move file1 into file2. Note that if file2 already exists its contents will be overwritten by file1)
passwd	Change password. (Prompts for information)
Ps	Show a list of currently running processes
pwd	Print working directory.
rm	Remove file(s). (Try "rm -i filename")
rmdir	Remove directory.
sort	Sort and collate lines.
umask	Set a mask which controls the permissions for any new files you create
vi	Standard UNIX editor.
who	Who is logged in on the system.

Description: *vi* is one of the UNIX editors. While it may seem primitive to those used to windows-based editors, it is a very powerful tool, and essential if you do not have a PC to do your editing on.

Objectives: At the end of this section the student will be able to:

1. Start vi, perform basic editing tasks, and save files.
2. Switch between Input Mode and Command Mode in vi.
3. Perform advanced editing tasks using Line Commands

Study Guide: To pass the test on this topic, you will need to know the following terms and concepts:

1. vi basics
starting and stopping vi
Command Mode vs. Input Mode

2. Navigating thru a document
simple cursor motion
move by pages, words

3. Input mode
entering and leaving input mode
typing and the input buffer

4. Deleting text
characters
words

5. Saving and Exiting
: commands

6. Changing and replacing text

7. : commands
line numbers
running UNIX commands

Since the configuration files in UNIX are plain text, it's extremely beneficial to learn how to use a text editor to update and change them. There are two editors, EMACS and vi, that have been included with every UNIX/Linux distribution since nearly the dawn of time. Which editor is better depends on which one you know, because most people only learn one or the other; and then they would rather fight than switch as the saying goes.

I learned vi, so that's what you'll learn too. (By the way, just like most things with weird spelling in UNIX there are a couple of ways to say vi. The first is to say "vee eye" and the second is to say "vigh" so that it rhymes with "sigh. Saying 6, like the Roman numeral vi is always wrong.)

If you have any experience with UNIX you probably know that there are several GUI based editors that you can use including `gedit` and `gnp` that are very similar to Notepad on Windows. So why not learn one of these? The main reason is that as an administrator you may need to do some editing during installation, when the GUI tools aren't available. The other reason is that if you get past the basics, you'll find that vi is a very powerful tool that is so well integrated with UNIX that you can use any of the shell commands within the editor to speed up your work.

The first thing to understand about vi is that it is NOT a wysiwyg editor. Once you get comfortable with it you will find that it's very powerful, but it's definitely not Word.

The next important concept about vi is that it has three different modes:

1. **Command Mode** – This is the mode that vi is in when you start it. Anything you type in this mode is interpreted as a command, and is not added to the text in the input buffer (what you see on the screen). The commands allow you to do things like cut and paste, move around etc.

2. **Input Mode** – In this mode anything you type is added to the input buffer. So you can type the text that you want to add to the document. Some versions of vi allow you to also delete text from the input buffer while you're in Input Mode, but some versions force you to go back to Command Mode to delete text. Since vi starts in Command Mode you have to tell it that you want to change modes by typing an appropriate command. (See below for a list). To switch from Input Mode back to Command Mode you always hit the `<esc>` key.

3. **Single Line Command Mode** – This mode makes vi act like some of the original (and ancient) text editors that could only display a single line of text at a time, and edit that single line. On UNIX systems it was editor named `ex` on DOS systems it was `edlin`. It may seem counter intuitive, but this is where some of the real power of vi lies. To get into this mode you must first be in Command Mode, then type the colon character ":". After you type the ":" the cursor will move to the lower left corner of the screen. At this point anything you type will be interpreted as a Single Line Command Mode command. After you hit the `<enter>` key the command will execute and you will be returned to normal Command Mode. I'm going to refer to this as ex mode, because Single Line Command Mode is too much to type.

5.1 Before you actually start using vi you may need to set up some parameters on both your telnet program and on the UNIX/Linux host. `vi` and the UNIX host communicate about the type of terminal you are emulating to help `vi` decide how many lines your terminal has, when to scroll etc. If these settings don't match the settings of your telnet session then vi (and you) will be very very confused. To ensure that everything is set correctly you should do the following:

In your telnet window
- Before connecting, go to the Terminal / Preferences menu
- Set the buffer size to 24 lines

After you log in type the following commands (be very careful about the spaces, you do not want any around the equal sign.)
- `set term=vt100`
- `stty rows 24`

The best way to learn about *vi* is to use it. This section gives step by step instructions for creating a file. In each step you see an action to perform and then the necessary keystrokes. Don't be concerned with complete accuracy, concentrate on learning the concepts. Learn about the concepts of using *vi* to create a file, how to move between command and input modes, and how to save your results. If you run into difficulties, you can quit and start over by pressing `<esc>` to make sure you're in Command Mode; then typing : **q!** and press `<enter>`.

Start	
Run from the shell. Do not type the "$", this just indicates the shell prompt	
`$ vi`	Starts vi with a new blank document
`$ vi filename`	Starts vi and opens the file named `filename`

Save & Exit	
Run these commands in vi. All of the commands for writing or exiting are in ex mode, so type `<esc>` to make sure you're in Command Mode, and then a colon ":" to get into ex mode.	
`:q`	Quit. This works if you haven't made any changes, but if you have any unsaved changes vi will display an error message and refuse to exit.
`:q!`	Really Quit. This forces vi to quit, even if you have unsaved changes.
`:wq`	Writes the file and then quits
`:x`	Writes the file and then quits. Just like `:wq`
`:w`	Writes the file but keeps vi running. (Like File > Save)

A. Make sure you are in your home directory, or one of your own sub directories; so that you have permission to create a file. Start *vi* by typing **vi** and pressing `<enter>`. You see the screen full of flush-left tilde "~" characters with the cursor in the upper left corner. This just shows you that you are working on a new file with nothing init.

B. Go into input mode to place characters on the first line, by pressing the **a** key (don't press `<enter>`) . This is the command to append characters to the first line, so you will not see the character “a” on-screen.

C. Add lines of text to the buffer. Type:

Things to do today. `<enter>`

a. Practice vi. `<enter>`
b. Buy Tony pizza for lunch. `<enter>`
c. Switch to AT&T long distance to thank them for inventing UNIX. `<enter>`
d. Spend more hours practicing vi. `<enter>`
e. Read old Dilbert cartoons about UNIX, finally "get it". <enter>

You can use the <Backspace> key to correct mistakes on the line you're typing. Don't worry about being precise here: this example is for practice. You will learn other ways to make changes in some of the later exercises.

D. Go from input mode back to Command Mode, by pressing the `<esc>` key. You can press `<esc>` more than once without worry; if you're already in Command Mode nothing happens although you may hear a beep.

E. Save your buffer in a file called vipract.1 by typing **:w vipract.1** and press `<enter>`. The characters **:w vipract.1** appear on the bottom line of the screen (the *status line).* (The characters should not appear in the text on your screen. If they do, you're still in Input Mode.) The **:w** command writes the buffer to the specified file. In this case, this command *saves or writes* the buffer to the file vipract1.

F. See your action confirmed on the status line. You should see something similar to the following on the status line:
`"vipract.1" [New File] 5 lines, 136 characters`

This statement confirms that the file vipract.1 has been created, is a new file, and contains 5 lines and 136 characters. Your character count may be different if you didn't type the information exactly as specified (spaces count as characters).

G. Exit vi. Type **:q** and press `<enter>`.

When vou type : **q,** you are still in Command Mode and see these characters on the status line. When you press `<enter>`, however, *vi* terminates and you return to the shell prompt.

5.2 Another way to learn about vi is to read the built in tutorial. This is available on some Linux distributions and is accessed by typing: `vimtutor`

5.3 Editing a file that already exists is done by typing vi followed by the name of the file and press `<enter>`. Try this on the file you created by typing: `vi vipract.1`

Add the following text. When you are done save and exit.

Things to do tomorow. `<enter>`
a. Practice vi some more. `<enter>`
b. Buy Tony Chinese food for lunch. `<enter>`
c. Call Al Gore and thank him for inventing the Internet `<enter>`

For the following exercises you will need to copy the files from /home/viExercises to one of your own directories.

5.4 Adding text is done by changing from Command Mode to input mode. There are several commands used to change into input mode, which command you use determines where you begin inputting text relative to the current cursor position.

A. Open the file insertTest
B. Position the cursor somewhere near the center of the text buffer
C. Type one of the characters shown in the left column table below. You will enter input mode, so anything you type will be added to the buffer, until you hit the `<esc>` key, which will take you back to Command Mode.
D. For each character in the left column, write a brief description of what happens when you hit the key.

a	
i	
A	
I	
o	
O	

5.5 Adding non-printing characters. Adding "normal" text in vi is simple, just go into insert mode and start typing. But sometimes you want to add non-printing characters. You can't add the non-printing characters directly by typing them, but there is a way to insert them. This can be done by first typing `<ctrl-v>` then a carat "^" followed by the character. For example, to add a `<ctrl-M>` type `<ctrl-v> ^ M`

5.6 Positioning the cursor is done via the `h,j,k,l` keys, or on some systems via the arrow keys. When I first learned vi I wondered why `h, j, k` and `l` for moving? Especially using the `l` key to move right, shouldn't `l` be for moving left? It took someone who knew how to type to tell me that these are the "home keys" for the right hand, so you're fingers should always be on these keys.

When you are in Command Mode, you can position the cursor to the position where you wish to insert additional text, delete text, correct mistakes, change words, etc. There a literally hundreds of ways to move around a file, so you should concentrate on finding a couple that are easy for you to remember.

Practice moving around the file you created. Other keys you can use to position the cursor in Command Mode include `<space>, <enter>, +, -, 0, $, w, b, e, H, M, and L`. Position your cursor somewhere near the center of the file, then type one of the preceding characters. In the space below, write down how each of these keys affects the position of the cursor.

h	One character left
j	One line down
k	One line up
l	One character right
e	
w	
3w	
b	
3b	
0	
$	
+	
-	
)	

(	
G	
gg	
<ctrl+f>	Move down (forward) one screenful of lines.
<ctrl+b>	Move up (backward) one screenful of lines.
<ctrl+d>	Move down 1/2 screenful of lines.
<ctrl+u>	Move up 1/2 screenful of lines.
M	
H	
L	

5.7 Undoing your last action can be done, as long as you haven't saved the file yet. The undo command is done from Command Mode, by typing **u**. It can be used to undelete, or if you inadvertently added some text in input mode, use <esc> to return to Command Mode, and undo everything you just did while in input mode.

Make sure you are in Command Mode, delete a line of text, then use the undo command to retrieve it. Next, enter input mode (use whichever key you like), enter 2 lines of text. Return to Command Mode and use the undo command, what happened to the text you entered ?

Some versions of vi have multiple levels of undo while others only have a single level. If there's only one level then you can undo the last change, but if you hit the u key again it will redo the last action. If there are multiple levels then each time you hit the u key you will undo another command.

u	Undo previous command.
:e!	Discard all edits since file was opened.

5.8 Deleting text is done from Command Mode (unless you are in Input Mode). *vi* allows you to delete a character, a word, all the text to the end of line, an entire line, or a number of lines. Again, vi provides many ways to do this, so you should find one or two that you can remember and stick with them.

Open the file deleteTest and position the cursor somewhere near the center of the document. Ensure that you are in command mode and then type the following character(s). In the space below, describe what happens when you hit the corresponding key(s):

x	
X	
dw	
D	
dd	
dG	

5.9 Visually deleting text – Newer versions of vi allow you to delete sections of text like you would in a wysiwyg editor. You may want to try this method, but make sure that you know one of the other methods as it's not available in all versions of vi.

The first step is to mark the section of text you want to delete. This is done in command mode by typing the `V` command and then using the movement keys (`h, j, k, l` etc.) to move around and highlight the section of text to delete. Once you have the text selected, hit the `d` key to delete the selection.

5.10 Changing or replacing text from the buffer is done from Command Mode. *vi* allows you to change a word or from the cursor position to the end of the line, or you can replace a single character, or a sequence of characters.

`r`	replaces the current character
`C`	replaces all characters to the end of line
`cw`	replaces the current word
`s`	replaces the current character, and leaves you in input mode
`n``s`	where *n* is a number, replaces n characters from the current position, and leaves you in input mode

Open the file changeTest and make the following changes:

A. In line 1 change the word "barn" to "brain"
B. In line 2 change "ate three" to "lately"
C. In line 2 change "feel" to "seem"
D. In line 3 change "bunny" to "funny"
E. In line 4 change "y" to "why"
F. In line 5 change "this guy" to "the sky"

5.11 Use vi to create a simple resume.

5.12 Use *vi* to create a copy of your schedule for this quarter. At a minimum include the class number and time. Use tabs to make the columns line up.

5.13 Use *vi* to create a "cheatsheet" of *vi* commands. Include at a minimum all of the commands in this exercise.

5.14 Customizing – You can customize the way vi behaves or the display by using the:`set` command. For example:

```
<esc>:set list
```

will have vi display non-printing characters such as tabs, line-feeds and carriage-returns. The following table shows some of the `set` options. The complete list of options will vary by distribution.

SET_COMMAND	EFFECT
`all`	Display all of the `:set` options
`set`	Display all of the current settings for `:set` options
`number or nu` `nonumber or nonu`	Display line numbers
`autoindent or ai` `noautoindent or noai`	Set or unset autoindent. When set, new lines will start at the same column as the previous.
`ignorecase or ic` `noignorecase or noic`	Set or unset ignorecase. When ignorecase is set, searches, substitutes and global commands are not case sensitive.

`fileformat=unix` `fileformat=dos`	Determines which characters to insert at the end of each line. DOS lines end with cr and lf, while UNIX lines end with lf.
`list` `nolist`	Show non-printing characters.
`wrapmargin=`*n*	Sets length of each line top *n* characters. If you type more characters the line will automatically wrap.
`tabstop=`*n*	Number of space characters to display for tabs.

Another thing you may be able to customize is the colors used by vi. You can change them by using the colorscheme command. For example:

```
colorscheme  darkBlue
```

sets the color scheme to darkBlue. Not all colorschemes are available on every release, or they may not be available at all. But if they are available they will be in the directory: `/usr/share/vim/vim61/colors/`

5.15 Saving Settings. If you set options in vi they will stay set until you exit. Luckily vi has a settings file that reads each time it starts, so you won't have to type in your settings every time you run vi. This file must be in your home directory, and it's usually called .exrc but some on some distributions it may be .vimrc or .gvimrc.

The startup file can contain set commands, other commands such as colorscheme, or comments. Comments are on lines that begin with the double quote character as shown in the following example file:

```
set number
" the colorschemes are in /usr/share/vim/vim61/colors/
colorscheme  darkBlue
```

Test this by doing the following:

A. Ensure that you are in your home directory
B. Edit the file ./vimrc
C. Add some set commands and change your color scheme
D. Save the file
E. Start vi and verify that the set commands were executed and the color scheme was changed.

5.16 ex Mode Commands – I'm sure there are many of you who compare vi to an editor like Word and find it not only lacking in every way, but obtuse and difficult to learn and use. But believe it or not there's an editor named ex that's even harder to use and learn. This is kind of comparing Starbuck's coffee (easy like Word) to making your own coffee (ok if you know how, like vi) to growing your own coffee beans and welding together your own coffee pot (hard no matter how you look at it, like ex).

The ex editor only allowed you to see and edit a single line of text at a time, which made it difficult to use. But it had several commands that could be used to modify several lines at once. These commands are available in vi, and you can run them in several ways including:

1. In Command Mode, type the colon character ":". You can then run a single ex command
2. You can create a file that contains nothing but ex commands, and then run it like a macro using the `:source` command.
3. You can force vi to only run ex commands by starting it with the `-e` option. (I don't think anyone has ever done this, but the option is there.)

Several of the ex commands for saving and exiting were listed above. This table lists more, plus commands for opening or inserting other files.

`:e fileName`	Edit a different file . Vi will display an error message if you have unsaved edits.
`:e! fileName`	Edit a different file even if there are unsaved edits
`:e!`	Discard all edits and reopen the current file
`:r fileName`	Inserts the contents of `fileName` below the current line.
`:r!shellCmd`	Runs the `shellCmd` and inserts the output below the current line. For example `:r! ls` would run the ls command and insert the list of files.
`:w newFileName`	Writes the current file to a new file named newFileName. However, this is not like **File > Save As**, because vi thinks you are still editing the original file. If you do another `:w` it will write to the old file, not newFileName. You can force vi to edit newFileName by using `:e newFileName` immediately after the `:w newFileName`.
`:3,7w fileName`	Writes lines 3 through 7 to fileName.
`:2,7w>>fileName`	Appends lines 2 through 7 to filename.

5.17 Running UNIX commands via ex Mode – Some UNIX/Linux releases have a version of vi that allow you to run commands that you would normally run on the command line, but have them affect the current text buffer. You could do the same type of thing by exiting vi and running command on the text file, but being able to do it from within vi is pretty convenient. Just like the other ex commands, the systems commands can affect the entire file or just a range of lines.

ex command	
`:%!sort`	Sort the file
`:%!sort -u`	Sort the file and delete duplicate lines
`:%!cat -n`	Add line numbers to the start of each line
`:%!cat -b`	Add line numbers to the start of each non-blank line

Test this by doing the following:

A. Open the file exCommands
B. Ensure that you're in command mode
C. Inspect the file to see the order of the lines
D. Type `:%!sort`
E. Inspect the file to see the order of the lines. Has the order changed?
F. Type `:%!cat -n` (The `-n` option tells cat to add line numbers to the each line being sent to stdout.)
G. Inspect the file. Has a line number been added to each line?

H. Assume that you had not made these changes in vi. Could you accomplish the same results by typing commands on the command line? What would you type?

5.18 Editing Multiple Files – It's possible to start vi and give it a list of files to edit. Typically this is done by using something like `vi *.htm` vi will open all files that end in `.htm`, but they can only be edited one at a time. To switch the currently open file use the following commands in ex mode.

`:buffers`	Lists all the file buffers
`:bn`	Go to the next file. (The `b` in the command stands for buffer, which is how vi refers to each open file as it's a bunch of text buffered in memory.) If you are on the last file you will "wrap around" back to

	the first file.
`:bp`	Go to the previous file.
`:b`***n***	Go to file number ***n***.
`:bl`	Go to the last file.
`:rew`	Go back to the first file (rewind)
`:bd`	Delete the current file from the buffer list. (The last saved version will still be on disk.) the file buffers are then renumbered, so that the list is continuous with no missing numbers

Test this by doing the following:

A. Open all of the files with the word "multiple" in the the filename by typing: `vi multiple*`
B. Ensure that you are in command mode
C. Type `:buffers` (
D. What files are listed as being in the buffers?
E. Which file is currently open for editing?
F. Type `:bn`
G. Now which file is open for editing?
H. Type `:bl`
I. Now which file is open for editing?
J. Type `:rew`
K. Now which file is open for editing?

5.19 Basic Cut and Paste Entire Lines– Like any good editor vi allows you to cut and paste. In vi you have to tell the editor which lines you want to cut or copy, since you can't highlight them like you do in a wysiwyg editor. This is done by positioning the cursor in the line you want to cut and typing `yy` for copy or `dd` for cut. One of the limits of this set of commands is that you have to copy/cut entire lines, you can't do parts of a line. You can copy or cut multiple lines by prefixing the command with the number of lines you want. For example, to cut 5 lines, position the cursor on the first line, then type: `5dd`

Pasting is done by positioning the cursor in the line above where you want the pasted lines to end up, then pressing the `p` key. Or you can paste above the current line by using the `P` command.

Practice this by opening the file copyAndPaste and doing the following:

- Copy the top 10 lines and paste them at the end of the file
- Cut the lines 11-20 and paste them at the end of the file

5.20 Basic Cut and Paste Individual Words– The following table shows commands that can be used to copy/cut words instead of entire lines. You paste them in using the `p` or `P` commands, but instead of ending up on the next/previous line they are pasted in the same line.

`dw`	`yw`	Current word
`d$`	`y$`	To end of line
`d0`	`y0`	To start of line
`d}`	`y}`	To end of paragraph
`d{`	`y{`	To start of paragraph

5.21 Registers (the vi clipboard) – There may be occasions when you want to cut/copy and paste more than one section at a time. In Windows when you cut/copy multiple sections you can access the clipboard and choose which one you want to paste. Vi has the concept of registers, which allow something like the Windows clipboard.

The way that it works is if you want to store more than one section you have to tell vi which register to store it in by adding the register identifier, a single letter, before the cut/copy command. For example `ayy` would copy the current line and store it in register "a". When you paste, you can choose the register to paste from by adding the register identifier before the paste command. For example `ap` would paste the contents of the a register. The default register doesn't have an identifier, so if you just use `dd`, `yy` or `p` you will be using the default.

The nice thing about just using the default register is that it's not hard to remember what it holds. But once you start storing multiple sections it can be a little more difficult to remember which registers you used and what they hold. Luckily you can use the Command Line Mode command `:reg` to display a list of all the registers and their content.

5.22 System Register – Some versions of vi (all versions of vim) have a special register called the system register. Unlike the normal registers which can copy/cut and paste sections of text in a document, the system register allows you to gather text from outside of vi and paste into a document. For example, if you want to get some text from a web page, you can highlight the text you want, then switch to vi and use the `*p` command to paste the contents of the system register.

5.23 Searching is done from ex Mode by typing **/** or **?** followed by the *string* you wish to search for. */string* searches forward thru the text, while **?***string* searches backward.

`/Tony`	Search from the current position forward for the next instance of `Tony`.
`?Tony`	Search from the current position backwards for `Tony`
`/`	Search down for the next occurrence of whatever you searched for last.
`?`	Search up for the next occurrence of whatever you searched for last.
`/Tony/p`	Highlight all occurrences of Tony

5.24 Search and Replace. The search and replace function in vi is very powerful because it uses wildcard patterns. The wildcards are built from the same regular expressions used by many of the shell commands such as grep and sed. All searching and replacing is done from ex mode

Basic `:s/oldPattern/newPattern`

The "s" stands for substitute and "/" characters are delimiters between the text to find and the text to replace it with. For example, to change "`tony`" to "`Anthony`" the command would be:

```
:s/tony/Anthony
```

Deconstructing this gives:

`:`	tells vi to run this command in ex mode
`s`	says to run the substitute command
`/`	is the delimiter between the command and the string you want to replace
`tony`	is the string you want to replace
`/`	is the delimiter between the old and new strings
`Anthony`	is the string you want to end up with

Typically use "/" as the delimiter, but it's possible to use other characters, otherwise it would be difficult to change text that has the "/" character in it. The substitute command will recognize the character immediately following the "s" as the delimiter. In the following example the "@" character is the delimiter: `:s@tony@Anthony`

The examples shown will find and change the first occurrence in a line. If there are multiple occurrences need to add a delimiter and a g (for global) to the end of the command. For example: `:s/tony/Anthony/g` You can also add gc instead of just g. The c stands for confirm, so you will have to answer yes or no before each change. This is good if you want to change some, but not all matches.

5.25 Search and Replace Over A Range Of Lines. Another limit of the basic form is that only makes changes to the current line. You can modify this by specifying a line number or range of line numbers to the start of the command.

If you just specify one line number then the substitution will happen on that line. For example

```
:23 s/tony/Anthony
```

says to make the substitution on line 23. If you deconstruct this you have:

`:`	tells vi to run this command in ex mode
`23`	line number to run the replace on
`s/tony/Anthony`	says to run the substitute command and change tony to Anthony

The single line form is also pretty limited in usefulness; typically when you want to make a change you want to make it for the entire file, or for large sections of the file, not just a single line. To do this, you specify a starting line and an ending line, separated by a comma. For example:

```
:12,48 s/tony/Anthony
```

Breaking this down you get:

`:`	tells vi to run this command in ex mode
`12`	line number to start the replace on
`,`	delimiter between start and end lines
`48`	line number to end replace on
`s/tony/Anthony`	says to run the substitute command and change tony to Anthony

If you want to specify a start line and how many lines to search, instead the ending line number, just replace the comma between the start and end line numbers with a semi-colon ";". For example to start the search and replace on line 12 and search the next 5 lines use:

```
:12;5 s/tony/Anthony
```

To make your life easier, and make it so you don't have to count lines, there are some shortcut symbols you can use for starting and ending lines.

`.`	the current line
`1`	the first line in the file (not really a special symbol, but very helpful.)
`$`	the last line in the file
`-n`	the line n lines before the current line
`+n`	the line n lines after the current line
`/pattern`	the next line containing the pattern (never matches the current line)

`%`	all the lines in the file. Shorthand for 1,$

For example the range `1 , .` would be all of the lines from the beginning of the file to the current line.

For each of the following ranges, write down the lines that would be affected by search and replace command:

`. , 10`	
`1 , $`	
`. , $`	
`-5 , $`	
`1 , +5`	
`/tony , $`	

For each of the following, write down the appropriate line number ranges:

	the entire file
	10 lines before current line to 3 lines after
	start of file to current line
	current line to end of file
	next line with the word "bond" to the end of the file
	start of file to 6 lines past current line

Description: The C Shell provides your interface to the computer. It takes the commands you type, and hands them to the kernel to be run. However, the shell does much more than this, and is also very customizable. This section provides you with some shell basics, and begins to expose you to the power of the UNIX shell.

Objectives: At the end of this section the student will be able to:

1. Define a UNIX shell, and be able to change shells on a temporary or permanent basis.
2. Describe the functions provided by the C shell.
3. Use history, change prompts, set aliases, etc., on a temporary or permanent basis.
4. Compare and contrast shell and environment variables.

Study Guide: To pass the test on this topic, you will need to know the following terms and concepts:

1. The UNIX Shell and it's purpose
- What a shell is - what functions they provide
- Differences between shells

2. Command Interpretation and Execution
- Prompting - read ahead capability
- Command Execution (built-ins vs. searching)
  - % command_name [option ...] [filename ...]
  - lists - command1;command2
- Search path and hash table (*rehash*)
- What happens when a command is executed

3. Customizing your environment
- Initialization Files
  - .login, .cshrc, .logout
- Shell Variables
  - set varname=value, unset varname
- Environment Variables
  - difference from shell variables, coordinating
  - env
  - setenv VARNAME value
- Aliases

4. History
- Repeating commands - !! !n !-n !string
- Quick Substitution - ^old_string^new_string

Exercises

6.1 Find out which shells are available on this computer by looking in the file /etc/shells. This is the file that the system administrator uses to list the available shell programs. List the contents of this file along with the common name for each shell program. For example the common name for /bin/sh is the Bourne Shell. (You may have to do some research to find out the common names.

Shell Program	Common Name
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

_____ _____

6.2 It may look like there are several shell programs to choose from, but there are several cases where what looks like a unique shell program is really a shortcut to another shell program. That is, there is only one shell program but it has 2 or more names. You can check this out for yourself by looking at the contents of the /bin folder. If you see something like the following it indicates that /bin/sh is really a shortcut to /bin/bash.

```
lrwxrwxrwx    1 root     root            4 Sep 21  2006 sh -> bash
```

Check each of the shells you listed in the previous exercise and determine whether they are an actual program or a shortcut. If the shell is a shortcut, list which program it points to.

6.3 Which shell are you currently running? If you type the `ps` command, it shows you the names of the programs you are currently running. What two programs should you always see? Run the `ps` command and write the results below:

6.4 You can start another shell program just like you start any other UNIX/Linux program, by simply typing the file name. For example, to start /bin/sh type: `/bin/sh` This shell will continue to run and process your commands until you exit from it. At the point you exit from the shell, your previous shell will return to processing your commands.

Start another shell program. Run the `ps` command. Is the new shell program running? Is the old shell program still running? Which shell is currently handling your commands and input?

Type: `exit`. Run the `ps` command. Is the new shell program running? Is the old shell program still running? Which shell is currently handling your commands and input?

6.5 The previous exercise demonstrated how you can temporarily start another shell, but what if you want to "permanently" change your shell. And how does the OS know which shell you want to use as your default anyway? Remember the /etc/passwd file? This is the file that contains an entry for each user, and each user entry contains the username, user id, etc. The last field for each user in the /etc/passwd file is the default shell that the user wants to use. Since this is file used to set up user accounts you can not edit it directly to change your information. However, as we have seen with `chfn` for changing your finger information, there is a utility program called `chsh` that you can use to change your default shell.

A. Use the `grep` command to find your entry in the /etc/passwd file by typing: `grep username /etc/passwd` where username is your username. Take a look at the last field, which holds your default shell.

B. Now run the `chsh` program and change your default shell to one of the shells supported by your system. The `chsh` program will ask you to enter your password, and then for the full path of the shell you want to use as your default.

Note – you want to change to an actual shell, not a shortcut. For example, if you're version of Linux has bash, and sh is a shortcut to bash, then you want to use `/bin/bash`, not /bin/sh

C. Run `grep` again to verify that your entry is the passwd file has changed. If you don't see anything in the default shell field, then you probably changed the shell to a shortcut. You can also test it by logging out and logging back in, and then running `ps` to see which shell you are using.

D. When you are finished make sure that you run `chsh` again to set your default shell back to /bin/tcsh. (You will want to be in /bin/tcsh for the rest of the exercises in this section.)

Command Processing

One of the primary tasks of the UNIX shell is to figure out what to do with what you have typed on the command line. While it may seem that the shell simply runs the command(s) that you've typed, there are actually several different places it will look. These include commands that are built into the shell, aliases you have created, commands that you have typed previously and want to reuse, commands from your search path, or commands that you explicitly point to using absolute or relative paths. All of these types of commands are demonstrated in the following exercises.

Built-ins

6.6 Built-in Commands – Every shell has some built-in commands. These commands are part of the shell, which means if you run them the shell process has the code to run them without loading another program into memory. In DOS the built-in commands were the common commands such as COPY and DIR, but in UNIX the built-ins are a little different. To see a list of the built-in commands, type: `builtins` (Note, the `builtins` command is a built-in command for the csh/tcsh shell, and may not exist in every shell.) If you cannot figure out what a specific command does, check the man pages for tcsh (or whatever shell you are running.) See if you can recognize which commands are used for shell programming.

Not all UNIX/Linux releases have the `builtins` command. Alternative methods for determining whether commands are built into the shell or are external are to use the commands `which` and `whereis`. They will show you the actual location of the command's executable file in the file system. If the command is found in file system then it's a pretty good indicator that it's not built in.

The following table shows most of the commands that are built-ins in DOS and their UNIX equivalents. Using the methods described above, determine whether or not the UNIX commands are built-in to the shell.

UNIX Builtin?	DOS	UNIX	Function
Y / N	`DIR`	`ls`	Show directory listing
Y / N	`CD, CHDIR`	`cd`	Changes directory
Y / N	`COPY`	`cp`	Copies file and directories
Y / N	`RENAME`	`mv`	Renames a file or directory
Y / N	`DEL`	`rm`	Deletes files
Y / N	`MKDIR`	`mkdir`	Creates a new directory
Y / N	`RMDIR`	`rmdir`	Removes an empty directory
Y / N	`TYPE`	`cat`	Display the content of a file
Y / N	`CLS`	`clear`	Clears the screen.
Y / N	`DATE`	`date`	Displays the date
Y / N	`SET`	`set`	Sets or displays the value of an environment (DOS) or shell (UNIX) variable

Aliases

6.7 Aliases - In UNIX an alias is a name you want to use for a command instead of the command's real name. Not all shells support aliases, but if they do they keep them in a table in the shell processes memory area.

A. To create an alias use you need to provide the name of the alias you want to use followed by the command you would like to run when the alias is typed. For example, to run the command `more /etc/passwd` each time you typed `showPass` you would type: `alias showPass more /etc/passwd`

Create the alias, and then test is by typing: `showPass`

B. The command portion of the alias can contain anything you type on the command line. So in addition to doing simple command substitution, you can build some fairly sophisticated macros. For example, you can string more than one command together by using the ";" between the commands as shown in the following example:
`alias 'showHome cd ~ ; ls -al'`

Create the alias, and then test is by typing: `showHome`

C. Create and test the following aliases

alias	**Command**
COPY	cp
del	rm –i
type	cat

D. To see the contents of your alias table type: `alias` Notice that there may be some that you did not create. These were added by the system administrator to a global login file that is processed for every user when they log in.

E. To remove an alias from the alias table type `unalias` Test this by removing the aliases you created in step C

F. The aliases you add are stored in the data section of memory for the shell process. This means that when you exit the shell the data in the alias "goes away". When you start the shell again you need to recreate each alias, which can be a time consuming. To make an alias appear to be "permanent" you can by adding to shell startup file `.cshrc` or your `.login` file.

Test this by editing your `.login` file and adding one of the aliases you created above. (If you don't know how to use one of the UNIX editors you can do your editing on the PC and then use FTP or WINscp to copy the file to your UNIX account. Also make sure that you have execute permissions on the file.) Logout and log back in, then either test the alias or check the alias table.

The `source` Command

Testing startup files like `.login` can be a pain in the rear. Is there an easier way than having to log out and log back in. After all, with UNIX can't you just execute any file by typing it's name or path? Why not just run them like any other executable? To answer these questions you need to remember what happens in memory when a process runs. If the command is not built into the shell, when it runs it gets it's own section of memory to hold it's process. And when the process is complete that section of memory is freed up and everything inside of it is gone. So you can execute the commands in the .login file by simply typing `.login`, but any changes made, such as creating an alias, will only be in memory until the .login process is done executing.

The solution to this problem is to use the `source` command. It tells the shell to process the specified file, and rather than running it in it's own process, have it affect the current process. For example, typing `source .login` processes the `.login` file, but changes the existing shell process. (This of course assumes that you're in your home directory so that `.login` exists.)

G. Placing arguments in an alias. The aliases you've created so far always perform the same commands. This can be helpful, but often times you want to be able to pass an argument or two the alias so that it can be more versatile. For example, say that you want to create an alias named `showDir` for `ls -al | more`. The `showDir` alias works as long as you want to always see the directory you're currently in. But if you want to see the listing for a different directory, say /etc, and try showDir /etc you won't get the expected results.

Why do you ask? Well, if you look at what's really happening it will all make sense. When the shell sees `showDir`, it replaces it with `ls -al | more`. Since you typed `showDir /etc` replacing the alias makes the

entire line read `ls –al | more /etc` The `more` command will throw an error because it doesn't work on directories.

What you really want to do is get the directory argument passed into the `ls` command. To do this in an alias you use `!^` when creating the alias to pass the first argument to the alias or `!*` to pass all of the arguments. For example, to make the `showDir` alias work the way we want it should be created using

```
alias showDir 'ls –al !* | more'
```

However, as you'll see below the `!` character means something special to the shell, so it has to be protected using a "\". The showDir alias should be created using:

```
alias showDir 'ls –al \!* | more'
```

Use what you've learned about passing arguments to modify the following alias. As written the alias will show you a detailed listing for all the process associated with your username. Modify it so that you can pass any user name into the `grep` portion of the alias code.

```
alias what  'ps –aux | grep $user | more'
```

H. Overriding an alias. Why create an alias? Sometimes you want to modify the default behavior of a command. For example, maybe you always want `ls` to use the `–al` options, but you just want to type `ls`, so you `alias ls 'ls –al'`. Or maybe you always want `rm` to ask before deleting any files so you `alias rm 'rm –i'`

But there may be occasions where you don't want to use the alias. For example you could be creating a command pipeline where you need to delete several files, but since you aliased `rm` to `rm –i` you have to provide a response to delete every file. You could delete the alias while you run the command pipeline, and then recreate it when you're finished. Or you could call the alias something other than `rm`. Or you could temporarily override the alias and tell the shell to ignore it. This is done by adding the slash character "`\`" to the front of the command. For example: `\ls`

Test this by creating the following alias

```
alias ls 'ls –al !* | more'
```

Now try and run the "normal" ls, not the alias by typing:

```
ls
```

Does the shell run your alias or the default `ls`?

Add the slash to tell the shell to override the alias:

```
\ls
```

Does the shell run your alias or the default `ls`?

I. Now it's time for the conceptual question about aliases. Why do you want to create them? My answer is that it's only worth doing if you have tasks that you will be doing more than once, otherwise it's easier to just type the commands.

And going the other direction, the tasks should be rather simple. By this I mean that if you have a task that you are going to repeat but it requires several arguments and will be a pain to write as an alias, then create a shell script instead. (We'll be doing this later in the quarter.)

Shell Variables

Before we talk about the next place shell may look for commands have to make a side trip and talk about shell variables. In programming a variable is similar to the memory buttons on a calculator. Variables provide a way of storing information or data so that it can be used later. But instead of pressing a button like you do on a calculator, you give the variable a name that you use to either store or retrieve the value you put in the variable. UNIX shells use many variables to track different things such as the directory you're currently in; or whether or not you want to protect files from accidental overwriting.

6.8 Predefined Shell Variables - *tcsh* provides many shell variables which you can use to make your time on earth more productive. Some of the variables along with a brief explanation of their function are shown in the following table.

`cwd`	Full pathname of the current working directory.
`histchars`	Two-characters. First character replaces ! for shell history substitution. Second replaces the carat (^) for history substitutions.
`history`	Number of commands to save in the history list.
`home`	The user's home directory.
`ignoreeof`	Prevents <ctrl-d> from closing shell and logging you off the system
`noclobber`	Prevents overwrite of existing files with redirect
`notify`	Have shell provide immediate notification when background jobs are completed instead of waiting until issuing a prompt.
`path`	The list of directories used to build hash table used for searching for commands.
`prompt`	The prompt used by the shell (default is "%")
`savehist`	The number of lines from the history list that are saved in .history
`verbose`	Display the command line to be executed after substitutions and expansion

To see a complete list of all the shell variables currently set along with their corresponding values, simply type: `set` (you may want to pipe this to `more`.)

Make a list of the shell variables that are displayed. (You can do this by redirecting the output from `set` to a file, and then copying the file to your PC.) Print this list and see if you can guess or figure out what the different variables hold?

You can see the value of a single variable by using the echo command and asking for the individual variable. For example: `echo $path` will show you the current value of the $path variable.

The Shell Path

6.9 Using the set command is fine if you want to see all of the shell variables or you can't remember the name of the exact variable you want. But if you do the name of the variable there's no need to wade through the entire listing; you can see the value of a single variable by using the `echo` command and asking for the individual variable. The only trick is that you have to add a `$` to the front of the variable name to tell the shell you're referring to the variable. For example: `echo $path` will show you the current value of the path variable. If you leave off the $ then shell and the echo command think you are talking about the some text characters.

Try the following two commands to ensure that you understand the purpose of the $:

```
echo $path
echo path
```

6.10 Setting and unsetting shell variables - If you want to change the value of one of the shell variables or create a new variable you still use the `set` command. The syntax for setting *csh/tcsh* shell variables is: `set`

`varname=value` Notice that you do **not** use the $ when you are setting the variable. You use it when you want to get the value of the variable., but not while you are setting it. Also, note that you need to ensure that there are **no** spaces on either side of the equal sign. If there are, some versions of the *csh/tcsh* shells may return an error.

For example, to create a shell variable called address and give it the value “2600 N 20th" you would type:

```
set address=”2600 N 20th"
```

You can also have a variable that acts as a flag, that is they are either on or off, but don’t hold a value. For example, the variable `noclobber` which we have seen tells the shell to not overwrite exiting files. To set shell variables which are flags type: `set varname`

To remove a cs*h/tcsh* variable use `unset`.

You can use the `set` command to either change existing variables, or create new ones. If you create a new variable, the variable name must contain only letters and numbers, no spaces or special characters. Create three new shell variables to hold your name, address and phone number. Use either the `set` or `echo` command to ensure that you have set them correctly.

6.11 The path variable – When you type a command the shell looks for built in commands and searches your alias table. Or, it looks in certain directories on the system for the command. The directories that it looks in are defined by the shell path variable. Without the path, you'd have to know the full path to every executable for every command you wanted use. For example, instead of just typing `ls` you'd have to know that it's in the `/bin` directory and type `/bin/ls`. So the shell path acts much like the DOS path, providing a list of directories that the shell should look in for commands, but with a few significant differences.

To see your current shell path type:

```
echo $path
```

To test the path use the `whereis` command to locate an executable for the following commands:

Command	**Location**
`ls`	`/bin/ls`
`clear`	
`chfn`	
`ps`	
`echo`	

Now run each of these commands. Do you have to provide the full path to the command, or can you just type the command name?

6.12 Changing your path – If you want to change your current shell path type there are two ways to do it, the hard way and the easy way. The hard way is to use the set command and type out all of the directories you want in the path. For example:

```
set path=(/bin  /usr/bin  /newDir)
```

The easy way is to still use the set command, but instead of typing all of the directories for the path, using $path to specify the directories already in the path, and then adding the new one(s) on the end. For example:

```
set path=($path  /newDir)
```

Change your path by adding the directory `/home/pathTest` to the end. Start by using the `echo` command to display the modified path variable.

echo $path

Next, use the set command and add `/home/pathTest` to the end of your path. Test this to see if it works by running the programs in /home/pathTest. They are `hello.sh` and `iq.sh`. If your path is correct you should be able to run these scripts by just typing their names no matter what directory you're in. If you have to supply the full path then something is wrong.

6.13 Adding new executables –The first big difference between the UNIX shell path and the DOS PATH is that DOS actually searches each directory in the PATH every time you type a command. This is pretty inefficient and slow since it may to require several disk accesses to find the command. The UNIX shells handle this in a much better way by traversing all of the directories in the shell path when the shell starts and building a table of commands and storing in the data area for the shell process. When you type a command the shell can quickly search this table and find the appropriate executable. This table is called the path hash table, since hashing is a method for optimizing this type of data for storage and searching.

The only downside to this is that the hash table is built when the shell starts, typically when you log in. So if you add any new commands to the directories in the shell path they won't be found since they weren't there when the shell started. This can be remedied by logging out and logging in again, or by using the `rehash` command which tells the shell to build the hash table again. Test this by doing the following

A. Ensure that you are in your home directory
B. Copy the directory /home/pathTest to your home directory by using `cp -r /home/pathTest .`
C. Modify your path and add this directory by typing `set path=($path ~/pathTest)`
D. Use the `cd` command to move into pathTest. List the files in the new directory using `ls`, you should see `hello.sh` and `iq.sh`.
E. Copy `hello.sh` to `hello2.sh`
F. Verify that it runs by providing the relative path by typing `./hello2.sh` Does it run?
G. Now test and see if it has been added to the files in your path. Try and run hello2.sh by typing `hello2.sh` Does it run? What can you determine about whether or not the hash table for the path automatically updates when you add new commands to a directory in your path?
H. Now update the hash table by typing `rehash` Try to run the file again by typing `hello2.sh` Does it run? Has the hash table for the path been updated?

6.14 Adding the current directory your path – Although it's highly discouraged for security reasons, you can add the current directory to the end of your path. You can actually add it to any part of the path, start middle or end, but again for security purposes if you feel you must add it you should add it to the end of the path. Searching the current directory works differently than other directories, since the current directory has to be searched every time you type a command instead of adding the commands to the path hash table. To add the current directory type the following:

```
set path=($path .)
```

Test this by doing the following:

A. Ensure that you are in your home directory
B. Copy the file `hello.sh` from the directory `/home/pathTest` to your home directory.
C. Type `hello.sh` to try and run the file. Does it run? Why or why not?

Environment Variables

The shell variables provide a valuable function for the shell, but they have one big weakness. They are only available to the instance of the shell that creates and uses them. If you change shells, the new shell won't have access to the other shell's data. And some shells track different data with the variables.

To handle this UNIX/Linux employs another set of variable called *environment variables*. These variables handle much of the same data as the shell variables, but they are available to any shell. The environment variables often even have the same names as the shell variables, however they use all capital letters for their names, while the shell variables are all lower case. For example, path would be the shell variable while PATH would be the environment variable.

6.15 Environment Variables - Just as with shell variables, a set of environment variables have been pre-set for all users. To see a list of the preset variables and their values, simply type: `setenv`, `printenv` or just `env`.

What do you see? How are these different than the shell variables?

6.16 Setting Environment Variables – The syntax for setting environment variables is one of those things that really varies depending on the shell being used.

In the Bourne shell you set the variable as you would any shell variable, but then use the export command to move it to an environment variable. For example:

```
set address='2600 North 20th'
export address
```

In the csh shell environment variables are set with the setenv command. The syntax is different than setting shell variables, as there is no equal sign. For example:

```
setenv ADDRESS '2600 North 20th'
```

To test this do the following:

A. Create a shell variable named email and give it the value joe@aol.com by typing

```
set email='joe@aol.com'
```

B. Verify that the shell variable has been set using the echo command:

```
echo $email
```

C. Check and see if an environment variable named EMAIL has been set by typing

```
echo $EMAIL
```

D. Create a environment variable named EMAIL and give it the value joe@aol.com by typing

```
setenv EMAIL 'joe@aol.com'
```

E. Check and see if the environment variable has been set by typing

```
echo $EMAIL
```

6.17 Synching Shell Environment Variables - In most UNIX/Linux releases changing some of the shell variables will also change the corresponding environment variable. This is critical for variables like path/PATH.

Compare the environment variable PATH with the shell variable path.

```
echo $PATH
echo $path
```

They should have the same directories listed, but with different delimiters. Make a change to the shell path by adding the /home directory to the end of the path. Check the shell and environment paths again. Have they both changed? What does this tell you about the synchronization between the two variables ?

Command Options Hierarchy

In the previous exercises you've seen that the shell looks in four different places for commands.

1. An absolute or relative path to the command.
2. Commands that are built into the shell
3. Aliases
4. In the hash table created from the directories in the shell path variable

But what if a command exists in more than one place? For example, if you make an alias for ls, and it's also in the hash table for the path, which ls will be executed. The following steps should help to determine which places the shell searches first for commands.

A. First we will test typing the path to a file, vs. finding a file in the hash table for the shell search path. You will create a file named ls, and test to see if it executes, or /bin/ls, the file in the hash table for the search path executes. Ensure that you are in your home directory

B. Create a file named ls and add the following echo command

```
cat > ls
echo 'this is my ls'
<ctrl-d>
```

C. Type either the absolute path to your ls or the relative path using one of the following:

```
./ls
~/ls
/home/yourUserName/ls
```

D. When you do this do you see the directory listing or 'this is my ls'?

E. Next we'll test an alias named ls against the /bin/ls in the hash table for the search path. Create an alias named ls and add the following echo command

```
alias ls echo 'this is my alias'
```

F. Type the following to run the ls command:

```
ls
```

G. When you do this do you see the directory listing or 'this is my alias'?

History

One of the real time saving features of csh/tcsh is the ability to store and repeat commands using something called history. It allows you to view/repeat commands that you've previously entered on the command line. You can even modify commands, which is helpful if you made a mistake typing, or it's a long command and you want to repeat it but change one thing.

Some of you may be saying why bother learning this, you can just use the arrow keys to scroll up and down through the commands you've typed before. The problem with relying on the arrow keys is that this is a function of terminal emulation program (putty in our case), and it may not always be available. In fact, it won't be available if you're sitting right at the UNIX computer, which is called the console. However, if you learn how to use the shell history it will always be available.

6.18 View your history - The csh shell keeps a list of commands that you type, which you can view with the `history` command. Type history and you should see a numbered list of the commands you previously entered, with the oldest commands at the top of the list.

The number of commands it remembers is set by the shell variable `$history`. If you have more commands then the oldest commands will be dropped as newer commands are entered. If you want to save more commands you can

always increase the value of the history variable. But don't make it too large as it takes up memory from the shell process, and there are practical limits.

6.19 Repeating commands – There are several ways to repeat commands you have already entered. To demonstrate this assume that you have the following commands in your history:

```
1  cd /etc
2  cat passwd
3  cd ~
4  date > myPasswd
5  cat  /etc/passwd  >> myPasswd
6  mkdir  tmpDir
7  chmod 700  tmpDir
8  mv  myPasswd  tmpDir
9  ls -al  tmpDir
```

If you type `!!` this tells the shell to repeat the last command in the history list, or in this case `ls -al  tmpDir`

Another method is to type `!`*n* where *n* is a number in the history list. For example, if you typed `!2` it would repeat command 2 from the history list or in this example `cat passwd`. You can also type `!-`*n*, which will go back *n* commands in the list. For example, `!-3` in this example would run item 7 in the history list `chmod 700  tmpDir` (I find it's easier to use `!`*n* than `!-`*n* as I can just what number to type instead of having to count the number of commands to subtract.)

If you type `!`*string* the shell will look for the most recent command that starts with the characters in the *string*. If you typed `!mv` in this example it would look for the most recent command that starts with `mv` and find item 8 in the history list `mv myPasswd  tmpDir` You can type as little as a single character, for example `!c` would look for the most recent command that starts with `c`. In this example it would find item 7 in the history list `chmod 700 tmpDir` You need to type enough characters in the string to get the command you want, so if you wanted to run item 5 in the history list again you would have to type at least `!ca`.

Using `!`*string* requires that the *string* be the first characters in the command. You can also `!?`*string*, which will search for the *string* anywhere in the command line. Using the example history list, `!?etc` would match item 5 in the history list `cat  /etc/passwd  >> myPasswd`

```
1  mkdir practiceDir
2  chmod 700  practiceDir
3  cd practiceDir
4  touch junk1 junk2 junk3
5  ls -al
6  date > junk4
7  cat  /etc/passwd  >> junk4
8  rm junk2
9  ls -al
10 cp ~/.login junk2
11 history
```

Assume that you are always typing command 12 (after the history command). What command will be executed when you type each of the following. (You can just write in the command number).

```
!9 _______________
!-4_______________
!! _______________
!h _______________
!c _______________
!ca_______________
!?login   _______________
```

How many ways are there to repeat the last command? Write down as many as you can think of?

6.20 Adding to command from history list – When you use the shell history the shell actually does a match and then expands the command before running the command line. This makes it possible to modify a command you entered previously by adding more before or after the expanded command.

For example, assume last command you entered was:

```
ls
```

If you type

```
!! -al | more
```

the shell will expand the `!!` to `ls` then add the `-al | more` on the end so the result would be

```
ls -al  | more
```

A. Type the following command `grep student /etc/passwd`
B. What would you type to repeat the previous command but add `| sort` to the end of it? Try it to ensure that it works.
C. What would you type to repeat the `grep` and `sort`, but redirect the output to the file junk.out? Try it to ensure that it works.

6.21 Retrieving part of a command – So far all of the methods for repeating commands from the history list have retrieved the entire command. Sometimes you just want to retrieve part of a command, such as either just the command or just the last argument. Most of the positional retrieval methods require specifying which command you want to repeat using one of the methods shown above; followed a second part to specify which portion of the command you want to retrieve.

`:^`	The first word (the ^ is the same as in regular expressions)
`:0`	The first word, same as :^
`:1`	The second word
`:n`	The nth + 1 word (since counting starts at 0)
`:$`	The last word
`:*`	All the words except the first (the command)
`:x-y`	The xth word through the yth word
`:-x`	All the words from 0 to x
`:x*`	All the words from x to the last
`:x-`	Just like :x*, but leaves off the last word

To demonstrate this, assume that you have the following in your history list:

```
1  mkdir practiceDir
2  chmod 700  practiceDir
3  cd practiceDir
4  touch junk1 junk2 junk3
5  ls -al
6  date > junk4
7  cat  /etc/passwd  >> junk4
8  rm junk2
9  ls -al
10 cp ~/.login junk2
11 history
```

Example 1
If you enter

```
!6:0
```

The shell will expand `!6` to date `> junk 4` but the `:0` says to only retrieve the 1st word, so it will only expand this to

```
date
```

If you enter

```
!6:0  >  junk 5
```

The shell will do the same expansion, then add `> junk 5` to the end, resulting in

```
date > junk5
```

Example 2
If you enter

```
cat !c:1
```

The shell will match `!c` to command 10 in the history list which is:

```
cp ~/.login junk2
```

but the :1 says to only retrieve the 2nd word resulting which is

```
~/.login
```

this is placed on the command line after the `cat` command resulting in

```
cat ~/.login
```

Practice
Practice retrieving a portion of a command in the history list. Assume that you have the following commands in your history list.

```
1  mkdir practiceDir
2  chmod 700  practiceDir
3  cd practiceDir
4  touch junk1 junk2 junk3
5  ls -al
6  date > junk4
7  cat  /etc/passwd  >> junk4
8  rm junk2
9  ls -al
10 cp ~/.login junk2
11 history
```

For each of the following, write down what the shell would execute after the history substitution(s) occur:

```
!9:0       _______________
cat !8:1   _______________
!9:0       _______________
rm !mk:$   _______________
!9:0       _______________
sort !7:$  _______________
!9:0       _______________
ls -al !t:2 _______________
```

6.22 Retrieving part of the last command – To be perfectly honest I rarely use the methods described in the previous section for retrieving part of a command. This is mainly because I don't like having to count or figure out exactly which part of the command I want to use. However, there are some history retrieval commands that I do use quite often. These only work to retrieve from the last command you typed, they won't work on older commands.

!$	The last word
!*	All the words (except the first)

For example, if you type:

```
touch junk7
```

Then type

```
chmod 700 !$
```

The shell will expand the `!$` to match `junk7` and the result will be

```
chmod 700 junk7
```

In this example using !$ doesn't save much typing, but it really does come in handy when the argument includes longer file and path names.

History Substitution

6.23 Another helpful feature of the shell history is the ability to run a previous commands, but change or substitute portions. The simplest form is `^old^new` which repeats the last command, but changes the string "*old*" to the string "*new*". For example, say the you typed:

```
cd /usr/lbi
```

The cd commands returns an error message and you realize that you misspelled lib. This can be remedied by typing

```
^bi^ib
```

The shell retrieves your previous command, then substitutes `ib` for `bi` which results in

```
cd /usr/lib
```

This method is good for fixing simple typos, but it has some limitations. The first is that it only works with the last command you entered. The second is that it only changes the first occurrence, there's no way to make it work in global mode. However, it doesn't expand filename globbing wildcards, so you can change characters like * or ? without having to worry about protecting them during the substitution.

The second form of substitution requires that you use one of the history command specifiers such as `!string`, followed by `:s/old/new/` For example:

```
!ch:s/tony/bob
```

Will find the last command that starts with `ch` and then replace the first occurrence of `tony` to `bob`. To change all occurrences, add a `g` before the `s/old/new`. (It would be nice if it were at the end, to stay consistent with the `sed s` command, but it goes at the start.) For example:

```
!ch:gs/tony/bob
```

This is also one of those nit-picky details that some releases of Linux don't support.

Practice

Practice retrieving a portion of a command in the history list. Assume that you have the following commands in your history list.

```
1  mkdir practiceDir
2  chmod 700  practiceDir
3  cd practiceDir
4  touch junk1 junk2 junk3
5  ls -al
6  date > junk4
7  cat  /etc/passwd  >> junk4
8  rm junk2
9  ls -al
10 cp ~/.login junk2
11 ls junk5
```

For each of the following, write down what the shell would execute after the history substitution(s) occur:

```
^5^2       _______________
^ls^ls -al _______________
!c:s/2/3/  _______________
!2:s/7/5   _______________
```

History Lists and Files

6.24 As explained previously, the history list is maintained in the shell's memory, and the number of commands saved is controlled by the `$history` shell variable. You can change the number of commands saved by changing the value of `$history`. For example

```
set history=100
```

sets the number of commands to be saved to 100. If you type more than 100 commands the older ones will be dropped as new ones are added.

Since the history commands are saved in memory they will be lost when you log out. However, you can instruct the shell to write them to a file, which will be read back into memory when you log in again. The name of this file is `.history`, and the number of commands you want to save in it is controlled by the `$savehist` shell variable. The value of `$savehist` should be the same as `$history` or lower. (For those engineers out there [and you know who you are] I know that some of you are dying to find out what happens if you make `$savehist` larger than `$history`, so go ahead and try it and see what you get.) You don't have to create the `.history` file, it's created for you automatically when you set `$savehist` and then exit the shell.

To start saving commands in your `.history` file do the following:

A. Before setting `$savehist`, verify that commands are not being saved between sessions of running your shell. Log out and then log back in. Verify that your command history has been restarted by typing:

```
history
```

B. Ensure that `$history` has been set. If you're not sure type:

```
echo $history
```

C. Tell the shell to save commands when you logout

```
set savehist=100
```

D. Logout and log back in. Check the contents of your command history by typing:

```
history
```

It should have several commands in it already, and they should be the commands you entered during your last shell session.

E. You can also inspect the contents of `.history` by typing

```
cat ~/.history
```

Remember that `$history` and `$savehist` are shell variables; so when you exit the shell they and their respective values go away. If you want to make them permanent add them to your shell startup file `.cshrc`, or `.login`.

Customizing the Shell - Other Shell Variables

6.25 Practice setting your prompt using the following prompt variables. Each time after you change your prompt, change directories to at least /, /usr/lib/ and back to your home directory, and to a sub-directory under your home directory named junk. (You will need to create junk if it does not exist). Write down what your prompt displays at each of these directories.

	`cd /`	`cd /usr/lib`	`cd or cd~`	`cd junk`
`set prompt=%/`	.....	.....	.....	.....
`set prompt=%~`	.....	.....	.....	.....
`set prompt=%C2`	.....	.....	.....	.....

Try two or the three of the following prompt variables, combinations of these variables, or combining these variables with strings. When you find a prompt you like, add it to your .cshrc or .login.

`%/`	Current working directory
`%~`	cwd. If it starts with $HOME, that part is replaced by a ~. In addition if a directory name prefix matches a user's home directory, that part of the directory will be substituted with ~user. NOTE: The ~user substitution will only happen if the shell has performed a ~ expansion for that user name in this session.
`%c or %.`	Trailing component of cwd, may be followed by by a digit to get more than one component, if it starts with $HOME, that part is replaced with a ~.
`%C`	Trailing component of cwd, may be followed by a digit to get more than one component, no ~ substitution
`%M`	The full machine hostname
`%m`	The hostname up to the first "."
`%S (%s)`	Start (stop) standout mode
`%B (%b)`	Start (stop) boldfacing mode.
`%U (%u)`	Start (stop) underline mode.
`%t or %@`	Current time of day, in 12-hour, am/pm format
`%T`	Current time of day, in 24-hour format.
`%p`	Current time in 12-hour format, am/pm format with seconds
`%P`	Current time in 24-hour format, with seconds
`%%`	A single %.
`%n`	The user name, contents of $user
`%d`	The weekday in <Day> format
`%D`	The day in dd format.
`%w`	The month in <Mon> format
`%W`	The month in mm format
`%y`	The year in yy format
`%l`	The line (tty) the user is logged on
`%L`	clear from prompt to end of display or end of line
`%#`	A `#' if tcsh is run as a root shell a `>' if not
`%?`	return code of the last command executed just before the prompt
`%R`	In prompt3 this is the corrected string; in prompt2 it is the status of the parser

6.26 **Redirection and noclobber -** In it's default state csh assumes that you know what you are doing, and will let you overwrite any file you have permission to write to. Some of us have a hard time remembering where we parked our cars, much less if we already have a file with a certain name, so luckily there's way to tell csh to help.

Create a file named junk, then try to overwrite it by typing:

```
cat /etc/passwd > junk
```

Notice that the shell just overwrites what was stored in junk without asking. You can prevent this by setting the shell variable **noclobber**. This is done by typing:

```
set noclobber
```

Test to see if it works by typing:

```
cat .login > junk
```

What message appears?

You can force the shell to temporarily ignore `$noclobber` and overwrite by using `>!` instead of just `>`. Test this by typing:

```
cat .login >! junk
```

inspect junk, was the file overwritten or protected?

If you want to permanently turn `$noclobber` off type:

```
unset noclobber
```

Shell Startup Files and .login

6.27 There are actually several startup files that get processed when you login to the system. Two of them, `.cshrc` and `.login` belong to you, and you can control what they do. The third is controlled by the system administrator, and is used to set things for every user that logs on to the system. In this exercise, you will check the contents of the system wide login file, and then check to see the order that your login files are processed.

The name of the system wide login file on this particular system is: `/etc/csh.login` Check the contents of this file. (Actually there are other files that get processed on this release of Linux, I just can't find them …) This is why you have aliases set, even though you have not created them, and may not even have a `.login` or `.cshrc`.

After the system wide login is processed, your startup files are processed. The `.login` file is processed when you login, and the `.cshrc` is processed each time you start the csh (or tsch) process. There's really no practical difference in which file to use for startup. However, the `.login` will be processed no matter which shell you use, while `.cshrc` will be processed every time you start a new copy of the shell. So if you are using different shells, put the common commands in `.login`, and the shell specific commands in `.cshrc` (or `.profile` for the bourne and korn shells).

Test to see the order that `.login` and `.cshrc` are processed by changing your prompt in both files, then logging in again. For example, in `.login`, set your prompt to "`login-last>` ", and in `.cshrc` set your prompt to "`cshrc-last>` ". When you login, one of the files will be processed first and the prompt will be set. Then, the second file will be processed and the prompt reset … when you see the prompt, you will know which file was processed last.

Job Processing

6.28 Most UNIX/Linux shells provide a function called job control, which allows you to run more than one process at a time. I know this causes most of you to yawn and think "so what", because you're used to a windows based GUI that allows you to do many things at the same time. But in the command line world it's kind of a big deal.

The shell usually executes what you type on the command line, and finishes it before displaying the prompt again. If you have a command that will take some extra time to execute you can tell the shell to place the command in the "background', where it will continue to execute, but return and display the prompt so you can continue to work. This is usually done for things like compiling large programs, or executing programs that are going to do a lot of calculating and not need any input from the user.

In UNIX jobs can either be running in the foreground, be running in the background, or they can be stopped. By default the shell starts a job for each command you type, and runs it until it's done. To run a job in the background you end the command with an ampersand "`&`". For example:

```
bigCalc.sh &
```

This starts the shellscript `bigCalc.sh` and places it in the background. You will see a message telling you that the job has been placed in the background, as well as a job number. You can always see the status of all your jobs by using the `jobs` command. You can move a command from the background back to the foreground by using the `fg` command. By default it takes the last command moved to the background, but if you have more than one job in the background you can specify which one to move to the foreground by adding the job number as the argument to the `fg` command. You can also move a command to the foreground by typing `%`*`n`* where *`n`* is the job number.

If you have a job running in the foreground and you type `<ctrl-z>` the job will be stopped. It's actually paused, but they call it stopped. If you type `bg` the job will then be moved to the background.

If you want to really stop a process or job, not just pause it, you can use the `kill` command. Before you do this you need to find the process-id, called the pid for short, for the job. This is done using the `ps` command. Once you know the pid you can terminate the process using

```
kill -9 pid
```

Practice

Practice putting jobs in the background, and then moving them back to the foreground by doing the following:

A. Start a simple command that requires more keyboard input, but place it in the background:

```
cat > deleteme &
```

B. Even though the `cat` command has not finished, the shell will display the prompt. Check to see that a job has been created for the `cat` command, and what it's number is by typing

```
jobs
```

C. Move the `cat` command job back to the foreground by typing either

```
fg
```

or

```
%1
```

The shell will tell you that it's resuming the `cat` command, and wait for more input. You can type some text if you wish, or just type `<ctrl-d>` to signal the end of the keyboard input and end the `cat` command.

D. Practice killing a job by starting the `cat` command and placing it in the background again

```
cat > deleteme &
```

E. Find the pid for the `cat` command using the `ps` command

```
ps
```

F. Use the `kill` command to stop the `cat` command

```
kill -9 pid
```

Review

1. List some of the functions that the shell provides for you.
2. When you type a command, the shell will look several places for a command with that name. Name these places.
3. In what order will the shell search these places?
4. What does the *rehash* command do for you?
5. When would you use *rehash*?
6. Does everyone on the system have to run the same shell?
7. Why would you want to change your shell?
8. How do you change your shell on a temporary basis? Permanently?
9. Which shells can you change to? Do all UNIX systems have all shells available?
10. How do you find out the built-in commands for a given shell?
11. How can you customize the shell? Can you make these changes so they are always available for you?
12. What are the names of your startup files?
13. What is a shell variable? Why do they exist?
14. Name some shell variables and what they hold?
15. What is an environment variable? How are they different from shell vars?
16. Name some variables that exist in both the shell and the environment.
17. List the steps in the login process? Can the root account set up things to effect all of users?
18. Which will win out, .login or .cshrc?
19. How do you list your currently defined aliases?
20. How do you create an alias?
21. How do you delete an alias?
22. Why would you want to create an alias?
23. How do you turn your history on?
24. How do you see the list of commands that you have already executed?
25. How do you repeat your last command? Name two ways.
26. How do you repeat the last command that starts with the letter `c`?

Description: Shell programming is like writing batch files for DOS, but much more powerful. With the power of UNIX utilities you can write simple macros, or entire programs.

Objectives: At the end of this section the student will be able to:

1. Write and execute simple shell scripts.
2. Write shell scripts that can do simple calculations and print messages.
3. Access UNIX commands from shell scripts.

Reading:
Study Guide: To pass the test on this topic, you will need to know the following terms and concepts:
The objective of the following exercises is to introduce you to shell programming. At the end of these exercises you will be able to

1. Introduction - Creating and running Bourne shell programs
- execute permissions
- `#!bin/sh`
- Comments
- Executing a shell script
- Problems running shell scripts

2. Running shell commands

3. Writing
- the *echo* command
- -n to prevent newlines.
- -e to expand
- Quotes (again)
- special characters
- ascii art

4. User Defined Variables
- data type
- defining and assigning values
- using variables

5. Predefined variables
- shell
- environment
- accessing

6. Numeric calculations
- expr
- bc

7. Reading user input

8. Debugging shell scripts

9. The if statement
- **What is truth, [test command] and 0 or 1**
- **Else**
- **Else if**

10. Loops
- **Truth again**
- **While and until**
- **Do done**
- **Counting loops**

Conditional loops
Checking user input

11. Command line arguments

12. Applying

Shell Programming Basics – Specifying shell interpreter, comments, and debugging

Specifying shell interpreter

Shell scripts are programs that you write using a combination of the same commands, pipes, quotes and wildcards that you have been learning; plus other programming commands such as "`if`" for making decisions and "`while`" for building loops.

In the previous section you learned about some of the UNIX shells. Another big difference between the various shells is the syntax they use for the programming commands, the `if` commands loops, etc. Most shell scripts are written using the bourne shell (sh, bash) syntax, so this is the shell programming syntax you will learn here. Just be aware that you can write shell scripts for other shells, but some of the syntax will be completely different.

All of the commands in your shell script will be interpreted by the shell each time it is run. But which shell will do the interpreting? If you don't specify which shell to use, your current default shell will do the interpreting. In our case, since you are using the tcsh shell, it will do the interpreting. This will cause problems if you use commands that use a different syntax. In fact, this is exactly the case you will face, as you will write your shell scripts for the bourne shell, and want that shell to run the script. You can specify a different shell by typing the shell name before the name of the shell script. The following example specifies the bourne shell should be used to run the script:

```
/bin/sh  script.sh
```

However, this requires that the user remembers to type `/bin/sh` every time they want to run the script. A better way is to include `#!/bin/sh` as the first line in your shell scripts. It specifies which shell should be used to interpret the script, but since it's built into the file it happens automatically. Just make sure that this is the first line in the file, and there are no spaces before the `#!/bin/sh`

Comments

Comments are little notes that programmers leave in their programs. They are ignored by the interpreter, and typically are used to provide little reminders about what the commands are doing. This is great if you are doing anything halfway tricky, but comments are always useful when you come back to look at something you or someone else has written. They can also be used for keeping track of revision information, dates that changes are made, again as reminders to yourself or anyone else that may need to understand what the script is doing. I also use comments to add a couple of jokes to my scripts. When I come back and edit a script after a couple of years I'll read the joke again, and my memory is bad enough that I'll laugh all over again.

In a shell script, comments are designated by the "`#`" character. Anything appearing on a line after this character will be ignored. The "`#`" can be at the start of a line, or anywhere in a line. If it's at the start, then the entire line will be ignored. If it appears later in a line, anything from that point on in the line will be ignored.

You may be asking yourself about the `#!/bin/sh` that I said to use as the first line in your shell script. It starts with a "`#`", so it should be a comment and be ignored, shouldn't it? Well … this is the one exception, and the reason that this must be the first line in the file. If you have a blank line before this, then it will be ignored.

For comments that span multiple lines there are two options. The first is to add the "`#`" at the start of each line. The second is to use something known as "whereis" or "here" redirection. This starts with the characters << followed by a text string. The shell ignores everything from this point until it encounters the text string again. For example:

```
<< changeLog
    9/12 Added new banner
    9/23 Removed end of summer sale items
    10/20 Added Halloween specials
changeLog
```

There's nothing special about the text string you use as the identifier; you can use any text you wish. Just make sure and avoid using text that you want to include in your comments.

Debugging
I make lots of mistakes when I program. The mistakes may be logic errors, such as thinking that 2+2=6; or they may be syntax errors such as using the wrong quotes or typing "daet" instead of "date". With all of the weird symbols used in shell programming it's easy to think that something is *should* be happening but have the shell do something completely different. To help debug shell scripts, there are two shell options that can be set to instruct the shell to show you the results of every wildcard expansion, and to also tell you exactly it's going to do with complex sets of commands. These debugging options are turned on by adding set –xv to the shell script. This is usually done right after the first line as shown:

```
#!/bin/sh
set -xv
```

While these options are useful for debugging, they really cause a lot of output to wade through. To turn the options off you can either remove the lines from the file or comment them out. Commenting them out is the method that is usually used, as it's easy to turn the options back on again if you need them. Simply add or remove the "#" to comment or uncomment the options.

HOW TO RUN A UNIX PROGRAM OR SCRIPT

Telling the UNIX Operating System that you want to run a program is pretty simple; you just type the name of the program file or script file. For example, the `ls` command is really in a file named `/bin/ls`. To run it, you just type `ls`. If you make a shell script and save it in a file named `carlos` you would run it by typing `carlos`

However, there are two things that may make this slightly more complicated.

The first is that the shell has to find the file you want it to run. If you remember from the section where you learned about the shell, the shell doesn't look everywhere for the file. It builds a table of all the commands in your `$path` when you login, but it won't find any new files you create unless you tell the shell to rebuild the table using the `rehash` command. And it won't find any files unless they are in one of the directories in your $`path`. And it won't look in the directory you're working in unless you add "." to your $`path`. So if you're working in a directory that's not in your path, then you will have to tell the shell to look for the file in this directory by adding a `./` to the start of the file. For example, if you want to run a program named `soundCheck` in the current directory, you would type `./soundCheck`.

If you try to run a shell script that you just created and you get the error "`Command Not Found`", then you've either forgotten to add the `./` or you mistyped the file name.

The second complication is permissions. It's very probable that your `umask` is set so that you will not have execute permission for any file you create. So when you try to run a file, you will get the error message "`Permission Denied`". To correct this, use the `chmod` command and set the permissions to 700, or 744, or anything that gives you execute permissions.

7.1 The first program you should write in any language is "hello world". Use `vi` to enter the following lines, save the file, change permissions so you can execute it, then run it.

```
#!/bin/sh
echo "Hello World"
```

7.2 Each program you write should include comments that describe the program. At a minimum all of your programs for this class must include your name and the assignment number in the comments. Add these comments to the program you just created.

7.3 Using the echo command, write a box containing your name and assignment #. You can use whatever characters you wish to create the box. Remember, if you want to use a character that has special meaning to the shell you will need to protect or escape it using quotes or the "\" character.

```
+--------------------------------+
|  Name:  Joe Unix               |
|  Assignment:  x.xx             |
+--------------------------------+
```

7.4 Use the `date` command to add the current date and time to the previous exercise.

7.5 Use the `echo` command and the `/  \  _` characters to print a triangle.

7.6 Use the `echo` command to print a hawk or eagle. (Need help? Search the Internet for "ASCII Art".)

The `echo` command works a little differently in the Bourne shell (sh and bash) than it does in the csh (tcsh) shell. If you want `echo` to expand `\t` to a tab (or any of the characters in the following table), then you must use the `-e` option for echo.

\a	alert (bell)
\b	Backspace
\c	suppress trailing newline
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	Backslash
\nnn	the character whose ASCII code is nnn (octal)

7.7 Write a program that prints a table of your class schedule for this quarter. Use tabs to make the columns line up. (Remember that `\t` is the tab character, and that you must use `echo -e` in the Bourne shell to have it expand the tabs.)

7.8 Write a program that prints out information from different shell variables. Print two columns, one showing the variable name and the other showing the current value. Use tabs to separate the two columns. You must print at least the current working directory and the user name of the person running the program. Change the permissions on the program so that your classmates can run the program (at least 755). Have one of your classmates run the program to test it.

Variables – remembering information to use later.

There are many variables that the shell sets for you, but you can also create your own. Creating variables is done by simply assigning one a value; you don't have to declare them or assign a type. For example, to create a variable called `car` and assign it the value `ford` you would type:

```
car=ford
```

To read the value from the variable, use the variable name with a $ on the front. For example:

```
echo "my car is a $car"
```

You need to be careful about ensuring that there are **no spaces on either side of the equal sign**. You can whatever you want as a variable name, as long as you only alphanumeric characters and start with a letter. However, it's suggested you use a name that will help you remember what it is the variable is storing.

7.9 Modify the program that prints your class schedule for this quarter. Use variables to hold the class names and times. For example:

```
#!/bin/sh
```

```
class1=”CS223 Unix”
time1=”8AM – 9AM”
echo -e  “$time1 \t$class1”
```

Reading user input.

Reading is done with the `read` command. The syntax for a simple read command is: `read varName` If you want the user to input something, you also must tell them what to enter. This is text generically called a *prompt*, and is done like any other printing using the `echo` command.

When the shell sees the `read` command, it takes whatever is typed and puts it into the specified variable. You can think of this as a 2-for-1 deal, it creates the variable and then stores whatever the user types into it. (The variable name can be whatever you wish, but it must be only letters or numbers, and start with a letter. It also helps if you use mnemonic names for your variables, so you have some clue regarding what you are storing in it.) For example:

```
#!/bin/sh
# Put your comments here

echo -n  “Please enter your name: ”
read name

echo “Hello  $name”
```

It’s possible to read multiple pieces of data into multiple variables. If there is more than one word in the input, each word can be assigned to a different variable, with spaces being used as the delimiter. Any words left over are assigned to the last named variable. For example:

```
echo –n "Please enter your first and last names"
read name1 name2
echo "Welcome to CBC $name2 $name1"
```

If there are more input words than variables then any words left over are assigned to the last named variable. If you ran the code above, and entered `Mr. Tony Sako` then `$name1` would be set to `“Mr.”` and `$name2` would be set to `“Tony Sako”`

If the user doesn’t enter enough information, then the remaining variables will be assigned the value “null”. Null is an ASCII character that represents that nothing is being stored.

For these reasons, it’s usually suggested that you design your programs to prompt for and read only one thing at a time.

7.10 Enter the following program that prompts the user for their first and last names. You should run it several times, giving it two names, one name, three names, all separated by spaces, all separated by commas, until you are comfortable with the way the shell handles reading variables

```
#!/bin/sh
# Put your comments here

echo -n  “Please enter first and last names: ”
read name1 name2
echo "Welcome to CBC $name2 $name1"
```

7.11 Write a program that prompts the user for their name then prints out a customized welcome message. The message must include the name, the current date and time, and a joke from the fortune file. For example, your program’s output may look like the following:

```
#########################################################################
#                                                                       #
#           Welcome Caterina                                            #
#                                                                       #
#           December 12, 2062                                           #
#           7:12 AM                                                     #
```

```
#                                                                              #
################################################################################

 Your fortune for today is:
 Trust Me. I'm the Doctor
```

NOTE: your Linux system will have the `date` program, but it may not have the `fortune` program. You can test this by typing `fortune`. If the program is on your system you will see a random fortune. If not, you will get the error message `command not found`.

Commands – using other UNIX commands, pipes, quotes, etc.

One of the great things about shell programming is that you can run any of the commands you would normally type on the command line. You can also use wildcards for filename expansion, pipes, redirects; anything you can type on the command line. For example, the following shell script would change to your home directory and then run the `ls` command:

```
#!/bin/sh

cd ~
ls - al
```

7.12 Write a program which does the following:

a. Prompts the user for a directory name
b. Creates the directory
c. Copies your .login and .cshrc into the new subdirectory

7.13 Write a program which does the following:

a. Prompts the user for a file name
b. Finds all of the lines in the file /etc/passwd which contain the phrase "student"
c. Takes these lines and removes all the fields except the first and the last. Remember that the fields are delimited by the ":" character. You should end up with the user name and default shell.
d. Sorts the remaining lines alphabetically
e. Places the sorted lines in the file that the user specified

Building shell scripts with the `script` command

While we can write shell scripts to accomplish almost any programming task, they are usually used for administrative tasks. If we want to perform complex calculations or store information in a database, we'd be a lot better off using a programming language like C or C++. But if we want to automate an administrative task, such as processing log files or performing the weekly backup, then shell scripts can be much easier.

There are two ways to build a shell script, you can use the editor and write it out, test it out, then repeat the editing and testing until the script works the way you want it to. Or … you can use the `script` command, which basically turns on a recorder which records everything you type. This is an easy way to build and test the script, you just type the commands you want to run. The only problem I have with the `script` command is that it records everything, typing errors, and all of the output that comes to the screen. So when you're done recording, you have to do some editing to clean out all of the extraneous stuff.

Using the `script` command to record your commands is pretty simple. Type `script`, followed by the name of the file you want the recording to be stored in. For example: `script backup.sh` will start the `script` command, and everything that appears on the screen will also be stored in `backup.sh`. If you don't supply a filename, then the default name `typescript` will be used.

If the script command starts correctly it will display a message similar to the following:

```
Script started, file is backup.sh
```

When you're done recording, type `exit` or hit `<ctrl-D>`. When the script command stops it displays a message similar to the following:

```
Script done, file is typescript
```

If you open the file created by the `script` command you will see that it's recorded **EVERYTHING**! Every time you hit the `<enter>` key it's added to the file as `^M`. If you are typing a command and make a mistake and hit the `<delete>` key, it will be recorded as `^H^[[K`. This is shown in the following example:

```
1 Script started on Sun Sep 11 14:21:06 2011
2 next> echo "hellloo^H^[[K^H^[[K^H^[[K^H^[[K^H^[[Ko world"^M^M
3 hello world^M
4 next> exit^M^M
5
6 Script done on Sun Sep 11 14:21:29 2011
```

To make this a usable script, all of the output and all of the extra characters must be edited out.

7.14 Using the script command, record a script that does the following:

a. Write "Hello World" to the screen
b. Write the current date to the screen (use the `date` command)
c. Uses `grep` to print you account information from the file `/etc/passwd`

d. When you're done recording the script type `exit`
e. Use the editor to clean up the script
f. Test out the script to ensure that it works

Calculations – Integer math with `expr` and floating point math with `bc`.

Doing calculations can be done using two UNIX commands, `expr` or `bc`. The difference between the 2 is that `expr` only does integer math, that is you will only get whole number results with no decimal portion. The `bc` command is used if you need real numbers with a decimal portion.

The syntax for using `expr` is:

```
expr 2 + 2
```

This works for addition and subtraction. If you want to multiply or divide you would use "`*`" or "`\`" as the operator. However, since these characters have special meaning to the shell they must be protected. So to multiply you would use:

```
expr $a \* $b
```

The output from `expr` goes to standard out (typically the monitor). If you want to store the result of your calculation in a variable for later use, then you will have to make use of the back ticks. For example:

```
c=`expr $a \* $b`
```

Using `bc` is a little different. When you run `bc`, you start it just by typing `bc`. Then you enter the commands you want it to run. So to run `bc` from a shell script you will have to (1) start it then (2) `echo` in the commands you want it to run. That is, you will use `echo` to print the commands, but instead of sending them to stdout, you will pipe them into the `bc` command. For example, to add two numbers together:

```
echo 12.5 + 22.7 | bc
```

The rest of the usage for bc is just like `echo`. You will have to protect the "`*`" or "`/`" operators, and if you want to store the result of your calculation in a variable for later use, then you will have to make use of the back ticks. For example:

```
c=` echo 2.5 \* 5.3 | bc`
```

7.15 Write a shell script which prompts the user for three integers, then multiplies them together and prints out the result.

7.16 Write a shell script which prompts the user for a three floating point numbers, then multiplies them together and prints out the result.

7.17 Write a shell script which calculates gas mileage. The script should do the following:

a. prompts the user and reads the number of miles driven
b. prompts the user and reads the number of gallons of gas used
c. uses `bc` to divide the miles driven by the gallons of gas to calculate the miles per gallon
d. prints out the result.

7.18 Write a car rental bill shell script. (You can either write the entire script, or modify the script tickets2.sh). At UNIX car rentals you can rents a car for $25 a day, plus $.05 for each mile you drive, plus $3.99/day insurance, plus 8% tax. The script should do the following

a. Read the users name, number of days the car was rented, and the number of miles driven
b. Calculate the daily, mileage, insurance, and tax payments, and the total payment
c. Print out a custom bill that includes the name, individual charges, and the total payment. (Try and format this so that it looks like a real car rental bill).

Making Decisions – the `if` statement and Boolean tests

The `if` statement is used for making decisions, and deciding to run a block of code or not. The basic syntax for the `if` statement is:

```
#!/bin/sh
# Basic if statement syntax.

echo –n "Please enter a number: "
read n1 junk

if [ $n1 -ge 0 ]
then
      echo "$n1 is a positive number"
fi
```

The code between the `then` and `fi` statements will only execute if the `if` statement's Boolean test is true. If the result of the test is false the code will be skipped. And yes; it does use `fi` to end the code.

The Boolean expression in the test can be used to compare two numbers (see below for comparing floating point numbers), two strings, or to check the status of files or strings. Multiple Boolean expressions can be combined using the AND or OR operators. For more details on these tests and operators read the man pages for the `test` command.

Integer Operators:

Expression	TRUE if
int1 –eq int2	int1 is equal to int2
int1 –ne int2	int1 is not equal to int2
int1 –lt int2	int1 is less than to int2
int1 –le int2	int1 is less than or equal to int2
int1 –gt int2	int1 is greater than to int2
int1 –ge int2	int1 is greater than or equal to int2

File Operations:

Expression	TRUE if
-e File1	File1 exists
-s File1	File1 exists and is not empty
-f File1	File1 exists and is not a directory
-d Directory1	Directory1 exists
-x File1	File1 has execute permission
-w File1	File1 has write permission
-r File1	File1 has read permission
File1 –ef File2	File1 and File2 are the same file (they have the same inode numbers)
File1 –nt File2	File1 is newer than File2 (newer modification date, not creation date)
File1 –ot File2	File1 is older than File2 (older modification date, not creation date)

String Operators:

Expression	TRUE if
String1=String2	The strings are identical
String1 != String2	The strings are not identical
-z String1	String1 has zero length (null)
-n String1	String1 has nonzero length
String1	String1 has nonzero length (same as –n)
-z String1	String1 is zero length (null)

Expression	TRUE if
(Expression)	Expression is TRUE
!Expression	Expression is FALSE
Expression1 –a Expression2	Expression1 AND Expression2 are TRUE
Expression1 –o Expression2	Expression1 OR Expression2 is TRUE

The `else` statement can be added to an if statement to have code that will execute if the expression in the `if` test is FALSE.

```
#!/bin/sh
# Basic if statement syntax.

echo –n "Please enter a number: "
read n1 junk

if [ $n1 -ge 0 ]
then
      echo "$n1 is a positive number"
else
      echo "$n1 is a negative number"

fi
```

The `elif` statement can be used to add additional tests and blocks of code. When this code is executed each test is checked in order. If a test is TRUE, then the code associated with that test is executed, but no other tests will be checked. If a test is FALSE the next test is checked. If an `else` statement is included, it's code will only be executed if all the other tests are FALSE.

```
#!/bin/sh
#set -xv
```

```
# This script demos making decisions using the if statement
# and elif which is "else if" in the Bourne shell

clear
echo -n "Enter your age: "
read age junk

if [ $age -gt 100  ]
then
  echo -e "Your age is exceptional "
elif [ $age –gt 65  ]
then
  echo -e "You can retire now if you want"
elif [ $age –gt 21  ]
then
  echo -e "Get up! It’s time to go to work!"
elif [ $age –gt 12  ]
then
  echo -e "You are a lucky teenager. Make the most of it."
elif [ $age –gt 5  ]
then
  echo -e "Will you help me with my phone."
fi
else
  echo-e Who wants ice cream?”
fi
```

As noted above the numeric test expressions built using `-gt` etc will only compare integers; they will exit with an error if you try and compare floating point numbers. To compare floating point numbers we have to use our old friend `bc` again. The next two scripts demonstrate the long way, which is easier to understand, followed by a shorter but more convoluted method.

```
#!/bin/sh
#set -xv
clear

echo -n "Enter num1: "
read num1 junk

echo -n "Enter num2: "
read num2 junk

# Save the 0 (FALSE) or 1 (TRUE) result from bc and test this in an
# if statement

result=`echo "$num1 > $num2" | bc`

if [ $result -eq 1 ]
then
  echo -e "\n\n$num1 is greater than $num2"
else
  echo -e "\n\n$num1 is not greater than $num2"
fi
```

The previous code can be made more compact and by moving the `bc` command inside the `if` test. This saves a little typing, but can be harder to understand if you haven't seen the separate steps

```
result=`echo "$num1 > $num2" | bc`

if [ `echo "$num1 > $num2" | bc` -eq 1  ]
then
```

```
  echo -e "\n\n$num1 is greater $num2"
else
  echo -e "\n\n$num1 is not greater than $num2"
fi
```

7.19 Modify the IQ calculator script.

a. Read the users age and shoe size
b. If the shoe size is larger than 15 print a message saying that sizes must be in US units (between 1and 15) not European units.
c. If the shoe size is 15 or less calculate the IQ by multiplying the age by the shoe size, then print the result

7.20 Modify the car rental bill shell script. You can either write the entire script, or modify the script tickets2.sh. At UNIX car rentals you can rents a car for $25 a day, plus $.05 for each mile you drive over 100 miles, plus $3.99/day insurance, plus 8% tax. The script should do the following

a. Read the users name, number of days the car was rented, and the number of miles driven
b. Calculate the daily and, insurance payments
c. Calculate the mileage payment by checking to see if the mileage is greater than 100. If it is 100 or less there is no charge. If it is greater than 100 then subtract 100 from the mileage and multiply by 0.05
d. Calculate the tax and the total payment
e. Print out a custom bill that includes the name, individual charges, and the total payment. (Try and format this so that it looks like a real car rental bill).

Repeating Code With Counting Loops – the `for` statement

The `for` statement is used to repeat code a specified number of times. The basic syntax for a `for` loop is:

```
#!/bin/sh
# Basic if statement syntax.
clear

for ( i=10; i<=20; i++ )
do
      echo "i is $i"
done
```

The code in the `for` statement that controls how many times the loop will repeat is the `( i=10; i<=20; i++ )` This code does three things:

1. The `i=10;` sets what is called an initial condition. It can be thought of as the loop's starting point. The variable used, in this case i, is referred to as the loop counter because it keeps track of how many times the loop has repeated.
2. The `i<=20;` is a Boolean test that determines how many times the loop will repeat or sets the ending point. Every time the `for` statement is executed this test is performed. If the results of the Boolean expression are TRUE then the loop code, all the code between the `do` and `done` statements, will be executed. If the results of the Boolean Expression are FALSE then the loop is completed and the script will continue with the commands immediately following the `done` statement. If the `done` statement is the last line in the script, the script will exit.
3. The `i++` is called the increment. This particular statement is a shortcut for i=i+1. The increment code is executed each time the loop reaches the `done` statement. After the loop counter is incremented the test is performed. Sooner or later the loop counter will be large enough that the test will be FALSE and the loop will be done.

7.21 Write a shell script that uses a loop to build a table of weights on the Earth and the corresponding weight on the Moon. You can either write the entire script, or modify the pounds2Kilos.sh.

a. The table should start at 0 pounds and go to 250 pounds counting by 10. (You can hard code these numbers into the loop)
b. To calculate the weight on the moon multiply the earth weight by 0.166
c. The table should have the headings "Earth Weight" and "Moon Weight"
d. The left hand column of the table should contain the Earth weight in pounds, the right column should contain the corresponding weight on the moon.

7.22 Modify the moon weight shell script so that it asks the user for the starting number, ending number and increment. You can either write the entire script, or modify the pounds2KilosUserInput.sh.

Repeating Code With Conditional Loops – the `while` *and* `until` *statements*

The `while` and `until` statements are used to build loops that repeat until a specified condition is met. The basic syntax for a `while` loop is:

```
#!/bin/sh

# Initialize the loop counter
salary=1
week=1

while [ $salary -le 10000000 ] #Same tests as for loops
do
  echo -e "$week: \t $salary"
  salary=`(echo $salary \* 2) | bc`  #Increment the counter!
  week=`expr $week + 1`
done
echo -e "$week: \t $salary"
```

A `while` loop is similar to a `for` loop in that it requires three, things, an initial condition, a test for completion and an increment. However in a `while` loop only the expression for testing for completion is on the same line as the `while` statement. The initial condition must be set before the `while` statement, and the increment must be set inside the loop code. Each time the loop code reaches the `done` statement it loops back up and repeats the test. As long as the test is TRUE the code between the `do` and `done` statements will be executed. The expressions in the completion test are the same expressions used in `for` loops.

```
#!/bin/sh

# Initialize the loop counter
salary=1
week=1

until [ $salary -gt 10000000 ] #Same tests as for loops
do
  echo -e "$week: \t $salary"
  salary=`(echo $salary \* 2) | bc`  #Increment the counter!
  week=`expr $week + 1`
done
echo -e "$week: \t $salary"
```

An `until` loop is exactly the same as a `while` loop, except the test is reversed. The loop code will repeat as long as the test is FALSE; when it's TRUE the loop is done.

7.23 Add a loop around the outside of the moon weight program. You can either write the entire script, or modify the moon weight script you created previously, or modify moonWeightUserInputRepeat.sh.
a. The script should prompt the user and read the starting weight, ending weight and increment.
b. To calculate the weight on the moon multiply the earth weight by 0.166
c. The table should have the headings "Earth Weight" and "Moon Weight"

d. The left hand column of the table should contain the Earth weight in pounds, the right column should contain the corresponding weight on the moon.
e. After the table is built, the script should ask the user if they want to repeat the process and read their response. If the user responds in the affirmative the script should repeat, otherwise the script should exit.