** Chapter 1 Commands and Chapter 2 vi**

Chapter 1 - Last half

logging out use **exit** or **logout**

**exit** terminates current shell

Viewing current shell:

echo $SHELL ---  shows login shell (it was NOT wrong in these notes!!!!)  
  
echo $0 --- shows current shell or shell script filename of running script

Appendix F UNIX Commands (pg. 828) list of commands

discussed in text book

command line syntax:

command\_name  [-switches] [argument 1] [argument 2]

[ ] – optional

command\_name -- name of an executable

[-switches] -- modify the operation

[arguments] -- parameters often file or pathnames

in UNIX, the current directory is called "Working directory"

ls             -- shows you the files in the working directory

ls –l        -- shows the files with a detailed listing

ls  –a      -- shows ALL files in the working directory including hidden files (to hide a file rename it with a period as the first character of the name e.g.   .profile).  
cd           -- change directory from working directory to whatever path that follows the command  
                 --default (no parameters) takes you home

date       -- todays date.. lots of switches available

cal           -- calender for the month -- cal 1978, will show the year 1978

who       -- list of users currently logged on to the system

who -H -- put headers over columns

who -q -- quick list usernames

who -u -- user idle times              0:18 – 18 minutes since last active old – over 24 hours since last active  .     – less than a minute

who –uH              -- idle time and headers

who am i              -- info about you the user

whoami                -- your username

Getting help

man       -- stands for manual and is used to get info about a command (ie.. man cal -- will show

options for the calender command)

man -k calender               --will do a text level search through the manuals and return anything related to the topic

q - to "quit" man

whatis   -- gives a brief description of the command that follows

Printing files

lpr filename        --print file(s) (to the printer that is local to the computer that unix is running on)

Some more commands (page 24)

tty          -- shows the name of the device or terminal that you are on

write user-id      --allows you to write a message to someone else

                ctrl d -- will end your write session (ctrl d - EOF - end of file)

stty -- sets your terminal with specific settings

stty erase ^H -- will make it so your backspace key works rather than produce a ^H

Control Keys

ctrl + z -- suspends a foreground job (can't log off until you resume or kill the process -- ps to get the pid, then kill pid)

fg -resumes a suspended job to the foreground

ctrl + c -- aborts or terminates a running foreground job

ctrl + d -- EOF(End Of File)

script -- record session of what is happening and all I/O

of what's happening on monitor it's default filename is typescript

script "filename" -- make a custom filename

with the above command ("exit" to exit)

script -a "filename" -- will append to an existing file (if file doesn't exist, it will get created)

COMMAND LINE EDITING\_\_\_\_\_\_

up arrow              to retrieve previously entered command

left and right      to move about the command

down arrows     to go back through commands if you used up arrow too many times

Viewing file contents

cat filename       -- this command will dump the files contents to the screen (cat short for concatenate)

more filename  -- will display a file (like cat), but 1 page at a time - use the spacebar to move to next screen

less filename     --does everything that more does, but also allows use of the up arrow to back up

head [-n] filename          --  shows the first n lines of a file where N is an integer - default is first 10 lines

tail [-n] filename              --  shows the last n lines of a file where N is an integer -- default is last 10 lines

uname                  --gets name of the operating system

uname –a            --(all) lots of information

              -r             release

             -n            system name

bc -- is the calculator (we will not use this during the course) ctrl D will get out of calculator

;               to separate commands on the same line use ";"

Commands from the chapter are listed on page 33

Homework Sessions I to IV pg. 37

**Chapter 2 text editors**

The vi text editor (ascii)

vi "filename"      -- open given file for editing  
                If the file does not exist it is created.

At start up you are in command mode:

press "i" (insert) to actually start editing  
             press "esc" to get back to command mode

In command mode:

":w"       in command mode will **write** the file (save)

":q"        in command mode put will quit vi

":wq"    write and quit

":q!"      quit without saving

 Item

**Chapter 3 File and Directory Commands**

Directory operations

                pwd -- stands for print working directory

                ls -l  -- long list of

                ls -a  -- lists everything including hidden files

                ls -i  -- lists all files and their inode numbers

                ls -p  -- will put a "/" in front of directory names

                ls -Rp -- shows all subdirectories as well (r is for recursive -- some systems only use R - case sensitive)

                                                  the p makes this a readble list

                \*      -- is a wild card for multiple characters (\*.txt will show all files that end with .txt)

                ?      -- is a wild card for a single character  ( ls ???.txt will show all files that are 3 chars + .txt )

                mkdir  -- is the command to make a directory (mkdir /users/faculty/mfechner/subdir) (mkdir subdir -- just goes into the working directory )

                rmdir  -- is the remove directory command (USELESS because the directory must be empty to perform this command

                cd     -- change directory (without an argument, takes you to your home directory)

                                                  cd           move to home (user)  
                                                  cd  ~       move to $HOME directory  
                                                  cd  ~/mysubdir  
                                                  cd  ..       move up one to parent directory  
                                                  cd  ../..    move up two levels  
                                                  cd  ./subdir   -- "." is the working directory. Same as: cd subdir

File and Directory Operations

                cp      -- copy file or directory

                cp -i   -- Interactive, will prompt before over writing existing files

                                                -- CANNOT do a wild card copy within a directory  
                                                -- you can use absolute or relative paths

                cp -r                         -- r for recursive - meaning sub directories too.

                cp -r dirA dirB        --copies first directory and all of it's sub directories into second directoy)  
                                                -- if the first argument is a directory, the 2nd must also be a directory COMMON SENSE!!

                mv                          -- is the command to move a file -- can also be used to rename files (page 100 )

                mv source destination

                mv /root/filename ~   --moves the file named, filename, from root user directory to the home directory

                mv oldfilename newfilename   --renames the file from oldfilename to newfilename

                                                (we're skipping LINKS for now but we'll come back to them)

                rm            -- is the remove command - delete (pg 106)

                rm -i       -- Interactive  - will prompt before removing file

                rm -R      -- Recursive deletion, deletes all sub directories and their files  
                                   -- this allows wild cards and can be used with absolute path or with relative paths

                echo rm -Rf \* -- this will print what the remove command WOULD delete if you performed the command, doesn't actually perform the command  
                                      -- you must have the -f to delete directories that are empty

 Item

**Chapter 3 Links**

Links

                                Link by itself means hard link.

                                Linking is a way to give a SINGLE file more than one name and/or place it in more than one directory.

                                This cannot be done across devices.  Also, directories cannot be linked.

**ln**                           -- link command

                example:

                ls

 chicken  
 cow

                ln chicken egg                                    egg is now linked to the same file as chicken

                Let us exam the two file to see if we can tell they are the same file:

                ls -l

                                -rwxrwxrwx 2 carter group 234 Oct 23 2014  chicken

                                -r--r--r--       1 carter group  45   May  4 2012  cow

                                -rwxrwxrwx 2 carter group 234  Oct 23 2014  egg

                                The first character on each line is the file type (- indicates regular file)

                                The next nine characters are the file's permissions.

                                The next number is the number of links to the file (link count).

                                Next is the file owner.

                                After the owner is the group(s). User groups are set up by the system administrator.

                                The number after that is the file size in bytes.

                                The date is that last modified date.

                                The last thing is the file name.

                Even though both files have the same characteristics and have two links to them, we cannot be sure.

**IMPORTANT:**

                To tell if two files are the same file:

                1) ls -i    #to get their inode numbers

                2) If the files have the same inode numbers, then they are the same file.

                ls -i         1616 Chicken

                                849  cow

                                1616 egg

                inode Numbers

**Indirectly, the inode number is the file's address on disk.**                                Directly, it is the index to that file's entry in the **Inode Table**.                                There is only one inode number for each file on the disk.

                inode Table

                                Contains the meta data about the files (on this disk). This includes:

**File's address on disk**  
                                **Link Count** (IMPORTANT)  
                                file size  
                                owner  
                                privileges  
                                time stamps  
                                (NOT the file name)

                Directories

                What is a directory?  What information is stored in a directory?

                                A directory is a file.

**A directory MUST contain the following information:**

**filename  
                                                inode number**

                                Question: Why do some systems limit filenames to 14 bytes - wouldn't 16 bytes make more sense?  
                                                What do you think the other 2 bytes are used for?

**Deleting files**

**When is a file deleted (its space on the disk is marked available)???  
                A file is deleted when its link count in the inode table goes to 0.**

                Question:

                Why can't we link directories?  
                hint: hierarchical directory structure

                Why can't we link across devices?  
                 Because each device manages its own files (different inode tables).

                Note: inode tables do not contain file names.

Symbolic Links (or soft links)

                Each physical device (floppy, hard drives, etc) manages it's own files and directories. This prevents us from linking across devices

                Instead we use a soft link.

**A symbolic link contains the actual pathname to the file or directory.**

                They can be used to link to files on different devices or to directories on the same or different devices.

**ln -s**  -- is the command to create a symbolic link

                ln -s /home/afile afilesymlink

                -- now afilesymlink will show up in the working directory

                -- if you do an ls -i on the directory

                       -- the inode numbers will be different between the symbolically linked file and the original file

                -- if the original file is deleted and you try to use the symbolic link, you will get an error (soft link still exists but file doesn't exist anymore)

                -- deleting a symbolic link will have no effect on the original file

                -- symbolic links are like shortcuts

 Item

**The Find File (find) Command page 108**

The Find File (find) Command page 108

 -- find searches a directory tree using the supplied search criteria  
 -- NOTE!! it tends to ignore the rules that other commands use so PAY ATTENTION!!!  
 find path -criteria -criteria -criteria       -- general usage  
   
 e.g.  
 find   /home/carter   -name   myfile   -print  
            |                                  |             |            
 starting directory           | filename   | print to stdout (not needed on linux platforms)

  -name criteria to search for a filename

 e.g.find . –name fred          # find files not named fred

 e.g.  
 to list all files in all the working directory and its sub directories  
 find .

 Listing will include directories in front of the filenames  
  find topdir -name filename  
   -> ./subdir/filename

   
 Wildcard Searches   
 must use single quotes

 find . -name 't\*.c'  
   
 find abosolute/or/relativepath  -type  -- looks for files of specified type in the tree

   d directory   
   f ordinary file   
   l SYMBOLIC link

   e.g. find . -type f   ---finds all ordinary (regular) files in the given path

 find path -mtime  +n -- modified at least n days ago (n element of I)  
     -n -- in the last n days

 find path -size   n -- files of size n blocks (block - 512 bytes)  
    +n -- greated than n blocks

 find path -perm  nnn    -- files with given octal permission code  
 find path -links n     -- files with given number of links (n element of I)  
 find path -inum n     -- file with given inode number

 Combining Criteria   
  -o  will OR our criteria

 the criteria must be placed in parentheses  
 the parentheses must be escaped

 e.g.  
 find .  \( -name myfile.c -o -name anotherfile.c \)  
 find ../.. \( -mtime +6 -o -name new.txt \)  
  files modified 6 or more days ago or named new.txt

 Escaping characters  
 To escape an character on the UNIX system:  
  \ backslash  
  "" pair of double quotes  
  ' '  pair of single quotes  
  e.g. to escape  a semi colon:  
   \;   or ";"  or  ';'  
    

Execute Action

 -exec

 used with find to perform an action on the found files.

  find path criteria -exec command {} ";"

     command to perfom on the found files  
     {} - is a place holder for path of the found files - just include it!  
     ";" you need the semi-colon, but it must be escaped by either\;, ";", or ';'

 To copy found files to an existing directory:  
  find /home/carter -type f -exec cp {} /home/you \;

   #there is a problem, because you are copying to a subdirectory  
   copy all ordinary (regular) files to the existing directory /home/you

 -ok    -- this is just like -exec, but it asks for confirmation (Y/N) before each action  
  e.g. find . -mtime -100 -ok rm {} ';'

 -newer     
  find . -name file -newer path/file  
      find all files newer than the given file

 Item

**Chapter 4 Permissions**

Directories need pass through permissions (x)  
   -- If you don't have pass through permission for all parent directories, you can't access the file/directory.

                r              read  
                file - can look at file contents (ie perform cat, more, head, tail)  
                directory - can list the filenames **only**

                w            write  
                file - can be edited (and deleted IF the directory ALSO has w)   
                directory - files can be added (files can be deleted IF the file ALSO has w)

                x              execute  
                file - can be run if it is an executable or a shell script  
                directory - can be accessed (cd to directory, but cannot read the directory if you do not have r).  
                Commands can be run on the directory.

Use the chmod (pronounced cha mod) – change mode - to change a file’s permissions.  
To access a directory, you MUST have execute permissions to ALL parent directories above it (page 125).

                \*\*\*\*\*\*Absolute (octal) codes page 128

                e.g.

                chmod 755 myfile

                                7              5              5  
                                user       group   other  
                                111         101         101  
                                rwx          r-x          r-x

                                   7 = 111 = read write execute  
                                   6 = 110 = read write  
                                   5 = 101 = read execute  
                                   4 = 100 = read only  
                                   3 = 011 = write and execute  
                                   2 = 010 = write only  
                                   1 = 001 = execute only  
                                   0 = 000 = none

                Octal chmod sets all permissions wiping out previous settings.

 Item

**Chapter 5 Redirection and Pipes**

CS2080 Chapter 5 Redirection                                                                                                                                 Page 146

Standard Streams

UNIX has the 3 usual standard streams

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Stream** |  | **Stream Descriptors**  **ksh and bash ONLY** |
| standard input | stdin | keyboard | 0 |
| standard output | stdout | monitor | 1 |
| standard error | stderr | monitor | 2 |

All of these can be redirected – changed so that their input comes from a file, or their output goes to a file.

5.3 Redirection

To temporarily change these 3 streams we use the redirection operators <, >, >>.

‘<’  Input

Most commands, by default, get their input from the stdin, to change this use  ‘<’.

command < name of file for input

ksh and bash can also use the stream descriptors.

command 0< file

example (using mail):

                mail –s “Subject Line” [pcarter3@uccs.edu](mailto:pcarter3@uccs.edu)  < anytextfile             #the file is the message body

example using cd (thank you to Patrick Leedom)

                cd  <   text file with path

Unfortunately examples like:

                cat < filelist.txt                works the same as                            cat filelist.txt  
                ls –l < filelist.txt               works the same as                            ls –l filelist.txt

‘>’, ’>>’ Output Redirection Operators

To send a command’s output to a file rather than screen

                command > outfile                                          csh, bash, ksh  
                command 1> outfile                                       bash, ksh

Overwrite existing

                command  1>|  outfile                                   bash, ksh  
command   >|   outfile                                   bash, ksh

                command   >!  outfile                                     csh ONLY

Append   >>

                command    >> file                                           bash, ksh, and csh  
command    1>>  file                                        bash, ksh

NOTE:  ‘>’ overwrites existing files (like >| or >!) when redirecting

                If **noclobber** option is **off**, file is overwritten.  
If **on**, you get an error and file is not overwritten.

5.14 How to turn **options ON**and**OFF**                                                                                                                     **pg 183**

|  |  |  |
| --- | --- | --- |
| **page 184 table 5.10** | **ksh and bash** | **csh** |
| set  (ON) | set –o noclobber | set noclobber |
| unset  (OFF) | set +o noclobber | unset noclobber |
| display all option settings | set -o | set |

Table of some common options **page 183 table 5.9**

|  |  |  |
| --- | --- | --- |
| **ksh and bash** | **csh** | **explanation** |
| noclobber | noclobber | do not overwrite existing files when redirecting |
| verbose | verbose | print command before execution |
| xtrace | xtrace may not work at command line | print command with values before execution |

2>, 2>>, 2>&1, >& Error Redirection

To send errors to a file.  Remember, only bash and ksh use the stream descriptors (0, 1, 2).  This means that csh can only redirect the errors to the same file as the stdout.

**ksh and bash**

                2>                                           send error messages to given file

                2>>                                        append to file

                2>&1                                     send error wherever you sent the **stdout**

**csh, ksh and bash**

                >&          send errors to wherever you sent the **stdout**

**Examples/ Explanations of Redirection of Both stderr and stdout**

1. to different files (ksh and bash only)  
   command 1>  outfile   2>  error file           # order makes no difference  
   ls file1 2>  errorfile   > outfile
2. to send both to same file (csh, bash, ksh)  
   If you used method 1), the first redirection opens the file, then the second cannot access it.  
   Specify where you want the stdout to go and then tell it to merge the output for the stderr  
                      command    >   file   >&                                csh, ksh, bash  
                   some other operators   
    1>&2,    >|          ksh, bash  
   >&!                        csh only

5.4 Pipes (examples)

From Ultimate UNIX (page 202) :

who  > users.txt                               # 1 user per line

wc  –l < users.txt              #wc – word count –l gives number of lines

outputs: 4                           # number of users on system

 but we could just:

who | wc  –l                       # same thing without creating users.txt

NOTE: ls | wc  –l                # number of files

Wang:

history | tail -8 > last8history.txt

your commands | just the last 8  >  file

5.5 tee ( Kernigan and Pike):

“Data flowing through a pipe can be tapped and placed in a file (but not another pipe) with the tee command.”

(date ; who) | tee save.txt | wc

outputs:  3          16           89

                lines       words   chars (bytes)

cat save.txt         #date and list of users

wc < save

output: 3             16           89

end Kernigan and Pike

Reads from the stdin, writes to stdout and one or more files.

-a to option to append

pwd | tee list.txt              #creates or overwrites list.txt and prints to stdout

ls | tee –a list.txt              # append to list.txt

cat list.txt:

                directory name

                list of files

 Item

**Chapter 5.6.4 Types of Command Execution 5.8 Quotes 5.9 Command Substitution**

---------chapter 5.6 4 types of command execution----------'

1- **Sequenced**, cram all your commands on one line and use semicolon to separate commands

                ls -l > list.txt ; ls –l  >>  list.txt

                Commands have no dependence on each other

2- **Grouped**, commands are separated; and grouped with () or {}

                ( pwd ; ls –l ) | lpr             # both outputs go into the pipe (without the grouping, only ls – l does)

3- **Chained**, same as pipes (output from first command becomes input for next command)

4- **Conditional**, and "&&" or "||" commands execute sequentially

                Both “short circuit”

                &&         if first command does not have an exit status of 0, second command does not execute

                                ls myfile && cat myfile

||           if first command has exit status of 0, second does not execute

NOTE exit status 0 ALWAYS means the command executed successfully.

-----------------chapter 5.8 Quotes --------------------------

                Meta Characters

                                characters that can either be plain text or they can have special meaning like \n, (), ;

                3 types of quotes \, “ ”, ‘ ‘

                \              escapes next char – the usual

                “ “           treats string as literal, unless it is another “, ` (back quote), $, \ (the usual)

**Problem: csh will ignore “\” for “\$var” and “\`command`” inside double quotes.  
                                So “ ”s invalidates \$ and \` in csh.**

                ‘ ‘            treats string as literal – UNIX does not peek inside to interpret special characters.

5.9 Command Substitution          pg 164

Convert the output of a UNIX command to a string; these can be embedded in another command.

bash and ksh:

echo “User $(whoami) is on the system $(hostname).”

                -> user jsmith is on the system sanluis.

echo “Your current directory is : $(pwd)”

                -> Your current directory is : /user/jsmith

csh, ksh, and bash (we will use this method):

echo “Your current directory is : `pwd`”  #these are back quotes (also called accent grave)

                -> Your current directory is : /user/jsmith

echo “The year is `date +%Y`”

                -> The year is 2017

 Item

**5.10 Jobs 5.12 Variables 5.13 Predefined Variables 5.14 Options**

**5.10 Jobs**

A job is a USER executed process (i.e. a command is entered at the prompt by the user). This distinguishes them from the generic idea of a process. (jobs are a subset of processes.)

Process - Something running on the system.

**Foreground Jobs**Foreground jobs must be run sequentially, because they take control of the keyboard (stdin) and the monitor (stdout), and don’t release them until they are completed (or suspended).

For foreground jobs, the user can:

<ctrl>+z    – suspend/interrupt  
fg                 – resume suspended to foreground  
<ctrl>+c    – cancel or terminate job

**Background Jobs**

A job can be made to execute in the ‘background,’ allowing the user to use the terminal for other things. This is done by placing an **ampersand (&)** at the end of the command.

$ command\_that\_takes\_a\_long\_time &

Problems: If a background job generates output, it will come up on the screen, so you could redirect the output (including errors to a file(s)).

$ cat myfile **>&** outfile **&**  //Send errors and output to outfile

When a job is placed in the background, UNIX comes back and gives the user a **Job Number** and a **Process ID** (PID) number.

$ sleep 200 &

[1] 568               //Job#, PID

The process id (PID) is the number the system uses. The job number is a “shortcut” that the user can use instead of the pid. If you are running 3 jobs, then they will be numbered 1, 2, and 3, but their pids will be larger and probably out of sequence. Eg. 1549, 1600, 1602.

**Background job commands**:

stop %n – interrupt/.suspend – may only work in csh

bg %n – resume suspended job to background

kill %n – terminate job

fg %n – move background job to foreground

To move a foreground job to the background:

<ctrl>+z        //Suspend it  
       bg                   //Send it to the bg – defaults to last “touched” job.

**jobs command**Use jobs to get a list of your jobs, job numbers, and their status

$ jobs

[1] – Running lpr bigfile  
      [2] + Running sleep 200

**Currency flags**

Appear after job number

**+**Usually the last job started/ “touched”  
-           Next to last job started/ “touched”

If a job is suspended, it will become +. The most recently suspended job is +. If you enter a job control command with no job #, the + job is used (%% - current job).

fg %+  
      kill %%  
      stop %-  
      bg %4

Remember, job numbers are only relevant to the particular user and their current session.

**Process ID** (pid)  
Each process, even if it is a child, is assigned a unique process ID number (pid).

**The ps command**ps will show you the processes and their pids that are running on your terminal.

ps –a shows processes for all users

ps –l long

ps –e all processes including system and user

ps –f full listing (shows process id of parent (ppid) for child processes)

**5.12 Variables** p. 176

WARNING: I have not thoroughly  reviewed these notes - they might contain errors  
Variable names start with an alpha character, or an underscore.  
The value of a variable is ALWAYS A STRING. You need to use quotes when setting a value, if it contains spaces or special characters.

**ksh and bash:**

variable=value (no spaces)

myVar=6

x=”Hello Student”

**csh**

set variable = value (spaces)

set myVar = 6

Set y = “Hello Student”

To see the variable’s contents:

echo $varname

echo the var is $myVar

Note: User defined variables are typically lower case. In addition, if you fork a new shell, your variables will not be available in the new shell (they only exist in the shell process they were defined in).

**5.13 Predefined Variables**

**Shell variables**Includes user variables  
Local to shell

These apply only to the current shell (process). They customize the shell.

$ set          #to view in csh (includes other “stuff”)

**Environmental variables**Transcend shells (global)

These are typically set at login (they can be changed at runtime) and stay with you as you move from shell to shell.

**To view  Environmental variables  
ksh and bash:**

$ set

**csh:**

$ setenv

Examples:

**$HOME** (environmental) (also $home in csh) holds your home directory.  
**$PATH**   holds the list of directories that will be searched to find the executable of a file interred at the command line without a path.  This includes commands!

**csh:**

setenv HOME/home/pcarter

**ksh and bash:**

HOME=/home/pcarter

**$PATH** (Environmental)  
When you type in a command (e.g. ls, cat,…), the shell looks through the directories listed in the PATH variable, until it finds the executable for that command (so their directories usually end in /bin).  
**This variable’s paths are ordered from system to user.**

**ksh, bash:**

echo $PATH  
       /bin:/user/local/bin

to set:  
      PATH=$PATH:$HOME/bin   #add your user bin folder

**csh:**

setenv PATH $PATH:$HOME/bin

**$SHELL** (environmental) (also shell in csh)

            echo $SHELL (Shows login shell)

**5.14 Options**

We have already looked at

            noclobber (don’t automatically overwrite existing files when redirecting)

Two other useful options are (for shell script)

            verbose – print command before executing

            xtrace – print the command and its arguments’ values before executing

**bash:**

ON:                             set –o xtrace  
OFF:                            set +o xtrace

Show all options:        set –o

 Item

**5.15 Shell/Environment Customization**

5.15 Shell/Environment Customization

Temporary

When we set an option, alias, or variable at the command line or in a shell script that we run ourselves.Â Â Once we log off, there settings are lost.

Permanent

Putting special commands, setting predefined variables, and setting options in startup and shutdown files will create â€œpermanentâ€ changes.Â Â These are login, shell startup, and logoff files.

Rundown of these files:

 Item

**Chapter 16 Command Execution Process**

 Item

**Chapter 17 File Expressions (work for all shells)**

UNIX and Shell Programming, Forouzan & Gilberg  pg.726

In an if statement, you might put:  
                if  ( -e filename )  then                    or  
                if  ( -e $var\_holding\_filename )  then

Operator             Explanation  
-e   filename       true if file exists  
-r   filename        true if user has read permissions for file  
-l   filename         true if file is a symbolic link  
-w   filename      true if user has write permissions for file  
-x   filename       true if user has execute permissions for file  
-o filename         true if user is file owner  
-f   filename        true if file is a regular file  
-d   filename       true if file is a directory  
-s   filename        true if file size is greater than 0  
-z   filename        true if file size is 0

Implied in each of these expression, is that if the expression is true, then the file also exists.

 Item

**Chapters 16 and 17 (partial)**

16.4 Output  
Has some interesting echo stuff - special characters.  
  
  
16.5 Input $<  
$<     Read Construct  
Used in scripts to get input from the user.  
set var = $<  
The value of var is everything the user types in up to the first "white space" - even if they put quotes around it.  
  
  
16.6 Exit Status  
csh uses $status to return the exut status of the last command executed (ksh uses $?).  
0 - is always success.

16.7 eval  
When we store complex commands in variables, or the name of another variable in a variable, we may need the interpreter to look a little harder at what we are doing.  eval does this.

16.8- 16.10 covered in previous classes

16.11 Command History  
     We don't need this for scripts - skip.

16.12 Already covered.  
  
  
  
17.1 Shell Scripts  
An ASCII text file  
line 1 of script - NO blank lines before  
#!/bin/csh    # this is the interpreter designator line  
  
  
Comments  
# -  comment to end of line

White space  
excess white space is ignored

Commands  
commands are separated by new lines or semicolons  
  
  
Executing Scripts  
csh filename  
or   
csh  -vx filename  #turns on verbose and xtrace for life of script - good for debugging  
  
  
We would like to make your script execute by just typing in the filename.  
1) change permissions to include execute  
2) place file in ~/bin, if you don't have a bin folder create one.  
3) fix the $PATH in login file to include $HOME/bin  
  
  
To fix $PATH  
1) determine your login shell  
2) edit the appropriate login file  
   FIRST make a back up of your login file  
    cp filename filename.bak  
PATH=$PATH:$HOME/bin  #keep the existing PATH and add new one to END!!! this is ksh & bash  
Now you can run your script simply by typing in its filename.

 Item

**Chapter 17 and 18**

 17.3  switch  
 17.4 loops  
         break  
         continue  
         goto  
17.5 Special Parameters  
18.8 Arrays  
18.1 Variable Evaluation { }

For the moment - 18.8 and 18.1 are the only sections we will cover in Chapter 18.  
We will cover section 18.6 Built-in Commands eventually.

 Item

**Chapter 9 Regular Expressions (and Chapter 10 grep)**

**Chapter 9 Regular Expressions pg 331**

A pattern of characters used to compare and match text.  This has many uses and is used by the grep, sed, and awk utilities (to name a few).

9.1Atoms

Specifies what text is to be matched and where (e.g. beginning of line, or word or end of line or word…).

\*Single Character  
e.g. Find all lines with an ‘L’ in them

\* Dot  ‘.’

Matches ANY single character – except a newline - \n

e.g. ‘.q’                 matches Aq, aq, Bq, bq, 5q …

 i.e. exactly 2 characters, the second must be ‘q’

\*Class [ ]

A set of ASCII characters.  If any one matches, then it is a match.

e.g. [aAq]            ‘a’, or ‘A,’ or ‘q’ will match.

                                Cow, Quick5 – no match               krauq, dAg – match

We can also specify a Range using the dash (hyphen) ‘-’.

e.g. [a-f], [0-9]

There is also a complement operator (NOT) ‘^’.

e.g. [^12ab]        matches all that are not 1, 2, a, or b.

Escape ‘\’ is needed to match ‘^’,’-’, since the class uses them.

To match brackets [] use [[a] or []a] where ‘a’ is just another character in the set.  The idea is that the type of bracket that you are looking for (these are not the droids you are looking for) needs to be the first member of the class.

Note:

To match any single alpha character [A-Za-z].

To match any single digit [0-9].

\*Anchors ^, $, \<, \>

                                ^             find pattern at START of line

                                $              find pattern at END of line

                                \<           pattern must occur at beginning of word (whitespace or start of line just before)

                                \>           pattern must occur at end of word (whitespace or end of line just after it)

\* Back Reference \1 \2 \3 \4 \5 \6 \7 \8 \9

These are used in conjunction with the SAVE operator.  They refer to a pattern that has already been found and saved into one of nine buffers.  We will see how they are used in the next section.

9.2 Operators

These allow us to combine atoms.

There are 5 types

\* Sequence – normal

e.g

^UNIX                   lines that start with UNIX

[1-3][0-6]             10, 11,…16,…36

^$                           blank line

^..$                        2 character line

[0-9][0-9]             any 2 digits

m..h                       e.g. math, m&\*h, mush, m90h,….

\* Alternation     ‘|’ (OR)

(Dos)|(dos)        either will match

(This)|(that)

\* Repetition

  \{m, n\}

atom goes just before

m- minimum number of times pattern MUST be repeated

n – maximum number of times pattern can be repeated

e.g.

A\{1,3\}                A, AA, or AAA

C9\{2,4\}              C99, C999, or C9999

The repetition operator may be used with or without one or the other of the min or max (but at least one of them).

e.g.

                                \{2\}      atom occurs EXACTLY 2 times

                                \{2,\}     occurs at least 2 times

                                \{,2\}     occurs 0 to no more than 2 times

Ryker\{,2\}         Ryke, Ryker, or Rykerr

Short Form Operators (Repetition cont.)

                                \* repeat atom 0 or more times                  \{0,\}

                                + atom must occur at least once                \{1,\}

                                ? atom occurs 0 or 1 time                              \{0,1\}

e.g.

…….

…….

\* Group Operator ()

Used to group multiple characters.

\*Save Operator \(…\) (may not always need to escape)

Copies a string into a buffer.

There are nine of these buffers (back references):

                                \1 \2  \3 \4  \5 \6 \7 \8 \9

For a given expression the first piece of text saved is put in \1, the second in \2 and so on.

Example of repeated (doubled) words.

…. see your notes from the board.

**Chapter 10 grep pg 355**Global Regular Expression Print

GREP

1) reads one line at a time into a buffer (from the stdin or a file).

2) searches the line for a match with a regular expression.

3) If the line matches, then it is printed to the stdout.

GREP is a ***filter***, so it can be used on either side of a pipe.

GREP searches an entire line at a time.  If there is a match, the entire line is sent to the stdout.

EXIT CODES  
GREP returns an exit status.

                0              EXIT\_SUCCESS

                1              Pattern not found

                2              File not found

NOTE: SED and AWK do not return exit status (only if there is a syntax error).

GREP cannot alter or truncate the line.  
It must search the whole file.  
It cannot base a match on a previous line.

10.2 grep, egrep (extended), and fgrep (fast)  pg 354

Options

                -c            displays **count** only of number of matching lines.  
                                If used with multiple files, displays filename prefixed with line count.

                -i             ignores case (UPPER and lower care match).

                -l             prints list of filenames that have at least one matching line.  Does not print the lines.

                -n            print line number and matching line.

                -v            prints lines that do not match.

grep, egrep and fgrep cannot use all of the regular expressions we have used.

fgrep  
Used to search for simple strings or patterns.

egrep  
Allows for more complex patterns than the other two.  Especially alternation | and group ( ).  
                (cat) | (dog)        cat OR dog

10.4  Searching for file content  pg. 363  
To search in all files in the working directory:

                                grep – l   ‘string’     \*

To search ALL Directories in a Path

                                find the\_path  -type f   -exec   grep -switch   ‘Regular Expression’ {} \;

                                find ~       -type –f     –exec -    grep   -l  ‘Raven’  {}  \;  
searches $HOME on down for files that have at least one line with “Raven” in them and prints only the name of the matching files.

 Item

**Make**

Make is an Automated Program Compilation Utility.

It determines from a set of rules the programmer supplies, which modules from a given program need to be recompiled (or re-linked) and recompiles (or re-links) them.

Make invokes a file named, makefile, in which the user has listed which file(s) must be updated if certain other files have been modified more recently. It then updates them (i.e. recompiles and links them).

Example:

If I had a C program that consisted of 2 source files:  
myMain.c   and myFunc.c

Then the makefile would look like:

my\_main.o: my\_main.c                 # Establish a dependency. The object file, .o, depends on the source file  
                gcc -c my\_main.c             #Then re-compile .c file (don't link) MUST HAVE A TAB AT START OF LINE  
my\_func.o: my\_func.c                   #If my\_func.c is newer than my\_func.o  
                gcc -c my\_func.c                                              #Then recompile  
my\_executable: my\_main.o my\_func.o                  #If the object files are newer than the executable, relink  
                gcc my\_main.o my\_func.o -o my\_executable      # -o allows user to name executable- otherwise it is a.out in unix

#These files may contain comments - just like scripts

If the file, makefile (or Makefile) is in the working directory, to run it just type:

                make

If the file had a different name, use –f:

                make –f name of makefile

### Interesting bash script that we did not have time for.

Posted on: Wednesday, March 21, 2018 3:48:48 PM MDT

Dear Class,

This script can be found at:

http://www.tldp.org/LDP/abs/html/contributed-scripts.html#DAYSBETWEEN

Have a good break and I will miss you,

Pam

#!/bin/bash

#Example A-7. days-between: Days between two dates

# http://tldp.org/LDP/abs/html/contributed-scripts.html#DAYSBETWEEN

# days-between.sh:    Number of days between two dates.

# Usage: ./days-between.sh [M]M/[D]D/YYYY [M]M/[D]D/YYYY

#

# Note: Script modified to account for changes in Bash, v. 2.05b +,

#+      that closed the loophole permitting large negative

#+      integer return values.

ARGS=2                # Two command-line parameters expected.

E\_PARAM\_ERR=85        # Param error.

REFYR=1600            # Reference year.

CENTURY=100

DIY=365

ADJ\_DIY=367           # Adjusted for leap year + fraction.

MIY=12

DIM=31

LEAPCYCLE=4

MAXRETVAL=255         #  Largest permissible

                      #+ positive return value from a function.

diff=                 # Declare global variable for date difference.

value=                # Declare global variable for absolute value.

day=                  # Declare globals for day, month, year.

month=

year=

Param\_Error ()        # Command-line parameters wrong.

{

  # basename extracts end of path (looks for last /) - gives filename

  echo "Usage: `basename $0` [M]M/[D]D/YYYY [M]M/[D]D/YYYY"

  echo "       (date must be after 1/3/1600)"

  exit $E\_PARAM\_ERR

}

Parse\_Date ()       # Parse date from command-line params.

#    # - strip shortest match from beginning

#    % - strip shortest from end

{                                                                                          # %% and ## do greedy pattern match

  month=${1%%/\*\*}     # strips /day/year to get month

  dm=${1%/\*\*}         # Day and month. Strip year off end of $1                             # (match at end to strip year off)

  day=${dm#\*/}                  #strip month off front

  let "year = `basename $1`"     #Not a filename, but works the same.                       #basename command strips off front of a path                           #(everything before last "/")

}

check\_date ()                 # Checks for invalid date(s) passed.

{

   # -gt Greater Than   -lt  Less Than

  [ "$day" -gt "$DIM" ] || [ "$month" -gt "$MIY" ] ||

  [ "$year" -lt "$REFYR" ] && Param\_Error

  # Exit script on bad value(s).

  # Uses or-list / and-list.

  #

  # Exercise: Implement more rigorous date checking.

}

strip\_leading\_zero () #  Better to strip possible leading zero(s)

{                     #+ from day and/or month

  return ${1#0}       #+ since otherwise Bash will interpret them

}                     #+ as octal values (POSIX.2, sect 2.9.2.1).

     # #  ${string#substring} - delete shortest match of substring

     #  + from FRONT of string

     # ## delete longest match from FRONT

     # % delete shortest match of substring from END of string

     # %% delete longest match of substring from END of string

     # n=$((10#$var)) convert to base 10 - which would drop leading 0's

day\_index ()         # Gauss' Formula:

{                    # Days from March 1, 1600 to date passed as param.

                     #           ^^^^^^^^^^^^^

  day=$1

  month=$2

  year=$3

  let "month = $month - 2"

  if [ "$month" -le 0 ]

  then

    let "month += 12"

    let "year -= 1"

  fi

  let "year -= $REFYR"

  let "indexyr = $year / $CENTURY"

  let "Days = $DIY\*$year + $year/$LEAPCYCLE - $indexyr \

              + $indexyr/$LEAPCYCLE + $ADJ\_DIY\*$month/$MIY + $day - $DIM"

  #  For an in-depth explanation of this algorithm, see

  #+   <http://weblogs.asp.net/pgreborio/archive/2005/01/06/347968.aspx>

  echo $Days

}

calculate\_difference ()            # Difference between two day indices.

{

  let "diff = $1 - $2"             # Global variable.

}

abs ()                             #  Absolute value

{                                  #  Uses global "value" variable.

  if [ "$1" -lt 0 ]                #  If negative

  then                             #+ then

    let "value = 0 - $1"           #+ change sign,

  else                             #+ else

    let "value = $1"               #+ leave it alone.

  fi

}

if [ $# -ne "$ARGS" ]              # Require two command-line params.

then

  Param\_Error

fi

Parse\_Date $1

check\_date $day $month $year       #  See if valid date.

strip\_leading\_zero $day            #  Remove any leading zeroes

day=$?                             #+ on day and/or month.

strip\_leading\_zero $month

month=$?

#next part is very clever; the function called in a command substitution

#therefore the resulting string will be the value of date1

let "date1 = `day\_index $day $month $year`"

Parse\_Date $2

check\_date $day $month $year                     #make sure the date is legitimate

strip\_leading\_zero $day

day=$?

strip\_leading\_zero $month

month=$?

date2=$(day\_index $day $month $year) # Command substitution again

calculate\_difference $date1 $date2

abs $diff                            # Make sure it's positive.

diff=$value

echo $diff

exit 0

#  Exercise:

#  --------

#  If given only one command-line parameter, have the script

#+ use today's date as the second.

#  Compare this script with

#+ the implementation of Gauss' Formula in a C program at

#+    <http://buschencrew.hypermart.net/software/datedif>