

- **Shell Commands of UNIX**

- **Unix Commands**

- When you first log into a unix system, you are presented with something that looks like the following:

```
CCOEW@localhost ~]$
```

- That “something” is called a **prompt**. As its name would suggest, it is prompting you to enter a command.
 - Every unix command is a sequence of **letters, numbers** and **characters**. But there are no spaces.

- Unix is also **case-sensitive**. This means that **cat** and **Cat** are different commands.
- The prompt is displayed by a special program called the **shell**.
- **Shells accept** commands, and **run** those commands.
- They can also be programmed in their own language. These programs are called “**shell scripts**”.

- There are two major types of shells in **unix**:
 - Bourne shells
 - C shells.
- *Steven Bourne* wrote the original unix shell **sh** and most shells since then end in the letters **sh** to indicate they are extensions on the original idea
- **Linux** comes with a Bourne shell called **bash** written by the Free Software Foundation.
- **bash** stands for **B**ourne **A**gain **S**hell and is the default shell to use running **linux**

- When you first login, the prompt is displayed by `bash`, and you are running your first unix program, the `bash shell`.
- As long as you are logged in, the *bash shell* will constantly be running.

• **Unix Commands**

• **obtaining help**

- The **man** command displays **reference pages** for the **command** you specify.
- The UNIX **man** pages (**man is short for manual**) cover every command available.
- To search for a **man** page, enter **man** followed by the name of the command to find .
- For example:

```
[CCOEW@localhost ~]$: man ls
```

SUMMARY OF LESS COMMANDS

Commands marked with * may be preceded by a number, N.
Notes in parentheses indicate the behavior if N is given.

h H Display this help.
q :q Q :Q ZZ Exit.

I

MOVING

e ^E j ^N CR * Forward one line (or N lines).
y ^Y k ^K ^P * Backward one line (or N lines).
f ^F ^V SPACE * Forward one window (or N lines).
b ^B ESC-v * Backward one window (or N lines).
z * Forward one window (and set window to N).
w * Backward one window (and set window to N).
ESC-SPACE * Forward one window, but don't stop at end-of-file.
d ^D * Forward one half-window (and set half-window to N).
u ^U * Backward one half-window (and set half-window to N).
ESC-) RightArrow * Left one half screen width (or N positions).
ESC-(LeftArrow * Right one half screen width (or N positions).

HELP -- Press RETURN for more, or q when done

- **man** (*obtaining help*)

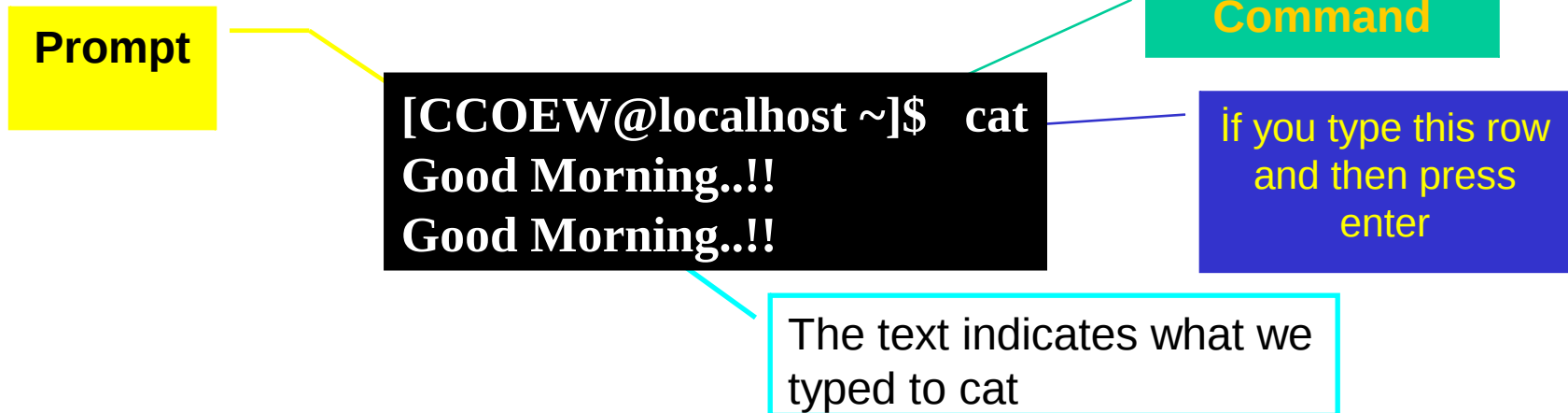
- There is also a **keyword function** in **man**.
- For example;
 - If you are interested in any commands that deal with **Postscript**, the printer control language for **Adobe**
 - Type **man -k ps** or **man -k Postscript**,
you'll get a listing of all commands, system calls, and other documented parts of unix that have the word "ps" (or "Postscript") in their name or short description.
- This can be very useful when you're looking for a tool to do something, but you don't know its name-or if it even exists!

- **cat**

- **cat** command is used to concatenate or displays the contents of a file.
- To use it, type **cat**, and then press **enter** key:

```
[CCOEW@localhost ~]$ cat
```

- This produces the correct result and runs the cat program.



- To end many unix command, type end-of-file command (EOF) *[hold down the key labeled “Ctrl” and press “d” (Ctrl+d)]*

- To display the contents of a file, type *cat filename*

```
bagriy@sariyer:~/EST_guz_2003/hafta_1> cat program1.c
/* C programlama
ilk program */
#include<stdio.h>
int main()
{
printf("ilk C programimiz \n");
return 0;
}
bagriy@sariyer:~/EST_guz_2003/hafta_1> █
```

- To see linux commands press **Tab** key,
- If you want to learn commands beginning with c you can write **c** then press **Tab** key

[Chhaya@localhost ~]\$ c

```

c++          chage          codepage     continue
c++decl      charset         col          control-panel
c++filt      chattr         colcrt       convert_smbpasswd
c2ph        checkalias     collateindex.pl  cp
c_rehash    chfn          colrm        cpio
cal         chgrp         column       cpp
calibrate_ppa  chmod        comm         cproto
cancel      chown         command      crontab
captainfo   chsh         comp         csh
card        chvt          comp_err     csplit
case        ci            compgen      ctags
cat         cjpeg         compile_et   cut
catchsegv   cksum        complete     cvs
cc          clear         composeglyphs  cvsbug
cd          cmp           compress     cxpm
cdecl      cmuwmtopbm   consolechars  cytune
chacl      co           consolehelper

```

• **Storing information**

- Unix provides **files** and **directories**.
- A **directory** is like a **folder**: it contains pieces of paper, or files.
- A large folder can even hold other folders-*directories can be inside directories*.
- In unix, the collection of directories and files is called the **file system**. Initially, the file system consists of one directory, called the **“root”** directory
- Inside **“root”** directory, there are more directories, and inside those directories are files and yet more directories.

- Each file and each directory has a **name**.
- A **short name** for a file could be **add**,
- while it's "**full name**" would be **/home/CCOEW/add**. The full name is usually called the **path**.
- The **path** can be divide into a sequence of directories.
- For example, here is how **/home/CCOEW/add** is read:

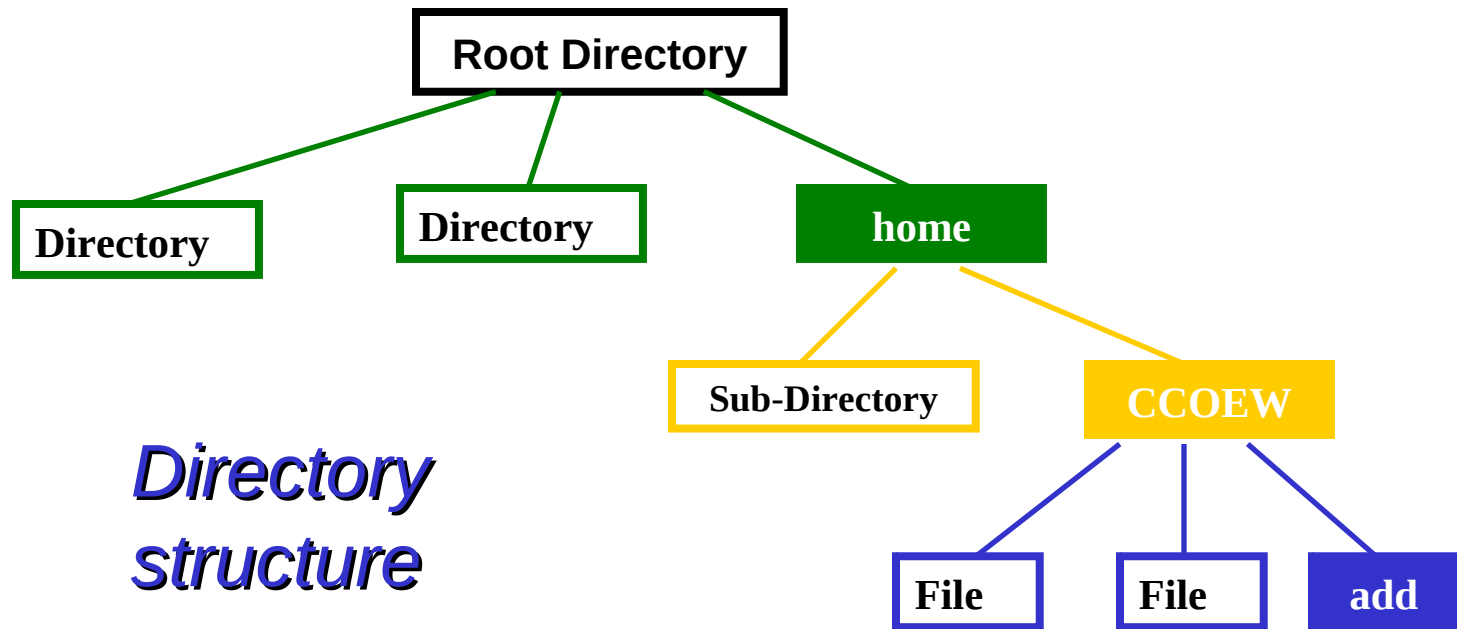
/home/CCOEW/add

The **initial slash** indicates the **root directory**. This signifies the directory called **home**. It is inside the root directory.

The **second slash** corresponds to the **directory CCOEW**, which is inside home.

add is inside **CCOEW**.

- A **path** could refer to either a **directory** or a **filename**, so add could be either.
- All the items before the short name must be directories.



- **Looking at directories with ls**

- The command `ls` lists files.
- If you try `ls` as a `command`, you'll see:

```
[Chhaya@localhost ~]$ ls
```

If you have files, `ls` lists the names of files in the directory

- If you want a **list of files** of a more active directory, try the **root directory**.

```
[Chhaya@localhost ~]$ ls /  
bin  etc    install mnt  root user var  
dev  home  lib     proc tmp  usr  vmlinux
```

“/” is a **parameter** saying what directory you want a list for.

Some commands have **special parameters** called options or switches. To see this try:

```
[Chhaya@localhost ~]$ ls -F /  
bin  etc/    install/ mnt/  root/  user/  var/  
dev/  home/  lib/     proc/  tmp/   usr/   vmlinux/
```

The **-F** is an **option**. It displays file types.

- An option is a special kind of parameter that starts with a **dash** “-”
- An option modifies how the program **runs**, but not what the program runs on.
- For **ls**, **-F** is an **option** that lets you see which ones are **directories**, which ones are special **files**, which are **programs**, and which are normal files.
- Anything with a **slash** “/” is a **directory**.
- **ls -l file*** displays files starting with “file”
- **ls -l** displays all details

```

bagriy@sariyer:~/EST_guz_2003/hafta_1> ls -l
total 56
-rwxr-xr-x  1 bagriy  users  13495 Eki  3 20:41 a.out
-rw-r--r--  1 bagriy  users   115 Eki  3 20:09 prg_1_1.c
-rw-r--r--  1 bagriy  users   424 Eki 11  2002 prg_1_2.c
-rw-r--r--  1 bagriy  users   215 Eki 11  2002 prg_1_3.c
-rw-r--r--  1 bagriy  users   201 Eki 11  2002 prg_1_4.c
-rw-r--r--  1 bagriy  users   324 Eki 11  2002 prg_1_5.c
-rwxr-xr-x  1 bagriy  users  13495 Eki  3 20:41 program1
-rw-r--r--  1 bagriy  users   107 Eki  3 20:41 program1.c
bagriy@sariyer:~/EST_guz_2003/hafta_1>

```


- Many unix commands are like **ls**.
- They have options, which are generally one character after a dash, and they have parameters.
- Unlike **ls**, some commands require certain parameters and/or options. You have to learn these commands.

- **pwd**

- **pwd** (**p**resent **w**orking **d**irectory) tells you your current directory.
 - *Most commands act, by default, on the current directory. For instance, **ls** without any parameters displays the contents of the current directory.*

- **cd**

- **cd** is used to **c**hange **d**irectories.
- The format of this command :
 - cd new-directory** (where new-directory is the name of the new directory you want).

- For instance, try:

```
[Chhaya@localhost ~]$ cd /home  
  
/home]$
```

- If you **omit the optional parameter** directory, you're **returned to your home**, or original directory. Otherwise, **cd** will change you to the specified directory.
- There are two directories used only for relative pathnames:
 - The directory “.” refers to the **current directory**
 - The directory “..” refers to the parent directory
- These are “**shortcut**” directories.
- The directory “..” is most useful in “backing up”:

```
[/usr/local/bin]$ cd ..  
[/usr/local]$
```

- **mkdir**

mkdir (**make directory**) is used to create a new directory,

- It can take more than one parameter, interpreting each parameter as another directory to create.

- **rmdir**

rmdir (**remove directory**) is used to remove a directory,

- **rmdir** will refuse to remove a **non-existent directory**, as well as a **directory that has anything in it**.

- **Moving Information**

- The primary commands for manipulating files under unix are **cp**, **mv**, and **rm**. They stand for **copy**, **move**, and **remove**, respectively.

- **cp**

- **cp** is used to copy contents of file1 to file2

cp file1 file2 (*contents of file1 is copied to file2 in the same directory*)

cp folder1/file1 folder2 (*contents of file1 is copied to file1 in the inside of folder2 directory*)

- **rm**

- **rm** is used to **remove** a file.
 - **rm filename** ---> removes a file named *filename*

- **mv**

- **mv** is used to **move** a file.
 - **rm filename** ---> removes a file named *filename*
- looks like **cp**, except that it **deletes the original file** after copying it.
- **mv** will **rename** a file if the second parameter is a **file**. If the second parameter is a **directory**, **mv** will **move** the file to the **new directory**, keeping it's shortname the same.

- **Some Other UNIX Commands**

- **The Power of Unix**

- The **power of unix** is hidden in small commands that don't seem too useful when used alone, but when combined with other commands produce a system that's much more powerful, and flexible than most other operating systems.
 - The commands include **sort, grep, more, cat, wc, spell, diff, head, and tail.**

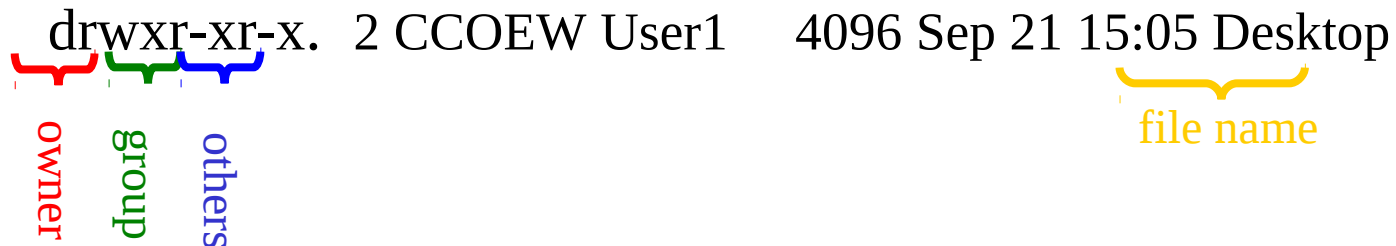
• Operating on Files

- In addition to the commands like `cd`, `mv`, and `rm`, you learned in shell section, there are other commands that just operate on files, but not the data in them.
- These include `touch`, `chmod`, `du`, and `df`.
- All of these files don't care what is in the file.

Some of the things these commands manipulate:

- **The time stamp:** Each file has three dates associated with it. These are **creation time**, **last modification time** and **last access time**.
- **The owner:** the owner of files
- **The group:** the group of users
- **The permissions:** read, write, execute permissions of files. The permissions tell unix who can access what file, or change it, or, in the case of programs, execute it. Each of these permissions can be toggled separately for the owner, the group, and all the other users.

drwxr-xr-x. 2 CCOEW User1 4096 Sep 21 15:05 Desktop



owner group others file name

read, write, execute
permissions of files

- **touch**
- **touch** will update the time stamps of the files listed on the command line to the current time.
- If a file doesn't exist, **touch** will create it.

- **chmod**

- **Chmod** (**ch**ange **mode**) is used to change the permissions on a file.

(owner) (group) (others)

chmod [number][number][number] file1

Number = (read)4 + (write)2 + (execute)1

- Example: **Chmod 754 file1**

for owner: *read, write and execute* permissions (4+2+1)

for group: *read and execute* permissions (4+0+1)

for others: only *read* permission (4+0+0)

- **System Statistics**

- Commands in this section will display statistics about the operating system, or a part of the operating system.

- **du**

du (**d**isk **u**sage) will count the amount of disk space for a given directory, and all its subdirectories take up on the disk.

- **df**

df (**d**isk **f**illing) summarizes the amount of disk space in use. For each file system, it shows the total amount of disk space, the amount used, the amount available, and the total capacity of the file system that's used.

- uptime

- It prints the amount of **time** the system has been “**up**”—the amount of time from the last unix boot
- **uptime** also gives the **current time** and the **load average**. The **load average** is the average number of jobs waiting to run in a certain time period.

- who

- Displays the **current users** of the system and when they logged in.
- If given the parameters **am i** (as in: **who am i**), it displays the current user.

• What's in the File?

- There are two major commands used in unix for listing files, `cat`, and `more`.
- `cat`
- `cat` shows the contents of the file.
`cat [-nA] [file1 file2 . . . fileN]`
- `cat` is not a user friendly command-it doesn't wait for you to read the file, and is mostly used in conjunction with pipes.
- However, `cat` does have some useful command-line options. For instance, `n` will number all the lines in the file, and `A` will show control characters.

- **more**

- **more** is much more useful, and is the command that you'll want to use when browsing ASCII text files

more [-l] [+linenumber}] [file1 file2 ... fileN]

- The only interesting option is **l**, which will tell **more** that you aren't interested in treating the character **Ctrl-L** as a "new page" character. **more** will start on a specified linenumber.

- **head**

head will display the first ten lines in the listed files.

head [- lines}] [l file1 file2 ... fileN]

- Any numeric option will be taken as the number of lines to print, so **head -15 frog** will print the first fifteen lines of the file **frog**

- **tail**



- Like **head**, **tail** display only a fraction of the file.
- **tail** also accepts an option specifying the number of lines.

tail [-lines] [l file1 file2 ... fileN]

- **file**



- **file** command attempts to identify what format a particular file is written in.

file [file1 file2 ... fileN]

- Since not all files have extensions or other easy to identify marks, the **file** command performs some rudimentary checks to try and figure out exactly what it contains.

- **Information Commands**

- The commands that will alter a file, perform a certain operation on the file, or display statistics on the file.

- **grep**

- **grep** is the **g**eneralized **r**egular **e**xpression **p**arser.
- This is a fancy name for a utility which can only search a text file.

grep [-nvwx] [-number] { *expression* } [*file1 file2 ... fileN*]

- **WC**

- **wc** (**w**ord **c**ount) simply counts the number of words, lines, and characters in the file(s).

`wc [-clw] [file1 file2 ... fileN]`

- The three parameters, **clw**, stand for **c**haracter, **l**ine, and **w**ord respectively, and tell `wc` which of the three to count.

- **spell**

- **spell** is very simple unix spelling program, usually for American English. **spell** is a filter, like most of the other programs we've talked about.

`spell [file1 file2 ... fileN]`

- **cmp**
- **cmp** compares two files.
- The first must be listed on command line, while the second is either listed as the second parameter or is read in form standard input.
- **cmp** is very simple, and merely tells you where the two files first differ.

```
cmp file1 [ file2]
```

- **diff**



- One of the most complicated standard unix commands is called **diff**.
- The GNU version of **diff** has over twenty command line options. It is a much more powerful version of **cmp** and shows you what the differences are instead of merely telling you where the first one is.

diff file1 file2

```
gzip [-v#] [file1 file2 ... fileN]
gunzip [-v] [file1 file2 ... fileN]
zcat [{file1 file2 ... fileN}]
```

- These three programs are used to **compress** and **decompress** data.
- **gzip**, or GNU Zip, is the program that reads in the original file(s) and outputs files that are smaller.
- **gzip** deletes the files specified on the command line and replaces them with files that have an identical name except that they have “.gz” appended to them.

- **tr**



- The “**tr**anslate characters” command operates on standard input-it doesn’t accept a filename as a parameter.
- Instead, it’s two parameters are arbitrary strings.
- It replaces all occurrences of **string1** in the input **string2**.
- In addition to relatively simple commands such as `tr frog toad`, **tr** can accept more complicated commands.

tr string1 string2

• **Editors**

- There are a lot of available editors under linux operating system.
- Amongst these **vi** is the most common one. One can claim that every unix system has **vi**.
- However, perhaps the simplest one of the editors is **gedit**.

Shell Scripting

- Start `vi scriptfilename.sh` with the line `#!/bin/sh`
- All other lines starting with `#` are comments.
- make code readable by including comments
- Tell Unix that the script file is executable

```
$ chmod u+x scriptfilename.sh
```

```
$ chmod +x scriptfilename.sh
```

- Execute the shell-script

```
$ ./scriptfilename.sh
```


My First Shell Script

```
$ vi myfirstscript.sh
```

```
#!/bin/sh
```

```
# The first example of a shell script
```

```
directory=`pwd`
```

```
echo Hello World!
```

```
echo The date today is `date`
```

```
echo The current directory is $directory
```

```
$ chmod +x myfirstscript.sh
```

```
$ ./myfirstscript.sh
```

```
Hello World!
```

```
The date today is Mon Mar 8 15:20:09 EST 2010
```

```
The current directory is /netscr/shubin/test
```

Shell Script

- Text files that contain sequences of UNIX commands , created by a text editor
- No compiler required to run a shell script, because the UNIX shell acts as an **interpreter** when reading script files
- After you create a shell script, you simply tell the OS that the file is a program that can be executed, by using the **chmod** command to change the files' mode to be executable
- Shell programs run **less quickly** than compiled programs, because the shell must interpret each UNIX command inside the executable script file before it is executed

Commenting

- Lines starting with # are comments except the very first line where #! indicates the location of the shell that will be run to execute the script.
- On any line characters following an unquoted # are considered to be comments and ignored.
- Comments are used to;
 - Identify who wrote it and when
 - Identify input variables
 - Make code easy to read
 - Explain complex code sections
 - Version control tracking
 - Record modifications

Quote Characters

There are three different quote characters with different behaviour.

These are:

- “ : **double quote**, weak quote. If a string is enclosed in “ ” the references to variables (i.e *\$variable*) are replaced by their values. Also back-quote and escape \ characters are treated specially.
- ‘ : **single quote**, strong quote. Everything inside single quotes are taken literally, nothing is treated as special.
- ` : **back quote**. A string enclosed as such is treated as a command and the shell attempts to execute it. If the execution is successful the primary output from the command replaces the string.

Example: `echo "Today is:" `date``

Echo

Echo command is well appreciated when trying to debug scripts.

Syntax : `echo {options} string`

Options: -e : expand \ (back-slash) special characters

-n : do not output a new-line at the end.

String can be a “weakly quoted” or a ‘strongly quoted’ string. In the weakly quoted strings the references to variables are replaced by the value of those variables before the output.

As well as the variables some special backslash_escaped symbols are expanded during the output. If such expansions are required the -e option must be used.

User Input During Shell Script Execution

- As shown on the hello script input from the standard input location is done via the read command.
- Example

```
echo "Please enter three filenames:"  
read filea fileb filec  
echo "These files are used:$filea $fileb $filec"
```
- Each read statement reads an entire line. In the above example if there are less than 3 items in the response the trailing variables will be set to blank ' '.
- Three items are separated by one space.

Hello script exercise continued...

- The following script asks the user to enter his name and displays a personalised hello.

```
#!/bin/sh  
echo "Who am I talking to?"  
read user_name  
echo "Hello $user_name"
```

- Try replacing “ with ‘ in the last line to see what happens.

Debugging your shell scripts

- Generous use of the `echo` command will help.
- Run script with the `-x` parameter.
E.g. `sh -x ./myscript`
or `set -o xtrace` before running the script.
- These options can be added to the first line of the script where the shell is defined.
e.g. `#!/bin/sh -xv`

Shell Programming

- Programming features of the UNIX/LINUX shell:
 - **Shell variables**: Your scripts often need to keep values in memory for later use. Shell variables are symbolic names that can access values stored in memory
 - **Operators**: Shell scripts support many operators, including those for performing mathematical operations
 - **Logic structures**: Shell scripts support **sequential logic** (for performing a series of commands), **decision logic** (for branching from one point in a script to another), **looping logic** (for repeating a command several times), and **case logic** (for choosing an action from several possible alternatives)

Variables

- **Variables** are symbolic names that represent values stored in memory
- **Three different types of variables**
 - **Global Variables:** Environment and configuration variables, capitalized, such as **HOME, PATH, SHELL, USERNAME, and PWD.**

When you login, there will be a large number of global System variables that are already defined. These can be freely referenced and used in your shell scripts.

- **Local Variables**

Within a shell script, you can create as many new variables as needed. Any variable created in this manner remains in existence only within that shell.

- **Special Variables**

Reversed for OS, shell programming, etc. such as positional parameters \$0, \$1 ...

A few global (environment) variables

SHELL	Current shell
DISPLAY	Used by X-Windows system to identify the display
HOME	Fully qualified name of your login directory
PATH	Search path for commands
MANPATH	Search path for <man> pages
PS1 & PS2	Primary and Secondary prompt strings
USER	Your login name
TERM	terminal type
PWD	Current working directory

Referencing Variables

Variable contents are accessed using '\$':

e.g. `$ echo $HOME`

`$ echo $SHELL`

To see a list of your **environment variables**:

`$ printenv`

or:

`$ printenv | more`

Defining Local Variables

- As in any other programming language, variables can be defined and used in shell scripts.
- Unlike other programming languages, variables in Shell Scripts are not typed.
- Examples :

`a=1234 # a is NOT an integer, a string instead`

`b=$a+1 # will not perform arithmetic but be the string '1234+1'`

`b=`expr $a + 1 ` will perform arithmetic so b is 1235 now.`

Note : +,-,/,*,**, % operators are available.

`b=abcde # b is string`

`b=' abcde ' # same as above but much safer.`

`b=abc def # will not work unless 'quoted'`

`b='abc def' # i.e. this will work.`

IMPORTANT NOTE: DO NOT LEAVE SPACES AROUND THE =

Variables

- `vi myinputs.sh`
`#!/bin/sh`
`echo Total number of inputs: $#`
`echo First input: $1`
`echo Second input: $2`
- `chmod u+x myinputs.sh`
- `myinputs.sh HUSKER UNL CSE`
Total number of inputs: 3
First input: HUSKER
Second input: UNL

Defining and Evaluating

- A shell variable take on the generalized form **variable=value** (except in the C shell).

```
$ set x=37; echo $x
```

```
37
```

```
$ unset x; echo $x
```

```
x: Undefined variable.
```

- You can set a pathname or a command to a variable or substitute to set the variable.

```
$ set mydir=`pwd`; echo $mydir
```

Arithmetic Operators

- **expr** supports the following operators:
 - arithmetic operators: +, -, *, /, %
 - comparison operators: <, <=, ==, !=, >=, >
 - boolean/logical operators: &, |
 - parentheses: (,)
 - precedence is the same as C, Java

Arithmetic Operators

- **vi math.sh**

```
#!/bin/sh
```

```
count=5
```

```
count=`expr $count + 1`
```

```
echo $count
```

- **chmod u+x math.sh**
- **math.sh**

Arithmetic operations in shell scripts

var++ , var-- , ++var , --var	post/pre increment/decrement
+ , -	add subtract
* , / , %	multiply/divide, remainder
**	power of
! , ~	logical/bitwise negation
& ,	bitwise AND, OR
&&	logical AND, OR

Shell Programming

- programming features of the UNIX shell:
 - *Shell variables*
 - *Operators*
 - *Logic structures*

Shell Logic Structures

The four basic logic structures needed for program development are:

- **Sequential logic:** to execute commands in the order in which they appear in the program
- **Decision logic:** to execute commands only if a certain condition is satisfied
- **Looping logic:** to repeat a series of commands for a given number of times
- **Case logic:** to replace “if then/else if/else” statements when making numerous comparisons

Conditional Statements (if constructs)

The most general form of the if construct is;

```
if command executes successfully
then
    execute command
elif this command executes successfully
then
    execute this command
    and execute this command
else
    execute default command
fi
```

However- elif and/or else clause can be omitted.

Examples

SIMPLE EXAMPLE:

```
if date | grep "Fri"  
then  
    echo "It's Friday!"  
fi
```

FULL EXAMPLE:

```
if [ "$1" == "Monday" ]  
then  
    echo "The typed argument is Monday."  
elif [ "$1" == "Tuesday" ]  
then  
    echo "Typed argument is Tuesday"  
else  
    echo "Typed argument is neither Monday nor Tuesday"  
fi
```

Note: = or == will both work in the test but == is better for readability.

Loops

Loop is a block of code that is repeated a number of times.

The repeating is performed either a pre-determined number of times determined by a list of items in the loop count (**for loops**) or until a particular condition is satisfied (**while** and **until loops**)

To provide flexibility to the loop constructs there are also two statements namely **break** and **continue** are provided.

for loops

Syntax:

```
for arg in list
do
    command(s)
    ...
done
```

Where the value of the variable ***arg*** is set to the values provided in the list one at a time and the block of statements executed. This is repeated until the list is exhausted.

Example:

```
for i in 3 2 5 7
do
    echo " $i times 5 is $(( $i * 5 )) "
done
```


The while Loop

- A different pattern for looping is created using the **while** statement
- The **while statement** best illustrates how to set up a loop to test repeatedly for a matching condition
- The while loop tests an expression in a manner similar to the if statement
- **As long as the statement inside the brackets is true, the statements inside the do and done statements repeat**

while loops

Syntax:

```
while this_command_execute_successfully
do
    this command
    and this command
done
```

EXAMPLE:

```
while test "$i" -gt 0    # can also be while [ $i > 0 ]
do
    i=`expr $i - 1`
done
```

Looping Logic

- Example:

```
#!/bin/sh
for person in Bob Susan Joe Gerry
do
    echo Hello $person
done
```

Output:

```
Hello Bob
Hello Susan
Hello Joe
Hello Gerry
```

- Adding integers from 1 to 10

```
#!/bin/sh
i=1
sum=0
while [ "$i" -le 10 ]
do
    echo Adding $i into the sum.
    sum=`expr $sum + $i `
    i=`expr $i + 1 `
done
echo The sum is $sum.
```

Switch/Case Logic

- The **switch logic** structure simplifies the selection of a match when you have a list of choices
- It allows your program to perform one of many actions, depending upon the value of a variable

Case statements

The case structure compares a string ‘usually contained in a variable’ to one or more patterns and executes a block of code associated with the matching pattern. Matching-tests start with the first pattern and the subsequent patterns are tested only if no match is not found so far.

case argument in

pattern 1) execute this command

and this

and this;;

pattern 2) execute this command

and this

and this;;

esac

Take-Home Message

- Shell script is a **high-level language** that must be converted into a **low-level (machine) language** by UNIX Shell before the computer can execute it
- UNIX shell scripts, created with the vi or other text editor, contain two key ingredients: a selection of **UNIX commands** glued together by Shell **programming syntax**
- UNIX/Linux shells are derived from the UNIX **Bourne, Korn,** and **C/TCSH** shells
- UNIX keeps three types of variables:
 - **Configuration; environmental; local**
- The shell supports numerous operators, including many for performing arithmetic operations
- The logic structures supported by the shell are **sequential, decision, looping,** and **case**

To Script or Not to Script

- Pros
 - File processing
 - Glue together compelling, customized testing utilities
 - Create powerful, tailor-made manufacturing tools
 - Cross-platform support
 - Custom testing and debugging
- Cons
 - Performance slowdown
 - Accurate scientific computing

Reference Books



- **Class Shell Scripting**
<http://oreilly.com/catalog/9780596005955/>
- **LINUX Shell Scripting With Bash**
<http://ebooks.ebookmall.com/title/linux-shell-scripting-with-bash-burtch-ebooks.htm>
- **Shell Script in C Shell**
<http://www.grymoire.com/Unix/CshTop10.txt>
- **Linux Shell Scripting Tutorial**
<http://www.freeos.com/guides/lsst/>
- **Bash Shell Programming in Linux**
http://www.arachnoid.com/linux/shell_programming.html
- **Advanced Bash-Scripting Guide**
<http://tldp.org/LDP/abs/html/>
- **Unix Shell Programming**
<http://ebooks.ebookmall.com/title/unix-shell-programming-kochan-wood-ebooks.htm>

