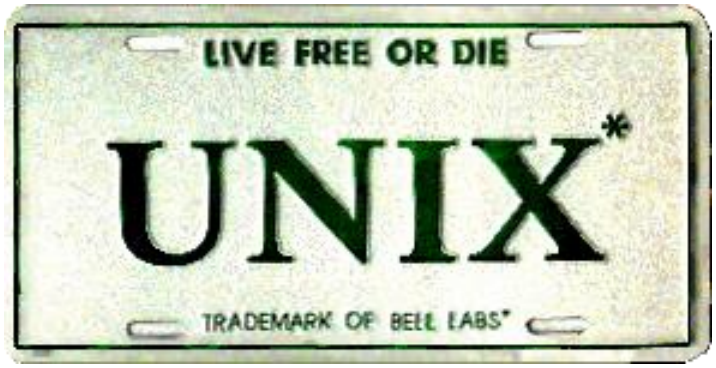

UNIX for Programmers and Users

Warning !

- The shell programming language does not type-cast its variables. This means that a variable can hold number data or character data.

```
count=0
count=Sunday
```

- Switching the TYPE of a variable can lead to confusion for the writer of the script or someone trying to modify it.

- So it is recommended to use a variable for only a single TYPE of data in a script.

- **QUOTING**

- There are often times when you want to inhibit the shell's wildcard-replacement, variable-substitution, and/or command-substitution mechanisms.

  The shell's quoting system allows you to do just that.

- Here's the way that it works:

  1) Single quotes(') inhibit wildcard replacement, variable substitution, and command substitution.

  2) Double quotes(") inhibit wildcard replacement only.

  3) When quotes are nested, it's only the outer quotes that have any effect.

- **QUOTING**

-Examples:

```
$ echo  3 * 4 = 12    ---> remember, * is a wildcard.
  3   a.c    b    b.c    c.c    4 = 12
$ echo  "3 * 4 = 12" ---> double quotes inhibit wildcards.
  3 * 4 = 12
$ echo  '3 * 4 = 12'  ---> single quotes inhibit wildcards.
  3 * 4 = 12

$ name=Graham
$ echo  'my name is $name - date is `date`'
  my name is $name - date is 'date'

$ echo  "my name is $name - date is `date`"
  my name is Graham - date is Wed, Aug 24, 2016  7:38:55 AM
$ -
```

- **QUOTING**

-Examples:

```
$ echo  3 * 4 = 12 $USER `date`

$ echo  “3 * 4 = 12 $USER `date`”

$ echo  ‘3 * 4 = 12 $USER `date`’

$ echo  ‘3 * 4 = 12 “$USER” `date`’

$ echo  “3 * 4 = 12 ‘$USER’ `date`”
```

Command Substitution

- The backquote “`” is different from the single quote “´ ”. It is used for command substitution:

$ LIST=`ls`

$ echo $LIST

hello.bash read.bash

$ PS1=“`pwd`---->”

/home/SRD---->

- We can also perform the command substitution by means of $(command)

$ LIST=$(ls)

$ echo $LIST

hello.bash read.bash

- **JOB CONTROL**

- Convenient multitasking is one of UNIX's best features, so it's important to be able to obtain a ***listing of the current processes*** and to ***control their behavior***.

1) **ps**, which generates a list of processes and their attributes, including their names, process ID numbers, controlling terminals and owner.

2) **kill**, which allows to terminate a process based on its ID number.

Utility : ps
ps -e

ps generates a listing of process-status information.

The -e option instructs ps to include all running processes.

Utility : sleep seconds

The sleep utility sleeps for the specified number of seconds and then terminates.

```
$ ( sleep 10; echo done ) &    ---> delayed echo in background.
27387                           ---> the process ID number.
$ ps
PID    TTY   TIME CMD
27355  pts/3  0:00  bash          ---> the long shell.
27387  pts/3  0:00  bash          ---> the subshell.
27388  pts/3  0:00  sleep 10      ---> the sleep.
27389  pts/3  0:00  ps            ---> the ps command itself!
$ done                    ---> the output from the background process.
```

The meaning of the common column headings of ps output:

Column	Meaning
PID	the ID of the process
TTY	the controlling terminal
TIME	Amount of CPU Time
CMD	the name of the command

- **Signaling Processes: kill**

- ***kill*** command terminates a process before it completes.

kill [-signalId] {pid}
kill -l

- kill sends the signal with code signalId to the list of processes.

- signalId may be the number or name of a signal.

- By default, ***kill*** sends a TERM signal (number 15),
  which causes the receiving processes to terminate.

- To send a signal to a process,
  you must either own it or be a super-user.

- To ensure a kill (forcefully), send signal number 9.

```
$ sleep 1000 & sleep 1000 & sleep 1000 & ---> create three process
[1] 16245
[2] 16246
[3] 16247

$ ps
PID   TTY     TIME    CMD
15705 pts/4    00:00:00 bash
16245 pts/4    00:00:00 sleep
16246 pts/4    00:00:00 sleep
16247 pts/4    00:00:00 sleep
16249 pts/4    00:00:00 ps

$ kill  16245                              ---> kill first sleep.
$ ps
PID   TTY     TIME    CMD
15705 pts/4    00:00:00 bash
16246 pts/4    00:00:00 sleep
16247 pts/4    00:00:00 sleep
16265 pts/4    00:00:00 ps
[1]   Terminated              sleep 1000
```

- **OVERLOADING STANDARD UTILITIES**

```
$ cat > ls          ---> create a script called "ls".
echo my ls
^D                  ---> end of input.
$ chmod +x  ls  ---> make it executable.

$ echo $PATH        ---> look at the current PATH setting.
/bin:/usr/bin:/usr/sbin
$ echo $HOME        ---> get pathname of my home directory.
/home/UG1

$ PATH=/home/UG1:$PATH    ---> update.
$ ls        --->  call "ls".
my ls      ---> my own version overrides "/bin/ls".
$ _
```

Note that only this shell and its child shells would be affected by the change to PATH; all other shells would be unaffected.

- *Eval BUILT-IN COMMAND*

The *eval* shell command executes the output of a command as a regular shell command.

It is useful for processing the output of utilities that generate shell commands.

-Example: execute the result of *echo* command:

```
$ echo  x=5                    ---> first execute an echo directly.
x=5

$ echo $x

$ eval  `echo x=5`             ---> execute the result of the echo.

$ echo $x
5
$ _
```

Paste: merge lines of files

$ paste file1 file2

Use any delimiter such as ‘-’ in between:

$ paste –d - file1 file2

gzip: compressing the files

gzip – Reduce the size of a file.

$ ls –l new.txt

-rw-r--r-- 1 shivram_2 None 85 Aug 27 15:37 new.txt

$ gzip new.txt

$ ls –l new.txt.gz

-rw-r--r-- 1 shivram_2 None 68 Aug 27 15:37 new.txt.gz

$ gunzip new.txt.gz #---- To expand the file

gzip: compressing multiple files

gzip – Reduce the size of a file.

$ tar -cf file.tar f1 f2 f3 #---- First combine the files

$ gzip file.tar #---- compress the file

$ gunzip file.tar.gz #---- Decompress

$ tar -xf file.tar #---- To extract the files