Unix Shell Programming

Basic Shell Programming

- A script is a file that contains shell commands
  - data structure: variables
  - control structure: sequence, decision, loop
- Shebang line for bash shell script:

  `#! /bin/bash`

  `#! /bin/sh`
- to run:
  - make executable: **`% chmod +x script`**
  - invoke via: **`% ./script.sh`**

Hello World Example

- vi first.sh
- Include the statement '*echo hello world*'
- Run ./first.sh

Bash shell programming

- Input
  - prompting user
  - command line arguments
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select
- Functions
- Traps

User input

- shell allows to prompt for user input

Syntax:

```
read varname [more vars]
```

- or

```
read –p "prompt" varname [more vars]
```

- words entered by user are assigned to **`varname`** and "**`more vars`**"
- last variable gets rest of input line

User input example

```
#! /bin/sh
read -p "enter your name: "
  first last

echo "First name: $first"
echo "Last name: $last"
```

Special shell variables

Parameter	Meaning
$0	Name of the current shell script
$1-$9	Positional parameters 1 through 9
$#	The number of positional parameters
$*	All positional parameters, “$*” is one string
$@	All positional parameters, “$@” is a set of strings
$?	Return status of most recently executed command
$$	Process id of current process

Examples: Command Line Arguments

```
% set tim bill ann fred
      $1  $2   $3  $4
% echo $*
tim bill ann fred
% echo $#
4
% echo $1
tim
% echo $3 $4
ann fred
```

The ‘set’ command can be used to assign values to positional parameters

bash control structures

- if-then-else
- case
- loops
  - for
  - while
  - until
  - select

if statement

```
if command
then
  statements
fi
```

- statements are executed only if **`command`** succeeds, i.e. has return status “0”

test command

Syntax:

```
test expression
[ expression ]
```

- evaluates ‘expression’ and returns true or false

Example:

```
if test -w "$1"
  then
  echo "file $1 is write-able"
fi
```

The simple if statement

```
if [ condition ]; then
  statements
fi
```

- executes the statements only if **`condition`** is true

The if-then-else statement

```
if [ condition ]; then
   statements-1
else
  statements-2
fi
```

- executes statements-1 if condition is true
- executes statements-2 if condition is false

THE IF…STATEMENT

```
if [ condition ]; then
    statements
elif [ condition ]; then
    statement
else
    statements
fi
```

- The word **`elif`** stands for “else if”
- It is part of the if statement and cannot be used by itself

Relational Operators

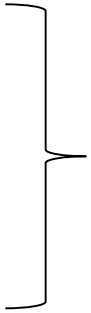
Meaning	Numeric	String
Greater than	-gt	
Greater than or equal	-ge	
Less than	-lt	
Less than or equal	-le	
Equal	-eg	= or ==
Not equal	-ne	!=
str1 is less than str2		str1 < str2
str1 is greater str2		str1 > str2
String length is greater than zero		-n str
String length is zero		-z str

Compound logical expressions

! not

&& and

|| or



and, or
must be enclosed within

[[ ]]

Example: Using the ! Operator

```
#!/bin/bash

read -p "Enter years of work: "
  Years
if [ ! "$Years" -lt 20 ]; then
    echo "You can retire now."
else
    echo "You need 20+ years to
  retire"
fi
```

Example: Using the && Operator

```
#!/bin/bash

Bonus=500
read -p "Enter Status: " Status
read -p "Enter Shift: " Shift
if [[ "$Status" = "H" && "$Shift" = 3 ]]
then
   echo "shift $Shift gets \$$Bonus bonus"
else
   echo "only hourly workers in"
   echo "shift 3 get a bonus"
fi
```

Example: Using the || Operator

```
#!/bin/bash

read -p "Enter calls handled:" CHandle
read -p "Enter calls closed: " CClose
if [[ "$CHandle" -gt 150 || "$CClose" -gt 50 ]]
   then
   echo "You are entitled to a bonus"
else
   echo "You get a bonus if the calls"
   echo "handled exceeds 150 or"
   echo "calls closed exceeds 50"
fi
```

File Testing

	Meaning
-d file	True if ‘file’ is a directory
-f file	True if ‘file’ is an ord. file
-r file	True if ‘file’ is readable
-w file	True if ‘file’ is writable
-x file	True if ‘file’ is executable
-s file	True if length of ‘file’ is nonzero

Example: File Testing

```
#!/bin/bash
echo "Enter a filename: "
read filename
if [ ! –r "$filename" ]
 then
   echo "File is not read-able"
 exit 1
fi
```

Example: File Testing

```
#! /bin/bash

if [ $# -lt 1 ]; then
        echo ”Not enter filename"
        exit 1
fi
if [[ ! -f "$1" || ! -r "$1" || ! -w "$1" ]]
then
  echo "File $1 is not accessible"
  exit 1
fi
```

EXAMPLE: IF... STATEMENT

```
# The following THREE if-conditions produce the same result

* DOUBLE SQUARE BRACKETS
read -p "Do you want to continue?" reply
if [[ $reply = "y" ]]; then
   echo "You entered " $reply
fi

* SINGLE SQUARE BRACKETS
read -p "Do you want to continue?" reply
if [ $reply = "y" ]; then
   echo "You entered " $reply
fi

* "TEST" COMMAND
read -p "Do you want to continue?" reply
if test $reply = "y"; then
   echo "You entered " $reply
fi
```

Example: if..elif... Statement

```
#!/bin/bash

read -p "Enter Income Amount: " Income
read -p "Enter Expenses Amount: " Expense

let Net=$Income-$Expense

if [ "$Net" -eq "0" ]; then
   echo "Income and Expenses are equal - breakeven."
elif [ "$Net" -gt "0" ]; then
   echo "Profit of: " $Net
else
   echo "Loss of: " $Net
fi
```

The case Statement

- use the case statement for a decision that is based on multiple choices

Syntax:

```
case word in
 pattern1) command-list1
 ;;
 pattern2) command-list2
 ;;
 patternN) command-listN
 ;;
esac
```

case pattern

- checked against word for match
- may also contain:

  `*`

  `?`

  `[ … ]`

  `[:class:]`
- multiple patterns can be listed via:

  `|`

Example 1: The case Statement

```
#!/bin/bash
echo "Enter Y to see all files including hidden files"
echo "Enter N to see all non-hidden files"
echo "Enter q to quit"

read -p "Enter your choice: " reply

case $reply in
  Y|YES) echo "Displaying all (really…) files"
         ls -a ;;
  N|NO)  echo "Display all non-hidden files..."
         ls ;;
  Q)     exit 0 ;;

  *) echo "Invalid choice!"; exit 1 ;;
esac
```

Example 2: The case Statement

```
#!/bin/bash
ChildRate=3
AdultRate=10
SeniorRate=7
read -p "Enter your age: " age
case $age in
  [1-9]|[1][0-2])   # child, if age 12 and younger
     echo "your rate is" '$'"$ChildRate.00" ;;
   # adult, if age is between 13 and 59 inclusive
  [1][3-9]|[2-5][0-9])
     echo "your rate is" '$'"$AdultRate.00" ;;
  [6-9][0-9])       # senior, if age is 60+
     echo "your rate is" '$'"$SeniorRate.00" ;;
esac
```

Example 3: The case Statement

```
DEPARTMENT=("Electronics and Communication", "Computer Science", "Information Technology")
for value in "${DEPARTMENT[@]}" do
case $value in    "Computer Science")
echo -n "Computer Science "     ;;
"Electrical and Electronics Engineering" | "Electrical Engineering")
echo -n "Electrical and Electronics Engineering or Electrical Engineering "     ;;    "Information Technology" | "Electronics and Communication")
 echo -n "Information Technology or Electronics and Communication "
;;
*)     echo -n "Invalid "     ;;
esac
done
```

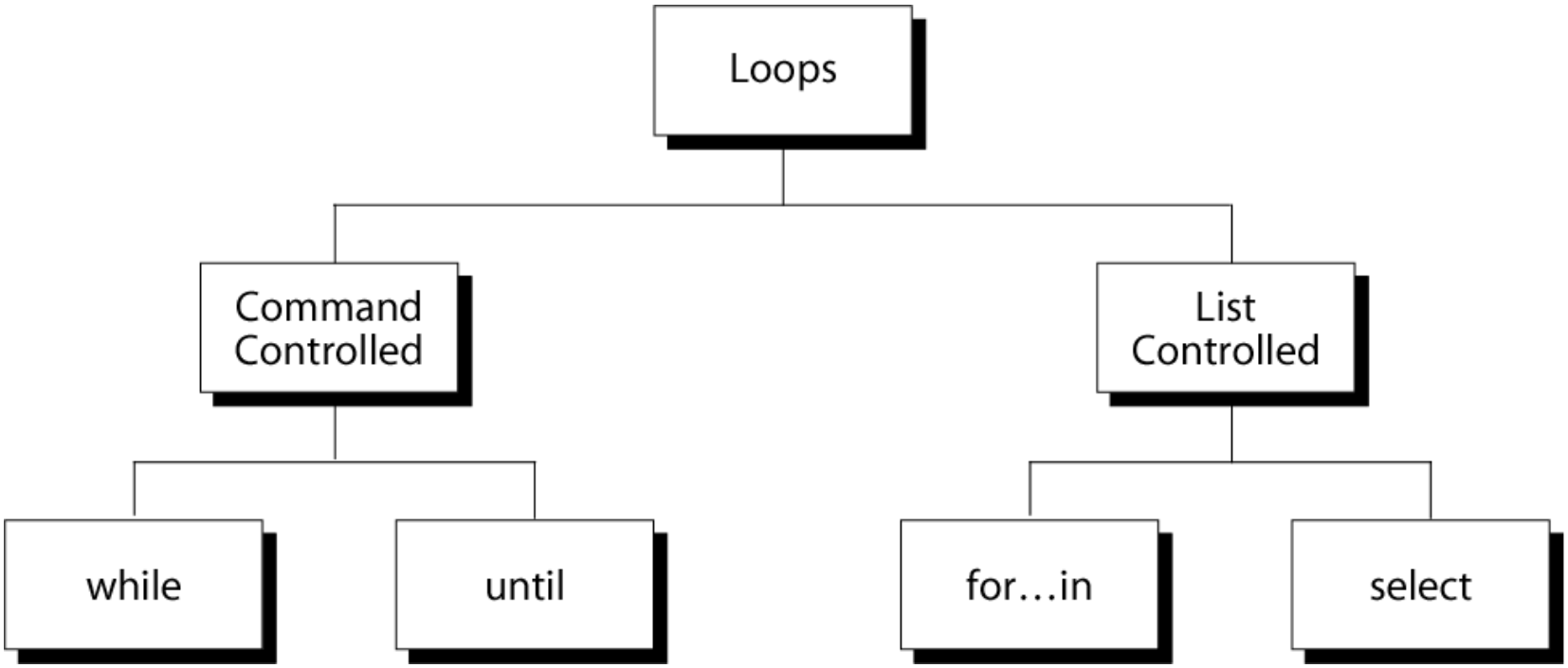
Bash programming: so far

- Data structure
  - Variables
  - Numeric variables
  - Arrays
- User input
- Control structures
  - if-then-else
  - case

Bash programming: still to come

- Control structures
  - Repetition
    - do-while, repeat-until
    - for
    - select
- Functions
- Trapping signals

Repetition Constructs



The while Loop

- Purpose:

  To execute commands in “command-list” as long as “expression” evaluates to true

Syntax:

```
while [ expression ]
do
 command-list
done
```

Example: Using the while Loop

```
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 10 ]
do
    echo The counter is $COUNTER
    let COUNTER=$COUNTER+1
done
```

Example: Using the while Loop

```
#!/bin/bash

Cont="Y"
while [ $Cont = "Y" ]; do
  ps -A #Display current process IDs
  read -p "want to continue? (Y/N)" reply
  Cont=`echo $reply | tr [:lower:] [:upper:]`
  #tr: translate from lower to upper
done
echo "done"
```

Activity: Using the while Loop

```
#!/bin/bash
# copies files from home- into the webserver- directory
# A new directory is created every hour

PICSDIR=/home/carol/pics
WEBDIR=/var/www/carol/webcam
while true; do
   DATE=`date +%Y%m%d`
   HOUR=`date +%H`
   mkdir $WEBDIR/"$DATE"
   while [ $HOUR -ne "00" ]; do
      DESTDIR=$WEBDIR/"$DATE"/"$HOUR"
      mkdir "$DESTDIR"
      mv $PICSDIR/*.jpg "$DESTDIR"/
      sleep 3600
      HOUR=`date +%H`
   done
done
```

The until Loop

- Purpose:

  To execute commands in “command-list” as long as “expression” evaluates to false

Syntax:

```
until [ expression ]
do
 command-list
done
```

Example: Using the until Loop

```
#!/bin/bash

COUNTER=20
until [ $COUNTER -lt 10 ]
do
    echo $COUNTER
    let COUNTER-=1
done
```

Example: Using the until Loop

```
#!/bin/bash

Stop="N"
until [ $Stop = "Y" ]; do
  ps -A
  read -p "want to stop? (Y/N)" reply
  Stop=`echo $reply | tr [:lower:] [:upper:]`
done
echo "done"
```

The for Loop

- Purpose:

  To execute commands as many times as the number of words in the “argument-list”

Syntax:

```
for variable in argument-list
do
 commands
done
```

Example 1: The for Loop

```
#!/bin/bash

for i in 7 9 2 3 4 5
do
    echo $i
done
```

Example 2: Using the for Loop

```
#!/bin/bash
# compute the average weekly temperature

for num in 1 2 3 4 5 6 7
do
   read -p "Enter temp for day $num: "
  Temp
   let TempTotal=$TempTotal+$Temp
done

let AvgTemp=$TempTotal/7
echo "Average temperature: " $AvgTemp
```

Select command

- Constructs simple menu from word list
- Allows user to enter a number instead of a word
- User enters sequence number corresponding to the word

Syntax:

```
select WORD in LIST
do
 RESPECTIVE-COMMANDS
done
```

- Loops until end of input, i.e. ^d (or ^c)

Select example

```
#! /bin/bash
select var in alpha beta gamma
do
        echo $var
done
```

- Prints:

```
1) alpha
2) beta
3) gamma
#? 2
beta
#? 4
#? 1
alpha
```

Select example

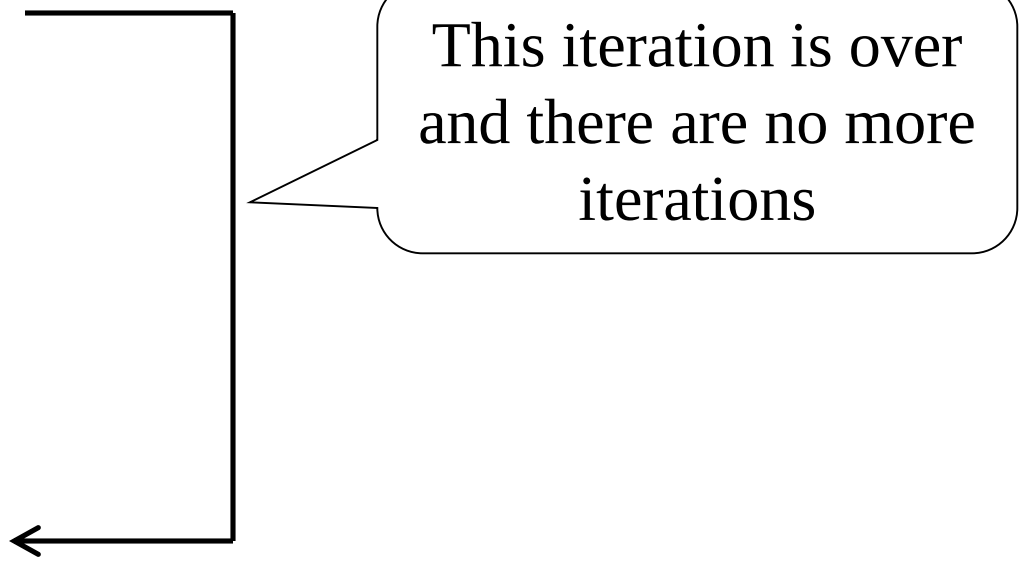
```
#!/bin/bash
echo "script to make files private"
echo "Select file to protect:"

select FILENAME in *
do
  echo "You picked $FILENAME ($REPLY)"
  chmod go-rwx "$FILENAME"
  echo "it is now private"
done
```

break and continue

- Interrupt for, while or until loop
- The break statement
  - transfer control to the statement AFTER the done statement
  - terminate execution of the loop
- The continue statement
  - transfer control to the statement TO the done statement
  - skip the test statements for the current iteration
  - continues execution of the loop

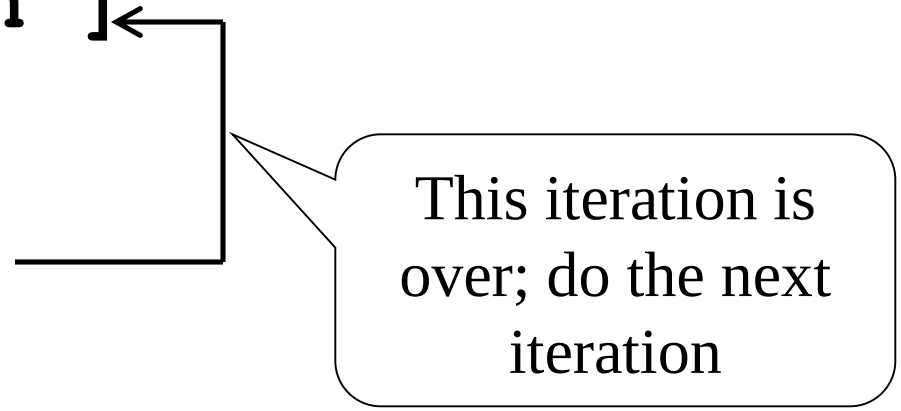
The break command

```
while [ condition ]
do
      cmd-1
      break
      cmd-n
done
echo "done"
```



The continue command

```
while [ condition ]
do
     cmd-1
     continue
     cmd-n
done
echo "done"
```



Example:

```
for index in 1 2 3 4 5 6 7 8 9 10
do
        if [ $index -le 3 ]; then
             echo "continue"
             continue
        fi
        echo $index
        if   [ $index -ge 8 ]; then
             echo "break"
             break
        fi
done
```

Bash shell programming

- Sequence
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select

DONE !

- Functions
- Traps

still to come

Shell Functions

- A shell function is similar to a shell script
  - stores a series of commands for execution later
  - shell stores functions in memory
  - shell executes a shell function in the same shell that called it
- Where to define
  - In .profile
  - In your script
  - Or on the command line
- Remove a function
  - Use unset built-in

Shell Functions

- must be defined before they can be referenced
- usually placed at the beginning of the script

Syntax:

```
function-name () {
    statements
}
```

Example: function

```
#!/bin/bash

funky () {
  # This is a simple function
  echo "This is a funky function."
  echo "Now exiting funky function."
}

# declaration must precede call:

funky
```

Example: function

```
#!/bin/bash
fun () { # A somewhat more complex function.
  JUST_A_SECOND=1
  let i=0
  REPEATS=30
  echo "And now the fun really begins."
  while [ $i -lt $REPEATS ]
  do
    echo "-------FUNCTIONS are fun-------->"
    sleep $JUST_A_SECOND
    let i+=1
  done
}
fun
```

Function parameters

- Need not be declared
- Arguments provided via function call are accessible inside function as $1, $2, $3, …

$# reflects number of parameters

$0 still contains name of script
(not name of function)

Example: function with parameter

```
#! /bin/sh
testfile() {
  if [ $# -gt 0 ]; then
     if [ $1 > 0 ]; then
        echo $1 is a positive number
     else
        echo $1 is a negative number
     fi
  fi
}

testfile 10
testfile -10
```

Example: function with parameters

```
#! /bin/bash
checktotal() {
count = $1
   while [ count -lt 100 ]
do
     let tot = tot + count
     let count+=1
done
 echo Total is $tot
}
Checktotal 10
```

Local Variables in Functions

- Variables defined within functions are global,

  i.e. their values are known throughout the entire shell program

- keyword “local” inside a function definition makes referenced variables “local” to that function

Example: function

```
#! /bin/bash

global="pretty good variable"

foo () {
        local inside="not so good variable"
        echo $global
        echo $inside
        global="better variable"
}

echo $global
foo
echo $global
echo $inside
```

Handling signals

- Unix allows you to send a signal to any process

- -1 = hangup **`kill -HUP 1234`**
- -2 = interrupt with ^C **`kill -2 1235`**
- no argument = terminate **`kill 1235`**
- -9 = kill **`kill -9 1236`**
  - -9 cannot be blocked

- list your processes with
  `ps -u userid`

Signals on Linux

```
% kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

- ^C is 2 - SIGINT

Handling signals

- Default action for most signals is to end process
  - term: signal handler

- Bash allows to install custom signal handler

Syntax:

```
trap 'handler commands' signals
```

Example:

```
trap 'echo do not hangup'  1 2
```

Example: trap hangup

```
#! /bin/bash
# kill -1 won't kill this process
# kill -2 will

trap 'echo dont hang up' 2

while true
do
        echo "try to hang up"
        sleep 1
done
```

Example: trap multiple signals

```
#! /bin/sh
# plain kill or kill -9 will kill
  this
trap 'echo 1' 1
trap 'echo 2' 2

while true; do
    echo -n .
    sleep 1
done
```

Debug Shell Programs

- Debugging is troubleshooting errors that may occur during the execution of a program/script
- The following two commands can help you debug a bash shell script:
  - echo

    use explicit output statements to trace execution
  - set

DEBUGGING USING "SET"

- The "set" command is a shell built-in command
- has options to allow flow of execution
  - –v option prints each line as it is read
  - –x option displays the command and its arguments
  - –n checks for syntax errors
- options can turned on or off
  - To turn on the option: set -xv
  - To turn off the options: set +xv

- Options can also be set via she-bang line

```
#! /bin/bash -xv
```

Summary: Bash shell programming

- Sequence
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select
- Functions
- Traps

DONE !

File Manupulation

- > -output redirect symbol
- >> -append
- Syntax: command > file name
- Eg: echo “first text” > file_name
  - date > file_name
  - ls –l > file_name

  see the content of the file
    - more file_name

- Eg: following command save the current user of the given host and his activities at the current session
  - echo A log file by $USER on $HOSTNAME >> demo.txt

Reading from a Text File

```
input= “demo.text”
While    read     var
Do
   echo         $var
Done < $input
```

```
while IFS= read –r var
do
 echo $var

done < $input
```

IFS - trims begining and end white spaces

-r – prevents adding / and escape

Assignment -01

- 1. Write a program to Prepare a text file containing 1000 integers
- 2. Write another program to read the content of the above text file into an integer array. Find all the odd numbers in that array.

grep- command

- Searches a file or files for lines that have a certain pattern
  - grep “aug” filename
- grep without filename waits for input from the key board
- Eg: grep “aug”
  - When a line with the word “aug” input, it prints the same line

Sort command

- Sorts the lines of a text file
- Eg: sort filename
- Options
  - -n numerical order
  - -r revers order
  - -f upper lower cases
  - +x ignore the first x fields when sorting files
  - sort –n filename

Pipes

- Connect two or more commands so that output of one command becomes an input to the other command
- Eg: ls –l | grep “aug” | sort +4n

Assignment-02

Create a text file with lots of lines that contain the word GNU and one line that contains the word GNU as well as the word Linux.

Then extract only the lines with the word Linux.

Store that line in another file myfile.txt.

Typeset / Declare Variables

- Permits modifying the properties of variables
- Syntax
  - declare –options variable_name
  - Options
    - -r read only
    - -I integer
    - -a array
    - -f function // list the function

Diff- command

- Check the difference between two files
- Syntax
  - diff –u <old-file> <new-file>

- Also check the difference between two directories
  - diff –u dir1 dir2

sed - command

- Replace original text with a new text
- Syntax:
  - sed –e ‘s/original_text/new_text/g’ file_name
  - s – replace command
  - g – all the words in the file
- Insert lines into a text file
  - Sed –e ‘7a\
    an extra line.\n
    One more line.’ filename

bc- calculator

- Use to perform arithmetic calculation
- Eg: echo 6.7 / 3.2 | bc
  - Output is 2. No decimal places
  - echo ‘scale = 3; 6.7 / 3.2’ | bc
  - echo 2 ^ 3 | bc

Assignment-03

- Write a function that accepts two values.
- Convert them into integers
- Find the power of the first integer to the second.

email

- Sending and receiving emails
- Email stored in /var/spool/mail/<username>
- To send a mail to a user in the same machine
  - mail –s " Hello There" <user name>
  - Mail program is waiting your message
  - When you finish typing the message then get into a new line and type . to end the message and send it

email

- To read mails type *mail*
- Type the number of the mail, which you need to read.
- Delete a mail
  - *delete <mail_number>*
- Reply a mail
  - *reply <mail_number>*
- Type *quit* to exit the mail reader

Netcat

- Netcat is a terminal application that is similar to the telnet program but has lot more features
- Netcat can be used to fetch a webpage
  - Eg: $ nc –v google.com 80
  - And then type GET / HTTP/1.O and press enter key twice
- Open a single socket server
  - nc –l –v 1234 or telnet
  - nc localhost 1234 or telnet

Assignment- 04

- Write a shell script to read an address of a webpage from the keyboard and save the content of that page in a text file. Then find the lines, which contains a given regular expression and store these lines in a separate file. The above task is iterated until the scrip is terminated by pressing Control + Z.

FTP

- FTP stands for *File Transfer Protocol*
- Used to transfer large files
- Eg: ftp metalab.unc.edu
  - password anything with @ sign
- FTP commands
  - ls –l
  - cd
  - get filename (read) or put filename (write)
  - quit

Sending file by email

- tar -czf - <mydir> | uuencode <mydir>.tar.gz \
  | mail -s "Here are some files"
  <user>@<machine>

A Sample TCP Session

- Find the host address
  - host cnn.com
- Display content of the index.html
  - nc or telnet

More communication commands

- Ipcalc / nslookup / dig
  - Displays IP information for a host. With the -h option, **ipcalc** does a reverse DNS lookup, finding the name of the host (server) from the IP address
  - Some interesting options to *dig* are +time=N for setting a query timeout to *N* seconds, +nofail for continuing to query servers until a reply is received, and-x for doing a reverse address lookup

Communication command contd..

- **traceroute**
  - Trace the route taken by packets sent to a remote host. This command works within a LAN, WAN, or over the Internet. The remote host may be specified by an IP address. The output of this command may be filtered by grep or sed in a pipe.
- ping
  - Broadcast an *ICMP ECHO_REQUEST* packet to another machine, either on a local or remote network.
  - ping –c 10 uwu.ac.lk #only 10 pings sent

Communication command contd..

- write
- This is a utility for terminal-to-terminal communication. It allows sending lines from your terminal (console or *xterm*) to that of another user.

Wall- (write to all) command

- $ echo message here | wall
- $ wall
  - Wait until you enter from the std in
- $ wall <<< ‘ message here’

Here Documents

- A way of getting text input into a script without having separate text file.
  - Eg: cat << EOF

    Current directory is :

    $PWD

  EOF

Alias

- **Define shortcuts for long commands**
  - **alias lm="ls -l | more”**
- lm replaces the long command of ls

And List and Or List

- command-1 && command-2 && command-3 && ... command-n
- Each command executes in turn, provided that the previous command has given a return value of *true* (zero). At the first *false* (non-zero) return, the command chain terminates (the first command returning *false* is the last one to execute).

These can effectively replace complex nested if/then or even case statements

Eg: and list

```
if [ ! -z "$1" ] && echo "Argument #1 = $1" && [ ! -z "$2" ] &&
echo "Argument #2 = $2"
then
  echo "At least 2 arguments passed to script."
  # All the chained commands return true.
else
  echo "Fewer than 2 arguments passed to script."
  # At least one of the chained commands returns false.
fi
```

Or list

- command-1 || command-2 || command-3 || ... command-n
- Each command executes in turn for as long as the previous command returns false. At the first true return, the command chain terminates (the first command returning true is the last one to execute). This is obviously the inverse of the "and list".

String Manipulation

- Length of a string
  - ${#string}
- Substrings
  - ${string:position}
  - ${string:position:length}
- Shortest substring match
  - echo ${string#substring}
  - #deletes the shortest match of $substring from front of $string
  - echo ${string%substring} #deletes back of the substring

Longest Substring Match

- ${string##substring}
- ${string%%substring}
- Find and replace patterns
  - ${string/pattern/replacement}
  - ${string//pattern/replacement} # replace all the matches
  - ${string/#pattern/replacement} #replace beginning
  - ${string/%pattern/replacement} #replace end

String converts into an array

```
str="My name is sugath"
st=($str)
echo ${st[@]}
cnt=0
check(){
for el in "${st[@]}"; do
        if [ "$el" == "$1" ]; then
        echo "$cnt”
        else
        let cnt+=1
fi
done
}
check name
```

Math Commands

- bc calculator
- Bash cannot handle floating point numbers hence pass the values to bc caculator
  - **variable=$(echo "OPTIONS; OPERATIONS" | bc)**
  - Variable=$(echo “scale=3; 2/10” | bc)

Invoking bc using here documents

```
    val= bc << EOF
    scale=2
    2.5*3.7
    EOF
echo $val
```

Eg: 01

```
v1=23.53
v2=17.881
v3=83.501
v4=171.63
var2=$(bc << EOF
        scale = 4
        a = ( $v1 + $v2 )
        b = ( $v3 * $v4 )
        a * b + 15.35
EOF )
echo $var2
```

Eg: 02

```
var3=$(bc -l << EOF
scale = 9
s ( 1.7 ) EOF )
 # Returns the sine of 1.7 radians.
# The "-l" option calls the 'bc' math library.
echo $var3
```

Eg: 03

```
# Now, try it in a function...
hypotenuse (){
# Calculate hypotenuse of a right triangle.
hyp=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF )
echo $hyp
}
# Can't directly return floating point values from a Bash function.
# But, can echo-and-capture: echo "$hyp”
hyp=$(hypotenuse 3.68 7.31)
echo "hypotenuse = $hyp"
```

dc -Calculator

- The dc (desk calculator) utility is stack-oriented
- Like bc, it has much of the power of a programming language.
- Similar to the procedure with bc, echo a command-string to dc.

Eg:

```
echo "[Printing a string ... ]P" | dc
# The P command prints the string between the preceding brackets.
# And now for some simple arithmetic.
echo "7 8 * p" | dc
#  Pushes 7, then 8 onto the stack,
# multiplies ("*" operator), then prints the result ("p" operator).
```

Eg: Hexadecimal number conversion

```
hexcvt ()
{
echo “$1 16 o p" | dc

#                o    sets radix (numerical base) of output.
#                  p  prints the top of stack.
# For other options: 'man dc' ...
return
}
hexcvt 18
```

awk

- Yet another way of doing floating point math in a script is using awk's built-in math functions

Eg: awk

```
awkscript(){
AWKSCRIPT=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
#  command(s) / parameters passed to awk
 # Now, pipe the parameters to awk.
echo $1  $2 | awk "$AWKSCRIPT”}
# An echo-and-pipe is an easy way of passing shell parameters to awk.
awkscript 10 12
```

Graphing data from shell scripting

- Gnuplot –command line tool for generating graphs
  - gnuplot –persist –e “plot sin(x)”
  - gnuplot –persist –e “plot [-5:5] sin(x)” #specify the range of x
  - gnuplot –persist –e “set terminal dumb; plot sin(x)”
  - gnuplot –persist –e “plot ‘file_name’ ” #plot data from a file

Gnuplot - Contd..

- gnuplot -persist –e “set terminal png; set title ‘Title Name’; set xlabel ‘x- label’; set ylabel ‘y-label’ ; set style fill solid 0.3; set style data box; plot ‘file_name’ “ > out.png

Gnuplot- Contd..

- Putting all the parameters into a script

```
set terminal png
    set title ‘ Title’
    set xlabel ‘x-Label’
    set ylabel ‘y-Label’
    set style fill solid 0.3
    set style data boxes
    set datafile separator comma
    plot ‘file_name’
```

Pass Options and Redirect Output

- gnuplot options.sh > out.png
- Graph multi-columns
  - plot for [i=2:5] ‘data.csv’ using i:xtic(1) title columnheader linewidth 4
    # for [i=2:5] –loop for column between 2 to 5 inclusive
    # using i:xtic(1) plot column using tick label from column 1
  # titleheader –use the first row as titles
  # linewidth 4- use a wider line width

Dialog boxes

- dialog --common-options --boxType "Text" Height Width --box-specific-option
- dialog --title "Hello" --msgbox 'Hello world!' 6 20
- Parameters with dialog

  --inputbox <text> <height> <width> [<init>]

  --calendar <text> <height> <width> <day> <month> <year>

- --infobox <text> <height> <width>
- --yesno <text> <height> <width>
- --checklist <text> <height> <width> <list height> <tag1> <item1> <status1>...
- --msgbox <text> <height> <width>
- --passwordbox <text> <height> <width> [<init>]
- --menu <text> <height> <width> <menu height>\ <tag1> <item1>...

Eg: Checklist

```
dialog --checklist "Choose toppings:" 10 40 3 \
        1 Cheese on \
        2 "Tomato Sauce" on \
        3 Anchovies off
```

Eg: Menu

```
dialog --menu "Choose one:" 10 30 3 \1 red 2
green  3 blue
```