Unix Programming

UNIT-III

B.Tech(CSE)-V SEM

UNIT-III : INTRODUCTION TO SHELL PROGRAMMING

CO3: Develop basic programs using shell script

Topics:

- -Shell Variables-The Export Command-The Profile File a Script Run during Starting
- -The First Shell Script-The read Command
- -Positional parameters-The $? Variable knowing the exit Status
- -More about the Set Command-The Exit Command
- -Branching Control Structures-Loop Control Structures
- -The Continue and Break Statement
- -The Expr Command: Performing Integer Arithmetic-Real Arithmetic in Shell Programs
- -The here Document(<<)-The Sleep Command
- -Debugging Scripts-The Script Command-The Eval Command-The Exec Command.
- Command Line Structure- Metacharacters.

Shell and its Types:

- In UNIX and LINUX, a shell refers to a program that is used to interpret the typed commands the user sends to the operating system.
- The closest analogy in Windows is the DOS Command Prompt.
- However, unlike in Windows, Linux and UNIX computers allow the user to choose what shell they would like to use.
- Types of Shells in UNIX/LINUX:
  - Bourne Shell
  - C-Shell.
  - Korn Shell (K-Shell).

Types of Shells:

- **Bourne Shell:**
  - The original Bourne shell is named after its developer at Bell Labs, Steve Bourne. It was the first shell used for the UNIX operating system, and it has been largely surpassed in functionality by many of the more recent shells. The shell prompt is $, Execution command is sh.
- **C Shell:**
  - Designed by Bill Joy at the University of California at Berkeley .The C shell, as its name might imply, was designed to allow users to write shell script programs using syntax very similar to that of the C programming language. The shell prompt is % and execution command "csh."

    It has two advantages over the Bourne shell.
    - Aliasing of commands
    - History Commands

Types of Shells(Contd..)

3. **Korn Shell:** It is a superset of Bourne shell. It offers a lot more capabilities and is decidedly more efficient than the other. It was designed to be so by David Korn of AT&T's Bell Labs.

   The Korn shell prompt is $ and execution command ksh.

Shell Variables

Variable is data name and it is used to store value. Variable value can change during execution of program.

Variables are 2 types:

1) Environment variables / System variables/unix defined variables
2) User defined variables.

Environment Variables

- Environmental variables are used to provide information to the programs you use.
- These variables control the behavior of the system.
- They determine the environment in which the user works.
- If environment variables are not set properly, the users may not be able to use some commands.
- Environment variables are so called because they are available in the user's total environment i.e. the sub-shells that run shell scripts and mail commands and editors.
- Some variables are set by the system, others by the users, others by the shell programs.
- **env command** can be used to display environment variables.

Environment Variables (Contd..)

- **HOME**
  - This variable indicates the home directory of the current user.
  - This variable is set for a user by the system admin in /etc/passwd.
  - **Example:** $echo $HOME
- **PATH**
  - This variable specifies the locations in which the shell should look for commands.
  - Usually, the PATH variable can be set as follows:
  - $PATH=/bin : /usr/bin
- **PS1 and PS2**
  - The shell has 2 prompts:
    - The primary prompt $ is the one the user normally sees on the monitor. $ is stored in PS1.
    - The user can change the primary prompt as follows:
  - $ PS1="HELLO>"
    - HELLO> //similar to windows
  - The secondary prompt > is stored in PS2.

Environment Variables (Contd..)

- **LOGNAME**
  - This variable contains the users login name.
  - Example:

```
$echo   $LOGNAME
      (or)
$echo  "${LOGNAME}"
```

- **SHLVL**
  - This variable contains the shell level that you are currently working in.

Example:

```
$echo      $SHLVL
 1
$sh                    #new shell
$echo      $SHLVL
 2
$exit
$echo      $SHLVL
 1
```

Environment Variables (Contd..)

- **SHELL**
  - This variable specifies the current shell being used by the users.
  - Different types of shells are:
    - 1)Bourne shell /bin/sh
    - 2) C-shell /bin/csh
    - 3) Korn shell /bin/ksh
  - To see users default shell

    $echo $SHELL

    /bin/bash
- **TERM**
  - This variable indicates the terminal type that is used.
  - Every terminal has certain characteristics that are defined in a separate control file in the terminfo directory.
  - If TERM is not set correctly, vi will not work and the display will be faulty.
- **env:** we can use this to view a list of environment variables and their respective values.

  $env

User defined Variables:

- A variable is a character string to which the user assigns a value.
- The value assigned can be a number, text, filename, device, or any other type of data.

Syntax:

<variable-name> = <value> **# variable definition**

- The value of variables are stored in the ASCII format.

Example:

```
$ x=50
$ echo   $x        #displays 50
```

- In command line, all words that are preceded by a $ are identified and evaluated as variables.

Uses of User defined Variables :

- **Setting pathnames:** If a pathname is used several times in a script, we can assign it to a variable and use it as an argument to any command.
- **Using command substitution**: We can assign the result of execution of a command to a variable. The command to be executed must be enclosed in backquotes (also called accent grave).

  Example: $echo "Today's date is `date` "
- **Concatenating variables and strings:** Two variables can be concatenated to form a new variable.

  Example:

  $ name=johny age=10

  $ echo $name $age # output: johny 10

  $echo Name of the boy is $name, and his age is $age

  output: Name of the boy is johny, and his age is 10

User defined Variables (Contd…)

unchanging variables: readonly function

- The values of which can only be read but not be manipulated.

Example:

```
$a=20
$ readonly  a
```

- **unset:** To clear local environment variable is by using unset command. To unset any local environment variable temporarily,

```
$unset  <var-name>
```

export command:

- In shell programming, the values of the variables set or changed in one program will not be available for other programs.

- What ever the variable created in the current shell is local variable to that shell.

Example: (current shell)

```
$name=rama
$echo  name
$echo $name     # output: rama
$sh             #newshell
$echo $name     # no output
```

$ (ctrl+d) Exit from current shell and goes to parent shell

```
$echo $name     # output: rama
```

export command (Contd…)

export: It is used to declare the variables as global. Making variables to all the substantial shells. Its acts as a global variable.

Example: (current shell)

```
$name=rama
 $export name
 $echo $name     # output: rama
 $sh                #newshell
  $echo $name     # output: rama
 $name=sita
 $echo  $name              # output: sita
```

$ (ctrl+d) Exit from current shell and goes to parent shell

```
  $echo $name     # output: rama
```

export command (Contd…)

Using –f option for exporting functions:

```
$name () { echo “hello”; }
```

- $export –f name
- $name

```
//prints “hello”
```

.profile file (a script run during starting)

- A profile file is a start-up file of an UNIX user.
- This file gets executed as soon as the user logs in.
- The .profile file is present in your home ($HOME) directory and lets you customize your individual working environment.
- However, the user can customize the profile as per their requirement.
- i.e. The user can
  - → assign suitable values to the environment variables.
  - → add and modify statements in the profile file.

The .profile file:

.profile file controls the following by default:

- Shells to open
- Prompt appearance
- Keyboard Sound.
- The .profile file contains your individual profile that overrides the variables set in the /etc/profile file.

- The user can view his ".profile" as follows:

```
$ cat .profile
MAIL= /var/mail/kumar
PATH=/bin:/usr/bin
PS1='$'
PS2='>'
SHELL=/usr/bin/bash
TERM= tty1
```

Shell Programming: Introduction

- A shell script contains a list of commands which have to be executed regularly.
- Collection of Unix Commands is **called as Shell script.**
- Shell script is also known as shell program.
- The user can execute the shell script itself to execute commands in it.
- A shell script runs in interpretive mode. i.e. the entire script is compiled internally in memory and then executed.
- Hence, shell scripts run slower than the high-level language programs.
- **".sh" is used as an extension** for shell scripts.

Shell Programming: Introduction (Contd…)

- The hash **symbol # indicates the comments** in the script.
- The shell ignores all the characters that follow the # symbol. However, this does not apply to the first line.
- The first line **"#! /bin/sh"** indicates the path where the shell script is available.

The First Shell Script:

- The shell script or a shell program is a set of commands that are executed together as a single unit.

- A shell script also includes:
  - Commands for selective execution (Conditional statements).
  - Commands for I/O operations like read and echo.
  - Commands for repeated execution (loops).
  - Shell variables and so…on.

- A shell script is named just like all other files with .sh extension.

- A shell program runs in the interpretive mode.

Steps in creating Shell Script

1. Create a file using an vi editor (or any other editor) . Name script file with extension .sh.

   vi

1. Start the script with **#!/bin/bash**
2. Write some code
3. Save the script file as **filename.sh**
4. Execute a shell script

   a) using sh command

   $sh filename.sh

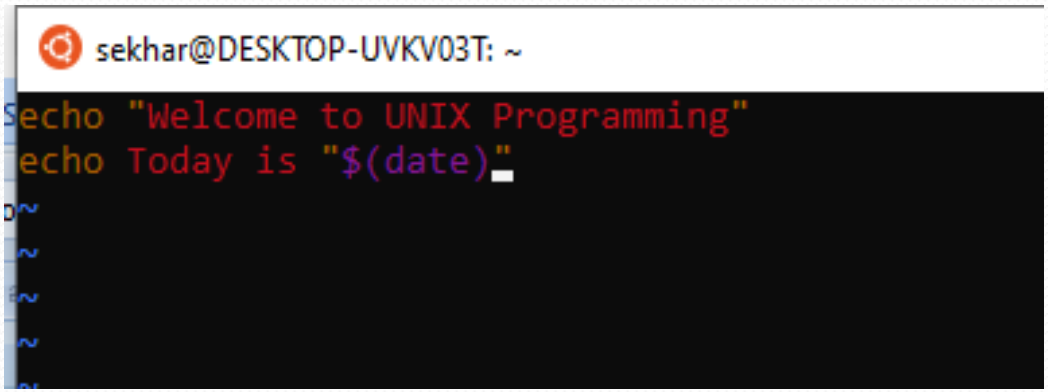
   b) using chmod command

   $chmod u+x filename.sh

   $./filename.sh

Sample shell script:

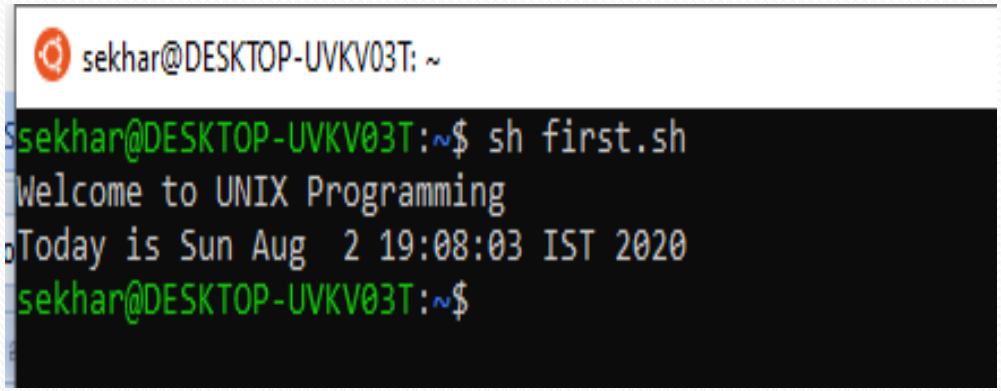
$vi first.sh



Execution of shell script can be done in two ways:

1. Using internal command i.e; sh .
2. Using external command i.e; chmod.

Execute the script using the command:
$sh first.sh



Comments in shell script:

- Comments are used to explain :
  - The purpose and Logic of the program.
  - Commands used in the program.
- The **# symbol** is used to represent the **comments** in the shell script.
- **Example:**

```
#This is a Comment Line.
#This is my first shell program.
```

read and echo (Interactive Shell):

- The **read** command can be used for taking input from the keyboard and **echo** to display output.

- It is shell's internal tool for making scripts interactive.

Syntax:

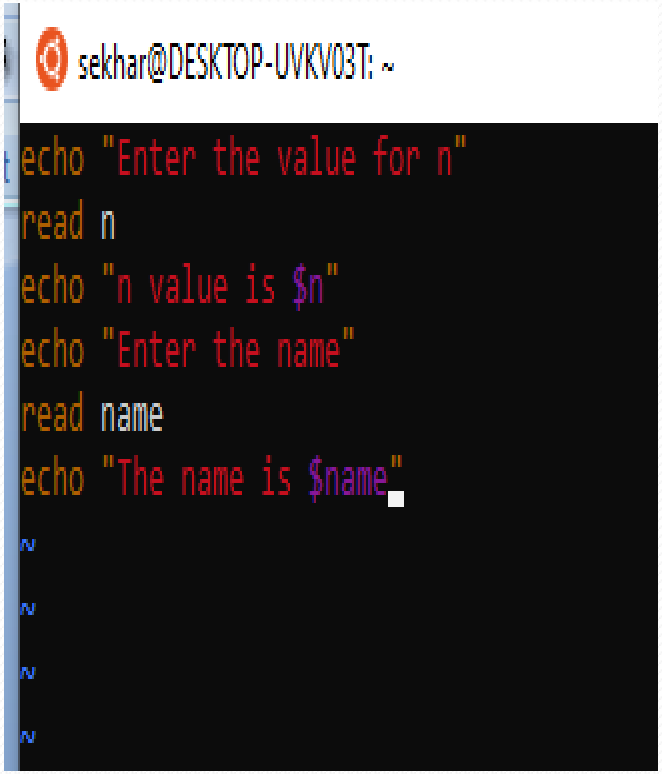
read <var_name>

- It is used with one or more variables.

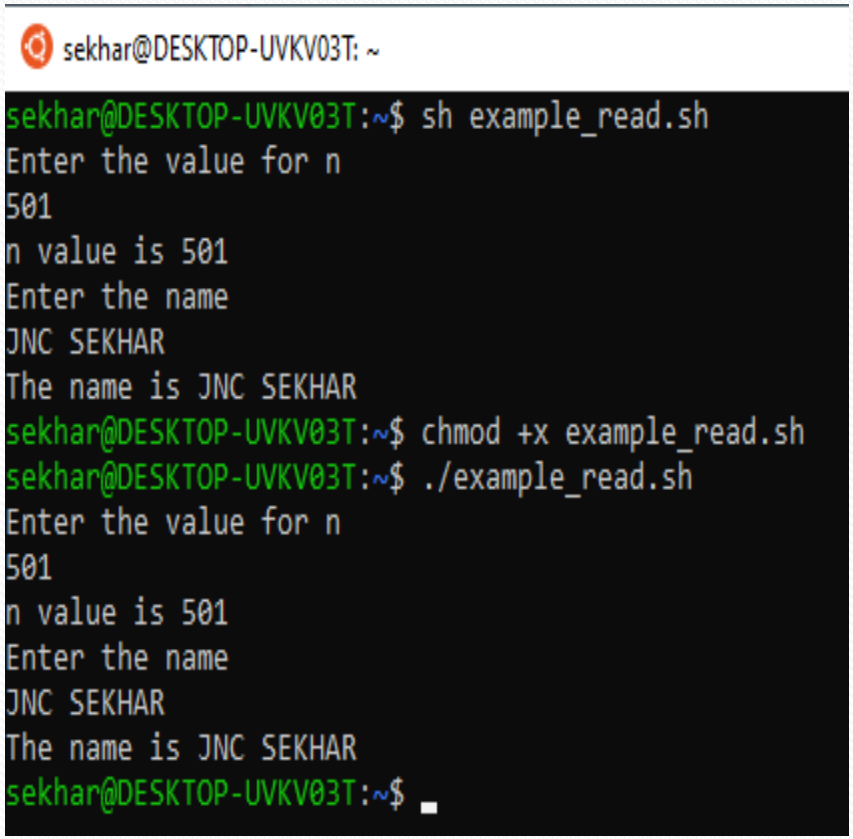
- The variables are used to hold inputs given with the standard input.

Example script for read command:

$vi example_read.sh



Executing script using two methods:



Multiple arguments using read command:

- The **read** command can take multiple arguments.

- In other words, values for more than one variable can be assigned or input using a single read command.

  Example:

  ```
  read  a   b   c
  ```

- Arguments are separated by space.

- If number of input values are less than the number of arguments, then the arguments or variables to which values are assigned will be initiated to null.

readonly command:

- The readonly command can be used to make variables readonly i.e. the user cannot change the value of variables(Constants).

- During shell scripting, we may need a few variables, which cannot be modified.

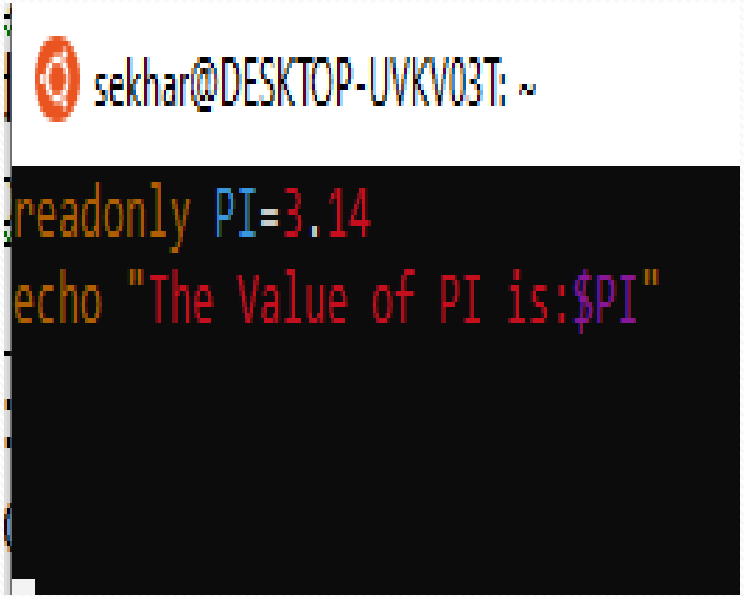
- This may be needed for security reasons.

Syntax:

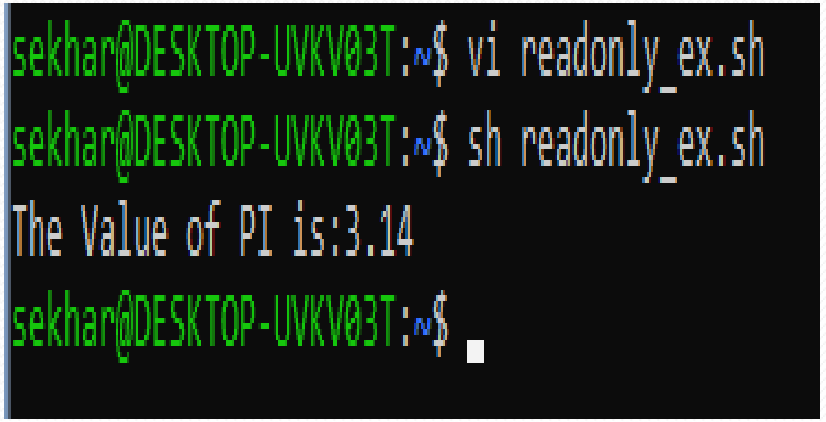
readonly <var_name>=value

Example for readonly:

$ vi readonly_ex.sh



$ sh readonly_ex.sh



Positional parameters/command line arguments:

- The arguments submitted with a shell script are called positional parameters.
- The **name of the command** run (usually the name of the shell script file) is put into a variable **$0.**
- The **first argument** (the 2nd word on the command line) is put into a variable **$1**.
- The **second argument** (the 3rd word on the command line) is put into a variable **$2** and so on.

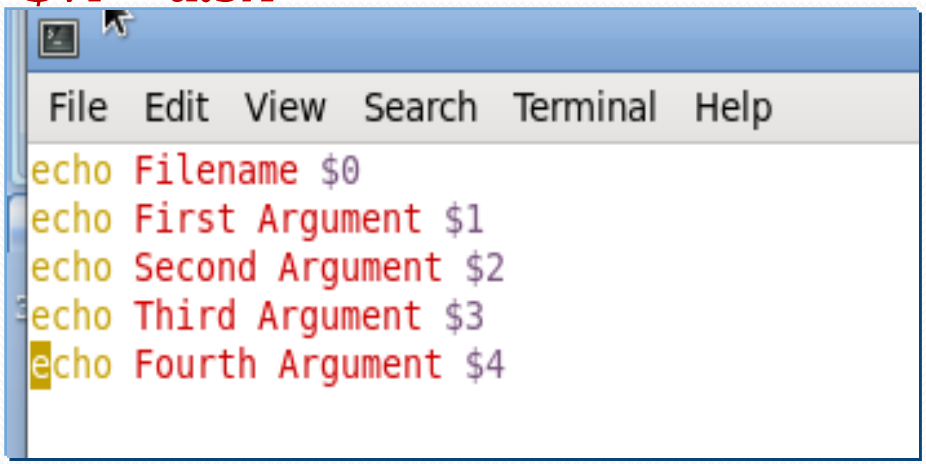
- The linux shell creates a maximum of 9 variables other than $0.represented as:

$0 $1 $2 $3 $4 $5 $6 $7 $8 $9

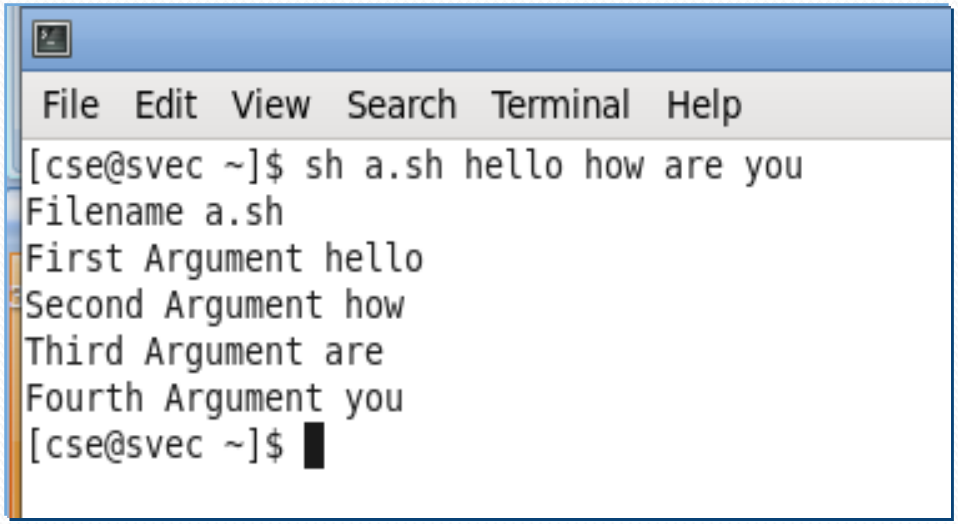
- The **variables $1 through $9 are called positional parameters** of the command line. Depending on the number of arguments specified in the command, the shell assigns values to some or all of these variables.

Positional parameters / command line arguments (Contd…)

$vi a.sh



OUTPUT

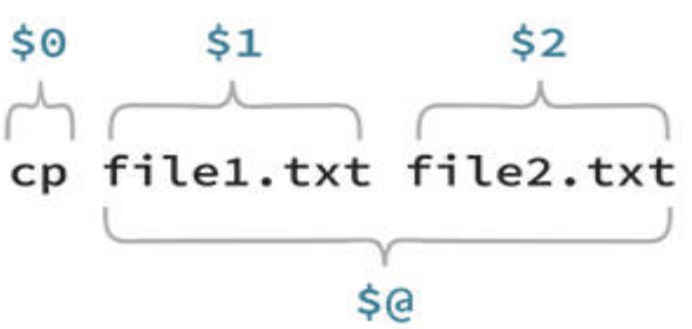


$#, $* and $@ Variables:

Besides the variables , $0 to $9, the shell also assigns values to the following special variables:

- **$# :** variable holds the **count of the total number of parameters** i.e; the number of arguments. (It is similar to argc in c)
- **$* :** variable holds **the list of all the arguments**. (It is similar to argv[] in c)
- **$@:** similar to $*, but yields each argument separately when enclosed in double quotes.
- **$? :** Exit status of last executed command.
- **$$:** ProcessID(PID) of current shell
- **$! :** PID of last background process.
- **$0 :** File name of current shell.
- When **$* and $@** are used within quotes, the contents of $* is considered as a single string whereas the contents of $@ are considered as independently quoted and considered as independent string arguments.

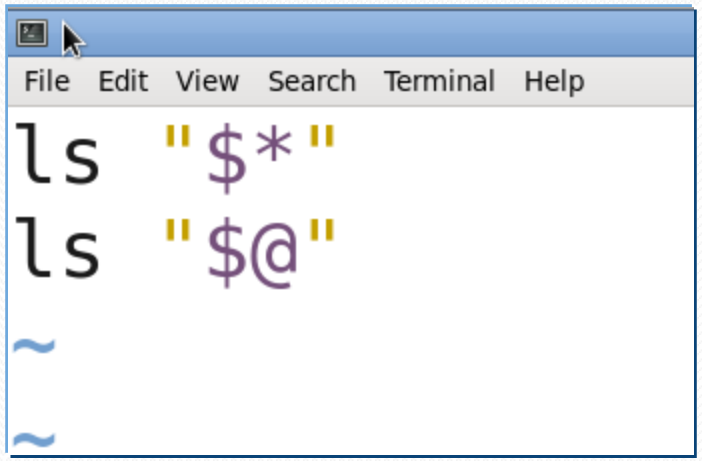
Positional parameters / command line arguments (Contd…)



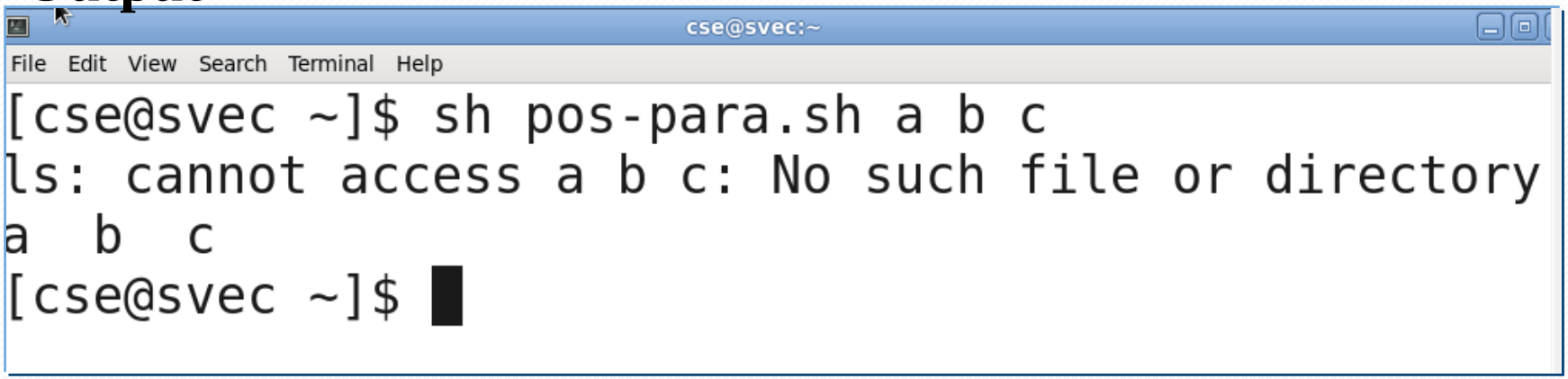
- The **$0** variable holds parameter **number 0**.
- It always represents the command or program to be executed.

Difference between $* and $@

$vi pos-para.sh



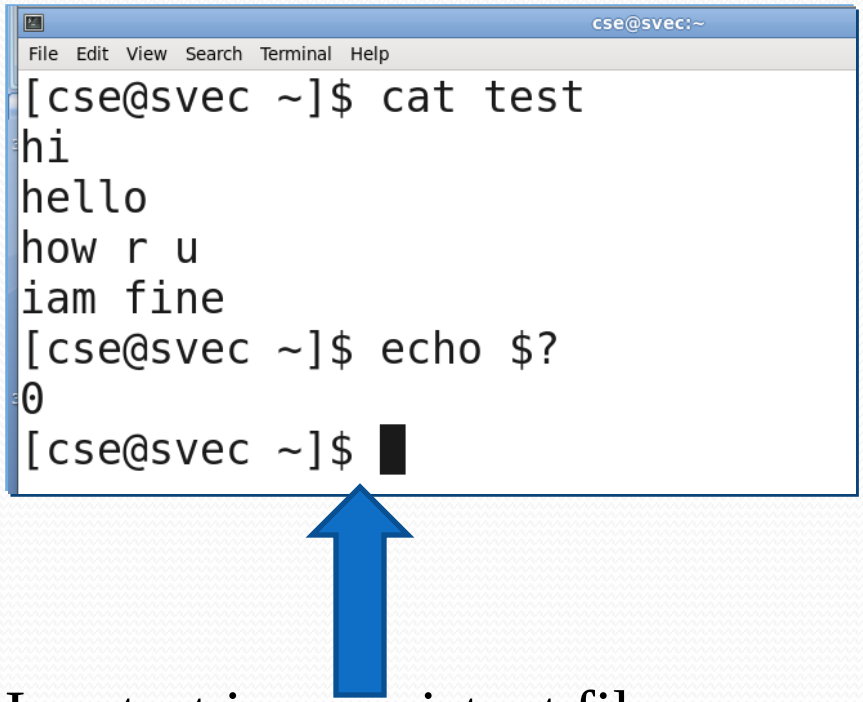
Output



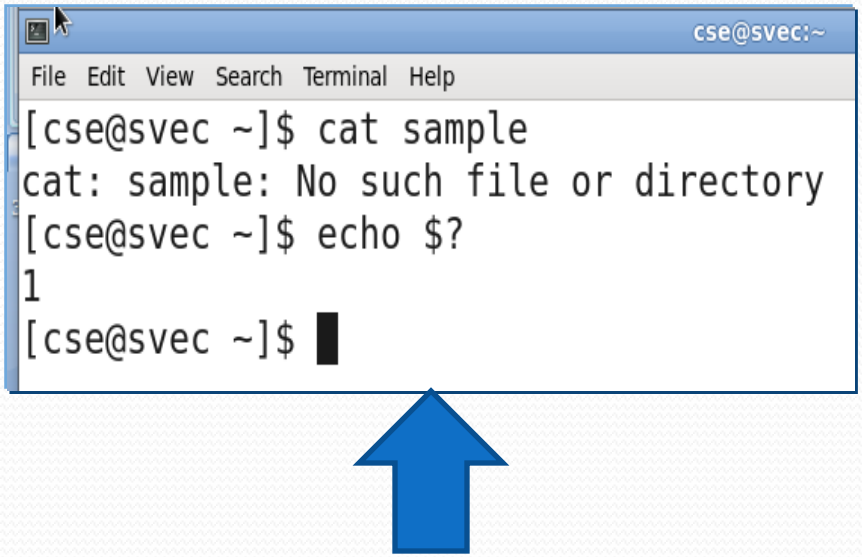
$? Variable (Knowing the exit status)

- Whenever a command is successfully executed, the program returns o(Zero).
- If a command is not executed successfully, a value other than zero will be returned.
- Logically, a **'o' is considered as true** and a **non-zero is considered as false.**
- These returned values are called program exit status.
- The program exit status will be available in shell special variable $?.
- An exit status value available in $? Is normally used in decision making in shell programs.

Example on $? :



Here test is an existent file



Here sample is an non- existent file

Example script on positional parameters:

$ vi pos_parameters.sh

```
echo "The total no of args are: $#"
echo "The script name is : $0"
echo "The first argument is : $1"
echo "The second argument is: $2"
echo "The total argument list is: $*"
```

Output:

```
sekhar@DESKTOP-UVKV03T:~$ chmod +x pos_parameters.sh
sekhar@DESKTOP-UVKV03T:~$ ./pos_parameters.sh 1 2 3 4 5
The total no of args are: 5
The script name is : ./pos_parameters.sh
The first argument is : 1
The second argument is: 2
The total argument list is: 1 2 3 4 5
sekhar@DESKTOP-UVKV03T:~$
```

set command:

- The set command is used to assign the values to the positional parameters on the command line.

Example:

```
$set India is my Country
$echo  $1          # displays India
$echo  $2          # displays is
$echo  $3          # displays my
$echo  $4          # displays country
$echo  $1  $4      # displays India Country
```

More about the set command:

- **The set command with out arguments:**
  - When set command is used with out arguments, it displays the contents of the system variables that are either local or exported.

- **The set command with options:**
  - Many options such as –x,-v, -- and others are allowed to be used with set command.
  - The options –x and –v are used to debug shell scripts.

- **The set command and the – option:**
  - Under certain circumstances, arguments to the set command are passed on through command substitution which is error-prone.
  - Such situation is handled using special option – (double-hyphen).

shift command:

- Only a maximum of 9 arguments are given in a command line.
- In case, more than 9 arguments are given in command Line, no error is indicated.
- Such type of situations are handled by shift command.
- In other words, shift command is used to handle excess command Line arguments.
- Shift command is used to shift the position of positional parameters.

Example:

```
$set  hello good morning how do you do welcome to Unix programming
shift  3
echo $1  $2  $3  $4
how do you do
```

exit command:

- The exit command is used to terminate the execution of the script.
- It is also used to exit from the shell that is being used currently.
- It is not necessary to use exit command at the end of every shell script.
- The shell recognizes end of the script automatically.
- The exit command can optionally use a numeric value as an argument (Eg: exit 0/1).
- A zero exit value indicates success and a non-zero value indicates failure.
- If no arguments are used, this command returns a zero.

Operators in UNIX:

- An operator is a symbol that is used for arithmetic and logical manipulations.

- The different operators in UNIX are:
  1) Arithmetic operators.
  2) Logical operators.
  3) Relational operators.

Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%

Operators in UNIX (Contd..)

1) Arithmetic Operators:

- The expr command is used to carry out basic arithmetic operations on integers.
- This command is used only when arithmetic operations are simple and are few.
- This command combines the following two functions:
  - Performs arithmetic operations on integers and
  - Manipulates strings.
- For complex arithmetic operations, we can use UNIX calculators like bc.

Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%

Example on Arithmetic Operators

$vi arthmetic.sh

```
File  Edit  View  Search  Terminal  Help
a=20 b=10
echo `expr $a + $b`
echo `expr $a - $b`
echo `expr $a \* $b`
echo `expr $a / $b`
echo `expr $a % $b`
~
~
```

Output:

```
cse@svec:~
File  Edit  View  Search  Terminal  Help
[cse@svec ~]$ sh arthmetic.sh
30
10
200
2
0
[cse@svec ~]$
```

Operators in UNIX (Contd..)

2) Logical Operators:

Logical AND and OR operations are specified as options instead of Operators.

Operation	Operator (or) Option
AND	-a
OR	-o
NOT	!

Operators in UNIX (Contd..)

3) Relational Operators:

All the relational operators are represented as Options.

Operation	Operator (or) Option
Greater than	-gt
Greater than or equal	-ge
Less than	-lt
Less than or equal	-le
Equal to	-eq
Not equal	-ne

Branching Control Structures:

- Program structures that are used to shift the point of execution are called Branching Control structures.

- These statements are called as selection statements or Conditional Statements or Decision-making Statements.

- There are total 4 conditional statements which can be used in shell programming:
  - if – then – fi statement
  - if – then – else - fi statement
  - if – then – elif –else -fi statement
  - case – esac statement (**case statement works just like switch statement in C**)

Branching Control Structures (Contd..)

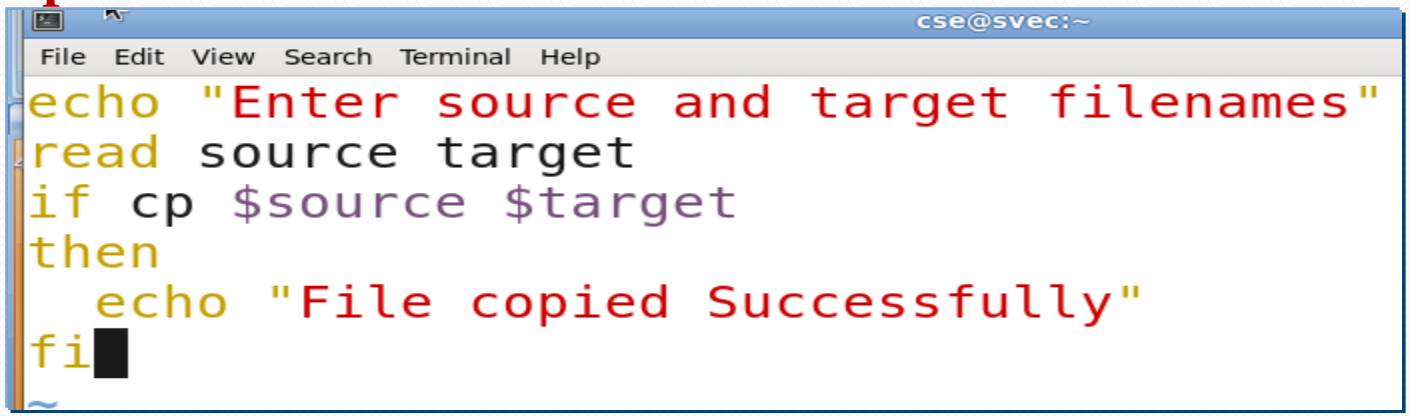
1) if – then – fi statement :

This block will be executed if specified condition is evaluated to true.

Syntax:

```
if   <Control-command / test_expression >
then
        true-block /statement
fi
```

Example:



Branching Control Structures (Contd..)

2) if – then – else - fi statement :

If specified condition in if is evaluated to true, the statement inside if block will be executed otherwise the statement in else block is executed.

Syntax:

```
if  <condition /test-expression >
then
        statement1
else
        statement2
fi
```

Example on if – then – else - fi statement :

$vi i.sh

cse@svec:~

File Edit View Search Terminal Help

```
echo " Enter Source and target filenames"
read source target
if cp $source $target
then
   echo "Files copied successfully"
else
   echo "Failed to copy the file"
fi
~
```

OUTPUT

cse@svec:~

File Edit View Search Terminal Help

```
[cse@svec ~]$ ls
a             a.sh  b1  d                          i1.sh  p            test
arthmetic.sh  b     c   gnome-terminal.desktop     i.sh   pos-para.sh
[cse@svec ~]$ sh i1.sh
 Enter Source and target filenames
a a1
Files copied successfully
[cse@svec ~]$ ls
a   arthmetic.sh  b   c  gnome-terminal.desktop  i.sh  pos-para.sh
a1  a.sh          b1  d  i1.sh                   p     test
[cse@svec ~]$
```

Branching Control Structures (Contd..)

3) if – then – elif –else -fi statement (else -if ladder):

- To use multiple conditions in one if-else block, then elif keyword is used in shell.
- If expression1 is true then it executes statement 1 and 2, and this process continues.
- If none of the condition is true then it processes else part.

- ***Syntax:***

```
if  <condition/test-expression1>
then
        statement(s)
elif  <condition/test-expression1>
then
            statement(s)
else
            statement(s)
fi
```

Branching Control Structures (Contd..)

4) case –esac statement:

- We can use multiple if...else statements to perform a multi-way branch. However , this is not always the best solution , especially when all of the branches depend on the value of a single variable.

- Shell supports case-esac statement which handles exactly this situation, and it does so more efficiently than repeated if...else statements.

- case statement works as a **switch statement** if specified value match with the pattern then it will execute a block of that particular pattern.

- If there is no match, the exit status of the case is zero.

Branching Control Structures (Contd..)

- ***Syntax:***

```
case  value  in                          #Beginning of case
   Pattern 1)
              Statement(s) to be executed if pattern1 matched
              ;;                         # Break
   Pattern 2)
              Statement(s) to be executed if pattern2 matched
              ;;
         *)
            Default condition to be executed
              ;;
esac                                     #End of case
```

Example on case-esac

Example: write a shell script to count the number of lines, words, characters

```
echo "Enter File name"
read fname
echo " Enter   1-line count, 2 – word count, 3-character count"
read ch
case  $ch in
1)  echo  `wc –l $fname`      ;;
2)  echo  `wc –w $fname`      ;;
3)  echo  `wc –lc$fname`      ;;
*)  echo  "Hai how are you"   ;;
esac
```

test command:

- This is a built-in shell command that evaluates the expression given to it as an argument.
- It return true if the evaluation of an expression returns a zero (0).
- It returns false if the evaluation of expression returns a non-zero value.
- The ***test command can also be replaced with [ ]*** (square brackets).

Syntax:

test(expression) / [expression]

- With test command, we can carry 3 types of tests:
  1. Numeric Tests
  2. String Tests.
  3. File Tests.

test command (Contd...)

1. Numeric Test:

- This is used when two numbers are compared using relational operators.

- They allow us to compare two values to see whether they are equal to each other, unequal or whether one is greater than the other. The below are the numerical test operators:

Operator	Meaning
-gt	greater than
-lt	Less than
-ge	greater than or equal to
-le	Less than or equal to
-ne	Not equal to
-eq	equal to

test command (Contd…)

Examples on Numerical test:

1.

```
echo "Enter the first number"
read a
echo "Enter the second number"
read b
if [ $a -gt $b ]
then
        echo " first number is greater than second number"
else
        echo " first number is less than second number"
fi
```

Example 2:

```
$x=5
$y=7
test $x –eq $y; echo $?      #returns 1 as the test fails.
test [ $x –lt $y ]; echo $?  # returns 0 as the test get success.
```

test command (Contd…)

2) File Test:

- Using these we can find out whether the specified file is an ordinary file (or) a directory, (or) whether it grants read, write (or) execute permissions , so on.

- Various File test Operators are:

Option	Meaning
-s file	True, if the file exists and has a size greater than 0.
-f file	True, if the file exists and is not a directory.
-d file	True, if the file exists and is a Directory.
-c file	True, if the file exists and is a Character Special file.
-b file	True, if the file exists and is a Block Special file.
-r file	True, if the file exists and have a read permission to it.
-w file	True, if the file exists and have a write permission to it.
-x file	True, if the file exists and have a Execute permission to it.
-k file	True, if the file exists and its sticky bit is set.

test command (Contd…)

Example on File test:

```
echo “Enter  Filename”
read   fname
if  [  -f  $fname  ]
then
    echo  “ you indeed entered a filename”
else
   echo  “ what you entered is not a filename”
 fi
```

test command (Contd...)

3) String Test:

- String tests are conducted to check:
  - Equality of strings.
  - Non-equality of strings.
  - zero- or non-zero length of a String.

Condition	Meaning
string1=string2	True, if the strings are same
String1!=string2	True, if the strings are different
-n string	True, if the length of the string is greater than 0
-z string	True, if the length of the string is 0
string	True, if the string is not a null string

test command (Contd…)

Examples on String test:

1.
```
echo "Enter your Name"
read name
if [ -z $name ]
  then
     echo " you have not entered your name"
   else
     echo "you have a nice name: $name"
fi
```

2.
```
echo -n "Enter the filename"
read fname
if test -f $fname
      then    echo "$fname is an ordinary file"
elif  test -d $fname
      then    echo "$fname is an directory file"
elif  test -s $fname
      then    echo "$fname is not an empty file"
   fi
```

Loop Control Structures / Iterative Control Structures:

- A loop involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied.

- The following are the Looping structures that are supported by UNIX shell programming:
  - using a for loop
  - using a while loop
  - using until loop

for loop:

- The **for** loop operates on lists of items. It repeats a set of commands for every item in a list.

Syntax 1:

```
for var-name in <list-of values>    # the list of values in the for loop should be separated by one/more blanks
    do
        Statement(s) to be executed for every word.
    done
```

Syntax 2:

```
for((expression1;expression2;expression3))
        do
                statement(s)
        done
```

Syntax 3:

```
for var in seq first incre last
        do
                statement(s)
        done
```

for loop Examples

1.
```
for i in rat mat
do
 echo $i
done
```

2.
```
for i in nick f1 f2
do
    cat $i
done
```

3.
```
for in `seq 1 2 10`
do
    echo $i
done
```

for loop Examples

4.
```
for((i=0;i<=10;i++))
do
    echo  $i
done
```

5.
```
echo "Enter n"
read  n
f=1
for((i=1;i<=n;i++))
do
    f=`expr   $f   \*   $i`
done
    echo  $f
```

while loop:

- The **while** loop enables you to execute a set of commands repeatedly until some condition occurs.

- It is a entry controlled iterative structure. Depending on the exit status of control command the commands are executed.

- If the exit status of command is true it returns 0 and if it is false it returns 1. As long as the condition is true the loop is executed in while.

Syntax:

```
while   <control condition>
do
    Statement(s) to be executed if command is true
done
```

while loop (Contd…)

Example: shell script for sum of digits using while loop

```
echo   “Enter n”
read  n
sum=0
r=0
while  [  $n   -gt    0 ]
    do
            r=`expr    $n    %  10`
           sum= `expr   $sum   +    $r`
            n=`expr   $n   / 10`
   done
echo   $sum
```

until Loop:

- The while loop is perfect for a situation where you need to execute a set of statements as long as the condition is true.
- Sometimes you need to execute a set of statements as long as the condition is false.
- In such cases, until loop is used.
- The until loop executes till the exit status of the control command is false and terminates when this status becomes true.

Syntax:

```
until   <control command>
    do
        Statement(s) to be executed until condition is true
    done
```

Difference between until and while loop

#print numbers from 1 to 10
#using while loop

```
i=1
while [  $i   -le   10  ]
  do
    echo  $i
     i=`expr  $i  +  1`
   done
```

#print numbers from 1 to 10
#using until loop

```
i=1
until [  $i   -le   10  ]
  do
    echo  $i
     i=`expr  $i  +  1`
   done
```

Unconditional statements:

The un-conditional statements in UNIX are:

a) break statement

b) continue statement

a) break statement: The break statement is used to terminate from loop or block.

Syntax: break

Example:

```
a=0
while [ $a -lt 10 ]
 do
     echo $a
     if [ $a -eq 5 ]
      then
          break
      fi
      a=`expr $a + 1`
 done
```

Unconditional statements(contd..):

b)continue statement: The continue statement is similar to the break statement except it causes the current iteration of the loop to exit rather than the entire loop.

Syntax: continue

Example:

```
for((x=0;x<=10;x++))
do
    y=`expr  $x  %  2`
        if  [  $y  –eq   0  ]
          then
          echo  “echo   $x”
          continue
          fi
      echo  “odd  $x”
done
```

expr command:

- The expr command is used to carry out basic arithmetic operations on integers.
- This command is used only when arithmetic operations are simple and are few.
- This command combines the following two functions:
  - Performs arithmetic operations on integers and
  - Manipulates strings.
- For complex arithmetic operations, we can use UNIX calculators like bc.

Integer arithmetic:

- Five operators used on integers: +, -, *, / and %.

Syntax:

```
expr  $op1  operator  $op2
```

Example:

```
$ x=5 y=3
$ expr  $x + $y          # outputs 8
$ expr  $x - $y          # outputs 2
$ expr $x \* $y          # outputs 15
$ expr  $x / $y          # outputs 1
$ expr  $x % $y          # outputs 2
$ z =`expr $x + $y`      # command substitution to assign a variable
$ echo $z                # outputs 8
```

String Handling:

- Three functions used on strings:
  - Finding length of string
  - Extracting substring
  - Locating position of a character in a string

Syntax:

```
expr   "exp1“   :   "exp2“
```

- On the left of the colon (:), the string to be worked upon is placed. On the right of the colon(:), a regular expression is placed.

String Handling (Contd..):

length of the String: The regular expression ".*" is used to print the number of characters matching the pattern.

Syntax:

```
expr  "string"  :  ".*"
```

Example:

```
$ expr  “srivasavi” : ‘.*’   # outputs 9
```

Extracting a Substring: expr command can be used to extract a string enclosed by the escape characters "\(" and "\)"

Syntax:

```
Expr  "string"  :  "\( substring \)“
```

Example:

```
$ expr  “srivasavi” : " \( vas \)"   # outputs ‘vas'
```

String Handling (Contd..):

Locating Position of a Character :

expr command can be used to find the location of the first occurrence of a character inside a string.

Syntax:

expr "string" : "[^ch]*ch" #ch → character

Example:

$ expr “srivasavi” : "[^a]*a" # outputs 2

Real arithmetic in shell programs:

- If both operands in an arithmetic operation are real, then that operation is called Real Arithmetic.
- The expr command works only on integers.
- Real arithmetic can be managed using the bc command (basic calculator) along with the scale function and the echo command.
- The output of the arithmetic expression is piped to the bc command.

  Example:

  $c=‘echo $a + $b | bc’

- Because of piping, echo does not display its output, rather it will direct its output to the bc command.

The here Document (<<)- I/O Redirection:

- The << symbol can be used to read data from the same file containing the script. This file is called as a here document.
- The term 'here' signifies that the data is here rather than in the file.
- Any command using standard input can also take input from a here document.

Syntax:

```
command << delimiter
document
delimiter
```

here Document (<<)- I/O Redirection(contd..)

For example:

```
$ mailx  vasavi << SRI
        Explore
        Dream
        Discover
 SRI
```

- The string (SRI) is delimiter.
- The shell treats every line delimited by SRI as input to the command mailx.
- vasavi at the other end will see 3 lines of message text with the date inserted by command.
- The word SRI itself doesn't show up.

here Document (<<)- I/O Redirection(contd..)

Using Here Document with Interactive Programs: A shell script can be made to work non-interactively by supplying inputs through here document.

Example:

```
$ wc -l << END
    Decide
    Commit
    Succeed
END
        #outputs number of lines = 3
```

The sleep command:

- sleep is a command-line utility that allows you to suspends the calling process for a specified time.
- In other words, the sleep command pauses the execution of the next command for a given number of seconds.

syntax:

sleep NUMBER[SUFFIX]...

- The NUMBER may be a positive integer or a floating-point number.
- The SUFFIX may be one of the following:

s - seconds (default)

m - minutes

h - hours

d - days

The sleep command (contd..)

- When no suffix is specified, it defaults to seconds.

- When two or more arguments are given, the total amount of time is equivalent to the sum of their values.

Example 1: Sleep for 5 seconds

sleep 5

Example 2: Sleep for 1 hour

sleep 1h

Example 3: Sleep for 1 day

sleep 1d

Debugging Scripts:

- When a script does not work properly, we need to determine the location of the problem.
- The UNIX shells provide a debugging mode.
- Run the entire script in debug mode or just a portion of the script.
- To run an entire script in debug mode, add **-x** after the #!/bin/[shell] on the first line:
- For Example :

  #!/bin/sh –x
- To run an entire script in debug mode from the command line, add a -x to the sh command used to execute the script:

  $ sh -x script_name

Debugging Scripts: options

Option	Meaning
set -x	Prints the statements after interpreting metacharacters and variables
set +x	Stops the printing of statements
set -v	Prints the statements before interpreting metacharacters and variables
set -f	Disables file name generation (using metacharacters)

- The set -v statement is similar to set -x, except, it shows the statement line before the shell performs any interpretation or substitution.

The script command:

- **script** command in UNIX is used to make typescript or record all the terminal activities.
- After executing the *script* command it starts recording everything printed on the screen including the inputs and outputs until exit.
- By default, all the terminal information is saved in the file *typescript* , if no argument is given.
- *script* is mostly used when we want to capture the output of a command or a set of command while installing a program.
- *script* command uses *two files* i.e. one for the terminal output and other for the timing information.
- **Syntax:**

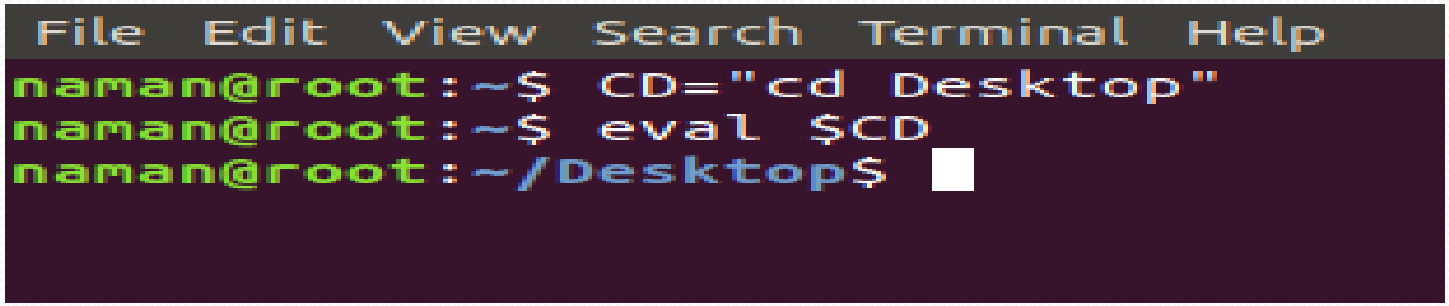
  script [options] [file]

The Eval command:

- ***eval*** is a built-in Linux command which is used to execute arguments as a shell command.
- It combines arguments into a single string and uses it as an input to the shell and execute the commands.
- **Syntax**

  eval [arg ...]

- In the below figure, you can see that *cd Desktop* command is stored in a variable "*CD*" as a shell command. Now you can use this "CD" as an argument with *eval* command.



The Exec command:

- The exec command has the following two abilities:
  - Running a command without creating a new process.
  - Redirecting standard input, output or error of a shell script from within the script.
- The exec command replaces the current shell process with the specified command.
- Normally, when we run a command, a new process is spawned (forked).
- The exec command does not spawn a new process.
- Instead, the current process is overlaid with the new command.

The Exec command:

- In other words, the exec command is executed in place of the current shell with out creating a new process.
- This command implements UNIX exec system call.
- **Syntax:**

  exec [-cl] [-a name] [command [arguments]] [redirection ...]
- **Options:**
  - **c:** It is used to execute the command with empty environment.
  - **a name:** Used to pass a name as the zeroth argument of the command.
  - **l:** Used to pass dash as the zeroth argument of the command.

Command Line Structure:

- A command is a program that tells the UNIX system to do something.
- **Syntax:**

  command [options] [arguments]

- Where an argument indicates on what the command is to perform its action, usually a file or a series of files.
- An option modifies the command, changing the way it performs.
- Commands are case-sensitive.

The command Line Structure:

- The components of an entered command may be categorized into one of four types: **command, option, option argument and command argument.**
- **command:** The program or command to run. It is the first word in the overall command.
- **option:** An option to change the behavior of the command. The available options are described in the command's manual page.
- **option argument:** Some options have their own arguments. For instance, sort has a -t option to specify a delimiter (-t :), and tar requires the -f option to be used with a filename (-f /dev/fdo).
- **command argument:** An argument to give the program some additional information, such as the name of a file or a string to search for.

Command Line Structure:

- **Example:**
  - Command and command are not the same.
  - Options are generally preceded by a hyphen(-).
  - For most commands, more than one option can be strung together.
- **Syntax:**

  command –[option][option][option]
- **Example:**

  ls –alR
- Will perform a long list on all files in the current directory and recursively perform the list through all sub-directories.

The command Line Structure:

- For most commands, we can separate the options, preceding each with a hyphen.
- **Syntax:**

  command –[option1]–[option2]–[option3]

- **Example:**

  ls –a-l-R

- options and syntax for a command are listed in the man page for the command.

UNIX Metacharacters:

- Metacharacters are a group of characters that have special meanings to the UNIX operating system.
- Metacharacters can make many tasks easier by allowing you to redirect information from one command to another or to a file, string multiple commands together on one line, or have other effects on the commands they are issued in.

UNIX Metacharacters:

Symbol	Meaning
>	Output redirection
>>	Output redirection (append)
<	Input redirection
*	File substitution wildcard; zero or more characters
?	File substitution wildcard; one character
[ ]	File substitution wildcard; any character between brackets
`cmd`	Command Substitution
$(cmd)	Command Substitution
|	The Pipe (|)
;	Command sequence
||	OR conditional execution
&&	AND conditional execution
( )	Group commands
&	Run command in the background
#	Comment
$	Expand the value of a variable
\	Prevent or escape interpretation of the next character
<<	Input redirection