DATE

Programming Constructs

* The shbang line :-

The "shbang" line is the very first line of the ~~shell~~ script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a #! followed by the full pathname to the shell, and can be followed by options to control the behaviour of the shell.

Eg:→ #!/bin/bash

* Comments :-

Comments are descriptive material preceded by a # sign. They are in effect until the end of a line and can be started anywhere on the line.

Eg:→ # This is a comment.

* Wildcards :-

There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards". These characters are neither numbers or letters. For example, the *, ?, and [] are used for filename expansion. The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell

DATE

they must be quoted.

Eg:- rm *, ls ??; cat file [1-3],
echo "How are you?"

* Displaying output :-
To print output to the screen, the echo command is used. Wildcards must be escaped with either a backslash or matching quotes.

Eg:- echo "How are you?"

* Local Variables :-
Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables can also be defined with the built-in declare function. Local variables are set and assigned values

Eg:- variable_name = value
declare variable_name = value

name = "John Doe"
x = 5

DATE

* Global variables :-

Global variables are called environment variables and are created with the export built-in command. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.

The built-in declare function with the -x option also sets an environment variable and marks it for export.

Eg :- export variable_name = value
declare -x variable_name = value

export PATH = /bin:/usr/bin:.

* Extracting values from variables :-

To extract the value from variables, a dollar sign is used.

Eg :- echo $variable_name
echo $name

echo $PATH

* Reading user input :-

The user will be asked to enter input. The read command is used to accept a line of input. Multiple arguments to read will cause a line to be broken into words, and

DATE

each word will be assigned to the named variable.

Eg:- echo "What is your name?"
read name
read name1 name2 ...

* Arguments :-
Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script

Eg:-
At the command line :-

$ scriptname arg1 arg2 arg3 ....

In a script :-

echo $1 $2 $3 Positional parameters

echo $* All the positional parameters

echo $# The number of positional parameters

* Arrays :-
The Bourne shell utilizes positional parameters to create a word list. In addition to positional parameters, the Bash shell supports an array

DATE

syntax whereby the elements are accessed with a subscript, starting at 0. Bash shell arrays are created with the declare -a command.

Eg:- set apples pears peaches. (positional parameters)
echo $1 $2 $3

declare -a array_name=(word1 word2 word3.)

declare -a fruit=(apples pears plums)

echo ${fruit[0]}

* Arithmetic :-

The Bash shell supports integer arithmetic. The declare -i command will declare an integer type variable. The Korn shell's typeset command can also be use for backward compatibility. Integer arithmetic can be performed on variables declared this way. Otherwise the (( )) (let command) syntax is used for arithmetic operations.

Eg:- declare -i variable_name

((n=9+5))

echo $n

PAGE

DATE

* Operators :-

The Bash shell uses the built-in test command operators to test numbers and strings, similar to C language operators.

Eg :-

Equality : == equal to
!= not equal to

Logical : && and
|| or
! not

Relational : > greater than
>= greater than, equal to
< less than
<= less than, equal to

* Conditional statements :-

The if construct is followed by an expression enclosed in parentheses. The operators are similar to C operators. The then keyword is placed after the closing paren. An if must end with an endif. The new [[ ]] test command is now used to allow pattern matching in conditional expressions. The old [ ] test command is still available for backward compatibility with the Bourne shell. The

DATE

case command is an alternative to if/else

Syntax :→ if test condition
then
commands
fi

Using square bracket []

if [condition]
then
commands
fi

Eg:-1). echo "enter a number"
read num
if test $num -gt 0
then
echo "$num is greater than 0"
else
echo "$num is not greater than 0"
fi
#end.

ii). echo "enter your age"
read age
if ["$age" -ge 18]
then
echo "you are eligible to vote"
else
echo "you are Younger!!".
fi

DATE

* CASE statement :-
The case construct works like C's switch statement, except that it matches patterns instead of numerical values.

Syntax :- case $variable in
pattern 1) command
command;;
pattern 2) command
command;;
pattern N) command
command;;
*) command
command;;
esac

Eg :- clear
echo "Enter a number (0-9)"
read number

case $number in
0) echo Zero;;
1) echo One;;
2) echo Two;;
3) echo Three;;
4) echo Four;;
5) echo Five;;
6) echo Six;;
7) echo Seven;;
8) echo Eight;;

PAGE

DATE

```
9) echo Nine;;
*) echo Invalid option !!!! Enter no. 1 to 9;;
esac
```

* LOOP - CONTROL STRUCTURES

Looping in a program allows a portion of a program to be repeated as long as the programmer wished.

Three types of looping constructs available are :-

1). The while loop
2). The for loop
3). The until loop

The while loop :-

The while loop is one of the most common and widely used loop control structure.

Syntax :-

```
while test_condition
    do
        command1
        command2
        . . .
        . . . .
    done
```

Here, while, do and done are keywords.

PAGE

DATE

Eg:-
```
n=10
while [$number -ge 1]
do
echo $number
number='expr $number -1'
done.
```

FOR loop :-

for loop is more frequently used as compared to the while and the until loops. The for loop allows us to specify a list of values which the control variable in the loop take. The loop is then executed for each value mentioned in the list.

Syntax :-
```
for VARIABLE in value1 value2 value3...
do
        COMMAND-BLOCK
done
```

Eg :-
```
clear
sum=0
for i in 1 2 3 4 5 6 7 8 9 10
do
 sum='expr $sum + $i'
done
echo "Sum of 10 numbers $num"
```

DATE

UNTIL condition

Another constructor which is very similar to the while loop is the until construct. The construct works in precisely the same manner with the one exception that it repeats a series of commands until a condition is met (as long as the condition is false).

Syntax :- until condition
do
commands
done

Eg :- number = 10
until [$number -lt 1] # while condition is false.
do
echo $number
number = `expr $number - 1`
done