Capgemini

UNIX

Lesson 15:Arithmetic Operation and command line argument

Lesson Objectives

- At the end of the session you will be able to understand:
  - Arithmetic operation
  - Command substitution
  - Command line argument(positional parameters)

15.1:Arithmetic operations
Arithmetic Operations

- UNIX shells do not support any notion of numeric data types such as integer or real.
- All shell variables are strings.
- Try this:
  - count=1
  - Rcount=$count+1

Shell arithmetic operations

- The shell will perform arithmetic expansion while it processes an arithmetic expression and then substitutes the result. When the expression is evaluated it is treated as if it were in double quotes. The shell evaluates the expression and replaces $((expression)) with the result. This is similar to how command substitution is performed.
- echo $[34-7+3]
  30
  echo $[8 - 9]
  -1
  echo $[(8+4) * 3]
  36
  echo $((945 / 9))
  105
- You do not have to use the "$" inside the expression for it to work correctly.
- a=45
  b=23
  echo $((3*a + 4*b))
  227
  echo $((3*$a + 4*$b))
  227

Example : arithmetic operation

- Here is a script that demonstrates how the arithmetic expansion can be used.

- ```
  #!/bin/bash
  Z=0
  LIMIT=500
  while [ "$Z" -lt "$LIMIT" ]
  ```
- ```
  do
  echo -n “$Z… ”
  Z=$( expr $Z + 1 )
  done
  echo
  exit 0
  ```

Old unix math comman let/expr command

- The UNIX command expr is used to evaluate expressions. In particular it can be used to evaluate integer expressions.
- Examples
  - echo `expr 5 + 6`
  - $ z=5 $ z=`expr $z+1` ---- Need spaces around + sign.
  - $ echo $z
  - 5+1 $
  - z=`expr $z + 1`
  - $ echo $z
  - 6

Let

- A Bash and Korn shell built-in command for math is **let**. As you can see, it is also a little picky about spaces, but it wants the opposite of what **expr** wanted. **let** also relaxes the normal rule of needing a $ in front of variables to be read.
- $ let z=5
- $ echo $z
- 5
- $ let z=$z+1
- $ echo $z
- 6 $
- let z=$z + 1 # --- Spaces around + sign are bad with let
- -bash: let: +: syntax error: operand expected (error token is "+")
- $let z=z+1 # --- look Mom, no $ to read a variable.
- $echo $z 7

15.2: Command Substitution Details

Command is enclosed in backquotes (`).

Shell executes the command first.

- Enclosed command text is replaced by the command output.

Display output of the date command using echo:

```
$echo The date today is `date`
The date today is Fri 27 00:12:55 EST 1990
```

Issue echo and date commands sequentially:

```
$echo The date today is; date
```

Example

Following instructions print pwd as a string:

- **var=pwd**

```
echo $var
Output: pwd
```

Following instructions execute PWD shell command and display the present working directory:

```
var=`pwd`
echo $var
Output:   /usr/deshpavan
```

15.3: Command Line Arguments (Positional Parameter) Details

Specify arguments along with the name of the shell program on the command line called as command line argument.

Arguments are assigned to special variables $1, $2 etc called as positional parameters.

special parameters

- $0 – Gives the name of the executed command
- $* - Gives the complete set of positional parameters
- $# - Gives the number of arguments
- $$ - Gives the PID of the current shell
- $! – Gives the PID of the last background job
- $? – Gives the exit status of the last command
- $@ - Similar to $*, but generally used with strings in looping constructs

Details

Arguments are assigned to special variables (positional parameters).

- $1 - First parameter , $2 - Second parameter,….
- Example:

```
echo Program: $0
        echo Number of arguments are $#
        echo arguments are $*
grep "$1" $2
echo "\n End of Script"
```

- Run script:

```
$ scr1.sh "Unix" books.lst          --$1 is UNIX  , $2 –books.lst
```

Shift statement

Command line arguments are treated as special variables within the script, the reason I am calling them variables is because they can be changed with the **shift** command. The command line arguments are enumerated in the following manner *$0*, *$1*, *$2*, *$3*, *$4*, *$5*, *$6*, *$7*, *$8* and *$9*. *$0* is special in that it corresponds to the name of the script itself. *$1* is the first argument, *$2* is the second argument and so on. To reference after the ninth argument you must enclose the number in brackets like this *${nn}*. You can use the **shift** command to shift the arguments 1 variable to the left so that *$2* becomes *$1*, *$1* becomes *$0* and so on, *$0* gets scrapped because it has nowhere to go, this can be useful to process all the arguments using a loop, using one variable to reference the first argument and **shifting** until you have exhausted the arguments list.

Example

$ set a b c d e

$ shift 2

$ echo $*

c d e

SUMMARY

- In this lesson, you have learnt:
  - Operators in script
  - Use of $ and `(back quotes)
  - Positional parameters

Review Questions

- Question 1: How to substitute command in shell script?
- Question 2: What is shift operator and use of it ?