

UNIT-4

Advanced Shell Programming

Textbook to refer: Sumitabha Das





Objectives of this course

- Shells and Sub-Shells, () and {}: sub- shell or Current Shell?
- export: Exporting shell variables
- Running a script in the Current Shell: The Command, let: Computation – A second Look (Korn and Bash)
- Arrays (Korn and Bash), String Handling (Korn and Bash), Conditional parameter substitution,
- Merging Streams, Shell Functions,
- eval: Evaluating Twice, The exec statement.



Introduction

- ❑ Shell-interpreter and an programming language.
- ❑ It is also a process, an environment which makes itself available to programs.
- ❑ It is required by system administrators.
- ❑ Constantly devise scripts and monitor correct system functioning.

Shells and Subshells

- When the shell executes a shell script it spawns a subshell at first, which in turn executes the commands in the script.
- When the script execution is complete the child shell withers away and returns control to the parent shell.
- Explicitly invoke a subshell to execute the shell script.
- The command representing the shell itself (sh,ksh,bash) can be used to read statements in join.sh.

```
sh join.sh
```

```
sh <join.sh
```

It is applicable only for executing shell scripts and not executables.

```
sh<a.out      //error
```

Shells and Subshells

- ❑ Even though the shell accepts a script name as argument we generally don't run shell scripts in the example shown above.
- ❑ We simply key in script name from shell prompt and run it as an executable.
- ❑ In this case current shell uses the subshell of same type to execute.
- ❑ However, if the script name contains the interpreter line in this form:

```
#!/usr/bin/ksh
```

Even though the login shell is Bourne shell it still uses Korn shell to execute the script.

- ❑ Specification of the interpreter line also help us to identify the shell script is meant to use.

() or {} :Sub-Shell or Current Shell?

- The shell uses two types of operators to group commands.
- 1. () - statements enclosed within parentheses are executed in a subshell.
- 2. {} - statements enclosed within curly braces are executed in current shell.
 - You have used first type to redirect the output of two commands with a single redirection symbol in a manner similar to this:

```
{ a.sh; b.sh; c.sh } > d.sh
```
 - Subshell considerations are not important here so we can use either form, but some applications require a set of commands to be run without spawning a child shell.

Usage of () operator to group commands

- To consider an example , let's use grouping operators with cd and pwd commands. Check your current directory and change with cd:

\$pwd

/home/kumar

\$(cd progs; pwd)

/home/kumar/progs //subshell execution

\$pwd

/home/kumar //parent login shell can't adopt the change.

Usage of {} operator

\$pwd

/home/kumar

\${cd progs; pwd;}

/home/kumar/progs //run in current shell without spawning

\$pwd

/home/kumar/progs //directory change is permanent.

Note:we need to precede the closing brace with a ;if both { and } appear in same line.

Shell script example

```
❑ if [ $# -ne 3 ]  
    then  
        echo "You have not keyed in 3 arguments"  
        exit 3  
    fi
```

Can be easily replaced with this sequence using curly braces:

```
[ $# -ne 3 ] && { echo "You have not keyed in 3 arguments"; exit 3; }
```

We cant use () because exit statement can terminate a script if it runs only in the same shell that is running the script.

Export: exporting the shell variables

- By default values stored in the shell are local to the shell and are not passed to a child shell.
- Export statement allow the variables to be recursively passes to all child processes so that once defined they are available globally.
- Consider a shell script which displays value of variable x ,assigns a new value to it and then displays the new value again:

```
$ cat var.sh
```

```
echo "The value of x is $x."
```

```
x=20
```

```
echo "The new value of x is $x"
```

Export: exporting the shell variables

```
$x=10; var.sh
```


The value of `x` is

//value of `x` is not visible in subshell

The new value of `x` is 20

```
$ echo $x      //value set inside the script doesn't affect value outside script  
10
```

Note: Because `x` is a local variable in login shell, its value can't be accessed by `echo` in the script which is run in a sub-shell.

- 
- ❑ To make x globally available we need to use export statement before the script is executed.

```
$x=10 ; export x
```

```
$var.sh
```

The value of x is 10 //value in parent shell is visible

The new value of x is 20

```
$echo $x                      //value reset inside script (child shell) is not available  
10                              outside(parent shell)
```

- ❑ **env** command also lists its exported variables.

- ❑ **Note:** A variable defined in a process is local to the process in which it is defined, and is not available in the child process. But when it is exported it is available recursively to all child processes. However when the child alters the value of variable the change is not seen in the parent.

Running a script in the current shell: the . command

- ❑ The dot command executes a script without using a sub shell. It does not require the script to have execute permission.
- ❑ Variable assignments made in shell start up file (.profile or .bash_profile) are always seen in the login shell.
- ❑ It is obvious that .profile is executed by login shell without creating a subshell.

```
{.profile;}
```

This does not work because .profile does not have execute permission in the file.

```
$ls -l .profile
```

```
-rw-r--r--  1 kumar group 7 27 feb 27 23:02  .profile
```

- ❑ You can make changes to .profile and execute it with the . Command without requiring to logout and log in again:

```
..profile
```

let: computation – a second look (korn and bash)

- ❑ Korn and Bash shell come with a built in integer handling facility that totally dispenses with the need to use expr.

- ❑ We can use let statement with and without quotes.

```
$let sum=234+456      //no whitespace after variable
```

```
$let sum="234+456"    //no whitespace after variable
```

- ❑ Note: if you are using whitespace for imparting better readability then quote the expression.

```
$let x=12 y=18 z=5
```

```
$let z=x+y+$z          //$ not required by let
```

```
$echo $z
```

```
35
```

- ❑ Let permits you to get rid of \$ altogether when making a assignment.
- ❑ Scripts run faster than ised with expr.

Second form of computation with ((and))

- Korn and Bash shell use(()) operators that replace let statement itself:

```
$x=22 y=28 z=5
```

```
$z=$(( x+ y+ z )) //whitespace is unimportant
```

```
$ echo $z
```

```
55
```

```
$z=$((z+1))
```

```
$echo $z
```

```
56
```

- POSIX recommends the use of ((and)) rather than let.

Arrays (Korn and Bash)

- ❑ Korn and bash support one dimensional arrays where the first element has index 0.

- ❑ Example to set and evaluate value of third element of the array prompt:

```
$prompt[2]="Enter your name"
```

```
$echo ${prompt[2]}
```

```
Enter your name
```

- ❑ Evaluation is done with curly braces and prompt[2] is just treated like a variable.
- ❑ Assigning group of elements, use a space delimited list having either of the two forms:

```
set -A month_arr 0 31 29 31 30 31 30 31 31 31 30 31 30 31 //korn only
```

```
month_arr=(0 31 29 31 30 31 30 31 31 31 30 31 30 31) //bash only
```

- ❑ first element has to be deliberately assigned to zero for obvious reasons.

Arrays (Korn and Bash)

```
$echo ${month_arr[6]}  
30
```

- Using *and @ as subscript you can display all elements of the array as well as number of elements.

```
$echo ${month_arr[@]}  
0 31 29 31 30 31 30 31 31 31 30 31 30 31
```

```
$echo ${#month_arr[@]}  
13
```

String Handling (Korn and Bash)

- ❑ Korn and Bash doesn't need `expr` as they have adequate string handling features themselves.
- ❑ They use wildcards but not regular expressions.
- ❑ All forms require usage of curly braces to enclose the variable name along with some special symbols.

- ❑ **Length of a String** - precede the variable name with a #

```
$name="Vinton cerf"
```

```
$echo ${#name}
```

```
11
```

- ❑ Can be used with if statement to check length of string

```
if [ ${#name} -gt 20 ] ; then
```

Extracting a string by pattern matching using % and

- # is used to match the string at the beginning.
- % is used to match the string at the end.

Both should be used inside curly braces when evaluating a variable.

- Remove the extension of filename

```
$filename=quotation.txt
```

```
$echo ${filename%txt}      //txt is stripped off  
quotation.
```

- %% -Wildcards to extract the hostname

```
$fqdn=java.sun.com
```

```
$echo ${fqdn%%.*}
```

Java

- Extract the base filename from a pathname

```
$filename="/var/mail/henry"
```

```
$echo ${filename##*/}
```

henry

Pattern matching operators

Form	Evaluates to the segment remaining after deleting
<code>\${var#pat}</code>	Shortest segment that matches the pat at beginning of \$ var
<code>\${var##pat}</code>	Longest segment that matches the pat at beginning of \$ var
<code>\${var%pat}</code>	Shortest segment that matches the pat at end of \$ var
<code>\${var%%pat}</code>	Longest segment that matches the pat at end of \$ var

Conditional Parameter Substitution

- Evaluating a variable depending on whether it has null or defined value.

- General form:

`${<var> : <opt><stg>}`

<var> is followed by colon and any of the symbols +,-,= or ? As <opt>

- **The '+' option:** var evaluates to stg if it is defined and assigned a nonnull string.

`found=`ls``

`echo ${found:+ "this directory is not empty"}`

- **the '-' option** : var evaluates to stg if it is undefined and assigned a null string.

`echo "Enter the filename:\c"`

`read fname`

`Fname=${fname:-emp.lst}`

Conditional Parameter Substitution

- **The '=' option:** makes the assignment to the variable that is evaluated.

```
echo "Enter the filename:\c"  
read fname  
grep $pattern ${fname:=emp.lst}
```

- Initialising a variable

```
x=1; while [ $x -le 10 ]  
Can be combined with one statement  
While [ ${ x:=1 } -le 10 ]
```

- **The '?' option** :evaluates if the parameter if the variable is assigned and it is non null.

```
echo "Enter the filename:\c"  
read fname  
grep $pattern ${fname:?}"no filename entered"
```

Merging Streams

- Instead of redirecting individual statements to output file we can merge altogether and redirect to specified stream.

`echo "None of the patterns found" 1>&2`

Sending the standard output to the destination of standard error.

- Notation `1>&2` merges standard output with standard error.

`2>&1` - Sending the standard error to the destination of standard output.

This notation is used whenever job is done in background and you want error messages to be held in the same file that contains actual output.

Shell functions

- ❑ It executes a group of statements as a bunch.
- ❑ It also returns a value optionally.
- ❑ Syntax:

```
function_name(){  
    statements  
    return value    // Optional  
}
```

- ❑ Function definition is followed by (), and body is enclosed with curly braces.
- ❑ On invocation function executes all statements inside the body.
- ❑ The return statement returns a value representing the success or failure of the function(and not a string value)


Simple example-when viewing list of large files in a directory

```
$ll (){  
ls -l $* | more  
}
```

Output

```
ll          //executes ls -l | more  
ll ux3rd??  //executes ls -l ux3rd ?? | more
```

Note: We can use ll function more like a procedure but a shell can also return its exit status with return statement. exit status can be gathered from shell parameter \$?



Generating a filename from system date

```
dated_fname(){  
set --`date`  
Year=`expr $6 : '..\(..\)`  
echo="$2$3_$Year"  
}
```

Output

```
$dated_fname  
Jan28_03
```

To continue or not to continue

```
anymore(){  
    echo "\n$1?(y/n) :\c" 1>&2  
    read response  
    case "$response" in  
        y|Y) echo 1>&2; return 0 ;;  
        *) return 1 ;;  
    esac  
}
```

Output

```
$anymore "Wish to continue"  
Wish to continue?(y/n) : n  
$echo $?  
1
```

Validating data entry

```
❏ Valid_string(){  
  while echo "$1 \c" 1>&2; do  
    read name  
    case $name in  
      "") echo "Nothing entered" 1>&2;continue;;  
      *) If [ `expr "$name":'.'*'`-gt $2 ] ;then  
        echo "Maximum $2 characters permitted" 1>&2  
        else  
          break  
        fi;;  
    esac  
  done  
  echo $name  
}
```



Eval: evaluating twice

- ❑ **eval** is a built in linux or unix command.
- ❑ The eval command is used to execute the arguments as a shell command on unix or linux system.
- ❑ Eval command comes in handy when you have a unix or linux command stored in a variable and you want to execute that command stored in the string.
- ❑ The eval command first evaluates the argument and then runs the command stored in the argument.

Eval examples

Example-1:

To execute that command stored in the string:

```
$ COMMAND="ls -lrt"
```

```
$ eval $COMMAND
```

output:

```
total 16
```

```
-rw-rw-r-- 1 user group 11 Oct 4 01:06 sample.sh  
-rw-rw-r-- 1 user group 0 Oct 4 01:06 test.bat  
-rw-rw-r-- 1 user group 0 Oct 4 01:06 xyz_ip
```



Eval another example

C="echo" ;a1="hello";A2="world!"

Eval \$C \$a1 \$a2

Output

Hello World




Exec statement

- Used to overlay a forked process.
- Need to overwrite the current shell itself with code of another program.

\$exec date

Tue Jan 8 21:21:52 IST 2003

- 
- ❑ **exec** is a critical function of any [Unix](#)-like [operating system](#).
 - ❑ Traditionally, the only way to create a new process in Unix is to [fork](#) it. The **fork** [system call](#) makes a copy of the forking program.
 - ❑ The copy then uses **exec** to *execute* the [child process](#) in its memory space.

exec rbash

Replace the current bash shell with [rbash](#), the restricted bash login shell. Because the original bash shell is destroyed, the user will not be returned to a privileged bash shell when **rbash** exits.