Shells and Sub-Shells

When the shell executes a shell script, it first spawns a sub-shell, which in turn executes the commands in the script. When script execution is complete, the child shell returns control to the parent shell.

- A shell script run with sh, ksh or bash need not have execute permission.

  $ sh script1.sh

- We can also simply key in the script name from the shell prompt and run it as an executable.

  $ script1.sh

  In this case, the current shell uses a sub-shell of the same type to execute it.

- If the script contains the interpreter line in this form:
  #!/user/bin/ksh
  Then, even though the login shell may be bash, it will use the Korn shell to execute the script.

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

The shell uses two types of operators to group commands.

- The ( ) statements enclosed within parentheses are executed in a sub-shell.
- The { } statements enclosed within curly braces are executed in the current shell only.

Examples of ( )

Example 1

Using ( ) we can redirect the output of two or more scripts/commands to another file.

```
$ (a.sh ; b.sh ; c.sh ) > d.sh
```

Example 2

```
$ pwd
/home/students
$ (cd bca ; pwd)
/home/students/bca                    Directory change is temporary
$ pwd
/home/students                        Back to original directory
```

In the above, working from a sub-shell (child), cd changed the working directory, but the parent can not adopt this change, so the original directory is back in place.

Examples of { }

Example 1

The same above command of cd and pwd if use with { } operators – do the different things.

```
$ pwd
/home/students
$ { cd progs ; pwd ; }
/home/students/bca                    Directory change is permanent
$ pwd
/home/students/bca                    Directory change is permanent
```

The two commands have now been executed without spawning a shell, no separate environment was created, and the change of directory became permanent even after the execution of the command group.

Example 2

Check the number of command line arguments and terminate the script with exit if the test fails.

```
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; }
```

The above statement must written in { } because it will run the command in same shell , so it will terminate the shell script. While the () use the sub sell so, an exit statement inside ( ) stop executing the remaining statement in the group, but that would not automatically terminate the script.

Exporting Shell Variables (export)

By default, the values stored in shell variables are local to the shell and are not passed on to a child shell. But the shell can also export these variables (with the export statement) recursively to all child processes so that, once defined, they are available globally.

```
$ cat > var.sh
$ echo The value of x is $x
x=20
echo The new value of x is $x
[ctrl-d]
```

Now first assign the value of x and then execute the script.

```
$ x=10;  var.sh
The value of x is                    value of x is not visible in a sub-shell
The new value of x is 20
$ echo $x                            Value set inside the script doesn't affect value
10                                   outside the script
```

Because x is a local variable in the login shell, its value can't be accessed by echo in the script, which is run in a sub-shell. To make x available globally, you need to use the export statement before the script is executed:

```
$ x=10 ; export x
$ var.sh                             Value in parent shell now visible here
The value of x is 10
The new value of x is 20
$ echo $x                            Value reset inside script (child shell)
10                                   is not available outside it (parent shell)
```

When x is exported, its assigned value (10) is available in the script. But when you export a variable, it has another important consequence; a reassignment (x=20) made in the script (a sub-shell) is not seen in the parent shell which executed the script.

Running a script in the current shell: The . command

. profile is a file that is executed by login shell without creating a sub shell.

If you try to execute the .profile in current shell by the following :

```
$ { .profile ; }
```

It will execute because .profile has not execute permission for the file.

```
$ ls –l .profle

-rw-r—r--    1    students    students    727    Aug 27 22:11    .profile
```

There is a special command used to execute a shell script without creating a sub-shell. The **.** (dot) command. you can make changes to .profile and execute it with the **.** command.

```
$ . .profile
```

Computation (let)

You can compute with the let statement with and without quotes.

```
$ let sum=12+13                    No whitespace after variable

$ let sum=”12 + 13”                No whitespace after variable
```

If you want to use whitespace for better readability that quotes the expression.

```
$ echo $sum

25
```

let can define three variables in single line.

```
$ let x=10 y=20 z=30
$let p=x+y+$z                                   $ is not require in let statement
$ echo $p
60
```

A Second form of computation with ((and)) The korn and bash use the (( )) operators that replace the let statement itself.

```
$ x=10 y=20 z=30
$ p=$((x+y   +z    ))                          Whitespace is unimportant
$ p=$((p+1))
$ echo $p
61
```

The ((and)) is alternate of let. Multiple dollar can replace with single in ((and)).

Arrays

Korn and Bash support one-dimensional arrays where the first element has the index 0. Following is the method to create an array and assign value to one of its indices.

Syntax

array_name[index]=value

Example

```
sname[0]=”Sahil”

sname[1]=”Samar”

sname[2]=”Punit”

sname[3]=”Akash”

sname[2]=”Rohit”
```

Following is the syntax for assigning a group of elements.

set –A num_arr 11 12 14 17 21 23 25	Korn only
num_arr=(11 12 14 17 21 23 25)	Bash only

Add an element in existing array

```
$ num_arr[7]=28
```

Accessing the value of array.

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

14
```

Using @ or * as subscript, you can display all elements of the array as well as the number of element.

The following will display all element.

```
$ echo ${num_arr[@]}

11 12 14 17 21 23 25 28

$ echo ${num_arr[*]}

11 12 14 17 21 23 25 28
```

The following will give you length of array.

```
$ echo ${#num_arr[@]}

8
```

Example of Array

```
echo –e “How many element you want to enter \c”

read n

i=1

while  [ $i <= $n ]

do

read a1[i]

i=`expr $i  + 1`

done

echo “Total elements are (one by one)”

i=1

while  [  $i <= $n  ]

do

echo ${a1[$i]}

i=`expr $i  + 1`



done
```

```
echo “All elements ${a1[*]}”
```

String Handling

Length of a String

The length of a string is easily found by preceding the variable name with a #.

Example:

```
$ name=”Welcome BCA”

$ echo ${ #name }

11
```

Extracting a String by Pattern Matching

You can extract the substring using a special pattern matching feature. The two character # and % are used for this purpose. # is used to match at the beginning and % at the end, and both are used in curly braces when evaluating a variable.

To remove the extension from a filename, we have used basename command. The same thing can be done using the format ${variable%pattern}

Example 1

```
$filename=student.lst

$echo ${filename%lst}

student.
```

The % symol after the variable name delete the string lst.

Example 2

To extract the hostname from the Domain

```
$ dn=java.sun.com

$ echo ${dn%%.*}

Java
```

Example 3

Extract the base filename from a pathname.

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

$ echo ${filename##*/}

tom
```

The above will delete the segment, /var/mail - the longest pattern that matches the pattern */ at the beginning.

Pattern Matching operator use by Korn and Bash

Form	The remaining segment after deletion
${var#pat}	Shortest segment that matches *pat* at beginning or *$var*
${var##pat}	Longest segment that matches *pat* at beginning of *$var*
${var%pat}	Shortest segment that matches *pat* at end of *$var*
${var%%pat}	Longest segment that matches *pat* at end of *$var*

Conditional Parameter Substitution

You can evaluate a variable depending on whether it has null or defined value. This feature is known as parameter substitution.

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

The variable <var> is followed by a colon and any of the symbols (operator) +,-,= or ? as <opt>, The symbol is followed by the string <stg>.

The + Option

var evaluate to *stg* if it is defined and assigned a nonnull string. This feature can be used to set a variable to the output of a command and echo a message if the variable is nonnull:

Example

```
found=`ls`

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

ls display nothing if it finds no files, in which case the variable found is set to a null string. However, the message is echoed if ls finds at least one file.

The – Option

However, var is evaluated to stg if it is undefined or assigned a null string (The opposite of + option).

Example

Inut file name, if the user do not enter filename then by default it should take student.lst

```
echo -e "Enter the filename : \c"

read fname

fname=${fname:-student.lst}

grep "bca" $fname
```

here, if fname is null if user not enter filename. So the student.lst will assign to fname.

The = Option

With the = option, you can use a parameter substitution with a command without making the intermediate assignment:

```
echo “Enter the filename : \c”

read fname

grep “bca” ${fname:=student.lst}
```

This feature is also use in initializing variable

```
x=1 ; while [ $x –le 10 ]
```

can now combined in to one

```
while [ ${x:=1}  -le 10 ]
```

The ? operator

It evaluates the parameter if the variable is assigned and nonnull. Otherwise, it echoes a string and kills the shell.

```
echo “Enter the filename : \c”

read fname

grep $pattern ${fname:?”No filename entered”}
```

If no filename is entered here, the message No filename entered is displayed. The script is also aborted without the use of an explicit exit command.

Evaluating twice (eval)

The shell's eval statement evaluates a command line twice. It suppresses (postpone) some evaluation in the first pass and performs it only in second pass.

```
$ prompt1="Roll No : "  ;  prompt2="Name : "  ;  prompt3="Class"

$ x=2; eval echo \$prompt$x

Name :
```

If we escape the first $ in \$prompt$x, so the first pass evaluates only $x, so now we have prompt1, The second pass evaluates $prompt1, this is done by prefixing the echo command with eval.

exec statement

The any command written with exec, the command overwrites the current process – often the shell. This has the effect of logging you out after the completion of the command.

```
$ exec date

Mon Aug 12 20:10:52 IST 2014

login:
```

Sometimes, you might want a user to run single program automatically on logging in, and then logged out automatically. For this, you can place the command in the .profile, duly preceded by exec. When the user logged in, the command execute automatically, and then logged out after completion of the command.