Unix Shell Scripting

Parameters

Arguments are passed from the command line into a shell program are called parameters. We can pass up to 9 parameters to any shell script each separated by a space. Each parameter can used within the shell script by using special variables $1 to $9 each corresponding its position as supplied in the command line.

Eg: Lets do a simple adding 2 number script where we pass those 2 numbers as parameters rather than doing a read or assigning value directly. Save the following code in *parm.sh* file.

```
#!/bin/bash
#This script the numbers to add from commandline (parameter passing)
#So within script we are not doing a read or assigning values to variables
#Instead we are using positional parameter variables $1 & $2
Z=`expr  $1  +  $2`
 echo  "Passed numbers are  $1   and   $2"
 echo "Sum is  $Z"
```

Execution

```
$ ./parm.sh     20  30
```

Note: Parameters after 9 can be accessed by usinf flower braces like ${n}

Parameter Shifting

Within a script supplies value of positional parameters can be shifted towards left. For example if 2 parameters ($1 and $2) are passed then shifting results in moving the value of $2 to $1. This technique is useful to view parameters fro 10th position.

Eg:

```
#!/bin/bash
shift
Z=`expr  $1  +  $2`
 echo  "Passed numbers are  $1   and   $2"
 echo "Sum is  $Z"
```

Execution

```
$ ./parm.sh     20  30
```

Execution

```
$ ./parm.sh     20  30  40
```

Changing the Values of Parameter Variables

Values of parameters can be changed within the script by using set command

Eg:

```
#!/bin/bash
 echo  "Passed Values are  $1   and   $2"
set  shell   script
echo  "Changed values from Script are  $1  &
$2"
```

Special Variables related to Parameters

- $* - It stores the complete set of positional parameters as a single string.
- $@ - Same as $*. Except that quoted parameters with space within treated as single unit.
- $# - It is set to the number of arguments supplied. This lets you design scripts that check whether the right number of arguments have been entered.
- $0 - Refers to the name of the script itself.

Functions

Big and complex scripts can be broken into smaller units which performs specific task. Following benefits of Function.

- Simplifies Code
- Allows reuse of code
- Easy to maintain

A Function can optionally take arguments and return values as well

Eg:

```
#!/bin/bash
Func_name()
{
Stments
Stment
}
#Main Script
Other statemnts
Func_name
Other stmnts
```

```
#!/bin/sh
# A simple script with a function...

add_a_user()
{


echo "Adding user $1 ...“
 useradd $1
 passwd $1 $2
  echo "Added user $1  with pass $2"
}

###
# Main body of script starts here
###
echo "Start of script..."
add_a_user bob  b0b123
add_a_user scott  Tiger

echo "End of script..."
```

Scope of Variable

It is surprise to know that there is no scoping in shell script for UDV. A variable assigned value is available in all the functions and main script as well. If the value is changed any of the functions or main scripts then same is reflected everywhere.

Eg:

```
#!/bin/sh

myfunc()
{

  x=2
}

### Main script starts here

x=1
echo "x before calling function is $x"
myfunc
echo "x after calling function is $x"
```

Local Variables

Scope of variable can be limited to that function or main script by making them "local" variables.

Eg

```
#!/bin/sh

myfunc()
{

  local x=2
}
### Main script starts here

x=1
echo "x before calling function is $x"
myfunc
echo "x after calling function is $x"
```

NOTE: local variables can be declared within functions only

Example with Return Value

Functions can return values that can be assigned to variables.

Eg

```
#!/bin/bash
myfunc()
{
 local SUM
SUM=`expr  $1 + $2`
 return $SUM
}

#Main Function
read -p "Enter a Number " X
read -p "Enter second number " Y
myfunc $X  $Y
 echo "Sum is  $?"
```

Creating a Library of useful Functions

It is often easier to write a "library" of useful functions, and source that file at the start of the other scripts which use the functions.

Lets say we have written all our useful functions in file called as myfunctions.sh and stored this file in /etc/ directory.

Write the following line at beginning of the script to call any of these functions.

source /etc/myfunctions.sh

Eg:

```
#!/bin/bash
source /etc/myfunctions.sh
read -p "Enter a Number " X
read -p "Enter second number " Y
myadd $X $Y
 echo "Sum is $?"
```

Example of Library

```
#!/bin/bash
myadd()
{
local SUM=0
SUM=`expr  $1 + $2`
 return $SUM
}
mysubt()
{
 local SUM
SUM=`expr  $1 - $2`
 return $SUM
}
mymulti()
{
 local SUM
SUM=`expr  $1 \* $2`
 return $SUM
}
mydiv()
{
 local SUM
SUM=`expr  $1 / $2`
 return $SUM
}
```

Debugging Script

Debugging helps us in finding where the error is occurring and what were the values of variables at that time. Debugging can be enabled by putting –x after shebang for example

```
#!/bin/bash -x
```

Or using set command immediately after shebang

```
#!/bin/bash
 set –x
```

Exiting on Error

Shell scripts don't exit on error but continue to execute next line. This behavior can be altered using -e with shebang or using set command. For example

```
#!/bin/bash –e
```

Or

```
#!/bin/bash
 set -e
```

Built-in Functions

Korn Shell Supports built-in math function listed below.

Function	Description
abs	Absolute value
log	Natural logarithm
acos	Arc cosine
sin	Sine
asin	Arc sine
sinh	Hyperbolic sine
cos	Cosine
sqrt	Square root
cosh	Hyperbolic cosine
tan	Tangent
exp	Exponential function
tanh	Hyperbolic tangent
int	Integer part of floating-point number

Good Practices

- ✓ Use good indentation
- ✓ Provide usage statements
- ✓ Use sensible commenting
- ✓ Provide comments that explain your code
- ✓ Exit with a return code when something goes wrong
- ✓ Use functions rather than repeating groups of commands
- ✓ Give your variables meaningful names
- ✓ Check that sufficient arguments are passed
- ✓ quote all parameter or variable expansions
- ✓ Check if needed files actually exist
- ✓ Always use set –e at the beginning of the script