Introduction to Shell Script

Script is a file consisting of commands. It can be written with any text editor. The first line of the script indicates the program that is used to interpret the script. Example:

```
#!/bin/bash
```

A simple shell script that displays the greeting "Hello, *user_name*." and date is shown below:

```
#!/bin/bash
echo "Hello, $USER."
echo Today is $(date).
```

- `echo` is used to display a line of text. It is optional to enclose the text with double quote.

- `USER` is the environment variable that stores the user name. Prefix a variable with a `$` sign to retrieve the value.

Since the script is written with text editor, it is a text file which is not executable. However, we can make the script executable by using the `chmod` command.

```
chmod <user list> <change> <permissions list>
```

Permissions list in letters:

`user list`: **u**: owner, **g**: group, **o**: others, **a**: all
`change`: **+** or **-**
`permissions list`: **r**, **w** or **x**

Example:

- Add execution permission to owner only: `chmod u+x file_name`
- Add execution permission to owner and group: `chmod ug+x file_name`

Permissions list in numerical:

3 octal digits indicate permissions for the user, group and others. Each octal digit represented by 3 binary bits, the binary bits indicate the read, write and execute permission (1 is enabled).

Example: `chmod 755 my_script`

WARNING: NEVER ever change a file permission to `000`!

To execute the script file, prefix the filename with "`./`", which means current directory. For example: `./my_script`

Variable and Expression

Format: `((expression))` or `[[ expression ]]`

Example:

```
#!/bin/bash
((x=5*8))
((y=$x +5))
((z=4**3))
echo Value of x is $x
echo Value of y is $y
echo Value of z is $z

p=5
q=$((4*5))
((r=$p+$q))
echo Value of p is $p
echo Value of q is $q
echo Value of r is $r
```

NOTE: There must be **no spaces** on either side of the equal sign.

Command Substitution

Command to be executed is placed in `()`.
Example:

```
#!/bin/bash
echo $(whoami) is on $(hostname).
echo “$(whoami) is on $(hostname).”
echo “$(whoami) is” on “$(hostname).”
echo “$(whoami) is on $hostname).”

info=”Your current working directory is: $(pwd).”
echo $info
```

Parameters

Parameters passed to the script are called by number, 0 – 9, preceded with a `$` sign.

`$0` refer to the name of the script itself.
`$@` refer to all parameters passed to the script, can be used to print out all parameters.
`$#` refer to the number of parameters passed to the script.

Example 1: A script called `addminus`

```
#!/bin/bash
sum1=$(($1 + $2))
echo “The numbers passed in are $1 and $2.”
echo The numbers passed in are $*
echo “The sum of $1 and $2 is $sum1.”
echo $1 - $2 is $(($1-$2))
```

Run the script in the terminal: `./addminus 4 5`

Example 2: A script called `calc_salary`

```
#!/bin/bash
name=$1
hour=$2
rate=$3
echo “The salary for $name is \$$(($hour * $rate)).”
```

Run the script in terminal and observe the output:

1. `./calc_salary Tom 20 5`
2. `./calc_salary Tom Cat 20 5`
3. `./calc_salary Tom 20`

Decision Structures: `if - else`

```
if [ expression ]
   then
      statements
   else
      statements
fi
```

Conditional Operators

```
-lt : less than
-gt : greater than
-le : less than or equal to
-ge : greater than or equal to
-eq or = : equal to
-ne or != : not equal to
```

Example: check the number of parameters passed to the script

```
   #!/bin/bash
   name=$1
   hour=$2
   rate=$3

   if [ $# -lt 3 ]
   then
   echo "Usage: $0 [name] [hours] [rate]"
   else
   echo “The salary for $name is \$$(($hour * $rate)).”
   fi
```

Run the script in terminal and observe the output:

1) `./calc_salary Tom 40 5`
2) `./calc_salary Tom 40`

`test` command

```
test [expression]
[ [expression] ]
```

Operands:

`-d file` True if `file` exists and is a directory.
`-f file` True if `file` exists and is a regular file.
`-n string` True if the length of string is non-zero.
`-s file` True if `file` exists and has a size greater than zero.
`-w file` True if `file` exists and is writable.
`-x file` True if `file` exists and is executable.
If `file` is a directory, true permission to search `file` is granted.
`-z string` True if the length of string string is zero.

Example: Read input and display

```
#!/bin/bash

echo -n "Type a word: "
read word
echo "The word you entered is: $word"
echo "Enter two words? "
read word1 word2
[ -z $word1 ] && [ -z $word2 ] && echo "Both are empty"
echo "Here is your input: \"$word1\" \"$word2\""
```

Example: Even and odd number

```
#!/bin/bash
if [ $(($1 % 2)) = 0 ]
  then
    echo "$1 is an even number."
  else
    echo “$1 is an odd number.”
fi
```

Example: Guess a vowel

```
#!/bin/bash
read -p "Guess a vowel: " ans
if [ $ans = "o" ]; then
  echo "You are correct."
else
  echo "Sorry. It's 'o'."
fi
```

Example: Check the existence of a file

```
#!/bin/bash
if [ $# -lt 2 ]; then
  echo Usage: $0 file_name
elif [ -f $1 ]; then
  echo "You got it."
else
  echo "No such file"
fi
```

Example: Check the existence of a directory

```
#!/bin/bash
if [ $# -lt 2 ]; then
  echo Usage: $0 directory_name
elif [ -d $1 ]; then
  echo "You got it."
else
  echo "No such directory"
fi
```

Example: Logical operator

```
#!/bin/bash
read -p "Guess a pet with 4 legs: " ani
if [ $ani = "cat" ] || [ $ani = "rat" ]; then
  echo "You got it."
else
  echo "Sorry. It's a cat or a rat."
fi
```

`case` Statement

Syntax:

```
case word in
   pattern1) statements
      ;;
   pattern2) statements
      ;;
   *) statements
      ;;
esac
```

Example 1: Yes/No

```
   #!/bin/bash
   read -p "Are you a student? (Y or N) " answer
   case $answer in
      Y|y) echo You are not allowed to enter.;;
      N|n) echo Welcome...;;
      *) echo Do you know \`Y\` and \`N\`?;;
   esac
```

Example 2: Display files in current directory

```
   #!/bin/bash
   read -p "List all files? (Y or N) " answer
   ans=$(echo $answer | tr [:lower:] [:upper:])
   case $ans in
      Y|YES)
               echo "Displaying all files..."
               ls -a
               ;;
      N|NO)
               echo "Displaying...except hidden files"
               ls
               ;;
      *) echo "Invalid answer!" ;;
   esac
```

Example 3: Menu selection

```
   #!/bin/bash
   echo "1. Display current working directory"
   echo "2. Display network configuration"
   echo "3. Exit"
   read -p "Your choice? " ans

   case $ans in
      1)echo -n "Your current working directory is "
           pwd
           ;;
      2)echo "Displaying..."; /sbin/ifconfig;;
      3)echo "Thank you.";;
      *) echo "Invalid answer!" ;;
   esac
```

Example 4: Pattern matching

```
#!/bin/bash
read -p "Is it morning? (YES or NO) " answer
case $answer in
   [Yy]|[Yy][Ee][Ss]) echo Good morning!;;
   [Nn]*) echo Good afternoon!;;
   *) echo Unrecognized answer!;;
esac
```

Example 5: Pattern matching
price of ticket:
≤ 12 years old: RM3.00
13 – 59: RM6.00
≥ 60: RM2.00

```
#!/bin/bash

read -p "Enter your age: " age
case $age in
   [1-9]|[1][0-2])echo "Ticket price: RM3.00";;
   [1][3-9]|[2-5][0-9])echo "Ticket price: RM6.00";;
   [6-9][0-9])echo "Ticket price: RM2.00";;
   *) echo "Invalid input";;
esac
```

Looping Structures: while Statement

```
while condition
do
   statements
done
```

Example 1:

```
   #!/bin/bash

   while true
   do
      read -p "Buy a ticket? (Y or N) " answer
      ans=$(echo $answer | tr [:lower:] [:upper:])
      case $ans in
         Y|YES)
            read -p "Enter your age: " age
            case $age in
               [1-9]|[1][0-2])echo "Ticket price: RM3.00";;
               [1][3-9]|[2-5][0-9])echo "Ticket price: RM6.00";;
               [6-9][0-9])echo "Ticket price: RM2.00";;
               *) echo "Invalid input";;
            esac
            ;;
         N|NO)
            echo "Thank you. Please come again."
            break;;
         *)echo "Invalid response"
      esac
   done
```

Example 2: print 0 – 9

```
   #!/bin/bash
   count=0
   while [ $count -le 9 ]
   do
      echo $count
      ((count++))
   done
```

`for` Statement

Syntax 1:

```
for var in list
do
  statements
done
```

Example:

```
  #!/bin/bash
  for i in 1 2 3 4 5 6 7 8
    do
    echo "$i power of 2 is $((i**2))."
  done
```

Syntax 2:

```
for ((initial value; conditional expression; updating expression))
```

Example 1:

```
  for ((num=1; num<10; num+=2))
  do
    echo "$num power of 3 is $((num**3))."
  done
```

Example 2: Expansion

```
  #!/bin/bash
  for file in $(ls)
  do
    echo The first 4 lines of $file:
    head -n 4 $file
    echo
  done
```

Example 3:

The following script will create a directory called `myhome` and then create another 9 directories, namely `user1`, `user2`,... , `user9`, in `myhome` directory.

```
#!/bin/bash
mkdir myhome

# create 9 directories
i=1
while [ $i -le 9 ]
do
  mkdir myhome/user$i
  ((i++))
done
```

The exit status of an execution is stored in the shell variable **`$?`**

Example:

```
#!/bin/bash
ls $1
if [ $? = 0 ] ; then
  echo File exists.
else
  echo Sorry, not found.
fi
```

Execute the script (assume that the script is named `chk_status`): `./chk_status rrr`

Sample output:

```
ls: cannot access rrr: No such file or directory
Sorry, not found.
```

The first line of the output is the error message from the command. To redirect the error message from the execution to null device, run the script as:

```
./chk_status rrr 2> /dev/null
```

Sample output:

```
Sorry, not found.
```

NOTE:

`2>`
Redirect error message to a specified file.
Example: `ls sss 2> errlog`

The **`2`** in **`2>`** is the file descriptor.

Descriptor	Name
0	Standard input
1	Standard output
2	Standard error

`1>&2` will redirect standard output to standard error.
`2>&1` will redirect standard error to standard output.

`>&` is **merging redirect** operator**.**

Command Expansion

Example: `rm file{1,2,3,A,B}`

Example:

```
#!/bin/bash
for name in file{1,2,3}; do
   echo “# Modified using for loop...” > $name
done
```

Subroutine in Script

Subroutine behaves almost same as a script. Arguments that are passed to a subroutine are stored in `$1`, `$2`, and so on.

Example:

```
   #!/bin/bash

   mysub(){
     echo “Argument 1: $1”
   }

   mysub “Argument to subroutine.”
```

Subroutine can return a value and the value will be stored in `$?`.

Example:

```
   #!/bin/bash
   mysub(){
     return 4
   }

   mysub “Argument to subroutine.”
   echo “Subroutine returned $?.”
```

Variables in a script are shared between subroutines and the main program body. To create a local variable, preceded the variable with the keyword **`local`**.

Example 1:

```
#!/bin/bash
mysub(){
   MYVAR=5
   echo "MYVAR in mysub() is $MYVAR."
}

MYVAR=4
echo "MYVAR is initialised to 4."
mysub
echo "MYVAR after mysub() returned is $MYVAR."
```

Example 2:

```
#!/bin/bash
mysub(){
   local MYVAR=5
   echo "MYVAR in mysub() is $MYVAR."
}

MYVAR=4
echo "MYVAR is initialised to 4."
mysub
echo "MYVAR after mysub() returned is $MYVAR."
```

Common subroutines can be stored in a separate script and included in another script that needs the subroutine(s). Create the following scripts.

Script 1: named as `subroutine`

```
#!/bin/bash
#save this script as 'subroutines'

max(){
   if [ $1 -gt $2 ] ; then
        return $1
   else
        return $2
   fi
}

add(){
   return $(($1 + $2))
}
```

Script 2: named as `sourcing`

```
#!/bin/bash

# save this script as 'sourcing'

# checking number of parameters
if [ $# -lt 2 ]; then
  clear
  echo Usage: $0 number_1 number_2
  echo
else

  # use 'source' to include script that contains subroutines
  # this technique is called script sourcing
  # syntax: source /path/to/script

  # the line after this comment includes the script “subroutine”
  # that contains max() and add() subroutines
  source ./subroutines

  max $1 $2
  echo "The maximum is $?."

  add $1 $2
  echo "The sum of $1 and $2 is $?."
fi
```