Shell Programming - Operator

Amit Patel

INPUT OUTPUT

- Until now we have been looking at commands that print out messages. When a command prints a message, the message is called output.
- Now we will look at the different types of output available to shell scripts.
- Specifically, the areas that we will cover are
  - Output to the screen
  - Output to a file
  - Input from a file
  - Input from users

OUTPUT

- Two common commands print messages to the terminal (STDOUT):
  - echo
  - printf
- The echo command is mostly used for printing strings that require simple formatting.
- The printf command is the shell version of the C language function printf. It provides a high degree of flexibility in formatting output.

ECHO

- The most common command used to output messages to the terminal is the echo command.
- Syntax: echo string
- Here string is the string you want printed. For example, the command

  $ echo Hi   OUTPUT: Hi

- You can also embed spaces in the output as follows:

  $ echo Safeway has fresh fruit   OUTPUT: Safeway has fresh fruit

- In addition to spaces, you can embed each of the following in the string:
  - Punctuation marks
  - Variable substitutions
  - Formatting Escape Sequence

ECHO OPTION

OPTION	MEANING
-n	Do not output a trailing newline.
-e	Enable interpretation of backslash escape sequences.
-E	Disable interpretation of backslash escape sequences (this is the default).

ECHO

Embedding Punctuation Marks

- Punctuation marks are used when you need to ask the user a question, complete a sentence, or issue a warning. For example the following echo statement might be the prompt in an install script:

```
$ echo Do you want to install?
```

- You can also use any combination of the punctuation marks. For example, the following command uses the comma (,), question mark (?), and exclamation point(!) punctuation marks:

```
$ echo Eliza, where the devil are my slippers?!?
Eliza, where the devil are my slippers?!?
```

ECHO

Variable Substitution

- Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.
- Variable substitution falls into two categories:
    - Actions taken when a variable has a value
    - Actions taken when a variable does not have a value

Following table contain various possible types of substitution.

ECHO

Form	Description
${var}	Substitue the value of *var*.
${var:-word}	If *var* is null or unset, *word* is substituted for **var**. The value of *var* does not change.
${var:=word}	If *var* is null or unset, *var* is set to the value of **word**.
${var:?message}	If *var* is null or unset, *message* is printed to standard error. This checks that variables are set correctly.
${var:+word}	If *var* is set, *word* is substituted for var. The value of *var* does not change.

ECHO

```
#!/bin/sh
echo ${var:-"Variable is not set"}
echo "1 - Value of var is ${var}“

echo ${var:="Variable is not set"}
echo "2 - Value of var is ${var}"

unset var
echo ${var:+"This is default value"}
echo "3 - Value of var is $var"

var="Prefix"
echo ${var:+"This is default value"}
echo "4 - Value of var is $var"

echo ${var:?"Print this message"}
echo "5 - Value of var is ${var}"
```

This would produce following result –

```
Variable is not set
1 - Value of var is
Variable is not set

2 - Value of var is
Variable is not set

3 - Value of var is
This is default value

4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix
```

ECHO

Formatting with Escape Sequence

- By using escape sequences you can format the output of echo.
- An escape sequence is a special sequence of characters that represents another character.
- When the shell encounters an escape sequence, it substitutes the escape sequence with a different character.
- The echo command understands following formatting escape sequences
  - \n Prints a newline character
  - \t Prints a tab character
  - \c Prints a string without a default trailing newline

ECHO

- The \n escape sequence is usually used when you need to generate more than one line of output using a single echo command.
- For example, the command

```
$ FRUIT_BASKET="apple orange pear"
$ echo -e "Your fruit basket contains:\n$FRUIT_BASKET"
```

Output:

```
Your fruit basket contains:
apple orange pear
```

- generates a list of fruit preceded by a description of the list.
- This example illustrates two important aspects of using escape sequences:
  - The entire string is quoted.

ECHO

- This example illustrates two important aspects of using escape sequences:
  - The entire string is quoted.
  - The escape sequence appears in the middle of the string and is not separated by spaces
- Whenever an escape sequence is used in the input string of an echo command, the string must be quoted to prevent the shell from expanding the escape sequence on the command line.
- You can always rewrite any echo command that uses the \n escape sequence as several echo commands. For example, you can generate the same output as in the previous example using the echo command

```
$ echo "Your fruit basket contains:"
$ echo $FRUIT_BASKET
```

PRINTF

- The printf command is similar to the echo command, in that it enables you to print messages to STDOUT. In its most basic form, its usage is identical to echo. For example, the following echo command:

```
$ echo "Is that a mango?"
```

is identical to the printf command:

```
$ printf "Is that a mango?\n"
```

- The only major difference is that

  The string specified to printf explicitly requires the \n escape sequence at the end of a string, in order for a newline to print. The echo command prints the newline automatically.

PRINTF

- The power of printf comes from its capability to perform complicated formatting by using format specifications.
- Syntax: printf format arguments
- Here, format is a string that contains one or more of the formatting sequences, and arguments are strings that correspond to the formatting sequences specified in format.
- The formatting sequences supported by the printf command are identical like c programming.
- The formatting sequences have the form: %[-]m.nx
- Here % starts the formatting sequence and x identifies the formatting sequences type.
- Depending on the value of x, the integers m and n are interpreted differently. Usually m is the minimum length of a field, and n is the maximum length of a field. If you specify a real number format, n is treated as the precision that should be used.

PRINTF

Letter	Description
s	String
c	Character
d	Decimal number
x	Hexadecimal number
o	Octal number
e	Exponential floating point number
f	Fixed floating point number

PRINTF

```
#!/bin/sh
 printf "%32s %s\n" "File Name" "File Type"
 for i in *;
do
    printf "%32s " "$i"
    if [ -d "$i" ]; then
        echo "directory"
    elif [ -h "$i" ]; then
        echo "symbolic link"
    elif [ -f "$i" ]; then
        echo "file"
```

```
#!/bin/sh

printf "%-32s %s\n" "File Name" "File Type"

 for i in *;

do

    printf "%-32s " "$i"

    if [ -d "$i" ]; then

        echo "directory"

    elif [ -h "$i" ]; then

        echo "symbolic link"

    elif [ -f "$i" ]; then

        echo "file"

 else

        echo "unknown"

   fi;

Done
```

<u>Output</u>

```
File Name                    File Type

RCS                          directory

dev                          directory

humor                        directory
```

read

- Read statement is the shell’s internal tool for taking input from the user.
- It make script interactive.
- It is used with one or more variable.
- Input supplied through the standard input is read into these variables.
- When it use like : read name
- The script pauses at that point to take input from the keyboard.
- Whatever you enter is stored in the variable “name”. Since this is a form of assignment, no $ is used before name.
- The following script use read to take a search string and filename from terminal

read

```
#!/bin/sh

Echo “Enter the pattern to be searched:\c”

Read pname

Echo “Enter the file to be used:\c”

Read fname

Echo “Searching for $pname from file $fname”

Grep “$pname” $fname

Echo “Selected  records shown above”
```

read

- Reading multiple values at a time.

```
read VAR1 VAR2 VAR3 VAR4
```

- Read values from a command

```
read VAR1 VAR2 VAR3 << ( echo OM SAI RAM)
echo "Enter values are $VAR1 $VAR2 $VAR3"
```

- Read user input and give some info to user what he has to give. For this use -p option to display some info when reading value.

```
read -p "Please enter one to ten numbers: " VAL1
```

HERE (<<) DOCUMENT

- An extremely powerful feature of the shell programming is its “Here document”.
- A here document is used to redirect input into an interactive shell script or program.
- The shell redirector of the form : “<<label” forces the input to the specified command to be the shell’s standard input, which is read until the line that contains only “label” is reached.
- We can run an interactive program within a shell script without user action by supplying the required input for the interactive program or interactive shell script. This is why it is called “here document”.
- Syntax:

```
myprogram<<EOF
mycommandA
mycommand B
EOF
```

HERE (<<) DOCUMENT

- This type of redirection tells the shell to read input from the current source (HERE) until a line contain only word (HERE) is seen.
- All of the lines read up to that point are then used as the standard input for a command.
- Files are processed in this manner are commonly called here documents.
- The label can be any string but both label must match.
- A here document is useful when you need to read something from standard input, but you don't want to create a file to provide that input; you want to put that input right into your shell script (or type it directly on the command line). To do so, use the << operator, followed by a special word:

HERE (<<) DOCUMENT

- Use here document as follows:

```
wc -w <<EOF
> This is a test.
> Apple juice.
> 100% fruit juice and no added sugar, colour or preservative.
> EOF
```

- Sample outputs:16
- The <<, reads the shell input typed after the wc command at the PS2 prompts, >) up to a line which is identical to word EOF.

HERE (<<) DOCUMENT

```
cat << %%
> This is a test.
> This test uses a here document.
> Hello world.
> This here document will end upon the occurrence of the string "%%" on a separate line.
> So this document is still open now.
> But now it will end....
> %%
```

HERE (<<) DOCUMENT

- Output

```
This is a test.
This test uses a here document.
Hello world.
This here document will end upon the occurrence of the string "%%" on a separate
line.
So this document is still open now.
But now it will end....
```

HERE (<<) DOCUMENT

- When using here documents in combination with variable or command substitution, it is important to realize that substitutions are carried out *before* the here document is passed on. So for example:

```
$ COMMAND=cat

$ PARAM='Hello World!!'

$ $COMMAND <<%

> `echo $PARAM`

> %
```

- Output

```
Hello World
```

Set Command

- We cant assign values to $1, $2…$9 as we do to any other user defined variables or system variables.
- Like a=10 or b=alpha is valid but $1=dollar and $2=100 is simply invalid.
- This positional parameter are set up by the command line arguments.
- There is another way to assign value to the positional parameter using set command.
- For example: $ set om sai ram
- Above command sets value $1 with om , $2 with sai and $3 with ram.
- To verify we use the echo statement like : $ echo $1 $2 $3
- Give output as : om sai ram
- On giving another set command, the old value of $1,$2….are discarded and the new values get collected.

Set Command

- For example: $ set om sai ram

  $ set BBA BCA BCOM BSC BED BE BFARM

  echo $1 $2 $3 $4 $5 $6 $7

- Above command sets value $1 with BBA , $2 with BCA and $3 with BCOM …to $7 with BFARM.
- To verify we use the echo statement like : $ echo $1 $2 $3
- Give output as : om sai ram
- On giving another set command, the old value of $1,$2….are discarded and the new values get collected.
- The date command by default display the current date and time in following format

  FRI SEP 19 11:30:45 IST 2015

- To display the information in any other format like : FRI 19 SEP 2015

Set Command

- For example: $ set `date`

  Echo $1 $3 $2 $4

- Let us see another way of setting values in positional parameters. Suppose that w

  $ cat test

  Good Morning. My name is Amit Patel.

- We can make set take the values to be assigned to positional parameter from this file by saying

  $ set `cat test`

  $ echo $1 $2 $6 $7

- Good Morning Amit Patel

SHIFT COMMAND

- Shift transfer the contents of positional parameters to its immediate lower numbered one.
- This is done as many time as the statement is called.
- When called once, $2 becomes $1, $3 becomes $2, and so on.
- For example:

```
$ echo "$@"
Wed Aug 19 04:10:30 IST 2015


$ echo $1 $2 $3
Wed Aug 19
```

SHIFT COMMAND

```
Shift
$ echo $1 $2 $3
Aug 19 04:10:30

Shift 2
$ echo $1 $2 $3
04:10:30 IST 2015
```

Note that the contents of the leftmost parameters, $1, are lost every time shift is invoked. In this way, we can access $10 by first shifting it and converting is to $9.

Shift Command

- We have used the set command to set upto 9 words.
- For Example: $ set Good morning Amit Patel. Hello, How are you. Hope you are fine.

  $ echo $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11

  Good morning Amit Patel. Hello, How are you. Hope Good0 Good1
- If we observed the output of last two word – Good0, Good1. This occurred in the output because at a time we can access only 9 positional parameters. When we tried to refer $10 it was interpreted by the shell as if you wanted to output the values $1 and 0. Same is true for $11 as $1 and 1.
- To access word after 9th word, we have to use shift.

  Shift 9

  Echo $1 $2 $3 output: you are fine.

How to find EXIT status?

- In **Unix** when you execute a command or a script, they will exit with a meaning full **exit status** for your understanding purpose. So that we can take necessary actions on the out come(pass, failed or partially completed) of those commands.
- In Unix some command are there, which will not show the output.
- So at this point how we can get exit status of such commands to know about the success of that command execution? For this we have inbuilt variable in Unix which will store the exit status of any command or script executed.
- The exit status is stored in "$?" internal(builtin) variable. The exit status value varies from 0 to 255. Some of the commonly used exit status are as below.
- 0 Successful execution of command
- 1 command fails because of an error during expansion or redirection, the exit status is greater than zero.