**VARIABLES**

## **Variable Substitution**

Used to store a value by name

**To Create**

VARIABLE\_NAME=VALUE

* x=10 if x already exist it is assigned new value
* x=”10 11 22” value containg space use quotes
* don’t use white space around =

**To get the value**

echo ${variable name}

**Varaible Names**

* Only letter, number and underscore are allowed
* 1st character shld be a letter or an underscore
* Variable names are case-sensitive
* Use lowercase variable name as best practise

An uninitialized variable has a "null" value -- no assigned value at all (*not* zero!).

${Variable\_name)

$(command)

Example:

hello="A B C D"

echo $hello # A B C D

echo "$hello" # A B C D

As you see, echo $hello and echo "$hello" give different results.

Why?

# =======================================

# Quoting a variable preserves whitespace.

other\_numbers="1 2 3"

If there is whitespace embedded within a variable, 83 #+ then quotes are necessary.

**Setting up Null variable**

hello=

Note that setting a variable to a null value is not the same as unsetting it, although the end result is the same.

**Uninitialized value**

echo "uninitialized\_variable = $uninitialized\_variable"

Uninitialized variable has null value (no value at all!). however it evaluates as 0 in an arithmetic operation

**Unset the variable**

Var=”cricket”

Unset Var

As it is uninitialized it contains null value, however it evaluates as 0 in an arithmetic operation

## **Bash Variables Are Untyped**

Unlike many other programming languages, Bash does not segregate its variables by "type." Essentially, Bash variables are character strings, but, depending on context, Bash permits arithmetic operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits.

Null variable transformed into an integer.

Undeclared variable transformed into an integer

## **Special variable types**

**Local variable**

* Variables visible only within a code block or function

**Environmental Variable**

* Variables that affect the behavior of the shell and user interface.
* Every time a shell starts, it creates shell variables that correspond to its own environmental variables. Updating or adding new environmental variables causes the shell to update its environment, and all the shell's child processes (the commands it executes) inherit this environment.
* If a script sets environmental variables, they need to be "exported," that is, reported to the environment local to the script
* A script can export variables only to child processes, that is, only to commands or processes which that particular script initiates. A script invoked from the command-line cannot export variables back to the command-line environment. Child processes cannot export variables back to the parent processes that spawned them.

## **Quoting Variables**

When referencing a variable, it is generally advisable to enclose its name in double quotes. This prevents reinterpretation of all special characters within the quoted string -- except $, ` (backquote), and \ (escape).

Use double quotes to prevent word splitting

An argument enclosed in double quotes presents itself as a single word, even if it contains whitespace separators.

List="one two three"

* Double quotes following an echo sometimes escape \. Moreover, the -e option to echo causes the "\t" to be interpreted as a tab.
* echo "hello\\abcd" >>>> hello\abcd

echo -e "x\ty" >>>>> x y

**Quoting can also suppress echo's "appetite" for newlines.**

*# echo $(ls -l)*

*total 8 -rw-rw-r-- 1 bo bo 13 Aug 21 12:57 t.sh -rw-rw-r-- 1 bo bo 78 Aug 21 12:57 u.sh*

*bash$ echo "$(ls -l)"*

*total 8*

*-rw-rw-r-- 1 bo bo 13 Aug 21 12:57 t.sh*

*-rw-rw-r-- 1 bo bo 78 Aug 21 12:57 u.sh*

## **Escape Variables**

* Escaping is a method of quoting single characters. The escape (\) preceding a character tells the shell to interpret that character literally **used with echo and sed**

echo "abcd\

> lfcd"

Abcdlfcd #This will print as one line.

echo "\v\v\v\v" # Prints \v\v\v\v literally

Use the -e option with 'echo' to print escaped characters.

echo –e “\v\v\v\v” Prints 4 vertical tabs

* \n means newline echo –e “my \nname is venkat \ndegree is BE”
* \t means horizontall tab
* \v means vertical tab
* \b means backspace
* \a means alert (beep or flash)
* \0xx translates to the octal ASCII equivalent of 0nn, where nn is a string of digits

**Example**

* echo "Hello" # Hello
* echo "\"Hello\" ... he said." # "Hello" ... he said.
* echo "\$variable01" # $variable01
* echo "The book cost \$7.98." # The book cost $7.98.
* echo "\\" # Results in \ & echo "\" gives syntax error
* variable=\\ echo "$variable" # \
* echo foo\

bar # Newline escaped. #foobar

## **Typing variables: declare or typeset**

The declare or typeset builtins, which are exact synonyms, permit modifying the properties of variables

declare/typeset options

-r readonly [declare -r var1=1] constant variable or radonly var1=1

-i integer [declare -i number]

-a array [declare -a indices] The variable indices will be treated as an array.

-f function(s) [declare –f function name]

-x export [declare -x var3] This declares a variable as available for exporting outside the environment of the script itself. -x var=$value

The declare command can be helpful in identifying variables, environmental or otherwise. This can be especially useful with arrays.

declare | grep HOME

declare | grep variableName output the value of the varible

## **Exit and Exit Status**

* The exit command terminates a script, just as in a C program. It can also return a value, which is available to the script's parent process.
* Every command returns an exit status (sometimes referred to as a return status or exit code). A successful command returns a 0, while an unsuccessful one returns a non-zero value that usually can be interpreted as an error code.
* Likewise, functions within a script and the script itself return an exit status. The last command executed in the function or script determines the exit status.

When a script ends with an exit that has no parameter, the exit status of the script is the exit status of the last command executed in the script (previous to the exit).

#!/bin/bash

COMMAND\_1

# Will exit with status of last command.

exit

echo exit $?

$? reads the exit status of the last command executed

The equivalent of a bare exit is exit $? or even just omitting the exit.

## **Positional Parameter**

* Arguments passed to the scripts from the command line $0 $1 $2 $3…..
* $0 Name of the scripts
* From the command-line, however, $0 is the name of the shell
* $1 $2 $3 …First Second and n Arguments
* $\* & $@ Displays All arguments passed to the scripts
* $# Total number of arguments
* $\_ last argument of previous command
* $! Process id of last background cmd
* $$ process id of current shell
* If a script expects a command-line parameter but is invoked without one, this may cause a *null variable assignment*, generally an undesirable result. One way to prevent this is to append an extra character to both sides of the assignment statement using the expected positional parameter. variable1\_=$1\_ # Rather than variable1=$1
* # This will prevent an error, even if positional parameter is absent
* After $9, the Arguments must be enclosed in brackets, for example, ${10}, ${11}, ${12}.

## **TEST OPERATOR**

[[ ]] (( )) Evalutes exit status

**FILE TEST OPERATOR**

-d directory

-f file

-e file exists

-s file is not zero size

-b block device

-c chater device

-p file is a pipe

-h file is symblolic link

-r file has read -w -x -g -u -k [sticky bit]

**Example**

if [[ -r $file ]]

then

echo "File has read access"

if [[ -f $file ]]

then

echo "File is an ordinary file"

if [[ -s $file ]]

then

echo "File size is not zero"

if [[ -e $file ]]

then

echo "File exists"

**Comparison Operator :**

**if [ "$a" -eq "$b" ] if [[ "$a" == "$b" ]]**

**if [ "$a" -ne "$b" ]**

**if [ "$a" -gt "$b" ] if [[ "$a" > "$b" ]]**

**if [ "$a" -ge "$b" ]**

**if [ "$a" -lt "$b" ] if [[ "$a" < "$b" ]]**

**if [ "$a" -le "$b" ]**

**if [[ "$a" != "$b" ]]**

Bash Shell Find Out If a Variable Is Empty Or Not

-z :: Checks if the given string operand size is zero; if it is zero length, then it returns true.

if [[ -z "$var" ]]

then

echo "\$var is empty"

else

echo "\$var is NOT empty"

fi

#2 Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.

if [ -n $string1 ] # string1 has not been declared or initialized.

then

echo "String \"string1\" is not null."

**str** Checks if **str** is not the empty string; if it is empty, then it returns false.

Logical AND &&

Logical OR operator ||

**Arithmetic operator ::**

Using **let** is similar to enclosing an arithmetic expression in double parentheses, for instance (( expr ))

let arg [arg ...]

$(( 10 + 3 )), result=13

$(( 10 - 3 )), result=7

$(( 10 \* 3 )), result=30

$(( 10 / 3 )), result=3

$(( 10 \*\* 3 )), result=1000

$(( 10 % 3 )), result=1

let "var += 5" results in var being incremented by 5

let "var \*= 4" results in var being multiplied by 4.

/= slash-equal (divide variable by a constant)

‘++` operator is used to increment the value of a variable by 1

$ i=39

$ echo $((++i+10)) Ans : 50

## **IF/THEN CONDITIONS**

#!/bin/bash

echo -n "Enter a number: "

read VAR

**if** [[ $VAR -gt 10 ]]

**then**

echo "The variable is greater than 10."

**fi**

**IF/ELSE**

#!/bin/bash

echo -n "Enter a number: "

read VAR

**if** [[ $VAR -gt 10 ]]

**then**

echo "The variable is greater than 10."

**else**

echo "The variable is equal or less than 10."

**fi**

**if…..elif…else Statement**

#!/bin/bash

echo -n "Enter a number: "

read VAR

**if** [[ $VAR -gt 10 ]]

**then**

echo "The variable is greater than 10."

**elif** [[ $VAR -eq 10 ]]

**then**

echo "The variable is equal to 10."

**else**

echo "The variable is less than 10."

**fi**

**Nested If Statement**

#!/bin/bash

echo -n "Enter the first number: "

read VAR1

echo -n "Enter the second number: "

read VAR2

echo -n "Enter the third number: "

read VAR3

**if** [[ $VAR1 -ge $VAR2 ]]

**then**

**if** [[ $VAR1 -ge $VAR3 ]]

**then**

echo "$VAR1 is the largest number."

**else**

echo "$VAR3 is the largest number."

**fi**

**else**

**if** [[ $VAR2 -ge $VAR3 ]]

**then**

echo "$VAR2 is the largest number."

**else**

echo "$VAR3 is the largest number."

**fi**

**fi**

#!/bin/bash

echo -n "Enter the first number: "

read VAR1

echo -n "Enter the second number: "

read VAR2

echo -n "Enter the third number: "

read VAR3

**if** [[ $VAR1 -ge $VAR2 ]] && [[ $VAR1 -ge $VAR3 ]]

**then**

echo "$VAR1 is the largest number."

**elif** [[ $VAR2 -ge $VAR1 ]] && [[ $VAR2 -ge $VAR3 ]]

**then**

echo "$VAR2 is the largest number."

**else**

echo "$VAR3 is the largest number."

**fi**

## **OPerators**

var++ var— post-increment, post-decrement

++var –var pre-increment,, pre-decrement

## Internal Variables

**$ echo $BASH**

**$ echo $$**

**$ echo $HOME**

**$ echo $HOSTNAME**

**$PATH**

**$PPID**

**$UID**

**$RANDOM**

**$ IFS**

Internal field separator

This variable determines how Bash recognizes fields, or word boundaries, when it interprets character

strings.

var1="a+b+c" IFS=+ echo $var1 # a b c

var2="d-e-f" IFS="-" echo $var2 # d e f

var3="g,h,i"

**AM I Root ??**1 #!/bin/bash

2 # am-i-root.sh: Am I root or not?

3

4 ROOT\_UID=0 # Root has $UID 0.

5

6 if [ "$UID" -eq "$ROOT\_UID" ] # Will the real "root" please stand up?

7 then

8 echo "You are root."

9 else

10 echo "You are just an ordinary user (but mom loves you just the same)."

11 fi

**# An alternate method of getting to the root of matters:**

20

21 ROOTUSER\_NAME=root

22

23 username=`id -nu` # Or... username=`whoami`

24 if [ "$username" = "$ROOTUSER\_NAME" ]

25 then

26 echo "Rooty, toot, toot. You are root."

27 else

28 echo "You are just a regular fella."

29 fi

## **Manipulating String**

**String Length**

stringZ=abcABC123ABCabc

echo ${#stringZ} Ans 15

echo `expr length $stringZ` Ans 15

## **FOR LOOP**

**The Read Command**

Read a line variable  
read a b # first word saved to a & rest saved to b

Read a b c # first to a, second to b, then rest to c

## **While Loop**

**while** **[** condition **]**

**do**

command1

command2

command3

**done**

While loops are frequently used for [reading data line by line from file](https://www.cyberciti.biz/faq/unix-howto-read-line-by-line-from-file/):