

Bash Scripting Knowledge Check

Bash is the default shell for most UNIX systems. Therefore, as a cyber warfare operator it will be necessary for you to understand Bash scripting.

This course will follow the outline below. Each section will have a basic lesson and then one or more knowledge checks per area. Answers can be found in the accompanying documents.

Course Outline

1. Variables
2. Arithmetic
3. Comparisons and Operators
4. If Statements
5. If / Else If / Else Statements
6. For Loops
7. While Loops

Variables

A variable is defined with '=', and there cannot be spaces between the variable name, equal sign, and value of the variable.

Strings can be defined with single or double quotes:

Single quotes: '<text>' *fully* captures the text, meaning the text is taken as literal characters.

Double quotes: "<text>" *partially* captures the text, meaning the text will expand variables.

A pound sign '#' begins a comment, which is not interpreted by Bash. These are simply human-readable notes.

Example:

```
#!/bin/bash
# A simple script to print strings
MY_STR='Hello World!'
echo $MY_STR
echo "My string is: $MY_STR"
echo 'My string is: $MY_STR'
```

Output:

```
Hello World!
My string is: Hello World!
My string is: $MY_STR
```

There are also a number of special variables that Bash can use:

- \$0 - The name of the Bash script
- \$1 - \$9 - The first 9 arguments provided to the Bash script
- \$# - How many arguments were passed to the Bash script
- \$@ - All the arguments supplied to the Bash script
- \$? - The exit status of the most recently run process
- \$\$ - The process ID of the current script
- \$USER - The username of the user running the script
- \$HOSTNAME - The hostname of the machine the script is running on
- \$SECONDS - The number of seconds since the script was started
- \$RANDOM - Returns a different random number each time it is referred to

- `$LINENO` - Returns the current line number in the Bash script

We can save the output of a command to a variable like so: `variable=$(<command>)`. This is known as command substitution.

Knowledge Check

```
#!/bin/bash
MY_STR="Hi there!"
echo 'My string is: $MY_STR'
echo "My string is: $MY_STR"
```

1. What will be the first line output by this script?
2. What will be the second line output by this script?

```
./example_script 0 1 2
```

```
#!/bin/bash
echo $0
echo "Printing $3, $2, $1"
echo $#
```

3. What will be the first line output by this script?
4. What will be the second line output by this script?
5. What will be the third line output by this script?

Arithmetic

The operators `+`, `-`, `*`, `/`, `var++`, `var--`, and `%` can be used to add, subtract, multiply, divide, increment, decrement, and modulus, respectively.

The command `let` evaluates an arithmetic expression and can be used to save the result to a variable.

Example:

```
#!/bin/bash
# Demonstrate usage of let with arithmetic expressions
let a=5+4
echo $a # prints 9
let "a = 5 + 4" # quotes with let allows us to use spaces
echo $a # prints 9
let "a = $1 + 30" # recall $1 is the first argument provided to the script
echo $a # prints the result of $1 plus 30
```

Output:

Calling `./example_script 9` will produce the following:

```
9
9
39
```

The command `expr` also evaluates an arithmetic expression but instead of saving to a variable it prints the answer.

Example:

```
#!/bin/bash
# Demonstrate usage of expr with arithmetic expressions
expr 5 + 4 # you don't use quotes with expr, and must have spaces
expr "5 + 4" # if quotes are used, expr simply prints the expression
expr 5+4 # if spaces are not used, expr simply prints the expression
expr 5 \* 4 # here, Bash requires an escape of the asterisk for multiplication
a=$( expr 5 + 4 ) # expr can be combined with command substitution
echo $a
```

Output:

```
9
5 + 4
5+4
20
9
```

Double parentheses can also be used to evaluate an arithmetic expression, like `expr`. This syntax can be combined with command substitution like so: `variable=$((<expression>))`

Note, you cannot simply write a line like `b=a+3` in Bash like you can in other programming languages. Bash must be given a command to evaluate the expression with mechanisms like `let`, `expr`, or `((<expression>))`. In fact, the error you will receive is `command not found`.

Example:

```
#!/bin/bash
# Demonstrate usage of (( )) with arithmetic expressions
a=$(( 4 + 5 )) # the double parentheses evaluates the arithmetic expression
echo $a # print 9
a=$((3+5)) # with this syntax, we can omit the spaces and it still works
echo $a # prints 8
b=$(( a + $a )) # you can choose whether to use $ with a variable, either works
echo $b # prints 16
(( b++ )) # the double parentheses will evaluate this arithmetic expression
echo $b # prints 17
c=$(( 4 * 5 )) # here, Bash does not need to escape the asterisk for multiplication
echo $c # prints 20
```

Output:

```
9
8
16
17
20
```

This is not truly arithmetic, but `${#var}` is extremely useful. It produces the length of the variable (its number of characters). For example, if `a='Hello World'`, then `${#a}` is 11. If `b=4953`, then `${#b}` is 4.

Knowledge Check

```
./example_script 2 3
```

```
let "num_1 = $1 * $2"
echo $num_1
expr $1 \* $2
expr $1 * $2
num_2=$(( num_1 / $1 ))
echo $num_2
```

1. What will be the first line output by this script?

- ☐ 2
- ☐ 3
- ☐ 6
- ☐ Error

2. What will be the second line output by this script?

- ☐ 2
- ☐ 3
- ☐ 6
- ☐ Error

3. What will be the third line output by this script?

- ☐ 2
- ☐ 3
- ☐ 6
- ☐ Error

4. What will be the fourth line output by this script?

- ☐ 2
- ☐ 3
- ☐ 6
- ☐ Error

```
#!/bin/bash
expr $RANDOM % 100
```

5. What will this script output?

Comparisons and Operators

Bash makes the Boolean comparison OR using double pipes '||' and AND using double ampersands '&&'.

The following conditionals can be used in bash. Note the different usage depending on whether you're comparing numbers or comparing strings.

Description	Numeric Comparison	String Comparison
less than	-lt	<
greater than	-gt	>
equal	-eq	= or ==
not equal	-ne	!=
less or equal	-le	N/A
greater or equal	-ge	N/A

Square brackets are a reference to the command `test`. That command evaluates whatever is in the square brackets and exits with a status. A status of 0 means it exited with success; a status of 1 means it exited with failure. In other words:

0 -> *True*

1 -> *False*

Example:

```
#!/bin/bash
string_a="UNIX"
string_b="GNU"
echo "Are the strings $string_a and $string_b equal?"
[ $string_a = $string_b ]
echo $? # Recall from above, this is the exit status of the most recently run process
```

Output:

```
Are the strings UNIX and GNU equal?
1
```

Example:

```
#!/bin/bash
num_a=100
num_b=100
echo "Is $num_a equal to $num_b?"
[ $num_a -eq $num_b ]
echo $?
```

Output:

```
Is 100 equal to 100?
0
```

In addition to the comparators for numbers and strings shown above, you can use these operators for more advanced functionality:

Description	Operator
length of string is greater than zero (not null)	-n <string>
length of string is zero (empty)	-z <string>
file exists	-e, -a <file>
file is regular i.e. not a directory or device	-f <file>
file size is greater than zero (not empty)	-s <file>
file is a pipe	-p <file>
file is a symbolic link	-h, -L <file>
file is a block device	-b <file>
file is a character device	-c <file>

file is associated with a terminal device	-t <file\>
check if stdin in a given shell is a terminal	-t 0
check if stdout in a given shell is a terminal	-t 1
check read, write, execute permission	-r, -w, -x <file\>

Knowledge Check

```
#!/bin/bash
[ -z $1 ]
echo $?
```

1. What is the output if the script is called like so?: `./example_script Hello`

- ☐ 0
- ☐ 1
- ☐ Hello
- ☐ No output

2. What is the output if the script is called like so?: `./example_script`

- ☐ 0
- ☐ 1
- ☐ Hello
- ☐ No output

If Statements

There are two common ways to write an if statement:

```
#!/bin/bash
if [<test>]; then
    echo "The test returned True"
fi

if [<test>]
then
    echo "The test returned True"
fi
```

It's programmer's choice which way to write the statement. Also, it is optional to indent the contents of the if statement. However, indentation is the accepted convention for readability, so it's the best practice. It becomes even more important with nested if statements.

Example:

```
#!/bin/bash
num_a=100
num_b=200
if [ $num_a -lt $num_b ]; then
    echo "$num_a is less than $num_b!"
fi
```

Output:

100 is less than 200!

Knowledge Check

```
#!/etc/bash
num_a=1000
num_b=1001
if [ $num_a != $num_b ]
then
    echo "$num_a and $num_b are not the same"
fi
```

What is the output of this script?

- ☐ 1000 and 1001 are not the same
- ☐ No output

If / Else if / Else:

For more complicated scenarios, Bash can evaluate an `if` and if that's false, check an 'else if' expression using `elif`. If that's false, it can check another 'else if', and if false, another 'else if', and so on. Finally, if none of the expressions have been true, Bash can execute the commands under `else`.

Example:

```
#!/bin/bash
# Demonstrate the use of if, elif, and else
if [ $1 -ge 18 ]
then
    echo You may go to the party.
elif [ $2 == 'yes' ]
then
    echo You may go to the party but be back before midnight.
else
    echo You may not go to the party.
fi
```

Output:

```
./example_script 23
```

You may go to the party.

```
./example_script 17 yes
```

You may go to the party but be back before midnight.

```
./example_script 17 no
```

You may not go to the party.

Combined with the Boolean operations already described, you can evaluate more complicated situations.

Example:

```
#!/bin/bash
# Check if a file is both readable and has a size greater than zero.
if [ -r $1 ] && [ -s $1 ]; then
    echo This file is useful.
fi
```

To evaluate expressions with OR or AND, wildcards, or REGEX, you can also use double brackets `[[]]`:

Example:

```
#!/bin/bash
if [[ $1 == n* || $1 == N* ]]
then
    echo "The string starts with 'n' or 'N'"
else
    echo "The string does not start with 'n' or 'N'"
fi
```

Knowledge Check

```
#!/etc/bash
if [ $1 -gt $2 ]
then
    echo "$1"
elif [ $2 -gt $1 ]
then
    echo "$2"
else
    echo "--"
fi
```

What does this script do?

For Loops

The for loop is a loop that iterates over each of the items in a given list. For each item in the list it will perform the given set of commands between the `do` and `done`.

Example:

```
#!/bin/bash
# A simple loop that will print out three names
names='Jim Bob John Greg'

for name in $names
do
    echo $name
done
```

Output:


```
Jim
Bob
John
Greg
```

A for loop can also iterate over a series of numbers. The series of numbers is specified between two parentheses and the start value and end value are separated by two periods. Eg. `for value in {1..5}`. Additionally, the value to increment or decrement can also be specified after the end value like this: `for value in {0..10..2}`. Also note that the start value can be greater than the end value in order to count down. Eg. `for value in {5..1}`. Examples showing the output for these different types of ranges are shown below.

Example:

```
#!/bin/bash
for value in {1..5}
do
    echo $value
done
```

Output:

```
1
2
3
4
5
```

Example:

```
#!/bin/bash
for value in {0..10..2}
do
    echo $value
done
```

Output:

```
0
2
4
6
8
10
```

Example:

```
#!/bin/bash
for value in {5..1}
do
    echo $value
done
```

Output:

```
5
4
3
2
1
```

There are a number of other lists that the for loop can iterate over:

- Numeric range specified without braces: `for val in 1 2 3 4 5`
- List of strings not in a variable: `for val in string1 string2 string3`
- Output of a linux command: `for val in $(<command>)`
- C-like for loop: `for ((c=1; c<=5; c++))`

Knowledge Check

```
#!/bin/bash
for val in {1..10}
do
    echo $val
done
```

1. What will be the first line output by this script?
2. What will be the fifth line output by this script?

```
./example_script /usr
```

```
#!/bin/bash
for entry in $(ls $1)
do
    echo $entry
done
```

3. What will be the first line output by this script?
4. Briefly describe what this script is doing

While Loops

The while loop evaluates an expression. As long as the expression is true it keeps executing the commands between `do` and `done`.

Example:

```
#!/bin/bash
counter=1
while [ $counter -le 5 ]
do
    echo $counter
    let "counter = counter + 1"
done
```

Output:

```
1
```

2
3
4
5

In this case the expression evaluated is `$counter -le 5`. As long as \$counter is less than 5 the loop will continue to execute.

Knowledge Check

```
#!/bin/bash
counter=5
while [ counter -gt 0]
do
    echo $counter
done
```

1. What will be the first line output by this script?
2. What will be the fifth line output by this script?

```
#!/bin/bash
counter=1
while [ $counter -lt 20]
do
    echo $counter
    if [ $((($counter % 2)) -eq 0)]; then
        echo "even"
    fi
    let "counter = counter + 1"
done
```

3. What will be the fourth line output by this script?
4. What is the sixth line output by this script?