Hello World Bash Shell Script – Bash Scripting Tutorial

First you need to find out where is your Bash interpreter located. Enter the following into your command line:

```
$ which bash

/bin/bash
```

This command reveals that the Bash shell is stored in **`/bin/bash`**. This will come into play momentarily.

The next thing you need to do is open our favorite text editor and create a file called **`hello_world.sh`**. We will use nano for this step.

```
$ nano hello_world.sh
```

Copy and paste the following lines into the new file:

```
#!/bin/bash

# declare STRING variable

STRING="Hello World"

# print variable on a screen

echo $STRING
```

NOTE: Every bash shell script in this tutorial starts with a shebang: **`#!`** which is not read as a comment. First line is also a place where you put your interpreter which is in this case: **`/bin/bash`**.

Navigate to the directory where your **`hello_world.sh`** script is located and make the file executable:

```
$ chmod +x hello_world.sh
```

Now you are ready to execute your first bash script:

```
$ ./hello_world.sh
```

The output you receive should simply be:

```
Hello World
```

Simple Backup bash shell script

When writing a Bash script, you are basically putting into it the same commands that you could execute directly on the command line. A perfect example of this is the following script:

```
#!/bin/bash

tar -czf myhome_directory.tar.gz /home/linuxconfig
```

This will create a compressed tar file of the home directory for user `linuxconfig`.
The `tar` command we use in the script could easily just be executed directly on the command line.

So, what’s the advantage of the script? Well, it allows us to quickly call this command without having to remember it or type it every time. We could also easily expand the script later on to be more complex.

Variables in Bash scripts

In this example we declare simple bash variable `$STRING` and print it on the screen (stdout) with `echo` command.

```
#!/bin/bash

STRING="HELLO WORLD!!!"

echo $STRING
```

The result when we execute the script:

```
$ ./hello_world.sh
```

```
HELLO WORLD!!!
```

Circling back to our backup script example, let's use a variable to name our backup file and put a time stamp in the file name by using the `date` command.

```
#!/bin/bash

OF=myhome_directory_$(date +%Y%m%d).tar.gz

tar -czf $OF /home/linuxconfig
```

The result of executing the script:

```
$ ./backup.sh

$ ls

myhome_directory_$(date +20220209).tar.gz
```

Now, when we see the file, we can quickly determine that the backup was performed on February 9, 2022.

Global vs. Local variables

In Bash scripting, a global variable is a variable that can be used anywhere inside the script. A local variable will only be used within the function that it is declared in. Check out the example below where we declare both a global variable and local variable. We've made some comments in the script to make it a little easier to digest.

```
#!/bin/bash

# Define bash global variable

# This variable is global and can be used anywhere in this bash script

VAR="global variable"



function bash {
```

```
# Define bash local variable

# This variable is local to bash function only

local VAR="local variable"

echo $VAR

}
```

```
echo $VAR

bash

# Note the bash global variable did not change

# "local" is bash reserved word

echo $VAR
```

The result of executing this script:

```
$ ./variables.sh

global variable

local variable

global variable
```

Passing arguments to the bash script

When executing a Bash script, it is possible to pass arguments to it in your command. As you can see in the example below, there are multiple ways that a Bash script can interact with the arguments we provide.

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

```
# use predefined variables to access passed arguments

#echo arguments to the shell

echo $1 $2 $3 ' -> echo $1 $2 $3'
```

```
# We can also store arguments from bash command line in special array

args=("$@")

#echo arguments to the shell

echo ${args[0]} ${args[1]} ${args[2]} ' -> args=("$@"); echo ${args[0]}
${args[1]} ${args[2]}'
```

```
#use $@ to print out all arguments at once

echo $@ ' -> echo $@'
```

```
# use $# variable to print out

# number of arguments passed to the bash script

echo Number of arguments passed: $# ' -> echo Number of arguments passed: $#'
```

Let's try executing this script and providing three arguments.

```
$ ./arguments.sh Bash Scripting Tutorial
```

The results when we execute this script:

```
Bash Scripting Tutorial  -> echo $1 $2 $3
```

```
Bash Scripting Tutorial  -> args=("$@"); echo ${args[0]} ${args[1]} ${args[2]}

Bash Scripting Tutorial  -> echo $@

Number of arguments passed: 3  -> echo Number of arguments passed: $#
```

Executing shell commands with bash

The best way to execute a separate shell command inside of a Bash script is by creating a new subshell through the `$( )` syntax. Check the example below where we echo the result of running the `uname -o` command.

```
#!/bin/bash

# use a subshell $() to execute shell command

echo $(uname -o)

# executing bash command without subshell

echo uname -o
```

Notice that in the final line of our script, we do not execute the `uname` command within a subshell, therefore the text is taken literally and output as such.

```
$ uname -o

GNU/LINUX

$ ./subshell.sh

GNU/LINUX

uname -o
```

Reading User Input

We can use the `read` command to read input from the user. This allows a user to interact with a Bash script and help dictate the way it proceeds. Here's an example:

```
#!/bin/bash


echo -e "Hi, please type the word: \c "

read  word

echo "The word you entered is: $word"

echo -e "Can you please enter two words? "

read word1 word2

echo "Here is your input: \"$word1\" \"$word2\""

echo -e "How do you feel about bash scripting? "

# read command now stores a reply into the default build-in variable $REPLY

read

echo "You said $REPLY, I'm glad to hear that! "

echo -e "What are your favorite colours ? "

# -a makes read command to read into an array

read -a colours

echo "My favorite colours are also ${colours[0]}, ${colours[1]} and
${colours[2]}:-)"
```

Our Bash script asks multiple questions and then is able to repeat the information back to us through variables and arrays:

```
$ ./read.sh

Hi, please type the word: Linuxconfig.org

The word you entered is: Linuxconfig.org

Can you please enter two words?

Debian Linux

Here is your input: "Debian" "Linux"

How do you feel about bash scripting?

good

You said good, I'm glad to hear that!

What are your favorite colours ?

blue green black

My favorite colours are also blue, green and black:-)
```

Bash Trap Command

The **`trap`** command can be used in Bash scripts to catch signals sent to the script and then execute a subroutine when they occur. The script below will detect a **`Ctrl + C`** interrupt.

```
#!/bin/bash

# bash trap command

trap bashtrap INT

# bash clear screen command
```

```
clear;

# bash trap function is executed when CTRL-C is pressed:

# bash prints message => Executing bash trap subrutine !

bashtrap()

{

echo "CTRL+C Detected !...executing bash trap !"

}

# for loop from 1/10 to 10/10

for a in `seq 1 10`; do

echo "$a/10 to Exit."

sleep 1;

done

echo "Exit Bash Trap Example!!!"
```

In the output below you can see that we try to `Ctrl + C` two times but the script continues to execute.

```
$ ./trap.sh

1/10 to Exit.

2/10 to Exit.

^CCTRL+C Detected !...executing bash trap !

3/10 to Exit.
```

```
4/10 to Exit.

5/10 to Exit.

6/10 to Exit.

7/10 to Exit.

^CCTRL+C Detected !...executing bash trap !

8/10 to Exit.

9/10 to Exit.

10/10 to Exit.

Exit Bash Trap Example!!!
```

Arrays

Bash is capable of storing values in arrays. Check the sections below for two different examples.

Declare simple bash array

This example declares an array with four elements.

```
#!/bin/bash

#Declare array with 4 elements

ARRAY=( 'Debian Linux' 'Redhat Linux' Ubuntu Linux )

# get number of elements in the array

ELEMENTS=${#ARRAY[@]}



# echo each element in array
```

```
# for loop

for (( i=0;i<$ELEMENTS;i++)); do

echo ${ARRAY[${i}]}

done
```

Executing the script will output the elements of our array:

```
$ ./arrays.sh

Debian Linux

Redhat Linux

Ubuntu

Linux
```

Read file into bash array

Rather than filling out all of the elements of our array in the Bash script itself, we can program our script to read input and put it into an array.

```
#!/bin/bash

# Declare array

declare -a ARRAY

# Link filedescriptor 10 with stdin

exec 10<&0

# stdin replaced with a file supplied as a first argument

exec < $1
```

```
let count=0

while read LINE; do

ARRAY[$count]=$LINE
((count++))
done

echo Number of elements: ${#ARRAY[@]}
# echo array's content
echo ${ARRAY[@]}
# restore stdin from filedescriptor 10
# and close filedescriptor 10
exec 0<&10 10<&-
```

Now let’s execute the script and store four elements in the array by using a file’s contents for input.

```
$ cat bash.txt
Bash
```

```
Scripting

Tutorial

Guide

$ ./bash-script.sh bash.txt

Number of elements: 4

Bash Scripting Tutorial Guide
```

Bash if / else / fi statements

Here is a simple `if` statement that check to see if a directory exists or not. Depending on the result, it will do one of two things. Please note the spacing inside the `[` and `]` brackets! Without the spaces, it won't work!

```
#!/bin/bash

directory="./BashScripting"



# bash check if directory exists

if [ -d $directory ]; then

      echo "Directory exists"

else

      echo "Directory does not exist"

fi
```

The output:

```
$ ./bash_if_else.sh
```

```
Directory does not exist

$ mkdir BashScripting

$ ./bash_if_else.sh

Directory exists
```

Nested if/else

It is possible to place an `if` statement inside yet another `if` statement. This is called nesting. Scripts can get a bit complex depending on how many `if` statements deep it is.

```
#!/bin/bash



# Declare variable choice and assign value 4

choice=4

# Print to stdout

echo "1. Bash"

echo "2. Scripting"

echo "3. Tutorial"

echo -n "Please choose a word [1,2 or 3]? "

# Loop while the variable choice is equal 4

# bash while loop

while [ $choice -eq 4 ]; do
```

```
# read user input
read choice
# bash nested if/else
if [ $choice -eq 1 ] ; then

    echo "You have chosen word: Bash"

else

    if [ $choice -eq 2 ] ; then
            echo "You have chosen word: Scripting"
    else

        if [ $choice -eq 3 ] ; then
                echo "You have chosen word: Tutorial"
        else
                echo "Please make a choice between 1-3 !"
                echo "1. Bash"
                echo "2. Scripting"
```

```
            echo "3. Tutorial"
            echo -n "Please choose a word [1,2 or 3]? "
            choice=4
        fi
    fi
fi
done
```

Output from the script:

```
$ ./nested_if_else.sh
         Bash
         Scripting
         Tutorial
Please choose a word [1,2 or 3]? 5
Please make a choice between 1-3 !
         Bash
         Scripting
         Tutorial
Please choose a word [1,2 or 3]? 2
You have chosen word: Scripting
```

Bash Comparisons

Bash can compare two or more values, either integers or strings, to determine if they are equal to each other, or one is greater than the other, etc.

Arithmetic Comparisons

-lt	<
-gt	>
-le	<=
-ge	>=
-eq	==
-ne	!=

Now let's use these operators in some examples.

```
#!/bin/bash

# declare integers

NUM1=2

NUM2=2
```

```
if [ $NUM1 -eq $NUM2 ]; then

     echo "Both values are equal"

else

     echo "Values are NOT equal"

fi
```

The result:

```
$ ./statement.sh

Both values are equal
```

Let’s try changing one of the numbers.

```
#!/bin/bash

# declare integers

NUM1=2

NUM2=1

if [ $NUM1 -eq $NUM2 ]; then

     echo "Both Values are equal"

else

     echo "Values are NOT equal"

fi
```

The result:

```
$ ./statement.sh

Values are NOT equal
```

Let’s add a little more complexity by including an `elif` statement and determing which number is larger.

```
#!/bin/bash

# declare integers

NUM1=2

NUM2=1

if   [ $NUM1 -eq $NUM2 ]; then

     echo "Both values are equal"

elif [ $NUM1 -gt $NUM2 ]; then

     echo "NUM1 is greater than NUM2"

else

     echo "NUM2 is greater than NUM1"

fi
```

The result:

```
$ ./statement.sh

NUM1 is greater than NUM2
```

String Comparisons

=	equal
!=	not equal
<	less then
>	greater then
-n s1	string s1 is not empty
-z s1	string s1 is empty

Let’s try comparing two strings to see if they are equal.

```
#!/bin/bash

#Declare string S1

S1="Bash"

#Declare string S2

S2="Scripting"

if [ $S1 = $S2 ]; then

     echo "Both Strings are equal"

else

     echo "Strings are NOT equal"

fi
```

The result:

```
$ ./statement.sh

Strings are NOT equal
```

And again with both string matching.

```
#!/bin/bash

#Declare string S1

S1="Bash"

#Declare string S2

S2="Bash"

if [ $S1 = $S2 ]; then

    echo "Both Strings are equal"

else

    echo "Strings are NOT equal"

fi
```

The result:

```
$ ./statement.sh

Both Strings are equal
```

Bash File Testing

In Bash, we can test to see different characteristics about a file or directory. See the table below for a full list.

-b filename	Block special file
-c filename	Special character file
-d directoryname	Check for directory existence
-e filename	Check for file existence
-f filename	Check for regular file existence not a directory
-G filename	Check if file exists and is owned by effective group ID.
-g filename	true if file exists and is set-group-id.
-k filename	Sticky bit
-L filename	Symbolic link
-O filename	True if file exists and is owned by the effective user id.
-r filename	Check if file is a readable
-S filename	Check if file is socket
-s filename	Check if file is nonzero size

-u filename	Check if file set-ser-id bit is set
-w filename	Check if file is writable
-x filename	Check if file is executable

The following script will check to see if a file exists or not.

```
#!/bin/bash
file="./file"
if [ -e $file ]; then
    echo "File exists"
else
    echo "File does not exist"
fi
```

The result:

```
$ ./filetesting.sh
File does not exist
$ touch file
$ ./filetesting.sh
File exists
```

Similarly for example we can use `while` loop to check if file does not exist. This script will sleep until file does exist. Note bash negator `!` which negates the `-e` option.

```
#!/bin/bash

while [ ! -e myfile ]; do
# Sleep until file does exists/is created
sleep 1
done
```

Loops

There are multiple types of loops that can be used in Bash, including `for`, `while`, and `until`. See some of the examples below to learn how to use.

Bash for loop

This script will list every file or directory it finds inside the `/var/` directory.

```
#!/bin/bash

# bash for loop
for f in $( ls /var/ ); do
    echo $f
done
```

A `for` loop can also be run directly from the command line, no need for a script:

```
$ for f in $( ls /var/ ); do echo $f; done
```

The result:

```
$ ./for_loop.sh

backups

cache

crash

lib

local

lock

log

mail

metrics

opt

run

snap

spool

tmp
```

Bash while loop

This `while` loop will continue to loop until our variable reaches a value of 0 or less.

```
#!/bin/bash
COUNT=6
# bash while loop
while [ $COUNT -gt 0 ]; do
    echo Value of count is: $COUNT
    let COUNT=COUNT-1
done
```

The result:

```
$ ./while_loop.sh
Value of count is: 6
Value of count is: 5
Value of count is: 4
Value of count is: 3
Value of count is: 2
Value of count is: 1
```

Bash until loop

An `until` loop works similarly to `while`.

```
#!/bin/bash
COUNT=0
# bash until loop
```

```
until [ $COUNT -gt 5 ]; do
        echo Value of count is: $COUNT
        let COUNT=COUNT+1
done
```

The result:

```
$ ./until_loop.sh
Value of count is: 0
Value of count is: 1
Value of count is: 2
Value of count is: 3
Value of count is: 4
Value of count is: 5
```

Control bash loop with input

Here is a example of `while` loop controlled by standard input. Until the redirection chain from STDOUT to STDIN to the `read` command exists the `while` loop continues.

```
#!/bin/bash
# This bash script will locate and replace spaces
# in the filenames
DIR="."
# Controlling a loop with bash read command by redirecting STDOUT as
```

```
# a STDIN to while loop

# find will not truncate filenames containing spaces

find $DIR -type f | while read file; do

# using POSIX class [:space:] to find space in the filename

if [[ "$file" = *[[:space:]]* ]]; then

# substitute space with "_" character and consequently rename the file

mv "$file" `echo $file | tr ' ' '_'`

fi;

# end of while loop

done
```

Bash Functions

This example shows how to declare a function and call back to it later in the script.

```
!/bin/bash

# BASH FUNCTIONS CAN BE DECLARED IN ANY ORDER

function function_B {

    echo Function B.

}

function function_A {

    echo $1
```

```
}
function function_D {
    echo Function D.
}
function function_C {
    echo $1
}
# FUNCTION CALLS
# Pass parameter to function A
function_A "Function A."
function_B
# Pass parameter to function C
function_C "Function C."
function_D
```

The result:

```
$ ./functions.sh
Function A.
```

```
Function B.

Function C.

Function D.
```

Bash Select

The `select` command allows us to prompt the user to make a selection.

```
#!/bin/bash

PS3='Choose one word: '

# bash select
select word in "linux" "bash" "scripting" "tutorial"
do
echo "The word you have selected is: $word"
# Break, otherwise endless loop
break
done

exit 0
```

The result:

```
$ ./select.sh
```

```
linux
bash
scripting
tutorial
```

```
Choose one word: 2
The word you have selected is: bash
```

Case statement conditional

The `case` statement makes it easy to have many different possibilities, whereas an `if` statement can get lengthy very quickly if you have more than a few possibilities to account for.

```
#!/bin/bash
echo "What is your preferred programming / scripting language"
echo "1) bash"
echo "2) perl"
echo "3) phyton"
echo "4) c++"
echo "5) I do not know !"
read case;
#simple case bash structure
# note in this case $case is variable and does not have to
```

```
# be named case this is just an example

case $case in
```

```
echo "You selected bash";;

echo "You selected perl";;

echo "You selected phyton";;

echo "You selected c++";;

exit
```

```
esac
```

The result:

```
$ ./case.sh

What is your preferred programming / scripting language
```

```
bash

perl

phyton

c++

I do not know !
```

```
3

You selected phyton
```

Bash quotes and quotations

Quotations and quotes are important part of bash and bash scripting. Here are some bash quotes and quotations basics.

Escaping Meta characters

Before we start with quotes and quotations we should know something about escaping meta characters. Escaping will suppress a special meaning of meta characters and therefore meta characters will be read by bash literally. To do this we need to use backslash \ character. Example:

```
#!/bin/bash

#Declare bash string variable

BASH_VAR="Bash Script"

# echo variable BASH_VAR

echo $BASH_VAR

#when meta character such us "$" is escaped with "\" it will be read literally

echo \$BASH_VAR

# backslash has also special meaning and it can be suppressed with yet another
"\"

echo "\\"
```

Here's what it looks like when we execute the script:

```
$ ./escape_meta.sh

Bash Script

$BASH_VAR

\
```

Single quotes

Single quotes in bash will suppress special meaning of every meta characters. Therefore meta characters will be read literally. It is not possible to use another single quote within two single quotes not even if the single quote is escaped by backslash.

```
#!/bin/bash


# Declare bash string variable

BASH_VAR="Bash Script"


# echo variable BASH_VAR

echo $BASH_VAR


# meta characters special meaning in bash is suppressed when  using single quotes

echo '$BASH_VAR  "$BASH_VAR"'
```

The result:

```
$ ./single_quotes.sh

Bash Script

$BASH_VAR "$BASH_VAR"
```

Double quotes

Double quotes in bash will suppress special meaning of every meta characters except **$**, **** and **`**. Any other meta characters will be read literally. It is also possible to use single quote within double quotes. If we need to use double quotes within double quotes bash can read them literally when escaping them with ****. Example:

```
#!/bin/bash


#Declare bash string variable

BASH_VAR="Bash Script"


# echo variable BASH_VAR

echo $BASH_VAR


# meta characters and its special meaning in bash is

# suppressed when using double quotes except "$", "\" and "`"


echo "It's $BASH_VAR  and \"$BASH_VAR\" using backticks: `date`"
```

The result:

```
$ ./double_quotes.sh

Bash Script

It's Bash Script and "Bash Script" using backticks: Thu 10 Feb 2022 10:24:15
PM EST
```

Bash quoting with ANSI-C style

There is also another type of quoting and that is ANSI-C. In this type of quoting characters escaped with \ will gain special meaning according to the ANSI-C standard.

\a	alert (bell)	\b	backspace
\e	an escape character	\f	form feed
\n	newline	\r	carriage return
\t	horizontal tab	\v	vertical tab
\\	backslash	\`	single quote
\nnn	octal value of characters (see [http://www.asciitable.com/ ASCII table])	\xnn	hexadecimal value of characters (see [http://www.asciitable.com/ ASCII ta

The syntax for ansi-c bash quoting is: `$' '` . Here is an example:

```
#!/bin/bash

# as a example we have used \n as a new line, \x40 is hex value for @

# and \56 is octal value for .
```

```
echo $'web: www.linuxconfig.org\nemail: web\x40linuxconfig\56org'
```

The result:

```
$ ./bash_ansi-c.sh

web: www.linuxconfig.org

email: web@linuxconfig.org
```

Arithmetic Operations

Bash can be used to perform calculations. Let’s look at a few examples to see how it’s done.

Bash Addition Calculator Example

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

```
let RESULT1=$1+$2

echo $1+$2=$RESULT1 ' -> # let RESULT1=$1+$2'

declare -i RESULT2

RESULT2=$1+$2

echo $1+$2=$RESULT2 ' -> # declare -i RESULT2; RESULT2=$1+$2'

echo $1+$2=$(($1 + $2)) ' -> # $(($1 + $2))'
```

The result:

```
$ ./bash_addition_calc.sh 88 12

88+12=100  -> # let RESULT1=$1+$2

88+12=100  -> # declare -i RESULT2; RESULT2=$1+$2

88+12=100  -> # $(($1 + $2))
```

Bash Arithmetics

Let’s see how to do some basic Bash aritmetics such as addition, subtraction, multiplication, division, etc.

```
#!/bin/bash



echo '### let ###'

# bash addition

let ADDITION=3+5

echo "3 + 5 =" $ADDITION



# bash subtraction

let SUBTRACTION=7-8

echo "7 - 8 =" $SUBTRACTION



# bash multiplication

let MULTIPLICATION=5*8
```

```
echo "5 * 8 =" $MULTIPLICATION

# bash division
let DIVISION=4/2
echo "4 / 2 =" $DIVISION

# bash modulus
let MODULUS=9%4
echo "9 % 4 =" $MODULUS

# bash power of two
let POWEROFTWO=2**2
echo "2 ^ 2 =" $POWEROFTWO


echo '### Bash Arithmetic Expansion ###'
# There are two formats for arithmetic expansion: $[ expression ]
# and $(( expression #)) its your choice which you use

```

```
echo 4 + 5 = $((4 + 5))

echo 7 - 7 = $[ 7 - 7 ]

echo 4 x 6 = $((3 * 2))

echo 6 / 3 = $((6 / 3))

echo 8 % 7 = $((8 % 7))

echo 2 ^ 8 = $[ 2 ** 8 ]
```

```
echo '### Declare ###'
```

```
echo -e "Please enter two numbers \c"

# read user input

read num1 num2

declare -i result

result=$num1+$num2

echo "Result is:$result "
```

```
# bash convert binary number 10001

result=2#10001
```

```
echo $result


# bash convert octal number 16

result=8#16

echo $result


# bash convert hex number 0xE6A

result=16#E6A

echo $result
```

The result:

```
$ ./arithmetic_operations.sh

### let ###

3 + 5 = 8

7 - 8 = -1

5 * 8 = 40

4 / 2 = 2

9 % 4 = 1

2 ^ 2 = 4

### Bash Arithmetic Expansion ###
```

```
4 + 5 = 9

7 - 7 = 0

4 x 6 = 6

6 / 3 = 2

8 % 7 = 1

2 ^ 8 = 256

### Declare ###

Please enter two numbers 23 45

Result is:68

17

14

3690
```

Round floating point number

Here is how to use rounding in Bash calculations.

```
#!/bin/bash

# get floating point number

floating_point_number=3.3446
```

```
echo $floating_point_number

# round floating point number with bash

for bash_rounded_number in $(printf %.0f $floating_point_number); do

echo "Rounded number with bash:" $bash_rounded_number

done
```

The result:

```
$ ./round.sh

3.3446

Rounded number with bash: 3
```

Bash floating point calculations

Using the `bc` bash calculator to perform floating point calculations.

```
#!/bin/bash

# Simple linux bash calculator

echo "Enter input:"

read userinput

echo "Result with 2 digits after decimal point:"

echo "scale=2; ${userinput}" | bc

echo "Result with 10 digits after decimal point:"

echo "scale=10; ${userinput}" | bc

echo "Result as rounded integer:"
```

```
echo $userinput | bc
```

The result:

```
$ ./simple_bash_calc.sh

Enter input:

10/3.4

Result with 2 digits after decimal point:

2.94

Result with 10 digits after decimal point:

2.9411764705

Result as rounded integer:

2
```

Redirections

In the following examples, we will show how to redirect standard error and standard output.

STDOUT from bash script to STDERR

```
#!/bin/bash



echo "Redirect this STDOUT to STDERR" 1>&2
```

To prove that STDOUT is redirected to STDERR we can redirect script's output to file:

```
$ ./redirecting.sh
```

```
Redirect this STDOUT to STDERR

$ ./redirecting.sh > STDOUT.txt

$ cat STDOUT.txt

$

$ ./redirecting.sh 2> STDERR.txt

$ cat STDERR.txt

Redirect this STDOUT to STDERR
```

STDERR from bash script to STDOUT

```
#!/bin/bash


cat $1 2>&1
```

To prove that STDERR is redirected to STDOUT we can redirect script's output to file:

```
$ ./redirecting.sh /etc/shadow

cat: /etc/shadow: Permission denied

$ ./redirecting.sh /etc/shadow > STDOUT.txt

$ cat STDOUT.txt

cat: /etc/shadow: Permission denied

$ ./redirecting.sh /etc/shadow 2> STDERR.txt

cat: /etc/shadow: Permission denied
```

```
$ cat STDERR.txt

$
```

stdout to screen

The simple way to redirect a standard output (stdout) is to simply use any command, because by default stdout is automatically redirected to screen. First create a file **`file1`**:

```
$ touch file1

$ ls file1

file1
```

As you can see from the example above execution of **`ls`** command produces STDOUT which by default is redirected to screen.

stdout to file

To override the default behavior of STDOUT we can use **`>`** to redirect this output to file:

```
$ ls file1 > STDOUT

$ cat STDOUT

file1
```

stderr to file

By default STDERR is displayed on the screen:

```
$ ls

file1  STDOUT
```

```
$ ls file2

ls: cannot access file2: No such file or directory
```

In the following example we will redirect the standard error (stderr) to a file and stdout to a screen as default. Please note that STDOUT is displayed on the screen, however STDERR is redirected to a file called STDERR:

```
$ ls

file1  STDOUT

$ ls file1 file2 2> STDERR

file1

$ cat STDERR

ls: cannot access file2: No such file or directory
```

stdout to stderr

It is also possible to redirect STDOUT and STDERR to the same file. In the next example we will redirect STDOUT to the same descriptor as STDERR. Both STDOUT and STDERR will be redirected to file “STDERR_STDOUT”.

```
$ ls

file1  STDERR  STDOUT

$ ls file1 file2 2> STDERR_STDOUT 1>&2

$ cat STDERR_STDOUT

ls: cannot access file2: No such file or directory

file1
```

File STDERR_STDOUT now contains STDOUT and STDERR.

stderr to stdout

The above example can be reversed by redirecting STDERR to the same descriptor as SDTOUT:

```
$ ls

file1  STDERR  STDOUT

$ ls file1 file2 > STDERR_STDOUT 2>&1

$ cat STDERR_STDOUT

ls: cannot access file2: No such file or directory

file1
```

stderr and stdout to file

Previous two examples redirected both STDOUT and STDERR to a file. Another way to achieve the same effect is illustrated below:

```
$ ls

file1  STDERR  STDOUT

$ ls file1 file2 &> STDERR_STDOUT

$ cat STDERR_STDOUT

ls: cannot access file2: No such file or directory

file1
```

or

```
ls file1 file2 >& STDERR_STDOUT

$ cat STDERR_STDOUT
```

```
ls: cannot access file2: No such file or directory

file1
```

Closing Thoughts