



# Shell “Bash” Scripting in GNU/Linux



Ahmed Elmayyah

Head of the Linux Committee, OSC



# What is a shell?

The simplest and most direct way for the user to use their machine.

BASH = Bourne Again SHell



**BASH**  
THE BOURNE-AGAIN SHELL



OS Layers

# What is a shell script and why use one?

A script runs a series of shell commands to:



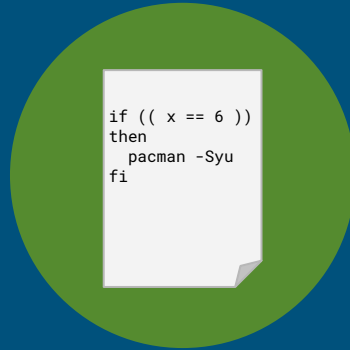
## Process files

Reading files, checking if files exist, etc..



## Automate tasks

Such as backing data up, automating package installations, automating updates, etc..



## Organise commands

Use Linux commands inside programs, as well as controlling the workflow of your system and commands.

And much more!

# Written: What is a shell script and why use one?

---

Running a series of shell commands to:

- Process files
  - Reading files
  - Checking if files exist
  - Checking if files have changed
- Automate tasks
  - Automating the backing up of data
  - Automating package installations
  - Automating updates
- Use Linux commands inside a program
  - Use Linux commands inside programs
  - Controlling the workflow of your system and commands.

# Command Line Format

---

Options modify the command's behaviour on the arguments.

EXAMPLES:

[COMMAND]

[OPTION]

[ARGUMENTS]

ls

-ld, -la, -lA

file or directory

rm

-r, -f, -ir

file or directory

binwalk

-B

file

You can find what a command's options are and what it does by running :

1- `man command`

2- `info command`

3- `command --help`

# CAUTION!

- Never run a Linux command unless you know what it does
- Take care if a command has a -f command as an option, it could mean “by force”.
- Never run any of the following commands unless you know what you’re doing:

`rm -rf /` or `rm -rf /*` → Deletes everything on your system, including Windows files.

`:( ){ :|:&} ;:` (aka Fork Bomb) → Keeps executing a function that calls itself until the system crashes.

`command > /dev/sda` → Data loss on the specified block.

`mkfs.ext3 /dev/sda` → Formats the specified block to ext3, don’t do so unless intended.

# Basic Useful Utilities

**cat** Reads the content of a file.

Example: `cat file.txt`

**grep** Searches for a string or pattern inside a file.

Example: `grep "flag" file.txt`

**head/tail** Display the first/last 10 lines of a file. Examples: `head file.txt / tail file.txt`

**tr** Replaces occurrences of a character with another. Examples: `cat file.txt | tr 'a' 'b'`

## IO redirection

**|** Pipes the output of one command to the input of another. Example: `cat file.txt | grep 'a'`

**>** Redirects the output of a command to a file Example: `grep "error" log.txt > err.txt`

**>>** Appends the output of a command to a file Example: `grep "error" log2.txt >> err.txt`

# Let's Solve!

- Tr: #1 #2
- Head: #1 #2
- Tail: #1 #2
- Grep: #1 #2 #3



# Bash Scripting



*I/O*

## Input and Output

---

Input, output, and passing arguments to shell scripts.



*x y*  
*arr*

## Variables

---

Using variables such as integers and strings to store data.



× −  
÷ +

## Arithmetic

---

Mathematical operations.

# Bash Scripting



```
if  
then  
else  
fi
```

## Conditionals

---

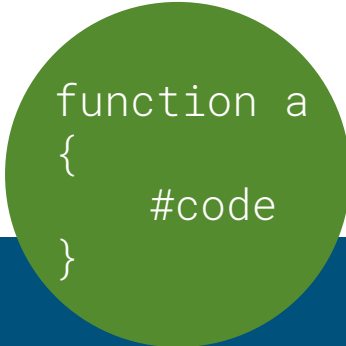
Flow control.



## Loops

---

Repeating the executions of commands.



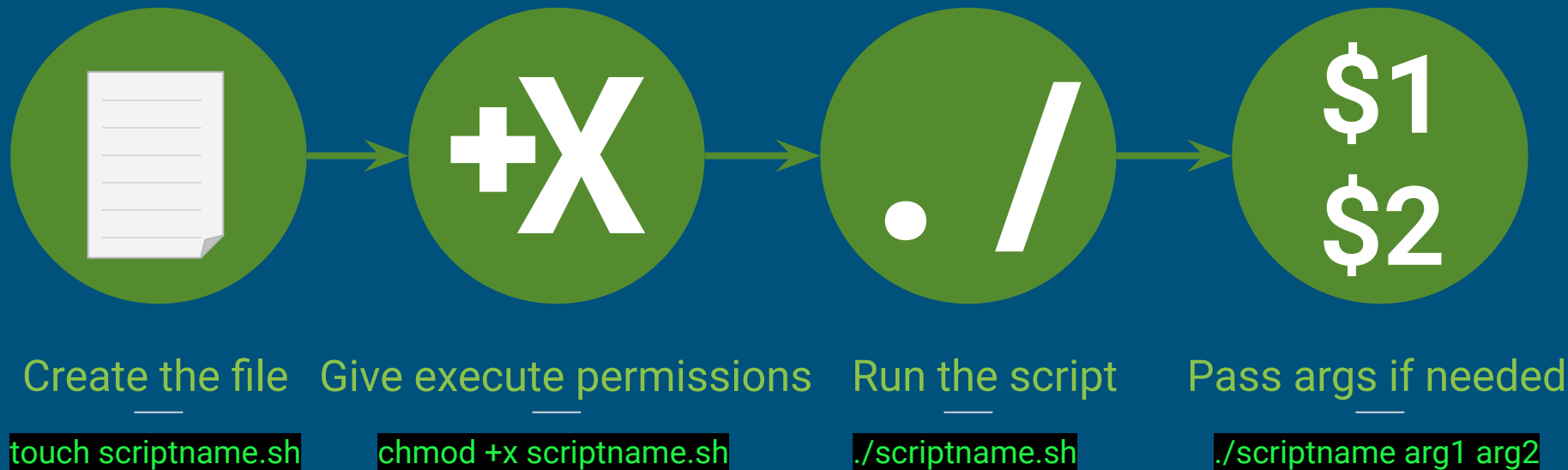
```
function a  
{  
    #code  
}
```

## Functions

---

Organising your script by dividing it into blocks.

# How to begin writing a shell script?



# Written: How to begin writing a shell script?

---

- Scripts are a series of commands running after each other.
- The shell reads the script line by line and executes the commands immediately.
  - Don't put 2 commands in the same line, if you do so then make sure to put a semicolon between them. Example: `read x; echo $x`
- You can start running any text editor and creating a text file (any extension works but let's make it .sh to make things clean).
- Usually, when files are made they don't have execution permissions. Give your file the permission by running the following command: `chmod +x SCRIPTNAME.sh`
- After you've written the script, you can run it by running: `./SCRIPTNAME.sh`
- To pass arguments to the script: `./SCRIPTNAME.sh argument1 argument2`

# Input and Output in Bash

---

## Input

- Input is referred to as STDIN
- You can pass arguments to a bash script as input.
- To prompt input to the user, you use the command: `read` which is roughly equivalent to `scanf()` in C or `input()` in Python.

## Output

- Output is referred to as STDOUT
- To print output to the user, you use the command: `echo` which is roughly equivalent to `printf()` in C or `print()` in Python.

# Variables in Bash

---

- Variables in bash hold values, which could be a number, a character, or a string of characters.
- Variables names are case-sensitive and can't start with a number, but can start with an underscore.
- To assign a value to a variable:
  - `varname="text with spaces"`
  - `varname='text with spaces without any processing'`
  - `varname=textwithoutspaces`
  - `varname=20`
- BASH does **NOT** support floating point integers natively.

# Variables in Bash

---

- You can also use `\` as an escape character to prevent certain characters from being processed by the shell such as the space character:
  - `x=text with spaces` → Syntax Error
  - `x=text\ with\ spaces` → value of x = "text with spaces"
- `varname` → refers to the variable.
- `$varname` → refers to the value of the variable.

# The Difference between ' ' and " "

- " " → Interprets what's inside it, including any expressions, variables, etc..
- ' ' → Interprets what's inside it literally, without calculating or expanding expressions.

```
GNU nano 2.9.8 script2.sh

x=2
echo "$x"
echo $x
echo x
echo 'x'
echo '$x'
```

## Output

```
2
2
x
x
$x
```



# Using Passed Arguments

- You can pass arguments to the script by running: `./SCRIPTNAME.sh argument1 argument2`

To use these arguments as variables, you can access their values by using `$X` where `X` is the order of the argument.

Example:

```
./script1.sh 452 SHELL_SCRIPTING
```

Output:

```
satharus@Argon: ~$ ./script1.sh 452 SHELL_SCRIPTING
452
SHELL_SCRIPTING
```

```
GNU nano 3.2 script1.sh

echo $1
echo $2
```

# Small Example

- What's the output? If we run `./script1.sh 5`
- What's the difference between lines 3 and 4?

```
GNU nano 2.9.8 script1.sh

echo $1
read x
echo x
echo $x

echo $2
|
```

# Small Example: Solution

- Output:
- The third line prints x as a character as that is how it interpreted it. The fourth line prints the value of the variable named x.

```
[satharus@Argon Desktop]$ ./script1.sh 5
5
Y ← User input
x
Y
  ← Notice the space, because $2 is empty
[satharus@Argon Desktop]$ |
```

# Let's Solve!

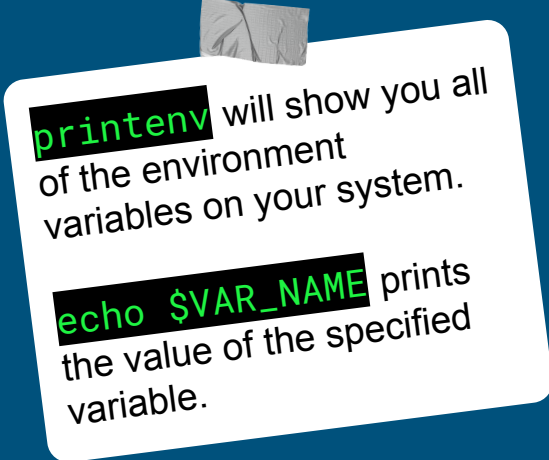
- A Personalized Echo:
  - Write a Bash script which accepts a name as input and displays a greeting: "Welcome (name)"
- A Personalized Echo Extra:
  - Write a Bash script which accepts a name as an ARGUMENT and displays a greeting: "Welcome (name)"

# Environment Variables

- Global system variables accessible by all the processes running under the operating system.
- Environment variables are useful to store system-wide values such as the directories to search for the executable programs (PATH).

## Variables Include:

- |                |                      |
|----------------|----------------------|
| • BASH_VERSION | Bash version         |
| • HOST_NAME    | Host name            |
| • HOME         | Home directory       |
| • PATH         | Executable locations |
| • TERM         | Default terminal     |
| • SHELL        | Default shell        |
| • EDITOR       | Default text editor  |



`printenv` will show you all of the environment variables on your system.

`echo $VAR_NAME` prints the value of the specified variable.

# Arithmetic in Bash

- You can do 6 basic arithmetic operators in Bash:
  - `a + b`      addition      (a plus b)
  - `a - b`      subtracting      (a minus b)
  - `a * b`      multiplication      (a times b)
  - `a / b`      integer division      (a divided by b)
  - `a % b`      modulo      (the integer remainder of a divided by b)
  - `a ** b`      exponentiation      (a to the power of b)
- Arithmetics can be done using the expression: `$( (expression) )`
  - Example: `a=$( (5 - 3 + $b) )`
  - Which means: variable `a` is equal `=` to the value of `$( )` the expression `(5 - 3 + $b)`

# Let's Solve!

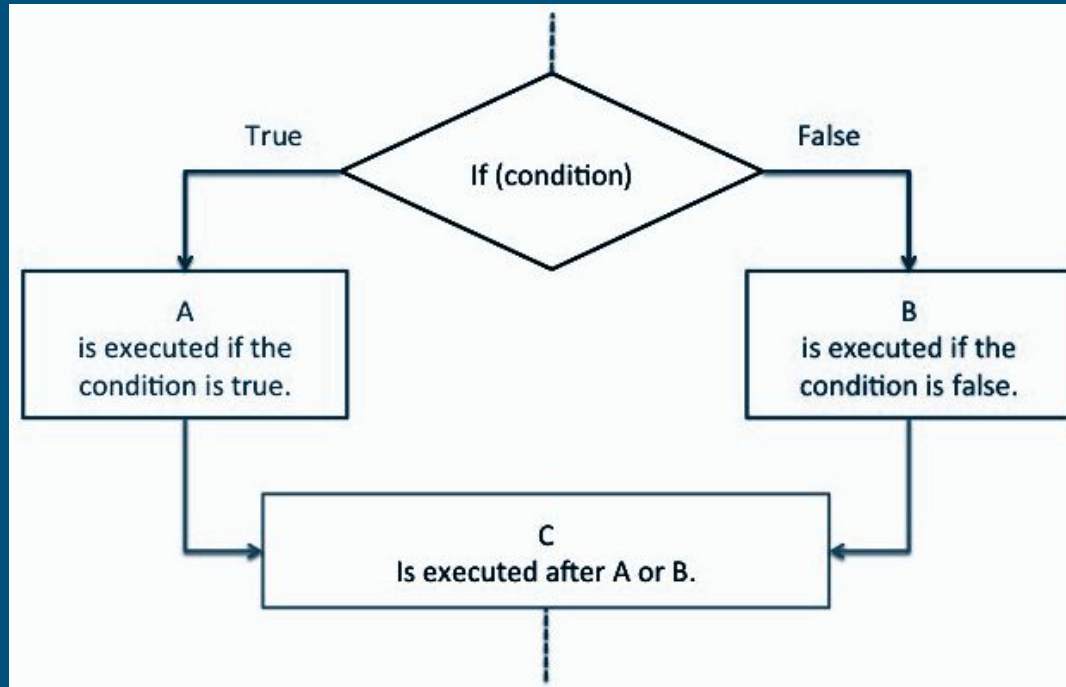
---

- The World of Numbers:
  - Given two integers, X and Y , find their sum, difference, product, and quotient.
- Extra:
  - Write a Bash script that takes 3 integers as arguments and computes their average, prints it, and then prints the average to the power of 2.
  - (After finishing conditionals)  
Write a Bash script that reads an equation in the form of 'A op B' where op can be any bash operator and outputs the result.

BREAK



# Conditionals



# Conditionals in Bash

- If conditions are used to control the program's flow.

## Conditional in C

```
GNU nano 2.9.8 C.c
if (x == 5)
{
    //DoSomething
}
```

## Conditional in Bash

```
GNU nano 2.9.8 Bash.sh
if [[ $x -eq 5 ]]
then
    #DoSomething
fi
```

“{” ‘opening brace’ is roughly equivalent to **then**

”}” ‘closing brace’ is roughly equivalent to **fi**

# More: Conditionals in Bash

```
if (x == "String")
{
    printf("1");
}

else if (x == "String 2")
{
    printf("2");
}

else
{
    printf("3");
}
```

C

```
if [[ $x = "String" ]]
then
    echo 1
elif [[ $x = "String 2" ]]
then
    echo 2
else
    echo 3
fi
```

Bash

# How to write conditionals in Bash

---

1. Start a condition with `if [[ condition ]]`
2. The next line contains `then` which is roughly equivalent to '{'
3. Write the commands that will execute if the condition is true.
4. End your condition with `fi` which is roughly equivalent to '}'
  - 4.1. Or start an `elif [[ condition ]]`, with `then` in the line after it.
    - 4.1.1. Write the commands that will execute if the elif condition is true.
    - 4.1.2. End your conditionals with `fi`
  - 4.2. Or start an `else`, with **NO** `then` in the line after it
    - 4.2.1. Write the commands that will execute if the else condition is true.
    - 4.2.2. End your conditionals with `fi`

# Usual Way of Comparing Numerical Variables

- General expression: `if [[ $var/value -condition $var/value ]]`

Expression in C	Expression in Bash	Evaluates to true when:
<code>a == b</code>	<code>\$a -eq \$b</code>	a is equal to b
<code>a != b</code>	<code>\$a -ne \$b</code>	a is not equal to b
<code>a &lt; b</code>	<code>\$a -lt \$b</code>	a is less than b
<code>a &gt; b</code>	<code>\$a -gt \$b</code>	a is greater than b
<code>a &gt;= b</code>	<code>\$a -ge \$b</code>	a is greater than or equal to b
<code>a &lt;= b</code>	<code>\$a -le \$b</code>	a is less than or equal to b

# Another Way of Comparing Numerical Variables

- You can use `(( ))` instead of `[[ ]]` to use C syntax
- General expression: `if (( var/value comparison var/value ))`

Expression in C	Expression in Bash	Evaluates to true when:
<code>a == b</code>	<code>a == b</code>	a is equal to b
<code>a != b</code>	<code>a != b</code>	a is not equal to b
<code>a &lt; b</code>	<code>a &lt; b</code>	a is less than b
<code>a &gt; b</code>	<code>a &gt; b</code>	a is greater than b
<code>a &gt;= b</code>	<code>a &gt;= b</code>	a is greater than or equal to b
<code>a &lt;= b</code>	<code>a &lt;= b</code>	a is less than or equal to b

# Comparing String Variables

- General expression: `if [[ var/value comparison var/value ]]`

Expression in C	Expression in Bash	Evaluates to true when:
<code>a == b</code>	<code>\$a = \$b</code>	a is the same as b
	<code>\$a == \$b</code>	
<code>a != b</code>	<code>\$a != \$b</code>	a is different from b
<code>strlen(a) == 0</code>	<code>-z \$a</code>	a is empty

# Combining Conditions

- You can combine conditions in Bash so that the overall condition evaluates based on them

Expression in C	Expression in Bash	Evaluates to true when:
(cond. A    cond. B)	[[ cond. A ]]    [[ cond. B ]]	A OR B is true
	[[ cond. A    cond. B ]]	
(cond. A && cond. B)	[[ cond. A ]] && [[ cond. B ]]	A AND B is true
	[[ cond. A && cond. B ]]	
(!cond. A)	[[ ! cond. A ]]	A is false



# Another Way of Combining Conditions

- You can use `(( ... ))` instead of `[[ ... ]]`

Expression in C	Expression in Bash	Evaluates to true when:
<code>(cond. A    cond. B)</code>	<code>(( cond. A ))    (( cond. B ))</code>	A OR B is true
	<code>(( cond. A    cond. B ))</code>	
<code>(cond. A &amp;&amp; cond. B)</code>	<code>(( cond. A )) &amp;&amp; (( cond. B ))</code>	A AND B is true
	<code>(( cond. A &amp;&amp; cond. B ))</code>	
<code>(!cond. A)</code>	<code>(( ! cond. A ))</code>	A is false

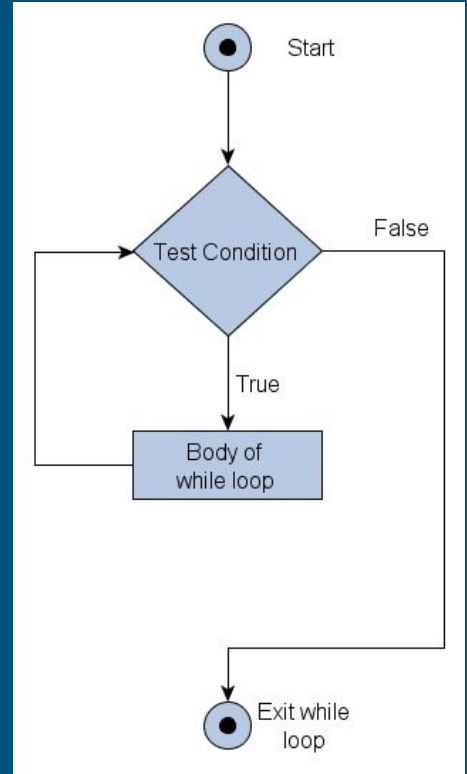
# Let's Solve!

---

- Getting started with conditionals:
  - If the character is 'Y' or 'y' display "YES". If the character is 'N' or 'n' display "NO". No other character will be provided as input.
- More on Conditionals:
  - Given three integers (X, Y, and Z) representing the three sides of a triangle, identify whether the triangle is Scalene, Isosceles, or Equilateral.
- Comparing Numbers:
  - Given two integers, X and Y, identify whether  $X < Y$  or  $X > Y$  or  $X = Y$ .



# Loops

- Loops are used to repeat a process/commands a certain no. of times.
- There are 3 types of loops in Bash (for, while, and until).
- The following slides cover for and while loops only, as until loops are very similar to while and are not used *that* differently.



# Loops in Bash

- For loops are used to loop in a certain range/array.

You can type the  char by pressing the  key (the key left to the "1" key)

## For loop in C

```
GNU nano 2.9.8 C.c

int x;

scanf ("%i",&x);
for (int i = 1; i <=x; i++)
{
    printf("%d\n", i);
}
```

## For loop in Bash

```
GNU nano 2.9.8 Bash.sh

read x
for i in `seq 1 $x`
do
    echo $i
done
```

`seq 1 $x` means "sequence from 1 to the value of x, '\$x' can be replaced with any other value.

Ex: `seq 1 12`

# More: Loops in Bash

- While loops keep repeating a block of commands until the condition becomes false

## While loop in C

```
GNU nano 2.9.8 C.c

int x = 1;

while (x < 11)
{
    printf("%d\n", x);
    x++;
}
```

## While loop in Bash

```
GNU nano 2.9.8 Bash.sh

x=1
while [[ $x -ne 11 ]]
do
    echo $x
    let x+=1
done
```

# More: Loops in Bash

- General Syntax:

## While loop

Bash.sh

```
while [[ CONDITION ]]
do
    #DoSomething
done
```

## For Loop

Bash.sh

```
for var in RANGE
do
    #DoSomething
done
```

# More: Loops in Bash

- Break and continue statements

```
GNU nano 2.9.8 Bash.sh
while [[ x -lt 10 ]]
do
    read i

    if [[ i -eq 0 ]]
    then
        break
    fi
    echo $i
done

#When the user enters 0,
#the code continues to run
#outside the loop
echo "break sent me here"
```

```
GNU nano 2.9.8 Bash.sh
while [[ x -lt 10 ]]
do
    read i

    if [[ i -eq 0 ]]
    then
        echo "Skipping the rest of the code!"
        continue
    fi
    echo $i
done

#When the user enters 0,
#the code skips the above
#line of code and continues
#to the next iteration
done
```

# Let's Solve!

---

- Looping with Numbers:
  - Use for loops to display the natural numbers from 1 to 50.
- Looping and Skipping:
  - Use for loops to display only odd natural numbers from 1 to 99.
- Compute the Average:
  - Given integers, compute their average correct to three decimal places.
    - Try solving it on your own, if you can't you can search for the `bc` command which can be used to print floating point variables.

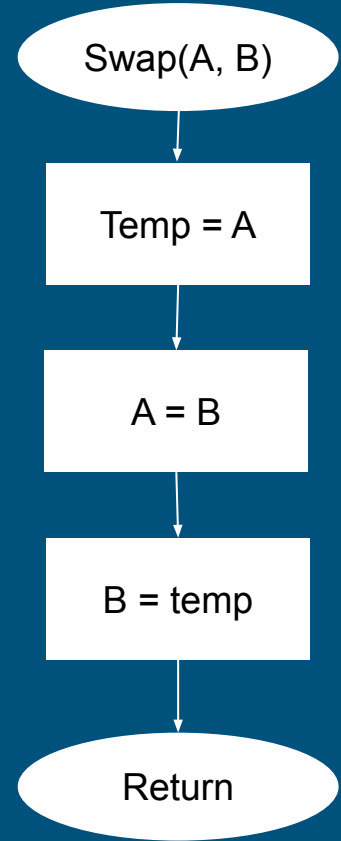
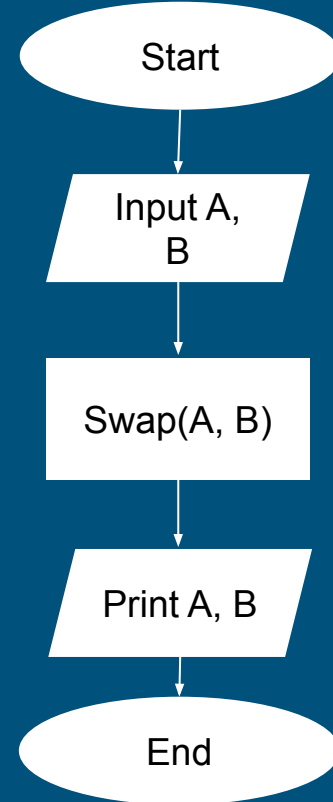
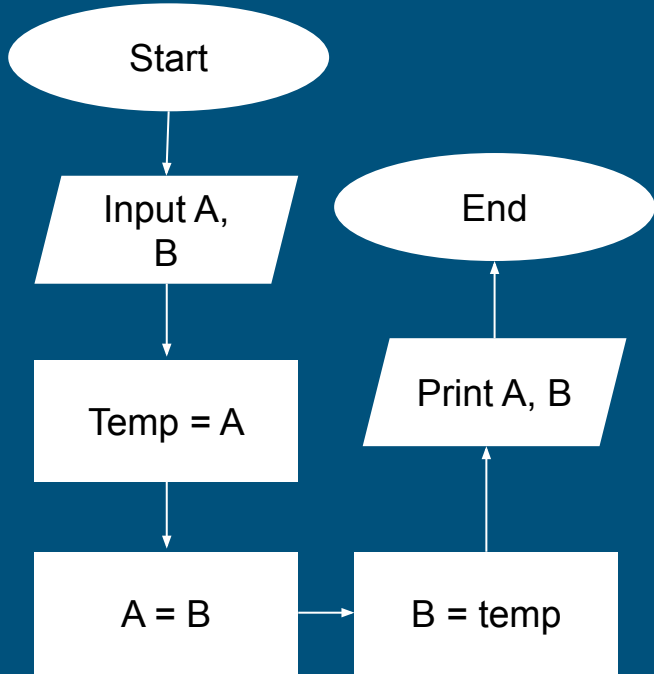


BREAK

# Functions

What's easier?

A B



# Functions in Bash

- You may have guessed that B is more organised and easier to write, which is true as it contains the least amount of repeated code.
- You can write functions in Bash to organise your code.
- You can also pass arguments to functions like you can pass them to scripts.

```
GNU nano 2.9.8 Bash.sh

function NAME #Function Definition
{
    #DoThings
}

NAME #Function call
#This makes what was written
#in the function body run here
```

# Alternatively: Functions in Bash

- This is just an alternative method of defining a function.

```
GNU nano 3.2      Bash.sh
NAME() #Function Definition
{
    #DoThings
}

NAME #Function call
```

# More: Functions in Bash

- Writing a function
- This is a function that prints “Hello!” 5 times.
- Output:

```
satharus@Argon: ~/Desktop$ ./Bash.sh
Hello!
Hello!
Hello!
Hello!
Hello!
```

```
GNU nano 2.9.8 Bash.sh

function hello
{
    for i in `seq 1 5`
    do
        echo "Hello!"
    done
}

hello
```

# More: Functions in Bash

- Passing arguments to a function and using them
- To use the arguments as variables, you can access their values by using `$X` where `X` is the order of the argument passed to the `fn`.
- This is a function that adds 2 numbers and prints them to the user
- Output:

```
GNU nano 2.9.8 Bash.sh

function add
{
    echo $(( $1 + $2 ))
}

add 3 5
```

```
satharus@Argon: ~/Desktop$ ./Bash.sh
8
```

# Let's Solve!

- ENGLISH\_CALC Exercise:

- In this exercise, you will need to write a function called ENGLISH\_CALC which can process sentences such as:
  - '3 plus 5'  $\rightarrow$  '3 + 5 = 8'
  - '5 minus 1'  $\rightarrow$  '5 - 1 = 4'
  - '4 times 6'  $\rightarrow$  '4 \* 6 = 24'

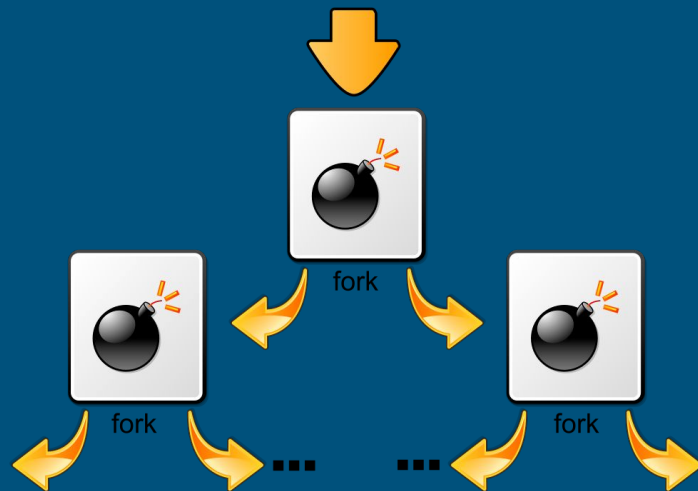
# The Fork Bomb

```
:( ) { : | :& } ; :
```

```
:( )      # Create a function named ' : '
{         # Start of the function body

    : | :& # Calls itself, once in the foreground
           # and once in the background
}         # End of the function body

:         # Function call
```





# References

---

- [LearnShell](http://www.learnshell.org/en/): <http://www.learnshell.org/en/>
- [TLDP](http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html): <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>
- [Ryan Tutorials](https://ryanstutorials.net/bash-scripting-tutorial/): <https://ryanstutorials.net/bash-scripting-tutorial/>
- [DevHints](https://devhints.io/bash): <https://devhints.io/bash>
- [LearnXinYMinutes](https://learnxinyminutes.com/docs/bash/): <https://learnxinyminutes.com/docs/bash/>
- [Bash Hackers](http://wiki.bash-hackers.org/): <http://wiki.bash-hackers.org/>
- [Use Linux as your OS\(Bucky\)](#)
- [Useful Linux Commands\(Quidsup\)](#)
- [Linux Family tree](#), in case you're curious :D

# *Thank you!*



[facebook.com/groups/osc.troubleshoot](https://facebook.com/groups/osc.troubleshoot)