

## **Tema 9 – *Shell Script***

CFGs DAW – Sistemas Informáticos

# ¿Qué es un *script*?

## Script

- Fichero que automatiza la ejecución de tareas (evitando que las realice individualmente el usuario)
  - Conjunto de comandos a ejecutar
  - E.g. → .bashrc
- Se utilizan lenguajes de *scripting*:
  - Bash → para entornos Linux.
  - Python.
  - PowerShell.
  - JavaScript.
  - Etc.

```
GNU nano 4.8                                     bash.sh
#!/bin/bash
read -p "Enter number: " x
if [ $x -eq 5 ]
then
    echo "Number matched.. End of for loop!"
    exit 0
fi
echo "Number doesn't match..."
```

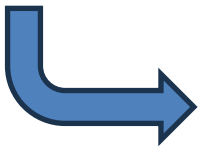
<https://tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide>

<https://www.gnu.org/software/bash/manual/bash.html>

Cheatsheet → <https://devhints.io/bash>

## ¿Qué es un *script*? → Utilidad

- Todos los viernes a las 22:00 tengo que realizar la copia de seguridad de muchos directorios del sistema.
- Tengo que ejecutar una serie de comandos para compilar e instalar *software* en cada uno de los 100 ordenadores de la empresa.
- Es necesario separar la información contenida en columnas en un fichero de texto de 100 MB, pasando todos los tabuladores a espacios
- ...



Lo haré a mano



PROGRAMARÉ UN *SCRIPT*



# ¿Cómo creo un *script* en Bash? ¿Cómo lo ejecuto?



Contenido del fichero

```
#!/bin/bash  
echo "hola mundo"
```

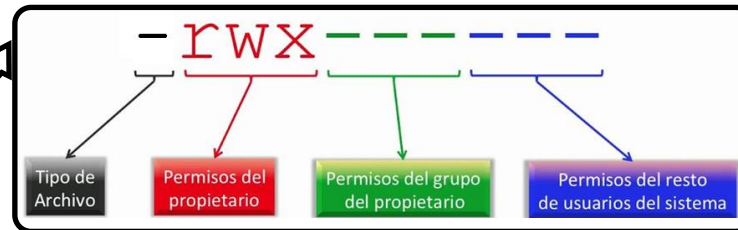
NombreDelScript.sh

Ejecución  
del script

I. \$ bash NombreDelScript.sh

II. \$ ./NombreDelScript.sh

```
A11PC24 $ ./NombreDelScript.sh  
hola mundo
```



# Variables

Un lugar o espacio en la memoria del sistema que puede contener un valor o no, en cuyo caso diríamos que la variable está vacía, y que puede cambiar de valor en un determinado momento.

- Almacenan valores que servirán para los programas
- Para configurar ciertos parámetros del entorno de trabajo

Pueden ser:

- **Locales:** sólo se verán desde la *shell* que se está ejecutando en ese momento
  - Asignar valor a la variable → **NombreVariable = “valor”**
  - Obtener el contenido (valor) de la variable → **echo \${NombreVariable}**
    - También se podría, pero mejor la anterior → **echo \$NombreVariable**
- **De entorno:** se verán desde cualquier parte del sistema (desde la *Shell* que la genera y desde cualquiera de las *shells* generadas a partir de la *Shell* primera -*subshells*-):
  - Por defecto, cualquier variable va a ser local → para convertirla en variable de entorno: **“export NombreVariable”**
  - Suelen escribirse en mayúsculas (no obligatorio)

# Variables (II)

## Uso de comillas

<https://atareao.es/tutorial/scripts-en-bash/variables-en-bash/>

miVariable=valor



miVariable= un valor



→ Necesitamos usar comillas

{ ' ,  
" "

```
#!/bin/bash
variable1=Juan
variable2='Esta es la casa de $variable1'
echo $variable2
```



Esta es la casa de \$variable1

```
#!/bin/bash
variable1=Juan
variable2="Esta es la casa de $variable1"
echo $variable2
```



Esta es la casa de Juan

¿Carácter de escape? → \ → Anula el significado especial del carácter que va detrás

## Variables (III)

```
#!/bin/bash

nombre=Hacker
#Esto es un comentario al código
read -p "Introduce tu edad: " edad

echo "Hola ${nombre}, tienes ${edad} años"
```



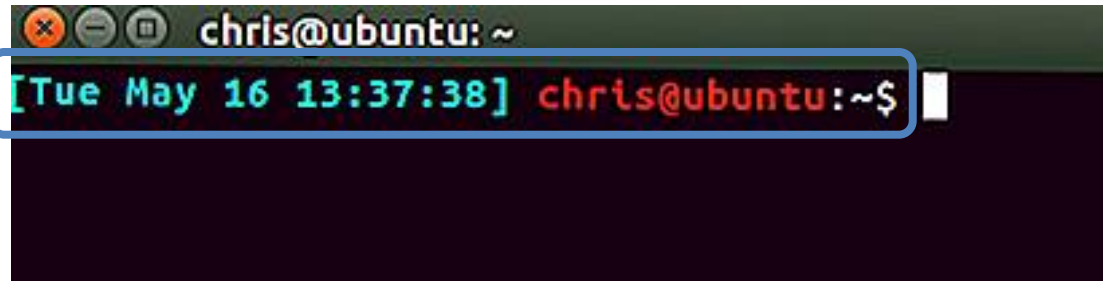
```
A11PC24 $ ./prueba
Introduce tu edad: 18
Hola Hacker, tienes 18 años
```

Además, en el ejemplo:

- Leemos de teclado (*read*)
- Comentarios al código (#)

# Variables predefinidas (algunas)

- \$HOME → Directorio del usuario
- \$HOSTNAME → Nombre de la máquina
- \$PID → Identificador del proceso actual
- \$PWD → Directorio en el que estamos situados
- \$RANDOM → Número aleatorio
- \$SECONDS → Tiempo que lleva encendido el sistema, en segundos
- \$UID → Identificador del usuario actual
- \$PS1 → Prompt (información antes del cursor de la terminal) primario

A screenshot of a terminal window with a dark background. The title bar shows window control icons and the text 'chris@ubuntu: ~'. The terminal content shows a green timestamp '[Tue May 16 13:37:38]' followed by a red prompt 'chris@ubuntu:~\$' and a white cursor. A blue line connects the 'Prompt' text in the list above to the prompt in the terminal.

```
chris@ubuntu: ~  
[Tue May 16 13:37:38] chris@ubuntu:~$
```



# Operaciones con variables

## • Aritméticas

expr

expr 4 + 5  
expr 8 - 5  
expr 10 \ / 3 (división)  
expr 20 % 3 (resto)  
expr 10 \ \* 3  
(multiplicación)

bc

Para números reales

var = \$(( 4 + 5 ))

También para -, \*, /

“expr” y “bc” son comandos de la *shell* (lo veremos más adelante)

## • Lógicas

&&

→ AND (y)

||

→ OR (o)

!

→ NOT (no)

## • Cadenas de texto

\${str:posición}

→ Extrae de str una subcadena desde posición

\${str:posición:longitud}

→ Extrae de str desde posición una subcadena con longitud

\${str/buscar/reemplazar}

→ Busca la primera ocurrencia de buscar y lo reemplaza

\${str//buscar/reemplazar}

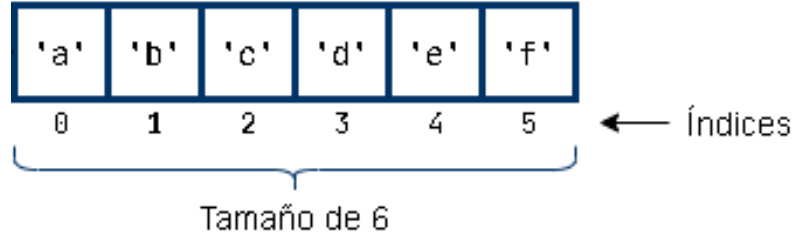
→ Busca todas las ocurrencias de buscar y lo reemplaza

concatenacion="\${cadena1}\${cadena2}Texto nuevo"

Primera  
posición de  
la cadena = 0

# Otro tipo de variables: arrays

Tanto números como caracteres o cadenas de texto



En realidad, las otras variables son un array de un único elemento

- Definición de arrays
  - declare -a Frutas → Declara un array
  - Frutas[1]="Pera" → Declara un array y asigna valor en posición 1
  - Frutas=('Manzana' 'Pera' 'Naranja') → Declara un array y lo inicializa con valores
- Utilización de los arrays
  - echo \${Frutas[0]} → Manzana
  - echo \${Frutas[1]} → Pera
  - Frutas[0]="Fresa" → echo \${Frutas[0]} → Fresa
- Eliminación
  - unset Frutas
  - unset Frutas[1] → Elimina un elemento del array

echo \${Frutas[@]} → Todo el array como array  
echo \${Frutas[\*]} → Todo el array como texto  
echo \${#Frutas[@]} → Número de elementos

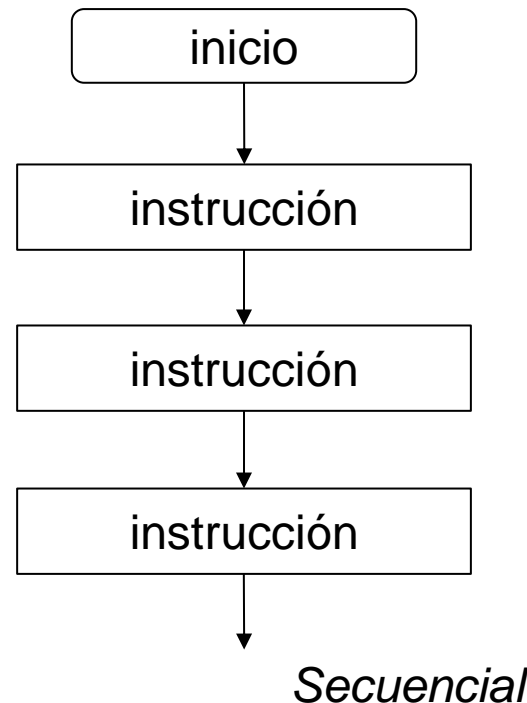
# Estructuras de control

Habitualmente, las instrucciones se ejecutan secuencialmente.

¿Qué sucede si queremos variar el flujo del programa?

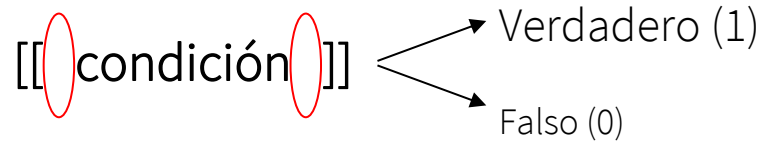
Por ejemplo: dependiendo de la entrada por teclado (se ha pulsado el signo “+” o el signo “-”), realizo un conjunto de acciones (sumar dos números) u otro (restar dos números) → Necesario estructuras de control:

- **De selección:** si se produce una condición, realizo un conjunto de acciones; si no se produce la condición, no las realizo.
- **De iteración:** repito un conjunto de acciones hasta que se cumpla una condición.



# Estructuras de control: condiciones y operadores relacionales

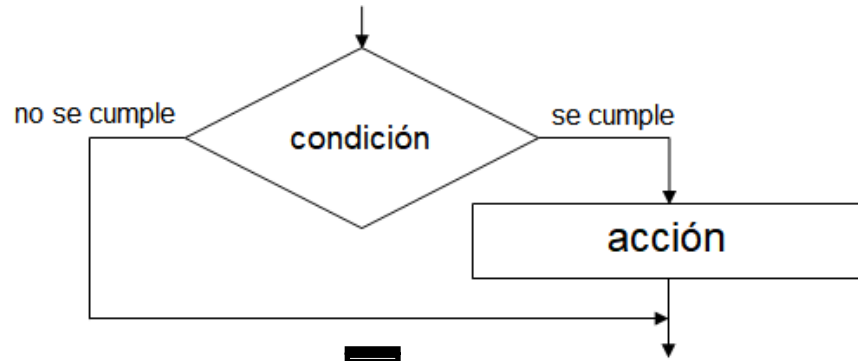
Para evaluar una condición:



Ficheros	Operación (Verdadero si...)
-f fichero	... es un fichero normal
-r fichero	... puede leerse
-w fichero	... puede escribirse
-x fichero	... puede ejecutarse
-d fichero	... es un directorio
-z fichero	... tiene tamaño > 0

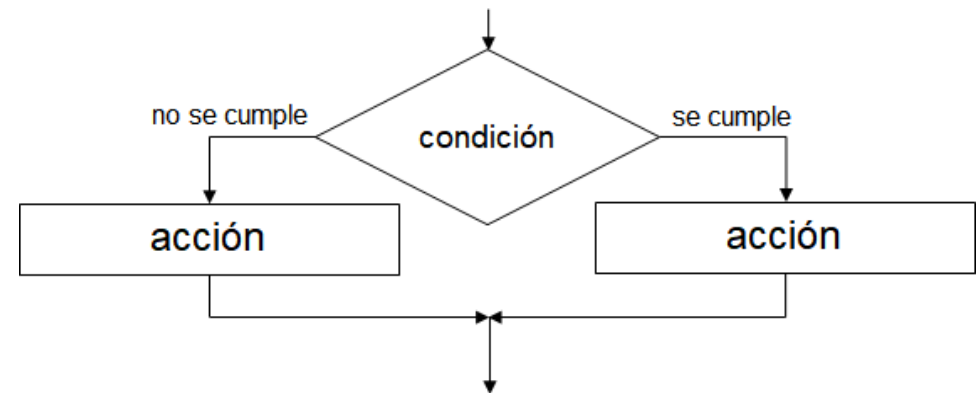
Números	Cadenas de texto	Operación
-eq	= / ==	Igual a
-ne	!=	No igual
-lt	\<	Menor que
-le		Menor o igual que
-gt	\>	Mayor que
-ge		Mayor o igual que
	-z	Está vacío
	-n	No está vacío

# Estructuras de control: selección



if [[ condición ]]  
then  
    comand  
    o/s  
fi

```
if [[ condición ]]  
then  
    comando/s  
elif [[ condición ]]  
then  
    comando/s  
fi
```



if [[ condición ]]  
then  
    comando/s  
else  
    comando/s  
fi

<https://victorroblesweb.es/2016/09/03/condicionales-en-shell-script/>  
<https://atareao.es/tutorial/scripts-en-bash/condicionales-en-bash/>

## Estructuras de control: selección (II)


```
#!/bin/bash

#Pide por pantalla los operandos
read -p "Introduce el primer número: " num1
read -p "Introduce el segundo número: " num2

#Pide por pantalla la operación
read -p "Introduce el operador (1 para suma, 2 para resta): " op

if [[ ${op} -eq 1 ]]
then
    resultado=$(( ${num1} + ${num2} ))
else
    resultado=$(( ${num1} - ${num2} ))
fi

echo ${resultado}
```



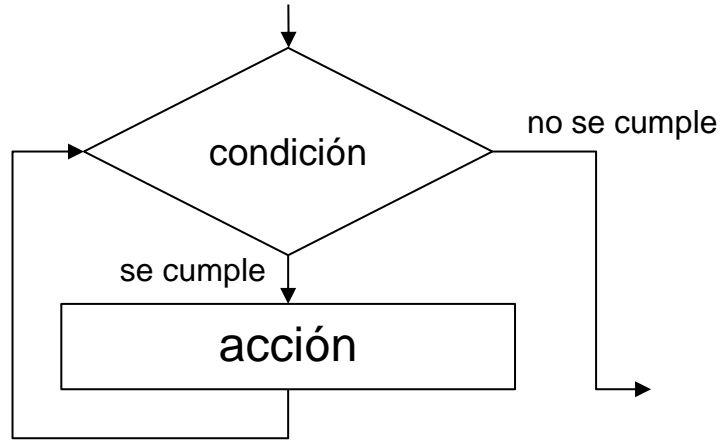
```
A11PC24 $ ./calculadora.sh
Introduce el primer número: 5
Introduce el segundo número: 2
Introduce el operador (1 para suma, 2 para resta): 1
7
```

### Otra estructura de selección

```
case $variable in
    expr1)
        comando/s
        ;;
    expr2)
        comando/s
        ;;
    ...
    *)
        comando/s
        ;;
esac
```

# Estructuras de control: iteración

También se conocen como “bucles”.



```
while [[ condición ]]  
do  
    comando/s  
done
```

```
until [[ false ]]  
do  
    comando/s  
done
```

```
for $variable in valores  
do  
    comando/s  
done
```

↓ Otra forma  
para el bucle

```
LIMIT=10  
for  
for (( x=1; x <= LIMIT ; x++ ))  
do  
    comando/s  
done
```

*break [n]* # no realiza “n” iteraciones del bucle

*continue [n]* # continua con la siguiente iteración del bucle

## Estructuras de control: iteración (II)

```
A11PC24 $ ./frutas.sh
Indica tus frutas favoritas. Pulsa 0 para terminar:
manzana
sandía
fresa
0
Tus frutas favoritas son:
manzana
sandía
fresa
```

```
#!/bin/bash

declare -a frutas

echo "Indica tus frutas favoritas. Pulsa 0 para terminar: "

read dato
contador=0

while [[ ${dato} != "0" ]]
do
    frutas[${contador}]=${dato}
    contador=$(( ${contador} + 1 ))
    read dato
done

echo "Tus frutas favoritas son: "
for fruta in ${frutas[@]}
do
    echo ${fruta}
done
```



# Ejecución de un script. Argumentos de entrada

```
#!/bin/bash

US0="US0: calculadora2.sh operando1 operación operando2"

if [[ $# -ne 3 ]]
then
    echo ${US0}
else
    resultado=$(( $1 "$2" $3 ))
    echo ${resultado}
fi
```



```
A11PC24 $ ./calculadora2.sh
US0: calculadora2.sh operando1 operación operando2
A11PC24 $
A11PC24 $ ./calculadora2.sh 5 + 2
7
```

## Variables especiales argumentos entrada

- \$0 → Nombre del *shell script*
- \$1, \$2... → Parámetros de entrada, según posición
- \$# → Número de parámetros de entrada
- @\$ → Variable que recoge el valor de todos los parámetros de entrada, pero como un array (matriz)
- \$\* → Variable que recoge el valor de todos los parámetros de entrada, pero como una cadena

## ... otras variables especiales

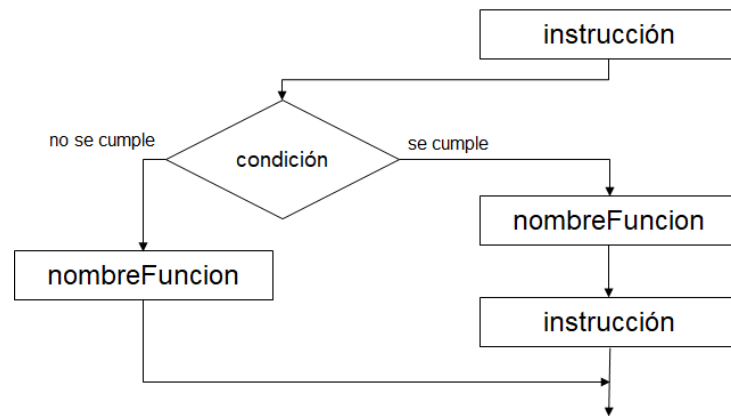
- \$! → PID del último proceso ejecutado
- \$? → Valor devuelto por el último comando ejecutado
- \$\$ → PDI del *shell script*

# Funciones

- Funciones → secuencia de instrucciones que se llaman mediante un identificador (nombre de la función) desde cualquier parte del código.
  - Pueden recibir argumentos de entrada → igual que hemos visto en los argumentos de entrada durante la ejecución de un *script* → nombreFuncion argumento1 argumento2 ... argumentoN
  - Tienen acceso a las variables del *script* y también pueden tener variables locales a la función.
  - Pueden devolver un resultado (o no)

```
function nombreFuncion(){  
    comando/s  
    var1=...  
    return valor #opcional  
}
```

<https://linuxize.com/post/bash-functions/>



# Excepciones

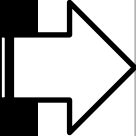
- También llamadas **Trap**
  - Eventos generados durante la ejecución del script y que pueden ser tratados
  - Muy útil en la señalización entre procesos de Linux → e.g. SIGINT

```
#!/bin/bash
mi_catch_2_3() {
    echo "Capturada la señal 2 o SIGINT SIGQUIT"
    echo "Terminando el proceso"
    kill -15 $$
}

trap "mi_catch_2_3" 2 3

while true; do
    true
done
```

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	17) SIGCHLD
18) SIGCONT	19) SIGSTOP	20) SIGTSTP	21) SIGTTIN
22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO
30) SIGPWR	31) SIGSYS	....	



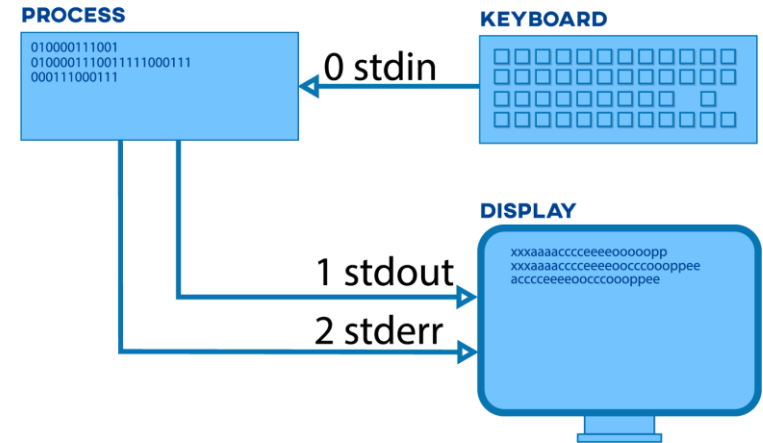
```
Al1PC24 $ ./excepcion.sh
^CCapturada la señal 2 o SIGINT SIGQUIT
Terminando el proceso
Terminado
```

Envío de señales a procesos → kill [señal] PID

# Redireccionamiento de E/S y ficheros

- Recuerda que STDIN, STDOUT y STDERR son tratados como archivos en Linux


Operador	Resultado
<	Redireccionar de STDIN
>	Redireccionar de STDOUT
>>	Redireccionar de STDOUT (si el fichero existe, añade y no lo borra)
2> / 2>>	Redirección del STDERR
1>&2	Redirección STDOUT a STDERR
2>&1	Redirección STDERR a STDOUT
>&	Redirección STDOUT Y STDERR

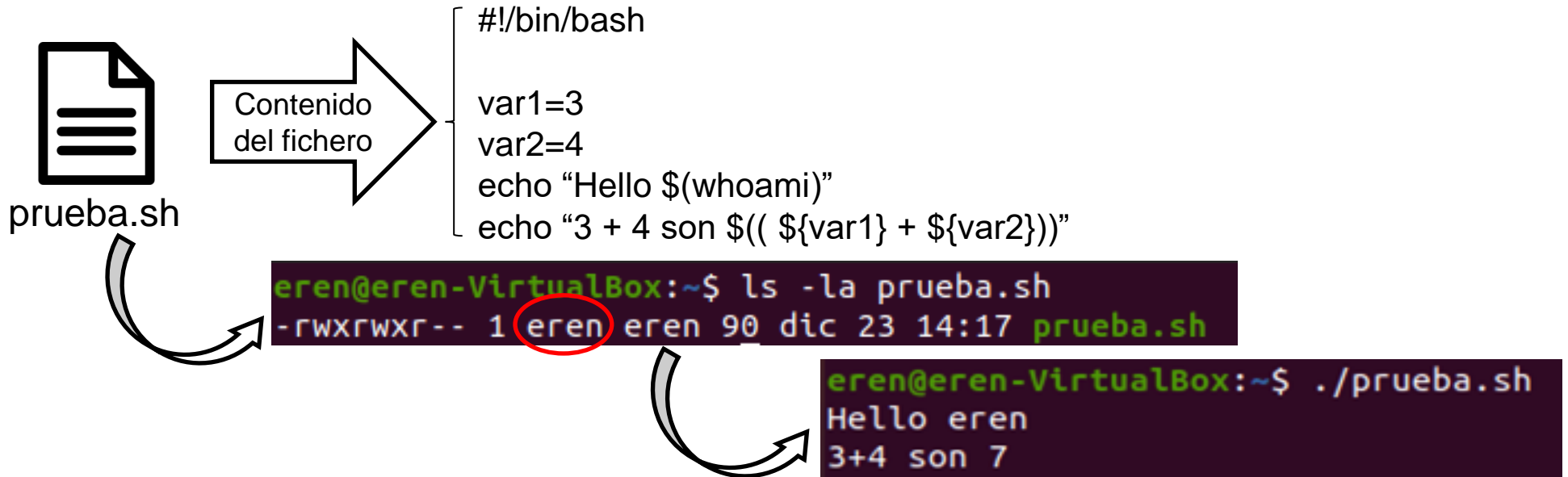


comando > archivo.txt #El resultado a archivo nuevo  
comando >> archivo.txt #El resultado añadido a archivo

cat myfile >/dev/null # elimina STDOUT  
rm nofile 2>/dev/null # elimina STDERR

# Comandos de Linux más utilizados en los *scripts*

- Pueden utilizarse comandos de la *shell* en mi *script* → 
- Para obtener el resultado del comando → \$(comando)



# Comandos de Linux más utilizados en los *scripts* (II)

## Comandos para mostrar ficheros

- `cat` → muestra un archivo por pantalla, todo seguido
- `more` → muestra archivo, pero parando cada vez que llene el espacio ofrecido por la pantalla
- `head` → Muestra las primeras líneas de un fichero (por defecto, 10).
  - `-n`: indica el número de líneas.
- `tail` → Muestra las últimas líneas de un fichero (por defecto, 10).
  - `-n`: indica el número de líneas.

# Comandos de Linux más utilizados en los *scripts* (III)

## Comandos de búsqueda

- **grep** → Muestra las líneas de un fichero dado que coinciden con un cierto patrón. Se utiliza para buscar texto mediante CLI
  - -r: busca recursivamente en los ficheros de un directorio
  - -n: muestra el número de línea en la que se ha encontrado el patrón
  - -i: ignora diferencia entre mayúscula/minúscula
  - -w: el patrón debe aparecer como palabra completa
  - -v: muestra las líneas que no contengan el patrón
  - -c: cuenta el número de líneas donde aparece el patrón
  - -o: no muestra la línea entera, muestra únicamente el patrón buscado
- **find** → Busca ficheros en el árbol de directorios, mostrando el nombre de los archivos encontrados que se correspondan con cierto conjunto de criterios.
  - -maxdepth, -name, -type, etc.

# Comandos de Linux más utilizados en los *scripts* (IV)

## Patrón (expresiones regulares):

- `.` → cualquier carácter, sólo uno (comodín)
- `?` → el carácter precedente es opcional
- `*` → item precedente que se repite 0 o más veces
- `+` → item precedente que se repite 1 o más veces
- `()` y `[]` → agrupan caracteres
  - `[]` → permiten cualquier carácter de los indicados dentro de los corchetes
  - `()` → permite el conjunto completo de caracteres indicados dentro de los paréntesis
- `{n,m}` → lo precedente se repite un mínimo de n veces y un máximo de m
- `^` → inicio de cadena o línea (e.g. `^el` → el curso)
- `$` → final de cadena o línea (e.g. `so$` → el curso)
- `\` → carácter de escape (recordatorio)

`grep -E 'ExpReg' archivo`  
`find ... -regex 'ExpReg'`

<https://sio2sio2.github.io/doc-linux/02.conbas/10.texto/01.regex.html>



# Comandos de Linux más utilizados en los *scripts* (V)

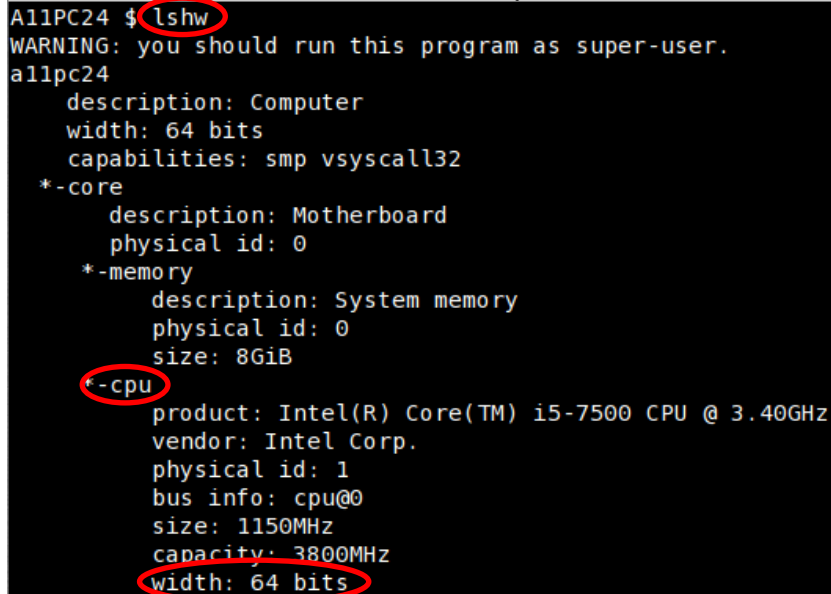
## Comandos para trabajar con los datos

- `wc` → (*word count*): muestra el número de líneas (-l), palabras (-w), caracteres (-m) o bytes (-b) de los ficheros que se le indiquen
- `sort` → muestra en orden ascendente el contenido de los ficheros que se les pasa como argumentos
  - -r: ordena en sentido inverso
  - -u: elimina líneas duplicadas
  - -k: ordena por columna
- `cut` → corta partes y secciones de cada línea
  - -d: delimitador (e.g. \t, "-", ".", etc.)
  - -f: (fields), campos seleccionados
- `tr` → (*translate*) Permite el reemplazo o eliminación de caracteres.
  - -s: (squeeze) Elimina la repetición continua de caracteres.
  - -c: (complement) Realiza la acción sobre todos los demás caracteres distintos a los indicados.
- `uniq` → agrupa líneas repetidas y las muestra como una sola.
  - -c: (count) Indica el número de veces que se ha repetido una línea.

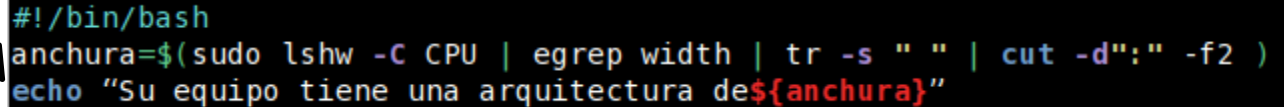
# Comandos de Linux más utilizados en los *scripts* (VI)

Crear un *script* que te diga si el procesador es de 32 o 64 bits

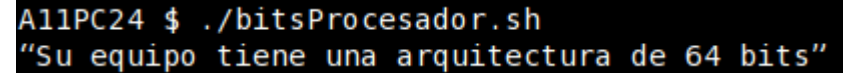
- ¿Qué comando me puede dar esa información?



```
A11PC24 $ lshw
WARNING: you should run this program as super-user.
allpc24
  description: Computer
  width: 64 bits
  capabilities: smp vsyscall32
*-core
  description: Motherboard
  physical id: 0
*-memory
  description: System memory
  physical id: 0
  size: 8GiB
*-cpu
  product: Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
  vendor: Intel Corp.
  physical id: 1
  bus info: cpu@0
  size: 1150MHz
  capacity: 3800MHz
  width: 64 bits
```



```
#!/bin/bash
anchura=$(sudo lshw -C CPU | egrep width | tr -s " " | cut -d":" -f2 )
echo "Su equipo tiene una arquitectura de${anchura}"
```



```
A11PC24 $ ./bitsProcesador.sh
"Su equipo tiene una arquitectura de 64 bits"
```

- Tubería → |
  - Se utiliza en CLI para conectar la salida estándar de un comando con la entrada estándar de otro.
  - E.g.: `grep "nombre" archivo.txt | wc -l`