Una guía rápida sobre programación en bash

GUÍA SOBRE BASH

SOLOCONLINUX

LUIS GUTIÉRREZ LÓPEZ

GUIA SOBRE BASH

Una guía rápida sobre programación en bash

Luis Gutiérrez López

Abril 2024

Resumen

Una guía de referencia rápida para programar en bash. Aprende los conceptos básicos para poder programar script en bash. Un resumen del uso de Parámetros, Variables, Arrays y Funciones. Se incluyen ejemplos reales de scripts como guía de aprendizaje.

Índice

Guía sobre la programacion bash

¿Que es un script bash?

Un script bash, es basicamente un conjunto de instrucciones que permiten ejecutar una serie de comandos en la Shell (terminal) de una forma secuencial.

Estos scripts, se pueden perfeccionar, hasta convertirse en un pequeño programa con el que realizar casi cualquier tarea administiva en GNU/Linux, ya sea mediante bucles, funciones, etc...

Ficheros bash

Para que Linux identifique el fichero a ejecutar como un script hay que realizar varios pasos:

1. Identificación de la shell a usar.

El fichero debe de tener obligatoriamente en su primera línea lo siguiente:

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

> TRUCO: Si queremos que un script bash se ejecute en modo 'depuración' incluiremos un -x al final de la linea anterior quedando así: `#!/bin/bash -x`

2. Permisos de ejecución.

El fichero a ejecutar debe tener permisos de ejecución, una vez creado el script bash hay que darle los permisos necesarios:

```
chmod 755 nombre_del_script
```

Tambien puedes usar:

```
chmod +x nombre_del_script
```

3. Nombre y extension.

El fichero a crear, puede tener cualquier extensión o incluso no tenerla, ya que Linux sabe que debe realizar al leer la primera línea del fichero (*paso 1*).

Aunque lo recomendable es identificarlo mediante la extension `.sh`

4. Todo lo que vaya detras de `#` se considera *Comentario*

```
# Esta linea es un comentario
echo "Bienvenido" # Sacar por pantalla 'Bienvenido' (Comando + Comentario)
```

5. Los scripts Bash y sus comandos se ejecutan de forma secuencial de la primera a la última linea, excepto las funciones como veremos más adelante.

Mi primer script bash

> Para la edición de los ficheros de script Bash, usaremos un editor en Linux, ya sea un editor gráfico (pluma, atom, vscode, etc.) o un editor de consola (vi, vim, nano, joe, etc.).

En mi caso suelo usar vi/vim, por ser uno de los que siempre están instalados por defecto en todos los equipos con GNU/Linux

Puedes revisar el artículo monográfico sobre VIM para aprender a usar Vim.

Vamos a crear nuestro primer script `hola.sh`

```
vim hola.sh
```

Dentro del editor escribiremos dentro las siguientes líneas:

```
#!/bin/bash
echo "Hola. Soy tu primer script Bash"
```

NOTA 1: `echo` es un comando Linux que nos muestra por pantalla un texto o el contenido/valor de una o varias variables.

NOTA 2: `printf` es un comando Linux que imprime texto o variables, pero permitiendo aplicar un formato a la salida.

Salimos guardando, para ello en vim pulsamos `ESC` y luego escribimos `:wq`

Ahora le damos permisos de ejecucion:

```
chmod 755 hola.sh
```

Ya está listo, vamos a probarlo.

Desde la consola/terminal/Shell, y desde la misma carpeta en donde se encuentra el script, escribe lo siguiente:

```
./hola.sh
```

Mostrará por pantalla lo siguiente:

```
Hola. Soy tu primer script Bash
```

Parámetros

Los párametros en bash

Un script Bash puede recibir parámetros y estos pueden ser procesados dentro de él.

Los parámetros que se pueden usar dentro de un script bash son:

Parámetro	Significado del parámetro
`$#`	Nº de parametros recibidos
`$0`	Nombre y ruta del propio script
`$1 ... $9`	Parámetros del 1 al 9 recibidos
`${N}`	Parámetro de la posicion N recibido
`$*`	Todos los parámetros recibidos (excepto `$0`) [1]
`$@`	Array de parámetros recibidos (excepto `$0`) [2]
`$$`	El PID (numero de proceso) del script
`$?`	El código de error del ultimo comando ejecutado

Comando `shift` para desplazar parámetros

`shift` Comando que desplaza los parametros recibidos (excepto `$0`) a una posición anterior.

Ejemplo: Hemos recibido dos parámetros, `$1` con el valor `UNO` y el segundo parámetro `$2` con el valor `DOS`.

Si ejecutamos `shift` el 2º parametro se desplaza y ocupa la posicion del 1º, el valor de este desaparece.

Tras la ejecución si revisamos el contenido de la variable `$1` obtendríamos `DOS`.

El comando shift se utiliza para procesar los parámetros y realizar distintas acciones cuando se ejecuta un script.

Un ejemplo de uso de un script con parámetros es:

```
./find-truncate.sh --directory /var/tmp --max-size 300M --days 15
```

Los parámetros son `--directory`, `--max-size` y `--days` a continuación de cada parámetro se indica el valor que tendrá.

Para explicarlo con detalle, vamos a crear nuestro propio script usando nuestros propios parámetros.

Procesaremos los parámetros recibidos que podremos recibir en cualquier orden y haremos uso de `shift`, para que independientemente del orden de los parámetros, los pueda procesar.

[1] Un elemento con todos los parametros en una cadena.
[2] Cada parametro es un elemento diferente, esta variable es la que se debe usar para procesar los parámetros recibidos.

Script usando `shift`

Script: `nombre-apellido.sh`

```
#!/bin/bash
# USO: ./nombre-apellido.sh --nombre NOMBRE --apellido APELLIDO
while [[ $# > 0 ]]  # Recorrer todos los parametros recibidos
do
   case "$1" in
        -n | --nombre )
            shift  # Encontre "-n" o "--nombre" Con shift salto al siguiente parametro que es el valor
            nombre="$1"  # guardo en la variable nombre el valor.
            shift  # Salto al siguiente parametro
            ;;
        -a | --apellido )
            shift
            apellido="$1"
            shift
            ;;
        * )
            # Algo no es lo que esperaba..
            shift # Ignoro y salto al siguiente parametro
            ;;
    esac
done

# Tengo parámetros almacenados en las variables correctas. Las puedo usar.
echo "Tu Nombre es: $nombre y tu Apellido es: $apellido"
```

Vamos ahora a ejecutarlo, vemos que funciona correctamente:

```
./nombre-apellido.sh --nombre Luis --apellido Gutierrez
    Tu Nombre es: Luis y tu Apellido es: Gutierrez
```

Si lo ejecutamos cambiando el orden de los parametros, también funciona:

```
./nombre-apellido.sh --apellido Gutierrez --nombre Luis
    Tu Nombre es: Luis y tu Apellido es: Gutierrez
```

Variables

En un script Bash se pueden usar variables para almacenar valores o el resultado de la ejecución de comandos.

La variables no son `tipadas` y en ellas se puede almacenar cualquier cosa:

- Texto
- Números
- Arrays

  Nota: No hay Booleanos en bash.

Variables Globales/Locales

Las variables pueden ser *globales* a nivel de todo el script o *locales* cuando están dentro de una función.

> Se distingue entre mayúsculas y minúsculas en la declaración de variables.

Declaración de variables

Para declarar una variable, simplemente hay que escribir el nombre de la variable, seguida de un igual y el valor que se le quiera asignar.

Veamos algunos ejemplos sobre como declarar diferentes variables.

Variables numéricas y de texto

```
VAR=n                    # Variable $VAR almacena un número
VAR=texto                # Variable $VAR con un texto
VAR='cadena de texto'    # Variable $VAR almacena texto (no interpretable)
VAR="cadena de texto"    # Variable $VAR almacena texto (interpretable)
```

> Observa que se pueden usar comillas simples o dobles para almacenar una cadena de texto, aunque su comportamiento varía, como veremos más adelante.

En un script podriamos usarlo así:

```
nombre=Luis
CALLE="Calle Larga"
Despacho=401
```

En el ejemplo anterior se crean las variables `$NOMBRE`, `$CALLE` y `$Despacho`

Si una variable va a contener espacios en blanco en la cadena de texto, deberemos usar comillas dobles " o simples ' para almacenarlo o sólo podremos almacenar la primera palabra recibida.

Variables para almacenar resultado de una ejecución

Podemos almacenar una variable el resultado de la ejecución de un comando.

Podemos realizarlo usando cualquiera de los siguientes formatos:

```
VAR=$(comando)
VAR=`comando`
```

Ejemplo para almacenar el resultado de comandos.

Se crea la variable `$Usuario` y se almacena el resultado de ejecutar el comando `whoami`

```
# Opcion 1 con $ (dolar)
Usuario=$(whoami)

# Opción 2 con comilla inclinada)
Usuario=`whoami`
```

Se almacena en `$Texto` el contenido del archivo `fichero.txt`

```
Texto=`cat fichero.txt` # Opcion 1
read Texto < cat fichero.txt # Opcion 2
```

Se almacena en `$Salida` el resultado de `stdout` y `stderr`

```
Salida=$(comando 2>&1)
```

Almacenamos los valores de otra variable `$OTRA`, en la segunda concatenamos Variables y Texto.

```
VarA=$OTRA
VarB="$V1 texto1 $V2 texto2 ..."
```

Almacenamos en la variable `$OPCION` el *primer parámetro* recibido

```
OPCION=$1
```

Se crea la variable `$SORTEMARAP` y se almacenan concatenados y en orden inverso los *3 primeros parámetros* recibidos

```
SORTEMARAP="$3 $2 $1"
```

En el paso 1 se crea `$Cadena`. En el paso 2 se le asigna a ella misma su valor anterior y además se incluye texto.

```
Cadena="En un lugar" #paso 1
Cadena="$Cadena de la Mancha" #paso 2
```

El resultado que se obtiene al final en `$Cadena` es: `En un lugar de la Mancha`

Interpretar o No Interpretar variables

Cuando se utilizan de comillas simples en una variable, todo el contenido incluido entre ellas se considera como *no-interpretable* y todo se trata como texto:

```
VAR='  NO procesar $V1'
echo $VAR
  NO procesar $V1
```

Si utilizamos comillas dobles, en el caso de existir alguna variable entre ellas, el valor de la variable encontrada es interpretado y sustituido por el valor que contenga:

```
VarA="En un lugar"
VarB='de la Mancha'
VarC="de cuyo nombre no quiero"
VarD=acordarme
TEXTO="$VarA $VarB $VarC $VarD"

echo $TEXTO
En un lugar de la Mancha de cuyo nombre no quiero acordame
```

Se pueden concatenar cadenas de texto tanto con comillas simples como dobles o sin ellas.

No es lo mismo usar ” que ’.

Las comillas simples impiden que se procese dentro de ellas el valor de las variables.

```
NONO='$3 $2 $1'
```

En este caso se crea la variable y se guarda `'$3 $2 $1'`. NO se guarda el valor de los parámetros

Arrays

Bash permite usar array, se pueden definir de varias formas:

- `declare ARRAY[n]`
- `typeset ARRAY[n]`
- `ARRAY=(a b c ... x y z)`

Creación de Arrays vacios

Podemos usar `declare` y `typeset`:

```
declare Marca[9] (opcion 1)
typeset Marca[9] (opcion 2)
```

En ambos casos se crea la variable `$Marca` que es un array vacio de 9 elementos.

Creación de Array sin definir

Podemos usar `declare`:

```
declare -a Colores
```

En este caso se crear una variable `$Colores` que es un array sin tamaño definido.

Arrays con elementor predefinidos

Simplemente usamos el formato `ARRAY=(a b c ...)`:

```
Frutas=(Pera Manzana Platano)
```

Se crea la variable `$Frutas` que es un array con las frutas indicadas.

```
ARRAY[n]=valor
Fijar valor de un elemento del array
Marca[0]="Tranqui-Cola"
Se almacena en la variable $Marca, que es un array, en su primer elemento (0) el valor indicado.
```

Fijar valor de un elemento del Array

Debemos indicar el nombre del Array y entre *corchetes* la posición del elemento, seguido de un igual y despues el valor a asignar.

Al igual que en todos los lenguajes la numeración de los elementos de un array va de 0 a N-1.

```
COCHE[0]="Seat"
COCHE[1]="Opel"
```

Usando las Variables

Ya hemos visto que para poder usar o leer una variable, hay que escribir siempre delante del nombre de la variable, el simbolo `$`

```
echo "$Usuario - $Despacho"
```

Usando los Arrays

Para los Arrays existe una sintaxis un poco especial para poder usarlos:

- `${ARRAY[n]}` Acceder al elemento `n` del array
- `${ARRAY[@]}` Devuelve todos los elementos del array (array completo)
- `${#ARRAY[@]}` Devuelve el tamaño del array (numero de elementos)
- `${!ARRAY[@]}` Índices del array

Veamoslo con algún ejemplo.

Devolver un elemento determinado:

```
Frutas=(Pera Manzana Platano) # Elementos 0, 1 y 2
echo ${Frutas[2]}
Platano
```

Si queremos ver todos los elementos del array:

```
Frutas=(Pera Manzana Platano)
echo ${Futas[@]}
Pera Manzana Platano
```

Si queremos saber cuantos elementos tiene el array:

```
Frutas=(Pera Manzana Platano) # Elementos 0, 1 y 2 => 3 elementos
echo ${#Frutas[@]}
3
```

Para ver los índices y elementos reales del array, debemos usar `${!ARRAY[@]}`

```
Frutas=(Pera Manzana Platano)
echo ${!ARRAY[@]}
0 1 2
```

Se puede dar la particularidad de incluir elementos en un array y dejar "huecos" y tener posiciones sin usar:

```
Frutas=(Pera Manzana Platano)
echo "Nº Elementos: ${#Frutas[@]}"
echo ${!Frutas[@]}
3
echo "Indices: ${!ARRAY[@]}"
0 1 2

# Incluimos nueva fruta (en 6ª posición)
Fruta[5]="melocoton"
echo "Nº Elementos: ${#Frutas[@]}"
echo ${!Frutas[@]}
4
echo "Indices: ${!ARRAY[@]}"
0 1 2 5
```

Mediante los índices podríamos recorrer un array al que le falten elementos intermedios y evitaríamos errores de elementos no existentes.

Concatenar Arrays

Podemos unir varios arrays, simplemente debemos usar los paréntesis para definir la variable que los almacena como array e incluir una coma `,` como elemento de concatenación:

```
Unix=('SCO', 'HP-UX', 'IRIX', 'XENIX')
Linux=('Debian', 'Suse', 'RedHat')
NIX=("${Unix[@]}", "${Linux[@]}")
echo ${NIX[@]}
SCO, HP-UX, IRIX, XENIX, Debian, Suse, RedHat
```

Estructuras de Control

IF

IF	IF - ELSE	IF - ELIF

`if [ condicion ]`
`then`
`comandos`
`fi`
	`if [ condicion ]`
	`then`
	`comandos`
	`else`
	`comandos`
	`fi`
		`if [ condicion ]`
		`then`
		`comandos`
		`elif [ condicion ]`
		`comandos`
		`else`
		`comandos`
		`fi`

FOR

FOR	FOR (Estilo C)

`for variable [in lista]`
`do`
`comandos $variable`
`done`
	`for ((i=0; i<5; i++))`
	`do`
	`comandos $variable`
	`done`

WHILE y UNTIL

WHILE	UNTIL

`while condicion`
`do`
`comandos`
`done`
	`until condicion`
	`do`
	`comandos`
	`done`

CASE

```
case variable in
    valor1)
        comandos_opcion-1
        ;;
    valor2 | valor3 | valor4)
        comandos_opcion-2-3-4
        ;;
    valorN)
        comandos_opcion-N
        ;;
    *)
        comandos_opcion-por-defecto
        ;;
esac
```

SELECT

```
select NumOpcion [in lista]
do
    comandos $NumOpcion
done
```

`$NumOpcion` es el Nº de opcion escogida de la lista

SALIR DE BUCLES

- `break` Sale del bucle
- `break N` Sale de N bucles anidados
- `continue` No finaliza los procesos/comandos que faltan del bucle y comienza la siguiente iteración

Operadores y Operaciones

Operadores Lógicos

- AND/Y: `&&` ó `-a`
- OR/O: `||` ó `-o`
- NOT/NEGACION: `!` Si queremos realizar un comparación con `if` debemos escribirlo de la siguiente forma:

`if [ condicion ]` Hay que *respetar los espacios* entre los corchetes.

Si queremos *combinar condiciones* los operadores lógicos van fuera de los corchetes:

```
if [ $a > $b ] || [ $j < $k ]
```

Operadores Aritméticos

- `+` Sumar
- `-` Restar
- `*` Multiplicar
- `/` Dividir
- `%` Resto de division (módulo)
- `**` Elevar
- `++` Incrementar
- `--` Decrementar

Para realizar operaciones aritméticas se debe usar dos sintaxis diferentes:

- Opcion 1: `$((operacion))`
- Opción 2: `$[operacion]`

Ejemplos de operaciones:

```
a) echo $((2+5)) => 7
b) echo $[21/7] => 3
c) i=1; echo $[++i] => i=2
d) echo $[3**2] => 9
```

Operadores de Cadenas

Se pueden realizar operaciones con las cadenas de texto directamente:

- `${#cadena}` Obtener longitud
- `${cadena:N}` Extraer subcadena a partir de la posición N
- `${cadena:N:M}` Extraer M caracteres a partir de la posición N
- `${cadena#texto}` Elimina texto si coincide al principio de la cadena
- `${cadena%texto}` Elimina texto si coincide al final de la cadena
- `${cadena/texto1/texto2}` Reemplaza en la cadena la 1ª coincidencia de texto1 por texto2
- `${cadena//texto1/texto2}` Reemplaza en la cadena todas las coincidencia de texto1 por texto2
- `${cadena/#texto1/texto2}` Reemplaza en la cadena texto1 por texto2 si aparece al principio
- `${cadena/%texto1/texto2}` Reemplaza en la cadena texto1 por texto2 si aparece al final
- `array=(${cadena/delimitador/ })` Transformar una cadena en un array segun el delimitador indicado

Comparadores

Comparadores Alfanuméricos/Texto

- `==` Igual
- `!=` Distinto
- `>` Mayor que
- `<` Menor que
- `-z` Es nulo
- `-n` Longitud no es cero (mayor que 0)

Comparadores Numéricos

- `-eq` Igual que (=)
- `-gt` Mayor que (>)
- `-ge` Mayor o Igual que (>=)
- `-lt` Menor que (<)
- `-le` Menor o Igual que (<=)
- `-ne` No igual/Distinto (!=)

Comparadores para ficheros (archivos y directorios)

- `-e` El Fichero/Directorio existe
- `-s` El Fichero existe y no está vacio
- `-d` El Directorio existe (y no es un fichero)
- `-f` El Fichero existe (y no es un directorio)
- `-r` Fichero/Directorio tiene permiso de Lectura
- `-w` Fichero/Directorio tiene permiso de Escritura
- `-x` Fichero tiene permiso de Ejecución, y si es Directorio se puede acceder a él
- `-O` Eres propietario del Fichero/Directorio
- `-G` El Fichero/Directorio es de tu grupo
- `-nt` El fichero es más reciente que el otro (F1 -nt F2)
- `-ot` El fichero es más antiguo que el otro (F1 -ot F2)

Entrada/Salida y Operadores de Redirección

Dispositivos de Entrada y Salida

- `stdin` Es el dispositivo de Entrada (teclado).
- `stdout` Es el dispositivo de Salida estandard (pantalla).
- `stderr` Es el dispositivo de Salida de Errores (generalmente la pantalla).

Otros dispositivos especiales (/dev)

- `/dev/null` Dispositivo nulo, en el que todo lo que entra en él se descarta.
- `/dev/random` Dispositivo de generación de números pseudo-aleatorios.
- `/dev/urandom` Dispositivo de generación de números aleatorios de alta calidad
- `/dev/zero` Dispositivo que genera únicamente caracteres *nulos* (0x00).
- `/dev/full` Dispositivo que está siempre lleno. Usado para test de *disco lleno*. Al leer de él se comporta como `/dev/zero`

Redirectores

- `>` Enviar salida (stout y stderr) hacia ...
- `<` Recibir contenido de ...

En Linux la entrada/salida está asociada a los siguientes *descriptores de ficheros*:

- *stdin* asociado a **0**
- *stdout* asociado a **1**
- *stderr* asociado a **2**

Todos estos descriptores se pueden usar para capturar los datos que pasan por ellos y reenviarlos a variables o ficheros.

Ejemplo: Listar el contenido de `/tmp/Carpeta1` y almacenarlo en el fichero `/tmp/lista_ficheros.txt`

```
ls -1 /tmp/Carpeta1 > /tmp/lista_ficheros.txt
```

Si tenemos acceso a la carpeta indicada, se almacenará en el fichero `lista_ficheros.txt` la lista de todos los ficheros existentes de esa carpeta.

Separación de la salida

Si por algún motivo, no tenemos acceso a la ruta, o se produce *cualquier error*, lo que obtendremos será un fichero en el que veremos mezclados la lista de ficheros que queriamos obtener, junto los mensajes de error que se produzcan.

Podemos separar las diferentes salidas (`stdout` y `stderr`), usando sus descriptores.

Vemos algunos ejemplos:

1. Enviar el resultado (stdout/1) del comando a un fichero y dejar que los mensajes de error salgan por pantalla:

   ```
   ls -1 /tmp/Carpeta1 1> /tmp/lista_ficheros.txt
   ```

2. Enviar el resultado (stdout/1) a un fichero y almacenar los errores (stderr/2) en otro fichero para revisar los logs:

   ```
   ls -1 /tmp/Carpeta1 1> /tmp/lista_ficheros.txt 2>/tmp/errores.txt
   ```

3. Enviar el resultado (stdout/1) a un fichero e ignorar los errores (stderr/2), evitando que salgan incluso por pantalla, redirigiendolos al dispositivo `/dev/null`:

   ```
   ls -1 /tmp/Carpeta1 1> /tmp/lista_ficheros.txt 2>/dev/null
   ```

4. Evitar que salgan ningun resultado, ni siquiera errores por pantalla:

   ```
   ls -l /tmp/Carpeta1 >/dev/null 2>&1
   ```

   Explicación. Primero se redirige el resultado a `/dev/null` y la salida 2 (stderr) se reenvia a la 1 (stdout) a la que ya habiamos enviado previamente su salida a /dev/null con lo que *no saldrá nada por pantalla.*

Tabla resumen de Redirecciones

Redirección	Acción que se produce
`> Fich`	Redirigir las dos salidas stdout y stderr hacia fich (fichero/dispositivo), se sobreescribe el fichero Fich cada vez. Se pierde el contenido anterior.
`>> Fich`	Redirigir las dos salidas stdout y stderr hacia fich (fichero/dispositivo), se añade la salida del comando al final del fichero Fich y si no existe se crea.
`1> Fich`	Redirigir salida estandard stdout hacia fich (fichero/dispositivo).
`2> Fich`	Redirigir salida de errores stderr hacia fich (fichero/dispositivo).
`1> Fich1 2> Fich2`	Redirige salida estandar stdout hacia Fich1, la salida de errores (stderr) hacia Fich2, Fich1 y Fich2 pueden ser ficheros ó /dispositivos.
`> Fich 2>&1`	Redirigir salida stdout hacia fich y la de errores stderr a donde apunta stdout.
`2>&1`	Redirigir salida de errores stderr a stdout.
`1>&2`	Redirigir salida stdout a la salida de errores stderr.
`< Fich`	Lee el contenido del fichero Fich y lo envia al comando que le precede.

Tuberias o Pipes

`|` Reenvia la salida al comando que le sigue. Se le conoce como 'pipe' o tubería.

Un ejemplo sencillo:

Tenemos una carpeta con varios ficheros y queremos su lista en orden alfabético ascendente y descendente:

```
$ ls -l
  total 24
  -rw-r--r-- 1 luisgulo luisgulo 1 ene 21 00:30 aaa
  -rw-r--r-- 1 luisgulo luisgulo 2 ene 21 00:30 bbb
  -rw-r--r-- 1 luisgulo luisgulo 3 ene 21 00:30 ccc
  -rw-r--r-- 1 luisgulo luisgulo 4 ene 21 00:30 ddd
  -rw-r--r-- 1 luisgulo luisgulo 5 ene 21 00:30 eee
  -rw-r--r-- 1 luisgulo luisgulo 6 ene 21 00:30 fff
```

El comando `sort` nos permite ordenar de modo normal o en modo inverso/*reverso* `sort -r` un fichero o cualquier dato que reciba.

Podemos *concatenar* (pasar) la salida de un comando a otro mediante *pipe* `|`

```
ls -1 | sort
    aaa
    bbb
    ccc
    ddd
    eee
    fff

ls -1 | sort -r
    fff
    eee
    ddd
    ccc
    bbb
    aaa
```

Se pueden concatenar las salidas de un comando a otro mediante tuberias (pipes).

Ejemplo de 'tuberias' consecutivas:

```
RESOLUCION=$(xrandr | grep current | awk -F "," '{print $2}' | awk '{print $2"x"$4}')

echo $RESOLUCION
2646x1024
```

Funciones

Las funciones nos permite agrupar varias acciones o grupo de comandos repetitivos dentro de un script, para usarlas cuando las necesitemos.

> Las funciones no se ejecutan al procesar el script, hay que llamarlas explícitamente para que se ejecuten.

Definir Funciones

Las funciones se puede definir de varias formas:

- Con la palabra reservada `function`:

```
function NombreFuncion {
  comandos
  ...
}
```

- Mediante `()` omitiendo `function`:

```
NombreFuncion () {
  comandos
  ...
}
```

Se omite la palabra reservada *function*, pero hay que identificar la función obligatoriamente mediante `()` para que se interprete como función.

En ambos formatos, los comandos a ejecutar en la función van entre llaves de inicio y cierre: `{ }`

Ámbito de las variables en las Funciones

- **Ámbito Global**: Una variable definida en la parte principal del script (fuera de cualquier función) es accesible (se puede leer y escribir) desde dentro de cualquier función.

Se define normalmente como `VARIABLE="valor"`

- **Ámbito Local**: Una variable definida con el atributo local es unicamente accesible desde dentro de esa función.

Se define dentro de la función con la palabra reservada `local`: `local VARIABLE="valor"`

Vamos a comprobar como se comportantan las variables según su ámbito.

Abre tu editor de texto (vi/vim, nano, joe) y copia y pega el siguiente contenido del script. El fichero lo vamos a grabar con el nombre: `que_pasa.sh`

```
#!/bin/bash
function AlgoPasa {
    local UNO="Texto uno"
    DOS="Texto dos"
    echo "DENTRO DE LA FUNCION -> Variable UNO = $UNO  y Variable DOS = $DOS"
}

# --- Parte Principal del programa ---
UNO="1" # UNO es una variable Global
DOS="2" # DOS es tambien variable Global

# Mostramos las variables globales UNO y DOS recien inicializadas.
echo "ANTES DE LA FUNCION -> Variable UNO = $UNO  y Variable DOS = $DOS"

# Llamamos a la función
AlgoPasa

# Mostramos de nuevo las variables globales UNO y DOS ...
echo "DESPUES DE LA FUNCION -> Variable UNO = $UNO  y Variable DOS = $DOS"
```

Damos permisos de ejecución al script: `chmod +x que_pasa.sh`

Y lo ejecutamos para comprobar que hace con las variables:

```
./que_pasa.sh

ANTES DE LA FUNCION -> Variable UNO = 1 y Variable DOS = 2
DENTRO DE LA FUNCION -> Variable UNO = Texto uno y Variable DOS = Texto dos
DESPUES DE LA FUNCION -> Variable UNO = 1 y Variable DOS = Texto dos
```

Salida y Devolución de valores

Tras la ejecución de cualquier programa o aplicación de Linux, se genera un código de salida.

Este código sirver para poder comprobar si se ha realizado correctamente o se ha producido algún error.

La estandarización de estos códigos de salida es la siguiente:

- **0**: El programa ha terminado correctamente y sin error.
- **n**: Donde n es cualquier valor distinto de cero (0). Indica que se ha producido algún error.

El desarrollador de la aplicación debe proporcionar la lista de códigos de salida ó códigos de error, para poder analizar lo que ha sucedido.

En un script de bash, debemos realizar lo mismo y finalizar siempre devolviendo un código de salida adecuado a su ejecución.

Para ello la última línea de código que se ejecute debe emitir ese código.

Esto se realiza mediante el comando: `exit N`

Ejemplo:

```
#!/bin/bash
RUTA="$1" # Recibimos como parámetro una ruta
ls -l $RUTA 2>/dev/null
CodError=$? # Capturamos en variable $CodError codigo salida del comando anterior
if [ $CodError -eq 0 ] ; then
    echo "Todo correcto"
else
    echo "Atencion: Se produjo algun error"
fi
exit $CodError
```

> NOTA: Recuerda que `$?` devuelve el codigo de salida/error de la última ejecución. Y lo podemos usar, consultar o escribir en la consola mediante: `echo $?`

Códigos de error en Funciones

Las funciones, tambien pueden devolver un código de salida (un número).

En lugar de utilizar *exit* las funciones tienen que usar la instrucción `return N` para notificar el código de salida/error que produzcan.

NOTA: `return` únicamente permite devolver un número como valor.

Si queremos que una función nos devuelva otro valor que no sea un número, por ejemplo una cadena de texto, un array o cualquier otra cosa, deberemo usar una varible global en la que almacenaremos el valor deseado.

Ejemplo de control de errores: `une-2palabras.sh`

```
#!/bin/bash
function Une {
    local palabra1=$1 # Primer parametro recibido por la funcion
    local palabra2=$2 # Segundo parametro recibido por la funcion
    Cadena="$palabra1 $palabra2" # Cadena es una variable Global
}

# Recibo 2 palabras desde la linea de comandos
P1="$1"
P2="$2"

if [ -z $P1 ] || [ -z $P2 ]
then
    codError=1
    echo "Escriba 2 palabras por favor"
else
    # LLamo a la funcion Une y le paso como parametros P1 y P2
    Une $P1 $P2
    echo "$Cadena"
    codError=0
fi
exit $codError
```

Ejemplos de scripts en BASH

A continuación se dejan una serie de scripts reales en bash, para su estudio.

Script para Leer un fichero línea a línea

leer_fichero.sh

```
#!/bin/bash
# Leer fichero linea a linea
FICHERO="$1"
if [ -f "$FICHERO" ]
    #|| [ -n "$FICHERO" ]
then
  # Leemos el fichero
  while read LINEA
  do
     # hacemos con la LINEA leida lo que deseemos
     echo "$LINEA"
  done < $FICHERO
  exit 0
else
    echo "Debe indicar un fichero y que contenga algo"
    exit 1
fi
```

Script para Grabar la Pantalla del escritorio en un fichero `.avi`

grabar_pantalla.sh

```
#!/bin/bash
# Este script graba la Pantalla 1
function Inicia {
    FICH=$(tempfile --suffix=.avi)
    zenity --info --text "** GRABACIÓN DE VÍDEO DE LA PANTALLA PRINCIPAL **\n\nSe va a grabar video del
    NUMMONITORES=$(xrandr |grep '*'|wc -l)
    # Solo se graba primer monitor
    RESOLUCION=$(xrandr |grep current|awk -F "," '{print $2}'|awk -v NP=$NUMMONITORES '{print $2/NP"x"$
    ffmpeg -y -f x11grab -s $RESOLUCION -r 25 -i :0.0 $FICH -loglevel quiet &
}

function Finaliza {
    killall -9 ffmpeg
    FICH=$(ls -1tr /tmp/*avi|tail -1)
    zenity --info --text "** GRABACIÓN DE VÍDEO DE LA PANTALLA PRINCIPAL **\n\n¡GRABACIÓN DETENIDA!\nSu
}

function Ejecuta {
    # Comprobamos si esta en ejecución
    EJECUTANDOSE=$(ps aux |grep -i ffmpeg|grep -v grep|wc -l|awk '{print $1}')
    if [ "$EJECUTANDOSE" = "1" ] ; then
       Finaliza
    else
       Inicia
    fi
}
# Lanzar el programa
Ejecuta
```

Script para facilitar `sudo` a los usuarios

sudo_seguro.sh

Activa 'sudo' en modo seguro para tu usuario (pide contraseña para cada ejecución)

```
#!/bin/bash
# SUDO-SEGURO (Se pide la clave del usuario siempre)
# Este script crea un fichero para poder usar sudo al usuario
echo "Debe proporcionar la clave de root cuando se le solicite"
echo "$USER ALL=(ALL:ALL) ALL" > /tmp/autorizado_$USER; su -c "cp /tmp/autorizado* /etc/sudoers.d/."
```

sudo_comodo.sh

Activa 'sudo' en modo cómodo para tu usuario (No pide contraseña para la ejecución)

```
#!/bin/bash
# SUDO-SEGURO (Se pide la clave del usuario siempre)
# Este script crea un fichero para poder usar sudo al usuario
echo "Debe proporcionar la clave de root cuando se le solicite"
echo "$USER ALL=(ALL:ALL) ALL" > /tmp/autorizado_$USER; su -c "cp /tmp/autorizado* /etc/sudoers.d/."
```

Script de ejemplo de uso de 'tput', 'printf' y arrays de texto

nieve.sh

```
#!/bin/bash

# Usamos tput (permite poner colores y/o mover el cursor por la pantalla)
LINEAS=$(tput lines)
COLUMNAS=$(tput cols)

# Lo declaramos como Array
declare -A CopoDeNieve
declare -A UltimosCopos

#Limpiamos antes de comenzar la nevada...
clear

# Funcion que mueve el copo por la pantalla (cayendo)
function mover_copo() {
    i="$1"
    if [ "${CopoDeNieve[$i]}" = "" ] || [ "${CopoDeNieve[$i]}" = "$LINEAS" ]; then
        CopoDeNieve[$i]=0
    else
        if [ "${UltimosCopos[$i]}" != "" ]; then
            printf "\033[%s;%sH \033[1;1H " ${UltimosCopos[$i]} $i
        fi
    fi
    printf "\033[%s;%sH*\033[1;1H" ${CopoDeNieve[$i]} $i
    UltimosCopos[$i]=${CopoDeNieve[$i]}
    CopoDeNieve[$i]=$((${CopoDeNieve[$i]}+1))
}

# Aqui empieza el codigo principal para hacer nevar.
while :
do
    # la variable del sistema $RANDOM devuelve un valor aleatorio :-D
    i=$(($RANDOM % $COLUMNAS))
    mover_copo $i
    for x in "${!UltimosCopos[@]}"
    do
        mover_copo "$x"
    done
    sleep 0.1
done
```