Sistemas Informáticos
1º DAW/DAM

Unidad 8

Programación de tareas/scripts en Linux

Autor: Arturo Bernal Mayordomo
Curso 2020/2021

Índice de contenido

1 ¿Qué es un script?

Un script no es más que fichero de texto que contiene una serie de instrucciones y órdenes que se ejecutarán en la línea de comandos de forma seguida. Estas órdenes funcionarían de la misma manera si las fuésemos introduciendo en la línea de comandos nosotros.

Para ejecutar un script tenemos 2 alternativas. La primera es llamar al intérprete pasándole el fichero de texto que contiene el script como primer parámetro:

bash script.sh
bash -x script.sh → Inicia el script en modo depuración de bash (recomendado para pruebas).

Esto sólo requiere que el archivo tenga permiso de lectura. La segunda es indicando en el archivo la ruta del intérprete de comandos en la primera línea de la siguiente forma: **#!/bin/bash** (en este caso /bin/bash es la ruta al intérprete bash, que será el que utilicemos, pero aquí se puede poner la ruta a nuestro intérprete favorito). En este último caso basta con darle permiso de ejecución y llamar directamente al script → **./script.sh**. También podemos añadir el directorio donde tenemos el script a la variable PATH. Por ejemplo, si el directorio es **/home/alumno/bin**, añadimos esto al archivo $HOME/.bashrc

export PATH=”$PATH:/home/alumno/bin”

De esta manera, simplemente escribiremos el nombre del script en la consola desde cualquier directorio y se ejecutará como otro comando del sistema.

Un script simple puede tener una serie de comandos seguidos y ya está:

```
#!/bin/bash
mkdir dir1
cd dir1
touch arch1
cd ..
ln -s dir1/arch1 enl1
```

Si queremos averiguar si un comando existe en nuestro sistema para no crear otro con el mismo nombre, por ejemplo, podemos saberlo con la orden:

type comando

Además de informarnos si existe, nos dirá si es un comando integrado en bash, un archivo ejecutable, un alias (tema 7) o una función.

2 Comentarios

Un comentario en un script es una línea que será ignorada por el intérprete de comandos. En estas líneas podemos poner cualquier nota o apunte que nos resulte útil, por ejemplo, para recordar en el futuro que es lo que realizaba una parte del script que hicimos hace tiempo.

Un comentario siempre empieza por #, lo que haya detrás será completamente ignorado hasta que pasemos a la siguiente línea (#!/bin/bash es una instrucción especial, no un comentario):

```
#!/bin/bash

#Este es un comentario cualquiera
#Creamos el directorio “dir1” y dentro creamos el archivo “arch1”
mkdir dir1
cd dir1
touch arch1
cd ..

#Creamos un enlace a dir1/arch1
ln -s dir1/arch1 enl1
```

3 Variables

Una variable se puede decir que es un nombre que se le da a un dato cualquiera para luego poder acceder a él sin problemas. Por ejemplo, si queremos preguntarle a alguien por su nombre en un script, habrá que almacenar ese nombre en una variable, ya que no podemos saber el nombre que introducirá. De esa forma podremos acceder a su nombre sin conocerlo, simplemente conociendo el nombre de la variable es suficiente:

```
#!/bin/bash

echo “Escribe tu nombre”
#Almacenamos el nombre que escriba el usuario en una variable llamada nombre
read nombre
#Ahora no sabemos su nombre, pero cuando lo escriba lo tendremos guardado en nombre
#correcto
echo “Hola $nombre”
#incorrecto
echo “Hola nombre”

#También podemos dar el valor a una variable de esta forma
var=”Hola”
echo “$var $nombre”
```

Para acceder al valor de una variable hay que ponerle el símbolo $ delante y para darle un nuevo valor no hace falta. Si queremos acceder a una variable, pero tiene texto pegado, podemos encerrar el nombre entre llaves '{}' detrás del símbolo $.

```
#!/bin/bash

var=”arch”
#Queremos crear un archivo llamado archnombre
touch ${var}nombre
#Así no busca una variable que se llame 'varnombre', sino 'var'
```

Si queremos que un carácter especial como $ se interprete como una carácter normal y no como un identificador de variables, hay que “escaparlo”, es decir, poner el símbolo '\' delante:

```
#!/bin/bash

var=20
echo “Esto me ha costado $var \$.”
#Esto mostraría la frase: Esto me ha costado 20 $.
```

4 Interacción con el usuario

4.1 echo

Como hemos visto en los ejemplos anteriores, la orden echo sirve para mostrar una cadena de texto por pantalla. Para incorporar una o más variables, basta con integrarlas en el texto entre comillas dobles precedida del símbolo $.

- **-n** → No hace un salto de línea al final del comando
- **-e** → Interpreta caracteres de escape como \n (salto de línea) o \t (tabulador), por ejemplo.

4.2 printf

La orden printf funciona como su homóloga en muchos lenguajes de programación (C, C++, Java, PHP, awk,…). Ofrece bastantes más posibilidades para el formato de texto que echo. Funciona poniendo como primer parámetro la cadena que mostraremos incorporando elementos especiales que se sustituirán por los parámetros añadidos a continuación en orden.

- **printf “Te llamas %s y tu edad es %d\n” “$nombre” “$edad”**
- No hace salto de línea automático como echo.
- **%s** → Se sustituye por un string.
  - **%20s** → Si el string tiene menos de 20 caracteres, te rellena por la izquierda con espacios. Si tiene más te lo muestra todo.
  - **%.20s** →Como el anterior pero limita como máximo a 20 caracteres.
  - **%20.12s** → Te muestra 12 caracteres rellenando 8 espacios a la izquierda (20 – 12)
  - **%-20s** y **%-20.12s** → Como los anteriores pero rellena por la derecha con espacios.
- **%d** → Se sustituye por un entero. **%.3d** → Si el número tiene menos de 3 cifras, rellena con 0 a la izquierda.
- **%f** → Se sustituye por un número decimal. **%.3f** →Muestra el número con un formato fijo de 3 decimales.
- http://es.wikipedia.org/wiki/Printf
- **Opción -v var** → En lugar de imprimir, guarda la cadena en una variable.

printf “m%s\n” a e i o u → Si las variables o parámetros sobrepasan a lo que hay que sustituir en la cadena, se autorepite (como si tuviera un bucle integrado)

ma me mi mo mu.

4.3 read

La orden **read** se puede usar para darle valor a una variable como ya se ha mostrado en los ejemplos. También se puede dar valores a más de una variable a la vez:

read var
hola que tal
var contendrá el texto “hola que tal”

read var1 var2 var3
hola que tal bien
var1 contendrá “hola”, *var2* contendrá “que” y *var3* contendrá “tal bien”
A cada variable se le asigna una palabra, y a a la última, el resto del texto.

Otra funcionalidad del comando **read** es la posibilidad de mostrar texto al usuario justo antes de la entrada de texto (lo que nos ahorra poner un echo o printf antes).

read -p “Dime tu nombre: ” nombre

Si no se le indica al final del comando una variable donde guardar lo escrito por el usuario, se guarda en la variable especial **$REPLY**.

- **-n num** → Lee un número de caracteres como máximo.
- **-s** → Oculta lo que el usuario escribe.
- **-r** → **recomendado** → Deshabilita los caracteres de escape \n\t etc.…
- **-t seg** → Segundos (con decimales) que espera el comando para una entrada de texto.
- **read -r -n 1 -s var** → Lee sólo 1 carácter y lo guarda en var (sin tener que presionar enter).

Utiliza **$IFS** para separar lo que mete en cada variable

- **IFS=”:” read var1 var2** → Usa : como separador, sólo durante la ejecución de read.

4.4 Mostrar varias líneas de texto por pantalla

Para mostrar varias líneas de texto por pantalla, podríamos usar varios **echo** o **printf**, O jugar con **\n** y **\t** dentro de una cadena. También podemos utilizar una característica del comando cat, para que nos muestre lo que queramos como si estuviera en un fichero:

cat <<END
Escribo un texto de varias líneas
Hasta que escriba END como última línea
Esto se comporta como si estuviera en un fichero de entrada, y lo muestra por pantalla.
END

5 Parámetros

Los parámetros son opciones o valores que se le pasan al programa cuando se ejecuta por línea de comandos, la estructura es la siguiente:

nombre_progama param1 param2 param3 param4 ...

Por ejemplo, al crear un directorio con mkdir podemos ver las equivalencias de la siguiente manera:

mkdir	*-p*	*dir1/dir1-1*	*dir2*	*dir3*
nombre_prog	param1	param2	param3	param4

Para acceder a los valores de los parámetros dentro del programa, se accede con las variables especiales $1, $2, $3, $4, ... Estarán numeradas según el orden del parámetro. Para acceder a los parámetros a partir del 9, deberán encerrarse los números entre llaves '{}' (${10}, ${11}, ...).

```
#!/bin/bash

echo “Me llamo $1 y tengo $2 años”
```

./script.sh Juanito 20
Me llamo Juanito y tengo 20 años

También tenemos las variables especiales:
$0 : Nombre del programa
$# : Número de parámetros enviados
$@ : Lista con todos los parámetros enviados

6 Comillas

Advertencia: Al copiar y pegar trozos de código de este documento que contengan comillas, puede que estas no se copien de forma correcta (caso de las comillas dobles casi siempre) y el programa no funcione bien. Por ello se recomienda, una vez pegado el trozo de código, volver a reescribir todas las comillas que aparezcan.

En la programación en el shell Bash existen 3 tipos de comillas:

'Comillas simples' : Las comillas simples encierran una cadena de texto que se representa después tal como se ha escrito, es decir, no se sustituyen ni variables ni caracteres especiales que estén dentro de este tipo de comillas.

“Comillas dobles “ (A la derecha de la tecla 0) : Las comillas dobles tienen una función similar a las comillas simples. La diferencia es que primero sustituyen las variables y los caracteres especiales por su valor y a continuación el resultado de esa sustitución es como si estuviera encerrado en comillas simples.

`Comillas invertidas` (A la derecha de la tecla p) : Las comillas invertidas se utilizan para encerrar un comando y después son sustituidas por la salida que produce la ejecución de ese comando. Se puede utilizar también **$(comando)** como equivalente a **`comando`**.

```
#!/bin/bash

var=”Mensaje”
echo 'os mando un $var'
#Esto escribirá: os mando un $var

echo “os mando un $var”
#Esto escribirá: os mando un Mensaje

echo `ls -l`
#Mostrará la salida de ejecutar ls -l

#También se pueden poner cadenas de texto seguidas encerradas en diferentes tipos
#de comillas
echo 'La variable $var tiene el valor:' “$var” `date`
echo 'La variable $var tiene el valor:' “$var” $(date)
#Esto mostrará: La variable $var tiene el valor: Mensaje lun feb 26 12:23:36 CET 2007
```

Cuando pongamos una cadena de texto sin encerrarla en comillas de ningún tipo se comportará principalmente como si la hubiéramos encerrado entre comillas dobles, pero cualquier cantidad de espacios seguidos, tabuladores o saltos de línea los interpretará como un sólo espacio.

```
#!/bin/bash

echo “Aquí van     5 espacios”
#Mostrará: Aquí van     5 espacios
echo Aquí van     5 espacios
# Mostrará: Aquí van 5 espacios
```

7 Condiciones

Como veremos más adelante, lo que hemos aprendido hasta ahora, nos deja muy limitados. ¿Cómo podríamos hacer un script que preguntase un número y dependiendo del número introducido hiciese una cosa u otra?, ¿Cómo se podría hacer una operación 100 veces seguidas sin tener que escribir nosotros el comando 100 veces dentro del script?. Este tipo de cosas tan simples no se pueden hacer simplemente con lo que hemos aprendido hasta ahora.

Antes de aprender a hacer cosas como las mencionadas anteriormente, vamos a aprender que es una condición, y como representarlas en el intérprete de comandos bash.

Una condición es algo que cuando se comprueba sólo puede dar 2 valores como resultado, el valor verdadero (se representa como 0 en bash script) que indica que la condición es válida, y el valor falso (se representa como 1) que indica lo contrario. Una condición puede ser por ejemplo “x es mayor que y”, siendo x e y números.

condición	**resultado**
verdadero	verdadero (0)
falso	falso (1)

Cuando queremos unir 2 o más condiciones para evaluar si son ciertas o no, es decir, podemos crear un programa que te pregunte la altura y el peso, y en función de eso mostrar mensajes diferentes. En este caso podemos evaluar 2 condiciones (altura y peso) de la siguiente forma:

Si altura es igual que x y peso mayor que y
Si altura es igual que x y peso menor que z
Si altura es mayor que f o peso menor que g
etcétera.

En este caso tendremos 2 operadores especiales, llamados operadores lógicos, estos son el operador "y" y el operador "o".

Operador "y": El operador "y" se evalúa como verdadero si todas las condiciones unidas con este operador son verdaderas, y falso en el caso contrario.

Imaginemos la evaluación de: "*condición1* y *condición2* y *condición3*"

condición1	condicion2	condicion3	resultado
Verdadero	Verdadero	Verdadero	Verdadero
Verdadero	Falso	Verdadero	Falso
Falso	Verdadero	Falso	Falso
Falso	Falso	Falso	Falso

En definitiva si cualquier valor evaluado con "y" es falso, el resultado será falso.

Operador "o": El operador "o" se evalúa como falso si todas las condiciones unidas con este operador son falsas, y verdadero en el caso contrario. Es decir, es suficiente con que se cumpla una de las condiciones.

Imaginemos la evaluación de: "*condición1* o *condición2* o *condición3*"

condición1	condicion2	condicion3	resultado
Verdadero	Verdadero	Verdadero	Verdadero
Verdadero	Falso	Verdadero	Verdadero
Falso	Verdadero	Falso	Verdadero
Falso	Falso	Falso	Falso

En definitiva si cualquier valor evaluado con "o" es verdadero, el resultado será verdadero.

También existe otro operador lógico que es la negación, en este caso lo vamos a representar como "!", cualquier condición que tenga delante esta negación da como resultado el valor contrario al de la condición, es decir, se niega el resultado de la condición dando como resultado lo contrario.

condición	**!condición**
verdadero	falso
falso	verdadero

Las condiciones se pueden agrupar entre paréntesis igual que si de expresiones matemáticas se tratara, simplemente hay que evaluar primero las condiciones de los paréntesis y sustituir el paréntesis por el resultado:

condición1 y condición2 y (condición3 o condición4)

condición1=verdadero
condición2=verdadero
condición3=falso
condición4=verdadero

verdadero y verdadero y (falso o verdadero) -> falso o verdadero = verdadero
verdadero y verdadero y verdadero
verdadero

Otro ejemplo:

!(*verdadero y (falso o falso o (verdadero y falso)))*
!(verdadero y (falso o falso o falso))
!(verdadero y falso)
!(falso)
verdadero

8 Valores devueltos por programas

En shell script de Linux, los programas cuando terminan de ejecutarse, devolverán un valor que indicará si se han ejecutado de forma correcta, o por el contrario, ha ocurrido algún tipo de error durante su ejecución. Cuando todo ha ido correctamente, devolverán el valor 0, mientras que si algo ha ido mal, devolverán un valor diferente de 0, normalmente 1 o -1.

Para terminar un programa devolviendo uno de estos valores hay que ejecutar la orden **exit** dentro del programa pasándole como primer y único parámetro el valor de salida. Si no se le pasa ningún parámetro se interpreta como valor 0, es decir, el programa ha ido correctamente.

```
#!/bin/bash

# Escribimos el programa y comprobamos si ha ido todo bien
# Si todo ha ido bien
exit 0
# Si algo ha fallado
exit 1
```

Con esto se puede deducir que el valor verdadero (todo correcto), se representa con el número 0, y otro valor diferente, por ejemplo el 1, se interpretará como falso. Esto puede liar un poco porque funciona justo al revés que en muchos lenguajes de programación ya que normalmente se suele asociar al 0 como valor falso.

9 Condiciones en shell script

Podemos representar un condición de 2 maneras cuando realizamos un script. La primera es ejecutando un comando y evaluando su salida (0 -> correcto, y, otro valor -> falso) y la segunda es mediante la orden **test** que sirve para evaluar expresiones.

9.1 Condiciones al ejecutar comandos

cada vez que ejecutemos un comando o programa y este termine nos devuelve un valor que indica si todo ha ido correcto o no, lo equivalente a verdadero y falso en este caso. Por ejemplo el comando:

cd dir1

Nos devolverá verdadero (0) si el directorio 'dir1' existe y ha podido entrar en el, y falso en caso contrario (algo ha ido mal, es decir, no ha podido entrar en el directorio, bien sea porque no existe, o porque no tiene permiso el usuario).

En este tipo de condiciones podemos utilizar los operadores 'y' y 'o' que vimos anteriormente. El operador y se representa '**&&**' y el operador 'o' se representa '||'. Los comandos se evalúan de izquierda a derecha, es decir, si tenemos algo del tipo: 'comando1 y comando2', el comando 2 sólo se ejecutará si el comando1 es verdadero (ha ido bien), ya que si unimos 2 condiciones con 'y', y alguna es falsa, el resultado siempre será falso, no nos hará falta evaluar la otra condición. Si las unimos con 'o', si el comando de la izquierda es verdadero, no ejecutará el de la derecha.

Imaginemos que queremos crear un archivo dentro de un directorio, pero no sabemos si el directorio existe:

cd dir1 && touch arch1

De esta forma, el comando '*touch arch1*' sólo se ejecutará si el comando '*cd dir1'* se ha ejecutado correctamente.

Otro ejemplo:

cd dir1 || echo "No he podido entrar en dir1"

De esta forma, echo "No he podido entrar en dir1" sólo se ejecutará si '*cd dir1*' ha fallado, es decir, si no podemos entrar en dir1 por alguna razón, mostrará un mensaje de error.

cd dir1 || cd dir2 && touch arch1

Primero evalúa '*cd dir1*', si todo es correcto, no evaluará *'cd dir2*'. Si '*cd dir1*' o '*cd dir2*' funcionan bien, creará un archivo 'arch1' dentro de ellos.

Digamos que va evaluando primero los 2 primeros comandos y el resultado lo evalúa junto con el siguiente de la derecha y así sucesivamente.

cd dir && cd dir3 && touch arch1 || echo “No he podido crear arch1 dentro de dir3”
V *&& cd dir3 && touch arch1 || echo “No he podido crear arch1 dentro de dir3”*
V *&&* ***F*** *&& touch arch1 || echo “No he podido crear arch1 dentro de dir3”*
F *&& / || echo “No he podido crear arch1 dentro de dir3”*
F *|| echo “No he podido crear arch1 dentro de dir3”*
F *||* ***V***
V

Los comandos con / debajo son los que no se han ejecutado. En este caso 'touch arch1' sólo se ejectura si 'cd dir3' se hubiera ejecutado sin problemas.

Si ponemos un símbolo de admiración '!' antes de un comando (separado por un espacio), estaremos haciendo una negación, es decir, si el resultado del comando es V, pasará a ser F y viceversa.

! cd dir || echo “enhorabuena, estamos dentro de dir”
cd dir && echo “enhorabuena, estamos dentro de dir”

Si queremos agrupar comandos (equivalente a usar paréntesis), encerramos el grupo de comandos entre llaves.

cd dir || mkdir dir && cd dir → No queremos que haga el segundo “cd dir” si la primera vez ha podido entrar.
cd dir || {mkdir dir && cd dir} → Forma correcta

9.2 Evaluación de expresiones. La orden 'test' y newtest [[.

En shell script, tenemos una orden o comando llamado test, que evalúa una serie de condiciones y devuelve el valor final, es decir, verdadero, o falso. Hay dos formas de llamar a este comando.

test condición
[condición]

Lo de arriba es un comando normal y corriente ('[' es un alias de test). Y aunque asegura una compatibilidad máxima en nuestros script también es bastante engorroso de utilizar. Por ejemplo, todos los caracteres especiales como los paréntesis para agrupar expresiones se deben escapar, las variables siempre se deben entrecomillar para evitar que al sustituirse si llevan espacios o caracteres especiales pasen cosas extrañas, etc.

Bash integra un operador similar llamado newtest y representado con doble corchete [[. Nos ahorrará muchos dolores de cabeza aunque sólo funcionará en intérpretes Bash. De ahora en adelante, utilizaremos principalmente este nuevo operador por comodidad y potencia, ya que entre otras cosas permite el uso de expresiones regulares con cadenas, y no hace falta entrecomillar las variables.

Cuando evaluamos una expresión con los corchetes '[]' o '[[]]', hay que tener muy en cuenta que se debe dejar un espacio entre el primer corchete y la condición y también al poner el último corchete.

Condiciones para variables:

- [[$variable]] → Comprueba si la variable existe y contiene un valor que no sea vacío.

Condiciones para ficheros: Trabajan con ficheros y algunos de los que existen son:

- *-e fichero* : Comprueba si el fichero existe
- *-f fichero* : Comprueba si el fichero existe y además es un fichero normal y corriente
- *-d fichero* : Comprueba si el fichero existe y además es un directorio
- *-r fichero* : Comprueba si el proceso (script) tiene permiso de lectura sobre el fichero
- *-w fichero* : Comprueba si el proceso (script) tiene permiso de escritura sobre el fichero
- *-x fichero* : Comprueba si el proceso (script) tiene permiso de ejecución sobre el fichero
- *-s fichero* : Comprueba si el fichero no es vacío (tiene más de 0 bytes)

- *fich1 -nt fich2* : Comprueba si fich1 se ha modificado más recientemente que fich2
- *fich1 -ot fich2* : Comprueba si fich1 se ha modificado antes (es más viejo) que fich2

Condiciones para cadenas de texto: Trabajan con texto y algunos de los que existen son:

- *texto1 = texto2 o texto1 == texto2* : Comprueba si las dos cadenas de texto son idénticas. Se pueden usar comodines en texto2 como se utiliza al coincidir nombres de archivo (*,? y []).
  - Por ejemplo **hola = h*a** se cumple.
- *texto1 != texto2* : Comprueba si las dos cadenas de texto son diferentes. Lo contrario que =.
- texto1 =~ texto2: Comprueba si texto1 cumple la **expresión regular** texto2.
  - Por ejemplo, podemos utilizar **$variable =~ ^[0-9]+$** para comprobar si una variable contiene un número.
- *-z texto* : Comprueba si la cadena de texto está vacía (longitud 0)
- *-n texto* : Comprueba que la cadena de texto no esté vacía (longitud > 0)

Condiciones para manejar números: Trabajan comparando valores numéricos:

- *num1 -eq num2* : Comprueba si los números son iguales
- *num1 -ne num2* : Comprueba si los números son distintos
- *num1 -gt num2* : Comprueba si num1 es mayor que num2
- *num1 -ge num2* : Comprueba si num1 es mayor o igual que num2
- *num1 -lt num2* : Comprueba si num1 es menor que num2
- *num1 -le num2* : Comprueba si num1 es menor o igual que num2

Condiciones 'y', 'o', 'no' y paréntesis: Se pueden unir condiciones o negarlas y agruparlas de la siguiente manera:

El operador 'y' se representa con '&&'.
num1 -eq num2 && num1 -gt num3 : Comprueba si num1 es igual a num2 y además mayor que num3.

El operador 'o' se representa con '||'.
-r arch1 || *-x arch1* : Comprueba si arch1 tiene permisos de lectura o de ejecución.

El operador 'no' se representa con '!'.
! -s arch1 : Comprueba que el fichero arch1 no contiene nada

Como ejemplo de ventaja del operador [[(doble corchete) frente a [(corchete), para utilizar los paréntesis había que escaparlos, es decir, poner el símbolo '\' antes de cada paréntesis, ya que estos son símbolos especiales para el intérprete de órdenes y causarían problemas. Además debían ir separados por espacios del resto de cosas. Los operadores 'y' y 'o' se representaban con -a y -o.

[$var -ne 0 -a \(-f arch1 -o -d arch1 \)]
[[$var -ne 0 && (-f arch1 || -d arch1)]]

http://mywiki.wooledge.org/BashFAQ/031

Al comportarse [[como un comando, podemos condicionar la ejecución de un comando o un grupo de comandos a la evaluación de la condición, como vimos en el punto anterior. Es una forma de ahorrarnos la estructura if que veremos en el siguiente punto:

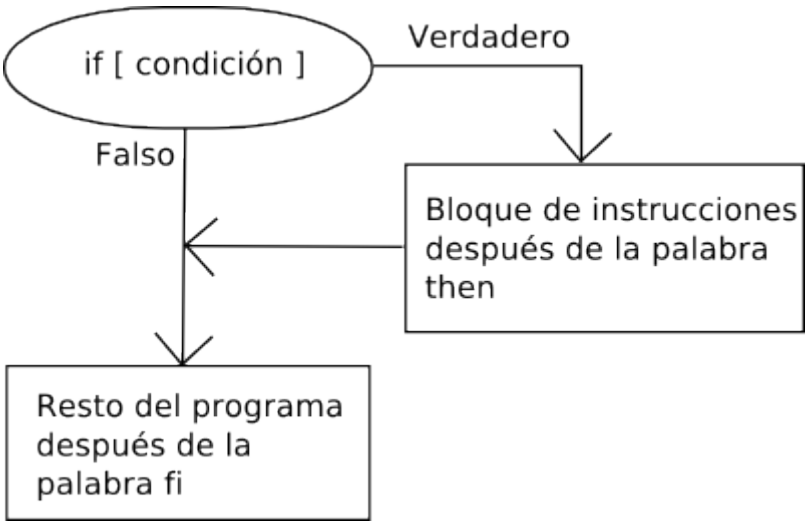
[[$1]] || { echo "Faltan argumentos" >&2; exit 1 } → Si no recibimos un argumento salimos.

10 Control de flujo de programa. Estructura *if*.

La estructura if permite tomar decisiones de una forma sencilla en el programa, normalmente para decidir si en función de una u otra situación ejecutaremos un código u otro. La estructura es la siguiente:

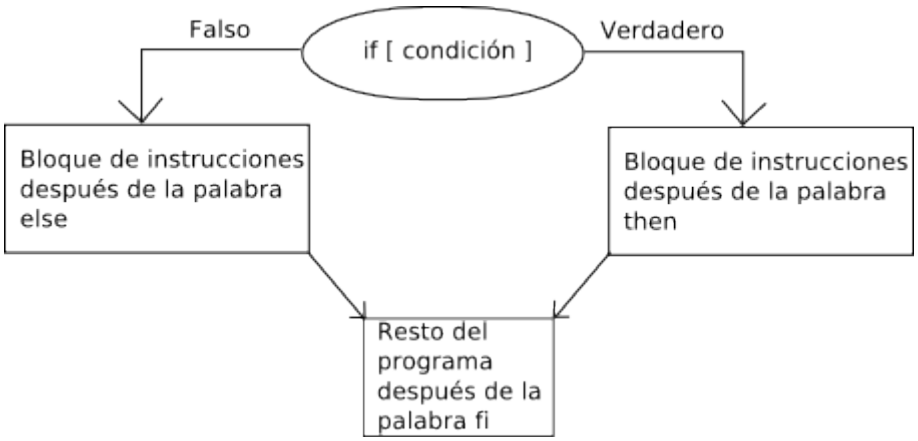
```
if [[ condición ]]
then
   # instrucciones a ejecutar si se cumple la condición
fi
```

De esta forma podemos incluir en el programa un conjunto de instrucciones que sólo se ejecutarán si la condición impuesta se cumple. Si no se cumple, no ejecutará ninguna instrucción que haya inmediatamente después de la palabra 'then'.



Hay una manera de que ejecute una serie de instrucciones cuando no se cumpla la condición. De esta forma podemos ejecutar instrucciones cuando se cumple la condición, y cuando no se cumple ejecutar otras.

if [[condición]]
then
 # instrucciones a ejecutar si se cumple la condición
else
*# instrucciones a ejecutar si **no** se cumple la condición*
fi

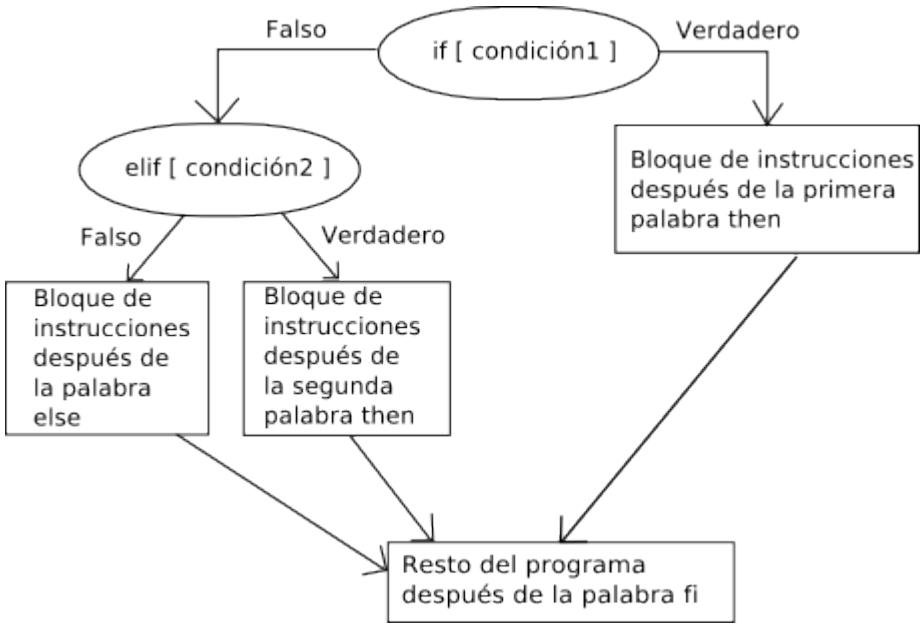


La condición else (lo que se ejecutará en el caso de que la condición no se cumpla no es obligatoria si no se necesita). Después de la palabra 'else' no se debe poner 'then'.

Por último, si queremos que cuando una condición se cumpla ejecutar unas instrucciones, cuando se cumpla otra, ejecutar otras, y así podemos hacer lo siguiente:

if [[condición1]]
then
 # instrucciones a ejecutar si se cumple la condición1
elif [[condición2]]
then
 # instrucciones a ejecutar si se cumple la condición2
else
*# instrucciones a ejecutar si **no** se cumple ninguna condición*
fi

Puede haber tantas instrucciones 'elif' como creamos necesario, pero sólo una instrucción 'else' como máximo (puede no haber ninguna), ya que es lo que se ejecutará cuando no se haya cumplido ninguna de las anteriores condiciones.



Después de cada condición elif también se pone la palabra then en la siguiente línea.

Existe una instrucción nula que se representa con dos puntos ':'. La podemos usar cuando no queremos que se ejecute nada en un punto de la estructura if por ejemplo.

```
if [[ condición ]]
then
   # Si no queremos ejecutar nada si es verdadera, podemos poner la instrucción nula
   :
else
   #instrucciones a ejecutar si no se cumple la condición (falsa)
fi
```

Ejemplos:

- Script que comprueba si recibe como mínimo 2 parámetros. Si recibe menos, muestra un mensaje de error y sale.

```
#!/bin/bash

if [[ $# -lt 2 ]]
then
   echo ”El programa necesita 2 parámetros al menos”
   exit 1
fi
```

- Script igual que el anterior, pero además comprueba que el primer parámetro sea igual a “s”, o a “n”, porque sino también dará error.

```
#!/bin/bash

if [[ $# -lt 2 ]]
then
   echo “El programa necesita 2 parámetros al menos”
   exit 1
else
   if [[ $1 != “s” -a $1 != “n” ]]
   then
      echo “El primer parámetro debe ser \”s \”o \”n\”.”
      exit 1
   fi
fi
```

- Script que pregunte por el nombre de 2 ficheros. Después comprueba si existe el primer fichero, y si existe, escribe el contenido del directorio dentro. Si no existe el primero, comprueba si existe el segundo como alternativa para hacer lo mismo. En caso de que no exista ninguno de los 2 mostrará un mensaje de error y terminará el programa.

```
#!/bin/bash

echo “Teclea nombre para el primer fichero a escribir: ”
read fich1
echo “Teclea nombre de un fichero alternativo, por si no existe el primero: ”
read fich2

if [[ -e $fich1 ]]
then
    ls -l >> $fich1
   echo “He escrito la información en $fich1”
elif [[-e $fich2 ]]
then
    ls -l >> $fich2
  echo “He escrito la información en $fich2”
else
  echo “Ninguno de los 2 ficheros recibidos existe”
  exit 1
fi
```

Recordatorio:

En realidad, los corchetes se comportan como cualquier otro comando, es decir, tienen una salida que es 0 si la condición se cumple, y otro valor si no se cumple. Esto quiere decir que podemos utilizar un simple comando (o varios concatenados) como condición, y podemos redirigir la salida de /dev/null para que no nos muestren nada por pantalla:

```
#!/bin/bash

if ! mkdir hola 2> /dev/null
then
  echo ”No he podido crear el directorio hola”
  exit 1
fi
```

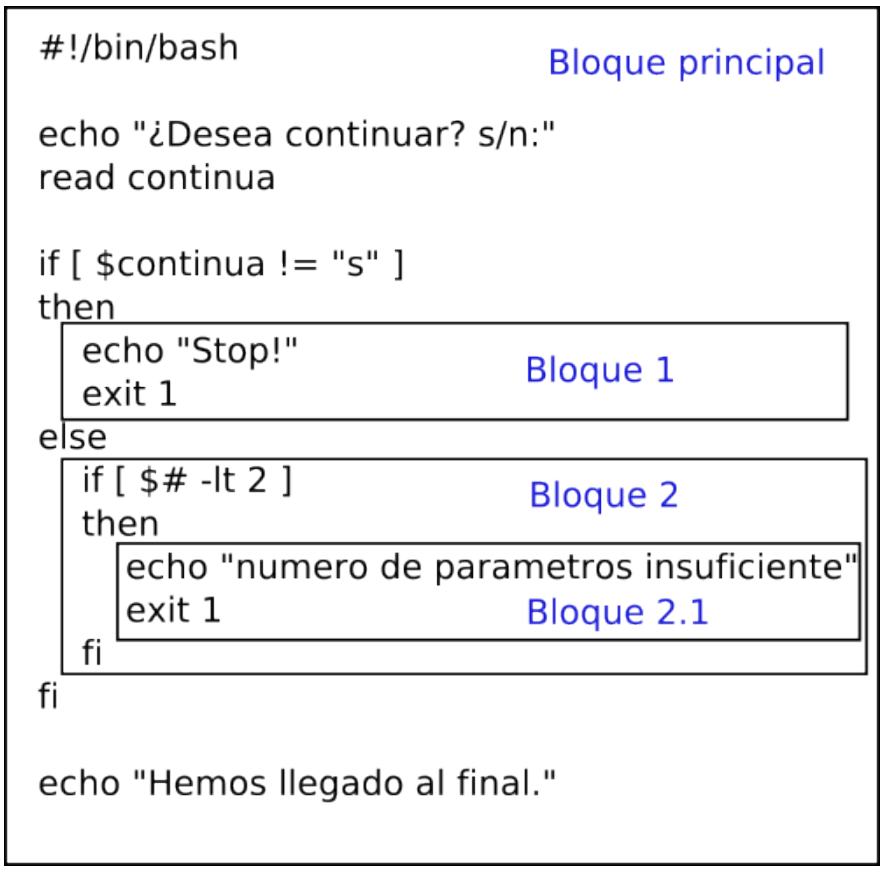
```
#!/bin/bash

if [[ ! -e “hola” ]] && ! mkdir hola 2> /dev/null
then
  echo ”El directorio hola no existe y no he podido crearlo”
  exit 1
fi
```

11 Bloques dentro de un programa

Antes de continuar, y para aclarar del todo el funcionamiento de una estructura como ***if***, y otras que veremos más adelante. Vamos a ver lo que es un bloque de instrucciones dentro de un programa y lo que puede significar.

Un bloque de instrucciones se puede decir que es un conjunto de instrucciones que se ejecutan una detrás de otra, y además, ese bloque está siempre delimitado por una instrucción o palabra clave que marca su inicio y por otra que marca su fin, excepto en el caso del bloque principal de programa. En el caso del bloque principal del programa, el inicio y el final de dicho bloque están marcados por el inicio y el final del fichero.



Como se puede observar en la figura anterior, se pueden meter bloques de instrucciones, unos dentro de otros. A esto se le llama anidar bloques. De esta forma cuando un bloque contiene otro dentro, se puede decir que el que contiene es el bloque *padre*, y el contenido es el bloque *hijo*.

Excepto el bloque principal, el resto de bloques, hemos visto que comienzan con una palabra clave y no terminan hasta que aparece otra para finalizarlo. En el caso de la estructura ***if***, cuando aparece la palabra 'then', se abre un bloque nuevo dentro de este (que se ejecutará sólo cuando se cumpla la condición). Este bloque finalizará cuando aparezca una nueva condición con 'elif', la palabra 'else', o la palabra de fin de estructura 'fi'.

Consejo: Cada vez que se abra un nuevo bloque, es aconsejable que las instrucciones comiencen un mínimo de 2 o 3 espacios después que las instrucciones del bloque padre. De esta forma diferenciaremos sin problema que instrucciones pertenecen a cada bloque.

12 Redireccionamiento

A estas alturas ya hemos visto que podemos construir expresiones del tipo:

echo "una frase" > archivo
ls -l >> archivo

Este tipo de expresiones, lo que hacen es redirigir la salida del programa, es decir, el texto que normalmente mostraría por pantalla, a un archivo que nosotros le especifiquemos.

Ya hemos visto que hay 2 formas de redirigir la salida de un comando o programa. La primera es mediante '>', que sustituye todo lo que hubiera anteriormente en el archivo por el texto de salida del programa (también crea el archivo si no existía antes). La segunda es mediante '>>', que hace exactamente lo mismo, sólo que no borra los contenidos anteriores del archivo de texto, sino que añade el nuevo texto al final del mismo.

Bueno, pues ahora hemos de distinguir 2 tipos de salidas que puede tener un programa, la salida estándar y la salida de error. La salida estándar son los mensajes que el programa envía cuando todo funciona de la forma esperada, es decir, correctamente. La salida de error son los mensajes que el programa envía cuando algo ha fallado. Un script puede generar ambas salidas si algunos comandos funcionan correctamente mientras que otros fallan (si no las redireccionamos, ambas aparecerán siempre por pantalla, juntas).

>, >>	Redireccionan la salida **estandar** a algún archivo que indiquemos
2>, 2>>	Redireccionan la salida de **error** a algún archivo que indiquemos
2>&1	Redirecciona la salida de error hacia la salida estándar

Ejemplo:

```
# El directorio dir ya existe

mkdir dir
# Producirá la salida: mkdir: no se puede crear el directorio «dir»: El fichero ya existe
# Esta salida es de error porque el comando ha fallado.

mkdir dir > salida.txt
# Como estamos redireccionando sólo la salida estandar y el mensaje era de error, nos
# seguirá apareciendo lo mismo de antes.

mkdir dir 2> salida.txt
# En estos casos si que se guardará el mensaje de error porque redireccionamos
# la salida correcta
```

Con esto también podemos llevar la salida normal de un programa a un archivo y la salida de error a otro archivo diferente y así analizar los mensajes por separado por ejemplo:

sh script.sh > salida.txt 2> error.txt

Si queremos que ambas salidas vayan al mismo archivo (o a **/dev/null**), el método recomendado es dirigir la salida estándar al archivo y posteriormente la salida de error redireccionarla hacia la estándar (el método de redirigir ambas salidas con **&>** está obsoleto y no se recomienda).

sh script.sh > salida.txt 2>&1

La salida estándar se numera como 1 (esto quiere decir que aunque podemos utilizar el símbolo '>' para redireccionar la salida estándar, podríamos usar '**1>**'), y la de error como salida 2.

Dentro de un script, podemos hacer por ejemplo, que la salida de un comando echo (o printf), salga por la salida de error, esto se hace de la siguiente forma.

```
#!/bin/bash

echo “elige un número del 1 al 3”
read opcion

if [[ $opcion = “1” ]]
then
     echo “Has elegido el número 1”
elif [[ $opcion = “2” ]]
then
     echo “Has elegido el número 2”
elif [[ $opcion = “3” ]]
then
     echo “Has elegido el número 3”
else
    echo “numero incorrecto” 1>&2
fi
```

De esta forma, el mensaje “numero incorrecto” saldrá por la salida de error, que además es lo que queremos, ya que es un mensaje de error. La expresión '**1>&2**' se podría interpretar como redireccionar el mensaje que normalmente aparecería por la salida estándar (1), a la salida de error (2).

A lo mejor nos interesa que alguna de las salidas (o ambas) no aparezca ni por pantalla, ni se guarde en ningún archivo. Esto se puede conseguir redireccionando la salida de un programa al fichero especial '**/dev/null**'. Este fichero, es como un agujero negro, que se tragará todos los mensajes que le enviemos y desaparecerán. De esta forma:

mkdir dir 2> /dev/null

Estamos redirigiendo la salida de error a /dev/null, lo que quiere decir que el comando no mostrará mensajes de error. En este caso, esto significa, que si ya existe el directorio 'dir' no se mostrará un mensaje de error.

Otra forma de saber, por ejemplo, si existe un fichero, además de [[-e fichero]], es ejecutar el comando ls sobre ese fichero. Si existe el fichero el comando funcionará bien, y si no existe dará un error. Como hemos visto, eso es equivalente a devolver verdadero o falso.

```
if [[ -e fichero]]
then
    echo “fichero ya existe”
else
    echo “fichero no existe”
fi
```

```
if ls fichero > /dev/null 2>&1
then
    echo “fichero ya existe”
else
    echo “fichero no existe”
fi
```

Como se observa, al ejecutar el comando '**ls fichero**', no nos interesa ni mensaje con el listado

con el fichero, ni el mensaje de error que pueda dar, solamente si funciona o no. Por eso redirigimos ambas salidas a **/dev/null** , para que no nos muestre ningún mensaje por pantalla

13 Expresiones matemáticas

Existen 2 comandos que evalúan expresiones matemáticas y lógicas. El primero de ellos es el comando **expr** que evalúa una expresión matemática, lógica, e incluso con cadenas de texto, y devuelve el resultado de la evaluación por la salida estándar.

expr 3 + 4
7

Este comando acepta los operadores matemáticos +, -, *, / y % (resto de una división). También acepta los operadores lógicos >, < , >=, <=, = y != (distinto de). También acepta operadores de cadenas de texto como '**length “cadena de texto”**' que te muestra la longitud que tiene una cadena de texto, o 'tes “cadena de texto” inicio longitud', que saca un trozo del texto con una longitud indicada empezando desde la posición indicada también.

expr length “esto es un texto”
16
Desde el carácter número 6 -> 'q', devuelve 3 caracteres.
expr substr “hola que tal” 6 3
que
Desde el carácter número 4 -> 'a', devuelve 7 caracteres.
expr substr “hola que tal” 4 7
a que t

- Los caracteres especiales, como por ejemplo *, >, <, (,), han de 'escaparse', es decir ponerles la barra invertida '\' delante para que se interpreten como caracteres normales.
- Todos los operadores y operandos deben de estar separados por un espacio entre si.
- Se pueden agrupar expresiones entre paréntesis
- Los operadores lógicos devuelven 1 si es verdadero y 0 si es falso.

Más ejemplos:

expr 2 * 3
6

expr 3 \< 2
0

expr \(2 + 2 \) \> 3
1

El otro comando que evalúa expresiones matemáticas y lógicas es **let**. Está limitado a este tipo de expresiones, pero es bastante más flexible que el anterior. Con este comando asignamos la evaluación de una expresión a una variable (es aconsejable casi siempre encerrar la expresión entre comillas ya que nos ahorrará problemas con caracteres especiales. El comando expr no soporta las comillas, pero let si).

Ejemplos:

let “var = 4 + 3”
echo $var
7

let “var = 4 > (6 – 4)”
echo $var
0

let “var = 4”
let “var = $var * 3”
echo $var
12

No hace falta con este comando, separar los operadores y los operandos por espacios, además, al estar encerrada la expresión entre comillas no hace falta 'escapar' los caracteres especiales.

Ahora un par de equivalencias útiles:
let “var = $var + 1” → let var++
let “var = $var - 1” → let var--

Por último, vamos a ver la diferencia entre asignar un valor a una variable utilizando el comando **expr** y el comando **let**.

```
# Forma clásica con expr, asígnar a una variable la salida de un comando
var=$(expr 3 + 4)

# Con let es más intuitivo
let “var = 3 + 4”
```

Importante: cuando se utiliza let, a diferencia de expr, utilizar '=' no significa comparar si dos números son iguales, sino asignar un valor a una variable. Para comparar si dos números son iguales se utiliza el doble igual '==' (dos símbolos de igual pegados).

Por último, tenemos el comando **bc**, que nos permite hacer operaciones con **decimales**. Como en realidad este comando es muy potente (se puede programar con él) suele leer los datos desde un archivo, pero en nuestro caso podemos pasarselos con una tubería de la siguiente forma:

variable=$(echo “5.5 * 2.5” | bc -l)

14 Variables avanzadas

Las variables siempre por defecto guardan strings, pero tienen atributos extra que podemos establecer con **declare**. Si usamos '+' en lugar de '-' delante de la opción, deshabilitamos la característica.

- **declare -i variable** → La variable sólo podrá guardar enteros. Si intentas asignarle un string, se pondrá a 0. Utiliza internamente el comando let, así que podemos asignarle un string con una operación, y la resolverá, asignando el resultado.
  - **declare -i var**
  - **var="4+5"** → var valdrá 9**.**

- **declare +i variable** → Borra la restricción de que la variable sea un entero.
- **declare -r VAR="Valor"** → Declara una constante (sólo lectura)
- **declare -a variable** → Fuerza a la variable a ser un array
- **declare -A variable** → Declara la variable como un array ASOCIATIVO. Sólo soportados por versiones de bash desde la 4 (MAC OS suele venir con la 3).
- **declare -p variable** → Te muestra el contenido de la variable y sus atributos. Muestra los arrays con índices al estilo de la función print_r de PHP.
- **declare -l variable** → Convierte la cadena que se le asigne a minúsculas automáticamente.
- **declare -u variable** → Convierte la cadena que se le asigne a mayúsculas automáticamente.
- **declare -i var1=0 var2=10 var3=4** → Varias declaraciones en la misma línea.
- **help declare** → Muestra una ayuda del comando (sirve con muchos comandos).

15 Metacaracteres y algunos caracteres especiales

Los metacaracteres son caracteres especiales que tienen un significado determinado cuando los utilizamos:

*****	Significa 'cualquier cosa', es decir 0 o más caracteres, da igual cuales.
?	Significa 'cualquier carácter'. 1 carácter cualquiera.
[]	Cualquiera de los caracteres encerrados entre los corchetes.

dir* → palabra dir, seguida de cualquier cosa (dir, direct, dir1, directorio, ...)
dir? → palabra dir, seguida de 1 carácter cualquiera (dir1, dira, dirz, dir7, ...)
dir[1234] → palabra dir, seguida de uno de los caracteres entre corchetes (dir1, dir2, dir3 o dir4). Por ejemplo 'dir9' no sería válida.
dir[a-j] → palabra dir seguida de una letra entre la 'a' y la 'j'. El guión significa un rango entre el primer carater y el segundo (dira, dirf, dird). Por ejemplo 'dirt' no sería válido.

Se pueden unir en una palabra tantos metacaracteres como queramos:

ls a*t?[1-9] → Lista archivos que se llamen por ejemplo: ate1, aperiti2, algo8, alt91.

Con el símbolo **'!'** O **'^'** negamos el carácter o el rango que tengamos detrás.

ls [!a]* → Archivos que **no** empiecen por la letra a
ls [!0-9]* → Archivos que **no** empiecen por un número
mv ?m[a-z].txt → Archivos cuyo primer carácter sea cualquiera, la segunda letra sea una m, la tercera, una letra entre 'a' y 'z', y después la extensión '.txt'.

Existen caracteres especiales representar para cadenas de texto, que no simbolizan letras ni números, sino que simbolizan espacios, tabulaciones, saltos de línea, etc... Estos caracteres se pueden consultar por ejemplo en el manual del comando echo (man echo). Algunos son:

\n → Salto de línea
\t → Tabulación

Para activar estos caracteres utilizando **echo**, hay que utilizar la opción -e. Ejemplos:

echo -e "Esto es una línea.\nEsta es otra"
Esto es una línea
Esto es otra

echo -n -e "1)Opción 1 \n2)Opción 2\n\tElige una opción: "
1)Opción 1
2)Opción 2
Elige una opción:

16 Estructura *case*

Esta estructura es muy fácil de entender una vez entendido como funciona la estructura if. Ya que el funcionamiento es similar, pero más limitado. La estructura es la siguiente:

```
# Se pueden poner tantas opciones como se quieran
case valor in
   valor1)
        #ordenes a ejecutar si el valor coincide con valor1;;
   valor2)
        #ordenes a ejecutar si el valor coincide con valor2;;
   valor3)
        #ordenes a ejecutar si el valor coincide con valor3;;

esac
```

Importante: Las instrucciones a ejecutar se pueden poner inmediatamente a continuación del valor o en la línea siguiente, eso no importa, lo que sí hay que tener en cuenta es que después de la última instrucción dentro de un valor debe haber siempre **dos punto y coma ';;'** seguidos, justo después de la instrucción o en la línea de abajo, como se prefiera.

En este caso lo que evaluaremos siempre es el valor que tiene una variable, y dependiendo del valor, se ejecutaran unas acciones u otras, algo similar a cuando realizábamos un menú con la estructura **if**, pero más claro. Aquí podemos ver la equivalencia.

```
echo “1) opcion1”
echo “2) opcion2”
echo -e -n “\tElige una opción: ”
read opcion

case $opcion in
    1)
        echo “Opción 1 elegida”;;
    2)
        echo “Opción 2 elegida”;;
    *)
        echo “Opción no válida” 1>&2;;
esac
```

```
echo “1) opcion1”
echo “2) opcion2”
echo -e -n “\tElige una opción: ”
read opcion

if [[ $opcion = “1” ]]
then
    echo “Opción 1 elegida”
elif [[ $opcion = “2”]]
    echo “Opción 2 elegida”
else
    echo “Opción no válida” 1>&2
fi
```

En este caso, las estructuras son idénticas, aunque la opción **case** es la más limpia. Esta se evalúa de la misma forma que los **if** y **elif**, es decir va comprobando de arriba a abajo las condiciones, y cuando encuentra una válida, ejecuta las ordenes que hay a continuación.

Como ya hemos visto los metacaracteres *, ?, y [], podemos intuir que si ponemos ***** como valor significará cualquier cosa, y entrará ahí siempre que no haya entrado antes en ninguno de los otros valores (como en el caso de **if**, sólo entrará en uno de los valores cuando se ejecuta, nunca en más). Por este motivo, se puede decir que el valor *, se comporta exactamente igual que **else**, es decir, si no ha encontrado antes una condición válida, entrará ahí siempre.

Aquí podemos ver un ejemplo, simple utilizando metacaracteres. En este caso vamos a preguntar al usuario por la confirmación de algo, y vamos a comprobar si el usuario ha contestado afirmativa o negativamente. A nosotros nos da igual si el usuario contesta “s”, “si”, “S”, “Si”, “si quiero”, “n”, “NO”, etc... Lo que nos interesa para evaluar la opción es saber si la primera letra es una 's' o una 'n' (mayúscula o minúscula), y el resto del texto nos da igual. Lo podemos hacer así:

```
echo “¿Desea continuar?:”
read resp

case $resp in
   [sS]*)  #Si la respuesta es una s o una S seguido de 'lo que sea'
        echo “Ha elegido continuar”;;
   [nN]*)  #Si la respuesta es una n o una N seguido de 'lo que sea'
        echo “Ha elegido no continuar”
        exit;;
    *)
        echo “Respuesta no válida” 1>&2;;
esac
```

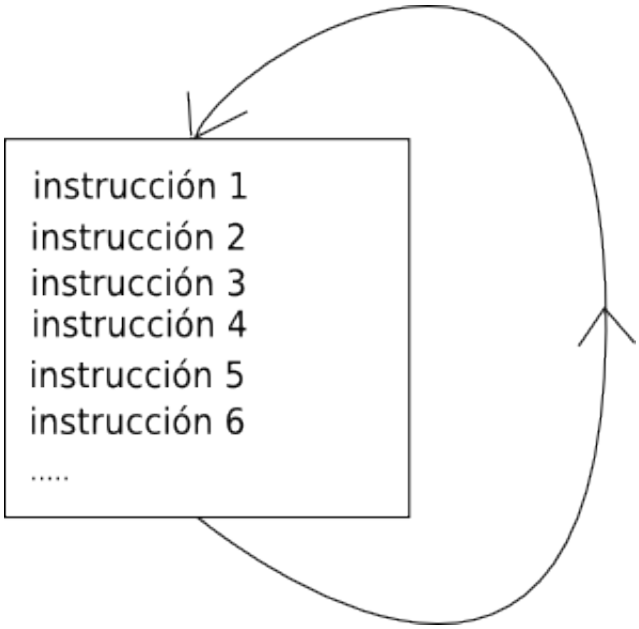
17 Bucles

Hasta ahora, las condiciones que hemos visto nos permiten ejecutar una serie de acciones diferentes según se cumpla una condición determinada, como por ejemplo que el valor de una variable sea 1, 2, 3, o 4, etc...

Lo que no sabemos hacer todavía es volver hacia atrás en un script. Siempre hemos ido hacia delante hasta que llegabamos al final del script, y este finalizaba. Pero podemos encontrarnos con la situación de que si el usuario no introduce una opción válida, podamos seguir preguntándole hasta que introduzca una opción que sí sea correcta, y de esa manera no tener que volver a ejecutar todo el script otra vez.

Se puede decir que un bucle, es un conjunto de instrucciones que se repiten, es decir, cuando llega al final de esas instrucciones, vuelve a empezar desde el principio del conjunto. Normalmente, no queremos que esas instrucciones se repitan todo el tiempo, y nunca terminen, ya que en ese caso estaríamos hablando de un **bucle infinito** (siempre se repite y nunca terminará). Por ello, estableceremos una condición o conjunto de condiciones que se deberán cumplir si queremos que las instrucciones dentro del bucle se vuelvan a repetir, o no.

Ejemplo de un bucle infinito. Podemos observar como después de ejecutar todas las instrucciones vuelve a empezar desde el principio:

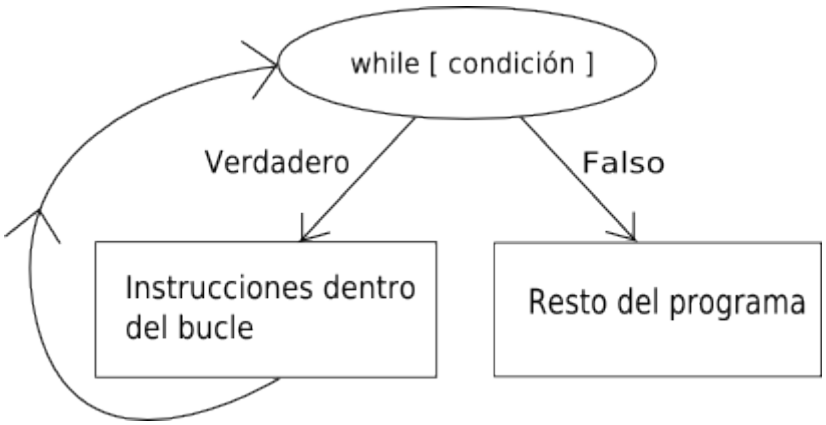


17.1 Estructura While (mientras)

Esta estructura establece un bucle dentro de si misma. Este bucle de instrucciones se repetirá siempre que la condición que pongamos sea cierta. Se puede decir que es como una estructura **if**, con la diferencia de que cuando termina, vuelve al principio a comprobar otra vez la condición una y otra vez hasta que esta no se cumpla.

La estructura **while** es así:

```
while [[ condición ]]
do
   # Instrucciones dentro del bucle
done
```



Como podemos observar, siempre que la condición sea cierta, se ejecutarán las instrucciones dentro del bucle una y otra vez, hasta que la condición sea falsa.

Esta condición nos puede servir para cuando detectamos un error, o que el usuario introduce una condición que no es válida y le queremos seguir insistiendo hasta que nos de una opción correcta.

Ejemplo:

```
#!/bin/bash

echo -n "¿Desea continuar (s/n)?: "
read -r -n 1 -s resp

# Mientras que la respuesta sea diferente de “s” y de “n”, sigue preguntando
while [[ $resp != "s" && $resp != "n" ]]
do
      echo "Respuesta no valida. ¿Desea continuar (s/n)?"
      read -r -n 1 -s  resp
done

if [[ $resp = “n” ]]
then
      exit
fi
```

Otro uso que le podemos dar a esta estructura es el de repetir algo un número determinado de veces:

```
repeticiones=3
i=0

# Mientras el contador sea menor que el total, repite
while [[ $i -lt $repeticiones ]]
do
	# Como la variable va de 0 a 2, le sumamos 1 al mostrarla
	# para que aparezcan números del 1 al 3
	echo "Repetición numero: " $(expr $i + 1)

	#incrementamos en una unidad la variable i
	let i++
done
```

Otro ejemplo:

```
echo "¿Cuantos directorios deseas crear?"
read total

num=1

# Si el número es mayor que el total, paramos de crear, mientras tanto, sí creamos.
while [[ $num -le $total ]]
do
	# Por si acaso existe el directorio, quitamos los mensajes de error
	mkdir dir${num} 2> /dev/null

	# incrementamos la variable num una unidad más para el siguiente directorio
	let num++
done

echo "Hemos creado $total directorios."
```

17.1.1 Recorrer las líneas de un archivo o salida de comando con While

Aprovechando la **redirección** <, a partir de la cual la entrada de un comando es el contenido de un archivo, y la capacidad de las estructuras de bash como **while** en este caso de evaluar comandos, podemos recorrer fácilmente las líneas de un fichero **redireccionandolo** a la estructura **while** y leyendo cada línea con la instrucción read.

```
while read -r
do
	printf "%s\n" "$REPLY"
done < archivo
```

Usando la capacidad de read de asignar varias variables, si cada línea del archivo contiene, por ejemplo, 3 campos separados por espacios (En realidad es la variable de entorno **$IFS** la que determina los separadores por defecto), podemos leer los campos de manera directa:

```
while read -r nom precio cant
do
        printf "Nombre: %s, Precio: %.2f, Cantidad: %d\n" "$nom" "precio" "cant"
done < archivo
```

¿Qué pasa si el separador de los campos no es el espacio, sino por ejemplo, el punto y coma?. Podemos modificar la variable IFS (temporalmente eso sí), asignándole el separador y todo funcionará automáticamente.

```
set -f; OLDIFS=$IFS; IFS=";"
while read -r nom precio cant
do
        printf "Nombre: %s, Precio: %.2f, Cantidad: %d\n" "$nom" "precio" "cant"
done < archivo
set +f; IFS=$OLDIFS
```

set -f es opcional y lo que hace es deshabilitar la expansión de caracteres especiales de * por nombres de archivo, cosa que bash intenta hacer siempre automáticamente. Con **set +f** se deja como estaba.

¿Y si queremos recorrer la salida de un comando redireccionada con pipe '|' en lugar del contenido de un archivo? Tampoco hay problema:

```
cat pedido.txt | sed -r 's/;/ /gi' | while read -r nom precio cant
do
        printf "Nombre: %s, Precio: %.2f, Cantidad: %d\n" "$nom" "precio" "cant"
done
```

¡Nota importante!

- Cuidado con esta última fórmula ya que lo que haya después de un pipe (en este caso la estructura **while**) se ejecuta en un subshell de bash o entorno privado (como si fuera la llamada a una función), por lo que las variables utilizadas ahí no tendrán valor en el resto del script (serán locales, aunque hayan sido declaradas o tuvieran valor antes). Para tareas como mostrar datos por pantalla o guardar datos en ficheros funciona perfectamente.
- Dentro de un while read … no podremos utilizar el comando read para leer lo que el usuario introduzca por consola ya que la entrada de datos está ocupada por el archivo en lugar de por la consola.

17.2 Estructura Until (Hasta)

La estructura **until** tiene una sintaxis idéntica a **while**, con la única diferencia que el bucle se repite **hasta** que la condición se cumpla, es decir, mientras sea falsa, el bucle se repetirá. Justo al contrario.

```
until [[ condición ]]
do
   # Instrucciones dentro del bucle
done
```

```
#!/bin/bash

# Este bucle se repetirá 9 veces
let "i = 1"
until [[ $i -gt 9 ]]
do
      echo "Repetición número $i"
      let i++
done
```

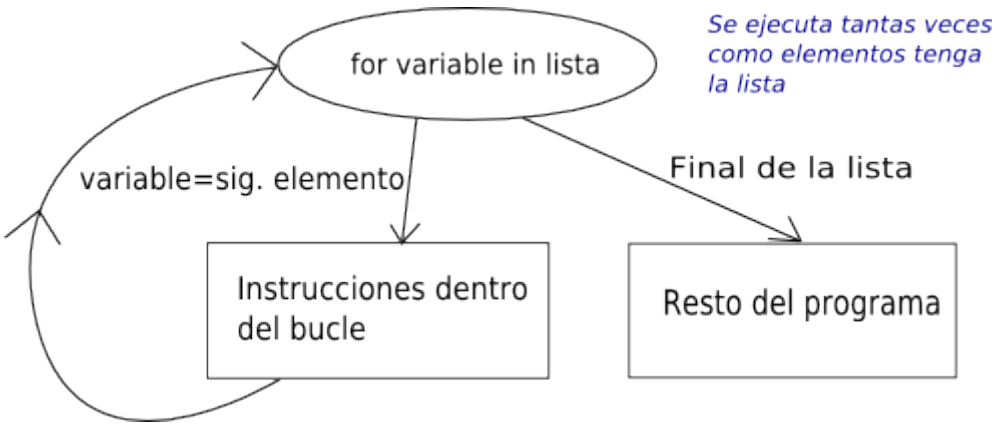
```
#!/bin/bash

# Este bucle se repetirá 9 veces
let "i = 1"
while [[ $i -le 9 ]]
do
      echo "Repetición número $i"
      let i++
done
```

17.3 Estructura For

La estructura **for** funciona de manera similar a la estructura **while**, aunque está pensada para ser usada en casos más concretos (algo similar a lo que pasa con la estructura **case** con respecto a **if**). La estructura if se puede usar de dos maneras. La primera es la siguiente:

```
for variable in lista
do
   # Instrucciones dentro del bucle
done
```



Este bucle se repetirá siempre tantas veces como elementos haya en la lista. En cada repetición la variable utiliza para iniciar el bucle tomará como valor el siguiente elemento de la lista, así en la primera repetición tomará el valor del primer elemento de la lista y así sucesivamente.

Esta lista puede ser una lista de palabras o números. Si queremos agrupar texto para que por ejemplo, si ponemos un nombre y apellido vayan juntos y no sean elementos diferentes debemos agruparlos entre comillas:

for nombre in Francisco Martínez Juan López # Se repetirá 4 veces (4 elementos)
for nombre in "Francisco Martínez" "Juan López" # Así es correcto, se repetirá 2 veces.

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

```
# Este bucle se repetirá 9 veces
for num in 1 2 3 4 5 6 7 8 9
do
	echo "Repetición número $num"
done

# Este también 9 veces pero mejor
for num in {1..9}
do
	echo "Repetición número $num"
done

# Creamos los siguientes directorios
for dir in dir1 dir2 dir3 dir4 dir5
do
	if [[ ! -e $dir ]]
	then
		mkdir $dir
	fi
done

# Mostramos los parámetros recibidos
echo "Número de parámetros $#"
num=1

for param in “$@”
do
	echo "Parámetro $num: $param"
	let num++
done
```

Comándo seq y rango numérico (nos puede ayudar con la estructura for):

- **seq 1 9** → Genera los números del 1 al 9
- **seq 1 2 9** → Igual, pero de 2 en 2 (números impares sólo).
- **seq 0.1 0.1 1** → Desde 0.1 a 1 incrementando en 0.1 cada vez.
- **seq -s '-' 1 9** → Utiliza el carácter '-' para separar los números (por defecto → \n)
- **seq -w 1 19** → Todos los números ocupan lo mismo (rellena con 0 a la izquierda)
- **seq -f '%.2f' 1 2.5 21** → Con **-f** indicas el formato decimal al estilo de **printf**

- **{1..9}** → Parecido a **seq 1 9**. Probad con **echo {1..9}**, y **echo “Num: “{1..9}**
- **{01..19}** → Números de 2 cifras
- **{0..9..2}** → Equivale a **seq 1 2 9**

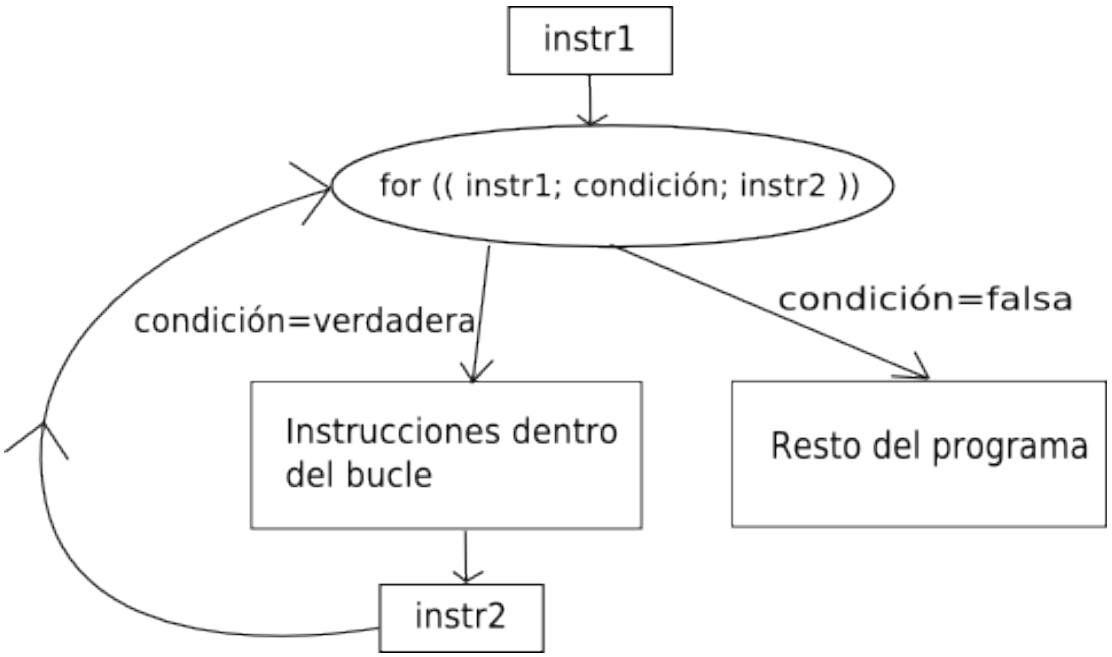
for i in {1..9}
for i in $(seq 1 9)

http://wiki.bash-hackers.org/syntax/expansion/brace

17.4 Estructura For estilo C

Existe, sin embargo, otra forma de utilizar el bucle **for**. Esta otra forma es muy útil sobre todo cuando queremos hacer un contador, y queda más limpio que utilizando la estructura **while**:

```
for (( instr1; condición; instr2 ))
do
    # Instrucciones que se repiten hasta que condición sea falsa
done
```



En esta estructura tenemos un bucle **for**, y encerrados entre un doble paréntesis una instrucción que se ejecutará justo antes de comenzar el bucle por primera vez (**instr1**), una **condición** que indicará si se ejecutan las instrucciones dentro del bucle o se terminará y una instrucción final que se ejecutará como última instrucción dentro del bucle (**instr2**). Se podría expresar así en pseudocódigo:

```
instr1
mientras condición=verdadera
   instrucciones dentro del bucle
   instr2
fmientras
```

Importante: las instrucciones y condiciones dentro del doble paréntesis se escriben de forma casi idéntica a las instrucciones cuando se utiliza el comando **let**. De esta manera, una condición de si un número es mayor que otro se escribirá **n1 > n2**, si se quiere comparar si dos números son iguales se escribirá **n1 == n2**, etc... (de forma idéntica a cuando se utiliza **let**)

Comparativa contador con **for** y **while**

#!/bin/bash # Este bucle se repetirá 9 veces **for** ((i = 1; i <= 9; i++)) **do**	#!/bin/bash # Este bucle se repetirá 9 veces let "i = 1" while [[$i -le 9]]

```
    echo "Repetición número $i"
done
```

```
do
    echo "Repetición número $i"
    let i++
done
```

En el ejemplo con **for**, realiza los pasos de forma idéntica a como los realiza en el ejemplo con **while**, pero sin embargo nos estamos ahorrando 2 líneas en el código del programa.

17.5 Ruptura de bucles: 'break' y 'continue'

Hasta ahora hemos visto que para salir de un bucle la condición de entrada en dicho bucle debe ser falsa, y además, sabemos que la instrucciones dentro del bucle se ejecutarán hasta el final antes de volver a repetirse.

Sin embargo, hay algunas ocasiones en las que podría interesarnos salir del bucle debido por ejemplo a un error o a una situación especial, sin esperar a que la condición de dicho bucle se evalúe a falso. En otras ocasiones podría interesarnos saltarnos alguna de las iteraciones(vueltas) del bucle y pasar a la siguiente. Para ello tenemos 2 instrucciones útiles:

break: Esta instrucción al ejecutarse, nos hace salir del bucle donde nos encontramos. El resto de instrucciones dentro del bucle no se llegan a ejecutar y además ya no se comprueba la condición de continuidad de bucle, simplemente salimos de él.

Aunque puede resultar bastante útil en ciertas ocasiones, siempre que se pueda es mejor utilizar la condición del bucle para salir de él. La instrucción **break** puede servirnos sobretodo para salir de un bucle al encontrar un error o algo inesperado y no pudiésemos seguir ejecutando el bucle con normalidad.

```
#!/bin/bash

encontrado=0

# Vamos a buscar el archivo 'arch' en una lista de directorios
for dir in dir1 dir2 dir3 dir4 dir5
do
    if [[ -e "${dir}/arch" ]]
    then
        encontrado=1
        # Ya no tiene sentido seguir buscando
        break
    fi
done

if [[ $encontrado -eq 0 ]]
then
    echo "Archivo no encontrado"
else
    echo "Archivo encontrado en $dir"
fi
```

continue: Esta instrucción, a diferencia de **break**, no termina el bucle, sino que se salta las instrucciones que haya desde la palabra **continue** hasta el final del bucle y pasa a la siguiente iteración (vuelta) del bucle. Esto nos puede resultar muy útil cuando queremos hacer algo con una lista de cosas pero tenemos algún tipo de excepción.

```
#!/bin/bash

echo -n "Introduce número máximo al que llegaremos: "
read nummax

echo -n "Introduce un número para que los números divisibles entre él no aparezcan: "
read divisor

for (( i = 1; i <= $nummax; i++ ))
do
	let "resto = $i % $divisor"
	if [[ $resto -eq 0 ]]
	then
		# Si es divisible nos saltamos el resto y pasamos al siguiente
		continue
	fi

	echo "$i "
done
```

Como se ha podido observar, la instrucción **continue**, en el caso de un bucle del tipo "**for** ((instr1; condición; instr2))", no afecta a **instr2**, que se sigue ejecutando siempre al final del bucle.

Importante: aunque estas dos instrucciones nos pueden resultar muy útiles en varias ocasiones, no conviene abusar de ellas (sólo utilizarlas cuando de verdad se necesite), ya que podrían hacer el código del programa más difícil de entender al alterar el comportamiento normal de un bucle e introducir comportamientos no esperados.

18 Saltos de línea en los scripts: ';' y '\'

En ocasiones, tal vez por elegancia, o por pura claridad (aunque suele ser subjetivo), nos podría interesar tener más de 1 instrucción que irían en líneas separadas, en una misma línea o viceversa. Tener una instrucción de 1 sola línea dividida en varias debido a su longitud.

El símbolo punto y coma **';'**, se utiliza para poder poner a continuación, lo que pondríamos el la línea siguiente. Se podría decir que es como introducir un salto de línea de forma artificial, aunque hay excepciones: detrás de las palabras especiales **then** y **do**, por ejemplo, aunque normalmente hagamos un salto de línea este no es obligatorio, y por lo tanto, no se puede poner punto y coma entre la palbara y la primera instrucción.

```
#!/bin/bash

echo -n "dime tu edad: "; read edad
if [[ $edad -lt 18 ]]; then echo "menor de edad"; else echo "mayor de edad"; fi
```

Por el contrario, la barra invertida '\' puede servir para lo contrario. Ya hemos explicado que la barra invertida sirva para que un carácter especial sea interpretado como si fuera texto normal. Pues bien, esto también es válido para el salto de línea, considerado un carácter especial. Cuando **justo delante** de un salto de línea se pone una barra invertida este pasa a ser como un espacio normal y corriente o como un tabulador, es decir, lo que haya en la línea de abajo es como si siguiera estando en la línea superior.

```
#!/bin/bash

echo "Este es un mensaje muy largo, y por eso voy a dividirlo \
en dos partes. Y así me cabe sin problemas, o tal vez debiera \
dividirlo en más. Lo que no hay que olvidar es que a continuación \
de la barra invertida es donde tiene que estar el salto de línea."

for dir in dir1 dir2 dir3 dir4 dir5 dir6 dir7 dir8 dir9 \
dir10 dir11 dir12 dir13 dir14 dir15 dir16
do
    mkdir $dir
done
```

19 Vectores (Arrays)

Un vector o array, básicamente se puede decir que es una variable que en lugar de contener un sólo valor, puede contener varios. Para acceder a cada valor, se utiliza un índice, que es el número que indica la posición que ocupan dentro del vector.

La forma más común de asignar valores inicialmente a un vector es similar a la que se utilizaría para asignar un valor a una variable normal, simplemente hay que encerrar todos los valores entre paréntesis.

vector=(valor1 valor2 valor3 valor4)

Los valores, al igual que en una lista de valores deben de estar separados por un espacio entre si. Para acceder a cada valor, simplemente hay que poner el índice entre corchetes después del nombre de la variable. El índice del primer valor siempre es **0**. El índice del último es el **número de valores – 1**. Es decir, en un vector con 4 elementos, los índices irían del 0 al 3.

```
#!/bin/bash

vector=( 10 5 2 6 )

echo "Primer elemento: ${vector[0]}"
echo "Segundo elemento: ${vector[1]}"
echo "Tercer elemento: ${vector[2]}"
echo "Ultimo elemento: ${vector[3]}"
echo "Elemento no existente: ${vector[5]}"
```

Como se ve, si intentamos acceder a un elemento que no existe, el que tiene índice 5 en este ejemplo, es como acceder a una variable a la que no hemos dado valor todavía, estaría vacía.

Digamos que la estructura de la variable **vector** en este caso se podría representar así.

INDICE	0	1	2	3
VALOR	10	5	2	6

Podemos añadir un elemento adicional al vector, o modificar uno existente, de la misma forma que accedemos a él.

vector[2]=15
vector[4]=3

INDICE	0	1	2	3	4
VALOR	10	5	15	6	3

Podemos asignar un valor a un índice, por ejemplo el 7, dejando índices vacíos en medio, **aunque esta no es un práctica muy aconsejable en la mayoría de las ocasiones**.

vector[7]=20

INDICE	0	1	2	3	4	5	6	7
VALOR	10	5	15	6	3			20

Hay un propiedad muy útil en la vectores, y es el hecho de que si utilizamos **'@'** como índice, nos muestra el vector como si fuera una lista.

${#array[@]} → Te devuelve el número de elementos que contiene el array.
${!array[@]} → Te devuelve los índices del array, aunque haya huecos.

Diferencia entre [*] y [@] (Para la lista de parámetros $* y $@) → Si ${variable[@]} está entre comillas, se sustituirá por los valores que contiene pero cada uno entre comillas (**mejor opción**). En el caso de [*] se entenderá que todo es un único string entre comillas. **for a in "${array[@]}"**

```
#!/bin/bash

vector=( uno dos tres cuatro cinco seis )

# Vamos a mostrar los valores del vector por pantalla
echo "${vector[@]}"
```

Gracias a esto podemos recorrer un vector de la siguiente forma:

```
#!/bin/bash

vector=( uno dos tres cuatro cinco seis )

for valor in ${vector[@]}
do
        echo -n "$valor "
done
```

```
for i in ${!vector[@]}
do
	echo -n "${vector[$i]}"
done
```

Otra propiedad es cómo obtener el número de elementos que contiene un vector:

```
#!/bin/bash

vector=( uno dos tres cuatro cinco seis )

# Vamos a mostrar el número de elementos del vector por pantalla
echo "Numero de elementos: ${#vector[@]} "
```

Y por ello sería equivalente recorrer los valores del vector de la siguiente forma:

```
#!/bin/bash

vector=( uno dos tres cuatro cinco seis )

# De esta manera debemos saber que hay 6 elementos en el vector
for (( i = 0; i < 6; i++ ))
do
	echo "Elemento ${i}: ${vector[$i]} "
done

# Aquí no hace falta saber cuantos elementos tenemos
for (( i = 0; i < ${#vector[*]}; i++ ))
do
	echo "Elemento ${i}: ${vector[$i]} "
done
```

Nota: Se puede borrar el valor de una posición del vector con el comando **unset**. Asimismo también se puede borrar el valor asignado a una variable (es como si ya no existiese).

vector=(4 5 3 6 7)
unset vector[4]

Ahora **vector** tendrá 4 elementos en lugar de 5, ya que hemos eliminado el último (índice 4).

También podemos inicialmente dar los valores uno a uno, en lugar de asignarle varios valores a la vez. En este caso, el orden no importa, pero no es aconsejable dejar indices sin valor.

vector[0]=”uno” vector[2]=”tres” vector[1]=”dos”	vector=(“uno” “dos” “tres”)

20 El doble paréntesis ((..))

El doble paréntesis, cuyo uso en un caso muy particular ya hemos visto con el bucle **for**, forma un estructura con funcionamiento muy similar al comando let. De hecho podemos hacer lo mismo que hacíamos con let (El dólar para leer las variables sigue siendo opcional en este caso solamente, sólo para leer, al darles valor siempre van sin dolar):

```
#!/bin/bash

(( n = 3 + 5 ))

echo $n    # La variable n valdrá 8
```

Si en lugar de usarlo como el **let** (asignar un valor a una variable), queremos que funcione más bien como usar $(**expr operación**) basta con ponerle el dólar '$' delante y se sustituirá por el resultado de la operación.

```
#!/bin/bash

n=5

echo “El doble de $n es $((n * 2))”   # El doble de 5 es 10
```

Lo mejor de usar el doble paréntesis, es que cuando ponemos una condición lógica dentro devuelve un valor verdadero o falso, con lo cual se hace posible utilizarlo como condición, para un **if** o para un bucle **while** por ejemplo:

```
#!/bin/bash

i=0

# Contador desde i=0 a i=4
while (( i < 5 ))
do
	if (( (i % 2) == 0 )) #Numero par
	then
		echo "Numero par: $i"
	else #Numero impar
		echo "Numero impar: $i"
	fi

	(( i++ ))
done
```

Otra posibilidad del doble paréntesis es la de poder hacer operaciones con vaariables, para

mostrarlas por pantalla, por ejemplo, si tenemos una posición de un array, y queremos mostrar una posición más (para empezar desde la 1 y no desde la 0), en ese caso se pone el símbolo $, seguido del doble paréntesis y la operación dentro. Ejemplo:

```
for (( i = 0; i < ${#vector[*]}; i++ ))
do
      echo "Elemento $((i+1)): ${vector[$i]} "
done
```

Finalmente cabe resaltar que todavía no hemos aprendido todas las posibilidades de programar con **bash script**. Aunque lo aprendido nos servirá para salir airosos de la mayoría de las situaciones, hay ocasiones en las que se requieren conocimientos más avanzados, y por ello, quien esté interesado puede buscar y consultar en los cientos de tutoriales y ejemplos que existen en internet. Allí podrás encontrar nuevas posibilidades para programar con bash script. ¡Simplemente usa tu buscador favorito para ello!.

21 Operando con decimales

Como ya hemos visto en las expresiones matemáticas, tenemos un comando llamado **bc** que nos permite operar con decimales (la opción -l utiliza la libería matemática del sistema y obtiene más precisión). También podríamos usar **awk**, pero su sintaxis es más compleja, y en este caso con **bc** nos sobra.

Bc es un lenguaje de programación sencillo orientado a operaciones matemáticas, soporta por ejemplo, condicionales, bucles, funciones, etc. Pero no vamos a profundizar en eso. Lo único que tendremos que saber es que las instrucciones se separan por punto y coma “;” (salto de línea también cuando están en un fichero). Sabiendo esto, podemos ver como cambiar la precisión del resultado usando la instrucción scale=precisión.

echo "5 * 7 /3.0" | bc -l
11.66666666666666666666

echo "scale=2; 5 * 7 /3" | bc -l
11.66

echo "scale=0; 5 * 7 /3" | bc -l
11

¿Cómo cambiariamos la precisión de un valor ya guardado en una variable?: Usando la instrucción scale y dividiendo esa variable entre 1 por ejemplo, para que no cambie su valor.

```
num=$(echo "5 * 7 /3.0" | bc -l) # num almacena 11.66666666666666666
num=$(echo "scale=2; $num/1" | bc -l) # Ahora num almacena 11.66
```

La cuestión es que el resto de operaciones con números, como puede ser una comparación en un condicional o bucle, tampoco entienden los decimales, así que tenemos que recurrir a este comando, que por suerte también soporta operaciones lógicas. Con bc, una operación lógica devuelve 0 (false), o 1 (true), así que podemos jugar con esto.

Cuando hacemos comparaciones con los corchetes basta comprobar si el resultado es igual a 0 o 1.

```
a=$(echo "5/3" | bc -l)
if [[ $(echo "$a < 2" | bc -l) -eq 1 ]] # true
then
    echo “Si”
fi
```

Si usamos doble paréntesis, la cosa es más sencilla, ya que no hay que comparar nada, 0 significa false directamente, y 1 significa true.

```
if (( $(echo "$a < 2" | bc -l)  )) # true
then
    echo “Si”
fi
```

22 Operando con cadenas

Bash, que tiene más posibilidades de las que se suelen conocer, también permite ciertas operaciones con cadenas (strings), pero no en forma de funciones o métodos como otros lenguajes, sino de operadores especiales.

Longitud de una cadena

```
cadena="Texto de prueba”
echo ${#cadena} # Muestra 15
```

Extraer un trozo de cadena (substring)

${variable:posicion} # Extrae desde la posición hasta el final (primer carácter → pos. 0)

${variable:posicion:longitud} # Extrae como máximo longitud caracteres desde posición

```
cadena="Texto de prueba”
echo ${cadena:3:5} # Muestra “to de”
```

Borrar el principio o final de una cadena

Hay que tener en cuenta que **no estamos usando expresiones regulares aquí**. El carácter “.”

es normal, y asterísco “*” indica 0 o más caracteres. Podemos usar corchetes para rangos de caracteres y “?” para 1 carácter cualquiera.

${string#substring} # Borra la coincidencia más **corta** posible desde el inicio.

${string%substring} # Borra la coincidencia más **corta** desde el final.

```
filename="bash.string.txt"
echo ${filename#*.} # Muestra string.txt
echo ${filename%.*} # Muestra bash.string
```

${string##substring} # Borra la coincidencia más **larga** posible desde el inicio.

${string%%substring} # Borra la coincidencia más **larga** desde el final.

```
filename="bash.string.txt"
echo ${filename##*.} # Muestra txt
echo ${filename%%.*} # Muestra bash
```

Reemplazar trozos de una cadena por otra cosa

Esto lo podríamos hacer con sed por ejemplo, usando una tubería, pero para operaciones sencillas de este tipo a lo mejor preferimos a veces usar esta otra alternativa (**Igual que antes, no trabaja con expresiones regulares**).

${string/patron/reemplazo} # Reemplaza la primera coincidencia

${string//patron/reemplazo} # Reemplaza todas las coincidencias

```
cad="Mi carpeta es /home/arturo"
echo ${cad/arturo/guest} # Muestra "Mi carpeta es /home/guest"
cad2="archivo.txt, cosas.txt, casa.txt, cielo.doc"
echo ${cad2//.txt/.dat} # Muestra "archivo.dat, cosas.dat, casa.dat, cielo.doc"
```

Cuidado con el asterísco, si ponemos **${cad2//c*.txt/nada.dat}**, mostrará "arnada.dat, cielo.doc", es decir, lo cogerá **todo** lo que pueda hasta el último **.txt**.

Otras opciones útiles:

${var/#patron/cadena} → Sustituye sólo si el patrón es el comienzo de la cadena
${var/%patron/cadena} → Sustituye sólo si el patrón es el final de la cadena

${var:-valor} → Si la variable está vacía o no existe devolverá este valor por defecto
${var-valor} → Como antes pero sólo si la variable no existe

${var:=valor} → Además de ser igual que **:-**, le asigna a la variable el valor.
${var=valor} → Además de ser igual que **-**, le asigna a la variable el valor.

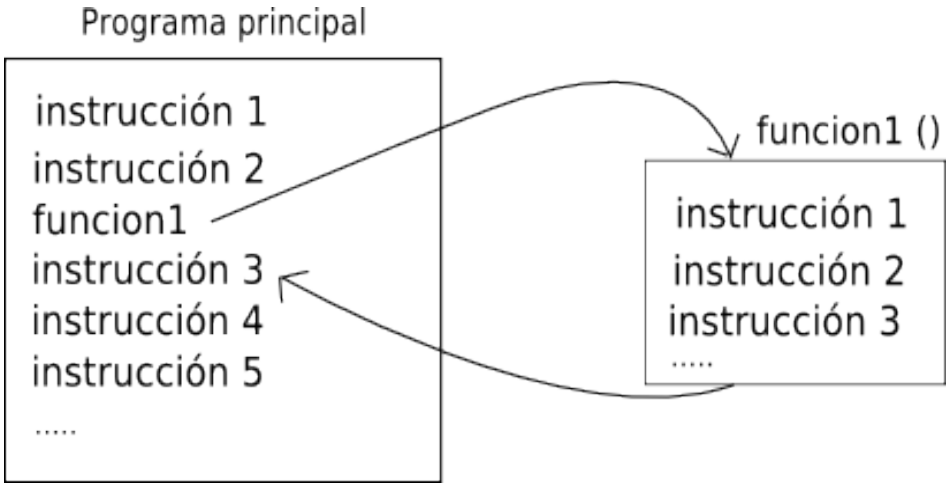
${var,} → Convierte a minúsculas la primera letra
${var,,} → Convierte todo a minúsculas
${var,,[AEIOU]} → Convierte a minúsculas sólo las vocales

${var^} → Convierte a mayúculas la primera letra
${var^^} → Convierte todo a mayúculas
${var^^[AEIOU]} → Convierte a mayúculas sólo las vocales.

23 Funciones

Aunque no vamos a profundizar demasiado en este tema, vamos a aprender lo que es una función y su utilidad de forma muy básica.

Una función se puede describir como un trozo del código o conjunto de instrucciones que se agrupan para realizar alguna función específica dentro del programa. Esta función específica puede ser una operación matemática, una búsqueda, ordenación, etc...



Una función se define con un nombre seguido de '()' y '{'. A continuación se escriben las instrucciones que formarán parte de la función y se terminará cerrando la llave '}'.

Para llamar a esa función simplemente hay que poner el nombre de la función en algún lugar del código y cuando se ejecute, nos trasladará al código dentro de la función. Cuando la función acabe volveremos al programa principal después de la llamada a la función.

```
#!/bin/bash

# Aquí sólo estamos definiendo la función,
# no se ejecutará hasta que se la llame
funcion () {
        echo "Soy una función"
}
```

```
echo "Vamos a llamar a la función..."
funcion
```

Una función, al igual que el programa principal cuando se ejecuta, puede recibir parámetro, y trabaja con ellos de forma idéntica. Es decir el primer parámetro se accederá con $1, el segundo con $2, el número de parámetros con $#, la lista con $*, etc...

Hay que tener en cuenta, que debido a ello, una función no tiene acceso directamente a los parámetros que recibe el programa, sino solamente a los parámetros con los cuales se llama a la función.

```
#!/bin/bash

# Aquí sólo estamos definiendo la función,
# no se ejecutará hasta que se la llame
funcion () {
	echo "He recibido $# parámetros"
	echo "Parametro 1: $1"
	echo "Parametro 2: $2"
}

funcion "par1" "par2"
```

La palabra **return** dentro de una función tiene una función similar a **exit** en el programa, sale de la función devolviendo un número que puede ser un valor entre 0 y 255. El valor de retorno de la última función llamada queda guardada en la variable **$?**.

```
#!/bin/bash

declare -r CORRECTO=0
declare -r ERROR=1

# Esta función comprueba si un archivo existe.
# Si existe devuelve 1->Verdadero, y si no 0->Falso
existe () {
	if [[ -e $1 ]]
	then
		return $CORRECTO
	else
		return $ERROR
	fi
}

existe "archivo1.txt"

# Comprobamos el valor devuelto por la función
if [[ $? -eq $CORRECTO ]]
then
	echo "El archivo existe."
else
	echo "El archivo NO existe."
```

```
fi

# Al devolver 0 si todo va bien, y distinto de 0 cuando hay error, llamar a la función es lo
# mismo que llamar a cualquier otro comando
if existe "archivo1.txt"
then
	echo "El archivo existe."
else
	echo "El archivo NO existe."
fi

# Repasemos esta otra posibilidad
existe "archivo1.txt" && echo "El archivo existe." || echo "El archivo NO existe."
```

Si lo último que ejecuta la función es un comando, y la salida de la función va a depender de si el comando falla o no, podríamos ahorrarnos la instrucción **return**, ya que la salida de ese comando se guardará en la variable global **$?** igualmente.

```
#!/bin/bash

# Función que comprueba si es un número el parámetro
isNumber() {
	[[ $1 =~ ^[0-9]+$ ]]
}

read -p "Dime tu edad: " edad
if ! isNumber $edad
then
	echo “Error: debes indicar un número”; exit 1;
fi
```

De esta manera podemos indicar que todos los mensajes de una función, por defecto, se redirijan a la salida de error.

```
#!/bin/bash

error () {
	//Mensajes de error
} >&2
```

Por último, y bastante importante ya que no tiene nada que ver con la gran mayoría de los lenguajes de programación y muestra una vez más las limitaciones que tiene bash script con las funciones, hablamos de las variables dentro de una función.

Las variables que se crean dentro de una función son globales, es decir, pueden ser accedidas posteriormente desde el programa principal y mantendrán el valor dado dentro de la función. Si queremos que una variable sólo exista dentro de una función, es decir sea local a la función, simplemente hay que poner la palabra **local** o utilizar **declare** delante cuando le demos valor. De esta forma, la variable local dejará de existir cuando salgamos de la función (sería como ejecutar **unset** variable al salir de la función).

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

```
suma () {
	local num1=$1
	declare num2=$2
	let "resultado = $num1 + $num2"
}

suma 4 6

# num1 y num2 no existen fuera de la función al ser locales
# así que no los mostrará por pantalla. Resultado sin embargo
# no ha sido definida como local, así que estará accesible desde fuera.
echo "$num1 + $num2 = $resultado"
```

Es importante saber que la funciones en un script de bash están más limitadas y se saltan algunas reglas que deben cumplir en otros lenguajes de programación, y que cuando sea posible se deberían intentar respetar:

- La funciones deberían tener un nombre único, es decir, no debería haber ninguna variable que se llamase igual. En bash script, sin embargo, te permite hacer esto (para acceder a la variable pones '$' delante, y si no, accedes a la función. Sin embargo, al funcionar de forma parecida a los comandos, no debería llamarse a la función como un comando existente.
- Las variables utilizadas en las funciones deberían ser locales, es decir, existir sólo para esa función, y sin embargo en bash script a menos que nosotros le digamos que son locales, todas las variables serán globales (accesibles y modificables en cualquier parte del programa).
- Los valores que devuelve una función se deberían utilizar como si fueran los que devuelve un comando con **exit**. De esta manera, una función se comportará como un comando interno del script a todos los efectos.
- Normalmente se define la cantidad de parámetros que puede (y debe) recibir una función (y muchas veces el tipo de parámetro también). Este no es el caso de bash script, en el cual podemos enviar a una función cualquiera una cantidad indeterminada de parámetros.

1 ANEXO

1.1 Definir color y posición del texto en la consola

Para hacer que el texto aparezca de un determinado color o en una determinada posición de la consola (definida por número de columna y de fila), se utilizan una serie de códigos especiales, que luego serán interpretados por el comando **echo** con la opción **-e** (igual que cuando queremos que funcionen los saltos de línea o tabuladores \n\t).

Estos son los códigos para colores y estilos de letra (subrayado, negrita), lo más útil es guardarlos en variables por ejemplo, al principio de nuestro script o en otro al cual llamaremos desde el nuestro (lo ejecutamos).

```
# Resetear el formato de texto (volver al original)
Color_Off='\e[0m'       # Text Reset

# Colores normales
Black='\e[0;30m'        # Black
Red='\e[0;31m'          # Red
Green='\e[0;32m'        # Green
Yellow='\e[0;33m'       # Yellow
Blue='\e[0;34m'         # Blue
Purple='\e[0;35m'       # Purple
Cyan='\e[0;36m'         # Cyan
White='\e[0;37m'        # White

# Colores + negrita
BBlack='\e[1;30m'       # Black
BRed='\e[1;31m'         # Red
BGreen='\e[1;32m'       # Green
BYellow='\e[1;33m'      # Yellow
BBlue='\e[1;34m'        # Blue
BPurple='\e[1;35m'      # Purple
BCyan='\e[1;36m'        # Cyan
BWhite='\e[1;37m'       # White

# Colores + subrayado
UBlack='\e[4;30m'       # Black
URed='\e[4;31m'         # Red
UGreen='\e[4;32m'       # Green
UYellow='\e[4;33m'      # Yellow
UBlue='\e[4;34m'        # Blue
UPurple='\e[4;35m'      # Purple
UCyan='\e[4;36m'        # Cyan
UWhite='\e[4;37m'       # White

# Color de fondo (anteriores era para el texto)
On_Black='\e[40m'       # Black
On_Red='\e[41m'         # Red
On_Green='\e[42m'       # Green
On_Yellow='\e[43m'      # Yellow
On_Blue='\e[44m'        # Blue
On_Purple='\e[45m'      # Purple
On_Cyan='\e[46m'        # Cyan
On_White='\e[47m'       # White
```

```
# Colores de alta intensidad
IBlack='\e[0;90m'       # Black
IRed='\e[0;91m'         # Red
IGreen='\e[0;92m'       # Green
IYellow='\e[0;93m'      # Yellow
IBlue='\e[0;94m'        # Blue
IPurple='\e[0;95m'      # Purple
ICyan='\e[0;96m'        # Cyan
IWhite='\e[0;97m'       # White

# Colores de alta intensidad + negrita
BIBlack='\e[1;90m'      # Black
BIRed='\e[1;91m'        # Red
BIGreen='\e[1;92m'      # Green
BIYellow='\e[1;93m'     # Yellow
BIBlue='\e[1;94m'       # Blue
BIPurple='\e[1;95m'     # Purple
BICyan='\e[1;96m'       # Cyan
BIWhite='\e[1;97m'      # White

# Colores de fondo de alta intensidad
On_IBlack='\e[0;100m'   # Black
On_IRed='\e[0;101m'     # Red
On_IGreen='\e[0;102m'   # Green
On_IYellow='\e[0;103m'  # Yellow
On_IBlue='\e[0;104m'    # Blue
On_IPurple='\e[10;95m'  # Purple
On_ICyan='\e[0;106m'    # Cyan
On_IWhite='\e[0;107m'   # White
```

Para probar estos colores, simplemente debemos meter la variable entre el texto (los cambios serán permanentes hasta que los volvamos a resetear o cambiar de nuevo):

echo -e "${Blue}Texto azul ${UGreen}Texto verde subrayado${Color_Off} Reset"

Definir la posición

Se hace de una forma parecida a usar colores, es decir, mediante un código:

\033[s → Guarda la posición actual del cursor, útil para luego volver a ella. Si no le indicamos una posición antes, guarda la posición de la siguiente línea a la actual.
\033[<fila>;<columna>f → Sitúa el cursor, para empezar a escribir, en la posición fila (línea), columna de la consola que indiquemos (empiezan en 1 ambas).
\033[u → Restaura la posición guardada anterior para seguir escribiendo.

\033[<N>A → Mueve el cursor arriba N líneas
\033[<N>B → Mueve el curso abajo N líneas
\033[<N>C → Mueve el cursor hacia delante N columnas
\033[<N>D → Mueve el cursor hacia atrás N columnas
\033[2J → Limpia la pantalla
\033[K → Limpia hasta el final de línea

echo -e '\033[2J\033[1;1fBienvenido a: \033[4;7f La consola\033[4C...Del futuro'

1.2 Manejo avanzado de parámetros y opciones

shift → Elimina el primer parámetro $1 y mueve el resto una posición hacia delante.
shift N → Elimina los primeros N parámetros

getopts opstring variable → Devuelve si al script se le han pasado opciones en el formato -a, -b, -c etc. (guión + una letra) opstring es una cadena que contiene todas las opciones, si alguna opción tiene después un argumento, se le pondrá ":" detrás. Ejemplo: **getopts "a:b" variable** → -a valor -b.

- Cada vez que llamemos a **getopts**, pondrá la siguiente opción en la variable. **getopts** devuelve false si no hay más opciones.

```
While getopts "b:s:r" var
do
        case $var in
                b) [[ ${OPTARG} =~ ^[0-9]+$ ]]  || {echo "${OPTARG} no es un
número" >2&; exit 1}
                num="${OPTARG}";;
                s)
                        …..;;
                r)
                        …..;;
                \?) // Si hay algún error por ejemplo que un argumento que requiere
parámetro no lo tenga (el propio comando da un error, así que salimos y punto.
                exit 1;;
        esac
done
```

La variable OPTIND guardará el índice del siguiente parámetro a procesar por getopts (o el que viene después si getopts ha terminado).

- **shift $((OPTIND - 1))** → Para empezar a procesar nosotros manualmente.

Si comenzamos la cadena **optstring** con dos puntos **":a:b:c"** pondremos a **getopts** en modo silencioso (no muestra errores automáticos).

- Con esta opción,sigue asignando el carácter **'?'** a la variable si encuentra una opción no válida, pero además, en **OPTARG** guardará la letra de la opción incorrecta.
- Si la opción es válida pero no contiene un argumento requerido, pondrá el símbolo **':'** en la variable, la opción también en **OPTARG**.

```
While getopts ":b:s:r" var
do
        case $var in
                b) [[ ${OPTARG} =~ ^[0-9]+$ ]]  || {echo "${OPTARG} no es un
número" >2&; exit 1}
                num="${OPTARG}";;
                s)
                        …..;;
                r)
                        …..;;
                :)
                        printf "La opción %s necesita un argumento." "$OPTARG"
```

```
                        exit 1;;

                \?)
                        printf “Opción %s inválida.” “$OPTARG”
                        exit 1;;
        esac
done
```