Manual Bash

RESUMEN

Contenido

1. Introducción

Un script es un archivo de texto en el que escribimos un programa con una sintaxis concreta (la especificada por la Shell Bash en nuestro caso), y con unas características no muy distintas a las que podría tener un programa sencillo escrito en C, es decir, tiene una ejecución secuencial de instrucciones, estructuras de control (if, while, …), llamadas a funciones, operadores, variables, etc.

Algunas características que podemos destacar son:

- Los scripts no se compilan, son "interpretados" por la Shell concreta utilizada.
- Cualquier comando y aplicación instalada en el sistema puede considerarse como una instrucción, puede invocarse desde el script.
- Las palabras reservadas se denominan "builtin comands". Se puede obtener una lista de ellos ejecutando "man builtins". Son palabras que no pueden utilizarse como nombre de variables, funciones, etc.
- Cada sentencia termina con el final de línea (no es necesario el caracter ";" como en otros lenguajes). Sin embargo, es posible ejecutar varias instrucciones escritas en una sola línea separados por ";".

  Si queremos que una instrucción continúe en la línea siguiente escribiremos el carácter "\" justo antes del final de línea
- Para comentarios se usa el carácter "#" al principio de la línea a comentar.

Un script de bash, es un script escrito para ser ejecutado en la Shell Bash, y por lo tanto, siguiendo su sintaxis y características propias.

2. Comandos internos (builtins)

A continuación, se muestran los comandos internos de bash:

```
:, ., [, alias, bg, bind,
break, builtin, case, cd, command, compgen, complete, continue,
declare, dirs, disown, echo, enable, eval, exec, exit, export, fc, fg,
getopts, hash, help, history, if, jobs, kill, let, local, logout, popd,
printf, pushd, pwd, read, readonly, return, set, shift, shopt, source,
suspend, test, times, trap, type, typeset, ulimit, umask, unalias,
unset, until, wait, while.
```

3. Ejecución de scripts

Básicamente, existen 3 formas de ejecutar un script:

1 Ejecutar el comando ***source*** pasándole como parámetro la ruta del script. De esta forma se interpreta el script dentro de la misma shell. Bash tiene un comando interno denominado "**.**" que es equivalente a source (es un alias, una forma de economizar la escritura)

2 Ejecutar el comando **bash** pasándole como parámetro la ruta del script. De esta forma se lanza una nueva shell que interpreta el fichero. Esta nueva shell morirá cuando termine de interpretar el script.

3 Convertir el script en un fichero **ejecutable** y lanzarlo como cualquier otro programa

El fichero que contiene un script puede tener cualquier nombre de fichero válido, aunque suele usarse la extensión "***.sh***".

Supongamos que tenemos un fichero llamado /home/usuario/hola.sh con el siguiente contenido:

```
echo "hola mundo"
```

Para ejecutarlo por el primer método, podemos usar indistintamente:

```
source /home/usuario/hola.sh
. /home/usuario/hola.sh
```

Aunque si nos encontramos ya en el directorio /home/usuario bastaría con ejecutar

```
source ./hola.sh
. ./hola.sh
```

Los ejemplos equivalentes para el segundo método de ejecución serían

```
bash /home/usuario/hola.sh
bash ./hola.sh
```

Para usar el tercer método tenemos que dar primero permisos de ejecución al script:

```
chmod "u+x" /home/usuario/hola.sh
/home/usuario/hola.sh
```

Un aspecto a tener en cuenta es que linux suele incluir varios intérpretes de comandos además de bash (como sh o csh). Para saber el intérprete que estamos usando por defecto podemos escribir

```
echo $SHELL
```

Los sistemas modernos suelen usar por defecto la shell bash. No obstante, se suele indicar, al comienzo de los scripts, la shell que debe usarse para interpretarlo, con un comentario especial que especifica la ruta a dicha shell. Nuestra versión final del script sería:

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

Por último, dado que la Shell no nos permite ejecutar un nuevo script hasta que no finalice la ejecución del actual, recordar que podríamos añadir el carácter "**&**" al final de la línea de ejecución del script, para lanzar otros comandos o aplicaciones concurrentemente.

4 Variables

El comando "***declare***" se usa para declarar variables, aunque realmente bash no exige su utilización. Las siguientes instrucciones serían válidas:

```
variable1="Texto de la variable 1"
declare variable2="Texto de la variable 2"
echo $variable1
echo $variable2
```

De este ejemplo nos quedaremos con dos cuestiones importantes:

- El carácter "***$***" se usa cuando nos referimos al valor de una variable, y por tanto no se usa en la declaración/inicialización, sólo en el acceso a su valor.
- Debemos tener cuidado con los espacios innecesarios en bash. Si escribimos de esta forma la declaración de la variable nos daría un error:

```
declare variable2 = "Texto de la variable 2"
```

Una de las utilidades de usar declare es que podemos especificar el tipo mediante argumentos (por ejemplo ***-i*** para enteros) ya que por defecto las variables se consideran de tipo "cadena de caracteres". Por ejemplo, el siguiente script:

```
variable3=45
declare -i variable4=45
variable3=$variable3+10
variable4=$variable4+10
echo $variable3
echo $variable4
```

mostraría el siguiente resultado, ya que en el primer caso estaríamos concatenando las cadenas de caracteres "45", "+" y "10", mientras que en el segundo estamos usando el operador + sobre los enteros 45 y 10:

```
45+10
55
```

Las variables que declaramos dentro de un script no serán accesibles cuando éste termine.

Exportar una variable:

Mediante el comando ***export*** conseguimos que la variable se exporte al ámbito de la Shell (se exportan las variables para que sean accesibles por los subprocesos que se creen). Para realizar la acción contraria usaremos el comando ***unset***, que deshace la declaración de una variable.

El problema de los espacios en blanco:

Si asignamos una cadena de texto como valor de una variable, deberíamos siempre entrecomillar la cadena si ésta tiene espacios en blanco (o bien escapar cada espacio en blanco con el carácter “\”), ya que dichos espacios podrían ser interpretados como la separación entre un comando y sus argumentos.

5 Entrecomillado de caracteres

Bash distingue 2 tipos de entrecomillado para cadenas de caracteres: ***fuerte y débil***. Su sintaxis se distingue en que en el fuerte se usan comillas simples, y en el débil comillas dobles.

A nivel funcional la gran diferencia está en que dentro de unas comillas fuertes bash no interpreta el contenido, lo considera tal cual lo escribimos, mientras que en el entrecomillado débil se interpretan caracteres como '***$***' para extraer el valor de las variables. Así, el siguiente ejemplo

```
echo "la variable 4 vale $variable4" #entrecomillado débil
echo 'la variable 4 vale $variable4' #entrecomillado fuerte
```

da como resultado

```
la variable 4 vale 55
la variable 4 vale $variable4
```

Por último, es interesante conocer que también se puede acceder al contenido de una variable mediante la sintaxis ***${<variable>}*** que tiene sentido cuando, dentro un entrecomillado débil, usamos la variable sin que esté seguida de un espacio. Veámoslo con un ejemplo

```
echo "$variable4__"
echo "${variable4}__"
```

La primera línea intenta mostrar el contenido de una variable llamada "***variable4__***", que será una cadena vacía, ya que la variable no está declarada (no la encuentra) y escribe en su lugar una cadena vacía. La segunda sí mostrará el resultado esperado.

Existe un tercer entrecomillado (invertido) para ejecución de comandos, que veremos en apartados posteriores.

6 Variables de Entorno

Las variables de entorno son un conjunto de variables que establecen la configuración del entorno en el que vamos a trabajar. Cuando abrimos una terminal se crea una nueva Shell en ese momento, con un entorno de trabajo cuya configuración viene especificada por las variables de entorno. Si modificamos las variables de entorno modificamos la configuración del entorno para esa Shell, pero sin afectar al resto de shells que puedan estar abiertas en ese momento, ni a las nuevas shells que se inicien (esto último no es del todo cierto, ¿por qué? … piénsalo).

Del mismo modo, cuando se ejecuta un script se crea una nueva Shell para ese script, con su copia de las variables de entorno. Cuando el script finaliza la Shell se destruye y las modificaciones hechas en las variables de entorno desde el script no afectarán a otros scripts.

Algunas de las variables de entorno más importantes son:

Variable	Descripción
SHELL	Ruta a la Shell que está utilizando en ese momento.
HOME	Ruta al directorio personal del usuario.
PWD	Directorio actual.
OLDPWD	Anterior directorio actual.
USER	Nombre de la cuenta de usuario.
PATH	Rutas (separadas por el carácter ":") en las cuales el sistema buscará un ejecutable cuando el usuario no especifica la ruta.
PS1	Prompt por defecto.
LANG	Idioma del sistema por defecto

Algunas de las variables son de sólo lectura, sólo el sistema puede modificarlas, como por ejemplo HOME, PWD o USER.

7. Variables de la SHELL

Además de las variables que definimos en nuestros scripts existen algunas predefinidas que van a sernos útiles. En la siguiente tabla se muestran las más importantes:

VARIABLE	DESCRIPCIÓN
$#	Número de parámetros que recibe el script
$0	Nombre del ejecutable. Cuidado porque con el primer método de ejecución devolvería source
$<n>	Valor del enésimo parámetro
"$@"	Cadena de parámetros como una sola cadena
IFS	**Internal Field Separator**. Es una variable especial que usa la shell amodo de separador. El valor por defecto es <espacio><tabulador><fin de línea> Ejemplo: IFS='|' echo "$*"
"$*"	Cadena de parámetros separados por IFS.
$?	Exit code del último comando ejecutado.

8 Usando comandos externos

Cuando ejecutamos un programa nos puede interesar tanto su salida por pantalla como el resultado de su ejecución (exit code). A continuación, se muestra un ejemplo de ambas situaciones:

```
salida1=$(sudo find -un_parametro_incorrecto)
retorno1=$?
salida2=$(sudo find /etc -name passwd)
retorno2=$?
echo "El resultado de ejecutar el primer comando es $retorno1"
echo "La salida por pantalla del primer comando es:"
echo  "$salida1"
echo "El resultado de ejecutar el segundo comando es $retorno2"
echo "La salida por pantalla del segundo comando es:"
echo "$salida2"
```

En este bloque estamos ejecutando el comando find con diferentes parámetros. En el primer caso fallará por pasarle un parámetro incorrecto, en el segundo encontrará el fichero /etc/passwd.
El resultado de la ejecución debe ser el siguiente:

```
El resultado de ejecutar el primer comando es 1
La salida por pantalla del primer comando es:

El resultado de ejecutar el segundo comando es 0
La salida por pantalla del segundo comando es:
/etc/passwd
/etc/cron.daily/passwd
/etc/pam.d/passwd
```

En ocasiones nos puede interesar tener en una variable la sentencia que queremos ejecutar. A continuación, se muestra un ejemplo

```
usuario='pablo'
fichero='/etc/passwd'
sentencia="grep $usuario $fichero"
salida=$($sentencia)
retorno=$?
echo $retorno
echo "$salida"
```

Por último, comentar qué si dentro de una sentencia necesitamos escribir comillas, tendríamos que utilizar códigos de escape (como **\"**) y el script puede resultar ilegible. Una solución es usar las ***comillas invertidas*** (`) que también evalúan un comando.

```
salida=`grep " " fichero.txt`
echo "$salida"
```

9. CONTROL DE FLUJO

9.1 Sentencias condicionales: if, then, elif, else

La sintaxis general de la sentencia if es la siguiente:

```
if condición ; then sentencias
elif condición; then sentencias
else
sentencias
fi
```

En bash nos encontramos con 2 tipos evaluaciones en un if:

- Evaluación de un código de terminación (devuelto tras la ejecución de un comando).
- Evaluación lógica de algún tipo de comparación (comparar cadenas, números o los atributos de ficheros).

9.1.1 Por código de terminación

Se utilizan cuando queremos realizar acciones en función del resultado de la ejecución de un comando. **El código de terminación de un comando es 0 cuando todo funciona correctamente** y un valor distinto para indicar una condición de error.

Veamos un ejemplo:

```
if cd mi_directorio 2>/dev/null
then
     echo "Ok"
else
     echo "Error"
fi
```

El comando que estamos ejecutando es "***cd mi_directorio***" que retornará ***0*** si existe el directorio y tenemos permisos para acceder a él, y ***1*** en caso contrario.

La redirección "***2>/dev/null***" se usa para que no se muestren por pantalla los mensajes de error del comando. Recuerda, esto lo vimos en el tema 4, en el apartado redirecciones.

El ejemplo mostrará por pantalla "***OK***" si consigue ir al directorio y "***ERROR***" en caso contrario.

9.1.2 Por evaluación lógica

La sintaxis de la condición es:

```
[ condición ]
```

Hay que prestar especial atención a los espacios después de "[" y antes de "]".

Para las condiciones se utilizan parámetros a modo de operadores. Por ejemplo, el parámetro “-gt” (greater tan) equivale a “>”.

Ejemplo:

```
var1=8
if [ $var1 -gt 7 ]; then echo “Mayor”; fi
```

Tabla de operadores de comparaciones numéricas:

Operador	Descripción
`-lt`	Less tan
`-le`	Less tan or equal
`-eq`	Equal
`-ge`	Greater than or equal
`-gt`	Greater tan
`-ne`	Not equal

Tabla de operadores de comparación de cadenas:

Operador	Verdadero si …
`str1 = str2`	Las cadenas son iguales
`str1 != str2`	Las cadenas son distintas
`str1 < str2`	str1 es menor lexicográficamente que str2
`str1 > str2`	str1 es mayor lexicográficamente que str2
`-n str1`	str1 es no nula y tiene longitud mayor a cero
`-z str1`	str1 es nula (tiene longitud cero)

Tabla de operadores lógicos:

Podemos combinar varias evaluaciones en un mismo if utilizando los operadores lógicos AND, OR y NOT

Operador	Descripción
`&&`	AND lógico
`||`	OR lógico
¡	NOT lógico.

Tabla de operadores para comprobar atributos de ficheros:

[-d FILE]	FILE existe y es un directorio
[-e FILE]	FILE existe
[-f FILE]	FILE existe y es un fichero regular
[-h FILE]	FILE existe y es un enlace simbólico.
[-r FILE]	FILE existe y podemos leerlo (tenemos permiso de lectura).
[-s FILE]	FILE existe y no está vacío.
[-w FILE]	FILE existe y tenemos permisos de escritura.
[-x FILE]	FILE existe y tenemos permiso de ejecución, o de acceso si es un directorio.
[FILE1 -nt FILE2]	Cierto si FILE1 ha sido modificado más recientemente que FILE2, o si FILE1 existe y FILE2 no.
[FILE1 -ot FILE2]	Cierto si FILE1 es más antiguo que FILE2, or si FILE1 existe y FILE2 no.
[FILE1 -ef FILE2]	Cierto si FILE1 y FILE2 hacen referencia al mismo dispositivo, o ficheros con el mismo número de inodo.

9.2 Sentencias Case

Mediante esta sentencia podemos realizar unas acciones específicas en función del valor de una expresión.

```
case expression in
    pattern1 )
        statements ;;
    pattern2 )
        statements ;;
    * )
        statements ;;
    ...
esac
```

A continuación, se muestra un ejemplo

```
echo "escriba una opción"
read opcion
case "$opcion" in
        "1") echo "ha escrito un 1";;
        "2") echo "ha escrito un 2";;
        *) echo "no se lo que significa $opcion";;
esac
```

9.3 Bucles

Los bucles en bash se implementan mediante los comandos ***for***, ***while*** y ***until***. El comando ***for*** tiene algunas variantes que permiten recorrer los parámetros, listas de palabras, ficheros de un directorio, etc. Veamos algunos ejemplos:

9.3.1 Bucle for

El bucle `for` en Bash es un poco distinto a los bucles for tradicionales de otros lenguajes como C o Java. Se parece más al bucle `for each` de otros lenguajes como php, ya que su sintaxis por defecto es la siguiente:

```
for var [in lista]
do
   …
   Sentencias que usan $var
   …
done
```

Si se omite in lista, se recorre el contenido de $@, pero, aunque vayamos a recorrer esta variable, conviene indicarlo explícitamente por claridad.

Además, para recorrer los argumentos del script, lo correcto es usar "$@" entrecomillado, ya que, sin entrecomillar, se pueden interpretar mal aquellos argumentos que tengan espacios en blanco.

Ejemplos:

- Recorre los parámetros de un script

```
for i; do echo $i;done
```

- El mismo ejemplo, usando la variable $@ explícitamente:

```
for i in "$@"
do
    echo $i
done
```

- Recorre una lista de palabras:

```
palabras="uno dos tres cuatro"
for i in $palabras; do echo $i; done
```

- Recorre la salida de un comando

```
for i in $(ls); do
   if [ -d $i ]; then
       echo "$i es un directorio"
   else
       echo "$i es un fichero"
   fi
done
```

- Sin embargo, también existe el bucle for al estilo de C

```
for ((i=0; i<4; i++)); do echo $i; done
```

9.3.2 Bucle while

La sintaxis es:

```
while comando; do; sentencias; done
```

Ejemplo:

```
declare -i i=0
while [ $i -lt 4 ]; do echo $i; i=i+1; done
```

9.3.3 Bucle until

La sintaxis es:

```
until comando; do; sentencias; done
```

Ejemplo:

```
declare -i i=0
until [ $i -gt 10 ]; do echo $i; i=$((i+1; done
```

10. Funciones

Para definir una función existen 2 formatos:

El estilo del Bourne Shell:

```
function nombreFuncion {
    …
    comandos bash
    ….
}
```

O el estilo del C Shell:

```
nombreFuncion() {
    …
    comandos bash
    ….
}
```

No existe diferencias entre ambos formatos, y usaremos ambos indistintamente.

Para ejecutar la función simplemente escribimos su nombre seguido de sus argumentos, como cuando ejecutamos un comando. Los argumentos actúan como parámetros de la función.

Por ejemplo, supongamos el siguiente script llamado “saluda”

```
# Script que llama a una función para saludar
function DiHola {
  echo “Hola $1”
}
DiHola
```

Intenta ejecutar el anterior script, por ejemplo, con la siguiente orden:

bash saluda Fernando

¿Qué crees que imprimirá en pantalla?

Si lo compruebas verás que ha imprimido “Hola” simplemente. La razón está en que los parámetros posicionales ($1, $2, …) son locales al script, y si queremos que lo reciba la función tendremos que pasárselo explícitamente, como se muestra a continuación:

```
# Script que llama a una función para saludar
function DiHola {
  echo “Hola $1”
}
DiHola $1
```

A diferencia de los parámetros posicionales, el resto de variables que definimos, con la sintaxis vista hasta ahora, en un script o función son globales, es decir, una vez definidas en el script son accesibles (y modificables) desde cualquier función. Veamos un ejemplo:

Supongamos el script “dondeestoy” siguiente:

```
# Ejemplo de variable global
function EstasAqui {
  donde=”Dentro de la función”
}

donde=”En el script”
echo $donde
EstasAqui
echo $donde
```

Si ejecutamos el script, el resultado obtenido será:

```
En el script
Dentro de la función
```

Variables locales a una función:

Si deseamos que una variable no posicional sea local debemos declararla con el modificador `local`, el cual sólo se puede usar dentro de las funciones.

Por ejemplo, modifiquemos el anterior script declarando local la variable de la función:

```
# Ejemplo de variable global
function EstasAqui {
  local donde=”Dentro de la función”
}

donde=”En el script”
echo $donde
EstasAqui
echo $donde
```

Si ejecutamos el script, el resultado obtenido será:

```
En el script
En el script
```

Por último, hay que comentar que **las variables globales también pueden ser definidas dentro de una función**, y ser utilizada después en cualquier otra parte del script. Lo único que hay que tener en cuenta es que, para que sea global, no debe ser declarada con el modificador `local`.

Cómo ver todas las funciones definidas en una sesión:

Podríamos ver qué funciones tenemos definidas en una sesión usando el comando:

```
declare -f
```

El Shell imprime las funciones y su definición ordenadas alfabéticamente. Si preferimos obtener sólo el nombre de las funciones podemos usar `declare -F`

11. Arrays

Bash introdujo los arrays clásicos (indexados con números enteros) en la versión 2, y en la versión 4 introduce los arrays asociativos (los índices pueden ser cadenas de texto).

Actualmente solo soporta arrays unidimensionales, aunque existen trucos para usar arrays de más de una dimensión.

11.1 Arrays indexados con números enteros:

Para declarar un array usaremos el comando “declare” con el parámetro “-a”. Por ejemplo, para declarar una variable de tipo array llamada “lista”, escribiremos la siguiente línea:

```
declare -a lista
```

En este caso la variable “lista” sería un array vacío, sin elementos. Podríamos ver como está declarada internamente, ejecutando el comando:

```
declare -p lista
```

También podríamos crear una variable de tipo array, asignándole directamente valores, sin necesidad de usar el comando “declare”, por ejemplo:

```
otraLista=(Carne 4 Fruta)
```

Visualicemos como ha quedado ahora internamente, ejecuta:

```
declare -p otraLista
```

Como habrás comprobado, aunque no lo hayamos indicado, ha asignado el índice 0 al primer elemento, el índice 1 al segundo y el índice 2 al tercer elemento.

Por lo tanto, podemos deducir que **los elementos en los arrays Bash se indexan empezando por el número 0**.

No obstante, podemos cambiar los índices de los elementos, indicándolos explícitamente, por ejemplo:

```
lista3=([5]=manzana [0]=arroz [3]=400)
```

Comprobemos como queda almacenado internamente:

```
declare -p lista3
```

Obsérvese que no hace falta suministra los elementos en orden, ni suministrarlos todos. Los índices donde no coloquemos un valor simplemente valdrán la cadena nula.

Si sólo indicamos algunos índices, los demás los asigna contando desde el último asignado, así, por ejemplo:

```
lista4=([5]=cosa casa perro)
declare -p lista4
```

Una forma muy utilizada de usar arrays, es rellenando su valor con la salida producida por la ejecución de un comando, por ejemplo

```
lista5=($(ls))
```

Veamos su contenido:

```
declare -p l ista5
```

Uso del operador `[]`

¿Cómo accedemos a los elementos?

Para acceder a los elementos de un array usamos el operador corchete `[]`, indicando el índice del elemento a acceder, y en este caso es obligatorio encerrar entre llaves la variable.

Por ejemplo, para acceder al elemento de índice "2" del array "lista5" escribiríamos "${lista5[2]}". Veamos como mostrar su valor con "echo":

```
echo ${lista5[2]}
```

Si no indicamos índice, por defecto tomaría el elemento de índice 0:

```
echo $lista5
```

Veamos ahora un ejemplo, usando el array "lista5", que ilustra la razón por la cual hay que encerrar la variable entre llaves:

```
echo $lista5[3]
```

Como habrás comprobado, la instrucción muestra en pantalla el primer elemento del array (el de índice 0), concatenando la cadena "[3]" a continuación.

Inicializar un array con el operador []

Otra forma de inicializar un array es introduciendo directamente valores en las posiciones que deseemos, por ejemplo:

```
lista6[2]=Europa
lista6[0]=Asia
declare -p lista6
```

Uso de los índices especiales * y @

Podemos usar los índices especiales * y @, los cuales retornan todos los elementos del array de la misma forma que lo hacen los parámetros posiciónales: Cuando no están encerrados entre comillas débiles ambos devuelven una cadena con los elementos separados por espacio, pero cuando se encierran entre comillas débiles @ devuelve una cadena con los elementos separados por espacio, y * devuelve una cadena con los elementos separados por el valor de IFS.

Veamos un ejemplo:

```
IFS=”,”
lista7=([5]=cosa [7]=casa perro)
echo ${lista7[*]}
echo ${lista7[@]}
echo “${lista7[@]}”
echo “${lista7[*]}”
```

En los 3 primeros echo del ejemplo anterior, el valor escrito en pantalla será:

```
cosa casa perro
```

En el último echo, en cambio, el valor escrito será:

```
cosa,casa,perro
```

Iterar sobre un array con un bucle for

Al igual que en los parámetros posicionales, podemos iterar sobre un array con un bucle for. Por ejemplo, con el array lista7:

```
for e in ${lista7[*]}
do
   echo $e
done
```

El bucle anterior sólo accedería a los elementos del array que tienen valor, e imprimiría:

Conocer la longitud de un array

Para conocer la longitud de un array podemos usar la forma `${#nombreArray[@]}`, es decir, poniendo el carácter “#” antes del nombre y utilizando el índice @ o *.

Por ejemplo, con el array lista7:

```
echo ${#lista7[@]}
```

La instrucción anterior devolvería el valor 3. Aunque el índice llegue hasta el 8, sólo nos devuelve el número de posiciones ocupadas.

Conocer la longitud del elemento ocupado por una posición del array

Para conocer la longitud de cualquier variable podemos usar la expresión `${#nombreVariable}`, y del mismo modo para conocer la longitud de la cadena almacenada en cualquier posición del array. Por ejemplo, la siguiente sentencia devolverá 4 (longitud de la cadena almacenada en la posición 5, “cosa”):

```
echo ${#lista7[5]}
```

Conocer los índices de los elementos no nulos

Para conocer los índices de los elementos de un array que no son nulos podemos usar la forma `${!nombreArray[@]}`, es decir, poniendo el carácter "!" antes del nombre y utilizando el índice "@", o bien "*".

Por ejemplo, con el array lista7:

```
echo ${!lista7[@]}
```

La sentencia anterior imprimirá en pantalla:

```
5 7 8
```

Sobrescribiendo un array

Si asignamos una variable de tipo array a otra variable de tipo array, los valores se pierden y se asignan los nuevos. Por ejemplo:

```
lista7=(hola "qué tal" adios)
declare -p lista7
```

Tras la asignación anterior, se habrán perdido los elementos 5, 7 y 8 que tenía el array asignados.

Eliminar un elemento de un array

Para eliminar un elemento de un array utilizaremos el comando unset sobre el índice del elemento a eliminar, por ejemplo:

```
unset lista7[2]
declare -p lista7
```

Eliminar un array

Al igual que cualquier otra variable, podríamos eliminar un array usando el comando unset sobre el nombre de la variable de tipo array. Por ejemplo, con el array lista7:

```
unset lista7
declare -p lista7
```

11.2 Arrays asociativos:

Para declarar un array asociativo usaremos el comando "declare" con el parámetro "-A". Por ejemplo, si queremos tener en un array la asociación entre ips y nombres de hosts haríamos:

```
declare -A hosts
hosts[192.168.1.1]=host1
hosts[192.168.1.2]=host2
```

12. Operaciones aritméticas

Hay distintas formas de realizar operaciones aritméticas en bash. Las más utilizadas se basan en el uso del doble paréntesis, o bien en el uso del comando let.

El comando let, como veremos en los ejemplos, convierte una cadena de texto en una expresión aritmética.

Veamos los ejemplos. Para ello declaramos “var” como variable de tipo entero, con un valor inicial de 1.

```
declare -i var=1
```

- **Suma:**

  Utilizando doble paréntesis, las dos líneas siguientes son equivalentes:

  ```
  var=$((var+5))
  ((var=var+5))
  ```

  Utilizando let:

  ```
  let “var=var+5”
  ```

- **Incremento:**

  Utilizando doble paréntesis, las dos sentencias siguientes son equivalentes:

  ```
  ((var+=1))
  ((var++))
  ```

  Utilizando let, las dos sentencias siguientes son equivalentes:

  ```
  let “var+=1”
  let “var++”
  ```

- **Resta:**

  Utilizando doble paréntesis, las dos líneas siguientes son equivalentes:

  ```
  var=$((var-5))
  ((var=var-5))
  ```

  Utilizando let:

  ```
  let “var=var-5”
  ```

- **Decremento:**

  Utilizando doble paréntesis, las dos sentencias siguientes son equivalentes:

  ```
  ((var-=1))
  ((var--))
  ```

  Utilizando let, las dos sentencias siguientes son equivalentes:

  ```
  let “var-=1”
  let “var--”
  ```

- **Multiplicación:**

  Utilizando doble paréntesis:

  ```
  var=$((var*10))
  ```

```
((var=var*2))
```

Utilizando let:

```
let "var=var*20"
```

- **División:**

  Utilizando doble paréntesis, las dos sentencias siguientes son equivalentes:

  ```
  var=$((var/2))
  ((var=var/2))
  ```

  Utilizando let, las dos sentencias siguientes son equivalentes:

  ```
  let "var=var/2"
  ```

- **Módulo (retorna el resto de una división entera "/"):**

  Utilizando doble paréntesis:

  ```
  var=$((var%35))
  ((var=var%8))
  ```

  Utilizando let, las dos sentencias siguientes son equivalentes:

  ```
  let "var=var%5"
  ```

- **Potencia (eleva un número al otro:**

  Utilizando doble paréntesis:

  ```
  Var=$((var**2))     # Elevado al cuadrado
  ((var=var**5))      # Elevado a la quinta
  ```

  Utilizando let, las dos sentencias siguientes son equivalentes:

  ```
  let "var=var**2"    # Elevado al cuadrado
  ```

13. Analizar los argumentos de un script con “getopts”

El comando “getopts” nos permite analizar y extraer los parámetros recibidos por un script durante su ejecución.

Para poder utilizar “getopts”, los parámetros deben ser un conjunto de una o más opciones precedidas por un guion, con las siguientes características:

- Cada opción es un único carácter, no pudiendo usarse los caracteres “`:`” ni “`?`”.
  En el siguiente ejemplo, el script “myscript” recibe como parámetro la opción “`-n`”

  ```
  myscript -n
  ```

- El script puede recibir como parámetros varias opciones, de la siguiente forma:

  ```
  myscript -d -f -g
  ```

  … pudiendo además usar la forma siguiente:

  ```
  myscript -dfg
  ```

- Las opciones pueden tener argumentos.

  Por ejemplo, el script “creaUsuario” crea una nueva cuenta de usuario en el sistema con las siguientes posibles opciones: la opción “`-n`” seguida del nombre de la nueva cuenta, la opción “`-p`” seguida de la contraseña, y la opción “`-w`” sin argumentos para indicar que la contraseña no caduca nunca:

  ```
  creaUsuario -n luisa -p dks23%R6 -w
  ```

Sintaxis

El comando “getopts” recibe dos argumentos obligatorios. Su sintaxis es la siguiente:

```
getopts cadenaDeOpciones nombreDeVariable
```

Funcionamiento

El comando “getopts” recibe como primer argumento una cadena de texto que representa cuales son las opciones permitidas y como hay que procesarlas. Como segundo argumento recibe el nombre de la variable que usaremos para saber en cada momento cual es la opción que está analizando. Veamos una explicación más detallada.

Explicación de los argumentos:

- `cadenaDeOpciones` (el primer argumento de getopts)

  Se trata de una cadena de texto que contendrá los caracteres que representan opciones válidas para el script. Por ejemplo, “`ab`” expresaría que las posibles opciones son `-a` y `-b`. Cualquier otra opción se considerará inválida.

Este argumento primer argumento de getopts no representa la obligatoriedad de las opciones, sólo su validez.

Opciones con argumentos:

Si deseamos especificar que una opción debe ir acompañada de un argumento, escribiremos el carácter "`:`" después de la letra de la opción. Por ejemplo, "`ab:`" indica que las opciones válidas son `-a` y `-b`, y que a la opción `-b` le sigue un argumento.

Silenciar los mensajes de error:

Por defecto, "getopts" muestra un mensaje de error cuando el script recibe como argumento una opción no válida. Así mismo, cuando una opción que espera recibir un argumento, no lo recibe, también generará un mensaje.

Si deseamos silencia los mensajes de getopts y tratar nosotros en el script dichas situaciones, deberemos incluir el carácter "`:`" al principio de la cadena de opciones. Por ejemplo, ":ab:" indicaría que las opciones válidas son "-a" y "-b", que "-b" espera recibir un argumento, y que getopts no va a generar ningún mensaje de error.

Ejemplos:

- `m` Chequea la opción `-m` sin argumentos, dando error las opciones no soportadas.
- `m:` Chequea la opción `-m` con argumentos, error las opciones no soportadas.
- `xyz` Chequea `-x`, `-y`, `-z`, sin argumentos, dando error las opciones no soportadas.
- `:xyz` Chequea `-x`, `-y`, `-z`, sin argumentos, silenciando los errores.
- `:xy:z` Chequea las opciones `-x`, `-y`, `-z`, esperando recibir un argumento únicamente la opción `-y`, silenciando los errores.

- `nombreDeVariable` (el segundo argumento de getopts)

Cada vez que se invoca al comando "getopts" dentro del script, éste procesa una opción (sólo una). La primera vez que se invoque procesará la 1ª opción que encuentre (de izquierda a derecha), la segunda vez procesará la opción situada en 2º lugar, etc.

En cada invocación, getopts almacenará la opción procesada en la variable, cuyo nombre pasamos como segundo parámetro. La variable almacenará el valor de la opción sin el guion.

Esta forma de actuar de getopts es especialmente útil para ser usado en un bucle while, utilizando su exit code como condición de terminación. Mientras getopts encuentre opciones devolverá el código de terminación 0 (aunque la opción encontrada no sea válida). Cuando no encuentre más opciones devolverá el código de terminación 1, facilitándonos la salida del bucle.

Si la opción va acompañada de un argumento, el valor de dicho argumento quedará almacenado en la variable de entorno OPTARG.

Ejemplo:

En el siguiente ejemplo, silenciamos los mensajes de error de getopts, y las posibles opciones son “-h”, “-d” y “-p”. La opción -h muestra la ayuda del script. Las opciones -d y -p esperan recibir un argumento. El argumento de -d es la ruta a un directorio que verificaremos que existe.

Dentro del bucle accederemos a la opción en curso utilizando la variable “opcionActual”.

Utilizamos una función llamada “ayuda” para mostrar un mensaje de uso si fuera necesario.

```
Function ayuda {
  cat<< BLOQUETXT
 Uso correcto del script:
  $0 [-h] [-d directorio] [-p unArgumento]
BLOQUETXT
}

while getopts ":hd:p:" opcionActual; do
  case $ opcionActual in
      p) echo "El argumento de p es ${OPTARG}"
          ;;
      s) # Verifico el argumento
           if [ -z $OPTARG ]; then
             ayuda
           elif [ -d $OPTARG ]; then
             echo “El directorio $OPTARG existe”
           else
             echo “El directorio $OPTARG no existe”
           fi
           ;;
      h | *) # Imprimo la ayuda
           ayuda
           ;;
  esac
done

exit 0
```