Nuestro primer script. 2

Uso de Variables en los Scripts. Expansiones. 3
$(orden).....3
$((operación aritmética)) 4
let. bc. 5

Funciones. 6

Estructuras Condicionales. 8
If. 8
If..else 10
if..elif..else 10
Case. 14

Estructuras iterativas. (Bucles). 16
For. 16
while y until 20
select 22

Paso de parámetros a un script. 25
Valores devueltos por las órdenes. 26
Un script completo de ejemplo 28

Ejercicios sobre scripts. 30

NUESTRO PRIMER SCRIPT.

Los scripts no son más que ficheros de texto ASCII puro, que pueden ser creados con cualquier editor del que dispongamos (vi, nano, gedit, emacs, etc.). Cread un fichero de texto de nombre primero.sh, con el siguiente contenido:

```
#!/bin/bash
echo "Hola Mundo"
```

La primera línea sirve para indicar que shell utilizamos (en nuestro caso bash) y donde puede ser encontrado en nuestro sistema (para saberlo, podemos hacer *locate bash*). Esta línea debe ser la primera de todos los scripts que realicemos.
La segunda línea de nuestro script, simplemente utiliza el comando para escribir en pantalla (echo) y escribe la línea Hola Mundo.

Una vez creado el fichero, debemos darle permisos de ejecución, mediante el comando

```
chmod a+x primero.sh
```

Posteriormente para ejecutarlo debemos llamarlo como *./permiso.sh* (el punto barra es para indicarle que lo busque en el directorio actual, ya que dicho directorio no estará seguramente incluido en el PATH del sistema).

Si queremos ejecutar un script para comprobar cómo funciona sin hacerlo ejecutable, podemos hacerlo mediante el comando *source primero.sh* que permite lanzar un script no ejecutable. La orden source también puede ejecutarse simplemente escribiendo un punto como hemos visto en el tema anterior.

Sin embargo, la orden source solo debe usarse para comprobar el script, una vez que tengamos el script completo y depurado debemos darle sus permisos de ejecución correspondientes.

Las comillas dobles que hemos usado para escribir Hola Mundo no son necesarias, y podéis comprobar como quitándolas el proceso se ejecuta exactamente igual. Sin embargo, es una buena práctica encerrar siempre los textos entre comillas dobles, y en caso de que contengan caracteres especiales (como el *, el $, etc.), es mejor usar comillas simples, que son más potentes que las comillas dobles. Probad lo siguiente:

```
echo esto es un asterisco * sin comillas
echo esto es un dólar y tres letras $ABC sin comillas
echo "esto es un asterisco * entre comillas dobles"
echo 'esto es un asterisco * entre comillas simples'
echo "esto es un dólar y tres letras $ABC entre comillas dobles"
echo 'esto es un dólar y tres letras $ABC entre comillas simples'
```

Si tenemos que ejecutar varias líneas y queremos escribirlas en una sola, podemos hacerlo usando el símbolo punto y coma para indicar que lo siguiente es otra línea, aunque este en la misma:

```
echo Hola ; pwd ; echo Adios    # esto son tres líneas en una sola.
```

También podemos hacer lo contrario, escribir una sola línea en varias. Para ello usamos el carácter contrabarra cuando queramos que nuestra línea se "rompa" y continué en la línea de abajo.

```
echo esto \
es una sola línea \
aunque ocupe 3 en pantalla. # esto es una linea escrita en tres.
```

En estos últimos ejemplos he aprovechado para mostraros como se pueden usar comentarios en los scripts. Basta con usar el símbolo almohadilla (#) donde queramos, todo lo que quede a la derecha de dicho símbolo es un comentario. Si usamos # como primer carácter de una línea, toda la línea es de comentario.

USO DE VARIABLES EN LOS SCRIPTS. EXPANSIONES.

Las variables de los shell scripts son muy simples, ya que no tienen tipo definido ni necesitan ser declaradas antes de poder ser usadas. Para introducir valor en una variable simplemente se usa su nombre, y para obtener el valor de una variable se le antepone un símbolo dólar.

```
#!/bin/bash
DECIR="Hola Mundo"
echo $DECIR
```

Este script realiza exactamente la misma función que el anterior, pero usando una variable.

Cualquier valor introducido en una variable se considera alfanumérico, así que si realizamos lo siguiente:

```
NUMERO=4  # No se debe dejar ningún espacio en la asignación.
echo NUMERO+3
```

Obtendremos por pantalla la cadena de caracteres 4+3.

$(ORDEN)

En Linux podemos usar varias expansiones en las líneas de comandos, que son especialmente útiles en los scripts. La primera expansión consiste en usar `$( )`. Esta expansión permite ejecutar lo que se encuentre entre los paréntesis, y devuelve su salida.

```
echo pwd        # escribe por pantalla la palabra pwd
echo $(pwd)     # ejecuta la orden pwd, y escribe por pantalla su
                resultado.
```

Así, por ejemplo, la siguiente instrucción copia el fichero /etc/network/interfaces en el directorio actual con el nombre red290411.conf (suponiendo que estamos en la fecha 29 de Abril de 2011).

```
NOMBRE_FICHERO="red"$(date +%d%m%y)".conf"
cp /etc/network/interfaces $NOMBRE_FICHERO
```

Como es lógico, es perfectamente posible no usar variables en el ejemplo anterior y hacerlo todo en una línea, pero es una buena práctica no complicar excesivamente cada una de las líneas del script.
Esto nos permitirá una modificación mucho más simple y la depuración en caso de que existan errores suele ser bastante más rápida, al menos mientras nuestro nivel de programación no sea bastante alto.

El efecto conseguido con *$(orden)* se puede conseguir también usando la tilde invertida `orden`.

```
usuario@debian6:~$ echo "Tenemos $(find /home/usuario -iname "*.sh" |
wc -l) scripts es nuestro directorio de usuario"
Tenemos 10 scripts es nuestro directorio de usuario
usuario@debian6:~$
```

```
usuario@debian6:~$ echo "Tenemos `find /home/usuario -iname "*.sh" | w
c -l` scripts es nuestro directorio de usuario"
Tenemos 10 scripts es nuestro directorio de usuario
usuario@debian6:~$
```

$((OPERACIÓN ARITMÉTICA))

Otra expansión que podemos usar es $(( )) (símbolo dólar pero con dos paréntesis). Los dobles paréntesis podemos sustituirlos si queremos por corchetes. $[ ].

Esta expansión va a tratar como una expresión aritmética lo que esté incluido entre los paréntesis, va a evaluarla y devolvernos su valor.

```
NUMERO=4
echo $(($NUMERO+3)) # sería lo mismo poner echo $[$NUMERO+3]
```

Obtenemos en pantalla el valor 7.

Aprovecho para explicar el comando *let*, que nos permite realizar operaciones aritméticas como la anterior, pero sin tener que usar expansiones ni dólares para las variables.

```
NUMERO=4
let SUMA=NUMERO+3
echo $SUMA
```

Obtenemos el mismo valor 7, y como vemos no hemos usado ni dólar, ni paréntesis.

Los operadores aritméticos que podemos usar para realizar operaciones son: Resta (-), Suma (+), División (/), Multiplicación (*) y Modulo o Resto (%).

```
usuario@debian6:~$ PRECIO_COPAS=60
usuario@debian6:~$ BEBEDORES=10
usuario@debian6:~$ ESCOTE=$[ $PRECIO_COPAS / $BEBEDORES ]
usuario@debian6:~$ echo Tenemos que pagar $ESCOTE euros cada uno.
Tenemos que pagar 6 euros cada uno.
usuario@debian6:~$
```

LET. BC.

Cread con nano un fichero con nombre media.sh con el siguiente contenido:

```
usuario@debian6:~$ cat media.sh
read -p "Introduce la nota de ISO : " NOTAISO
read -p "Introduce la nota de PAR : " NOTAPAR
read -p "Introduce la nota de BD  : " NOTABD
SUMA=$[ $NOTAISO + $NOTAPAR + $NOTABD ]
MEDIA=$[ $SUMA / 3 ]
echo "La media es $MEDIA"
```

Haced ejecutable media.sh y ejecutadlo. Si lo habéis escrito bien veréis como realmente os da la media aritmética de las tres notas introducidas.

La línea

MEDIA=$[$SUMA / 3]

Podría haberse escrito también como

let MEDIA=SUMA/3

Si ejecutamos el script, veremos que hay un problema, podéis comprobar como bash no trabaja con decimales, de modo que si introducimos por ejemplo 10, 10 y 6 nos dirá que la media es 8, mientras que la media realmente es 8,66.

Podemos obligar a que bash trabaje con decimales utilizando un comando que sirve como calculadora en Linux, este comando es *bc*. Este comando admite un gran número de parámetros, pero en estos apuntes vamos a usarlo simplemente para indicar que queremos obtener decimales en las operaciones. Para ello simplemente haremos el siguiente cambio..

MEDIA =$[$SUMA / 3]

Lo sustituiremos por

MEDIA=$(echo "scale=4; $SUMA/3" | bc –l)

Vemos cómo debemos generar una salida con echo, el primer campo scale indica cuantos decimales queremos obtener (4 en este caso), luego y separado por un punto y coma ponemos la operación aritmética que deseamos realizar, sin necesidad de poner corchetes, dobes paréntesis o usar let. El resultado de este echo lo enviamos al comando bc –l mediante una tubería.

Para entenderlo un poco mejor, comprobad las siguientes líneas:

```
usuario@debian6:~$ echo $[ 20 / 6 ]
3
usuario@debian6:~$ echo "scale=2; 20/6" | bc -l
3.33
usuario@debian6:~$ echo "scale=4; 20/6" | bc -l
3.3333
usuario@debian6:~$
```

FUNCIONES.

Usar funciones en los scripts es muy simple. Basta con usar la siguiente estructura al **principio** del script:

```
function nombre_función {
        líneas de la función
}
```

Estas líneas de la función no se ejecutarán al procesar el script, sino que solo se ejecutarán cuando en el cuerpo del script usemos nombre_funcion.
Ejemplo:

```
#!/bin/bash
function doble {
        echo "voy a doblar el valor de numero"
        let NUMERO=NUMERO*2
}
NUMERO=3
echo '$NUMERO vale : ' $NUMERO
doble                                   # llamamos a la función
echo '$NUMERO vale : ' $NUMERO
```

Podría parecer en una lectura rápida del script anterior, que estamos pasando por let NUMERO=NUMERO*2 antes de asignarle el valor 3. No es cierto, ya que aunque veamos esas líneas físicamente anteriores a la asignación, solo serán procesadas cuando en el script escribimos doble.

Por defectos, todas las variables que usemos son globales, es decir, que las funciones y el script las comparten, pueden modificar sus valores, leer las modificaciones realizadas, etc. Sin embargo, en determinadas ocasiones nos puede interesar que las variables sean locales a la función, es decir, que si la función modifica su valor el script no se entera…

```
#!/bin/bash
function saludo {
        NOMBRE="Jose Antonio"
        echo "Hola señor $NOMBRE encantado de conocerle"
}
NOMBRE="Juana"
saludo
echo "En el script principal, mi nombre es $NOMBRE"
```

En este ejemplo, vemos como nos aparece *En el script principal, mi nombre es Jose Antonio*, ya que cuando en la función se modifica NOMBRE, se modifica en todo el ámbito del programa.

Esto es así porque al inicializar la variable NOMBRE como NOMBRE=”Jose Antonio” estamos creando una variable global, visible tanto en la función como en el script.
Sin embargo…

```
#!/bin/bash
function saludo {
        local NOMBRE=”Jose Antonio”
        echo “Hola señor $NOMBRE encantado de conocerle”
}
NOMBRE=”Juana”
saludo
echo “En el script principal, mi nombre es $NOMBRE”
```

Vemos como ahora, al anteponer local a la variable NOMBRE en la función, las modificaciones que se realicen sólo afectan a la propia función, por lo que en pantalla vemos como aparece *En el script principal, mi nombre es Juana*.

ESTRUCTURAS CONDICIONALES.

IF.

La principal estructura condicional de los scripts en shell es el if (Sí en inglés):

```
if [ expresión ]; then
        Realizar si expresión es verdadera
fi
```

La expresión es cualquier expresión lógica que produzca un resultado verdadero o falso. Si estamos operando con cadenas alfanuméricas, los operadores que podemos utilizar son los siguientes:

Operadores de comparación de cadenas alfanuméricas	
Cadena1 = Cadena2	*Verdadero si Cadena1 es IGUAL a Cadena2*
Cadena1 != Cadena2	*Verdadero si Cadena1 NO es IGUAL a Cadena2*
Cadena1 < Cadena2	*Verdadero si Cadena1 es MENOR a Cadena2*
Cadena1 > Cadena2	*Verdadero si Cadena1 es MAYOR que Cadena2*
-n Variable1	*Verdadero si Cadena1 NO ES NULO (tiene algún valor)*
-z Variable1	*Verdadero si Cadena1 ES NULO (está vacía o no está definida)*

```
usuario@debian6:~$ if [ "jose" = "juan" ]; then echo "iguales"; fi
usuario@debian6:~$ if [ "jose" = "jose" ]; then echo "iguales"; fi
iguales
usuario@debian6:~$ _
```

```
usuario@debian6:~$ cat dia.sh
#!/bin/bash
DIA=$( date +%A )
if [ $DIA="viernes" ]; then
        echo "Bravo, por fin es viernes"
fi
```

Los anteriores operadores sólo son válidos para comparar cadenas, si queremos comparar valores numéricos, hemos de usar los siguientes operadores:

Operadores de comparación de valores numéricos.	
Numero1 -eq Numero2	*Verdadero si Numero1 es IGUAL a Numero2. (equal)*
Numero1 -ne Numero2	*Verdadero si Numero1 NO es IGUAL a Variable2. (not equal)*
Numero1 -lt Numero2	*Verdadero si Numero1 es MENOR a Variable2. (less that)*
Numero1 -gt Numero2	*Verdadero si Numero1 es MAYOR que Variable2. (greater that)*
Numero1 -le Numero2	*Verdadero si Numero1 es MENOR O IGUAL que Numero2. (less or equal).*
Numero1 -ge Numero2	*Verdadero si Numero1 es MAYOR O IGUAL que Numero2 . (greater or equal).*

```
usuario@debian6:~$ cat dia2.sh
#!/bin/bash
DIA=$( date +%A )
if [ $DIA="viernes" ]; then
        echo "Bravo, por fin es viernes"
fi
```

```
usuario@debian6:~$ cat numeros.sh
#!/bin/bash
NUMERO=7
if [ $NUMERO -gt 5 ]; then
        echo "Numero es mayor que 5"
fi
if [ $NUMERO -lt 5 ]; then
        echo "Numero es menor que 5"
fi
```

Si usamos operadores de comparación numéricos con valores de cadena, el sistema nos dará un error como el siguiente:

```
usuario@debian6:~$ if [ "jose" -eq "jose" ]; then echo "iguales"; fi
bash: [: jose: se esperaba una expresión entera
usuario@debian6:~$
```

```
usuario@debian6:~$ NUMERO=juana
usuario@debian6:~$ if [ $NUMERO -gt 5 ]; then echo "Mayor de 5"; fi
bash: [: juana: se esperaba una expresión entera
usuario@debian6:~$
usuario@debian6:~$
```

IF..ELSE

La estructura if podemos ampliarla usando la construcción else (en caso contrario) y elif (en caso contrario si…).

La estructura simple de else es la siguiente:

if [expresión 1]; then
Realizar si expresión 1 es verdadera
else
Realizar si expresión 1 es falsa
fi

```
usuario@debian6:~$ cat dia3.sh
#!/bin/bash
DIA=$( date +%d )
if [ $DIA -lt 15 ]; then
        echo "Estamos en la primera quincena del mes"
else
        echo "Estamos en la segunda quincena del mes"
fi
```

IF..ELIF..ELSE

Una estructura con elif (else if) tiene la siguiente forma:

if [expresión1]; then
Realizar si expresión1 es verdadera

elif [expresión2]; then
Realizar si expresión1 es falsa, pero es verdadera expresión2

elif [expresión3]; then
Realizar si exp1 y exp2 son falsas, pero es verdadera expresión3

else
realizar si todas las expresiones anteriores son falsas

fi

```
#!/bin/bash
NUMERO_HIJOS=3
if [ $NUMERO_HIJOS -eq 1 ]; then
        echo "Tienes un unico hijo"
elif [ $NUMERO_HIJOS -eq 2 ]; then
        echo "Tienes dos hijos"
elif [ $NUMERO_HIJOS -ge 3 ]; then
        echo "3 o más hijos, eso es familia numerosa"
else
        echo "No tienes ningún hijo"
fi
```

Hay que tener muchísimo cuidado con los espacios en blanco, y seguramente durante nuestros primeros scripts casi todos los errores vendrán por haberlos usado mal en las estructuras if.

Hay que recordar que los corchetes llevan espacios en blanco tanto a izquierda como derecha, que el punto y coma sin embargo va pegado al corchete cerrado, y que SIEMPRE hay que poner espacios en blanco en las expresiones.

Veamos algunos errores muy comunes, para que no los cometáis.

If [3 -eq 5]; then	bash: [3: command not found. Hemos usado [3 en lugar de [3
if ["Jose" -eq "Jose"]; then	Bash: [: jose: integer expression expected. Debíamos haber usado =
if [3 = 4]; then	Esto no nos devolverá error, y parece que funciona, pero en realidad no es así, hay que usar -eq
if [3 > 4]; then	Esto devuelve verdadero. Sirva como prueba que no hay que usar operadores de cadena para comparar números como dijimos anteriormente.
If [jose=Antonio]; then	No hemos dejado espacios en la condición =. Esta expresión da como valor verdadero. Mucho cuidado con este error, que nos puede volver locos en depuración.

Otro error muy común es el siguiente:

```
#!/bin/bash
PROFESOR="Juana"
if [ $PROFSOR = "Juana" ]; then
        echo "Hola Juana"
fi
```

Este programa nos devuelve por pantalla el siguiente error:
bash: [= unary operador expected

Que traducido resulta, me he encontrado un [(corchete abierto) y luego un operador (el =) sin nada en medio, y eso no funciona.

Revisando el programa anterior, vemos como nos hemos equivocado en el nombre de la variable, por lo cual $PROFSOR no tiene ningún valor (es nula) y por lo tanto al no valer nada, el programa lo que ve es lo siguiente: if [= "Juana"].

Hemos visto operadores aritméticos y operadores para cadena, pero en las expresiones podemos utilizar cualquier operación que nos devuelva un valor lógico (0 para verdadero). Por ejemplo, podemos usar la función test del bash, que funciona de la siguiente forma:

Operaciones condicionales usando test.	
-a fichero	Verdadero si fichero existe
-d fichero	Verdadero si fichero existe, y es un fichero de tipo directorio

-f fichero	Verdadero si fichero existe, y es un fichero regular.
-r fichero	Verdadero si fichero existe y se puede leer
-w fichero	Verdadero si fichero existe y se puede escribir
-x fichero	Verdadero si fichero existe y se puede ejecutar
fichero1 -nt fichero2	Verdadero si fichero1 es más nuevo que fichero2
fichero1 -ot fichero2	Verdadero si fichero1 es más viejo que fichero2

Si lo necesitamos, podemos anidar expresiones usando tanto and (y, &&) como or (o, ||).

```
if [ expresión1 ] && [ expresión2 ]; then
        se ejecuta si expresión1 Y expresión2 son verdaderas
fi

if [ expresión1 ] || [ expresión2 ]; then
        se ejecuta si expresión1 O expresión2 son verdaderas
fi
```

También podemos usar el operador not (!) para indicar una negación.

```
if ! [ expresión1 ]; then
        se ejecuta si expresión1 NO es verdadera
fi
```

Para hacer algunos ejercicios, vamos a aprovechar para explicar mejor cómo le podemos pedir datos al usuario. Se hace con la orden read y es muy simple de usar:

```
read -p "texto de la pregunta" variable
```

La ejecución del script se parará, mostrará por pantalla el texto de la pregunta, y dejará que el usuario escriba la respuesta, cuando pulse INTRO la respuesta dada se introducirá como valor de variable.

read también puede ser usada sin el parámetro -p, de la forma read variable. También podemos hacer que lea un determinado número de caracteres, sin obligar a que el usuario pulse intro, con el parámetro -n número_de_caracteres. El parámetro -s silencia el eco (no se ve por pantalla lo que el usuario escribe).

Ahora que sabemos usar el read, hagamos por ejemplo un programa que nos permita indicar si un número introducido es par o impar.

```
#!/bin/bash
# parimpar.sh - script que nos pide un número e indica si es par o impar.
clear
read -p "Introduzca un número : " NUMERO
let RESTO=NUMERO%2
if [ $RESTO -eq 0 ]; then
        echo "El número $NUMERO es par"
else
        echo "El número $NUMERO es impar"
fi
```

Haced vosotros un script que pida un número por pantalla, e indique si es un múltiplo de 10 o no. Haced otro script que nos pida un nombre por pantalla y nos diga si dicho nombre comienza por la letra jota mayúscula.

Hagamos un script un poco más complicado… vamos a pedir al usuario un número de 3 cifras y vamos a indicar si es capicúa.

```
#!/bin/bash
# capicua.sh - script que nos pide un número de tres cifras e indica si es
# capicúa o no.
clear
read -n 3 -p  "Número entre 100 y 999 (no pulses INTRO) : " NUMERO
echo  # este echo sirve para introducir un retorno de linea
if [ $NUMERO -lt 100 ]; then
        echo "Lo siento, has introducido un número menor de 100"
else
        PRIMERA_CIFRA=$(echo $NUMERO | cut -c 1)
        TERCERA_CIFRA=$(echo $NUMERO | cut -c 3)
        if [ $PRIMERA_CIFRA = $TERCERA_CIFRA ]; then
                echo "El número $NUMERO es capicúa."
        else
                echo "El número $NUMERO ni es capicúa ni ná".
        fi
fi
```

Es evidente que podíamos haber hecho este último script mucho más corto, por ejemplo usando una línea como:

```
if [ $(echo $NUMERO | cut -c 1) = $(echo $NUMERO | cut -c 3) ]; then
```

Pero eso ya queda al gusto de cada programador. A mí personalmente me gustan los programas que pueden entenderse simplemente echándole una ojeada al fuente, y lo aconsejo fuertemente al menos hasta que tengáis un nivel de programación muy alto. (Y aun entonces, facilita mucho la modificación posterior de los programas).

Cuando hacemos un script de varias líneas como el anterior, es posible que cometamos algún fallo. Una opción que podemos usar para depurar los scripts y encontrar rápidamente los errores, es añadir un -x en la llamada al bash de la primera línea. Esto hará que cada línea antes de ejecutarse sea mostrada por pantalla tal y como la está interpretando el bash.

```
#!/bin/bash -x
```

Para que esto funcione, es necesario que hagamos el script ejecutable, no es válido si lanzamos el script con la orden source o con el punto. Hay que ejecutar el script haciéndolo antes ejecutable con chmod y luego ejecutándolo con ./script.

Como ejemplo del if, haced un script que nos diga si en el directorio actual hay más de 10 ficheros o no.

Haced otro script que nos pida la altura de 3 personas en centímetros, y nos diga por pantalla la mayor de esas alturas.

Otro más que nos pida la edad y nos diga por pantalla en que década nacimos. (La década de los 70, la de los 80, la de los 90, etc.). Suponemos que todo el mundo tiene más de 15 años y nadie tiene más de 60.

Y el último script por el momento debe hacer lo siguiente: nos pedirá por pantalla el nombre de un mes (enero, febrero, etc.) y nos dirá por pantalla el número de días que tiene ese mes. (Así, si introducimos diciembre nos responderá "Diciembre tiene 31 días"). Para este script se considera que no existen los años bisiestos, así que febrero siempre tendrá 28 días.

CASE.

Hemos visto la principal estructura condicional que es el if, pero tenemos alguna otra a nuestra disposición, como el case. Esta estructura nos permite ejecutar varias acciones, dependiendo del valor de una variable o expresión.

```
case VARIABLE in
        valor1)
                se ejecuta si VARIABLE tiene el valor1
                ;;
        valor2)
                se ejecuta si VARIABLE tiene el valor2
                ;;
        *)
                Se ejecuta si VARIABLE tiene otro valor distinto
                ;;
esac
```

Veamos un ejemplo de utilización del case.

```
usuario@debian6:~$ cat cp.sh
#!/bin/bash
read -n 3 -p "Introduzca los tres primeros digitos del codigo postal : " CP
echo
case $CP in
        110)
                echo "Cadiz capital" ;;
        112)
                echo "Algeciras" ;;
        114)
                echo "Jerez de la Frontera" ;;
        *)
                echo "Ese codigo no esta contemplado" ;;
esac
```

En el case, no solo podemos preguntar por valores directos, sino que también podemos utilizar los comodines que vimos anteriormente. Veamos un par de ejemplos de case utilizado junto con comodines.

```
usuario@debian6:~$ cat cp.sh
#!/bin/bash
read -n 5 -p "Introduzca el codigo postal : " CP
echo
case $CP in
        110*)
                echo "Cadiz capital" ;;
        112*)
                echo "Algeciras" ;;
        114*)
                echo "Jerez de la Frontera" ;;
        [1110-1111]*)
                echo "San Fernando" ;;
        1112*)
                echo "Campo Soto" ;;
        *)
                echo "Ese codigo no esta contemplado" ;;
esac
```

```
usuario@debian6:~$ cat vocal.sh
#!/bin/bash
read -n 1 -p "Dime una letra en minuscula : " LETRA ; echo
case $LETRA in
        a|e|i|o|u)
                echo La letra es una vocal ;;
        *)
                echo La letra es una consonante ;;
esac
```

Como ejercicio, haced un script con nombre horóscopo.sh que nos pida el año en que nacimos (4 cifras) y nos diga por pantalla que animal nos corresponde según el horóscopo chino. Para calcularlo debemos dividir el año entre 12 y el resto nos indicará el animal según la siguiente tabla.

0	El Mono	4	La Rata	8	El Dragón
1	El Gallo	5	El Buey	9	La Serpiente
2	El Perro	6	El Tigre	10	El Caballo
3	El Cerdo	7	El Conejo	11	La Cabra

El mensaje que queremos obtener en pantalla debe ser:
Si naciste en 1965 te corresponde La Serpiente según el horóscopo chino.

ESTRUCTURAS ITERATIVAS. (BUCLES).

Las principales estructuras iterativas que podemos usar en shell scripts son for, while, until y select.

FOR.

La estructura básica de for es la siguiente:

for VARIABLE in conjunto; do
Estas líneas se repiten una vez por cada elemento del conjunto,
Y variable va tomando los valores del conjunto uno por uno.
done

Ese conjunto que aparece en la estructura del for, es normalmente un conjunto de valores cualesquiera, separados por espacios en blanco o retornos de línea. Así, si queremos mostrar los días de la semana por pantalla podríamos hacerlo mediante este script:

```
usuario@debian6:~$ cat dias_semana.sh
#!/bin/bash
for DIA in lunes martes miercoles jueves viernes sabado domingo; do
        echo dia de la semana : $DIA
done
```

```
usuario@debian6:~$ ./dias_semana.sh
dia de la semana : lunes
dia de la semana : martes
dia de la semana : miercoles
dia de la semana : jueves
dia de la semana : viernes
dia de la semana : sabado
dia de la semana : domingo
usuario@debian6:~$
```

Así, por ejemplo, si queremos obtener por pantalla los números del 1 al 10 podríamos hacerlo de la siguiente forma:

```
usuario@debian6:~$ cat ejemplo_flor.sh
#!/bin/bash
for NUM in 1 2 3 4 5 6 7 8 9 10; do
        echo "NUM vale $NUM en este paso."
done
```

La potencia del comando for viene de la flexibilidad de valores que admite el conjunto de valores, ya que podemos crear dicho conjunto con una orden del sistema operativo. En el siguiente ejemplo vamos a usar como conjunto los nombres de los ficheros con extensión sh del directorio actual:

```
#!/bin/bash
for NOMBRE in $( ls *.sh ); do
        echo "Un script es $NOMBRE"
done
```

El conjunto puede ser cualquier salida de cualquier orden, y formara elementos utilizando el espacio en blanco como separador de elementos. Fijaros en el siguiente ejemplo:

```
usuario@debian6:~$ cat NOMBRES.TXT
ana
juana
luis
ango
emilio
jose antonio
fernando

usuario@debian6:~$ cat ejemplo_for2.sh
#!/bin/bash
for C in $( cat NOMBRES.TXT ); do
        echo "La variable vale $C en este paso."
done

usuario@debian6:~$ . ejemplo_for2.sh
La variable vale ana en este paso.
La variable vale juana en este paso.
La variable vale luis en este paso.
La variable vale ango en este paso.
La variable vale emilio en este paso.
La variable vale jose en este paso.
La variable vale antonio en este paso.
La variable vale fernando en este paso.
usuario@debian6:~$
```

Vemos como utilizamos como conjunto el contenido de un fichero. Vemos también como la línea “jose antonio” la divide en 2 elementos distintos debido al espacio en blanco.

Existe una orden en GNU/Linux que nos permite obtener una secuencia de números como salida de la orden, esta orden es seq.

seq último-número
seq primer-número último-número
seq primer-número incremento último-número

Modifiquemos el ejercicio de mostrar los números del 1 al 10 que hicimos anteriormente, usando esta vez la orden seq.

```
usuario@debian6:~$ cat ejemplo_flor.sh
#!/bin/bash
for NUM in $( seq 10 ); do
        echo "NUM vale $NUM en este paso."
done
```

Vamos a realizar un ejemplo algo más complejo utilizando for y seq. Vamos a crear un script llamado suma100.sh que nos va a decir por pantalla cuanto suman todos los números del 1 al 100, es decir, 1+2+3+4+5…+100.

```
usuario@debian6:~$ cat suma100.sh
#!/bin/bash
SUMA=0
for NUM in $( seq 1 100 ); do
        let SUMA=SUMA+NUM
done
echo "Los numeros del 1 al 100 suman : " $SUMA
```

Modificad el anterior ejercicio para que el script sume todos los números pero no entre 1 y 100, sino entre dos números que pida el script por pantalla.

Desde la versión de bash 3.0 se introdujo un cambio en el for que permite utilizar directamente rangos sin tener que usar la orden seq. Si estamos seguros de que contamos con un bash moderno podemos utilizar la siguiente característica del for:

```
usuario@debian6:~$ cat for-rango1.sh
#!/bin/bash
for NUM in {1..5}; do
        echo NUM vale $NUM
done
usuario@debian6:~$
```

Desde la versión de bash 4.0 se introdujo otro cambio, que permite utilizar también incrementos en los rangos, de la siguiente manera:

```
usuario@debian6:~$ cat for-rango2.sh
#!/bin/bash
# de 1 a 50 con incremento de 5
for NUM in {1..50..5}; do
        echo NUM vale $NUM
done
```

Como vemos, podemos utilizar los rangos de la siguiente forma:

{INICIO..FINAL}
{INICIO..FINAL..INCREMENTO}

Si nuestro bash es de un sistema relativamente moderno, podemos usar estos rangos sin ningún tipo de problemas, que tienen la ventaja adicional de ser algo más rápidos que la orden seq.

Si no estamos seguros de los sistemas sobre los que se ejecutara nuestro script podemos usar el seq para conseguir una mayor compatibilidad, aunque hoy en día es muy difícil encontrar en ningún sistema un bash inferior al 4.0.

El for de bash también permite utilizar el formato basado en trio de expresiones común al lenguaje C.

for ((expresión-inicio; condición-para-seguir; expresión-de-paso))

Normalmente utilizaremos este formato de la siguiente forma:

for ((VARIABLE=inicio; condición-para-seguir; incrementamos))

Veamos un ejemplo, con un script que como salida nos muestra los números pares entre 2 y 40.

```
usuario@debian6:~$ cat for-c.sh
#!/bin/bash
for (( NUM=2; NUM<=40; NUM=NUM+2 )); do
        echo $NUM
done
usuario@debian6:~$
```

Podemos crear un bucle infinito de la siguiente forma:

```
usuario@debian6:~$ cat for-infinito.sh
#!/bin/bash
for (( ; ; )); do
        echo "Esto es un bucle infinito. Pulsa Control C para matarlo."
done
usuario@debian6:~$
```

Podemos utilizar la instrucción break para salirnos inmediatamente de un bucle for. Fijaros en el siguiente ejemplo:

```
usuario@debian6:~$ cat dobles.sh
#!/bin/bash
for (( ; ; )); do
        read -p "Dime un número (introduce 0 para salir) : " NUMERO
        if [ $NUMERO -eq 0 ]; then
                break
        else
                echo ".......el doble es $[ $NUMERO * 2 ]"
        fi
done
echo "Hemos salido del bucle y se acabo el programa."

usuario@debian6:~$
```

Este tipo de elementos (bucles infinitos, break, etc.) se consideran como “poco elegantes” desde el punto de vista de la programación y es mejor acostumbrarse a no usarlos, ya que existen otro tipo de alternativas más refinadas. Sin embargo son herramientas potentes y es conveniente conocerlas.

Imaginemos que queremos copiar a un llaverito USB (montado en /media/usbdisk por ejemplo) todos los scripts que tengamos en nuestro directorio home, sin importar en que directorio estén, podríamos hacerlo fácilmente con este script:

```
#!/bin/bash
for programa in $( find ~ -iname "*sh" 2> /dev/null ); do
        echo "copiando el script :" $programa
        cp $programa /media/usbdisk
done
```

Ya que estamos, mejoremos el script anterior para que cree un directorio scripts en nuestro llaverito, pero únicamente si no existe.

```
#!/bin/bash
if ! [ -d /media/usbdisk/scripts ]; then
        mkdir /media/usbdisk/scripts
fi
for programa in $( find ~ -iname "*sh" 2> /dev/null ); do
        echo "copiando el script :" $programa
        cp $programa /media/usbdisk
done
```

WHILE Y UNTIL

Cuando no queremos recorrer un conjunto de valores, sino repetir algo mientras se cumpla una condición, o hasta que se cumpla una condición, podemos usar las estructuras while y until.

La estructura del while es la siguiente:

```
while [ expresión ]; do
        estas líneas se repiten MIENTRAS la expresión sea verdadera
done
```

La estructura del until es la siguiente:

```
until [ expresión ]; do
        estas líneas se repiten HASTA que la expresión sea verdadera
done
```

Ambas estructuras, tanto while como until realmente realizan exactamente lo mismo, al efectuar la comprobación de la expresión en la primera línea, no como en otros lenguajes.

Veamos un ejemplo de un script usando la estructura while (mientras).

```
#!/bin/bash
#doble.sh - script que pide números y muestra el doble de dichos números.
# el script continua ejecutandose mientras que no se introduzca 0.
read -p "Dime un número (0 para salir) : " NUMERO
while [ $NUMERO -ne 0 ]; do
        echo "El doble de $NUMERO es :" $(($NUMERO*2))
        read -p "Dime un número (0 para salir) : " NUMERO
done
```

Ahora veamos cómo queda el script, usando la estructura until (hasta).

```
#!/bin/bash
#doble.sh - script que pide números y muestra el doble de dichos números.
# el script continua ejecutandose mientras que no se introduzca 0.
read -p "Dime un número (0 para salir) : " NUMERO
until [ $NUMERO -eq 0 ]; do
        echo "El doble de $NUMERO es :" $(($NUMERO*2))
        read -p "Dime un número (0 para salir) : " NUMERO
done
```

Otro ejemplo, vamos a mostrar por pantalla los número del 1 al 20

```
#!/bin/bash
NUMERO=1
until [ $NUMERO -gt 20 ]; do
        echo "Número vale :" $NUMERO
        let NUMERO=NUMERO+1
done
```

SELECT

La última estructura iterativa que vamos a ver es select. Esta nos permite realizar una iteración o bucle, pero presentando un menú por pantalla para que el usuario escoja una opción. Su estructura general es la siguiente:

select VARIABLE in conjunto_opciones; do
Aquí variable toma el valor de una de las opciones del conjunto
done

Esta estructura como vemos es muy parecida a la del for, pero presenta la principal diferencia en que por definición se crea un bucle sin final, no hay un valor inicial y un valor límite, el bucle se repetirá eternamente, lo que nos obliga a salirnos del mismo por las bravas, bien con break que nos permite salirnos del bucle o con exit que nos permite salirnos del script entero.

Veamos un ejemplo:

```
usuario@debian6:~$ cat select1.sh
#!/bin/bash
select RESP in Chiste Refran Poema Salir; do
        case $RESP in
                Chiste)
                        echo "Soy muy indeciso... o no" ;;
                Refran)
                        echo "Cria cuervos y tendras muchos" ;;
                Poema)
                        echo "Tu corazón hace tolón" ;;
                Salir)
                        break
        esac
done

usuario@debian6:~$
```

Al ejecutar el script por pantalla nos presentará un menú automáticamente formado por el conjunto de opciones que hemos puesto en el select.

```
usuario@debian6:~$ . select1.sh
1) Chiste
2) Refran
3) Poema
4) Salir
#?
```

Y automáticamente el script realizará un read, pidiendo que el usuario introduzca un valor. El valor debe ser uno de los números otorgados a las opciones (1.4 en nuestro caso). Aunque el valor que introduzcamos es un número, dentro del script podéis comprobad como la variable del select no toma este valor numérico, sino el texto de la opción.

```
usuario@debian6:~$ . select1.sh
1) Chiste
2) Refran
3) Poema
4) Salir
#? 1
Soy muy indeciso... o no
#? 3
Tu corazón hace tolón
#? 2
Cria cuervos y tendras muchos
#? 2
Cria cuervos y tendras muchos
#? 1
Soy muy indeciso... o no
#? 4
usuario@debian6:~$
```

Podemos comprobar como el bucle es infinito, y la única forma de salir es usando la opción Salir que en el script ejecuta un break.

Al igual que sucedía con el for, es perfectamente posible crear el conjunto mediante una instrucción. Así por ejemplo, la instrucción ls nos devuelve un conjunto formado por todos los ficheros del directorio actual. Vamos a trabajar sobre esta idea:

```
#!/bin/bash
select FICHERO in $( ls ); do
        echo Has seleccionado el fichero $FICHERO
        # Ahora podríamos borrarlo, copiarlo, visualizarlo, etc.
Done
```

Si ejecutáis ese script, veréis dos cosas: Como el conjunto está formado por todos los ficheros del directorio actual, y como es imposible detener la ejecución del script, como no sea matando el proceso en primer plano con Control + C

Por cierto, si en el conjunto ponemos directamente el símbolo asterisco (*) veremos que tiene la misma función que un ls, devuelve el listado de ficheros del directorio actual.

```
select FICHERO in *; do
```

Hagamos otro ejemplo sobre el select, un poco más avanzado. Vamos a mostrar por pantalla un menú con todos los mp3 que existan en el directorio home del usuario actual, y vamos a dejar que escoja uno de ellos para reproducirlo. (Para ello uso un reproductor de mp3 desde línea de comandos podemos usar la orden mpg321, si no la tenéis instalado lo podéis instalar con un apt-get install mpg321).

```
#!/bin/bash
clear
select MP3 in $( find . -iname "*mp3" ); do
        echo "Voy a reproducir el mp3 : " $MP3
        mpg321 $MP3 &> /dev/null
done
```

Volvemos a tener el problema de que la ejecución no se acabará nunca, a menos que la interrumpamos mediante control c. Vamos a arreglarlo forzando la opción Salir en el conjunto:

```
#!/bin/bash
clear
CONJUNTO=$(find . -iname "*mp3")
CONJUNTO=$CONJUNTO" Salir"
select MP3 in $CONJUNTO; do
        if [ $MP3 = "Salir" ]; then
                break
        fi
        echo "Voy a reproducir el mp3 : " $MP3
        mpg321 $MP3 &> /dev/null
done
```

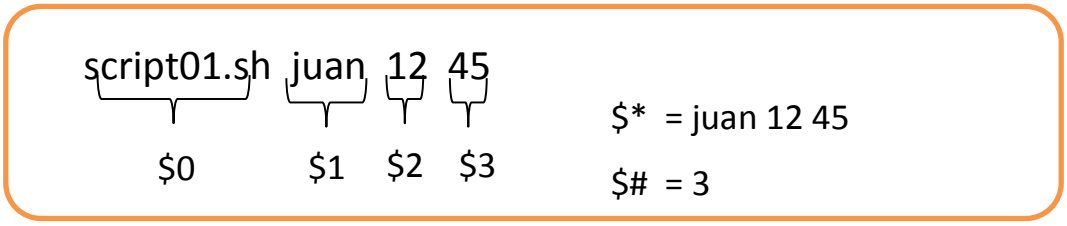
Como siempre en Informática, este script tan sencillo se puede complicar hasta lo inimaginable.

Por ejemplo, este script necesita para funcionar que los nombres de los archivos mp3 no contengan espacios en blanco, ya que el conjunto separa sus valores por este carácter. Así, si tuviéramos una canción con nombre La Gasolina, veríamos que en el menú nos aparecen dos opciones 1) La y 2) Gasolina, por lo que el script como es obvio no funcionará. ¿Se os ocurre alguna manera de solucionarlo?

PASO DE PARÁMETROS A UN SCRIPT.

Podemos pasar parámetros tanto a los scripts como a las funciones. Los parámetros en bash se indican como un símbolo dólar ($) seguido de un número o carácter. Los principales parámetros que se pueden usar son:

Parámetros	
$1	Devuelve el 1º parámetro pasado al script o función al ser llamado.
$2	Devuelve el 2º parámetro.
$3	Devuelve el 3º parámetro. (Podemos usar hasta $9).
$*	Devuelve todos los parámetros separados por espacio.
$#	Devuelve el número de parámetros que se han pasado.
$0	Devuelve el parámetro 0, es decir, el nombre del script o de la función.



Para comprobar lo anterior, cread un script como el siguiente:

```
#!/bin/bash
# parámetros.sh - script sobre parámetros.
echo "El primer parámetro que se ha pasado es " $1
echo "El tercer parámetro que se ha pasado es " $3
echo "El conjunto de todos los parámetros : " $*
echo "Me has pasado un total de " $# " parámetros"
echo "El parámetro 0 es : " $0
```

Si hacemos este script ejecutable, y lo llamamos como:

./parámetros.sh Caballo Perro 675 Nueva York

Obtendríamos por pantalla lo siguiente:

El primer parámetro que se ha pasado es Caballo
El tercer parámetro que se ha pasado es 675
El conjunto de todos los parámetros : Caballo Perro 675 Nueva York
Me has pasado un total de 5 parámetros
El parámetro 0 es: ./parámetros.sh

Como he indicado antes, también podemos pasarle parámetros a las funciones, usando el mismo método y las mismas posibilidades que para los scripts completos.

```
#!/bin/bash
function mayor_edad {
        if [ $1 -ge 18 ]; then
                echo Si, es mayor de edad
        else
                echo No, es menor de edad
        fi
}
read -p "Dime la edad del que quiere entrar : " EDAD
echo voy a comprobar si puede entrar o no.
mayor_edad $EDAD
```

Como práctica, intentad modificad el script que hicimos explicando los if, que tenía como misión indicar si un numero introducido era capicúa o no. Modificadlo de tal modo que en lugar de pedir el número al usuario mediante un read, use directamente el número pasado como parámetro 1. Es decir, el script se ejecutará así: *./capicua.sh 767*

VALORES DEVUELTOS POR LAS ÓRDENES.

Existe un parámetro especial, el $? que nos devuelve el valor del resultado de la última orden.

Es decir, después de ejecutar cualquier orden o comando del sistema (o casi cualquier orden mejor dicho) podemos comprobar el valor de $? que tendrá un 0 si todo ha ido bien, y otro valor cualquiera en caso de que haya fallado. Comprobarlo es muy simple:

Desde la línea de comandos, haced un cd a un directorio que no existe, por ejemplo

cd /juegos/faluyah

y luego mirad el contenido de $? con un

echo $?

Comprobareis como $? vale 1, es decir, indica que la última orden no funcionó correctamente.

Ahora haced un cd a un directorio que si exista

cd /etc/network

y luego mirad el contenido de $? con un

echo $?

Comprobareis como vale 0, es decir, indica que la última orden funciono sin problemas.

Este parámetro puede sernos muy útil realizando scripts, ya que nos permite una forma rápida y cómoda de ver si todo está funcionando bien o no.

Como ejemplo, realizad un script con nombre borrar.sh. Dicho script aceptará como parámetro el nombre de un fichero. El script debe eliminar ese fichero, pero antes debe guardar una copia de seguridad del mismo en el directorio papelera que debemos crear en nuestro home de usuario. Una vez comprobado que funciona, pasadle como parámetro el nombre de un fichero que el usuario no tenga permisos para borrar (recordad que además debe estar en un directorio en el que el usuario no tenga el permiso de escritura). Como es obvio, el script nos dará un error al intentar borrar dicho fichero, pues precisamente después de ese rm es donde podemos colocar un if preguntando por $?, de modo que interceptemos el error y avisemos al usuario de que dicho fichero no ha podido ser borrado.

Modificar el script anterior, de modo que si el usuario no le pasa ningún parámetro, el script se dé cuenta y avise de ello por pantalla.

UN SCRIPT COMPLETO DE EJEMPLO

Hagamos un script que nos permita simular el juego ese de pensar un número y que el jugador lo adivine proponiendo varios números, a los que se responderá únicamente si se han quedado cortos o se han pasado. Vamos a realizarlo llevando un control de cuantos intentos llevan y un contador de record que nos permitirá mostrar las 3 personas que lo han resuelto en menos intentos.

```
#!/bin/bash
# juego1.sh -       script que permite jugar a adivinar un número en varios
#                   intentos y llevando un control de los 3 mejores.
clear
# Si pasamos como parametro x borramos fichero record
if [ $# -ne 0 ]; then # para controlar que se han pasado parámetros.
        if [ $1 = x ]; then
                echo "Borrando fichero de records."
                rm record.txt
        fi
fi
#
# Ahora vamos a leer el fichero de records para imprimirlo.
#
if [ -f record.txt ]; then
        POS=0
        for CAMPEON in $(cat record.txt); do
                let POS=POS+1
                NOMBRE=$(echo $CAMPEON | cut -d: -f2)
                NUMERO=$(echo $CAMPEON | cut -d: -f1)
                echo "En posición $POS esta $NOMBRE con $NUMERO
                intentos"
        done
else
        echo '*********************************************'
        echo "No hay ningún record todavía. Aprovecha la oportunidad"
        echo '*********************************************'
fi
#
# comenzamos el juego en sí.
#
CONTADOR=1
let MINUMERO=RANDOM  # $RANDOM nos da un número aleatorio.
# MINUMERO=3 # para hacer pruebas, descomentar esta línea y comentar
# la anterior.
echo ' ' ; echo ' '
echo '*************************************************'
read -p "Dime tu nombre : " NOMBRE
echo ' ' ; echo ' '
read -p "Llevas $CONTADOR intentos. Dime un número: " NUMERO
#
#
```

```
until [ $NUMERO -eq $MINUMERO ]; do
        if [ $NUMERO -gt $MINUMERO ]; then
                echo "El número que has metido es mayor"
        else
                echo "El número que has metido es menor"
        fi
        let CONTADOR=CONTADOR+1
        read -p "Llevas $CONTADOR intentos. Dime un numero: " NUMERO
done
#
echo Hombreeee, por fin acertaste.
# grabamos el record en el fichero (primero los intentos y luego el nombre)
echo $CONTADOR:$NOMBRE >> record.txt
# ordenamos para dejar arriba los que lo han hecho en menos intentos
# y nos quedamos con las 3 primeras líneas
sort record.txt -g | head -3 > recordtemp.txt
cp recordtemp.txt record.txt
```

Una cosa interesante que podéis ver en este script, es como hemos aprovechado las funciones de las ordenes Linux para simplificar el programa. Todo el tema de ordenación de los record, quedarnos solo con los 3 primeros, comprobar si el usuario actual ha batido algún record, etc., lo hemos realizado simplemente con un comando bash de Linux, en este caso un head que viene de un sort. Si no lo hubiéramos hecho así, el script sería mucho más largo y complicado.

EJERCICIOS SOBRE SCRIPTS.

1) Crear un script con nombre crear.sh que admita dos parámetros, el primero indicará el nombre de un fichero, y el segundo su tamaño. El script creará en el directorio actual un fichero con el nombre dado y el tamaño dado en Kilobytes. En caso de que no se le pase el segundo parámetro, creará un fichero con 1.024 Kilobytes y el nombre dado. En caso de que no se le pase ningún parámetro, creará un fichero con nombre fichero_vacio y un tamaño de 1.024 Kilobytes.

   Ejemplo:
   crear.sh aguado 546 (creará el fichero aguado con 546 K de tamaño).
   crear.sh panadero (creará el fichero panadero con 1.024 K de tamaño).
   crear.sh (creará el fichero fichero_vacio con 1.024 K de tamaño).

2) Modificar el ejercicio anterior para que antes de crear el fichero compruebe que no exista. En caso de que exista avisará del hecho por pantalla y creará el fichero pero añadiéndole un 1 al final del nombre (aguado1, por ejemplo). Si también existe un fichero con ese nombre, lo creará con un 2 al final del nombre, así seguiremos hasta intentar el 9. Si también existe un fichero con 9 al final del nombre, avisará del hecho y no creará nada.

3) Crear un script llamado cuenta_atras.sh que nos pida dos números cualesquiera, deberá mostrar por pantalla todos los números entre el mayor de los dos introducidos y el menor. (Así si introducimos 20 1 nos mostrará los números del 20 al 1, si introducimos 56 89 nos mostrará los números del 89 al 56).

4) Script que nos diga por pantalla: Nuestra dirección IP es : xxx.xxx.xxx.xxx

5) Hacer un script que nos diga por pantalla buenos días, buenas tardes o buenas noches según la hora del sistema. (Elegir vosotros las horas de día, tardes y noches).

6) Hacer un script que acepte como parámetro una palabra. El script debe reescribir la palabra por la pantalla, pero cambiando la a por un 1, la e por un 2, la i por un 3, lo o por un 4 y la u por un 5.

7) Hacer un script que acepte como parámetro un número. El script debe avisar por pantalla si no se le pasa ningún parámetro, o si el parámetro que se le pasa no es un número. Una vez comprobado que le ha pasado un número, dibujara por pantalla tantos caracteres @ como indique el número. (Así, si se le pasa como parámetro al script un 12, dibujará por pantalla @@@@@@@@@@@@ (12 veces la @).

8) Script que nos diga por pantalla cuantos usuarios reales tiene nuestro sistema (usuarios que tengan un directorio creado en /home), nos deje elegir de una lista el nombre de uno de ellos, y le realice automáticamente una copia de seguridad de todo su directorio home en /home/copiaseguridad/nombreusuario_fecha. Nombreusuario será el nombre del usuario, y _fecha será un símbolo _ y la fecha actual del sistema. Nos referimos a usuarios normales que tengan creado una carpeta en /home.

9) Hacer un script que nos pida el número de alumnos de una clase. Posteriormente irá pidiendo la nota de cada una de ellos para la asignatura de ISO. Al final indicará el número de aprobados, el número de suspensos y la nota media.

10) Hacer un script que nos pida por pantalla nuestro nombre, y nos diga cuantas letras tiene.

11) Hacer un script que admita como parámetros el nombre de dos ficheros. El script debe avisar si los parámetros pasados no existen como ficheros, o si no se le pasan parámetros, o si solo se pasa un parámetro. Una vez comprobados que se le han pasado dos ficheros, el script debe mostrar por pantalla el tamaño en KB de cada uno de ellos.

12) quita_blancos.sh. Este script debe automáticamente, renombrar todos los ficheros del directorio actual de modo que se cambien todos los espacios en blanco de los nombres de los ficheros por subrayados bajos (el carácter _). Así, si en el directorio indicado como parámetro hay un fichero como Mi carta de amor al ejecutar el script cambiará su nombre por Mi_carta_de_amor. Esto debe hacerse automáticamente para todos los ficheros del directorio actual que tengan espacios en blanco en el nombre.

13) lineas.sh. Script que aceptará tres parámetros, el primero será un carácter cualquiera, el segundo un número entre 1 y 60 y el tercero un número entre 1 y 10. El script debe dibujar por pantalla tantas líneas como indique el parámetro 3, cada línea formada por tantos caracteres del tipo parámetro 1 como indique el número indicado en parámetro 2. El script debe controlar que no se le pase alguno de los parámetros y que los números no estén comprendidos entre los límites indicados. Ejemplo: ./líneas.sh k 20 5 (escribirá 5 líneas, cada una de ellas formadas por 20 letras k.

14) Crear un fichero con nombre palabra.txt y escribir en el mismo una palabra en una única línea. Modificar el script anterior, de modo que no se le pase el carácter. En su lugar se usará la palabra leída de dicho fichero palabra.txt. Ejemplo: ./lineas2.sh 6 5 (escribirá 5 líneas, cada una de ellas formadas por repetir 6 veces la palabra que este escrita en palabra.txt).

15) Tenemos un directorio que contiene, entre otras cosas, scripts de shell. Se desea modificarlos, insertando entre su primera y segunda línea el copyright del autor, la fecha y el nombre del fichero. Por ejemplo, el script hola_mundo.sh

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

quedaría:

```
#!/bin/bash
# FILE: hola_mundo.sh
# (c) Ango. You can redistribute this program under GNU GPL.
# mié abr 11 14:30:08 CEST 2007
echo hola mundo
```

El script se usará de la siguiente forma:

pon_cabecera.sh /home/usuario/scripts /home/usuario/licencia.txt

El primer parámetro indica el directorio donde están los scripts con extensión .sh. El segundo es el fichero con el mensaje de copyright (c) (una única línea).