Tema 8
Scripts en Linux (bash)

Autor: Arturo Bernal Mayordomo

Curso 2019/2020

Script de bash

- Fichero con una serie de ordenes bash:
  - Comandos
  - Variables
  - Condicionales, bucles,....
- Al ejecutarlo se realizan, en el orden escrito las órdenes que hay en el fichero.
  - bash script.sh → bash -x para depuración
  - ./script.sh → Requiere permisos de ejecución.
  - script.sh → Si añadimos el directorio al PATH.
- Empiezan por **#!/bin/bash** y acaban con **exit 0**.
- Los comentarios empiezan por **#** y son ignorados hasta el final de la línea.

Variables

- Para darles valor (siempre sin **$** delante)
  - **variable=valor** → el símbolo = siempre pegado
  - **read variable** → valor escrito en el terminal.
  - Las variables no tienen tipo. Empiezan a existir en cuanto se les da un valor
- Para sustituir por su valor (con **$** delante)
  - **echo "valor = $variable"** → Las variables no hace falta concatenarlas con el texto.
  - Si precio=3, y queremos que aparezca el texto 3euros, escribimos **${precio}euros**. Las llaves delimitan el nombre de la variable cuando lo necesitemos.

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**.
  - No hace salto de línea automático como echo.
  - printf “Te llamas %s y tu edad es %d\n” “$nombre” “$edad”
  - **%s** → Se sustituye por un string. **%20s** → Si el string tiene menos de 20 caracteres, te rellena por la izquierda 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.
  - Opción **-v** var → En lugar de imprimir por pantalla, guarda la cadena en una variable.

read

- Podemos hacer un **read** con más de una variable a la vez (separadas por espacio).
  - Lo escrito por el usuario se repartirá entre las variables usando el espacio como separador.
  - Si hay más variables que valores, las últimas quedan sin valor.
  - Si hay más valores que variables, la última variable se queda con el texto restante.
  - Utiliza la variable de entorno $IFS para el separador:
    - **IFS=":" read var1 var2** → Usa **':'** como separador para lo que guarda en cada variable, sólo durante la ejecución de read.

read (II)

- Con el usuario interactuamos mostrándole mensajes por pantalla con **echo**, **printf**, o la opción **-p** de **read**.
  - **read -p “Dime tu nombre: ” nombre**
- Si no se le indica al final del comando una variable, lo escrito por el usuario se guarda en la variable especial **$REPLY**.
- Opciones de read
  - **-r** (recomendada siempre): Ignora caracteres de escape como \ y los interpreta como texto.
  - **-n NUM** → Lee NUM caracteres como máximo. Cuando los ha recibido para de leer.
  - **-s** → Oculta el texto que el usuario escribe.

Mostrar varias líneas de texto

- 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

Parámetros

- Cuando ejecutamos un programa (al igual que a una función), podemos pasarle parámetros.
  - **bash programa.sh 38 “hola”**
- Los parámetros los podemos leer con **$1, $2, $3,...** Estas son variables especiales de sólo lectura.
  - **$0** → Nombre de programa
  - **$#** → Número de parámetros recibidos
  - **$@** → Lista o cadena con todos los parámetros
- A partir de 10, deben ir con llave → **${10}, ${11}**
- Podemos usar una variable que indique el número de parámetro. Ejemplo → **i=2; ${!i}** → igual que ${2}

Comillas

- Las **'comillas simples'**, no sustituyen variables que haya dentro, todo lo muestran de forma literal.
- Las **“comillas dobles”**, si sustituyen variables. Para no sustituir una variable, escapamos **$ → \$**.
- Las **`comillas invertidas`** (acento abierto de valenciano), ejecutan un comando y se sustituyen por la salida del comando.
  - **mkdir `whoami`** → Crea una carpeta con el nombre de mi usuario.
- Utilizar **$(comando)**, con paréntesis, hace lo mismo que `comando`. Ambos se pueden usar dentro de comillas dobles integrados con texto.

Valores devueltos por programas

- Aunque no es obligatorio, puede ser recomendable indicar si nuestro programa ha terminado correctamente o hay un error.
  - Se indica mediante el comando **exit** seguido de un número de finalización (0 → ok, 1 → error).
  - En realidad distinto de 0 indica error. Es decir, que funciona al revés que los condicionales clásicos en programación. Algo así como **0** → **true** y **!= 0** → **false**.

Condiciones con comandos

- Deberíamos saber lo siguiente:
  - Operador **||** (o) → Si valor de la izquierda es "**true**", no se comprueba el de la derecha.
  - Operador **&&** (y) → Justo al revés, izquierda "**false**".
  - Existe operador negación **!**.
- Podemos condicionar la ejecución de un comando, o un grupo de comandos (agrupados con { }) a que el anterior falle o funcione (debe haber separación de espacio entre { } y lo de dentro, y acabar con ';').
  - **cd dir || { mkdir dir && cd dir; }**
  - Si dir no existe, ejecuta mkdir dir, si lo puede crear, entonces entra.
  - Si dir existe, entra, y el paréntesis derecho ni se ejecuta.

Condicionales

- Normalmente encerrados entre [[ ]] (separados con espacio todos los elementos).
- Variables:
  - **[[$variable]]** → Comprueba si existe y el valor no es vacío
- Ficheros:
  - **-e fichero** → True si existe archivo o directorio
- Cadenas:
  - **cad1 = cad2** → True si son iguales
- Números:
  - **num1 -eq num2** → True son iguales
- En los apuntes están todas las posibilidades.

Estructura if

```
If [[ condicion1 ]]
then
    bloque1
elif [[ condicion2 ]]
then
    bloque2
else
    bloque3
fi
```

- elif (else if). Como es obvio **elif** y **else** se pueden omitir si no se necesitan. Siempre se cierra con **fi**.

Estructura if (II)

- 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.
- Podemos utilizar un simple comando (o varios concatenados) como condición:
  - **if ! mkdir hola 2> /dev/null**
  - **if [[! -e “hola”]] && ! mkdir hola 2> /dev/null**
  - **[[$1]] || { echo “Faltan argumentos” >&2; exit 1 }**

Redireccionando la salida

- Redireccionando con >, o con >>, la salida a un archivo, estamos redireccionando los mensajes normales (salida estándar).
- Para redireccionar mensajes de error se utiliza 2> o 2>> (salida de error).
  - **ls dir1 > salida.txt 2> error.txt**
- Para redirigir error y estándar al mismo fichero:
  - **Comando > fichero 2>&1**
- Mostrar mensaje de error en nuestro script:
  - **echo “No existe el archivo” 1>&2**
- Ambos aparecen por defecto en el terminal, pero al redireccionar se tratan de forma diferente.

Operaciones matemáticas (expr)

- Cada número o elemento es un parámetro. Uso:
  - **expr 3 + 4** → Mostrará 7 por pantalla.
  - **expr length “hola”** → Mostrará 4
  - **expr substr “Extintor” 3 2** → Muestra “ti”
- ¡Cuidado con caracteres especiales!
  - Caracteres como *, (,), deben ir escapados *, o entre comillas “*”.
- TODO debe ir separado por espacios, paréntesis incluidos.
- Asignar resultado de operación a variable.
  - **res=`expr 3 + 3`**
  - **res= $(expr 3 + 3)**

Operaciones matemáticas (let)

- No opera con cadenas, sólo números.
- Síntaxis más fácil, toda la operación va entre comillas.
- Asigna la variable dentro de la operación. No requiere ``, ni $().
- No hay que escapar caracteres.
- Para leer valor de una variable no hace falta $, aunque se puede usar. Para asignar valor, **nunca**.
- No hace falta separar los elementos por espacio.
- **let “a=(4+3)*7”**
- **let a++** y **let a--** (sin comillas esto)

Variables avanzadas

- Las variables siempre por defecto guardan strings, pero tienen atributos extra que podemos establecer con **declare**.
- **declare -i variable** → La variable sólo podrá guardar enteros. **variable=”4+5”** hará la operación y guardará 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.
- **declare -p variable** → Te muestra el contenido de la variable y sus atributos.

Metacaracteres

- Debe quedar claro: **No** son expresiones regulares. La comparación de cadenas con **if**, **while** o **case** los utilizan → [[hola = h*a]].
- * → 0 o más caracteres cualquiera
- **?** → 1 carácter cualquiera
- **[ ]** → 1 carácter a elegir (se admiten rangos)
- **[!]** → 1 carácter que no esté entre los incluidos.
  - **[abc]*.???** → archivos que empiecen por 'a', 'b' o 'c' y acaben con una extesión de 3 caracteres.
- Con la opción **-e**, el comando **echo** permite:
  - Uso de "**\n**" (salto de línea) y "**\t**" (tabulación)
- La opción **-n** elimina el salto de línea final de **echo**.

Case

- Equivale a la estructura **switch** de otros lenguajes.
- Puede comprobar números y cadenas (sin comillas). Admite metacaracteres.
- **;;** en última instrucción o en la línea de abajo equivale a **break**

```
case $resp in
    [sS]*)  #Si la respuesta empieza por s o S
        echo “Ha elegido continuar”;;
    [nN]*)  #Si la respuesta empieza por n o N
        echo “Ha elegido no continuar”
        exit 0;;
     *)
        echo “Respuesta no válida” 1>&2;;
esac
```

While

- Sintáxis muy similar a **if**. Utiliza el mismo tipo de condiciones.

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

- En los apuntes se explica como recorrer las líneas de un fichero o la salida de un comando con **while**.

For … in ...

- Esta estructura sería comparable a un **foreach**.
- Recibe una lista elementos o una variable con esos elementos (como cadena de texto). Utiliza el espacio y el salto de línea como separadores.

```
for num in 1 2 3 4 5 6 7 8 9
do
        echo "Repetición número $num"
done
```

seq y rangos

- **seq 1 9** → Genera los números del 1 al 9
- **seq 1 2 9** → Igual, pero de 2 en 2 (números impares sólo).
- **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 (( ))

- Es la estructura más similar a la que tienen lenguajes tipo C y sus derivados.
- Las condiciones son más familiares.
- Para leer variables no hace falta **$**, pero puede usarse, para darle valor nunca.

```
for (( i = 1; i <= 9; i++ ))
do
        echo "Repetición número $i"
done
```

Break y continue

- Se debe evitar abusar de ellos, pero en alguna ocasión pueden ser útiles.
- Si ejecutamos **break**, se sale del bucle inmediatamente, sin comprobar nada y sin terminar de ejecutar acciones posteriores dentro del mismo.
- Ejecutando **continue**, simplemente nos saltamos el resto de instrucciones dentro del bucle, y pasamos directamente a la próxima iteración.
  - Si estamos en un for, automáticamente se ejecuta la instrucción de incremento, o se pasa al siguiente elemento de la lista.

Separar instrucciones. Instrucciones en más de 1 línea.

- Normalmente es el salto de línea es lo que separa a una instrucción de otra.
- Si queremos más de una instrucción en una línea, debemos separalas por punto y coma ';'.
- Si una instrucción la queremos dividir en más de una línea, debemos escapar el salto de línea, es decir, poner el caracter '\' antes del salto de línea.

```
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."
```

Arrays

- Mismo concepto que en la mayoría de lenguajes.
- Array con valores inciales:
  - **array=(“v1” 5 “v2” 98)** #Separados por espacio
- No hace falta inicializar variable como array.
- Se le puede dar valor a sus posiciones directamente. Recomendado de forma seguida desde la 0:
  - **array[0]=5**
- Para leer una posición del array, obligatoriamente debemos utilizar llaves.
  - **${array[0]}**

Arrays (II)

- **${array[@]}** → Devuelve todos los elementos del array separados por espacio.
- **${#array[@]}** → Devuelve el número de elementos que contiene el array.
- **unset array[3]** → Borra el elemento 3.
- Podemos llenar el array con la salida de un comando, los espacios y saltos de línea actuaran como separadores de elementos:
  - **array=($(cat archivo.txt | cut -d ';' -f 2))**

Salto de línea como separador

- Puede que necesitemos no utilizar el espacio como separador de elementos, sino sólamente el salto de línea.
- Esto es útil si queremos guardar valores de un archivo por ejemplo.

  set -f; OLDIFS=$IFS; IFS=$'\n'

  a=($(cat file))

  set +f; IFS=$OLDIFS
- **$'\n'** → Si no ponemos dolar delante no lo interpreta como salto de línea, sino de forma literal.
- La última instrucción vuelve a poner el separador por defecto, como estaba antes de la primera.

El doble paréntesis (( ))

- Además de para el **for**, podemos usarlo más.
- Como sustituto del comando let:
  - **((suma = 12 + 43)); ((suma++))**
- Como sustituto de expr para mostrar el resultado directamente (con $ delante):
  - **echo "4 * 13 es igual a $((4 * 13))"**
- Como sustituto de los corchetes en las condiciones numéricas (mismo formato que con **let,** sólo números).
  - **if (($a < $b))**
  - **while (($i < 10 && $a >= $b))**
- Permite operaciones "ternarias" estilo C:
  - **((mayor = $a > $b?$a:$b))**
- El símbolo $ delante de la variable es opcional en este caso.

Operar con decimales

- Las operaciones de comparación númericas en bash y usando let, expr, (( )), etc. no están preparadas para tratar números con decimales.
- El comando **bc** (basic calculator) nos permitirá hacer las mismas operaciones que los comandos que conocemos, permitiendo el uso de decimales.
- En realidad es un intérprete de un lenguaje de programación matemático, y tiene estructuras como if, while, funciones, etc...
- http://en.wikipedia.org/wiki/Bc_(programming_language)

bc

- Este comando interpreta el código de un archivo, pero podemos usarlo mediante una tubería.
  - **echo “5 / 3” | bc -l**
- Siempre tendremos que usarlo así. Las operaciones lógicas devuelven 0 (false) o 1 (true)
  - **if [[$(echo “4.5<4.6” | bc -l) -eq 1]]**
  - **if (($(echo “4.5<4.6” | bc -l)))**
    - Usando (( )) no hace falta comparar → 0 false, 1 true
- Para ajustar la precisión de los decimales usamos la instrucción **scale** (las instrucciones se separan por ;)
  - **echo "scale=2; 5 * 7 /3" | bc -l** # 11.66
- Para cambiar la precisión de un número ya almacenado:
  - **echo "scale=2; $num/1" | bc -l**

Operaciones con cadenas

- Ninguna de estas operaciones modifica la variable
- **${#cadena}** → Longitud de la cadena de la variable
- **${cadena:3:5}** → Extrae 5 caracteres desde la posición 3 (empieza en la 0).
- **${string#substring}** → Borra la coincidencia más corta desde el principio (ver apuntes)
- **${string##substring}** → Borra la coincidencia más larga desde el principio (ver apuntes)
- **${string%substring}** → Borra la coincidencia más corta desde el final (ver apuntes)
- **${string%%substring}** → Borra la coincidencia más larga desde el final (ver apuntes)

Operaciones con cadenas (II)

- **${string/patron/reemplazo}** → Reemplaza la primera coincidencia en la cadena:
  - cad="Mi carpeta es /home/arturo"
  - echo ${cad/arturo/guest} # Muestra "Mi carpeta es /home/guest"
- **${string//patron/reemplazo}** # Reemplaza todas las coincidencias
  - cad2="archivo.txt, cosas.txt, casa.txt, cielo.doc"
  - echo ${cad2//.txt/.dat} # Muestra "archivo.dat, cosas.dat, casa.dat, cielo.doc"

Operaciones con cadenas (III)

- **${var/#patron/cadena}** → Sustituye sólo si el patrón es el comienzo de la cadena
- **${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.

Funciones

- Las funciones se utilizan tanto para separar código que podamos reutilizar más de 1 vez en el programa como para hacer el programa más legible.

```
funcion () {
        echo "Soy una función"
}
```

- Se definen sin parámetros con el formato anterior. Y se les llama poniendo sólo el nombre, como si fuera un comando.
  - **funcion** → Llamada a la función anterior.

Funciones (II)

- Para pasar parámetros a la función, se realiza igual que cuando llamamos a un script. Y dentro de la función se reciben igual ($1, $2,....)

```
funcion () {
        echo "He recibido $# parámetros"
        echo "Parametro 1: $1"
}
funcion "parámetro 1"
```

- Las funciones tienen una instrucción **return**, pero están limitadas a devolver un número entre 0 y 255, y nada más. Como cualquier comando con **exit**.
- Cuando la función ha terminado se accede al valor que ha devuelto mediante la variable **$?**.

Funciones (III)

- Las variables que utilicemos en una función, por defecto son todas **globales**.
  - Podemos usar esto para limitaciones del **return**.
- Para indicar que una variable es **local** a la función, pondremos la palabra **local** delante de la misma la primera vez que le demos valor, o usar **declare**.

```
funcion () {
        local par1=$1
        echo "par1 vale $par1 y dejará de existir en breve."
}
```

- Las funciones en bash pueden llamarse igual que otra variable, ya que se utiliza el dolar ($) o la ausencia de el para diferenciarlas.

Funciones (IV)

- Si la salida de una función es el resultado de un comando no hace falta return:

```
isNum() {
    [[ $1 =~ ^[0-9]+$ ]]
}
isNum $var || echo “Error: no es un número”
```

- Podemos redirigir por defecto todos los mensajes de una función por la salida de error:

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

Source

- Podemos declarar funciones o variables en un archivo de script separado y utilizar source para ejecutarlo y cargar dichos datos en memoria.
  - Básicamente estamos dividiendo el programa en módulos.
- La diferencia entre usar source y lanzar otro script con bash, es que en la segunda opción, se crea una instancia de bash aparte, y las variables y funciones desaparecen cuando termina el script.
  - **source** → ejecuta el script en la instancia actual de bash.
  - Ejemplo: **source funciones.sh**

Caracteres de colores

- Podemos cambiar el estilo de texto con una serie de códigos.
- Para que funcionen debemos usar la opción **-e** del comando echo (lo mismo que para usar \n y \t)
- La forma más cómoda de usar estos estilos es usando variables tal como viene en los apuntes.
- Cada vez que cambiemos un estilo y terminemos de usarlo, deberíamos poner el código que quita los estilos **'\e[0m'**, o todo continuará con ese estilo.
- **echo -e "\e[0;31mTexto de color rojo\e[0m"**
- **echo -e "${Red}Texto de color rojo${Color_Off}"**

Reposicionando el cursor

- Por ahora cada vez que escribimos, siempre aparece en la línea siguiente a la actual.
- Podemos posicionar el cursor en la parte de la consola que queramos y escribir ahí.
- Son códigos especiales y se usan con la opción **-e**.
- **\033[s** → Guarda la posición actual del cursor. Si no le indicamos posición antes, guarda la línea siguiente.
- **\033[<fila>;<columna>f** → Mueve el cursor a la posición indicada (empiezan en 1) → **\033[3;1f**
- **\033[u** → Vuelve a la última posición guardada.
- **echo -e -n "\033[2J\033[1;1f\033[s\033[4;4f\e[0;31mTexto de color rojo\e[0m\033[u\033[K"**

Reposicionando el cursor (II)

- **\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