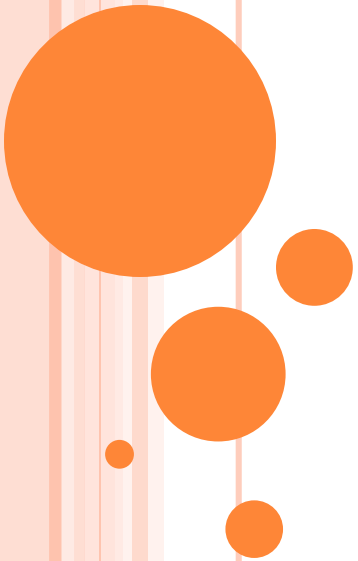
Administración de Sistemas



Interacción y programación para la administración de sistemas: Estándar IEEE std 1003.1 (Posix): shell y herramientas II

Índice

- Variables
- Variables predefinidas
- Sustitución de comandos
- Comando Test
- Estructuras de control condicionales
- Bucles
- Entrada/Salida
- Funciones
- Depuración

Variables

- Las variables no tienen tipo, son “strings”
  - Algunos strings se pueden interpretar como números por algunos comandos
- No hace falta declarar las variables
  - Si se lee el valor de una variable sin asignar, su valor es el string vacío
- El nombre de las variables debe:
  - empezar por una letra o _
  - seguida por cero o mas letras, números o _ (sin espacios en blanco)

VARIABLES

- Asignar un valor:

  nombre_variable=valor

  (OJO: Sin espacios alrededor del '=')

  Por ejemplo:

  ```
  $> una_variable=hola
  $> un_numero=15
  ```

- Acceder a las variables: *$nombre_variable*

  ```
  $> var=HOLA
  $> echo $var
  HOLA
  ```

VARIABLES

- El $ es una “sustitución de variable”

  $> var=HOLA

  $> echo $var

  HOLA

  1. Sustitución de la variable: echo HOLA
  2. Ejecución: HOLA

- ¿Qué pasa si ejecuto …?

  $> comando=ls

  $> $comando

Variables

- Los espacios en blanco se tienen en cuenta:
  - usar comillas para incluirlos en la variable

```
$> nombre="Pepe Pota"
$> echo $nombre
Pepe  Pota
```

- Se utilizan llaves si la variable es parte de una palabra mayor: *${nombre variable}*

```
$ nombre="Pepe Pota"
$ echo ${nombre}mo
Pepe Potamo
```

VARIABLES

- Uso de variables:
  - programación scripts shell
  - control del entorno de ejecución del shell (*PATH*, *HOME*, …)
- Dos tipos:
  - variables locales: visibles solo desde el shell actual. Se pueden mostrar ejecutando set
  - variables globales o de entorno: visibles en todos los shells. Se pueden mostrar con env o printenv

Variables de entorno

- Para ver las variables de entorno, **env** o **printenv**

  HOME : directorio base del usuario

  SHELL : el programa ejecutable para el shell que se utiliza

  UID : el id del usuario en curso

  USER, USERNAME : el nombre del usuario

  TERM : el tipo de terminal en uso

  DISPLAY : la pantalla de X-Windows

  PATH : El path de ejecución del usuario

  PS1/PS2/…: los “prompts” de comandos

  PWD: el directorio actual

  MANPATH : el path para las páginas del manual

Variables predefinidas

- Es posible pasar parámetros a un script: los parámetros se recogen en las variables **$1** a **$9**

Variable	Uso
$0	el nombre del script
$1 a **$9**	parámetros del 1 al 9
${10}, ${11}, …	parámetros a partir del 10
$#	número de parámetros
$*	todos los parámetros (como una secuencia)
$@	todos los parámetros (explícitamente separados)

- Comando **shift** (sin parámetros). Mueve todos los parámetros una variable a la izquierda ($1 pierde su valor y coge el de $2, $2 el de $3 y así hasta el $9).
  - También se puede utilizar ***shift n***, lo mismo pero desplazando **n**

Variables predefinidas

- Otras variables:
- $?
  - Contiene el estado de salida (ejecución correcta = 0, o errónea != 0) del último comando (o proceso) ejecutado
- $$
  - Contiene el id del proceso en curso
- $!
  - Contiene el id del último proceso enviado como tarea de fondo

Variables predefinidas

- Ejemplo ($$):

```
$> cat shellid.sh
#!/bin/bash
echo "Shell ejecutando el script, PID = $$"
$> echo "PID actual = $$"
$> PID actual = 6919
$> bash shellid.sh
Shell ejecutando el script, PID = 26824
$> . shellid.sh
Shell ejecutando el script, PID = 6919
```

Sustitución de comandos: $(…)

- `$(comando)` o `` `comando` ``
  - Ejecuta comando y reemplaza `$(comando)` por su salida estándar
  - Comando se ejecuta dentro de un subshell y la salida de la sustitución es la salida standard del comando
  - Ejemplo:

```
as@as $ dirs=$(ls /)
as@as $ echo “$dirs”
bin
boot
dev
…
var
```


fuente: https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html

Comando test

- Estado de la salida
  - Estado en el que terminó la ejecución de un comando o script (recogido en la variable **$?**)
  - Ejemplo:

```
$> ls /bin/ls
/bin/ls
$> echo $?
0 # estado correcto
$> ls /bin/ll
ls: /bin/ll: No hay tal fichero o directorio
$> echo $?
1 # estado incorrecto
```

Comando test

- Comandos que devuelven siempre el mismo estado de salida :
  - **true** (sin parámetros): estado de salida = 0 (correcto).
  - **false** (sin parámetros): estado de salida = 1 (erróneo).
- Comando **exit <estado_salida_opcional>**
  - Termina la ejecución de un script y, si se explicita, devuelve un estado de salida determinado; si no, queda el estado de salida del último comando ejecutado.

Comando test

- Da un estado de salida según un conjunto de condiciones.
- Tipos de condiciones
  - Longitud de un string.
  - Comparación de dos strings.
  - Comparación de dos números.
  - Verificar el tipo de un fichero.
  - Verificar los permisos de un fichero.
  - Combinar condiciones.
- Dos formatos equivalentes

test *expr* : **test** $1 = hola

[*expr*] : [$1 -gt $2 -o $1 -eq $2]

*** (ojo a los espacios en blanco entre los corchetes y la expresión)

Comando test

- Ejemplo:

```
if [   "$1" = "hola" ]
then
   echo "Hola a ti tambien"
else
   echo "No te digo hola"
fi
if [   $2   ]
then
   echo "El segundo parametro es $2"
else
   echo "No hay segundo parametro"
fi
```

- En el segundo if, **test** da estado de salida 0 si $2 tiene algún valor; !=0 si la variable no está definida o contiene null ("")

COMANDO TEST

- Expresiones sobre strings

-z string	length of **string** is 0
-n string	length of **string** is not 0
string1 = string2	if the two **strings** are identical
string != string2	if the two **strings** are NOT identical
string	if **string** is not NULL

- Expresiones sobre enteros

int1 -eq int2	first int is equal to second
int1 -ne int2	first int is not equal to second
int1 -gt int2	first int is greater than second
int1 -ge int2	first int is greater than or equal to second
int1 -lt int2	first int is less than second
int1 -le int2	first int is less than or equal to second

COMANDO TEST

- Expresiones sobre ficheros

-r file	file exists and is readable
-w file	file exists and is writable
-x file	file exists and is executable
-f file	file exists and is a regular file
-d file	file exists and is directory
-h file	file exists and is a symbolic link
-c file	file exists and is a character special file
-b file	file exists and is a block special file
-p file	file exists and is a named pipe
-u file	file exists and it is setuid
-g file	file exists and it is setgid
-k file	file exists and the sticky bit is set
-s file	file exists and its size is greater than 0

- Operadores lógicos

!	reverse the result of an expression
-a	AND operator
-o	OR operator
(expr)	group an expression, parentheses have special meaning to the shell so to use them in the test command **you must quote them**

Comando test

- Ejemplos:

```
$> test -f /bin/ls -a -f /bin/ll ; echo $?
1
$> test -c /dev/null ; echo $?
0
$> [ -s /dev/null ] ; echo $?
1
$> [ ! -w /etc/passwd ] && echo "No puedo escribir"
No puedo escribir
$> [ $$ -gt 0 -a \( $$ -lt 5000 -o -w file \) ]
```

Estructuras condicionales

- Comandos simples :
  - **comando1 && comando2** : el segundo comando solo se ejecutará si el primero devuelve un estado de salida = 0, es decir, ejecución correcta.
  - **comando1 || comando2** : el segundo comando solo se ejecutará si el primero da lugar a un estado de salida erróneo, es decir, != 0.
- Ejemplo con &&:

```
$ ls /bin/ls && ls /bin/ll
/bin/ls
ls:  /bin/ll:  No hay tal fichero o directorio
$  echo  $?
1
$ ls /bin/ll && ls /bin/ls
ls:  /bin/ll:  Non hay tal fichero o directorio
$  echo  $?
1
```

Estructuras condicionales

- Comando **if**: Solo chequea el estado de salida de un comando, no existe el concepto de "condición"

```
if comando1
then
   comando
   comando...
[elif comando2
then
   comando
   comando...]...
[else
   comando
   comando...]
fi
```

- También suele escribirse: **if** *comando1* **; then**

Estructuras condicionales

- Ejemplo

```
hour=$(date | cut -c12-13)
if [ "$hour" -ge 0 -a "$hour" -le 11 ]
then
  echo "Buenos dias"
elif [ "$hour" -ge 12 -a "$hour" -le 17 ]
then
  echo "Buenas tardes"
else
  echo "Buenas noches"
fi
```

Estructuras condicionales

- Comando **case** (patrones de sustitución de ficheros, y adicionalmente el operador "|")

```
case valor in
   patron 1 )
      comandos si value = patron 1
      comandos si value = patron 1  ;;
   patron 2 )
      comandos si value = patron 2  ;;
   *)
      comandos por defecto  ;;
esac
```

- *patron* puede incluir comodines y el símbolo adicional "|" usado como OR.
- Si *valor* no coincide con ningún patrón se ejecutan los comandos después del *)
  - esta entrada es opcional

Estructuras condicionales

- Ejemplo:

```
#!/bin/bash
echo -n "Respuesta: "
read  RESPUESTA
case $RESPUESTA in
  S* | s*)
    RESPUESTA="SI" ;;
  N* | n*)
    RESPUESTA="NO" ;;
  *)
    RESPUESTA=“PUEDE" ;;
esac
echo $RESPUESTA
```

Bucles

- Comando **for**

```
for variable in palabra1 palabra2 ... palabraN
do
   lista_de_comandos
done
```

- Ejemplo

```
LISTA="10 9 8 7 6 5 4 3 2 1"
for var in $LISTA
do
   echo $var
done
```

Bucles

- Ejemplo

```
count=0
for param in "$@" # para cada parámetro
do          # visualiza el numero de parámetro y su valor
  count=$((count+1))
  echo "el parametro $count es $param"
done
```

- Ejemplo (recorrer los ficheros *.bak de un directorio):

```
dir="/var/tmp"
for file in $dir/*.bak
do
  rm -f $file
done
```

Bucles

- Comando **while** :

```
while comando
do
   lista_de_comandos
done
```

- Ejemplo :

```
count=10
while [ $count -ge 0 ]
do
   echo $count
   count=`expr $count - 1`
done
```

- Comando **until** :

```
until comando
do
   lista_de_comandos
done
```

- Ejemplo :

```
count=10
until [ $count -lt 0 ]
do
   echo $count
   count=`expr $count - 1`
done
```

Bucles

- Comando **break** : sale inmediatamente de un bucle, o de un número definido de bucles anidados (for, while o until).

  break

  break *número_de_bucles*

- Comando **continue** : sale de la iteración en curso para comenzar la siguiente en el mismo bucle, o en bucles anidados si parámetro.

Bucles

- Los parámetros $* y $@ solo se diferencian en los bucles si van entrecomillados:
  - "$*"
    - se expande a una sola palabra, conteniendo todos los parámetros y con el valor de cada parámetro separado por el primer carácter de la variable especial IFS (por defecto, un espacio)
  - "$@"
    - cada parámetro se expande a una palabra separada; los parámetros entrecomillados se consideran uno solo aunque lleven espacios
- La variable predefinida IFS contiene los caracteres separadores. (por defecto: espacio y tabulación).

Bucles

- Ejemplo:

```
#!/bin/bash IFS=":"
for par in $*
  do echo "Parametro es:   $par"
done
echo    "====================================="
for par in $@
  do echo "Parametro es:   $par"
done
echo    "====================================="
for par in "$*"
  do echo "Parametro es:   $par"
done
echo    "====================================="
for par in "$@"
  do echo "Parametro es:   $par"
done
```

Bucles

- Ejemplo:

```
$> bash parms3.sh hola "como estas hoy?" bien gracias
Parametro es:   hola
Parametro es:   como
Parametro es:   estas
Parametro es:   hoy?
Parametro es:   bien
Parametro es:   gracias
==================================
Parametro es:   hola
Parametro es:   como
Parametro es:   estas
Parametro es:   hoy?
Parametro es:   bien
Parametro es:   gracias
==================================
Parametro es:   hola como estas hoy? bien gracias
==================================
Parametro es:   hola
Parametro es:   como estas hoy?
Parametro es:   bien
Parametro es:   gracias
```

Funciones

- Podemos definir funciones en un script de Shell de 2 maneras:

```
funcion() {
   comandos
}
```

```
function funcion {
     comandos
}
```

- y para llamarla:

```
funcion p1 p2 p3
```

- Siempre hay que definir una función antes de llamarla

Funciones

```
#!/bin/bash
# Definicion de funciones
funcion1() {
comandos
}
funcion2() {
comandos
}
# Programa principal
funcion1 p1 p2 p3
```

Funciones: paso de parámetros

- La función referencia los parámetros pasados por posición, es decir, $1, $2 . . . , y $* para la lista completa:

```
$ cat funcion1.sh
#!/bin/bash
funcion1() {
echo "Parametros pasados a la funcion: $*"
echo "Parametro 1: $1"
echo "Parametro 2: $2"
}
# Programa principal
funcion1 "hola" "que tal" adios
$
$ bash funcion1.sh
Parametros pasados a la funcion: hola que tal adios
Parametro 1: hola
Parametro 2: que tal
```

Funciones: variables locales

- Es posible definir variables locales en las funciones:

```
$ cat locales.sh
#!/bin/bash
testvars() {
   local localX="localX en funcion"
   X="X en funcion"
   echo "Dentro de la funcion: $localX, $X, $globalX"
}
# Programa principal
localX="localX en main"
X="X en main"
globalX="globalX en main"
echo "Dentro de main: $localX, $X, $globalX"
# Llamada a la funcion
testvars
echo "Otra vez dentro de main: $localX, $X, $globalX"
```

FUNCIONES: VARIABLES LOCALES

```
$ bash locales.sh
Dentro de main: localX en main, X en main, globalX en main
Dentro de la funcion: localX en funcion, X en funcion,
globalX en main
Otra vez dentro de main: localX en main, X en funcion,
globalX en main
```

FUNCIONES: RETURN

- Después de llamar a una función, $? tiene el código se salida del último comando ejecutado.
- También se puede poner de forma explícita usando ***return*** (permite devolver un entero entre 0 y 255)

```
#!/bin/bash
funcion2() {
if [ -f /bin/ls -a -f /bin/ln ]; then
     return 0
else
     return 1
fi
}
# Programa principal
if funcion2; then
     echo "Los dos ficheros existen"
else
     echo "Falta uno de los ficheros - adios"
     exit 1
fi
```