IV Jornadas Sistema Operativo Linux

Programando Scripts en Bourne Shell

Andrés J. Díaz <ajdiaz@mundo-r.com> ¿<ajdiaz@gpul.org>?

¿Qué es un Shell?

Definición técnica:

- «El shell es una parte del SO encargada de validar los comandos introducidos por parte del usuario, para posteriormente enviar las órdenes a otra parte del sistema operativo para su ejecución.»

Definición «cutre»:

- El programita que traduce lo que tecleamos en cosas que entienda el sistema operativo.

¿Qué es un Shell tipo Bourne?

Tipos de shell en UNIX:

- Bourne (sh)
  - Stephen Bourne 1979
- C (csh)
  - Bill Joy 1980
- Korn (ksh)
  - David Korn 1980

¡Y Bash!

- «Bourne Again Shell» (Otro shell Bourne)
- Proyecto GNU
- Estándar IEEE POSIX

El Bourne Shell

Dos estilos de Shell Bourne:

- Estilo BSD
- Estilo SysV

Pero con POCAS diferencias:

BSD:

- echo -n "sinretornodelinea"

SysV:

- echo "sinretornodelinea\c"

El C Shell

Bill Joy, Univ. de Berkeley (1980)

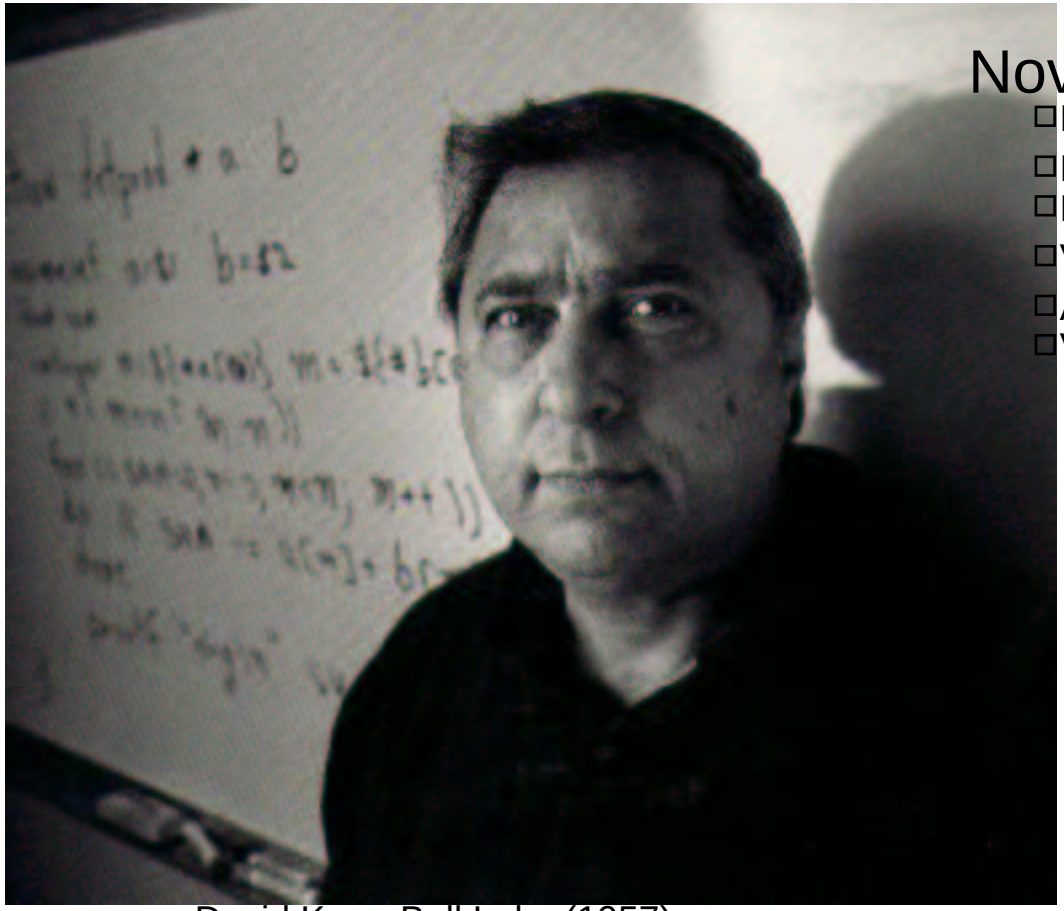
Novedades

- Mejoras en interactividad
- Expansión de tildes (~)
- Expansión de llaves ({a,b,...,n})
- Builtins: pushd, popd, logout...

Además...

- ¡Creador de VI!

El Shell de Korn



David Korn, Bell Labs (1957)

Novedades

- Expresiones aritméticas
- Historial de comandos
- Más expansiones de parámetros
- Vectores («Arrays»)
- Alias
- Variables enteras

Bourne Again Shell (BASH)

¿Otro shell?

- Proyecto GNU, por lo tanto libre
- Mejoras de csh y de ksh
- Más funcionalidad

Mejoras con respecto a Korn Shell (ksh88)

- for aritmético: for ((expr1; expr2; expr3)); do list; done
- de acuerdo con POSIX
- «tilde expansion»
- sustitución de procesos con tuberías
- expansión del «prompt»
- redirección: &> (stdout y stderr), <<<, [n]<&palabra-, [n]>&palabra-
- «!»: extensión de historial del estilo csh
- «**»: operador de exponenciación
- «Arrays» de tamaño ilimitado

¿Por qué un Shell tipo Bourne?

Creado para ser programable

- Programación estructurada
- Muy mejorado con BASH
- Está más extendido
- Csh NO es un buen shell de programación

- Es mucho más cómodo :-)

Posee estructuras de control «builtin»

- Sentencias de bifurcación (if, case ...)
- Bucles (for, while ...)
- Diferentes tipos de variables

y... ¡Funciones!

¿Por qué NO CSH?

No soporta descriptores directos a ficheros:

- BASH: cmd 2>/dev/null
- CSH: (cmd > /dev/tty) >& /dev/null

Ortogonalidad de comandos (i.e. sleep 1 |while)

- BASH: espera por un «do ... done»
- CSH: error en while y se procesa ¿sleep |?

Espupideces varias sin nombre científico ;-)

- % kill -1 ‘cat foo‘
  - ERROR!!!! --> ‘cat foo‘: Ambiguous.

- % /bin/kill -1 ‘cat foo‘
  - ¡¡¡¡¿¿¿¿¿OK?????!!!!

¿Para qué programar en Shell?

Nos ayuda en nuestra vida diaria :-)

- Evitar la repetición sistemática de comandos
- Personalizar utilidades

Por pura comodidad

- Crear utilidades automáticas que hagan labores de mantenimiento o instalación
- Autodetección de hardware

Para trastear un poco

- Meterse con los que usan Perl para hacer trivialidades ;-)
- Juegos para BOFHs (expulsar usuarios...)

Sentencias y comandos

Ejecutar comandos

- Llamada directa al ejecutable (/bin/ls o ls si está en el PATH)
- Usando un alias (alias ll='ls -l')
- Mediante una variables (LS=/bin/ls; $LS)

Separar comandos

- Cada línea un comando
- En la misma línea separado por punto y coma (;)

Comandos y expresiones

Listas de comandos

- En el propio entorno shell: { comando1; comando2; ...; comando n }
- En un subshell: (comando1; comando2; ...; comando n)

Expresiones Aritméticas

- ((expresión))
- variable=$((expresión))
- variable=$[expresión] (¡NO!)

Expresiones Condicionales

- [[expresión]]
- [expresión]
- test expresión

«Builtins» versus Comandos

Diferencias:

- Comandos son ejecutables independientes del shell
- Builtins con funciones propias de shell

Builtins:

- [[ ]]
  - Evalúa expresiones
- (( ))
  - Evaluación aritmética
- :
  - Sentencia vacía
- .
  - Lee y ejecuta ficheros

Expresiones Aritméticas (I)

var++ var--
- Post-incremento y post-decremento

++var --var
- Pre-incremento y pre-decremento

- +
- Signos negativos y positivos

! ~
- Negación lógica y bit a bit

**
- Exponenciación

* / %
- Producto, división y módulo

+ -
- Suma y resta aritmética

Expresiones Aritméticas (II)

<< >>
- Desplazamiento de bits

<= >= < >
- Comparaciones lógicas

== !=
- Igualdad y no igualdad

&
- Y binario (AND)

^
- O-Exclusivo binario (XOR)

|
- O binario (OR)

&&
- Y Lógico

Expresiones Aritméticas (y III)

||
- O Lógico

?:
- Evaluación condicional («si en línea»)

= *= /= %= += -= <<= >>= &= ^= |=
- Asignación

,
- Separador de sentencias

[base#]n
- Indicador de base numérica
  - i.e: 16#f --> 15

0xn, 0Xn
- Indicador de base hexadecimal

0n
- Indicador de base octal

Comparaciones (I)

Numéricas:

- [$a -eq 1] --> a = 1
- [$a -ne 1] --> a <> 1
- [$a -lt 1] --> a < 1
- [$a -gt 1] --> a > 1
- [$a -le 1] --> a <= 1
- [$a -ge 1] --> a >= 1

Cadenas:

- [$a = ’hola’] --> a = «hola»
- [$a != ’hola’] -> a <> «hola»
- [-z $a] -------> a = «»
- [-n $a] -------> a <> «»
- [$a] ----------> a <> «»

Comparaciones (y II)

Ficheros:

- [a -ef b] --> el fichero «a» tiene el mismo dispositivo e inodos que «b»
- [a -nt b] --> «a» es más moderno que «b»
- [a -ot b] --> «a» es más antiguo que «b»
- [-b fich] --> «fich» existe y es un fichero especial de bloques
- [-c fich] --> «fich» existe y es un fichero especial de carácteres
- [-d fich] --> «fich» existe y es un directorio
- [-e fich] --> «fich» simplemente existe :-)
- [-f fich] --> «fich» existe y es un fichero regular
- [-h fich] --> «fich» existe y es un enlace simbólico
- [-p fich] --> «fich» existe y es una tubería
- [-r fich] --> «fich» existe y tiene permisos de lectura
- [-w fich] --> «fich» existe y tiene permisos de escritura
- [-x fich] --> «fich» existe y tiene permisos de ejecución

Variables (I)

Características:

- No hay tipos de contenido «variables comunistas» :-)
- Locales al script, a menos que se exporten
- Se accede a su contenido con el símbolo dólar ($)
- Identificador «¡CASE SENSITIVE!»
  - Por convenio en mayúsculas si se exportan
- Hay variables «especiales»
- Realmente $ expande un parámetro, no son variables en el sentido estricto.

Asignación:

- VARIABLE=valor
- VARIABLE=’valor’ (no se expande el contenido de valor)
- VARIABLE="valor" (se expande el contenido de valor)

Acceso:

- $VARIABLE

Variables (II)

La sentencia «declare»:

- declara variables con ciertos atributos:
  - exportables: declare -x VARIABLE=valor
  - sólo lectura: declare -r VARIABLE=valor
  - vectores (arrays): declare -a VARIABLE=valor
- análoga a la antigua «typeset»

La sentencia «export»

- export VARIABLE[=valor]
- igual que declare -x
- exporta una variable al entorno padre
- accesible desde el shell que invocó el script

Variables (III)

«export» exporta la variable desde el shell o subshell al entorno

padre.

Variables (IV)

Variables Especiales:

- $$ --> PID del script actual
- $_ --> PID del shell padre
- $? --> Valor de retorno de la última ejecución
- $# --> Número de argumentos
- $! --> PID del último trabajo en segundo plano
- $0 --> Nombre de la llamada al script

- $1, $2, ..., $n --> Argumentos
- $@, $* --> Todos los argumentos

- PS1,PS2...Pn --> Prompts :-)

Variables (y V)

Expresión de parámetros:

- ${parametro} --> Análogo a $parámetro (DEMO)
- ${parametro:-valor} --> Si parametro es vacío, devuelve valor
- ${parametro:+valor} --> Si parametro es no vacío, devuelve valor
- ${parametro:=valor} --> Si parametro es vacío, le asigna y devuelve valor
- ${parametro:?[valor]} --> Si parametro es vacío, muestra el mensaje de error valor

Sustitución de comandos

- ‘comandante‘ --> ejecuta el comando cuyo nombre es el devuelto por comandante (DEMO)
- $(comandante) -> Ídem
  - $(cat fichero) <=> $(< fichero)

Patrones

¿Qué es un patrón?

- Una expresión que engloba un contenido variado
- Al expandirse se consigue todos los comandos que engloba
- Similar a los conocidos «comodines»
- El carácter nulo nunca puede ser expandido
- Los caráceteres que constituyen un patrón deben ser expandidos si no se quieren interpretar como tales.

Patrones en Bourne (y Csh):(DEMO)

- ? --> El interrogante simboliza cualquier carácter, pero sólo uno.
- * --> El asterisco simboliza una ristra de cualquier longitud y de cualquier carácter
- [...] --> Simboliza al menos uno de los carácteres encorchetados

- ¡¡{...} NO ES UN PATRÓN!!

Redirección (I)

Salida estándar

- comando > fichero

Entrada estándar

- comando < fichero

En general:

- comando [n]>{&n | fichero]

Podemos redirigir a

- fichero
- descriptor de fichero (i.e dispositivo)

Redirección (II)

Descriptores de fichero

- 0 --> Entrada estándar
- 1 --> Salida estándar
- 2 --> Salida de errores estándar

- /dev/fd/n --> dirige al descriptor «n»
- /dev/stdin --> Análogo a 0
- /dev/stdout --> Análogo a 1
- /dev/stderr --> Análogo a 2

- /dev/tcp/host/puerto --> redirige a la conexión establecida con el host «host» en el puerto «puerto » mediante tcp. Análogamente para udp.

Redirección (III)

Redirección entre descriptores(DEMO)

- x>&y --> La salida del descriptor x se redirigirá al descriptor (que no fichero) y
- &>&y --> Redirige todos los descriptores de salida al descriptor y (serviría si y es un fichero)
- >y --> Borra el contenido del fichero y

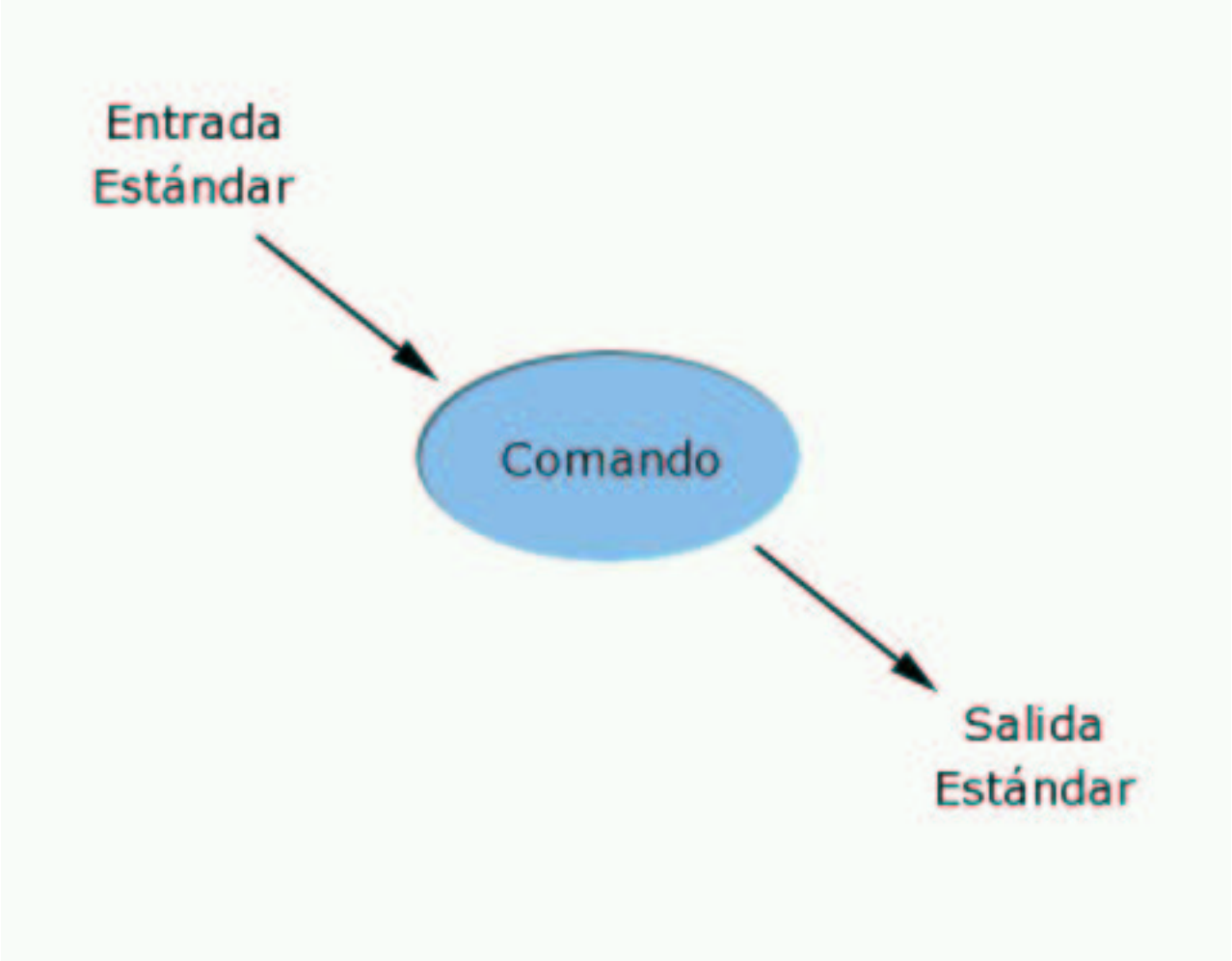
Problema: La redirección a un fichero sobreescribe el mismo
Solución: comando >> fichero

- Anexa la salida del comando «comando» al fichero «fichero»
- Sirve también para trabajar con descriptores

Delimitadores: comando << delimitador(DEMO)

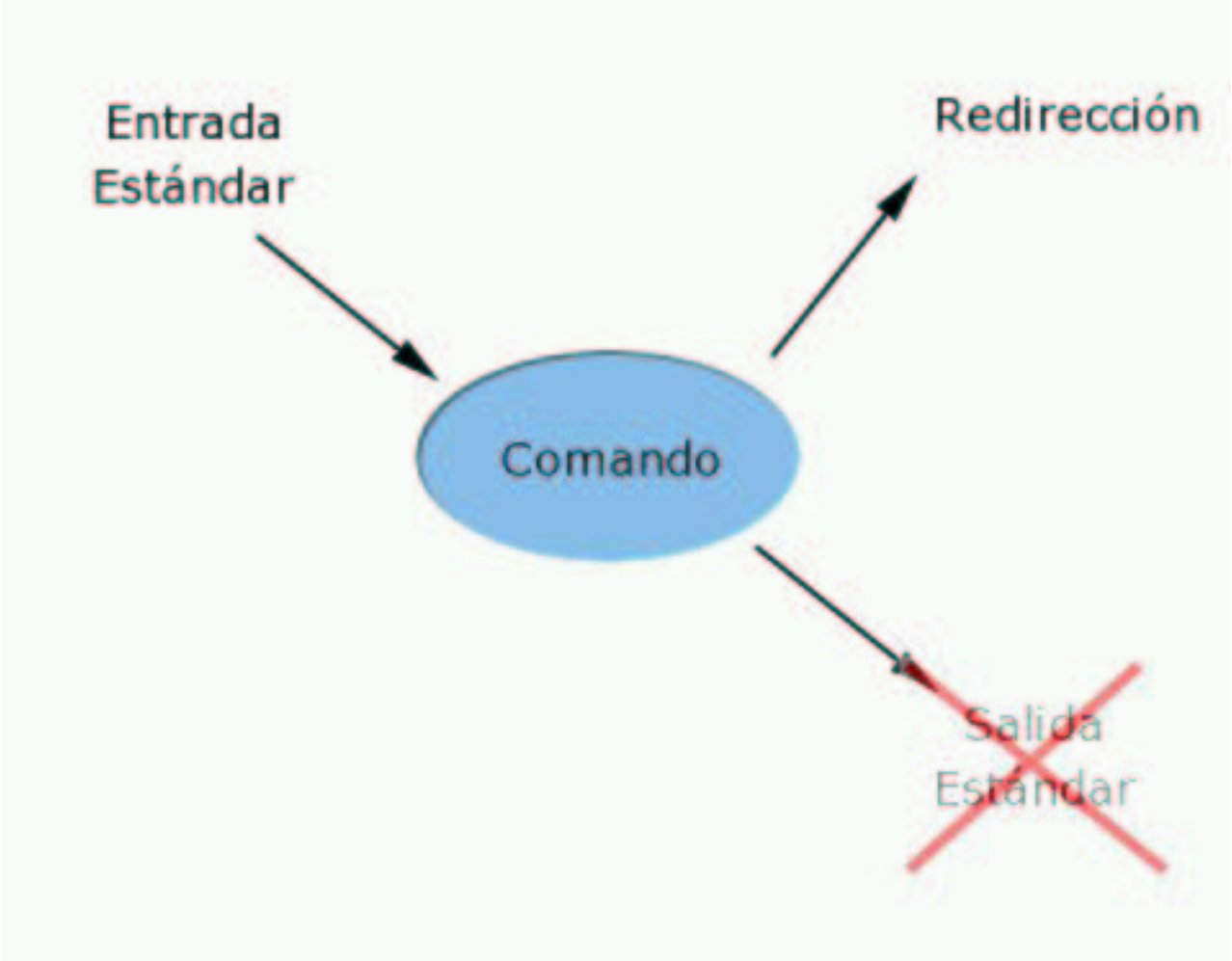
- Lee de la entrada estándar todos los carácteres (expandiéndolos) hasta que se encuentre « delimitador»

Redirección (IV)



Ejecutamos «comando»

Redirección (y V)



Ejecutamos «comando >redirección»

Tuberías

¿Qué hacen?

- Dirigen la salida de un comando a la entrada de otro
- Conectan dos comandos mediante la salida del primero y la entrada del segundo

¿Cómo lo hacen?

- Mediante el carácter «|»

Ejemplos:

- echo ’hola’ | cat
- ls -l | grep .txt

Bifurcación (I)

```
if <condición>
then
    <acción>
[ else
    <acción> ]
fi
```

- Condición debe ser una expresión condicional
- Acción debe ser un comando o una lista de comandos(DEMO)

Bifurcación (II)

```
case <variable> in
    valor)  acción;;
    valor2) acción;;
    ...
    valorn) acción;;
esac
```

- Variable es cualquier variable (haya sido declarada o no)
- ValorX son los valores de estudio de la variable, estos valores pueden ser patrones de shell.
- Acción debe ser un comando o una lista de comandos(DEMO)

Bifurcación (y III)

```
select <variable> in
<lista-de-elementos>
do
    acción
done
```

- Select crea un menú muy rudimentario con cada uno de los elementos de «lista-de-elementos».

  La variable «variable» tomará el valor seleccionado. El prompt de elección vendrá dado por la variable global PS3.(DEMO)

Bucles (I)

```
while <expresión>
do
    <acción>
done
```

- Expresión es una expresión condicional y acción una lista de comandos. Mientras la expresión resulte verdadera la acción o acciones se llevarán a cabo.(DEMO)

Bucles (II)

```
for <variable> in <variable2>
do
    acción
done
```

- El bucle se realizará tantas veces como elementos contenga «variable2», dichos elementos han de estar separados por espacios. La variable «variable» irá tomando consecutivamente los valores de la lista.(DEMO)

Bucles (III)

```
for ((expresión1;expresión2;expresió
n3))
do
    acción
done
```

- Similar a un bucle for de C o Java.
- Sólo funciona en BASH.(DEMO)

Bucles (IV)

```
for i in $(seq 1 10)
do
    acción
done
```

- Forma de Fiesh para convertir un FOR propio de Bourne en otro más similar al de los lenguajes estructurados tradicionales

- Con la aparición del FOR de la diapositiva anterior en BASH, esta forma resulta inútil.

Bucles (y V)

```
until <expresión>
do
    acción
done
```

- Caso inverso de while. Mientras expresión sea falsa se realizarán las acciones del cuerpo del bucle.(DEMO)

Vectores («Arrays») (I)

Arrays en BASH:

- Poco usados, se prefieren cadenas separadas por espacios
- Muy usados a nivel interno: DIRSTACK, COMP_WORDS...
- Son dinámicos :-)
- El índice comienza en cero (0)

Declaración:

- declare -a NOMBRE_ARRAY
- NOMBRE_ARRAY=()
- NOMBRE_ARRAY=(valor1 valor2 ... valorn)

Trabajo con vectores:

- NOMBRE[x]=y --> En la posición x del vector NOMBRE se guarda y
- NOMBRE=() --> Se inicializa el vector NOMBRE y se vacía

Vectores («Arrays») (II)

Acceso a vectores:

- ${NOMBRE[x]} --> Devuelve el valor en la posición x del vector NOMBRE
- ${NOMBRE[@]} --> Devuelve todos los valores del vector, separados por espacios
- ${NOMBRE[*]} --> Ídem, pero filtrados por el contenido de la variable IFS
- ${#NOMBRE[@]} --> Cardinal del vector, número de elementos.

Borrar elementos:

- unset NOMBRE[x]
  - La posición se elimina, no hay datos en NOMBRE[x]
  - El siguiente dato estará en NOMBRE[x+1]
- unset NOMBRE
- unset NOMBRE[@]
- unset NOMBRE[*]
  - Elimina todo el array

Vectores («Arrays») (III)

Cosas «curiosas»:

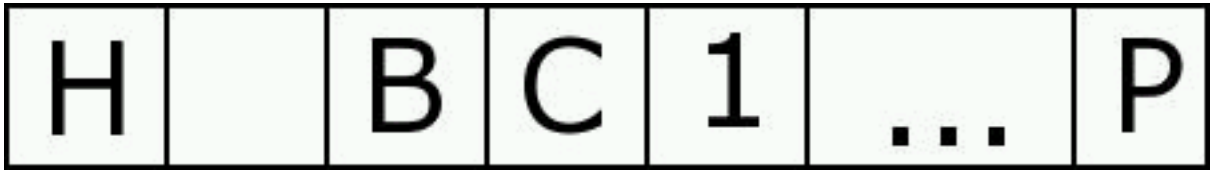
VECTOR[1000001]=’hola’

- VECTOR tiene un sólo elemento
- El elemento está en la posición 1000001

VECTOR[pepe]=’adios’

- El elemento se introduce en la posición 0
- ¡NO DA ERROR!

Vectores («Arrays») (y IV)



$VECTOR

- H

$VECTOR[1]

- H[1]

${VECTOR[1]}

${VECTOR[n]}

- P

${#VECTOR[@]}

- 5

${VECTOR[@]}

- H B C 1 P

Funciones (I)

```
function nombre {
    comandos
}

nombre () {
    comandos
}
```

- La palabra clave function no funciona en todos los tipos de shell BOURNE.
- Los paréntesis sólo se pueden separar por espacios, no por otro carácter.
- Los argumentos a la función serán almacenados en $1, $2..., al abandonar la función se restauran los valores originales del script
- ¡¡¡Hay recursividad!!!(DEMO)

Funciones (y II)

Variables locales

- Se declaran con «declare -l» o bien con «local»
  - ie: local i=0;
- Sólo existen durante el funcionamiento de la función
- Su valor se libera automáticamente al salir de la misma

Ver todas las funciones

- declare -F -> Lista rápida
- declare -f -> Lista detallada

Trucos ;-) (I)

«Parsing» de ficheros:

```
grep '^*.=.*$' <fichero.conf | while read line; do
    variable="${line%%=*}"
    valor="${line##*=}"

    case $variable in
        ...
    esac

done
```

- El fichero contiene variables de la forma:
  - variable=valor
- El case analizará las palabras clave

Trucos ;-) (y II)

Analisis de campos con $1, $2...:

```
IFS=',' ; set $campos; IFS=' '
for ((i=1;$#!=0;i++)); do
    echo "Campo $i: $1"
    shift
done
```

- $campos contiene:
  - argumento1, argumento2 ...

Bibliografía y «webliografía»

Ínternet:

- «Csh Programming Considered Harmful» (comp.unix.shell)
- «El Cómo de la Programación en BASH» (www.insflug.org)
- Historia del shell Korn (http://www.kornshell.com/info)

Manual:

- bash(1), ksh(1), sh(1), csh(1), zsh(1)
- test(1), [(1)

Libros:

- Rosenblatt Bill: «Learning the Korn Shell». Ed. O'Reilly
- Cameron & Rosenblatt: «Learning the Bash Shell». Ed. O'Reilly

¡GRACIAS!

(aplausos y preguntas)