Recopilatorio de comandos para scripting en bash LINUX/UNIX

Martín Gil Blanco

Índice

declare, typeset y readonly

`declare e typeset` son sinónimos. Permiten declarar variables.

```
si@si-VirtualBox:~$ declare var1='AAA'
si@si-VirtualBox:~$ var2='BBB'
si@si-VirtualBox:~$ declare | grep var[12]
var1=AAA
var2=BBB

si@si-VirtualBox:~$ var1='CCC'
si@si-VirtualBox:~$ declare -p | grep var[12]
declare -- var1="CCC"
declare -- var2="BBB"
```

- `declare -r` no permite volver a redeclarar una variable. En este caso tanto `declare -r` como `readonly` para declarar variables de solo lectura funcionan del mismo modo.

```
si@si-VirtualBox:~$ declare -r var3='CCC'
si@si-VirtualBox:~$ declare -p | grep var3
declare -- _="var3=CCC"
declare -r var3="CCC"
```

```
si@si-VirtualBox:~$ var3='NO PUEDO DECLARAR'
bash: var3: readonly variable
```

```
si@si-VirtualBox:~$ readonly var3='No puedo declarar de nuevo'
bash: var3: readonly variable
si@si-VirtualBox:~$ readonly var4='No puedo declarar de nuevo'
si@si-VirtualBox:~$ declare -p | grep var4
declare -- _="var4=No puedo declarar de nuevo"
declare -r var4="No puedo declarar de nuevo"
```

- `declare -p` permite ver todas las variables del sistema junto con las nuevas vaiables creadas en la shell.

---

set ,unset y env

El comando `set` en Linux, sin opciones, muestra los nombres y valores de cada variable de la shell en un formato que puede ser reutilizado como entrada. `set` como comando permite ver las variables de entorno, variables locales y globales. Con la opción "-o" o "+o", permite establecer o quitar respectivamente atributos de la shell.

```
$ set -o # Muestra todas las opciones activadas (on) y desactivadas (off) en la
shell
$ set +o # Muestra todas las opciones activadas (-o) y desactivadas (+o) en la
shell, con el formato de las órdenes utilizadas para lograr ese resultado
$ set -o noclobber # Con esta opción, la shell bash no puede sobrescribir un
archivo existente con los operadores de redirección (>, >&, <>). Añade la opción
noclobber a la variable SHELLOPTS, siendo esta variable una lista de elementos
separados por dos puntos, de opciones activas de la shell.
$ set +o noclobber # Desactiva el comando anterior, es decir, se activa la
posibilidad de sobrescribir archivos con los operadores de redirección
$ set -C # Equivale a set -o noclobber
$ set +C # Equivale a set +o noclobber
```

`unset` elimina variables de la shell. No puede eliminar variables de solo lectura (definidas con "readonly" o "declare -r").

```
si@si-VirtualBox:~$ declare papelera='AAA'
si@si-VirtualBox:~$ echo $papelera
AAA
si@si-VirtualBox:~$ unset papelera
si@si-VirtualBox:~$ echo $papelera
```

El comando `env` ejecuta un programa con un entorno modificado según los parámetros con los que se ejecute. Esto significa que ejecuta un programa definiendo qué variables de entorno reconoce. Sin opciones o

nombre de programa, el comando muestra el entorno resultante (las variables globales del entorno), similar al comando `printenv`.

- `env bash`: Este comando ejecuta el shell de bash con el entorno actual, es decir, utiliza las variables de entorno existentes tal como están en ese momento.
- `env variable=JEJEJE bash`: Aquí se ejecuta el shell de bash con una variable de entorno llamada `variable` establecida en el valor "JEJEJE". Esto significa que al abrir el shell, la variable `variable` tendrá el valor "JEJEJE".
- `env -i bash`: Este comando ejecuta el shell de bash con un entorno vacío. La opción `-i` indica "ignorar el entorno existente", por lo que no se pasan variables de entorno al nuevo shell bash, dejándolo con un entorno limpio.

A modo de resumen podemos decir:

- set -> todas las variables
- env -> variables de un entorno

```
nuevo@si-VirtualBox:~$ env | wc -l
32

nuevo@si-VirtualBox:~$ set | wc -l
2136
```

```
si@si-VirtualBox:~$ env | wc -l
54

si@si-VirtualBox:~$ set | wc -l
2161
```

---

source, export y env

`source` y el carácter punto `.` son sinónimos. Leen y ejecutan los comandos existentes en un archivo dado en el entorno actual de la shell. El archivo no necesita ser ejecutable y se busca primero en las rutas del PATH y luego en la ruta actual (pwd) de ejecución del comando.

```
source $HOME/.bashrc # Recarga el archivo $HOME/.bashrc en la shell actual
```

El comando `export` permite exportar variables al entorno actual del shell, de manera que una vez exportadas, son válidas tanto en el entorno actual del shell como en cualquier subshell.

```
si@si-VirtualBox:~$ export HHH='hes' # Declara una variable llamada HHH con el
valor 'hes' y además exporta la variable para que pueda ser reconocida en el
entorno actual de la shell.
si@si-VirtualBox:~$ env | grep HHH
HHH=hes
si@si-VirtualBox:~$ echo $SHLVL
1

si@si-VirtualBox:~$ bash
si@si-VirtualBox:~$ echo $SHLVL
2
si@si-VirtualBox:~$ env | grep HHH
HHH=hes
si@si-VirtualBox:~$
```

Es importante destacar que esto no se puede realizar a la inversa, es decir, las variables exportadas en una subshell no van a afectar a la shell padre.

```
si@si-VirtualBox:~$ echo $SHLVL
2
si@si-VirtualBox:~$ export noAfecta="Variable de subshell"
si@si-VirtualBox:~$ env | grep noAfecta
noAfecta=Variable de subshell
si@si-VirtualBox:~$ exit
exit

si@si-VirtualBox:~$ echo $SHLVL
1
si@si-VirtualBox:~$ env | grep noAfecta
si@si-VirtualBox:~$
```

Como se puede ver podemos crear tambien variables que no se puedan borrar ni editar y luego exportarlas.

```
si@si-VirtualBox:~$ declare -r noBorrar="No me puedes borrar JAJA"
si@si-VirtualBox:~$ echo $noBorrar
No me puedes borrar JAJA

si@si-VirtualBox:~$ declare -p | grep noBorrar
declare -r noBorrar="No me puedes borrar JAJA"

si@si-VirtualBox:~$ noBorrar2="Vamos a editar la variable"
-bash: noBorrar2: readonly variable

si@si-VirtualBox:~$ export noBorrar
si@si-VirtualBox:~$ declare -p | grep noBorrar
declare -rx noBorrar="No me puedes borrar JAJA"
```

Comillas en variables de bash

Lo que estás viendo es un ejemplo de cómo se comportan las variables en bash (o cualquier shell de Unix) con diferentes tipos de comillas y la expansión de comandos.

1. Aquí estás asignando el valor "ls" a la variable "a".

```
si@si-VirtualBox:~$ a=ls
```

2. Al imprimir el valor de la variable "a" con `$a`, bash expande la variable y ejecuta el comando "ls", mostrando el contenido del directorio actual.

```
si@si-VirtualBox:~$ echo $a
ls
```

3. Al imprimir '$a' entre comillas simples, bash toma el texto literalmente y no expande la variable "a", por lo que muestra "$a".

```
si@si-VirtualBox:~$ echo '$a'
$a
```

4. Al imprimir "$a" entre comillas dobles, bash expande la variable "a" y ejecuta el comando "ls", mostrando el contenido del directorio actual.

```
si@si-VirtualBox:~$ echo "$a"
ls
```

5. Aquí estás intentando ejecutar el comando "ls" usando la expansión de comandos, pero la forma en que está escrito no es correcta. Esto ejecuta el resultado de "ls" como un comando independiente, lo que significa que si hay archivos o directorios en el directorio actual, intentará ejecutarlos como comandos y puede generar errores si no son ejecutables.

```
si@si-VirtualBox:~$ echo `$a`
Desktop Documents Downloads env-exemplo1.sh Music Pictures Public script.sh snap
Templates Videos
```

Uso de {variable}

Definición de variables.

```
si@si-VirtualBox:~$ a=1
si@si-VirtualBox:~$ b=2
si@si-VirtualBox:~$ c=3
```

1. `echo $a-$b-$c`: Aquí, bash expande las variables y las une con el guión (-) como se indica en la cadena de formato. El resultado es "1-2-3".

```
si@si-VirtualBox:~$ echo $a-$b-$c
1-2-3
```

2. `echo ${a}_${b}_${c}`: Al utilizar las llaves `{}` para delimitar las variables, bash entiende claramente que la variable es "a", "b" y "c", mientras que el guión bajo (_) se utiliza como un literal y se concatena entre ellas. El resultado es "1_2_3".

```
si@si-VirtualBox:~$ echo ${a}_${b}_${c}
1_2_3
```

3. `echo $a_$b_$c`: En este caso, bash interpreta la expresión `$a_` como una variable llamada "a*" y expande su valor, que es "3" según lo definido anteriormente, y las demás variables y guiones bajos son ignorados. Entonces, solo imprime el valor de "a*" que es "3". Es importante destacar que el guión bajo no se considera un separador de nombres de variables en bash, por lo que "a_" se interpreta como una variable diferente a "a".

```
si@si-VirtualBox:~$ echo $a_$b_$c
3
```

Nota: Siempre es preferible hacer uso de `{}` para invocar a variables.

Ejecución de scripts

Shebang: También conocido como hashbang o sha-bang, es una convención en sistemas operativos tipo Unix que se utiliza en scripts para indicar qué intérprete de comandos debe ser utilizado para ejecutar el script. El shebang consiste en los caracteres "#!" seguidos de la ruta al intérprete. Por ejemplo:

1. En scripts bash:

```
#!/bin/bash
```

2. En scripts python:

```
#!/usr/bin/env python3
#!/usr/bin/env python
```

1. `chmod -x env.sh && bash env.sh`

   Si ejecutamos bash env.sh, no es necesario tener permisos de ejecución en el script y estamos `ejecutando el script en una subshell`, por lo que al finalizar el script se elimina la subshell.

2. `chmod +x env.sh && ./env.sh`

   Si el shebang es #!/bin/bash y lo ejecutamos mediante ./env.sh, siempre y cuando el script tenga permisos de ejecución, estamos `ejecutando el script en una subshell`, por lo que al finalizar el script se elimina la subshell. Es análogo a la ejecución mediante el comando bash.

3. `chmod -x env.sh && . ./env.sh`

   Si ejecutamos mediante . ./env.sh o source ./env.sh, no es necesario tener permisos de ejecución y estamos `ejecutando el script en la shell actual`.

Es fundamental comprender de que forma se ejecutan nuestros scripts para poder comprender si van a modificar aspectos de nuestro entorno o no. En el siguiente ejemplo podemos apreciar como el nivel de shell que es diferente en función de como lanzamos nuestro script.

Script `env-ejemplo1.sh`

```
#!/bin/bash
env | sort | grep -v '^_' | tee env1.txt
```

Si ejecutamos con las opciones 1 o 2, es decir, con `bash` o con `./` se ejecutará el script en una subshell por lo que no afectará a mi entorno.

Podemos ver que al lanzarlo con `bash` (o con `./`) estamos en el nivel de shell 1.

```
si@si-VirtualBox:~$ bash env-ejemplo1.sh
COLORTERM=truecolor
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
DESKTOP_SESSION=ubuntu
DISPLAY=:0
GDMSESSION=ubuntu
...

si@si-VirtualBox:~$ cat env1.txt | grep SHLVL
SHLVL=1
```

En este caso, ejecutamos directamente el comando y podemos ver que estamos en el nivel de shell 0, es decir, el mismo nivel de shell donde lanzamos comandos. Esto quiere decir que las modificaciones de este comando si podrían afectar a mi entorno a diferencia del caso anterior.

```
si@si-VirtualBox:~$ env | sort | grep -v '^_' | tee env2.txt
COLORTERM=truecolor
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
DESKTOP_SESSION=ubuntu
DISPLAY=:0
GDMSESSION=ubuntu
...

si@si-VirtualBox:~$ cat env2.txt | grep SHLVL
SHLVL=0
```

```
si@si-VirtualBox:~$ source env-ejemplo1.sh
COLORTERM=truecolor
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
DESKTOP_SESSION=ubuntu
DISPLAY=:0
GDMSESSION=ubuntu
...

si@si-VirtualBox:~$ cat env1.txt | grep SHLVL
SHLVL=0
```

Este es el motivo por el cual cuando queremos modificar nuestro entorno se hace uso de ficheros como `.bashrc` y este se lanza con `source`. El objetivo es hacer una modificación de nuestro entorno.

Cargamos .bashrc con `.` por lo tanto al ser lo mismo que `source` se convierten en variables de entorno las variables definidas dentro a las que se le aplica un `export`.

```
si@si-VirtualBox:~$ cat .profile | grep ".bashrc"
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
    . "$HOME/.bashrc"
```

---

subshells

Si colocamos una secuencia de órdenes entre paréntesis, forzamos a que estos comandos se ejecuten en una subshell.

```
$ (readonly AAA='aes'; echo $AAA; unset AAA; echo $AAA); AAA='bes'; echo $AAA;
unset AAA;
echo $AAA # En la subshell definida con paréntesis y en la shell, las variables
AAA son variables distintas.
```

Variables locales y globales en un script

`global, local`

Todas las variables en los scripts bash, a menos que se definan de otra manera, son globales, es decir, una vez definidas pueden ser utilizadas en cualquier parte del script. Para que una variable sea local, es decir, tenga sentido solamente dentro de una sección del script, como en una función, y no en todo el script, debe ser precedida por la sentencia: local.

`caso A`: Sin sentencia local para la variable `NOMBRE`.

```
si@si-VirtualBox:~$ cat script.sh
    #!/bin/bash

    function dentro_variable_local() {
            NOMBRE="DENTRO"
            echo ${NOMBRE}
    }

    NOMBRE="FUERA"
    echo ${NOMBRE}

    dentro_variable_local
    echo ${NOMBRE}

si@si-VirtualBox:~$ ./script.sh
FUERA
DENTRO
DENTRO
```

`caso B`: Con sentencia local para la variable `NOMBRE`.

```
si@si-VirtualBox:~$ cat script.sh
    #!/bin/bash

    function dentro_variable_local() {
            local NOMBRE="DENTRO"
            echo ${NOMBRE}
    }

    NOMBRE="FUERA"
    echo ${NOMBRE}

    dentro_variable_local
    echo ${NOMBRE}

si@si-VirtualBox:~$ ./script.sh
FUERA
DENTRO
FUERA
```

---

arrays en bash

declarar arrays

Para declarar un array podemos hacerlo de varias maneras.

- `declare -a`: Utilizado para declarar una variable como un array en Bash. Esto significa que la variable podrá contener múltiples valores, accesibles a través de índices numéricos. Esta opción es útil para asegurar que una variable se comporte como un array, incluso si no contiene valores al principio.

```
si@si-VirtualBox:~$ vectorUno[0]=nuevo

si@si-VirtualBox:~$ declare -p | tail -3
declare -- snap_bin_path="/snap/bin"
declare -- snap_xdg_path="/var/lib/snapd/desktop"
declare -a vectorUno=([0]="nuevo")

si@si-VirtualBox:~$ vectorUno[1]=nuevo2
si@si-VirtualBox:~$ vectorUno[3]=3

si@si-VirtualBox:~$ declare -p | tail -3
declare -- snap_bin_path="/snap/bin"
declare -- snap_xdg_path="/var/lib/snapd/desktop"
declare -a vectorUno=([0]="nuevo" [1]="nuevo2" [3]="3")
```

```
si@si-VirtualBox:~$ vectorDos=(uno dos tres)
si@si-VirtualBox:~$ declare -p vectorDos
declare -a vectorDos=([0]="uno" [1]="dos" [2]="tres")
```

```
si@si-VirtualBox:~$ vectorTres=([0]=Uno [1]=Dos [4]=Cuatro)
si@si-VirtualBox:~$ declare -p vectorTres
declare -a vectorTres=([0]="Uno" [1]="Dos" [4]="Cuatro")
```

```
si@si-VirtualBox:~$ declare -a vectorCuatro[0]=Hola
si@si-VirtualBox:~$ echo ${vectorCuatro[0]}
Hola
```

- `declare -p`: Esta opción muestra información sobre una o más variables, incluyendo su tipo y valor. Es útil para depurar y entender el estado de las variables en un script de Bash.

```
si@si-VirtualBox:~$ declare -p vectorUno
declare -a vectorUno=([0]="nuevo" [1]="nuevo2" [3]="3")
```

- `local` es una palabra clave que se utiliza dentro de funciones para declarar variables locales. Esto significa que las variables definidas como locales solo son visibles y accesibles dentro de la función donde se declaran, y no afectan a variables con el mismo nombre fuera de la función. Por ejemplo:

```
mi_funcion() {
    local variable_local="Este es un valor local"
    echo "Dentro de la función: $variable_local"
}

mi_funcion
echo "Fuera de la función: $variable_local"  # Esto mostrará un valor vacío o
error, ya que $variable_local no está definido fuera de la función
```

- `local -a` es una variante de `local` que se utiliza para declarar variables locales como arrays. Esto asegura que la variable declarada sea tratada como un array dentro de la función. Por ejemplo:

```
mi_funcion() {
    local -a array_local
    array_local=("elemento1" "elemento2" "elemento3")
    echo "Dentro de la función: ${array_local[@]}"
}

mi_funcion
echo "Fuera de la función: ${array_local[@]}"  # Esto mostrará un valor vacío o
error, ya que $array_local no está definido fuera de la función
```

Referenciar Arrays

- `${nombre[índice]}` donde nombre es el nombre del array e índice es la posición entera que contiene su valor correspondiente.
- `${nombre[@]}` donde nombre es el nombre del array y el índice @ representa todos los elementos del vector. Devuelve una cadena con los elementos separados por espacio.
- `${nombre[*]}` donde nombre es el nombre del array y el índice * representa todos los elementos del vector. Devuelve una cadena con los elementos separados por espacio.
- `"${nombre[@]}"` donde nombre es el nombre del array y el índice @ representa todos los elementos del vector. Devuelve una cadena con los elementos separados por espacio.
- `"${nombre[*]}"` donde nombre es el nombre del array y el índice * representa todos los elementos del vector. Devuelve una cadena con los elementos separados por el primer caracter de la variable separador de campos IFS.
- `${nombre}` equivale a `${nombre[0]}`

```
$ nombre=(primero segundo tercero)
$ declare -p nombre
declare -a nombre=([0]="primero" [1]="segundo" [2]="tercero")

$ echo ${nombre[*]}
primero segundo tercero
$ echo "${nombre[*]}"
primero segundo tercero

$ set | grep ^IFS
IFS=$' \t\n'
$ IFS=$'Z\t\n'
$ set | grep IFS
IFS=$'Z\t\n'

$ echo ${nombre[*]}
primero segundo tercero
$ echo "${nombre[*]}"
primeroZsegundoZtercero

Es decir, a "${nombre[*]}" Le afecta el primer caracter de la variable separador
de campos IFS.
```

Eliminar arrays

- `unset nombre`: Elimina el array declarado como `nombre` siempre que no esté declarado en modo lectura.
- `unset nombre[índice]`: Elimina el índice del array `nombre` en la posición [índice] siempre que no esté declarado en modo lectura.

Recorrer arrays

- Recorre los índices del array declarado como `nombre` y muestra cada valor en pantalla. Siempre utiliza comillas dobles para evitar errores de separadores como espacios que puedan existir en los valores de los índices.

```
for i in "${nombre[*]}"
do
echo $i
done
```

- Recorre los índices del array declarado como `nombre` y muestra cada valor en pantalla. Siempre utiliza comillas dobles para evitar errores de separadores como espacios que puedan existir en los valores de los índices.

```
for i in "${nombre[@]}"
do
```

```
echo $i
done
```

Longitud de los arrays e índices

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

- Muestra el número de elementos existentes en el array declarado como `nombre`, es decir, proporciona el número de posiciones ocupadas en el array.

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

- Muestra los índices de los elementos que no son NULOS en el array declarado como `nombre`.

```
echo ${#nombre[índice]}
```

- Muestra el número de caracteres de longitud del índice `índice` para el array declarado como `nombre`.

```
si@si-VirtualBox:~$ vector3=(uno dos tres)
si@si-VirtualBox:~$ echo ${vector3[*]}
uno dos tres
si@si-VirtualBox:~$ echo ${vector3[@]}
uno dos tres
si@si-VirtualBox:~$ echo ${vector3[0]}
uno
si@si-VirtualBox:~$ echo ${vector3}
uno
si@si-VirtualBox:~$ echo ${vector3[1]}
dos
si@si-VirtualBox:~$ echo ${vector3[2]}
tres
si@si-VirtualBox:~$ echo ${vector3[3]}

si@si-VirtualBox:~$ echo ${#vector3[@]}
3
si@si-VirtualBox:~$ echo ${!vector3[@]}
0 1 2
si@si-VirtualBox:~$ echo ${#vector3[0]}
3
si@si-VirtualBox:~$ echo ${#vector3[1]}
3
si@si-VirtualBox:~$ echo ${#vector3[2]}
4
```

Arrays como parámetros de funciones

En la llamada a la función utilizamos como parámetro la notación "nombre[@]", y dentro del cuerpo de la función utilizamos la referencia indirecta ${!número}, donde número es el número del parámetro en la llamada a la función: 1, 2, 3...

```
f_nombre() { # Definición de la función llamada f_nombre
    parametros=("${!1}") # Referencia indirecta del parámetro $1 dentro de la
definición del array parametros
    echo "${parametros[@]}" # Muestra todos los valores contenidos en los índices
del array llamado parametros
} # Fin del cuerpo de la función f_nombre

nombres=(Anxo Brais) # Se declara la variable array nombres con los valores Anxo y
Brais en los índices 0 y 1 respectivamente.
f_nombre "nombres[@]" # Llamada a la ejecución de la función llamada f_nombre
donde el primer parámetro $1 toma el valor "nombres[@]", lo cual es equivalente a
los valores de todos los índices del array llamado nombres
```

Prácticas sobre arrays

Declarar y eliminar arrays

Muestra el valor de la variable llamada "curso" y cómo está declarada en caso de existir.

```
declare -p curso
```

Crea automáticamente el array llamado "curso" donde en el índice 0 toma el valor '2015-2016'.

```
curso[0]='2015-2016'
```

Genera un nuevo array llamado "curso" donde los índices 0 y 1 toman los valores '2017-2018' y '2018-2019' respectivamente.

```
curso=('2017-2018' '2018-2019')
```

Aunque no existe el índice 2 en el array llamado "curso", la sintaxis del comando genera un nuevo array con solo un índice: el índice 2 que toma el valor '2019-2020'.

```
curso=([2]='2019-2020')
```

Genera un nuevo array llamado "curso" donde solo se hacen referencia en la declaración a los índices 0 y 5, por lo que el resto de los elementos del array irán guardando sus valores a partir del índice 5. Así, los índices 0, 5, 6 y 7 toman los valores '1111-1112', '1112-1113', '1113-1114' y '1114-1115' respectivamente.

```
curso=([0]='1111-1112' [5]='1112-1113' '1113-1114' '1114-1115')
```

Muestra el valor de la variable array llamada "curso" y cómo está declarada.

```
declare -p curso
```

Declara la variable array llamada "curso" en modo solo lectura.

```
declare -r curso
```

No puede eliminar la variable "curso" ya que está declarada en modo solo lectura.

```
unset curso
unset curso[@]
unset curso[*]
```

No puede eliminar el elemento de la posición 1 de la variable "curso" ya que está declarada en modo solo lectura.

```
unset curso[1]
```

Nota: No es lo mismo pedir por teclado unos datos para el array que pedir los datos para un índice concreto del array.

1. **Pedir datos para un array**

Lee desde la entrada estándar (teclado) una lista de palabras como índices en un array llamado "curso5", comenzando desde 0.

Los campos separadores de palabras, que serán recogidos en los índices del array, están determinados por el valor de la variable separador de campos $IFS.

```
read -p 'Escriba valores del array en una lista: ' -a curso5
```

Muestra el mensaje 'Escriba valores del array en una lista: ' y recoge desde la entrada estándar (teclado) la lista de palabras como índices en un array llamado curso5, empezando por 0. Los campos separadores de palabras, que serán recogidos en los índices del array, están determinados por el valor de la variable separador de campos $IFS.

```
1 2 3
```

Como el valor de IFS=$' \r\n' y el número de palabras de la lista insertada es 3, se sustituyen los valores de los índices 0, 1 y 2 por estos nuevos: 1, 2 y 3 respectivamente.

2. **Pedir datos para una posición del array**

Recoge por teclado el valor del índice 1 del array llamado "curso5".

```
read -p 'Escriba valor para el índice 1: ' curso5[1]
```

Asigna el valor '4 5 6' al índice 1, ya que en la asignación al índice no se tiene en cuenta la variable IFS.

```
4 5 6
```