Arrays en BASH

En Bash tenemos la posibilidad de usar arrays. Un array es una variable con varios elementos y tienen muchísima utilidad. Por ejemplo, para recoger el listado de archivos que hay en una carpeta.

Para crear un array hay varias formas, una de ellas es definir la variable del array y añadirle todos los elementos de golpe:

```
1  #!/bin/bash
2  numeros=(1 2 3 4 5)
```

Este script creará un array con cinco elementos (del 1 al 5).

También se puede usar esta otra forma:

```
1  #!/bin/bash
2  declare -a numeros=(1 2 3 4 5)
```

Mostrar todos los elementos del array

Si queremos mostrar el array completo podemos usar este sencillo comando:

```
1  echo ${numeros[@]}
```

Como se puede ver tiene una sintaxis muy fácil de recordar. Tan fácil que solo tendrás que consultarla unas diez veces antes de poder empezar a recordarla. No se puede decir que Bash tenga una sintaxis bonita.

También se puede usar con un asterisco en lugar de la arroba:

```
1  echo ${numeros[*]}
```

En realidad no es tan difícil de recordar. El $ es el símbolo que se usa para referirnos a las variables, las llaves son necesarias siempre que queremos acceder a los valores de un array y, por último, entre corchetes va el índice del elemento al que queremos acceder (en este caso el '*' se refiere a todos los elementos).

Hay una diferencia entre usar * y @ que explicaré en otra entrega.

Acceder a un elemento del array

Si queremos acceder a un elemento del array podemos usar la forma:

```
1  echo ${numeros[0]}
```

Al igual que sucede en la mayoría de los lenguajes de programación el primer elemento del array tiene el índice 0. Este ejemplo nos mostraría, por tanto, un 1.

Ojo, Bash no comprueba los límites de un array. Es posible hacer algo así sin tener ningún error:

```
1    echo ${numeros[10]}
```

De forma similar podemos cambiar un elemento del array:

```
1    #!/bin/bash
2
3    numeros=(1 2 3 4 5)
4    numeros[1]="Cambiado"
5    echo ${numeros[*]}
```

Este script mostrará el siguiente resultado:

1 Cambiado 3 4 5

Porque hemos modificado el segundo elemento (que tiene el índice 1).

También podemos añadir nuevos elementos:

```
1    #!/bin/bash
2
3    numeros=(1 2 3 4 5)
4    numeros[5]=6
5    echo ${numeros[*]}
```

Esto mostrará:

```
1    1 2 3 4 5 6
```

Recuerda, para modificar o añadir un elemento a un array se hace como con una variable normal, no se pone el $ al principio.

Si usamos un índice con un número mayor el resultado será, aparentemente, el mismo:

```
1    #!/bin/bash
2
3    numeros=(1 2 3 4 5)
4    numeros[10]=6
5    echo ${numeros[*]}
```

Esto mostrará:

```
1    1 2 3 4 5 6
```

La diferencia está en que, ahora, el 6 es el elemento con índice 10 y no índice 5 como tenía en el ejemplo anterior.

También podemos usar este método para crear un nuevo array:

```
1    #!/bin/bash
2
3    numeros[0] = 1
4    numeros[1] = 2
```

Es decir, cuando asignamos un elemento a una variable automáticamente se crea el array.

Índices de un array en BASH

Podemos ver qué índice tiene cada elemento del array usando:

```
1  echo ${!numeros[*]}
```

Usado en el ejemplo anterior quedaría:

```
1
2  #!/bin/bash
3
4  numeros=(1 2 3 4 5)
5  numeros[10]=6
6  echo "Contenido del array:"
7  echo ${numeros[*]}
8  echo "Índices del array:"
9  echo ${!numeros[*]}
10
```

Y mostraría:

```
1  $ ./arrays
2  Contenido del array:
3  1 2 3 4 5 6
4  Índices del array:
5  0 1 2 3 4 10
```

El primer elemento tiene el índice 0, el segundo el índice 2,... y el último el índice 10 (porque así lo hemos querido).

Número de elementos en un array

Añadiendo el símbolo '#' tenemos el tamaño del array:

```
1  #!/bin/bash
2
3  numeros=(1 2 3 4 5)
4  echo ${#numeros[*]}
```

El tamaño de un elemento

Podemos conocer directamente el tamaño de un elemento del array usando la # al acceder a dicho elemento:

```
1  #!/bin/bash
2
3  numeros=(Primero Segundo Tercero)
4  echo ${#numeros[1]}
```

El resultado será un 7, que es el número de letras de la palabra “Segundo”.

Ahorrando teclas al crear un array

Cuando crearmos un array podemos ahorrarnos un montón de trabajo usando rangos:

```
#!/bin/bash

cifrasLetras=( {A..Z} {a..z} {0..9} )
echo ${cifrasLetras[*]}
```

Esto va a crear un array con el siguiente contenido:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9
```

Por supuesto podemos usar los rangos empenzando en la letra o el número que queramos:

```
#!/bin/bash

cifrasLetras=( {N..X} {b..m} {6..21} )
echo ${cifrasLetras[*]}
```

Que resultará en:

```
N O P Q R S T U V W X b c d e f g h i j k l m 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

Un ejemplo real: meter el listado de archivos en un array

Algo bastante habitual suele ser meter un listado de archivos en un array y luego poder trabajar con ese array. Se puede hacer usando el comando ls dentro de unas comillas invertidas:

```
#!/bin/bash
ficheros=(`ls`)
echo ${ficheros[*]}
echo ${#ficheros[*]}
```

En la variable ficheros tendríamos el listado de los archivos y veríamos que el tamaño del array coincide con el número de archivos.

Ojo, si olvidásemos el paréntesis (que es lo que indica que estamos definiendo un array) se guardaría como una cadena:

```
#!/bin/bash
ficheros=`ls`
echo ${ficheros[*]}
echo ${#ficheros[*]}
```

El tamaño de este array sería siempre 1.

Incluir otros ficheros en BASH

En ocasiones nuestros scripts en bash se hacen demasiado largos y complejos. También es posible que haya partes comunes a varios scripts. En estas ocasiones puede ser conveniente separar parte del código en varios ficheros. O incluso podríamos querer cargar los datos de configuración qu estén en otro fichero.

Pero ¿cómo hacemos para usar código que hay en otro script en el nuestro?

Incluir un fichero en un script

Incluir código en un *script* es muy sencillo, basta con usar una de estas opciones:

```
1  . fichero
2  source fichero
```

Veamos un ejemplo. Crea un fichero llamado *prueba* y copia el siguiente código:

```
1  #!/bin/bash
2  echo "Voy a incluir el otro fichero"
3  . otro_fichero
4  echo "El otro fichero ha sido incluido"
```

y necesitamos otro fichero al que, de manera muy imaginativa, vamos a llamar *otro_fichero*. Copia en él el siguiente código:

```
1  echo "Este es el otro fichero"
```

No olvides hacer que *prueba* sea ejecutable:

```
1  $ chmod +x prueba
```

El resultado al ejecutarlo será:

```
1  $ ./prueba
2  Voy a incluir el otro fichero
3  Este es el otro fichero
4  El otro fichero ha sido incluido
```

¿Debo usar *source* o '.'?

Source no es POSIX (por resumir muuucho digamos que POSIX es un estándar para, entre otras cosas los intérpretes de comandos o *shell*) así que puede no estar disponible en tu consola.

En *Bash* sí está disponible y es exactamente lo mismo *source* que '.' así que puedes usarlos indistintamente.

Diferencias entre ejecutar un script e incluirlo

Posiblemente te habrás fijado en que *otro_fichero* no es ejecutable pero se ha ejecutado el código que había en su interior.

Si hubiésemos hecho dentro de *script*:

```
#!/bin/bash

./otro_fichero
```

El resultado hubiese sido un error porque *otro_fichero* no es ejecutable:

```
$ ./prueba
./script: línea 3: ./otro: Permiso denegado
```

Es decir, al incluir un fichero lo que hacemos es "copiar" el contenido de ese fichero en nuestro *script*. Es como si ese código estuviese dentro del fichero que estamos ejecutando.

Si usamos la opción de ejecutar *otro_fichero* estamos llamando a un *script* externo y ése deberá tener sus permisos de ejecución.

Problemas de seguridad

Incluir un fichero puede ser una fuente de problemas de seguridad. Imagina un script que se ejecuta como *root*. Este script incluye un fichero que está en la carpeta a la que tienen acceso otros usuarios. Un usuario malvado y malandrín podría hacer cambios en ese fichero incluido y ejecutar los comandos que quiera. ¡Y esos comandos se ejecutarán como administrador!

Veamos un ejemplo. Por un lado tenemos el fichero *script*:

```
#!/bin/bash
. otro_fichero
y por otro lado tenemos otro_fichero en el que teclearemos:
whoami
```

NOTA: *whoami* es un comando que nos muestra el usuario con el que estamos ejecutando comandos.

El resultado será:

```
$ ./prueba
gorka
```

En mi caso se muestra mi nombre de usuario.

Ahora cambia de cuenta y accede como root (puedes hacerlo directamente tecleando):

```
$ sudo -i
```

y en una carpeta a la que solo tenga acceso este usuario root crea el fichero prueba con el siguiente contenido:

```
#!/bin/bash
. /home/gorka/otro_fichero
```

Creo que sobra decir que en lugar de */home/gorka* tendrás que poner la ruta donde tengas *otro_fichero*.

Hazlo ejecutable:

```
$ chmod +x prueba
```

Ahora ejecuta el fichero y verás el escalofriante resultado:

```
$ ./prueba
root
```

¡El comando whois se ha ejecutado usando el usuario *root*! Imagina si hubiesen metido cualquier cosa desagradable ahí.

En una próxima entrega veremos cómo filtrar un fichero de configuración para poder incluirlo sin tanto peligro.

Mi script no encuentra el fichero que quiero incluir

En los primeros ejemplos hemos visto que para incluir un fichero simplemente escribíamos su nombre. Pero eso puede ser una fuente de problemas. Por ejemplo, si llamas desde un *cron* o desde otra carpeta puede que tu script no encuentre el fichero que quieres incluir. Vamos a verlo.

Tenemos el fichero *prueba*:

```
#!/bin/bash

./otro_fichero
```

y el fichero *otro_fichero*:

```
echo "Soy el otro fichero"
```

Al ejecutar *prueba*:

```
<a name="__DdeLink__397_1143315117"></a>$ ./prueba
Soy el otro fichero
```

Ahora prueba a cambiar de carpeta. En mi caso yo he guardado el fichero en una carpeta llamada */home/gorka/pruebas*. Voy a ejecutar el *script* desde la carpeta */home/gorka*:

```
$ pruebas/prueba
pruebas/script: línea 3: otro_fichero: No existe el archivo o el directorio
```

¿Por qué sucede ésto? Porque prueba está buscando el fichero otro_fichero en la carpeta donde se está ejecutando (en este caso en /home/gorka).

Esto mismo te va a suceder cuando ejecutes el script desde un cron o desde otro usuario. Hay que tener cuidado.

¿Cómo podemos evitarlo? Hay varias posibles soluciones, pero una de las más seguras es usar la variable de entorno $BASH_SOURCE:

```
1  #!/bin/bash
2  ruta_completa=${BASH_SOURCE[0]}
3  carpeta_script=${ruta_completa%/*}
4  . $carpeta_script/otro_fichero
```

Ahora sí funcionará:

```
1  $ pruebas/prueba
2  Soy el otro fichero
```

NOTA: Este “truco” no funciona con enlaces simbólicos. En otra entrega hablaré sobre para qué sirve $BASH_SOURCE y cómo hacer funcionar ésto con enlaces simbólicos.