
BASH Unidad 6

Control de flujo, bucles, argumentos y entrada de usuario

Tabla de contenidos

Control de flujo (II)

Además del comando **if**, existen otros que también sirven para controlar el flujo del programa

Comando case

El ejemplo del comando **if** que vimos la semana pasada, era el de una bifurcación simple, ya que para la ejecución del programa solo había dos caminos posibles. Supongamos que ahora tenemos una bifurcación multi-elección como la siguiente:

```
#!/usr/bin/env bash

echo -n "Introduce un número entre 1 y 3, ambos incluidos > "
read character
if [ "$character" = "1" ]; then
    echo "Has introducido un uno."
elif [ "$character" = "2" ]; then
    echo "Has introducido un dos."
elif [ "$character" = "3" ]; then
    echo "Has introducido un tres."
else
    echo "No has introducido un número dentro del rango."
fi
```

Esto no es muy bonito, por lo que la shell proporciona una solución elegante a este problema a través del comando **case**.

```
#!/usr/bin/env bash

echo -n "Introduce un número entre 1 y 3, ambos incluidos > "
read character
case "$character" in
    1 ) echo "Has introducido un uno."
        ;;
    2 ) echo "Has introducido un dos."
        ;;
    3 ) echo "Has introducido un tres."
        ;;
    * ) echo "No has introducido un número dentro del rango."
esac
```

En general, el comando **case** tiene la siguiente forma:

```
case palabra in
    patrón ) comandos ;;
esac
```

- **Este comando básicamente ejecuta los “comandos” del “patrón” que encaje con “palabra”**.
- Se puede tener cualquier número de patrones.
- Los patrones son básicamente texto y comodines, como los usados en la expansión de nombres de ruta: ‘*’, ‘?’, [caracteres], [!caracteres], etc.
- El patrón especial “*” encaja con cualquier palabra, por lo que debe usarse al final del comando **case** para ejecutar comandos cuando “palabra” no encajan con otros patrones —como *default* en los *switch* de C++—.

- Si una lista de comandos termina en ";;" entonces ningún otro patrón del comando **case** es comprobado —como la sentencia *break* de C++—.

```
#!/usr/bin/env bash

echo -n "Escribe un dígito o una letra > "
read character
case $character in

    [[:lower:]] | [[:upper:]] ) # Comprobar letras
        echo "Has escrito la letra $character"
        ;;

    [0-9] )                     # Comprobar dígitos
        echo "Has escrito el dígito $character"
        ;;


    * )                         # Comprobar cualquier otra cosa
        echo "No has escrito ni un dígito ni una letra"
esac
```

Bucles

En cualquier lenguaje de programación es común que haya alguna manera de hacer bucles. **Un bucle permite la ejecución repetitiva de una sección del script, basándose en el estado de salida de un comando**. La shell proporciona tres comandos para hacer bucles: **while**, **until** y **for**.

Comandos while y until

El comando **while hace que una sección de código se ejecute una y otra vez mientras el estado de salida del comando especificado en los argumentos es *verdadero***.

```
#!/usr/bin/env bash

number=0
while [ "$number" -lt 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

Como se puede observar, creamos la variable "number" inicializada a 0 y en cada iteración comprobamos si el valor de dicha variable es menor de 10. Si no es así, se ejecuta el código delimitando entre **do** y **done**; que simplemente muestra el valor actual de "number" y lo incrementa en 1.

El comando **until** funciona exactamente de la misma manera, **excepto que la sección de código se ejecuta una y otra vez mientras el estado de salida del comando especificado en los argumentos es *falso***.

```
#!/usr/bin/env bash

number=0
until [ "$number" -ge 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

Obsérvese que en las condiciones del comando **while** y el comando **until** hemos usado las expresiones “-lt” y “-ge”. **Estas expresiones de test se utilizan para hacer comparaciones aritméticas, a diferencia de "=", "!=", "<"," >", etc. que en BASH sirven para hacer comparaciones de cadenas**.

Expresión	Descripción
número1 -eq número2	Verdadero si *número1* es igual a *número2*.
número1 -ne número2	Verdadero si *número1* no es igual a *número2*.
número1 -gt número2	Verdadero si *número1* es mayor que *número2*.
número1 -ge número2	Verdadero si *número1* es mayor o igual que *número2*.
número1 -lt número2	Verdadero si *número1* es menor que *número2*.
número1 -le número2	Verdadero si *número1* es menor o igual que *número2*.

Comando for

La forma del comando **for** es la siguiente:

```
for variable in palabras; do
   comandos
done
```

Básicamente, **for asigna una palabra de la lista de “palabras” a “variable”, ejecuta la lista de “comandos” y repite esto una y otra vez hasta que todas las palabras han sido usadas**. Aquí tenemos un ejemplo:

```
#!/usr/bin/env bash

for i in word1 word2 word3; do
    echo $i
done
```

En este ejemplo:

1. A la variable “i” se le asigna la cadena “word1”.
2. La sentencia “echo $i” es ejecutada.
3. A la variable “i” se le asigna la cadena “word2”.
4. La sentencia “echo $i” es ejecutada.
5. Y repetimos…

Lo interesante del comando **for** es que hay muchas maneras de construir la lista de palabras porque para eso se pueden usar todo tipo de expansiones. Por ejemplo, a continuación construiremos la lista de palabras usando comodines para seleccionar los archivos `.txt` y copiarlos añadiendo la extensión `.bak`.

```
for filename in *.txt; do
    cp $filename $filename.bak
done
```

También podemos usar sustitución de comandos. A continuación construiremos la lista de palabras a partir de una sustitución de comandos, lo que nos permitirá tomar el fichero `.bash_profile` y contar el número de palabras en el archivo y el número de caracteres en cada palabra:

```
#!/usr/bin/env bash

count=0
for i in $(cat ~/.bash_profile); do
    count=$((count + 1))
    echo "La palabra $count ($i) contiene $(echo -n $i | wc -c)
caracteres"
done
```

Comandos break y continue

En las listas de comandos a ejecutar dentro de los comandos **while**, **until** y **for** es posible usar los siguientes comandos para controlar la ejecución:

- **break**, rompe la ejecución del bucle dentro del que es invocado.
- **continue**, reanuda la iteración del bucle dentro del que es invocado.

Funciones y sustitución de comandos

La última vez que vimos nuestro script, se parecía a algo como esto, con algunas funcionalidades adicionales:

```
#!/usr/bin/env bash
```

```
# sysinfo - Un script que informa del estado del sistema

##### Constantes

TITLE="Información del sistema para $HOSTNAME"
RIGHT_NOW=$(date +"%x %r%Z")
TIME_STAMP="Actualizada el $RIGHT_NOW por $USER"

##### Estilos

TEXT_BOLD=$'\x1b[1m'
TEXT_ULINE=$'\x1b[4m'
TEXT_GREEN=$'\x1b[32m'
TEXT_RESET=$'\x1b[0m'

##### Funciones

system_info
{
      ...
}

show_uptime
{
      ...
}

drive_space
{
      ...
}

home_space
{
      ...
}

##### Programa principal

cat << _EOF_
$TEXT_BOLD$TITLE$TEXT_RESET

$TEXT_GREEN$TIME_STAMP$TEXT_RESET
_EOF_
```

```
system_info
show_uptime
drive_space
home_space
```

Pero el resultado no es del todo correcto, porque la idea es que la información mostrada por las funciones del final se muestra entre el encabezado —con el título— y el pie del informe —con la marca de tiempo—.

Lo bueno es que las funciones también se pueden invocar dentro de una sustitución de comandos, así

```
##### Programa principal

cat << _EOF_
$TEXT_BOLD$TITLE$TEXT_RESET

$(system_info)
$(show_uptime)
$(drive_space)
$(home_space)

$TEXT_GREEN$TIME_STAMP$TEXT_RESET
_EOF_
```

El resultado es que la salida estándar de las funciones es capturada y sustituida donde están los $(...). Así se genera un texto completo del informe, que luego muestra el comando “cat”.

Parámetros posicionales

Ya tenemos la mayoría de las características deseadas funcionando, pero hay un par de cosas más que podemos añadir:

1. Guardar el informe automáticamente en un archivo especificado en la línea de comandos al ejecutar el script.
2. Ofrecer un modo interactivo que le pida al usuario un nombre de archivo, advirtiéndole si el archivo especificado existe y preguntándole si desea sobreescribirlo.
3. Ofrecer una opción de ayuda con la que se muestre información de cómo se usa el programa.

Todas estas características usan las opciones de la línea de comandos y los argumentos. **Para manejar opciones en la línea de comandos de un script, se usan los llamados *parámetros posicionales* de la shell. Estos son una serie de variables especiales**

$0...$9 que contienen los argumentos de la línea de comandos indicada para ejecutar el script.

Supongamos que ejecutamos un programa con la siguiente línea de comandos:

```
yo@mihost:~$  mi_programa palabra1 palabra2 palabra3
```

Si “mi_programa” fuera un script de BASH, los parámetros posicionales contendrían lo siguiente:

- $0 contendría “mi_programa”
- $1 contendría “palabra1”
- $2 contendría “palabra2”
- $3 contendría “palabra3”

Para comprobarlo podemos usar el siguiente script:

```
#!/usr/bin/env bash

echo "Parámetros posicionales"
echo '$0 = ' $0
echo '$1 = ' $1
echo '$2 = ' $2
echo '$3 = ' $3
```

Detectando argumentos de la línea de comandos

Con frecuencia se desea comprobar si hay argumentos con los cuales trabajar. Por suerte hay un par de formas de hacer esto.

La primera es comprobar si $1 contiene algo:

```
#!/usr/bin/env bash

if [ "$1" != "" ]; then
    echo "El parámetro posicional 1 contiene algo"
else
    echo "El parámetro posicional 1 está vacío"
fi
```

La segunda es usar la variable $#, que contiene el número de ítems en la línea de comandos, sin contar el nombre del propio comando que es almacenado en $0.

```
#!/usr/bin/env bash

if [ $# -gt 0 ]; then
```

```
    echo "Tu línea de comandos contiene $# argumentos"
else
    echo "Tu línea de comandos no contiene argumentos"
fi
```

Opciones de la línea de comandos

Muchos programas soportan tanto opciones de línea de comandos largas como cortas. Por ejemplo, para mostrar un mensaje de ayuda, muchos programas suelen admitir “-h” o la opción más larga “--help”. Los nombres largos de opciones suelen ir precedidos de “--” por lo que nosotros también adoptaremos esta convención para nuestros scripts.

Así es como debería cambiar la sección del programa principal de nuestro script para procesar la línea de comandos —los cambios respecto al ejemplo anterior se indican en negrita—:

```
# Opciones por defecto.
interactive=
filename=~/sysinfo.txt

...

##### Programa principal

usage()
{
    echo "usage: sysinfo [-f filename] [-i] [-h]"
}

write_page()
{
    # El heredoc se puede indentar dentro de la función si
    # se usan tabuladores y "<<-EOF" en lugar de "<<".
    cat << _EOF_
$TEXT_BOLD$TITLE$TEXT_RESET

$(system_info)
$(show_uptime)
$(drive_space)
$(home_space)

$TEXT_GREEN$TIME_STAMP$TEXT_RESET
_EOF_
}
```

```
# Procesar la línea de comandos del script para leer las opciones
while [ "$1" != "" ]; do
    case $1 in
        -f | --file )
            shift
            filename=$1
            ;;
        -i | --interactive )
            interactive=1
            ;;
        -h | --help )
            usage
            exit
            ;;
        * )
            usage
            exit 1
    esac
    shift
done

# Generar el informe del sistema y guardarlo en el archivo indicado
# en $filename
write_page > $filename
```

El funcionamiento de estas modificaciones es el siguiente:

1. Inicializamos la variable “interactive” a vacía. Esto viene a decir que el modo interactivo no está activo por defecto porque solo se activa si vale 1.
2. Después inicializamos la variable “filename” para que contenga el nombre de archivo de salida por defecto. Si no se especifica nada más en la línea de comandos, se usa este nombre de archivo.
3. **El inicializar previamente estas variables nos permite tener una configuración por defecto en caso de que el usuario no indique nada a través de la línea de comandos.**
4. A continuación, construimos un bucle **while** para que itere a través de todos los elementos de la línea de comandos y procese cada uno con un **case**. El comando **case** detectará cada posible opción y modificará las variables anteriores como corresponda.

El comando “shift”

¿Cómo funciona realmente este bucle?. Pues basándose en la magia del comando **shift**.

shift es un comando interno de la shell que opera sobre los parámetros posicionales. **Cada vez que se invoca a shift, este desplaza todos los parámetros posicionales, una**

---

unidad a la izquierda. Es decir, que $2 se convierte en $1, $3 se convierte en $2, $4 se convierte en $3 y así sucesivamente. Por ejemplo:

```
#!/usr/bin/env bash

echo "Empiezas con $# parámetros posicionales"

# Iterar hasta que se hayan usado todos los parámetros
while [ "$#" -gt 0 ]; do
    echo "Parámetro 1 igual a $1"
    echo "Ahora tienes $# parámetros posicionales"

    # Desplazar todos los parámetros una posición
    shift
done
```

Obteniendo el argumento de una opción

La opción “-f” del script que procesa la entrada, requiere un nombre de archivo válido como argumento: “-f nombre_de_archivo”. Es por esto que se usa shift otra vez para mover $1 al siguiente ítem de la línea de comandos —de “-f” al nombre del archivo— y asignarlo a “filename”.

Más adelante se debería comprobar el contenido de la variable “filename” para asegurarnos de si el archivo ya existe o no.

Otras cosas que hacer con los parámetros posicionales

La variable $@ contiene todos los parámetros de la lista de los parámetros posicionales.

```
#!/usr/bin/env bash

for i in "$@"; do
    echo $i
done
```

Con el comando **for** se puede usar el parámetro $@ en la lista de palabras para recorrer todos los argumentos de la línea de comandos. Esta técnica se usa frecuentemente para procesar listas de archivos proporcionadas a través de la línea de comandos.

Por ejemplo, prueba el siguiente script proporcionándole una lista de archivos o un argumento con comodines como ‘*’:

```
#!/usr/bin/env bash

for filename in "$@"; do
```

```
    result=
    if [ -f "$filename" ]; then
        result="$filename es un archivo regular"
    else
        if [ -d "$filename" ]; then
            result="$filename es un directorio"
        fi
    fi
    if [ -w "$filename" ]; then
        result="$result y es escribible"
    else
        result="$result y no es escribible"
    fi
    echo "$result"
done
```

El siguiente script compara los archivos de dos directorios y lista aquellos archivos del primer directorio que no están presentes en el segundo:

```
#!/usr/bin/env bash

# cmp_dir - programa para comparar dos directorios

# Comprobar que se han proporcionado los argumentos requeridos
if [ $# -ne 2 ]; then
    echo "usage: $0 directory_1 directory_2" 1>&2
    exit 1
fi

# Asegurar que ambos argumentos son directorios
if [ ! -d $1 ]; then
    echo "¡$1 no es un directorio!" 1>&2
    exit 1
fi

if [ ! -d $2 ]; then
    echo "¡$2 no es un directorio!" 1>&2
    exit 1
fi

# Procesa cada archivo en directory_1, comparándolo
# con los de directory_2
missing=0
for filename in $1/*; do
    fn=$(basename "$filename")
```

```
    if [ -f "$filename" ]; then
        if [ ! -f "$2/$fn" ]; then
            echo "$fn falta en $2"
            missing=$((missing + 1))
        fi
    fi
done
echo "$missing archivos perdidos"
```

Los parámetros posicionales también se usan al pasar argumentos a funciones. Si, por ejemplo se llama a una función así:

```
mi_funcion "Hola Mundo" 10 error.txt
```

Dentro de la función se accede a estos argumentos como $1, $2 y $3. Eso significa que dentro de una función no se tiene acceso directo a los argumentos de línea de comandos del script, si no se almacenan en una variable o se pasan como argumentos a la función.

Actividad 3

Es hora de parar un momento y trabajar sobre un ejercicio propuesto.

Recuerda actualizar el script para que admita las opciones "-f", "-i" y "-h" tal y como hemos visto. En caso de indicar "-h" se debe mostrar la ayuda, sin hacer nada más. Mientras que "-f" debe ir seguida del nombre del archivo dónde guardar el informe. Si no se usa la opción "-f", el archivo se guardará en `~/sysinfo.txt`.

Entrada de teclado

Por el momento nuestros scripts no son interactivos, es decir, no requieren ningún tipo de entrada por parte del usuario. A continuación veremos cómo hacer que nuestros scripts hagan preguntas y usen las respuestas

read

El comando **read** se puede usar para leer desde la entrada estándar. **Básicamente, toma la entrada estándar y la almacena en la variable indicada en el argumento**:

```
#!/usr/bin/env bash
echo -n "Di algo > "
read message
echo "Has dicho: $message"
```

Cuando se invoca el comando **read** con "message" como argumento, este espera a que el usuario introduzca una línea de texto —terminada en salto de línea— y la asigna a la variable "text".

Además, el comando **read** acepta diversas opciones:

- **Si se indica más de una variable en los argumentos del comando read, la línea introducida por el usuario se divide en palabras y cada una se asigna a una de las variables**, excepto la última que recibe el resto de la línea —p. ej. `read palabra1 palabra2 palabra3 resto`—.
- **'-t' seguido por un número de segundos indica el tiempo de espera máximo**. Esto significa que el comando **read** terminará si el usuario no proporciona una entrada antes del número de segundos especificados, en lugar de esperar indefinidamente.
- **'-s' indica al comando read que lo que escriba el usuario no debe ser mostrado**. Esto puede ser muy útil cuando se le pregunta al usuario por una contraseña u otra información confidencial.

```
#!/usr/bin/env bash

number=0

echo -n "Piensa un número y dímelo > "
read number

# Recuerda, $(( )) es para hacer operaciones aritméticas
if [ $((number % 2)) -eq 0 ]; then
    echo "El número es par"
else
    echo "El número es impar"
fi
```

Construyendo un menú

En un programa de consola, una forma común de crear una interfaz de usuario es usando menús. Un menú es una lista de opciones entre las que el usuario puede escoger.

En el siguiente ejemplo usamos lo que hemos aprendido sobre bucles y el comando **case** para crear un menú sencillo:

```
#!/usr/bin/env bash

selection=
until [ "$selection" = "0" ]; do
    echo "
    MENÚ DEL PROGRAMA
```

```
    1 - Mostrar el espacio libre en disco
    2 - Mostrar la memoria disponible

    0 - salir del programa
"
    echo -n "Introduzca su elección: "
    read selection
    echo ""
    case $selection in
        1 ) df ;;
        2 ) free ;;
        0 ) exit ;;
        * ) echo "Por favor, introduzca 1, 2 o 0"
    esac
done
```

- El propósito de usar el comando **until** es ciclar mostrando una y otra vez el menú cada vez que el comando de una elección haya terminado.
- El bucle continuará hasta que la elección sea igual a 0, momento en el que el comando **exit** hará terminar el programa.

El programa anterior se puede mejorar pidiendo al usuario que pulse ENTER después de que cada selección se haya completado y que limpie la pantalla antes de volver a mostrar el menú:

```
#!/usr/bin/env bash

press_enter()
{
    echo -en "\nPulsa ENTER para continuar"
    read
    clear
}

selection=
until [ "$selection" = "0" ]; do
    echo "
    MENÚ DEL PROGRAMA
    1 - Mostrar el espacio libre en disco
    2 - Mostrar la memoria disponible

    0 - salir del programa
"
    echo -n "Introduzca su elección: "
    read selection
```

```
    echo ""
    case $selection in
        1 ) df ;;
        2 ) free ;;
        0 ) exit ;;
        * ) echo "Por favor, introduzca 1, 2 o 0"
    esac
    press_enter
done
```

Comando select

Construir menús es lo suficientemente común como para que BASH nos ofrezca una solución alternativa y más cómoda mediante el comando **select**:

```
#!/usr/bin/env bash

selection=
select selection in HDD RAM Salir; do
    case $selection in
        HDD ) df ;;
        RAM ) free ;;
        Salir ) exit ;;
        * ) echo "Por favor, introduzca 1, 2 o 3"
    esac
done
```

En este caso, el comando **select** se encarga de mostrar el menú de forma cíclica, asignar un número a cada opción y guardar en la variable “selection” la elección del usuario.

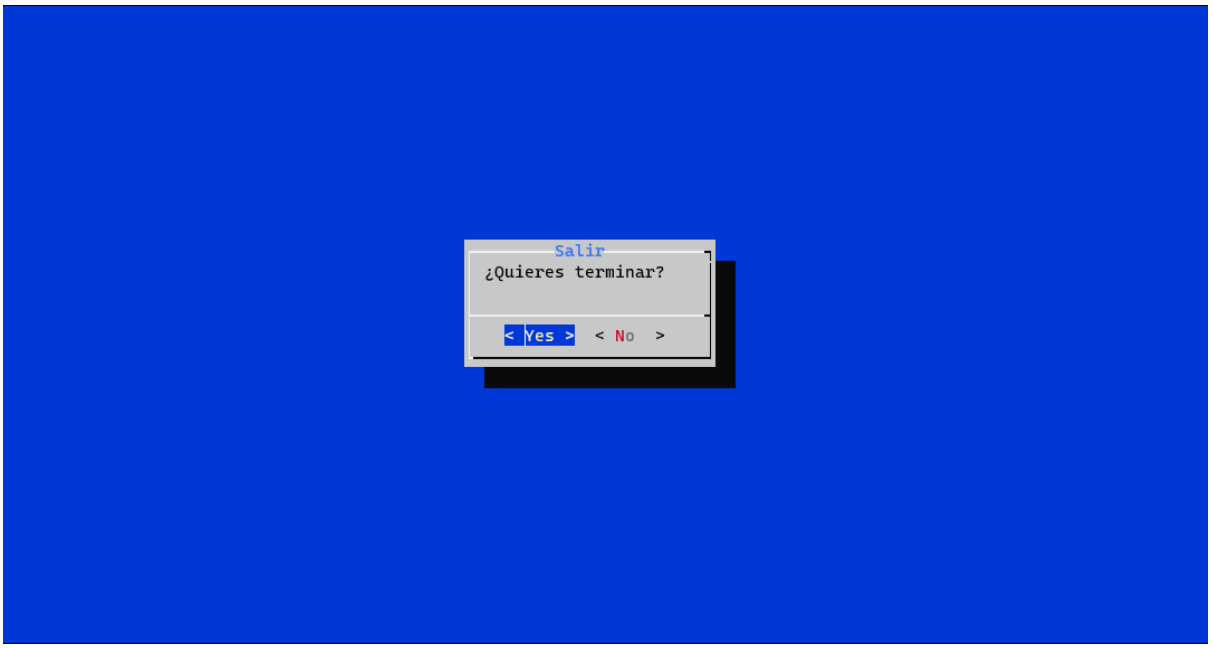
dialog

Podemos dar un toque más profesional al modo interactivo de nuestro script usando el comando dialog. Este comando permite crear cuadros de diálogo en la terminal. Solo es necesario indicarle el mensaje y el tipo de pregunta. La respuesta se puede leer de la salida de error o comprobando el estado de salida.

Por ejemplo, si ejecutamos esto:

```
if dialog --title "Salir" --yesno "¿Quieres terminar?" 6 25
then
    echo "Terminando..."
    exit 0
fi
```

obtendremos algo similar a esto:



Actividad 4

Es hora de parar un momento y trabajar sobre un ejercicio propuesto.

Implementar opción interactiva

Si el usuario llama al script con la opción "-i", este se ejecutará en modo interactivo. En caso contrario, se ejecutará en modo no interactivo, con el comportamiento que hemos implementado hasta ahora.

En el modo interactivo, antes de generar el informe:

1. Preguntará al usuario si el informe se mostrará por pantalla o se guardará en un archivo.

   ```
   Mostrar el informe del sistema en pantalla (S/N):
   ```

2. Si se debe guardar en un archivo, el script mostrará un mensaje como:

   ```
   Introduzca el nombre del archivo [~/sysinfo.txt]:
   ```

   De forma que entre corchetes se muestra el nombre de archivo que defecto, que se usará en caso de que el usuario pulse ENTER sin escribir ningún nombre de archivo. Ese nombre de archivo por defecto debe ser el indicado con la opción "-f". Si el usuario ejecutó el script sin usar dicha opción, el nombre por defecto será

`~/sysinfo.txt`.

3. Finalmente, el script comprobará si el archivo donde guardar el informe ya existe. Si ya existe, preguntará al usuario si desea sobreescribirlo:

   ```
   El archivo de destino existe. ¿Sobreescribir? (S/N)
   ```

   Si el usuario indica que no, el script termina sin hacer nada más. Si el usuario indica que sí, el archivo es sobreescrito al generar el nuevo informe.
