---

# BASH Unidad 4

Expansiones y entrecomillado

# Tabla de contenidos



# Expansión

Cada vez que se escribe un comando y se presiona la tecla ENTER, BASH realiza una serie de operaciones en el texto de la línea, que transforman el comando, antes de que sea ejecutado:

- A esto se lo denomina *expansión*.
- **Con la *expansión* en BASH, un usuario escribe "algo" que posteriormente es expandido antes de que lo interprete para tomar la acción indicada.**

Veamos un ejemplo con el **comando echo**. Este es un **comando interno** de la BASH (es implementado por la propia shell en lugar de ser un ejecutable externo) **que imprime por la salida estándar aquello que se le indica en los argumentos**:

---

```
yo@mihost:~$ echo Hola mundo
Hola mundo
```

Probemos con otro ejemplo:

```
yo@mihost:~$ echo *
bin Descargas  Desktop  Documentos  Dropbox  Imágenes  Insync
Música Vídeos  VirtualBox VMs  Workspace
```

**¿Por qué la salida no es la esperada?**

- Porque la shell *expande* '*' en algo diferente antes de ejecutar el comando "echo".
- Concretamente, justo después de pulsar la tecla ENTER y antes de ejecutar el comando `echo`, la shell automáticamente reemplaza '*' por todos los archivos (y directorios) en el directorio actual, porque '*' es un comodín.
- El comando nunca ve '*', sino solamente el resultado de la expansión. De ahí que sea ese resultado el que imprima

A continuación veremos los diferentes mecanismos de expansión utilizados por la BASH.

## Expansión del nombre de ruta (Pathname Expansion)

Este es el mecanismo por el que los comodines ('*', '?', etc.) son sustituidos. Supongamos que nuestro directorio personal tiene el siguiente contenido:

```
yo@mihost:~$ ls
bin Descargas  Desktop  Documentos  Dropbox  Imágenes  Insync
Música Vídeos  VirtualBox VMs  Workspace
```

Y ahora probemos algunas expansiones del nombre de ruta:

```
yo@mihost:~$ echo D*
Desktop  Documentos  Dropbox

yo@mihost:~$ echo *s
Descargas Documentos Imágenes Vídeos VirtualBox VMs

yo@mihost:~$ echo [[:upper:]]*
Descargas  Desktop  Documentos  Dropbox  Imágenes  Insync Música
Vídeos  VirtualBox VMs  Workspace

yo@mihost:~$ echo /usr/*/lib
/usr/local/lib /usr/X11R6/lib
```

**Si al intentar la expansión no se encuentran archivos y directorios cuya ruta encaje con el patrón, esta no tiene lugar:**

---

```
yo@mihost:~$ echo /usr/*/local
/usr/*/local
```

### Ejemplos con comandos

| Comando | Resultado |
|---|---|
| cp *.txt text_files | Copia todos los archivos en el directorio actual que terminan en ".txt" a un directorio denominado "text_files". |
| mv my_dir ../*.bak my_new_dir | Mueve el subdirectorio "my_dir" y todos los archivos que terminan en ".bak" en el directorio padre del directorio de trabajo a un directorio de nombre "my_new_dir". |
| rm *~ | Borra todos los archivos en el directorio actual que terminan en "~". |

## Expansión de tilde (Tilde Expansion)

El caracter tilde "~" tiene un significado especial:

- Cuando es usado al principio de una palabra, expande a la ruta del directorio personal de usuario indicado.
- Si no se especifica ninguna palabra, expande al directorio personal del usuario actual

```
yo@mihost:~$ echo ~
/home/yo

yo@mihost:~$ echo ~usuario
/home/usuario
```

**Si al intentar la expansión, el usuario indicado no existe, la expansión no tiene lugar.**

### Ejemplos con comandos

| Comando | Resultado |
|---|---|
| cd ~ | Cambia el directorio de trabajo al directorio personal del usuario actual. Es equivalente a invocar solamente **cd** |
| cd ~usuario | Cambia el directorio de trabajo al directorio personal del usuario "usuario" |

---

# Expansión aritmética

La shell permite que se puedan hacer operaciones aritméticas con enteros (no soporta flotantes) usando este tipo de expansión:

```
yo@mihost:~$ echo $((2 + 2))
4

yo@mihost:~$ echo $(( (5**2) * 3))
75

yo@mihost:~$ echo Cinco dividido por dos es igual a $((5/2))
Cinco dividido por dos es igual a 2


yo@mihost:~$ echo ...y sobra $((5%2)).
...y sobra 1.
```

# Expansión de llaves (Brace Expansion)

La expansión de llaves permite crear múltiples cadenas de texto a partir de un patrón que hace uso de llaves que pueden contener:

- Una lista de cadenas separadas por coma.
- O un rango de enteros o caracteres simples

```
yo@mihost:~$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back

yo@mihost:~$ echo Archivo_{1..3}
Archivo_1 Archivo_2 Archivo_3

yo@mihost:~$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Además, la expansión de llaves puede anidarse:

```
yo@mihost:~$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

## Ejemplos con comandos

Creemos directorios para ordenar nuestras fotos por fecha:

---

```
yo@mihost:~$ mkdir Imágenes
yo@mihost:~$ cd Imágenes
yo@mihost:~$ mkdir {2020..2025}-{01..12}
yo@mihost:~$ ls

2020-01  2021-01  2022-01  2023-01  2024-01  2025-01
2020-02  2021-02  2022-02  2023-02  2024-02  2025-02
2020-03  2021-03  2022-03  2023-03  2024-03  2025-03
2020-04  2021-04  2022-04  2023-04  2024-04  2025-04
2020-05  2021-05  2022-05  2023-05  2024-05  2025-05
2020-06  2021-06  2022-06  2023-06  2024-06  2025-06
2020-07  2021-07  2022-07  2023-07  2024-07  2025-07
2020-08  2021-08  2022-08  2023-08  2024-08  2025-08
2020-09  2021-09  2022-09  2023-09  2024-09  2025-09
2020-10  2021-10  2022-10  2023-10  2024-10  2025-10
2020-11  2021-11  2022-11  2023-11  2024-11  2025-11
2020-12  2021-12  2022-12  2023-12  2024-12  2025-12
```

## Expansión de parámetros (Parameter Expansion)

Esta característica, más útil en el desarrollo de scripts que en la línea de comandos, está relacionada con la capacidad que tiene el sistema de definir variables. Es decir, pequeñas porciones de datos identificadas por un nombre que son accesibles para los procesos.
Por ejemplo, la variable "USER" contiene el nombre del usuario que invoca la expansión de parámetros y su contenido se puede conocer así:

```
yo@mihost:~$ echo $USER
yo
```

Para ver la lista completa de variables se pueden usar los comandos printenv (o env sin más opciones).

```
yo@mihost:~$ printenv
```

🚨Es importante recordar que **si la variable no existe, la expansión tiene lugar sustituyendo la variable por una cadena vacía:**

```
yo@mihost:~$ echo $SUER

yo@mihost:~$
```

💡Por eso, para evitar problemas, **cuando desarrollemos scripts se recomienda activar la opción "-u"** de BASH, que nos avisará si intentamos utilizar una variable a la que no se le haya asignado previamente un valor:

---

```
yo@mihost:~$ echo $SUER
```
✅Por defecto, no hay error

```
yo@mihost:~$ set -u
```
⬅️Activar la opción "-u"

```
yo@mihost:~$ echo $SUER
bash: SUER: unbound variable
```
❌Error al usar una variable no asignada

## Sustitución de comandos

La sustitución de comandos nos permite usar la salida estándar de un comando como una expansión:

- Entre los paréntesis de la sustitución de comandos se pueden utilizar toda clase de expresiones, lo que significa que incluso se pueden emplear tuberías.
- Existe una sintaxis alternativa, usada en scripts antiguos, que usa las comillas invertidas `comand`.

```
yo@mihost:~$ echo $(ls)
bin Descargas  Desktop  Documentos  Dropbox  Imágenes  Insync
Música Vídeos  VirtualBox VMs  Workspace

yo@mihost:~$ ls -l $(type -P cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

En el caso anterior estamos pasando el resultado de **type -P cp** como un argumento a **ls**, proporcionándonos la información del ejecutable del comando **cp** sin que tengamos que conocer su ruta completa.

Una expresión equivalente sería:

```
yo@mihost:~$ ls -l `type -P cp`
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

En el siguiente ejemplo, el resultado de la tubería se convierte en la lista de argumentos del comando **file**:

```
yo@mihost:~$ file $(ls /usr/bin/* | grep bin/zip)
/usr/bin/zip:      ELF 64-bit LSB  executable, x86-64, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux
2.6.26, stripped
/usr/bin/zipcloak: ELF 64-bit LSB  executable, x86-64, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux
2.6.26, stripped
/usr/bin/zipgrep:  POSIX shell script, ASCII text executable
/usr/bin/zipinfo:  ELF 64-bit LSB  executable, x86-64, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux
2.6.26, stripped
```

---

```
/usr/bin/zipnote:  ELF 64-bit LSB  executable, x86-64, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux
2.6.26, stripped
/usr/bin/zipsplit: ELF 64-bit LSB  executable, x86-64, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux
2.6.26, stripped
```

# Entrecomillado

Ahora que entendemos como funcionan las expansiones, es hora de aprender a usar el entrecomillado para controlarlas, evitando que ocurran cuando no nos interesen:

```
yo@mihost:~$ echo esto es una    prueba
esto es una prueba

yo@mihost:~$ echo The total is $100.00
The total is 00.00
```

En el primer ejemplo, el mecanismo encargado de dividir las palabras eliminará los espacios entre palabras antes de pasar cada una como un argumento al comando **echo** (word-splitting). Mientras que en el segundo ejemplo, la expansión de parámetros sustituye "$1" por una cadena vacía, ya que se hace referencia a una variable indefinida.

## Comillas dobles

Si se pone un texto entre comillas dobles, todos los caracteres especiales pierden su significado:

- Las excepciones son "$", "\" barra invertida, "´" (comilla invertida).
- Por tanto, la división de palabras o *word-splitting* (que hace que se pierdan los espacios), la expansión del nombre de ruta, expansión de tilde y la expansión de llaves son ignoradas.
- Pero la expansión de parámetros, la expansión aritmética y la sustitución de comandos se mantienen.

```
yo@mihost:~$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

Usando comillas dobles se puede detener la división de palabras:

```
yo@mihost:~$ ls -l "two words.txt"
-rw-r-----  1 yo yo    4251 oct 21 19:38 two words.txt
yo@mihost:~$ mv "two words.txt" two_words.txt
```

Sin embargo, otros caracteres especiales, como "$", siguen manteniendo su significado:

---

```
yo@mihost:~$ echo "$USER $((2+2)) $(cal)"
yo 4     Octubre 2013
do lu ma mi ju vi sá
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

## Comillas simples

Si es necesario suprimir todas las expansiones, se usan comillas simples:

```
yo@mihost:~$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/yo/ls-output.txt a b foo 4 yo

yo@mihost:~$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 yo

yo@mihost:~$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

## Efecto de las comillas en la sustitución de comandos: "$()"

Por defecto, la división de palabras busca espacios, tabuladores y saltos de línea y los considera delimitadores entre palabras. Eso significa que los espacios, tabuladores y saltos de línea no entrecomillados no son parte del texto y se pierden, ya que sirven como separadores de los argumentos:

```
yo@mihost:~$ echo esto es una    prueba
esto es una prueba
```

Por ejemplo, en nuestro ejemplo anterior el comando **echo** recibe exactamente 4 argumentos: "esto" "es" "una" y "prueba". Si añadimos comillas, la división de palabras es desactivada:

```
yo@mihost:~$ echo esto "es una    prueba"
esto es un    prueba
```

haciendo que los espacios entre las comillas no sean considerados delimitadores, sino parte del argumento. Así que en ejemplo anterior el comando **echo** recibe exactamente 2 argumentos: "esto" y "es una  prueba".

El hecho de que los saltos de línea sean considerados delimitadores por el divisor de palabras, puede tener efectos interesantes en el mecanismo de sustitución de comandos

---

"$()":

```
yo@mihost:~$ cal
    Octubre 2013
do lu ma mi ju vi sá
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

yo@mihost:~$ echo $(cal)
Octubre 2013 do lu ma mi ju vi sá 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

yo@mihost:~$ echo "$(cal)"
    Octubre 2013
do lu ma mi ju vi sá
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

La primera sustitución sin comillas da como resultado 40 argumentos para el comando **echo**. Mientras que la segunda, usando las comillas, genera un único argumento que incluye espacios y saltos de línea.

## Escapado de caracteres mediante barra invertida

Cuando se quiere desactivar la expansión para un único caracter, en lugar de entrecomillarlo, se le puede preceder de una barra invertida o barra de escape:

```
yo@mihost:~$ echo "El saldo para el usuario $USER es: \$5.00"
El saldo para el usuario yo es: $5.00
```

Además, es común que se utilice el escapado para eliminar el significado especial que tienen ciertos caracteres para la shell cuando se usan en los nombres de archivo ("$", "!", "&", " ", etc.):

```
yo@mihost:~$ mv bad\&filename good_filename
```

Para permitir el uso de la barra invertida "\" en los nombres de archivo, esta debe escaparse a sí misma:

```
yo@mihost:~$ mv other_bad\\filename other_good_filename
```

---

Sin embargo, no debemos olvidar que, tal y como hemos comentado anteriormente, la barra invertida "\\" pierde su significado especial dentro de las comillas simples:

```
yo@mihost:~$ echo 'El saldo para el usuario $USER es: \$5.00'
El saldo para el usuario $USER es: \$5.00
```

## Otros usos de la barra invertida

La barra invertida puede tener otros usos en BASH:

- Para ignorar los saltos de línea al introducir comandos
- Para enviar caracteres especiales a la salida estándar con ayuda del comando **echo**

### Para ignorar los saltos de línea

Al consultar la ayuda de algunos comandos, se puede observar que existen opciones que consisten en un solo guión acompañado de una letra (p. ej. "-r") o de dos guiones seguidos de un nombre más largo (p. ej. "--reverse"):

```
ls -r
ls --reverse
```

Las opciones cortas son interesantes en la línea de comandos, cuando se quiere usar la shell de la forma más eficiente posible. Sin embargo, es aconsejable emplear las opciones largas en los scripts, ya que se entienden mejor en el caso de que haya que revisarlos meses después de haberlos escrito.

Obviamente, cuando se usa la forma larga es muy sencillo que el comando acabe ocupando demasiado. En ese caso se puede usar la barra invertida para informar a la shell que debe ignorar el salto de línea, de tal forma que asuma que el comando continúa en la línea siguiente:

```
ls -l \
   --reverse \
   --human-readable \
   --full-time
```

### Insertar caracteres especiales

La barra invertida también permite insertar caracteres especiales en el texto que se imprime en la salida estándar, con la ayuda de la sintaxis especial de entrecomillado `$'cadena'`. Por ejemplo, imprimiendo directamente la cadena de caracteres:

```
yo@mihost:~$ echo $'Insertando varias\n\n\nlíneas en blanco'
Insertando varias
```

---

```
líneas en blanco
yo@mihost:~$ echo $'\aMi ordenador \"emitió un pitido\".'
Mi ordenador "emitió un pitido".
```

O, también, usando variables:

```
yo@mihost:~$ texto=$'Palabras\tseparadas\tpor\ttabuladores'
yo@mihost:~$ echo "$texto"
Palabras        separadas       por     tabuladores
```

Los caracteres especiales más comunes son:

| Carácter de escape | Nombre | Posibles usos |
|---|---|---|
| `\a` | alerta | Hacer que la terminal emita un pitido, si el sistema lo soporta |
| `\b` | retroceso | Eliminar el carácter anterior |
| `\e`<br>`\E` | escape | Inserta un carácter de escape (en octal 033, hexadecimal 0x1B o decimal 27) |
| `\f` | salto de página | Enviando esto a una impresora se expulsa la página actual |
| `\n` | salto de línea | Cambiar de línea en el texto |
| `\t` | tabulador | Insertar tabuladores horizontales en el texto |
| `\\` | barra invertida | Insertar una barra invertida |
| `\'` | Comilla simple | Insertar una comilla simple |
| `\"` | Comilla doble | Insertar una comilla doble |
| `\nnn` | | Insertar el carácter de 8 bits cuyo valor en octal es *nnn*. |
| `\xHH` | | Insertar el carácter de 8 bits cuyo valor en hexadecimal es HH. |
| `\uHHHH` | | Insertar el carácter Unicode cuyo valor en hexadecimal es HHHH. |
| `\UHHHHHHHH` | | Insertar el carácter Unicode cuyo valor en hexadecimal es HHHHHHHH. |

---

### Alternativa con "echo -e"

Como alternativa, se puede usar el comando **echo** con su opción "-e" para interpretar secuencias de escape, aunque esta funcionalidad depende de la implementación específica de **echo**:

```
yo@mihost:~$ echo -e "Palabras\tseparadas\tpor\ttabuladores"
Palabras        separadas       por     tabuladores

yo@mihost:~$ echo -e "\aMi ordenador \"emitió un pitido\"."
Mi ordenador "emitió un pitido".

yo@mihost:~$ echo -e "DEL C:\\WIN10\\LEGACY_OS.EXE"
DEL C:\WIN10\LEGACY_OS.EXE
```

💡Utiliza preferentemente la sintaxis `$'cadena'`, ya que es una característica nativa de BASH, mucho más flexible y no depende de la implementación específica del comando **echo**, lo que hace tu código más portable y predecible.