

Tema 6 - Editores de Texto y Bash Scripting

ÍNDICE

1. INTRODUCCIÓN

2. EDITORES DE TEXTO

- 2.1 nano
- 2.2 vim

3. MI PRIMER SCRIPT

4. SCRIPTS: CONCEPTOS BÁSICOS

- 4.1 Estructura de un script
- 4.2 Cómo ejecutar un script en Bash

5. SINTAXIS DE SHELL SCRIPT

- 5.1 Caracteres Especiales
- 5.2 Uso de variables en los scripts
- 5.3 Argumentos de un script y variables especiales
- 5.4 Sustitución de comandos (command-substitution)
- 5.5 Visibilidad, ámbito o scope de las variables
- 5.6 Longitud de las variables
- 5.7 Concatenar cadenas de texto usando variables

6. OPERADORES EN SHELL SCRIPT

6.1 Operaciones aritméticas con números enteros

- 6.1.1 Método de los dobles paréntesis para operaciones aritméticas
- 6.1.2 Comando `expr` para operaciones aritméticas
- 6.1.3 Comando `let` para operaciones aritméticas

6.2 Operador bc (basic calculator): números reales

6.3 Comando `test`

6.4 Corchetes simples y dobles: `[]` y `[[]]`

6.5 Comando `read`

7. ESTRUCTURAS DE CONTROL

- 7.1 Estructuras Condicionales (if, case)
- 7.2 Bucles (for, while, until)**
- 7.3 Ruptura de Sentencias de Control (break, continue)

8. FUNCIONES

- 8.1 Valor de retorno de una función. Comando `return`
- 8.2 Valores de salida de una Función
- 8.3 Ámbito de las variables de un script y su uso con funciones
- 8.4 Paso de Parámetros especiales/posicionales a las funciones

9. ARRAYS

- 9.1 Declarando un "array"
- 9.2 Asignar un valor a una posición de un "array"
- 9.3 Accediendo a un "array"
- 9.4 Tamaño de un "array" y elementos no nulos de un "array"
- 9.5 Eliminar un array o una posición del array
- 9.6 Un *array* puede generarse a partir de la salida de un comando
- 9.7 Uso de bucles para recorrer un "array"
- 9.8 Arrays y operaciones con variables

10. ANEXO: Tipos de variables y accesibilidad a dichas variables

1. INTRODUCCIÓN

En este tema aprenderemos cuales son las características básicas de los editores de texto más extendidos para crear programas en Shell script de Linux.

Conoceremos además cuál es la estructura de un programa Linux de Shell script y aprenderemos a crear programas en Shell script (bash) para que puedan cumplir con la funcionalidad y los requisitos planteados.

Conocer Shell script en su versión bash es fundamental para poder automatizar tareas de supervisión, monitorización, gestión de usuarios, grupos, etc.

2. EDITORES DE TEXTO

En este tema se ven muy brevemente los editores de texto **nano** y **vim**. Se han elegido estos dos porque son muy comunes, y suelen venir preinstalados en las distribuciones más populares, por ejemplo, *nano* es el editor por defecto en distribuciones Debian o derivadas, como Ubuntu y vim lo es en las distribuciones Red Hat y derivadas, como CentOS. El objetivo de este tema es tener unas nociones básicas que nos permitan editar nuestros propios *scripts*.

2.1 nano

nano es un sencillo editor de texto. Se puede iniciar *nano* con un nuevo fichero o se puede editar un fichero ya existente:

```
# Inicia nano con un nuevo fichero (inicialmente vacío)
nano

# Edita un fichero ya existente
nano /etc/crontab

# Edita un fichero y muestra los números de línea
nano -l /etc/crontab

Cuidado al editar este fichero. Al final de este Tema crearemos un script y
practicaremos los editores.
Con `Ctrl + X` nos salimos

man nano
```

A continuación se especifican algunos de sus atajos de teclado (*shortcuts*) comunes, algunos de ellos aparecen en la parte inferior de la pantalla:

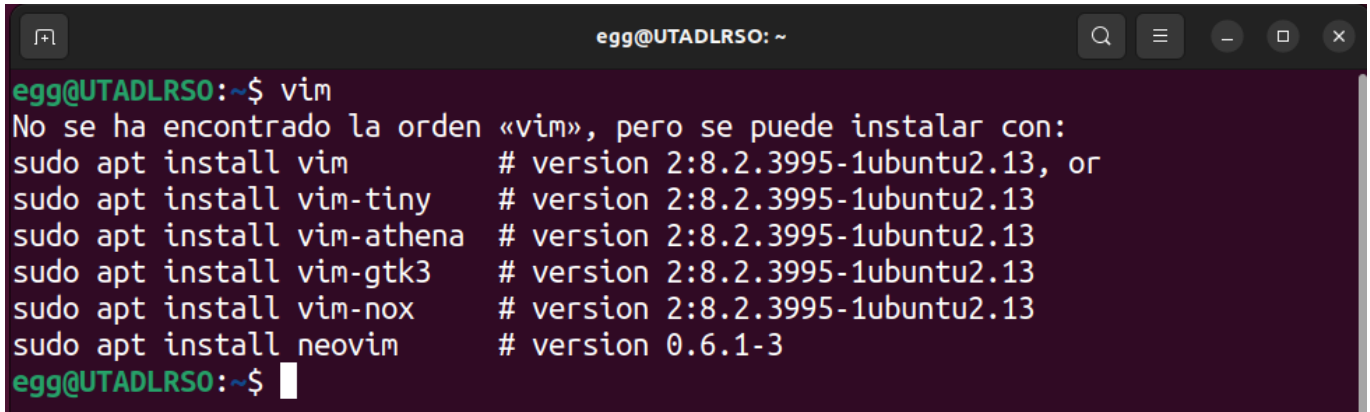
- **Ctrl + O**: Guarda el fichero que se está editando. Si no se especificó ningún fichero a editar, preguntará por el nombre de fichero que se quiere guardar.
- **Ctrl + X**: Sale de *nano*. Cuando hay cambios sin guardar, preguntará al usuario si desea guardar los cambios en el fichero. Y si no se especificó ningún fichero, preguntará por el nombre de fichero que se quiere guardar.
- **Ctrl + K**: Corta una línea, y la copia en el portapapeles.
- **Ctrl + U**: Pega el contenido del portapapeles.
- **Ctrl + W**: Busca un texto en el documento.
- **Ctrl + G**: Muestra la ayuda básica del programa.
- **Ctrl + _**: Ir a la línea (pregunta por el número de línea).
- **Alt + U**: Deshace la última edición.
- **Alt + 6**: Copia texto seleccionando.
- **Ctrl + W** y sobre todo **Ctrl + _** son importantes para buscar líneas de script con fallos.

2.2 vim

vim es un potente y completo editor de textos, que además permite ser ampliado mediante *plugins*.

Si tecleamos vim en Ubuntu, nos devuelve un mensaje de que no está instalado, y nos indica como instalarlo !!:

```
$vim
```

A screenshot of a terminal window titled 'egg@UTADLRSO: ~'. The prompt is 'egg@UTADLRSO:~\$'. The user has entered 'vim'. The terminal output shows a message: 'No se ha encontrado la orden «vim», pero se puede instalar con:' followed by a list of installation options with their versions: 'sudo apt install vim # version 2:8.2.3995-1ubuntu2.13, or', 'sudo apt install vim-tiny # version 2:8.2.3995-1ubuntu2.13', 'sudo apt install vim-athena # version 2:8.2.3995-1ubuntu2.13', 'sudo apt install vim-gtk3 # version 2:8.2.3995-1ubuntu2.13', 'sudo apt install vim-nox # version 2:8.2.3995-1ubuntu2.13', and 'sudo apt install neovim # version 0.6.1-3'. The prompt returns to 'egg@UTADLRSO:~\$'.

Para instalar vim en Ubuntu hay que ejecutar:

```
sudo apt install vim
```

```
# Si ejecutamos "vim" sin nombre de fichero entonces se inicia vim con un nuevo  
fichero vacío.  
$ vim
```

En la siguiente imagen se ve la información que aparece al arrancar el editor vim, se muestra unas pequeñas instrucciones para salir y para obtener ayuda. Observa que para salir debes escribir `:q` + tecla Intro.

Esta pantalla se presenta cuando vamos a crear un script nuevo (programa shell), desde el principio. Para hacer esto se ejecuta desde el prompt "vim".



```
# Si lo que queremos es editar un fichero ya existente:
```

```
$ vim ~/.bashrc
```

Nota: El fichero `.bashrc` incluye la definicion de **alias** (por ejemplo `la='ls -A'`) y los colores usado en el Terminal de Ubuntu.

Nota. ~ se obtiene en Windows se obtiene con Alt + 126 y en Linux con Alt Gr + ñ.
Tambien se obtiene en ambos con AltGr + 4

```
man vim
```

El manejo de *vim* es más complejo que el de *nano*. *Vim* tiene dos modos de funcionamiento:

- Modo de comandos (**ESC**). Las teclas que pulsamos, en lugar de aparecer escritas en el documento, son interpretadas por Vim como comandos y nos permiten realizar acciones como grabar, salir, copiar, pegar, etc
- Modo de inserción (**i**). Nos permite introducir caracteres en el fichero, en la posición actual del cursor, al estilo de los editores básicos a los que estamos acostumbrados

Al iniciar *vim*, se accede al modo de comandos. Para empezar a editar un fichero (y pasar al modo de inserción), se pueden pulsar diferentes teclas, como por ejemplo:

- **i: Insert** - Entra al modo de inserción (en el punto en el que esté el cursor).

- **a: Append** - Entra al modo de inserción (en el carácter siguiente al punto en el que esté el cursor).
- **s: Supress** - Entra al modo de inserción pero para suprimir (elimina el carácter en el que esté el cursor).

En la terminología de *vim*, lo que se ve en pantalla se denomina *buffer*. Por lo que cuando se guarda un fichero, en realidad se guarda el *buffer* en el fichero.

Desde el modo de inserción se puede editar el fichero, y cuando se quiera realizar alguna acción, se debe volver al modo de comandos (pulsando la tecla **ESC**).

Algunas de las operaciones básicas que se pueden hacer desde el **modo de comando** son:

- **:w**: Guarda el fichero. Si no se especificó ningún fichero a editar, preguntará por el nombre de fichero que se quiere guardar.
- **:q**: Sale de *vim*. Cuando hay cambios sin guardar, preguntará al usuario si desea guardar los cambios en el fichero. Y si no se especificó ningún fichero, preguntará por el nombre de fichero que se quiere guardar.
- **:q!**: Sale de *vim*, forzando la salida, de tal forma que se perderán los cambios no guardados.
- **:númeroDeLínea**: Ir a la línea *númeroDeLínea*. Por ejemplo: **:4**, va a la línea número 4 del fichero, y sitúa el cursor en el primer carácter del fichero.
- **dd**: Corta la línea actual (y la copia en el portapapeles).
- **p**: Pega el contenido que está en el portapapeles, en la línea siguiente a la actual (desplaza el texto existente hacia abajo).
- **P**: Pega el contenido que está en el portapapeles, en la línea actual (desplaza el texto existente hacia abajo).
- **Y**: Copia la línea actual en el portapapeles.
- **/**: Se usa la barra / seguida de un término de búsqueda, para buscarlo en el fichero.
- **n**: Mientras se está en una búsqueda, se desplaza a la siguiente ocurrencia del término de búsqueda en el fichero.
- **N**: Mientras se está en una búsqueda, se desplaza a la anterior ocurrencia del término de búsqueda en el fichero.
- **:%s/uno/otro/g**: Reemplaza el texto *uno* por el texto *otro* en todo el fichero.
- **u**: Deshace la última edición.

Fichero **.vimrc**

El fichero **\$HOME/.vimrc** almacena la configuración de *vim* para un usuario en particular. En este fichero se puede, por ejemplo, activar el resaltado de código, especificar a cuántos espacios se traduce un tabulador, mostrar los números de línea, etc.

Si no encuentras el fichero **.vimrc** en tu directorio "home" puedes crearlo para particularizar y configurar el uso de *vim* para tu usuario.

```
# Activa la sintaxis coloreada según el tipo de fichero de texto.
syntax on

# muestra las coincidencias en los resultados de la búsqueda
set showmatch
```

```
# búsqueda sin importar mayúsculas y minúsculas
set ignorecase

# para ver marcados los resultados de una búsqueda
set hlsearch

# para ver los primeros resultados de la búsqueda mientras la estás escribiendo
set incsearch

# cambia los tabs por espacios
set expandtab

# un tab son 4 espacios
set tabstop=4

# Auto indenta, es decir, cuando se pulsa intro se pone el cursor debajo del
principio de la línea de arriba.
set autoindent

# muestra el número de línea, útil para edición de código.
set number

#muestra el número fila y de columna en la línea de estado.
set ruler
```

Se deja aquí un ejemplo básico de fichero `.vimrc`:

```
syntax on
set showmatch
set tabstop=2
set hlsearch
set ruler
```


3. MI PRIMER SCRIPT

Un shell-script puede ser un simple fichero de texto que contenga uno o varios comandos.

Para ayudar a la identificación del contenido a partir del nombre del archivo, **es habitual que los shell scripts tengan la extensión ".sh"**, por lo que seguiremos siempre este criterio.

```
mi_script.sh
```

Para crear nuestro primer script vamos a describir brevemente su estructura. **La estructura básica de un shell-script es la siguiente:**

```
#!/bin/bash                                <-- Shebang
# Comentarios: líneas no interpretables    <-- Comentarios

echo "Hola Mundo"                          <-- Contenido del script
ls -l ~
```

El "**shebang**" permite especificar el intérprete de comandos con el que deseamos que sea interpretado el resto del script.

La sintaxis de esta línea es la secuencia #! seguida del ejecutable del shell deseado (ruta completa al ejecutable de la Shell).

NOTA: Profundizaremos sobre la estructura de un script en el siguiente apartado.

El lenguaje shell es un lenguaje interpretado, podemos por tanto editar y ejecutar **scripts** de forma que se lean las líneas de texto del fichero (terminadas en \n), se analizan y se procesan.

Recuerda que **para ejecutar un script debe tener permisos de ejecución generales**, si no los tuviese, para asignárselos bastaría ejecutar:

```
$chmod +x script.sh
```

Vamos a crear nuestro primer script

Recuerda que puedes incluir en tu script todos los comandos que se teclean manualmente en la entrada estándar, esto nos permite automatizar tareas repetitivas o realizar funciones.

Nuestro primer script será un "Hola mundo" y como editor de texto **usaremos nano por simplicidad:**

Escribimos:

```
$nano holamundo.sh
```

Nos aparecerá un fichero vacío que podemos editar:

```
#!/bin/bash
echo "Hola Mundo"
```

Vamos a crear ahora un script algo más diferente y vistoso:

```
$ nano PCinfo_y_todos_los_ficheros_de_Linux.sh
```

En nano escribimos:

```
#!/bin/bash

lscpu
#lscpu presenta la arquitectura de tu CPU, arquitectura (32/64 bits), modelo,
número de procesadores, compatibilidad con virtualización y memoria caché L1, L2 y
L3.

sleep 5
cat /proc/cpuinfo
# cpuinfo presenta toda la informacion de tu procesador

sleep 5

ls -lR /
```

Salvamos el fichero **Ctrl + O** y salimos de nano **Ctrl + X**

Para hacer que el script sea ejecutable (cambiar el modo a 'x') tecleamos:

```
chmod a+x PCinfo_y_todos_los_ficheros_de_Linux.sh
```

para ejecutarlo, necesario teclear delante del script **./**:

```
$ ./PCinfo_y_todos_los_ficheros_de_Linux.sh
```

!!Importante!: para parar el script teclear Ctrl + C

Prueba a hacer:

```
$ echo "Hola Mundo" >Hola.txt
$ ls -l
```

Podemos ver como, en el listado, el fichero Hola.txt aparece en color blanco, porque no es un ejecutable y como PCinfo_y_todos_los_ficheros_de_Linux.sh aparece en verde, al ser un fichero con permiso de ejecución. Los directorios aparecen en azul, como ya sabíamos.

Ejercicio de clase

Modificar el script ejecutable holamundo.sh para que imprima **Hola Mundo** y despues de 5 segundos imprima la configuracion IP de Ubuntu, mediante el comando **ip a**

Para ampliar

nano es un editor sencillo, mientras que **vim** es mucho más complejo, y dado que este tema es sólo una pincelada de ambas herramientas, se recomienda buscar información sobre ellos para sacarles todo el partido.

Opcionalmente, se puede investigar acerca de los (numerosísimos) *plugins* que existen para *vim*, o también buscar más información sobre cómo configurar el fichero *.vimrc*.

4. SCRIPTS: CONCEPTOS BÁSICOS

Ya hemos comentado que **los scripts, de forma similar a los programas, permiten automatizar tareas y agrupar comandos que, normalmente, se escribirían en la shell.** De hecho, cualquier comando o expresión que se puede ejecutar en la shell, puede ser ejecutado en un script, y viceversa.

Este tema es una introducción a la programación de *scripts de shell*, usando la shell de *Bash*. El contenido es eminentemente práctico.

4.1 Estructura de un script

Shebang: #!

Ya hemos adelantado que **la primera línea de un script se llama shebang y hace referencia al intérprete que ejecutará el contenido del fichero** (el código del *script*).

Por ejemplo, el *script* anterior `holamundo.sh` se compone de:

- **#!** : A estos dos caracteres en la primera línea de un *script* se les denomina **shebang**.
- **/bin/bash** : Junto al *shebang* se especifica el **intérprete** del *script* (en este caso, es **/bin/bash**).

Nosotros normalmente usaremos el intérprete `/bin/bash`, pero podrían usarse diferentes intérpretes de *shell*, como: `/bin/csh`, `/bin/zsh`, etc.

También se puede usar un *shebang* para ejecutar código escrito en algún otro lenguaje de *scripting*, como Python o Ruby.

Importante:

- Es imprescindible que sea la primera línea del script, ya que, en caso contrario, sería interpretado como un comentario (comienza con el carácter #).
- Puede haber espacios entre `#!` y el ejecutable del "shell".
- El shebang no es obligatorio (en caso de que no aparezca se intentará usar el mismo tipo de shell desde el que se ha invocado el script).

```
#!/usr/bin/python
#!/usr/bin/node (java)
#!/usr/bin/ruby
```

La sintaxis de los shell-scripts se caracteriza por ser bastante estricta en su escritura, especialmente en lo que se refiere a la inserción u omisión de espacios en blanco entre las palabras especiales. Ten esto muy en cuenta a la hora de escribir los scripts que se proponen.

4.2 Cómo ejecutar un script en Bash

Básicamente, **cuando se ejecuta un script en una shell, ésta creará un sub-proceso, dentro del cual se ejecuta el script.** El *script* necesitará tener permisos de ejecución.

No obstante, **la utilización del shebang y la creación de este sub-proceso está condicionada por la forma en que sea invocado el shell-script, existiendo 3 opciones:**

- **Explícita:** escribiendo explícitamente qué shell se desea invocar y pasando como argumento el nombre del script. Se crea un sub-proceso nuevo (se ignora el shebang).

```
$/bin/bash script_ejemplo.sh
```

- **Implícita:** invocando al script como si fuera un ejecutable, lo que requiere asignar permisos de ejecución al script. Se ejecuta con `./` porque el directorio local no está en la variable `PATH`. Se crea un sub-proceso nuevo.

```
$/script_ejemplo.sh
```

- **Implícita con `./`** (equivale a importar el script en la shell): el script será interpretado por el mismo proceso del shell responsable de la línea de comandos desde la que se invoca el script. Es decir, en este caso no se crea un sub-proceso.

```
$. script_ejemplo.sh
```

NOTA: Nosotros usaremos la forma "Implícita" de ejecución.

Vamos a preparar un script y probaremos las tres formas aprendidas de ejecución para observar los resultados.

Crear el script: `script_ejempl.sh`. Utilizar cualquiera de los editores `vim` o `nano` para editar el fichero.

```
$ vim script_ejemplo.sh
```

Escribir en el editor el siguiente contenido:

```
#!/bin/bash

echo Hola
# ps w: devuelve los procesos en ejecución en formato largo (wide) y los
subprocesos (f)
ps wf

# "$$" devuelve el id del proceso que se está ejecutando
echo "Proceso que ejecuta el script: $$"
```

Aseguramos de que el script tenga permisos de ejecución (al menos para el usuario) y ya podemos hacer las pruebas

```
$ chmod a+x script_ejemplo.sh
```

Vamos seguir practicando, ahora usaremos un here-document para crear un script.....

```
$ cd

# Crea el script 'hola.sh', mediante el método "here-document":
$cat <<SCRIPT >hola.sh
#!/bin/bash

echo "Hola, Bash"
cd
pwd
ls -la
SCRIPT

# Otorga permiso de ejecución al script:
$ chmod a+x hola.sh

# Ejecuta el script (se usa ./ porque el directorio actual no está en el $PATH):
$./hola.sh
```

5. SINTAXIS DE SHELL SCRIPT (Bash)

En este apartado veremos las características básicas de la sintaxis de bash para la creación de scripts: uso de variables, argumentos especiales y posicionales, caracteres especiales como \$, *, |, { }, [], etc, explicando el significado de cada uno de ellos en bash.

Se explicarán los diferentes tipos de entrecomillado con los que se trabaja, haciendo especial mención a sus diferencias (comillas dobles vs comillas simples).

Se estudiarán también las palabras reservadas, como construir expresiones, los operadores a utilizar así como la sintaxis de las sentencias de control.

5.1 Caracteres especiales

Son caracteres que tienen un significado especial para la shell. Algunos ya los conocemos como:

- ~ (directorio home)
- & (ejecución en background)
- | (pipe o tubería)
- * (asterisco o estrella de Kleen: comodín para expresiones regulares)
- ? (reemplazo de carácter)
- ! (negación)
- < y > (redirecciones)
- ; (secuenciar comandos)
- etc

Algunos otros importantes a la hora de hacer scripts son:

- **Dólar \$:** valor de variable
- **Almohadilla #:** Comentario
- **Backslash o carácter de escape \:** Indica que el siguiente carácter debe preservar su valor literal.
 - El carácter de escape se elimina de la línea una vez procesado.
 - Si aparece al final de una línea, significa "continuación de línea" e indica que el comando continúa en la siguiente línea (puede ser utilizado para dividir líneas muy largas).

```
$echo "Hola que tal \  
como estas"  
  
> Hola que tal como estas
```

- **Comillas simples ' ':** todo texto 'entrecomillado' con comillas simples mantendrá su valor literal, no se producirá ninguna ni sustitución y será considerado como una única cadena.
- **Comillas dobles " ":** si es un texto lo que va entrecomillado entonces son equivalentes a las comillas simples, en otro caso se producen sustituciones de variables por su valor.
- **Acento invertido o sustitución de comando `:** Sirve para expandir el resultado de un comando.

Un poco más acerca de las comillas dobles y simples:

```
# Comillas dobles dentro de comillas dobles. Se interpreta todo y desaparecen las
comillas dobles
echo " Aquí hay "comillas dobles" dentro de comillas dobles (escape)..."
>Se interpretan y desaparecen todas las comillas dobles.

# Para mantener "comillas dobles" dentro de comillas dobles - Se escapan comillas
dobles
echo " Aquí hay \"comillas dobles\" dentro de comillas dobles (escape)..."
> Se mantienen las comillas dobles, pero los backslash \ desaparecen.

# Comillas simples dentro de comillas dobles. Se imprime literalmente el texto
echo " Aquí hay 'comillas simples' dentro de comillas dobles..."
> Las comillas simples se mantienen

# Comillas dobles dentro de comillas simples. Se imprime literalmente el texto
echo ' Aquí hay "comillas dobles" dentro de comillas simples...'
> Las comillas dobles se mantienen.

# No puede haber comillas simples dentro de comillas simples, si quieres que se
mantengan
# se deben escapar las comillas simples con '\'' o se concatenan caracteres y se
usa ''''
echo ' Aquí hay '\''comillas simples'\'' dentro de comillas simples...'
echo ' Aquí hay' ''''comillas simples'''' 'dentro de comillas simples....'
>Para usar comillas simples dentro de comillas simples debemos usar la
concatenación de cadenas de caracteres y escape.
```

5.2 Uso de variables en los scripts

Además de las variables de entorno, se pueden definir y usar variables internas en un *script* (variables de *shell*).

Antes de definir nuestras propias variables recordemos algunas variables de entorno estudiadas ya en el Tema 2. Las variables de entorno, vienen definidas por defecto en Bash, se refieren al entorno en el que trabajas y recordemos que puedes ver su valor con el comando `env`.

Algunas variables interesantes como:

- `$SHELL` que indica el shell que estás ejecutando
- `$HOME` la ruta del usuario
- `$USERNAME` el nombre del usuario
- `$HOSTNAME` nombre de la máquina
- `$PATH` la ruta por defecto donde encontrar binarios
- etc.

No obstante podemos crear nuestras variables que podemos usar en nuestros scripts. Veamos la sintaxis básica de manejo de variables.

Operacion	Sintaxis
Sólo Definición	VAR=" " o VAR=
Definición y/o Inicialización/Modificación	VAR=valor
Expansión (Acceso a Valor)	\$VAR o \${VAR}
Eliminación de la variable	unset VAR

Ejemplos:

```
# Crea la variable MSG y le asigna un valor (signo = sin espacios)
MSG='Hola caracola'

# Hacer la prueba de dejar espacios. Ver los mensajes de error
MSG ='Hola caracola'
MSG= 'Hola caracola'

# Para usar el valor de una variable, se usa el carácter $ antes de su nombre:
echo "El mensaje es: $MSG"

# Es buena práctica además rodear la variable con {}
echo "El mensaje es el mismo: ${MSG}"
```

Recuerda:

Cuando se desea obtener el valor de una variable se debe usar el \$ y utilizar además las dobles comillas " .

Cuando se usan comillas simples ('), no hay susitución de variables, ya que las comillas simples construyen textos literales, es decir, las comillas simples no expanden el valor de las variables. Veamos algunos ejemplos:

```
# Como se ha visto, cuando se usan comillas dobles ("), se sustituye la variable
# por su contenido (para eso se usa el símbolo $).
echo "El mensaje es: $MSG"

# Cuando se usan comillas simples ('), no hay susitución de variables, ya que las
comillas simples construyen textos literales.
# Las comillas simples no expanden el valor de las variables. Veamos:
echo 'El mensaje es: $MSG'
```

Los nombres de las variables distinguen mayúsculas y minúsculas

```
#Los nombres de las variables distinguen mayúsculas y minúsculas
msg='A message to you, Trudy'
MSG='Mrs. Gullible'
```

```
echo "msg = ${msg}"  
echo "msG = ${msG}"
```

Existe la posibilidad de utilizar la función `command-substitution` (función que veremos más adelante, permite ejecutar un comando) y almacenar su salida en una variable. **Esta función ocurre cuando utilizamos el símbolo del `$` o el acento invertido: ```**

```
# Se puede usar la sustitución de comandos para almacenar la salida de un  
# comando en una variable (command-substitution):  
CURR_DIR=$( pwd )  
echo "El directorio actual es: ${CURR_DIR}"  
  
# En este caso los espacios no importan  
CURR_DIR=$( pwd )  
CURR_DIR=$(pwd )  
  
# El resultado es el mismo si la sustitución de comandos se rodea con comillas  
# dobles:  
CURR_DIR="$( pwd )"  
echo "El directorio actual sigue siendo: ${CURR_DIR}"  
  
cd  
MI_DIR=`pwd`  
echo "Mi directorio de trabajo es: $MI_DIR"
```

5.3 Argumentos de un script y variables especiales

En Bash, hay algunas variables especiales que están definidas por defecto, y **que se utilizan para pasarlas como argumentos o parámetros de entrada a la hora de ejecutar los scripts.**

Hay **otras que refieren al propio script, al proceso que ha ejecutado el script, o a la máquina en la que se ha ejecutado el script.** Así, debemos conocer las siguientes:

- `$0` representa el nombre del script
- `$1` – `$9` los primeros nueve argumentos que se pueden pasar a un script en Bash
- `$#` el número de argumentos que se pasan a un script
- `$@` Lista de todos los argumentos que se han pasado al script
- `$*` Cadena con el contenido completo de los parámetros pasados al script
- `?` código de salida del último proceso/comando que se ha ejecutado
- `$$` es el ID del proceso asociado a la ejecución del script

NOTA: La diferencia entre `$*` y `$@` es que en este último podemos iterar sobre los valores devueltos, mientras que en el primero no al constar de un solo valor.

Veamos un ejemplo:

```
cd

cat <<'SCRIPT' >args.sh
#!/bin/bash

echo "-----"
echo "El nombre de este script es: $0"
echo "El primer argumento es: $1"
echo "El segundo argumento es: $2"
echo "El número de argumentos es: $# "
echo "Todos los argumentos son: $@"

# Puedo hacer copias de los argumentos:
FIRST_ARG=$1
echo "El primer argumento es = ${FIRST_ARG}"
echo
SCRIPT

chmod a+x args.sh

./args.sh
./args.sh "Esto es Bash!"
./args.sh "¿Cuál es la respuesta?" 42
./args.sh "Tortilla" "Cebolla" "Patata" 5
```

NOTA: Un detalle importante acerca de este *script*, es que al usar el método *here-document* **el primer campo de terminación aparece entre comillas simples (')**. Esto se hace para evitar que se sustituyan las **variables (como \$1, \$2, ...)** por su contenido en el momento de generar el fichero *args.sh*.

El contenido del fichero *args.sh* será:

```
#!/bin/bash

echo "-----"
echo "El nombre de este script es: $0"
echo "El primer argumento es: $1"
echo "El segundo argumento es: $2"
echo "El número de argumentos es: $# "
echo "Todos los argumentos son: $@"

# Puedo hacer copias de los argumentos:
FIRST_ARG=$1
echo "El primer argumento es = ${FIRST_ARG}"
echo
```

5.4 Sustitución de comandos (command-substitution)

Sirve para ejecutar un comando *in-situ* y utilizar su salida. Para realizar una sustitución de comandos, se usa la construcción `$()`. Para ejecutar un comando también se puede utilizar el acento invertido ```

Hemos visto más arriba como almacenar el resultado en una variables.

```
echo "Estamos a $( date )."

CURR_DIR=$( pwd )
ls -la ${CURR_DIR}

MI_DIR=`pwd`
echo $MI_DIR
```

5.5 Visibilidad, ámbito o scope de las variables

Una variable declarada dentro de un *script* sólo es visible por ese *script*: es una **variable de shell**. Si se desea que los hijos de ese *script* tengan acceso a las variables, hay que **"exportarlas"**. Cuando se exporta una variable de *shell*, **se convierte en una variable de entorno**, que es visible desde los procesos hijos.

Por ejemplo, un *script* (*sh1.sh*) llama a otro *script* (*sh2.sh*), y se desea que una variable declarada dentro de *sh1.sh* sea visible desde *sh2.sh*: hay que **exportar** la variable desde *sh1.sh*. El comando o palabra para exportar una variable es **"export VAR"**.

Creamos *sh1.sh*

```
cat <<'SH1' >sh1.sh
#!/bin/bash

# File: sh1.sh

echo "====> Inicio de ejecucion de sh1"
var1='Galaxia'
var2=44444

# Exportamos la variable que contiene el directorio actual
export CURR_DIR=$( pwd )
echo "[$0]: ${CURR_DIR}"

echo -e "[$0]: var1 = ${var1} \t var2 = ${var2}"

# Exportamos var1
export var1

# Ejecutamos sh2 desde sh1
./sh2.sh

echo -e "[$0]: var1 = ${var1} \t var2 = ${var2}"
```

```
SH1
```

Creamos `sh2.sh`

```
cat <<'SH2' >sh2.sh
#!/bin/bash

# File: sh2.sh
echo -e "\n====> Inicio de ejecucion de sh2"

echo "$0: ${CURR_DIR}"

echo -e "$0: var1 = ${var1} \t var2 = ${var2}"

var1='Constelación'
export var1

echo -e "$0: var1 = ${var1}"

echo -e "Fin de sh2.... \n"

SH2
```

El contenido de los ficheros `sh1.sh` y `sh2.sh` sería:

- **sh1.sh:**

```
#!/bin/bash

# File: sh1.sh

echo "====> Inicio de ejecucion de sh1"
var1='Galaxia'
var2=44444

# Exportamos la variable que contiene el directorio actual
export CURR_DIR=$( pwd )
echo "$0: ${CURR_DIR}"

echo -e "$0: var1 = ${var1} \t var2 = ${var2}"

# Exportamos var1
export var1

# Ejecutamos sh2 desde sh1
./sh2.sh
```

```
echo -e "[\$0]: var1 = \${var1} \t var2 = \${var2}"
```

- **sh2.sh:**

```
#!/bin/bash

# File: sh2.sh
echo "====> Inicio de ejecucion de sh2"

echo "[\$0]: \${CURR_DIR}"

echo -e "[\${0}]: var1 = \${var1} \t var2 = \${var2}"

var1='Constelación'
export var1
echo -e "[\$0]: var1 = \${var1}"

echo -e "Fin de sh2.... \n"
```

Vamos a ejecutarlos y observa bien que pasa con las variables. Hacemos que ambos scripts tengan permisos de ejecución y ejecutamos **sh1.sh**:

```
chmod a+x sh1.sh
chmod a+x sh2.sh

# Ejecutamos sh1.1 (siempre de forma implícita)
./sh1.sh
```

Eliminar el valor de una variable

Además de exportar una variable, también puede eliminarse (su contenido pasa a estar indefinido, *null*). Para ello se usa el comando **unset**.

```
export myVar='Andrómeda'
echo "myVar = \${myVar}"

unset myVar
echo "myVar = \${myVar}"
```

5.6 Longitud de las variables

La forma nativa de obtener la longitud de una variable en bash es:

`${#var}`

Creamos un script llamado `len.sh`:

```
cat <<'LEN' >len.sh
#!/bin/bash

# File: len.sh

msg='Tortilla de patatas'
echo "La longitud de '${msg}' es: ${#msg}"

num=7251
echo "La longitud del número '${num}' es: ${#num}"

LEN
```

El contenido del fichero `len.sh` es:

```
#!/bin/bash

# File: len.sh

msg='Tortilla de patatas'
echo "La longitud de '${msg}' es: ${#msg}"

num=7251
echo "La longitud del número '${num}' es: ${#num}"
```

Damos permiso de ejecución y probamos. Observa el resultado:

```
chmod a+x len.sh
./len.sh
```

Más pruebas con algunas variables de entorno útiles

Creamos usando here-document un script denominado `env_vars_demo.sh`

```
cat <<'ENVVARS' >env_vars_demo.sh
#!/bin/bash
```

```
#File: env_vars_demo.sh
echo "Directorios de la variable 'PATH': $PATH"

cd
echo "Accediendo al directorio: $( pwd ) ..."
echo "    Status = $?"          # Devuelve 0

echo "Accediendo al directorio: /root ..."
cd /root
echo "    Status = $?"          # Devuelve 1

echo "Este script tiene el PID: $$"
echo "Este script ha sido ejecutado por: $USER"
echo "Este script se ejecuta en el host: $HOSTNAME"

sleep 2
echo "Este script lleva ejecutándose ${SECONDS} segundos"

# La variable RANDOM genera un número pseudo-aleatorio entre 0 y 32767.
echo "Un número aleatorio: ${RANDOM}"
echo "Otro número aleatorio: ${RANDOM}"
echo "Y otro número aleatorio más: ${RANDOM}"

echo "Esta es la línea $LINENO del script."

ENVVARS
```

El contenido del fichero `env_vars_demo.sh` sería:

```
#!/bin/bash

# File: env_vars_demo.sh
echo "Directorios de la variable 'PATH': $PATH"

cd
echo "Accediendo al directorio: $( pwd ) ..."
echo "    Status = $?"

cd /root
echo "    Status = $?"

echo "Esto utilizando la shell: $SHELL"
echo "Este script tiene el PID: $$"
echo "Este script ha sido ejecutado por: $USER"
echo "Este script se ejecuta en el host: $HOSTNAME"

sleep 2
echo "Este script lleva ejecutándose ${SECONDS} segundos"

# La variable RANDOM genera un número pseudo-aleatorio entre 0 y 32767.
echo "Un número aleatorio: ${RANDOM}"
```



```
echo "Otro número aleatorio: ${RANDOM}"  
echo "Y otro número aleatorio más: ${RANDOM}"  
  
echo "Esta es la línea $LINENO del script."
```

Ejecuta y observa el resultado prestando atención al valor que toman las variables utilizadas.

```
chmod a+x env_vars_demo.sh  
./env_vars_demo.sh
```

5.7 Concatenar cadenas de texto usando VARIABLES

Crea el siguiente script y ejecutalo:

```
#!/bin/bash  
cadena1='La unión hace '  
cadena2='la fuerza'  
cadena3=$cadena1$cadena2  
echo "$cadena3"
```

Resultado:

```
La unión hace la fuerza
```

6. OPERADORES DE SHELL SCRIPT

6.1 Operadores aritméticos (números enteros)

Por defecto, una variable en Bash, independientemente de su contenido, **es tratada como una cadena de texto** y no como un número. Sin embargo, y gracias a la evolución de bash podemos hacer operaciones aritméticas aunque es importante aclarar que **bash sólo opera con enteros..**

Para realizar operaciones matemáticas con números reales necesitaremos utilizar `bc` (lo veremos más adelante).

Así, *bash* ofrece varias formas de realizar operaciones aritméticas con números enteros:

- *let*
- *expr*
- La expansión con dobles paréntesis.

Algunas de las operaciones matemáticas más comunes son:

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto o Módulo
++	Pre/Post-incremento (incrementa 1)
--	Pre/Post-decremento (decrementa 1)
**	Potencia

Para nuestras operaciones **usaremos la expansión mediante los dobles paréntesis**.

6.1.1 Método de los dobles paréntesis para operaciones aritméticas

Este método realiza la expansión de una expresión aritmética que se encuentra dentro de los dobles paréntesis: **((*expresion*))** si además queremos devolver el resultado en una operación entonces precedemos los dobles paréntesis del **\$**

A continuación veamos algunos ejemplos de varias operaciones matemáticas sencillas. Para evaluar la expresión, se rodea dicha expresión matemática con dobles paréntesis, precedidos del símbolo **\$**, es decir **\$((*expr*))**. Esto permite además incluir una operación matemática dentro de otra.

```
((x=3+3))
echo $x
```

```
((x++))
echo $x

((y=z+10)); echo $y

# Y si usamos el $ delante de los dobles paréntesis:

echo $((z=25))
echo $((z++))
echo $((z=z+10))
```

Vamos a crear ahora el siguiente script `math_ops.sh`

```
cd
cat <<'MATH' >math_ops.sh
#!/bin/bash

# File: math_ops.sh

# Se usa expansión: con dobles paréntesis que rodean a una expresión matemática
a=$((3+4))
echo "a = ${a}"

# Es buena práctica usar espacios para mejorar la claridad y la legibilidad
a=$(( 40 + 2 ))
echo "Ahora, a = ${a}"

# Se pueden usar variables como operandos de las expresiones matemáticas.
b=$(( $a - 10 ))
echo "b = ${b}"

# Dentro de los dobles paréntesis se puede omitir el símbolo $:
b=$(( 10 + a + 10 ))
echo "Ahora, b = ${b}"

# Se puede usar ++ para incrementar en 1.
(( b++ ))
echo "Autoincremento: b = ${b}"

# Como en otros lenguajes, se puede sumar y asignar en una sentencia
(( b += 5 ))
echo "Suma y asigna: b = ${b}"

# Operaciones comunes
a=$(( 5 + 7 ))
echo "5 + 7 = ${a}"

a=$(( 5 - 7 ))
echo "5 - 7 = ${a}"

a=$(( 5 * 7 ))
```

```
echo "5 * 7 = ${a}"

a=$(( 5 / 7 ))
echo "5 / 7 = ${a}"

a=$(( 7 / 5 ))
echo "7 / 5 = ${a}"

a=$(( 7 % 5 ))
echo "7 % 5 = ${a}"

a=$(( 5 ** 7 ))
echo "5 ** 7 = ${a}"

MATH
```

El contenido del fichero `math_ops.sh` es:

```
#!/bin/bash

# File: math_ops.sh

# Se usa sustitución: con dobles paréntesis que rodean a una expresión matemática
a=$((3+4))
echo "a = ${a}"

# Es buena práctica usar espacios para mejorar la claridad y la legibilidad
a=$(( 40 + 2 ))
echo "Ahora, a = ${a}"

# Se pueden usar variables como operandos de las expresiones matemáticas.
b=$(( $a - 10 ))
echo "b = ${b}"
# Se puede omitir el símbolo $:
b=$(( 10 + a + 10 ))
echo "Ahora, b = ${b}"

# Se puede usar ++ para incrementar en 1
(( b++ ))
echo "Autoincremento: b = ${b}"

# Como en otros lenguajes, se puede sumar y asignar en una sentencia
(( b += 5 ))
echo "Suma y asigna: b = ${b}"

# Operaciones comunes
a=$(( 5 + 7 ))
echo "5 + 7 = ${a}"

a=$(( 5 - 7 ))
echo "5 - 7 = ${a}"
```

```

a=$(( 5 * 7 ))
echo "5 * 7 = ${a}"

a=$(( 5 / 7 ))
echo "5 / 7 = ${a}"
a=$(( 7 / 5 ))
echo "7 / 5 = ${a}"

a=$(( 7 % 5 ))
echo "7 % 5 = ${a}"

a=$(( 5 ** 7 ))
echo "5 ** 7 = ${a}"

```

Prueba a ejecutarlo y observa el resultado:

```

chmod a+x math_ops.sh
./math_ops.sh

```

6.1.2 Comando **expr** para operaciones aritméticas

Comando antiguo de Unix. Esta en desuso. Hay que dejar espacios alrededor de los operadores y operandos

```

var=$(expr 1+1) # da como resultado 1+1 (y no 2 como esperaríamos).
echo $var

# Lo correcto es:
var=$(expr 1 + 1)
echo $var

# el * hay que escaparlos:
var=$(expr 5 \* 3).

```

Necesario usar con precaución, mejor usar las otras opciones.

6.1.3 Comando **let** para operaciones aritméticas

Otra opción para realizar operaciones matemáticas es **let**.

let valua cada argumento como una expresión aritmética. Las operaciones se evalúan como enteros, mientras que una división por cero arrojará un error.

let

```

let z=14; let z++; let z=z+10; echo "$z"          # Devuelve 25

```

6.1.4 Concatenación de cadenas (en variables) para operaciones aritméticas

```
#!/bin/bash
# Operaciones usando la concatenación de cadenas de texto
op='+'
echo $op
a=5
echo $a
b=6
echo $b
res=$(( ${a}${op}${b} ))
echo " $a $op $b = $res"
```

```
5 + 6 = 11
```

6.2 Operador bc (basic calculator): números reales

El comando **bc** se puede considerar **un lenguaje de programación en si mismo**, y como tal tiene muchas opciones y cierta complejidad al igual que el comando **awk** que vimos en temas anteriores.

bc permite las **operaciones con números con decimales, en punto flotante, y además incluye varias funciones matemáticas**

Al igual que **awk** soporta un modo interactivo:

- **bc -i**: para iniciar el modo interactivo
- **bc -l**: para iniciar **bc** y poder utilizar las funciones de su librería matemática
- **quit**: para salir del modo interactivo
- **Se pueden crear programas/scripts en bc** (no es obligatorio que la extensión sea **.bc**, es más un convenio): **bc -l script.bc**. La sintaxis del lenguaje es sencillo pero ligeramente diferente a la sintaxis de **bash**. Se pueden utilizar funciones como **print()** y **read()** para entrada y salida de datos. No será objeto de estudio en este tema.
- Los comentarios en **bc** se escriben así: */ esto es un comentario /*

Nosotros **utilizaremos el modo NO interactivo, para usar bc en nuestros bash scripts**

En **Ubuntu** está incluido, pero en otras distribuciones, por ejemplo en **CentOS**, no está incluido.

Soporta estructuras de control (bucles) y condicionales (**if**). "**bc**" incluye las siguientes funciones especiales:

Funciones Especiales (bc)

sqrt ()

Sirve para calcular raíces cuadradas

scale ()

Esta funcion determina el numero de decimales con los que se va a trabajar. Por defecto es 0. Para asignar un valor a scale haremos: **"scale=valor"**

length ()

Indica el número total de dígitos significativos con los que se va a trabajar.

ibase y obase

Definen la base de los números de entrada y salida, por defecto es 10. Valores de base 2 a 36, soportan de 0-9 y de A -Z.

Funciones de las librería matemática

Las funciones de la librería matematica se invocan con (-l), son las siguientes:

- s(x) seno de x, en radianes
- c(x), coseno de x en radianes
- a(x), arco tangente, devuelve un valor en radianes
- l(x), logaritmo natural (base e)
- e(x), función exponencial
- j(n, x), función de Bessel de orden n en x, devuelve el orden n, de la función de Bessel

Operadores aritméticos y Expresiones básicas

- - **expr**: se usa el guión para la negacion de la expresión
- ++**var**, --**var** Pre incremento/decremento de la variable en 1, y este es el valor de la expresión
- **var++**, **var--** El resultado de la expresión es su valor, y despues se Pos incrementa/decrementa.

(expr) El uso de los **paréntesis fuerza la evaluación de la expresión en primer lugar**. Es importante para asegurar el orden de ejecución de la expresión si no se conoce la prioridad de la precedencia de los operadores.

- expr + expr (suma)
- expr - expr (resta)
- expr / exp (división)
- exp % expr (resto)
- exp * exp (multiplicación)
- expr1 ^ expr2 . Potencia de base "expr1" y exponente "expr2". En este último caso el exponente (expr2) debe ser entero.

Con bc también se pueden utilizar los operadores de asignación con la siguiente sintaxis:

- var += valor >> var = var + valor
- var -= valor >> var = var - valor
- var *= valor >> var = var * valor
- var /= valor >> var = var / valor
- var ^= valor >> var = var ^ valor
- var %= valor >> var = var % valor

Operadores y Expresiones relacionales de comparación:

Son:

- `expr1 < expr2`, Menor: el resultado es 1 si `expr1` es estrictamente menor que `expr2`
- `expr1 > expr2`, Mayor
- `expr1 <= expr2`, Menor o igual
- `expr1 >= expr2`, Mayor o igual
- `expr1 == expr2`, Igual
- `expr1 != expr2`, Distinto

Operaciones booleanas:

- `!exp`, es la negación de la expresión. El resultado es 0 si la expresión es distinta de cero.
- `exp && exp`, AND, el resultado es 1, si ambas no son cero
- `exp || exp`, OR, el resultado es 1, si una no es cero

Soporta algunas estructuras de control:

Soporta bucles y sentencias condicionales: `while`, `for`, `if`

Como usar bc en scripts:

El comando **"bc"** funciona como una calculadora, es decir le podemos pasar una serie de operaciones y/o expresiones y `bc` evalúa el resultado.

Comando "echo" usando bc:

La forma más sencilla y común de usar `bc` en scripts es con un comando `echo`, que imprima una expresión (como resulta de diferentes operaciones), y la pasa al comando `bc` a través de la tubería (`|`).

Sintaxis: `bash echo "expresion" | bc [-l]`

La expresión puede contener las funciones y operadores vistos más arriba.

Veamos algunos ejemplos de como utilizar `bc`:

```
echo "6.5 / 2.7" | bc
```

El resultado será 2, debido a que por defecto `bc` **no** funciona con números en coma flotante. Para que los soporte usamos la sentencia `scale` (número de decimales). Asignamos a `scale` el valor de decimales que queremos utilizar.

Podemos separar distintas operaciones mediante `;"`

```
# Podemos separar distintas operaciones mediante ";"
```



```
echo "scale=2; 7 / 5" | bc
echo "scale=2; 355/113" | bc
echo "scale=6; 355/113" | bc
```

Calcular el número de decimales o el número de dígitos de un número

Veamos como averiguar el número de decimales o el número de dígitos de un valor.

El número, .12345 tiene una longitud (length=) de 5 y una escala (scale=) de 5, mientras que el número 12345.67890 tiene una longitud de 10 y una escala de 5

```
echo "length(.12345)" | bc
echo "scale(.12345)" | bc

echo "length(12345.12345)" | bc
echo "scale(12345.12345)" | bc
```

Y ejecutar comparaciones:

```
echo "2 > 1" | bc          # Devuelve 1
echo "1 != 2" | bc        # Devuelve 1
echo "2 > 1 && 2 != 1" | bc # Devuelve 1
echo "2 < 1 || 1 != 1" | bc # Devuelve 0
echo "! 0" | bc           # Devuelve 1
echo "! 3" | bc           # Devuelve 0
```

Asignar el resultado a variables

Otra forma útil de usar **bc** en scripts, es asignar a una variable al resultado del valor obtenido de la calculadora (echo | bc), como se hace en el siguiente ejemplo.

```
z=$(echo "4.1+5.2" | bc)
echo $z

# Es lo mismo que...
z=$(echo "4.1+5.2" | bc);echo $z
```

Consejos y recordatorios sobre el uso de bc

bc

A veces, no es intuitivo usar variables dentro de una sentencia que usa bc, ni como almacenar en una variable los resultados. Vamos a ver un ejemplo.

Incluir la variable x mediante (\$x) en una función.

```
# Calculo del seno hiperbólico de 3
# El seno hiperbólico está definido mediante la siguiente ecuación:  $\sinh(x) = (e^x - e^{-x})/2$ 

x=3                                # x vale 3
y=$(( -1*$x ))                    # y vale -3
echo " scale=3; shx=$(( (e($x)-e($y))/2 )); shx " | bc -l
```

La forma más elegante y potente de usar bc dentro de scripts es almacenando los resultados en variables globales, que pueden ser usadas posteriormente. Los valores almacenados en variable internas de bc se pierden.

Vamos a probarlo:

```
# Calculamos como antes el seno hiperbólico de 3
x=3
y=$(( -1*$x ))
echo " scale=3; shx=$(( (e($x)-e($y))/2 )); shx " | bc -l
20.036                          # Resultado

echo "$shx"                     # Pierde su valor, por se variable interna a bc (no tiene ningún
valor)

# La solución para usar el resultado de una expresión es almacenarlo en una
variable global:
senhx=$((echo " scale=3; shx=$(( (e($x)-e($y))/2 )); shx " | bc -l))
echo $senhx                     # Mantiene el valor
echo $shx                       # No tiene valor
```

Más ejemplos (sólo para ampliar):

- 1. Calcular el seno (pi), con 1000 decimales, cuando pi tiene 5 decimales $\pi = 3.14159$

```
# Ejecutamos primero para ver que obtenemos
echo "scale = 1000; s(3.14159)" | bc -l

# Guardamos el resultado en una variable
resultado=$((echo "scale = 1000; s(3.14159)" | bc -l))
echo $resultado
```

- 2. Calcular el número pi, con 1000 decimales, basandote en que el arctangente de 1 es $\pi/4$, es decir, **arcotangente de 1 = $\pi/4$** . Recuerda que en bc la función arctangente(X) es, a(x).

```
pi=$(echo "scale=1000; 4*a(1)" | bc -l)
echo $pi
```

- 3. Calcular 3.5 elevado a 4 con escala de 10. Podemos ver como se pueden realizar una cadena de expresiones matemáticas, en una línea, separadas por ";"

```
# Calculamos el resultado de una potencia con: ^= (3.5 elevado a 4 y se lo
asignamos a "var")
echo "scale= 10; var=3.5;var^=4;var" | bc

# Encadenamos mediante ";" y asignamos el resultado final a una variable "x"
x=$( echo "scale= 10; var=3.5;var^=4;var" | bc)
echo $x
```

- 4. Calcular 255 en hexadecimal

```
x=$(echo "obase=16;255" | bc)
echo $x
```

OJO con el comportamiento de los post/pre incrementos/decrementos de las variables

```
echo 'var=10;++var' | bc    # Devuelve 11
echo 'var=10;var++' | bc   # Devuelve 10
```

6.3 Comando **test**

Para evaluar (testear) expresiones en *Bash*, se usa el comando** **test**. **El comando *test* compara dos expresiones mediante un operador que nos permite obtener el resultado de la condición.** Además, se pueden anidar condiciones mediante los operadores lógicos.

test devuelve valor 0 (true/verdadero) y valor 1 (false/falso)

```
# Comprueba si la variable $num es igual a 1
num=1
test $num -eq 1
```

```
echo "El resultado de del comando anterior es: $?"

# Aquí solo cambia la forma en la que mostramos el resultado (comillas '')
# Puedes probar...
num=1
test $num -eq 1
echo "El resultado de 'test \$num -eq 1' es: $?"

# Hacemos ahora esta comprobación
num=2
test $num -eq 1
echo "El resultado de del comando anterior es: $?"

# Mostramos el resultado de otra forma
num=2
test $num -eq 1
echo "El resultado de 'test \$num -eq 1' es: $?"

man test
```

IMPORTANTE: cuando se usan variables con *test*, deben inicializarse primero. Si se usa *test* con una variable sin inicializar, se puede producir un error.

Todas las operaciones que puede hacer *test*, se pueden usar en las condiciones de las estructuras de control (como condiciones de los *if*, o en los bucles) utilizando la notación `[]` o `[[]]`, que estudiaremos más adelante.

En la página del manual de *test* se especifican todas los posibles operadores que acepta *test*.

Se muestran a continuación los operadores más comunes:

Operador	Operación (Compara ...)
! EXPRESION	la EXPRESION es falsa (NOT).
-n STRING	Devuelve true si la longitud de STRING es mayor que 0.
-z STRING	Devuelve true si la longitud de STRING es 0 (está vacío).
STRING1 = STRING2	el contenido de STRING1 es igual al de STRING2.
STRING1 != STRING2	el contenido de STRING1 es distinto al de STRING2.
INTEGER1 -eq INTEGER2	INTEGER1 es igual a INTEGER2 (valor numérico).
INTEGER1 -ne INTEGER2	INTEGER1 es distinto a INTEGER2 (valor numérico).
INTEGER1 -gt INTEGER2	INTEGER1 es mayor que INTEGER2 (valor numérico).
INTEGER1 -ge INTEGER2	INTEGER1 es mayor o igual que INTEGER2 (valor numérico).
INTEGER1 -lt INTEGER2	INTEGER1 es menor que INTEGER2 (valor numérico).
INTEGER1 -le INTEGER2	INTEGER1 es menor o igual que INTEGER2 (valor numérico).

Operadores con ficheros	Operación (Compara ...)
-e /path/to/file	existe '/path/to/file'.
-f /path/to/file	existe '/path/to/file' y es un fichero regular.
-d /path/to/file	existe '/path/to/file' y es un directorio.
-h /path/to/file	existe '/path/to/file' y es un enlace simbólico.
-r /path/to/file	existe '/path/to/file' y puede ser leído.
-w /path/to/file	existe '/path/to/file' y puede ser escrito.
-x /path/to/file	existe '/path/to/file' y puede ser ejecutado.
-s /path/to/file	existe '/path/to/file' y no está vacío (tamaño > 0 bytes).
-O /path/to/file	existe '/path/to/file' y soy su propietario.

Ejemplos:

```
# Recuerda que $? es una variable especial que almacena el valor del comando
ejcutados justo antes
test 6 -gt 2
echo$?      # Devuelve 0 Cierto

test 9 -lt 8
echo $?     # Devuelve 1 Falso

test ! 9 -lt 8
echo $?     # Devuelve 0 Cierto

test -f /etc/passwd
echo $?

test -f /etc
echo $?

# Necesario el espacio en blanco entre el corchete de abrir [ y -f
[ -f /etc ]
echo $?

# Necesario el espacio en blanco entre el signo ! y el corchete de abrir
! [ -f /etc ]
echo $?
```

-eq vs =

Conviene destacar que `-eq` se comporta de forma diferente a `=`, como muestra el siguiente ejemplo:

```
test 007 -eq 7
echo "El resultado de 'test 007 -eq 7' es: $?"

test 007 = 7
echo "El resultado de 'test 007 -eq 7' es: $?"
```

Como puede apreciarse, `"="` realiza una comparación de strings, mientras que `"-eq"` realiza una comparación numérica

6.4 Corchetes simples y dobles: `[]` y `[[]]`. Evaluación de Expresiones relacionales

El uso de los corchetes simples `[]` es un operador equivalente a `test` y por tanto puede usarse de la misma manera, es decir, permite evaluar expresiones relacionales (comparaciones). Es por tanto muy utilizado en la evaluación de las expresiones condicionales de las sentencias `if - then -else`.

Si se va a usar `bash` como interprete, se recomienda utilizar siempre el comando compuesto condicional de doble corchete `[[...]]`, en lugar de la versión de corchete único compatible con Posix `[...]`. Eso es porque dentro de un `[[...]]` compuesto, la expansión de nombres de variables no provocan errores si no tienen valor o si contienen espacios en blanco.

NOTA: Tanto los corchetes como los dobles corchetes no forman parte de la sentencia "if" sino que son un operador equivalente a "test" para evaluar expresiones.

Veremos ejemplos del uso de los corchetes simples y dobles en las condiciones de las diferentes estructuras de control (if y bucles).

6.5 Comando read

Cuando durante la ejecución de un script en `bash` es necesario preguntar algún dato al usuario utilizaremos el comando `read`.

Con `read` podemos leer datos desde la entrada estándar. El uso de este comando es bastante sencillo. Veamos un ejemplo:

```
#!/bin/bash

echo "Introduce tu nombre:"
read nombre          # El valor se almacena en la variable nombre
echo "Tu nombre es: $nombre"
```

Al utilizar el comando `read` el script quedará detenido esperando que los datos sean introducidos desde la entrada estándar.

El comando **read** se puede combinar con diferentes opciones. La opción **-p** nos permite añadir un mensaje para invitar al usuario a introducir los datos y no salta de línea hasta recibir el valor de entrada.

Sintaxis: `read -p "mensaje" nombre_var`

Veamos el siguiente ejemplo:

```
#!/bin/bash

read -p "Escribe tu color favorito: " color
echo
if [ "$color" != "" ]
then
    echo "El $color es tu color favorito"
else
    echo "Sin preferencia de color...."
fi
echo
```

El comando **read** se puede combinar con más parámetros para diferentes usos. Por ejemplo, la opción **-t** "**seg**" espera un máximo de segundos y si no se ha introducido ningún valor el script sigue adelante.

```
#!/bin/bash

read -t 5 -p "Escribe tu color favorito: " color
echo
if [ "$color" != "" ]
then
    echo "El $color es tu color favorito"
else
    echo "Sin preferencia de color...."
fi
echo
```

Otra opciones interesantes del comando `read`, es por ejemplo **-s**:

- (-s): no se hace echo de lo que se esta escribiendo en el terminal

Podemos usar a la vez las opciones **read -sp** y hacer ejemplos como:

```
#!/bin/bash

# File: login.sh

read -p 'Username: ' LOGIN_USERNAME
read -sp 'Password: ' LOGIN_PASSWORD
echo
echo "Gracias $LOGIN_USERNAME, ahora puedes acceder al sistema"
echo "Aunque no hayas visto la contraseña, está aquí: $LOGIN_PASSWORD"
```

El comando `read` **permite leer varias variables a la vez**:

```
read var1 var2 ... varn
```

Veamos el siguiente ejemplo:

```
#!/bin/bash

# File: planets.sh

echo 'Dime tres planetas: '
read planet1 planet2 planet3
echo "Primero visitaremos ${planet1}, para enterarnos de qué va esto."
echo "Si necesitamos seguridad, podemos pasar por ${planet2}."
echo "Nos reuniremos con el resto del equipo en ${planet3}."
```

Resumiendo: La siguiente tabla muestra algunas de la **opciones más comunes** de `read`:

Opción	Descripción
-a ARR_NAME	Las palabras son asignadas en orden a los índices del <i>array</i> ARR_NAME (ojo, sobrescribe ARR_NAME).
-d DELIM	Se usa el carácter DELIM para finalizar la operación de <i>read</i> (por defecto <i>read</i> termina cuando se pulsa <i>ENTER</i> , es decir, con <i>/n</i>).
-n NUMCHARS	El comando <i>read</i> finaliza después de haber leído NUMCHARS caracteres.
-p PROMPT	Muestra el mensaje PROMPT delante de la entrada de texto.
-s	Modo silencioso: no se muestra el texto que se escribe (útil con contraseñas).
-t TIMEOUT	Hace que <i>read</i> termine y devuelva un error, si no se completa la entrada antes de TIMEOUT segundos.

Opción	Descripción
-u FD	Lee desde el descriptor de fichero FD.

En la página de manual de *read* se pueden ver todas las posibilidades que ofrece.

```
man read
```

También se puede usar en bucles (que estudiaremos más adelante), para hacer programas en los que el *prompt* invita a escribir:

```
#!/bin/bash

# File: teclea.sh

STR='vamos'
while [ "$STR" != 'q' ]; do
    echo 'Dime algo, o teclea q para salir:'
    read STR
    echo ">>> $STR"
done
```

7. ESTRUCTURAS DE CONTROL

7.1 Estructuras de Control Condicionales (if, case)

IF

Se usa una estructura **if** para evaluar el resultado de una expresión booleana.

También se puede utilizar para evaluar la salida de un comando.

Sintaxis:

```
if condicion_A; then
    comando1
    comando2

elif condicion_B; then
    comando3
    comando4
... (puede haber tantos else-if como sean necesarios)

else
    comandoN
fi

# o bien:

if condicion_A
then
    comando1
    comando2

elif condicion_B
then
    comando3
    comando4
... (puede haber tantos else-if como sean necesarios)

else
    comandoN
fi
```

```
# grep -s (silent). Suprime mensajes de error sobre ficheros que no existen o no
se pueden leer
if grep -s "pull request" ./README.md; then
    echo ">>> grep OK"
else
    echo "<<< grep KO"
```

```
fi
```

Expresiones y Operadores relacionales de la sentencias if

Utilizaremos los corchetes simples o dobles: `[]` y `[[]]`.

Recuerda que su función es equivalente al comando `test` y que sirven para evaluar expresiones relacionales.

Vamos a practicar con ejemplos, y veremos algunas diferencias en el uso de cada uno.

```
num=5

if [ $num -gt 20 ]
then
    echo "Wow, el número $num es mayor que 20!! :D"
elif [ $num -gt 10 ]
then
    echo "Bien, el número $num es mayor que 10 :)"
else
    echo "Vaya, el número $num es menor o igual que 10 :("
fi

# Con dobles corchetes [[ .. ]]

num=25
if [[ $num -gt 25 ]]
then
    echo "Wow, el número $num es mayor que 25!! :D"
elif [[ $num -gt 15 ]]
then
    echo "Bien, el número $num es mayor que 15 :)"
else
    echo "Vaya, el número $num es menor o igual que 15 :("
fi
```

El ejemplo anterior también destaca la importancia del espaciado. Por una parte, hay un espacio entre la palabra `if` o `elif` y el correspondiente corchete de apertura `[`. También hay espacios entre los corchetes (`[]`) y su contenido (por ejemplo `$num -gt 10`).

Además, para una mayor claridad, se han indentado los bloques de código del `if` y del `else`.

Recuerda: En lugar de poner `if` y `then` en líneas distintas, se pueden poner en la misma línea usando un `;`

```
num=5

if [ $num -gt 20 ]; then
```

```
    echo "Wow, el número $num es mayor que 20!! :D"
elif [ $num -gt 10 ]; then
    echo "Bien, el número $num es mayor que 10 :)"
else
    echo "Vaya, el número $num es menor o igual que 10 :("
fi
```

Evaluación de Expresiones booleanas: Operadores lógicos

Son aplicables a todas las estructuras de control. Los siguientes ejemplos muestran su uso con *if*, pero son extrapolables a cualquier estructura de control.

Operadores:

- AND: &&
- OR: ||
- NOT: !

```
# AND: &&
num=12

if [ $num -gt 10 ] && [ $num -le 20 ]
then
    echo "El número $num es mayor que 10, y menor o igual que 20 :)"
else
    echo "Vaya, el número $num es menor o igual que 10, o mayor que 20 :("
fi

# OR: ||
nombre='Perry'
apellido='Meison'

if [ "$nombre" = 'Perry' ] || [ "$apellido" = 'Foster' ]
then
    echo 'O bien el nombre es Perry, o bien el apellido es Foster'
else
    echo 'Ni el nombre es Perry, ni el apellido es Foster'
fi

# NOT: !
file='/etc/tortilla'

if [ ! -f "$file" ]
then
    echo "Vale, el fichero '$file' no existe..."
else
    echo "¡Suerte! Existe el fichero '$file'."
```

```
fi
```

Operadores alternativos de AND y OR:

- AND: `-a`
- OR: `-o`

Existe una forma alternativa de referirse a los operadores `&&` (AND) y `||` (OR).

```
```bash
AND: -a
num=12

if [$num -gt 10 -a $num -le 20]
then
 echo "El número $num es mayor que 10, y menor o igual que 20 :)"
else
 echo "Vaya, el número $num es menor o igual que 10, o mayor que 20 :("
fi

OR: -o
nombre='Perry'
apellido='Meison'

if ["$nombre" = 'Perry' -o "$apellido" = 'Foster']
then
 echo 'O bien el nombre es Perry, o bien el apellido es Foster'
else
 echo 'Ni el nombre es Perry, ni el apellido es Foster'
fi
```

### Importante:

```
[$num -gt 10] && [$num -le 20] # CORRECTO !!!! (Standar POSIX)
[[$num -gt 10 && $num -le 20]] # CORRECTO TAMBIÉN !!!! (bash/ksh)

[$num -gt 10 && $num -le 20] # INCORRECCTO !!!!
[$num -gt 10 -a $num -le 20] # CORRECCTO !!!! Extensión de POSIX no portable.
Evitar su uso!!!!
```

## CASE

La estructura **case** se usa para comparar una variable con una serie de patrones. Su comportamiento también puede realizarse con *if*, pero a veces un *case* es más elegante. Veamos un ejemplo típico: Sintaxis:

```
case cadena_texto in
 patron1) lista-compuesta1;;
 patron2) lista-compuesta2;;
 ...
 patronN) lista-compuestaN;;
 *) lista-defecto [;;] #coincide con todo
esac
```

```
cd
cat <<'CASE' >case.sh
#!/bin/bash

File: case.sh

case "$1" in
 'start')
 echo 'start...'
 ;;
 'stop')
 echo 'stop...'
 ;;
 'restart')
 echo 'restart...'
 ;;
 *)
 echo 'huh?'
 ;;
esac

CASE

chmod a+x case.sh
./case.sh
```

El contenido del fichero **case.h** es:

```
#!/bin/bash

File: case.sh

case "$1" in
```

```
'start')
 echo 'start...'
;;
'stop')
 echo 'stop...'
;;
'restart')
 echo 'restart...'
;;
*)
 echo 'huh?'
;;
esac
```

## 7.2 Bucles (for, while, until)

### FOR

Un bucle nos permite repetir un bloque de código tantas veces como queramos. En el caso de "for" se repetirá el código que está entre **do** y **done** tantas veces como elementos haya en la lista de valores que va después de **in**.

#### Sintaxis:

```
for VAR in lista_valores
do
 comando
done

for VAR in lista_valores; do
 comando
done
```

El nombre de la variable **VAR** debe aparecer obligatoriamente junto con la palabra reservada **for** en la misma línea. Y **lista\_valores** debe estar obligatoriamente en la misma línea que la palabra reservada **in**.

El bucle **for** es un poco diferente a otros lenguajes. En *Bash* **se usa para iterar sobre un conjunto de elementos de una lista**, tomándose cada valor de dicha lista como una cadena de caracteres que puede ser objeto de expansión.

Ejemplo:

```
Itera sobre el conjunto de números dado
for i in 1 2 3 4 5
do
 echo "Item: $i"
```

```
done
```

## Comando seq

Suele ser habitual el uso del **comando** externo **seq** para generar una lista de valores. Si bien este comando no está recogido en el estándar POSIX, es habitual su presencia en la mayoría de los sistemas UNIX/Linux. El comando **seq** presenta la sintaxis:

```
seq valor_inicial valor_final
```

siendo ambos valores números enteros. La salida del comando es la secuencia de números enteros entre ambos valores extremos indicados.

```
seq 1 10
```

### Más ejemplos de uso de for:

```
Itera sobre los nombres de fichero recogidos con 'ls'
for i in $(ls ~); do
 echo "Item: $i"
done

Itera sobre los nombres de fichero recogidos en un directorio
for i in ~/* ; do
 echo "Item: $i"
done

Itera sobre el conjunto de números determinado
for i in 1 2 3 4 5
do
 echo "Item: $i"
done

Itera sobre los números 1 al 10
for i in $(seq 1 10);
do
 echo "Num: $i"
done

Recordar que la sustitución de comandos, además de la notación anterior
$(), también puede emplear las comillas invertidas (` `) (back-ticks)

for i in `seq 1 10`;
do
 echo "Num: $i"
done

Itera sobre un RANGO
```



```
for i in {1..5}
do
 echo "Num: $i"
done

Itera sobre un RANGO, con un PASO
for i in {1..10..2}
do
 echo "Num: $i"
done

También se puede iterar sobre elementos de distinto tipo:
for i in tortilla 7 * 8 cebolla
do
 echo "Item: $i"
done
```

Como se acaba de ver en el bloque de código anterior, **un bucle for es muy apropiado para recorrer el contenido de un directorio.**

**IMPORTANTE:** Usando el "\*" se expande el contenido del directorio en el que estamos:

```
for i in *
do
 echo "Item: $i"
done

for i in /etc/*
do
 echo "Item: $i"
done
```

**Podemos probar más cosas:**

```
for i in tortilla 7 "*" 8 cebolla
do
 echo "Item: $i"
done

for i in tortilla 7 * 8 cebolla
do
 echo "Item: $i"
done

for i in tortilla 7 8 "cows are going mad"
do
 echo "Item: $i"
done
```

**También se puede usar el formato del lenguaje C. Importante los dobles paréntesis**

```
for ((k=0; k<=7; k++)); do
 echo "k = ${k}"
done
```

## WHILE

**El bucle `while` ejecuta un bloque de código mientras se cumpla una condición (expresión de control).** Termina su ejecución cuando la condición deja de ser cierta (o cuando se ejecuta *break*).

Sintaxis:

```
while condicion
do
 cmd1
 cmd2
done

while condicion; do
 cmd1
 cmd2
done
```

**Ejemplo:**

```
COUNTER=0
while [$COUNTER -lt 10]; do
 echo "El contador es: $COUNTER"
 ((COUNTER++))
done
```

**A veces es necesario implementar un bucle infinito para la ejecución de nuestro script**, es decir, que se esté ejecutando siempre hasta que se cumpla una determinada condición.

Se puede hacer un *while* infinito de varias formas. **A continuación se muestra una típica utilizando los ":" como condición del while.**

```
cat <<'INF' >infinite_while.sh
#!/bin/bash

File: infinite_while.sh
El comando read espera leer algo desde la entrada estándar.
```

```
Los ":" como condición del while indican que siempre se cumple la condición.
Equivalente a "true"

while :
do
 echo 'Dime algo, o pulsa CTRL + C para salir:'
 read STR
 echo ">>> $STR"
done

INF

chmod a+x infinite_while.sh
./infinite_while.sh
```

El contenido de *infinite\_while.sh* es:

```
#!/bin/bash

File: infinite_while.sh
El comando read espera leer algo desde la entrada estándar.
Los ":" como condición del while indican que siempre se cumple la condición.
Equivalente a "true"
while :
do
 echo 'Dime algo, o pulsa CTRL + C para salir:'
 read STR
 echo ">>> $STR"
done
```

## Uso del comando **read** para leer un archivo línea a línea en bash

Es común usar un bucle *while* para leer (y procesar) cada línea de un fichero y utilizar el comando *read* para leer dicho fichero.

Esta es la forma de hacerlo:

```
while read linea; do
 echo -e "Línea: $linea"
done < fichero
```

El archivo de entrada ("fichero") es el nombre del archivo que deseamos que esté abierto para su lectura mediante el comando "read".

Dicho comando lee el archivo línea por línea, asignando cada línea a la variable "línea".

Una vez que se procesan todas las líneas, el bucle "while" terminará. El separador de campo interno (IFS) se establece en la cadena nula para conservar los espacios en blanco iniciales y finales, que es el comportamiento predeterminado del comando de lectura (read).

### Veamos un ejemplo:

**Crearemos un fichero de registros que usaremos como fichero de entrada para lectura** línea a línea mediante el comando read. Campos del fichero:

- En el primer campo de cada registro (línea) se indica el nombre de una banda de música
- En el segundo fecha de lanzamiento del Disco
- En el tercero el nombre del álbum
- Cuarto: discográfica

El fichero a crear se llamará **records.csv**

```
Creamos un fichero de registros (tipo csv, cuyos campos estan separados por ";"). Este fichero sera el que usaremos de entrada para leer linea a linea mediante el comando read.
```

```
El formato de cada línea de este fichero es:
```

```
Nombre_de_banda_de_musica;año;nombre_disco1;discografica
```

```
cat <<'RECORDS' >records.csv
```

```
His Hero is Gone;1996;Fifteen Counts of Arson;Prank Records
```

```
His Hero is Gone;1997;Monuments to Thieves;Prank Records
```

```
His Hero is Gone;1998;The Plot Sickens;The Great American Steak Religion
```

```
Tragedy;2000;Tragedy;Tragedy Records
```

```
Tragedy;2001;Tragedy;Skuld Releases
```

```
Tragedy;2002;Can We Call This Life?;Tragedy Records
```

```
Tragedy;2002;Vengeance;Tragedy Records
```

```
Tragedy;2003;Vengeance;Skuld Releases
```

```
Tragedy;2003;Split 7" with Totalitär;Armageddon Label
```

```
Tragedy;2004;UK 2004 Tour EP;Tragedy Records
```

```
Tragedy;2006;Nerve Damage;Tragedy Records
```

```
Tragedy;2012;Darker Days Ahead;Tragedy Records
```

```
Tragedy;2018;Fury;Tragedy Records
```

```
RECORDS
```

Ya tenemos el fichero que vamos a leer, **ahora crearemos un script (records\_count.sh)** que leerá y procesará cada una de las líneas del fichero de entrada **records.csv** y debe contar cuantos registros (líneas) hay de cada una de las bandas. Para ello:

- Leerá cada línea del fichero
- Cogerá el primer campo de cada línea y comprobará a que banda pertenece: "His Hero is Gone" o "Tragedy".
- Cuando se cumpla un determinado patron, el script incrementará la variable correspondiente que lleva la cuenta de las líneas que pertenecen a cada banda. Por tanto, necesitaremos dos contadores.
- Al final el script debe imprimir el total del número de líneas que pertenecen a cada banda.

El contenido del script `records_count.sh` sería el siguiente:

```
#!/bin/bash

Inicializamos los contadores en los que vamos a contar las líneas que cumplen
cada patrón.

Contador de líneas que cumplen el patron "His Hero is Gone"
HHIG=0
#Contador de líneas que cumplen el patron "Tragedy"
TGDY=0

Leemos cada línea del fichero "records.csv" en un bucle while y la almacenamos
en la variable "f"
while read f; do
 band="$(echo "${f}" | cut -d';' -f1)"
 case ${band} in
 'His Hero is Gone')
 ((HHIG++))
 echo ">>> $HHIG" ;;
 'Tragedy')
 ((TGDY++))
 echo ">>> $TGDY" ;;
 *)
 echo '>>> Banda desconocida... ($f)' ;;
 esac
done < records.csv

echo ">>> Total HHIG = ${HHIG}"
echo ">>> Total TGDY = ${TGDY}"
```

Asignamos permiso de ejecución y ejecutamos el script

```
chmod a+x records_count.sh
./records_count.sh
```

## UNTIL

El bucle `until` es muy similar al bucle `while`, pero *until* se ejecuta mientras la condición es falsa, y termina su ejecución cuando la condición se vuelve verdadera.

```
COUNTER=20
until [$COUNTER -lt 10]; do
 echo "La variable COUNTER = $COUNTER"
 ((COUNTER-=1))
```

```
done
```

Observar que el código anterior devuelve el mismo resultado que el siguiente usando el bucle `while`.

```
COUNTER=20
while [$COUNTER -gt 9]; do
 echo "La variable COUNTER = $COUNTER"
 ((COUNTER-=1))
 echo $COUNTER
done
```

### 7.3 Ruptura de Sentencias de Control (break, continue)

Se utilizan para romper el funcionamiento normal de las estructuras repetitivas `for`, `while` y `until`.

#### BREAK

Como en otros lenguajes, en *Bash* la sentencia `break` **interrumpe la ejecución de un bucle**, ya sea *for*, *while* o *until*. **Break detiene todas las iteraciones restantes** de la estructura de control.

```
for i in {1..10..2}
do
 if [${i} -gt 5]; then
 break
 fi
 echo "Num: ${i}"
done

echo 'Hecho!'
```

#### CONTINUE

*Bash* también incorpora una sentencia *continue*, que dentro de un bucle, **deja de ejecutar la iteración actual** y pasa a la siguiente iteración.

```
for i in {1..10..2}
do
 if [${i} -eq 5]; then
 continue
 fi
 echo "Num: ${i}"
done

echo 'Hecho!'
```

## Algunos consejos sobre if, while, until, for, corchetes simples y dobles

Recordar que los corchetes de la expresión de test, tienen que tener **espacio** a ambos lados, si no --> fallo

```
#!/bin/bash
i=0
while [$i -lt 10]; do # --> fallo [:missing '] ' (falta espacio al
 echo " $i" final)
 ((++i))
done

while [$i -lt 10];do # --> fallo [: comand no found ((falta espacio al
 echo " $i" principio))
 ((++i))
done

i=0
while [$salir -lt 1]; do # --> fallo [: -lt: se esperaba operador
 echo " $i" unario
 ((++i)) # la variable salir está sin inicializar
done
```

Si se utiliza if con nada entre el then y el fi también se produce fallo, si se está haciendo un esqueleto insertar de forma provisional algo entre el then y el fi, ej un echo "".

```
if [$i -lt 10]; then # --> fallo
fi
```

Recordar la forma de incluir varias condiciones en un if (|| -o o && -a):

```
if [$nombre = "juan"] || [$nombre = "pepe"]; then
```

--> Si nombre es juan o pepe se ejecuta el then

### Los corchetes dobles:

- 1. Admiten la comparación de dos variables; `bash if [[ $a > $b ]]` En anteriores versiones de bash los corchetes simples sólo admiten una variable (operador unario) `if [ $a -eq 1 ]`
- 2. Los dobles corchetes son mas potentes que los sencillos: Se pueden usar operadores `||` `&&` y con cadenas `<>` Se pueden usar comodines `*`

- 3. Admite operaciones con doble paréntesis dentro del doble corchete: `bash if [[ $((($1 % 2)) = 0 )]`



## 8. FUNCIONES

Como en otros lenguajes de programación, en *Bash* se pueden declarar funciones para agrupar bloques de código. Una cosa es declarar una función y otra ejecutar una función. Par usar una función o invocarla es preciso haberla declarado/definido antes.

En Bash, **se puede definir una función en una sola línea**, en este caso es muy importante tener en cuenta los espacios después de la primera llave y antes del primer comando, y antes de la última llave. Es decir, hay que dejar espacio entre las llaves y los comandos para que no de un error. Además entre comandos debes, utilizar ";" incluso después del último comando. Ejemplo:

```
mi_funcion() { echo "hola mundo"; echo "adios mundo"; }
```

**Otra opción para declarar una función es utilizando la palabra clave `function`.** De esta manera la declaración de tu función tendría el siguiente aspecto:

```
function mi_primera_funcion(){
 echo Hola Mundo
}
```

**Para hacer uso de una función, solo tenemos que invocarla por su nombre.**

```
mi_primera_funcion
```

Veamos un ejemplo sencillo:

```
#!/bin/bash

File: saludamos.sh
saluda() {
 echo "Hola, terrícola"
}

echo "Invocamos función saluda()...."
saluda
```

### 8.1 Valor de retorno de una función. Comando return

Como el resto de comandos en *Bash*, una función devuelve un número entero indicando el resultado de salida (ejecución del último comando de la función). Típicamente un valor de retorno de 0 implica una ejecución

correcta. Un valor mayor que 0, implica que hubo errores durante la ejecución.

**Podemos devolver un valor si se usa la sentencia `return`.**

```
#!/bin/bash

File: saludamos2.sh

Funcion saluda()
saluda() {
 echo "Hola, terrícola"
 return 42
}

echo "Invocamos funcion saluda()..."
saluda
echo ">>> saluda() ha devuelto: $?"

Funcion saluda_mas()
saluda_mas() {
 echo "Hola terrícola, es un placer recibirte a bordo de nuestra nave"
}

echo "Invocamos funcion saluda_mas()..."
saluda_mas
echo ">>> saluda_mas() ha devuelto: $?"
```

## 8.2 Valores de salida de una Función

En realidad, **aunque una función sólo devuelva un entero, puede producir salidas de varias formas:**

- Puede modificar **variables externas**.
- Puede invocar el comando *exit* para terminar el *script*.
- Puede devolver el resultado, utilizando el comando **echo**. En este caso el valor se guarda en una variable usando el método command-substitution o sustitución de comandos para invocar la función:  
`miVar=$( miFuncion )`.

Veamos algunos ejemplos de **como devolver el resultado de una función:**

### Ejemplo 1:

```
#!/bin/bash

File: suma_valor.sh

suma_diez() {

 read -p "Introduce un valor: " value
```

```
El resultado se devuelve mediante echo dentro de la función.
 echo $(($value + 10))

}

Invoca la función mediante el metodo de sustitución de comando.
result=$(suma_diez)

echo "Resultado de la funcion: $result"
```

### Ejemplo 2:

```
#!/bin/bash

File: datos_so.sh

Define osname()
function osname() {
 local os="$(uname -s)"
 local ver="$(uname -r)"
 echo "${os} ${ver}"
 return 17
}

Invoca a osname()
osname
echo ">>> osname() ha devuelto: $?"

Podemos invocar la función y almacenar
n=$(osname)

echo ">>> n = ${n}"
```

### Ejemplo 3:

```
#!/bin/bash

File: saludando.sh

saluda() {
 echo "Hola, terrícola"
}

saluda
echo ">>> Invocar saluda() ha devuelto: $?"
```

```
s=$(saluda)
echo ">>> El resultado de invocar mediante sustitución de comando es = ${s}"
```

### 8.3 Ámbito de las variables de un script y su uso con funciones

**VARIABLES:** Como se puede ver en los ejemplos siguientes, cuando se declara una variable en un script, por defecto su **ámbito es global**, y puede ser leída o modificada dentro de una función. Se pueden declarar variables de **ámbito local**, que sólo existen dentro de una función, con la expresión **local**.

```
#!/bin/bash

File: program_demo.sh

Ámbito de variables

Inicializamos dos variables (globales)
num=10
val=5
echo -e ">>> num = ${num}\tval = ${val}\t\t[1]"

Definimos funcion varDemo
function varDemo() {
 local num=42
 echo -e "varDemo(): num = ${num}\tval = ${val}\t\t[2]"

 # Modificamos el valor de "num" (local)
 num=51
 # Modificamos el valor de "val" (global)
 val=7
}

echo -e ">>> num = ${num}\tval = ${val}\t\t[3]"

Invocamos Funcion
varDemo
echo ">>> varDemo() ha devuelto: $?"
echo -e ">>> num = ${num}\tval = ${val}\t\t[4]"

Invocamos de nuevo la función mediante el método de sustitución de comando
v=$(varDemo)
echo ">>> Resultado invocación método de comando v = ${v}"

##-----

function varDemo_other() {
 alpha=42
 beta=24
 echo -e "varDemo_other(): alpha = ${alpha}\tbeta = ${beta}\t\t[5]"
```

```

}

echo -e ">>> alpha = ${alpha}\tbeta = ${beta}\t[6]"

v=$(varDemo_other)
echo ">>> Resultado invocación método de comando v = ${v}"

```

## 8.4 Paso de Parámetros especiales/posicionales a las funciones

A la función se le pueden pasar los mismos parámetros que puede recibir un script: \$0, \$1-\$9, \$#, \$?, \$@, \$\$

**IMPORTANTE:** Consideraciones a tener en cuenta que el parámetro \$0 siempre hace referencia al nombre del script (no de la función) y el parámetro \$\$ hace referencia al ID del proceso del script.

Veamos un ejemplo:

```

#!/bin/bash

File suma_parametros.sh
addnum() {

 if [$# != 2]; then

 echo "Número de parámetros incorrecto"

 else
 # Los sumo
 echo $(($1 + $2))
 fi
}

echo -n "Sumando 10 y 15: "
value=$(addnum 10 15)
echo $value

echo -n "Sumando otros números: "
value=$(addnum 10)
echo $value

echo -n "Sumando otros números: "
value=$(addnum $1 $2)
echo $value

```

## 9. ARRAYS

En *Bash*, como en otros lenguajes, los *arrays* son colecciones de elementos, cada uno de los cuales se almacena en una de las posiciones de dicho array. En *Bash* los elementos pueden ser de diferentes tipos. A continuación se muestran algunas operaciones típicas con *arrays*:

### 9.1 Declarando un "array"

En Bash existen diferentes formas de declarar un array,

- Asignando valores, por ejemplo: `colores[0]='rojo'`
- Utilizando la palabra clave **declare**, de la forma: `declare -a colores`
- O directamente con el siguiente formato `colores=('rojo' 'azul')`

Los elementos del array pueden ir sin comillas, salvo que alguno de los elementos lleve un espacio en blanco, en cuyo caso las comillas son necesarias o bien es necesario escapar el espacio.

```
#!/bin/bash

3 formas de crear un array:
colores[0]=rojo
miArr[7]=33

declare -a colores

colores=('rojo' 'azul claro')
planetas=('Júpiter' 'Urano' 'Venus')
```

### 9.2 Asignar un valor a una posición de un "array"

Los elementos de los arrays en Bash empiezan a contar en la posición 0, aunque al hacer una asignación podemos cambiar los índices indicándolo explícitamente.

```
#!/bin/bash

Asignar un valor a una posición del array:
miArr[7]=33
frutas=(melon piña sandia)
frutas=([2]=melon [1]=piña [0]=sandia)
```

### 9.3 Accediendo a un "array"

- Para acceder de forma individual a cada una de los elementos de un array, se debe utilizar la siguiente sintaxis, `${array[i]}`. Así por ejemplo para imprimir el segundo de los colores, utilizarías `echo ${colores[1]}`.

- Si lo que utilizamos es solo el nombre del array, es decir, `echo $array` obtendremos el **primer elemento del array**. En nuestro ejemplo `echo $colores`.
- Si en lugar de imprimir un solo elemento en concreto, **queremos imprimirlos todos en la misma línea**, entonces debemos ejecutar `echo ${array[@]}`. En nuestro caso `${colores[@]}`.
- **Los índices especiales: @ y \* funcionan igual cuando no se usan las comillas dobles**. Si usamos comillas, la @ devuelve los elementos del array separados por blancos mientras que el \* devuelve los elementos del array separados por el caracter IFS (Internal Field Separator).

```
#!/bin/bash

colores=('rojo' 'azul')
planetas=('Júpiter' 'Urano' 'Venus')
miArr[7]=33

Acceder a los elementos de un array:
echo "colores[1]: ${colores[1]}"
echo '$colores:' "$colores"
echo "colores[@]: ${colores[@]}"
echo "miArr[7] = ${miArr[7]}"

Para imprimir cada elemento del array en una línea es necesario utilizar un
bucle
for i in "${colores[@]}"
do
 echo $i
done

También se puede acceder a varios elementos del array. Este ejemplo extrae dos
elementos empezando en la posición 1
echo ${planetas[@]:1:2}

Ejemplos de uso de @ y *. Ambos funcionan igual cuando no se usan las comillas
dobles. Si usamos comillas, la @ devuelve los elementos del array separados por
blancos mientras que el * devuelve los elementos del array separados por el
caracter IFS.
IFS=,
echo ${colores[@]}
echo ${colores[*]}
echo "${colores[@]}"
echo "${colores[*]}"

echo '-----'

Más ejemplos
echo "planetas[*] = ${planetas[*]}"
echo "planetas[2] = ${planetas[2]}"
planetas[3]='Saturno'
```

```
echo '-----'
```

## 9.4 Tamaño de un "array" y elementos no nulos de un "array"

- **Para saber cuantos elementos tienen un array** en Bash, debemos utilizar `${#array[@]}`. Así, en el ejemplo que estamos utilizando de los colores, si quieres conocer cuantos elementos hay en dicho array tienes que utilizar `echo ${#colores[@]}`.
- **Si queremos saber la longitud de uno de los elementos del array**, entonces debemos referirnos a la posición de dicho elemento: `${#array[0]}`. En nuestro ejemplo, si ejecutas `echo ${#colores[0]}` te devolverá 4, el número de letras de la palabra 'rojo'.
- **Si queremos saber que elementos son no nulos** debemos usar: `${!array[@]}`. Nos devuelve los índices de los elementos no nulos.

```
#!/bin/bash

planetas=('Júpiter' 'Urano' 'Venus')
planetas[3]='Saturno'

Ejemplos de contenido, tamaño e índices no nulos de un array.
echo "Contenido del array: planetas[@] = ${planetas[@]}"
echo "Tamaño del array: planetas[@] = ${#planetas[@]}"
echo "Índices no nulos: !planetas[@] = ${!planetas[@]}"

Veamos que pasa ahora
planetas[9]='Neptuno'
echo "Contenido del array: planetas[*] = ${planetas[*]}"
echo "Tamaño del array: planetas[*] = ${#planetas[*]}"
echo "Índices no nulos del array: !planetas[*] = ${!planetas[*]}"

Tamaño de un array (número de elementos):
echo "El array 'planetas' tiene: ${#planetas[*]} elementos."

Tamaño del contenido de una de sus posiciones:
echo "planetas[1] tiene: ${#planetas[1]} caracteres/dígitos."
```

## 9.5 Eliminar un array o una posición del array

Se utiliza el comando **unset** seguido de la posición del array que se desea eliminar o seguido del nombre del array.

```
#!/bin/bash
planetas=('Júpiter' 'Urano' 'Venus')
```



```
Eliminar una posición
unset planetas[2]
echo "planetas[2] = ${planetas[2]}"

Ha desaparecido "Venus"
echo "planetas[@]= ${planetas[@]}"

Eliminar un array
unset planetas
echo "planetas[*] = ${planetas[*]}"
```

## 9.6 Un *array* puede generarse a partir de la salida de un comando:

```
#!/bin/bash

lista=($(ls ~)) # Los paréntesis exteriores son para declarar el array
"lista"
echo "${lista[*]}"
echo "${#lista[*]}"
```

## 9.7 Uso de bucles para recorrer un "array"

Es común recorrer los elementos de un *array* con un bucle *for*:

```
#!/bin/bash

Creamos un array
numeros=(1 2 3 4)

Para iterar sobre todos los elementos del array hay que convertirlo primero en
una lista mediante el uso de "@", es decir usano ${array[@]}.

for num in ${numeros[@]}; do
 echo -e "num = ${num}"
done
```

## 9.8 Arrays y operaciones con variables

```
#!/bin/bash
Sintaxis para recuperar el valor de un array, siendo el indice una variable y
guardando el resultado en otra variable
array[1]=5
indice=1
var=${array[indice]}
```

```
echo "El valor de array[1] es $var"
```

```
#!/bin/bash
Operaciones con elementos de un array
indice=2
array[indice]=10
array[3]=20
array[4]=30
suma=$((array[3] + array[4] + array[indice]))
echo $suma
```

```
#!/bin/bash
Añadir elementos a un array, después del último elemento, incluso cuando hay
índices vacíos

array[0]=1; array[3]=2
echo "1. Contenido array: ${array[@]}"; echo "1. Índices no nulos: ${!array[@]}"

Añade un elemento (6) en la siguiente posición vacía a la última
array+=(6)
echo "2. Contenido array: ${array[@]}"; echo "2. Índices no nulos: ${!array[@]}"
```

```
#!/bin/bash
Si se asigna "" a una posición de un array, dicha posición no es nula, es decir,
el índice no desaparece

array[3]=""
echo "3. Contenido array: ${array[@]}"; echo "3. Índices no nulos: ${!array[@]}"
```

## 10. Tipos de variables y accesibilidad a dichas variables

Como se ha visto en el apartado de las funciones, las variables en bash, son de ámbito global y por defecto no tienen un tipo específico. Mediante el uso de expresiones como **local**, se puede restringir su ámbito. También hay formas de restringir su contenido, o si se prefiere, su tipo.

declare

Se **puede especificar el tipo de dato que puede almacenar una variable** con la expresión **declare**, como se muestra a continuación (lo hemos visto para declarar un array):

Opción	Descripción
--------	-------------

Opción	Descripción
-a	La variable es un <i>array</i> .
-f	La variable sólo contiene nombres de funciones.
-i	La variable será tratada como un número entero.
-p	Muestra los atributos y valores de cada variable.
-r	La variable es de sólo lectura (constante).
-x	Marca la variable para ser exportada.

Veamos algunos ejemplos con el uso de `declare`:

```
#!/bin/bash

declaro una variable de tipo entero
declare -i NUM=42

echo ">>> NUM = ${NUM}"

Veamos que pasa si a una variable de tipo entero le asigno un string
NUM='tortilla'
echo ">>> NUM = ${NUM}"

Muestra atributos y valores de la variable
declare -p NUM
```

## readonly

Permite **especificar que una variable no se puede modificar**, es marcarla como de sólo lectura.

Veamos un ejemplo:

```
#!/bin/bash

readonly NUM=42

echo ">>> NUM = ${NUM}"

Arrojará un error
NUM=56
echo ">>> NUM = ${NUM}"
```