# COMANDOS SHELL Y PROGRAMACIÓN EN LA SHELL DEL BASH

## SISTEMAS OPERATIVOS
## I.T. INFORMÁTICA DE GESTIÓN

# Índice de Contenidos

1.- REPASO CONCEPTOS BÁSICOS DE UNIX/LINUX....1
1.1.- El sistema de ficheros....1
1.1.1.-Moviéndonos por el sistema de ficheros....2
1.2.- Manipulación....3
1.3.- La ayuda del sistema....3
1.4.- Patrones (metacaracteres de sustitución)....4
1.4.1.-Igualando un carácter simple con ?....5
1.4.2.-Igualando cero o más caracteres con *....5
1.4.3.-Igualando cero o más caracteres con []....6
1.4.4.-Abreviando nombre de archivos con { }....6
1.5.- Búsqueda....7
1.6.- Tipos y contenidos de ficheros....7
1.7.- Permisos....9
1.7.1.-Resumen de comandos....11
2.- COMANDOS SHELL I....12
2.1.- El sistema Operativo Linux....12
2.1.1.-Estructura e Interfaces de Linux....13
2.2.- Los interpretes de órdenes....14
2.3.- Una sesión de trabajo....16
2.3.1.-El sistema de archivos....17
2.3.2.-El shell de entrada....18
2.4.- Los procesos....19
2.4.1.-Algunas herramientas para vigilar procesos....20
2.4.2.-Metacaracteres sintácticos....21
2.4.3.-Órdenes para el control de trabajos....26
2.5.- Metacaracteres de entrada/salida o de dirección....29
2.6.- Empaquetado y compresión de archivos con tar y gzip....34
2.7.- Ejercicios....38
3.- COMANDOS SHELL II....42
3.1.- Historia de órdenes....42
3.2.- Autocompletar con el tabulador....46
3.3.- Metacaracteres....46
3.3.1.-Metacaracteres sintácticos....47
3.3.2.-Metacaracteres de nombre de archivos....47
3.3.3.-Metacaracteres de citación....48
3.3.4.-Metacaracteres de entrada/salida o de dirección....50
3.3.5.-Metacaracteres de expansión/sustitución....50

3.4.- Los alias....51
3.4.1.-Definición y eliminación de alias....51
3.4.2.-Listar la definición de los alias existentes....52
3.4.3.-Renombrar o redefinir una orden existente....52
3.4.4.-Crear una nueva orden....52
3.5.- Variables....53
3.5.1.-Variables de entorno y variables locales....53
3.5.2.-Creación de variables....54
3.5.3.-Personalizar el prompt....56
3.5.4.-Variables de entorno internas....58
3.5.5.-Exportar variables....59
3.6.- Scripts....59
3.6.1.-Primeros pasos con scripts....60
3.6.2.-Variable en los scripts....62
3.6.3.-Paso de argumentos a los scripts....64
3.7.- Ejercicios....68
4.- PROGRAMACIÓN BASH I....71
4.1.- Variables con tipo....71
4.2.- Expresiones aritméticas....73
4.2.1.-Expansión del shell $((expresión))....74
4.2.2.-Similitud con las expresiones aritméticas C....75
4.2.3.-El comando interno let....75
4.3.- Las sentencias condicionales....76
4.3.1.-Las sentencias if, elif y else....76
4.3.2.-Los códigos de terminación....77
4.3.3.-La sentencia exit....78
4.3.4.-Operadores lógicos y códigos de terminación....78
4.3.5.-Test condicionales....79
4.3.6.-If aritmético....85
4.4.- El bucle for....85
4.5.- Bucles while y until....88
4.6.- Entrada y salida de texto....90
4.6.1.-El comando interno echo....90
4.6.2.-El comando interno printf....92
4.6.3.-El comando interno read....95
4.7.- Ejercicios....97
5.- PROGRAMACIÓN BASH II....99
5.1.- La sentencia case....99
5.2.- La sentencia select....100
5.3.- La sentencia shift....102

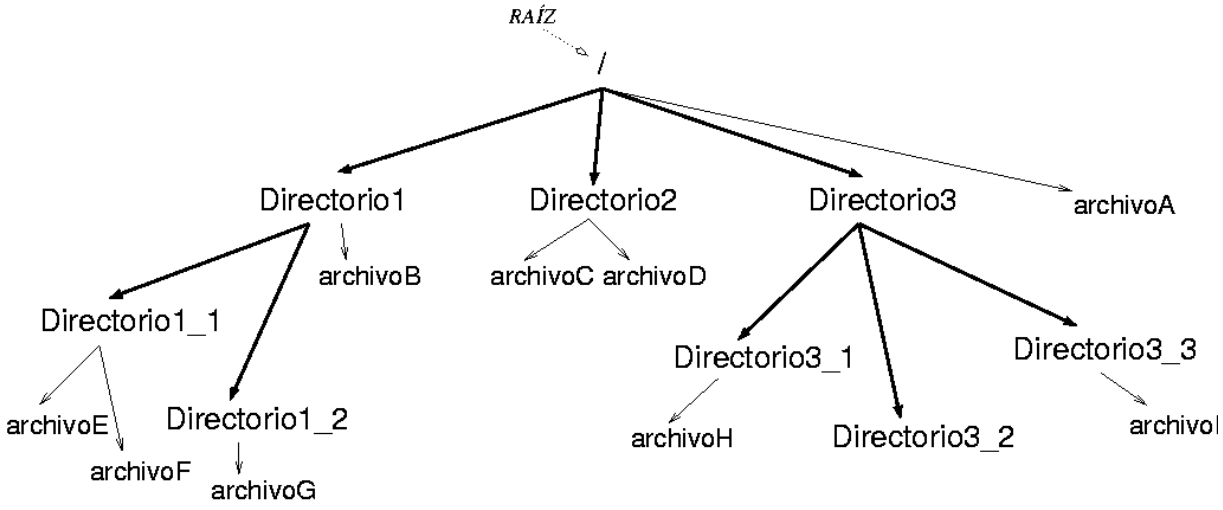
5.4.- Listas (Arrays).....103
5.5.- Funciones.....108
5.6.- Operadores de cadena.....110
5.6.1.-Operadores de sustitución.....110
5.6.2.-Operadores de búsqueda de patrones.....114
5.6.3.-El operador longitud.....117
5.7.- Sustitución de comandos.....117
5.8.- Los ficheros de configuración de Bash.....118
5.9.- Ejercicios.....120

# 1.- REPASO CONCEPTOS BÁSICOS DE UNIX/LINUX

## 1.1.- El sistema de ficheros

Todo sistema operativo necesita guardar multitud de archivos: desde los de la configuración del sistema, los de log, los de los usuarios, etc. En general, cada sistema operativo utiliza su propio sistema de ficheros, caracterizándolo en muchos aspectos como pueden ser el rendimiento, seguridad, fiabilidad, etc.

Lo primero que debemos tener claro es que todo el sistema de ficheros parte de una misma raíz, a la cual nos referiremos con el carácter '/ '. Es el origen de todo el sistema de ficheros y sólo existe una. Para organizar los ficheros adecuadamente, el sistema proporciona lo que llamaremos directorios (o carpetas), dentro de las cuales podemos poner archivos y más directorios. De esta forma conseguimos una organización jerárquica como la que vemos en la siguiente figura:




*RUTA archivoA: /archivoA*
*RUTA archivoB: /Directorio1/archivoB*
*RUTA archivoC: /Directorio2/archivoC*
*RUTA archivoD: /Directorio2/archivoD*
*RUTA archivoE: /Directorio1/Directorio1_1/archivoE*
...

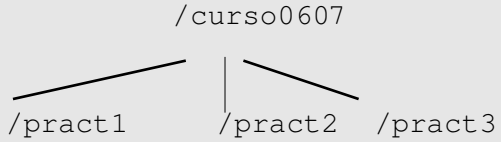
*Figura 1.1: Sistema de archivos*

El comando básico para crear un directorio es **mkdir.** Por su parte, si queremos crear un archivo se puede utilizar cualquier editor de texto, como se verá más adelante.

### 1.1.1.- Moviéndonos por el sistema de ficheros

Para movernos por la estructura de directorios debemos utilizar los comandos para listar contenidos y cambiar de carpeta. Si queremos ver lo que hay en el directorio donde estemos situados podemos listar los contenidos utilizando el comando ***ls***. Debemos tener en cuenta que por defecto el comando no nos muestra los archivos que empiezan por un punto. Para cambiar de directorio podemos utilizar el comando ***cd***. Si no le pasamos ningún parámetro, por defecto nos situará en nuestro directorio **home;** si le pasamos como argumentos "**..**", nos situará en el directorio superior. Generalmente, se le suele indicar dónde queremos ir, pasándolo de forma absoluta o relativa. De forma relativa significa que partiremos del directorio donde estemos en el momento de ejecutar el comando. Por ejemplo, si estamos en el directorio "/Directorio1/Directorio1_1/" y queremos ir al "/Directorio3/", deberíamos introducir el siguiente comando: "***cd ../../Directorio3"***. De forma absoluta siempre partimos de la raíz, de manera que el comando que utilizaríamos para el ejemplo anterior sería: ***"cd /Directorio3"***. Para saber en qué directorio estamos podemos utilizar el comando ***pwd.***

1. Visualice el camino completo del directorio actual.
2. Visualice los ficheros del directorio actual.
3. Cree la siguiente estructura de directorios dentro de su directorio home.



## 1.2.- Manipulación

Ahora que ya sabemos movernos correctamente por la jerarquía de directorios también necesitamos saber cómo copiar, eliminar y manipular correctamente otros aspectos de los ficheros. El comando ***rm*** es el que se encarga de eliminar los archivos que le indiquemos. Para eliminar un directorio, podemos utilizar el comando ***rmdir***, aunque sólo lo borrará cuando éste esté vacío. Para copiar archivos de un lugar a otro tenemos el comando ***cp***, con el cual siempre debemos indicarle el fichero o directorio origen y el lugar o nombre de destino, aunque sea en el directorio actual. Si en lugar de copiar los archivos queremos moverlos de sitio podemos utilizar el comando ***mv***. Este comando también se utiliza para renombrar archivos.

```
4.Crea en el directorio pract2 un archivo que se llame
  apuntes.txt y otro que se llama ejercicios.txt
5.Escribe al menos 5 líneas en cada uno
6.Copia el fichero apuntes.txt  en el directorio pract3
7. Mueva el fichero ejercicios.txt al directorio pract1
8.Borra el archivo apuntes.txt del directorio pract2
9.Borra el directorio pract2
10.Cámbiale el nombre al fichero apuntes.tx por notas.txt
```

## 1.3.- La ayuda del sistema

Todos los comandos que veremos tienen multitud de opciones y parámetros diferentes que nos permiten manipularlos de la forma que queramos. Desde el principio se tuvo muy en cuenta que es imprescindible contar con una buena documentación para todos ellos. Igualmente, toda esta información también es necesaria para los ficheros de configuración del sistema, las nuevas aplicaciones que utilizamos, etc. Es por esto que el mismo sistema incorpora un mecanismo de manuales con el que podemos consultar casi todos los aspectos de los programas, utilidades, comandos y configuraciones existentes. El comando más utilizado es el ***man***, que nos enseña el manual del programa que le indiquemos como parámetro. Por defecto esta documentación es mostrada utilizando el programa ***less***, con el cual podemos desplazarnos hacia adelante y hacia atrás con las teclas de AVPÁG y REPÁG, buscar una palabra con el carácter "/" seguido de la palabra y

“q” para salir.

Si hay más de un manual disponible para una misma palabra, podemos especificarlo pasándole el número correspondiente de la sección deseada antes de la palabra, por ejemplo **“**`man 3 printf`**”**. Como los otros comandos, ***man*** también tiene multitud de opciones diferentes documentadas en su propio manual “man man”. Una de estas opciones que nos puede ir muy bien en las ocasiones que no sepamos exactamente el programa que estamos buscando es “-k”' (el comando ***apropos*** hace casi exactamente lo mismo). Con **“**`man -k`**”** seguido de una palabra que haga referencia a la acción que queramos realizar se buscará por entre todos los manuales del sistema y se mostrarán los que en su descripción o nombre aparezca la palabra indicada. De esta forma, podemos encontrar todo los queramos sin tener que recurrir a ningún libro o referencia externa al sistema.

Si el manual no nos proporciona toda la información que necesitemos también podemos usar el comando ***info***, que es lo mismo que el manual pero aún más extendido. Si lo único que queremos es tener una breve referencia de lo que hace un determinado programa podemos utilizar el comando ***whatis***.

```
11.Pida ayuda sobre los comandos marcados en negrita
12.Busque comandos que permitan comparar (cmp).
```

## 1.4.- Patrones (metacaracteres de sustitución)

Un mecanismo muy útil que nos proporciona el sistema son los *patterns* (patrones). Hasta ahora hemos visto como aplicar ciertas operaciones sobre un determinado archivo. Cuando estamos manipulando un sistema, en muchos casos nos interesará aplicar alguna de las operaciones que hemos visto pero sobre un grupo grande de ficheros. Los patrones nos permitirán aplicar las operaciones que queramos especificando, en una sola instrucción, varios ficheros que cumplan con una serie de características especificadas. Debemos verlos como plantillas de nombres, que nos permiten hacer referencia de forma abreviada a una serie de archivos cuyos nombres siguen un patrón. La Tabla 1.1 muestra estos metacaracteres.

| *Metacaracter* | *Descripción de la función* |
|---|---|
| ? | Comodín a cualquier carácter simple |
| * | Iguala cualquier secuencia de cero o más caracteres |
| [] | Designa un carácter o rango de caracteres que, como una clase, son igualados por un simple carácter. Para indicar un rango, mostramos el primer y el último carácter separados por un guión (-). **Con el símbolo ! indicamos negación**. |
| {} | Abreviar conjuntos de palabras que comparten partes comunes |
| ~ | Se usa para abreviar el camino absoluto (path) del directorio home |

*Tabla 1.1: Patrones*

### 1.4.1.- Igualando un carácter simple con ?

Es muy frecuente crear archivos con algún patrón como parte de su nombre. Por ejemplo, para realizar unas prácticas, nombramos cada archivo como practica_1.txt, practica_2.txt, practica_3.txt, etc. Si quisiéramos obtener información de los archivos con **ls,** podríamos dar las órdenes:

```
$ ls  -l pr actica_1 .txt  practica_ 2.txt  p racti ca_3.t xt
-rw-r--r-- 1 mluque   mluque        3 Oct  1 19:44 practica_1.txt
-rw-r--r--  1 mluque   mluque        3 Oct  1 19:44 practica_2.txt
-rw-r--r-- 1 mluque   mluque        3 Oct  1 19:44 practica_3.txt
```

```
$ ls  -l pr actica_?. txt  #Forma más compacta; cada ? iguala un carácter.
-rw-r--r--   1 mluque   mluque       3 Oct  1 19:44 practica_1.txt
-rw-r--r--   1 mluque   mluque       3 Oct  1 19:44 practica_2.txt
-rw-r--r--   1 mluque   mluque       3 Oct  1 19:44 practica_3.txt
```

### 1.4.2.- Igualando cero o más caracteres con *

El carácter especial * iguala cero o más caracteres. Por ejemplo, los archivos listados

anteriormente, podemos verlos usando la cadena de igualación practica_*:

```
$ ls -l  practic a_*
-rw-r--r--   1 mluque   mluque       3 Oct  1 19:44 practica_1.txt
-rw-r--r--   1 mluque   mluque       3 Oct  1 19:44 practica_2.txt
-rw-r--r--   1 mluque   mluque       3 Oct  1 19:44 practica_3.txt
```

Recordad que en Linux no existe el concepto de nombre y extensión a la hora de nombrar archivos. El punto es un carácter más permitido en el nombre de un archivo.

### 1.4.3.- Igualando cero o más caracteres con []

Los corchetes definen una lista, o clase de caracteres, que se pueden igualar con un sólo carácter. A continuación, ilustramos algunas formas de caracterizar grupos de archivos:

| | |
|---|---|
| `[A-Z]*` | Iguala todos los archivos que comienzan con una letra mayúscula. |
| `*[aeiou]` | Iguala cualquier archivo que finalice con una vocal. |
| `tema.*[13579]` | Iguala los temas que finalizan con un número impar |
| `tema.0[1-3]` | Iguala tema.01, tema.02, tema.03. |
| `[A-Za-z][0-9]*` | Iguala los archivos que comienzan con una letra (mayúscula o minúscula), seguida de un dígito, y cero o más caracteres. |
| `[!A-Z]*` | Iguala los archivos que **no** comiencen por una letra mayúscula |

```
13.Sitúate en tu home
14.Listar todos los archivos que empiecen por s,
15.Listar todos los archivos que contengan una a
16.Listar todos los archivos que empiecen por "a"' o por ``b'' y
  que contengan cualquier otra cadena.
```

### 1.4.4.- Abreviando nombre de archivos con {}

El uso de las llaves ({}), solas o combinadas con los anteriores caracteres especiales (?,*,[]), nos

van a permitir formas expresiones de nombres de archivos más complejas. Las llaves contienen una lista de uno o más caracteres separados por comas. Cada ítem de la lista se utiliza en turno para expandir un nombre de archivo que iguala la expresión completa en la que están inmersas las llaves.

Por ejemplo, `a{f,e,d}b` se expande en `afb`, `aeb` y `adb`, en este orden exactamente. Las llaves se pueden utilizar más de una vez en una expresión. La expresión `s{a,e,i,o,u}{n,t}` se expande en `san`, `sen`, `sin`, `son`, `sun`, `sat`, `set`, `sit`, `sot` y `sut`.

## 1.5.- Búsqueda

Otro tipo de operación muy útil es la búsqueda de ficheros. Tenemos varios comandos que nos permiten realizar búsquedas de diferentes tipos sobre todos los ficheros del sistema. ***find*** es el comando más versátil para realizar esta acción. ***locate*** es otro comando pero, a diferencia del anterior, utiliza una base de datos interna que se actualiza periódicamente y nos permite hacer búsquedas bastante más rápidas. Para acabar con los comandos de búsqueda, ***whereis*** está orientado a la búsqueda de los archivos binarios (los ejecutables), de ayuda o los de código fuente de un determinado programa.

```
17.Busca todos los archivos que empiecen por m en el directorio
  /usr/bin
18.Busca la localización del comando ls
```

## 1.6.- Tipos y contenidos de ficheros

Utilizar la extensión para determinar el tipo de un archivo no es un sistema muy eficaz ya que cualquiera puede cambiarla y generar confusiones y errores en el sistema.

Los archivos que tenemos en nuestro sistema pueden ser de muchos tipos diferentes: ejecutables, de texto, de datos, etc. A diferencia de otros sistemas, que utilizan la extensión del archivo para determinar de qué tipo son, GNU/Linux utiliza un sistema denominado de *magic numbers*, determinando con un número mágico el tipo de fichero según sus datos. El comando ***file*** nos lo indica.

```
$/tm p/pr ueb a$ f ile *
practica_1.txt: ASCII text
practica_2.txt: ASCII text
practica_3.txt: ASCII text
```

```
14.Mira de que tipo son los archivos contenidos en el directorio
  pract3
15.Averigua de que tipo son los archivos que empiezan por m en
  /usr/bin
```

Si necesitamos ver el contenido de un fichero, uno de los comandos más básicos es el ***cat***. Pasándole el nombre/s del archivo/s que queramos ver, sencillamente se muestra por pantalla.

```
$ /tmp/prueba$ cat practica_1.txt
p1
```

Para ficheros muy extensos nos irán mucho mejor los comandos ***less*** o ***more***, que permiten desplazarnos por el fichero de forma progresiva. Si el tipo de fichero es binario y queremos ver qué contiene podemos utilizar los comandos ***hexdump*** u ***od*** para ver el contenido de forma hexadecimal u otras representaciones.

```
16.Visualiza el contenidos del archivo practicas.txt
```

Otro tipo de comando muy útil son los que nos buscan un cierto patrón en el contenido de los ficheros. Con el comando ***grep*** le podemos pasar como segundo parámetro el nombre del archivo y como primero el pattern que queramos buscar.

| | |
|---|---|
| `grep "pr intf"  *.c` | **Busca la palabra printf en todos l los archivos acabados en .c** |

```
$ gr ep  p *.txt
practica_1.txt:p1
practica_2.txt:p2
practica_3.txt:p3
```

Con ***cut*** podemos separar en campos el contenido de cada línea del fichero especificando qué carácter es el separador, muy útil en tareas de administración del sistema para su automatización. De igual forma, con **paste** podemos concatenar la líneas de diferentes ficheros. También podemos coger un determinado número de líneas del principio o fin de un archivo con los comandos ***head*** y ***tail*** respectivamente. Con ***wc*** podemos contar el número de líneas o palabras, la máxima longitud de línea de un fichero, etc.

```
17.Busca la sílaba ma en el fichero apuntes.txt
18.Visualiza las primeras lineas del fichero asignatura.txt
19.Visualiza las ultimas líneas del fichero apuntes.txt
```

Finalmente, para acabar con esta sección de manipulación de ficheros lo único que nos falta por ver es cómo comparar diferentes archivos. Igual que con las otras operaciones, tenemos varios comandos que nos permiten hacerlo. **diff**, **cmp** y ***comm*** realizan comparaciones de diferentes formas y métodos en los ficheros que les indiquemos.

## 1.7.- Permisos

En cualquier sistema operativo multiusuario necesitamos que los ficheros que guardamos en nuestro disco puedan tener una serie de propiedades que nos permitan verlos, modificarlos o ejecutarlos para los usuarios que nosotros definamos. Aunque hay varias alternativas para hacer esto UNIX utiliza el sistema clásico, que combinado con todos los mecanismos de gestión de usuarios y grupos nos permite cualquier configuración posible. La idea es definir, para cada fichero o directorio, a qué usuario y grupo pertenece y qué permisos tiene para cada uno de ellos y para el resto de usuarios del sistema. Ejecutando ``***ls -l***'' veremos como por cada archivo del directorio donde estemos aparece una línea parecida a la siguiente:

```
-rwxr-xr-x  1  user1  grupo1    128931 Feb 19 2000 gpl.txt
```

Los primeros diez caracteres (empezando por la izquierda) nos indican los permisos del fichero de la siguiente forma:

*Carácter 1*: esta entrada nos indica si es un fichero o un directorio. En caso de ser un fichero

aparece el carácter -, mientras que por los directorios aparece una d'.

*Caracteres 2,3,4*: nos indican, respectivamente, los permisos de lectura, escritura y ejecución para el propietario del fichero. En caso de no tener el permiso correspondiente activado encontramos el carácter ``-'' y sino ``r'', ``w'' o ``x'' según si lo podemos leer (Read), escribir (Write) o ejecutar (eXecute).

*Caracteres 5,6,7*: estos caracteres tienen exactamente el mismo significado que anteriormente pero haciendo referencia a los permisos dados para los usuarios del grupo al que pertenece el fichero.

*Caracteres 8,9,10*: igual que en el caso anterior pero para todos los otros usuarios del sistema.

```
20.Determine que permisos tienen los archivos de los directorios
  pract1, pract2, pract3
```

Para cambiar los permisos de un determinado archivo podemos utilizar el comando ***chmod***. Debemos tener en cuenta que sólo el propietario del archivo (o el root) puede cambiar estos permisos ya que sino todo el mecanismo no tendría ningún sentido. Los permisos se indican con un número de tres cifras comprendido entre el 000 y el 777, donde cada una de las cifras codifica los permisos (lectura, escritura y ejecución) asociados al propietario del archivo, al grupo al que pertenece el archivo y al resto de usuarios, respectivamente. Algunos ejemplos

```
744 = 111 100 100 = rwx r-- r--
777 = 111 111 111 = rwx rwx rwx
654 = 110 101 100 = rw- r-x r--
```

Para cambiar el propietario de un fichero, existe el comando ***chown***, que sólo puede utilizar el root por razones de seguridad. Para cambiar el grupo de un determinado archivo se puede utilizar el comando ***chgrp***. Como podemos suponer, cuando un usuario crea un nuevo archivo, el sistema pone como propietario el usuario que lo ha creado y perteneciente al grupo primario del mismo usuario. Los permisos que se ponen por defecto al crear un nuevo archivo los podemos configurar con el comando ***umask.***

### 1.7.1.- Resumen de comandos

En la Tabla 1.2 se pueden ver la mayoría de los comandos mencionados anteriormente.

| *Comando* | *Descripción* | *Comando* | *Descripción* |
|---|---|---|---|
| **apropos** | informa sobre un comando | **locate** | búsqueda de ficheros |
| **cat** | visualiza el contenido de un fichero | **ls** | lista el contenido de un directorio |
| **cd** | cambia de directorio | **man** | enseña el manual sobre un comando |
| **cmp,diff** | compara ficheros | **mdkir** | crea un directorio |
| **cp** | copia archivos | **more** | igual a cat |
| **chgrp** | cambia el grupo de un archivo | **mv** | mueve archivos |
| **chmod** | cambia los permisos de un archivo o directorio | **pwd** | muestra el directorio en que nos encontramos |
| **chown** | cambia el propietario de un archivo | **rm** | borra archivos |
| **file** | indica el tipo de fichero | **rmdir** | borra directorios |
| **find** | búsqueda de ficheros | **tail** | visualiza n lineas del final del fichero |
| **grep** | busca un patrón en un fichero | **umask** | cambia los permisos por defecto de un archivo al ser creado |
| **head** | visualiza n lineas del comienzo del fichero | **wc** | cuenta líneas/palabras |
| **info** | idem man pero más extenso | **whatis** | breve referencia sobre un comando |
| **less** | igual a cat | **whereis** | búsqueda de archivos binarios |

*Tabla 1.2: Comandos más usuales*

# 2.- COMANDOS SHELL I

## 2.1.- El sistema Operativo Linux

Linux es un Unix libre, es decir, un sistema operativo, como el Windows o el MS-DOS (sin embargo, a diferencia de estos y otros sistemas operativos propietarios, ha sido desarrollado por miles de usuarios de computadores a través del mundo, y la desventaja de estos es que lo que te dan es lo que tu obtienes, dicho de otra forma no existe posibilidad de realizar modificaciones ni de saber como se realizó dicho sistema), que fue creado inicialmente como un hobbie por un estudiante joven, Linus Torvalds, en la universidad de Helsinki en Finlandia, con asistencia por un grupo de hackers a través de Internet. Linux tenía un interés en Minix, un sistema pequeño o abreviado del UNIX (desarrollado por Andy Tanenbaum); y decidido a desarrollar un sistema que excedió los estándares de Minix. Quería llevar a cabo un sistema operativo que aprovechase la arquitectura de 32 bits para multitarea y eliminar la barreras del direccionamiento de memoria.

Torvalds empezó escribiendo el núcleo del proyecto en ensamblador, y luego comenzó a añadir código en C, lo cual incrementó la velocidad de desarrollo, e hizo que empezara a tomarse en serio su idea.

Comenzó su trabajo en 1991 cuando realizó la versión 0.02, la cual no se dio a conocer porque ni siquiera tenía drivers de disquete, además de llevar un sistema de almacenamiento de archivos muy defectuoso. Trabajó constantemente hasta 1994 en que la versión 1.0 del núcleo(KERNEL) de Linux se concretó. Actualmente, la versión completamente equipada es la 2.6.18.1 (versión de octubre de 2006), y el desarrollo continúa (se pueden consultar en kernel.org).

Linux tiene todas las prestaciones que se pueden esperar de un Unix moderno y completamente desarrollado: multitarea real, memoria virtual, bibliotecas compartidas, carga de sistemas a demanda, compartimiento, manejo debido de la memoria y soporte de redes TCP/IP.

La parte central de Linux (conocida como núcleo o kernel) se distribuye a través de la Licencia Pública General GNU, lo que básicamente significa que puede ser copiado libremente, cambiado y

distribuido, pero no es posible imponer restricciones adicionales a los productos obtenidos y, adicionalmente, se debe dejar el código fuente disponible, de la misma forma que está disponible el código de Linux. Aún cuando Linux tenga registro de Copyright, y no sea estrictamente de dominio público, la licencia tiene por objeto asegurar que Linux siga siendo gratuito y a la vez estándar.

Por su naturaleza Linux se distribuye libremente y puede ser obtenido y utilizado sin restricciones por cualquier persona, organización o empresas que así lo desee, sin necesidad de que tenga que firmar ningún documento ni inscribirse como usuario. Por todo ello, es muy difícil establecer quienes son los principales usuarios de Linux. No obstante se sabe que actualmente Linux está siendo utilizado ampliamente en soportar servicios en Internet, lo utilizan Universidades alrededor del todo el mundo para sus redes y sus clases, lo utilizan empresas productoras de equipamiento industrial para vender como software de apoyo a su maquinaria, lo utilizan cadenas de supermercados, estaciones de servicio y muchas instituciones del gobierno y militares de varios países. Obviamente, también es utilizado por miles de usuarios en sus computadores personales. El apoyo más grande, sin duda, ha sido Internet ya que a través de ella se ha podido demostrar que se puede crear un sistema operativo para todos los usuarios sin la necesidad de fines lucrativos.

Linux tiene una mascota oficial, el pingüino de Linux, que fue seleccionado por Linus Torvalds para representar la imagen que se asocia al sistema operativo que él creó.

Básicamente podemos decir que hoy Linux es un sistema muy completo. El proyecto de Linus Torvalds aún no ha terminado, y se piensa que nunca se terminará por ésta continua evolución de la Informática.

### 2.1.1.- Estructura e Interfaces de Linux

Como muchos sistemas, Linux puede verse como una pirámide (Figura 2.1). En la base tenemos el hardware, y sobre él, el sistema operativo. Su función es controlar el hardware y suministrar la interfaz de llamadas al sistema a todos los programas. Esta interfaz permite a los usuarios crear y gestionar procesos, archivos, y otros recursos. Como las llamadas al sistema deben hacerse en

ensamblador, el sistema dispone de una biblioteca estándar para facilitar la labor del programador, y que puede ser invocada desde un programa en C. Además del sistema operativo y la biblioteca de llamadas al sistema, Linux suministra un gran número de programas estándares. Entre estos se incluye intérpretes de órdenes, compiladores, editores, y utilidades de manipulación de archivos.

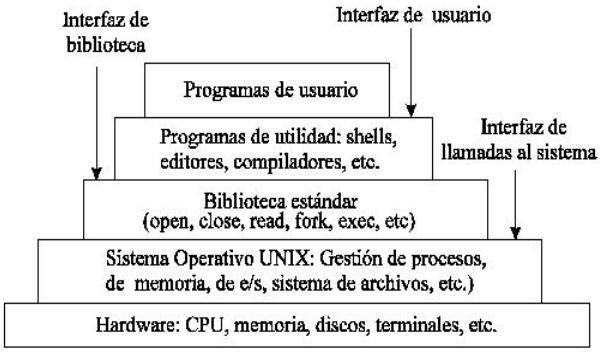



*Figura 2.1: Las capas del sistema UNIX*

El objetivo de nuestras prácticas es aprender a manejar la interfaz de usuario mediante el uso de órdenes y shell.

## 2.2.- Los interpretes de órdenes

Un intérprete de órdenes, o shell en la terminología Linux, está construido como un programa normal de usuario. Esto tiene la enorme ventaja de que podemos cambiar de intérprete de órdenes según nuestras necesidades o preferencias. Existen diferentes shells: el Bourne Again Shell (bash), el TC shell (tcsh), y el Z shell. Estos shells no son exclusivos de Linux, se distribuyen libremente y pueden compilarse en cualquier sistema UNIX. Podemos ver los shell de los que dispone nuestro sistema mirando en el archivo /etc/shells.

El shell es un programa que básicamente realiza las siguientes tareas:

```
for(;;)     {
      imprime indicador de órdenes;
      lee la línea de ordenes;
      analiza la línea de ordenes (arg0,arg1,...,>,<,|,&,...);
      prepara entorno según lo que aparece en línea de ordenes;
      crea un proceso para ejecutar orden;
```

```
        if (estamos en el proceso hijo)     {
              ejecuta la orden dada en arg0;
        else /* es el proceso padre */
              if (línea ordenes no aparece el símbolo &)
                    espera hasta que finalice el hijo;
        }
}
```

Cada shell, además de ejecutar las órdenes de LINUX, tiene sus propias órdenes y variables, lo que lo convierte en un lenguaje de programación. La ventaja que presenta frente a otros lenguajes es su alta productividad, una tarea escrita en el lenguaje del shell suele tener menos código que si está escrita en un lenguaje como C.

Respecto a cuando utilizar el shell y cuando utilizar otro lenguaje de programación como C, indicar como recomendación general que debemos utilizar el shell cuando necesitemos hacer algo con muchos archivos, o debamos de hacer la misma tarea repetitivamente. No deberíamos usar el shell cuando la tarea sea muy compleja, requiera gran eficiencia, necesite de un entorno hardware diferente, o requiera diferentes herramientas software.

Respecto a las órdenes, en el bash (el que nosotros vamos a utilizar en prácticas) podemos encontrar los siguientes tipos:

1. **Alias**: son abreviaciones para órdenes existentes que se definen dentro de la memoria del shell.
2. **Ordenes empotradas**: rutinas implementadas internamente en el shell.
3. **Programas ejecutables**: programas que residen en disco.

Cuando el shell está preparado para ejecutar una orden, evalúa el tipo de orden según el orden que aparece en la lista anterior: comprueba si es un alias; si no, si es una orden empotrada; y por último, un programa ejecutable (en este último caso la eficiencia es menor pues hay que acceder a disco para localizar el ejecutable de la orden). Por tanto, si tenemos un alias con el mismo nombre que un programa ejecutable, primero ejecutaría el alias.

Los propios shell se comportan como una orden, es decir se pueden ejecutar.

```
Mira de que shells dispone el sistema y entra en alguna de ellas.
```

**Entrar en una shell = nombre de la shell (las shell se encuentran en /bin/)**

**Salir de una shell = exit, o Ctrl+D**

## 2.3.- Una sesión de trabajo

Cuando arrancamos el sistema, tras muchos mensajes de inicialización, aparece una pantalla gráfica que nos pedirá las siguiente información

```
login: i62xxxx
password:
```

En este momento, el programa /bin/login verifica nuestra identidad comprobando el primer campo del archivo /etc/passwd que contiene nuestro nombre usuario. Si nuestro nombre esta allí, compara el password dado, o palabra calve, con la forma encriptada del mismo que hay en el archivo /etc/shadow. Una vez verificado, el programa login establece el entorno de trabajo que se pasará al shell, es decir, se asignan a las variables HOME, SHELL, USER, y LOGNAME los valores extraídos del archivo /etc/passwd. Después, se crea el shell de entrada o login shell, con lo que podemos iniciar la sesión de trabajo.

La palabra clave debe mantenerse en secreto para nuestra protección y cambiarse con cierta frecuencia (al menos dos veces al año). Podemos cambiar la clave de acceso con la orden **passwd**.

Una vez en el sistema, disponemos de un manual en línea para consultar la sintaxis y opciones de sus órdenes, e incluso algunos ejemplos de como se utiliza. El manual en línea se puede consultar con la orden **man**, cuya sintaxis es:

**man [opciones] [sección] orden**

Por ejemplo, si queremos ver que hace y cuales son las opciones de la orden **ls**, que lista el contenido de un directorio, podemos ejecutar

```
man ls
```

Sólo comentar la opción **-k** que muestra las secciones que están relacionadas con una palabra clave. Esta es muy útil cuando sabemos lo que queremos hacer pero no conocemos la orden que lo hace. La orden **apropos** realiza la misma función.

```
man -k <palabra clave>
apropos <palabra clave>
```

```
Busca las páginas de ayuda relacionadas con los programas shell
```

### 2.3.1.- El sistema de archivos

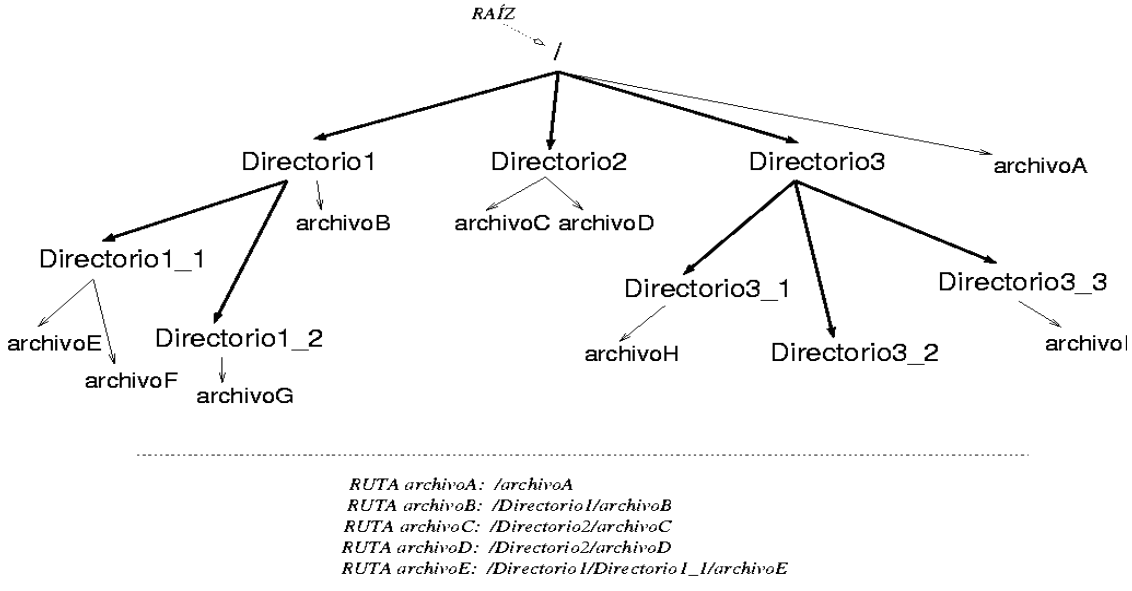



*Figura 2.2: El sistema de archivos de Linux*

Uno de los elementos del sistema con el que interaccionamos muy frecuentemente es el sistema de archivos. En Linux, todos los archivos están almacenados en una jerarquía con estructura de árbol, que consta de archivos y directorios, similar al de Windows. En la parte más alta del árbol, está el *directorio raíz*, que se representa con un /, de él cuelgan todos los directorios y archivos del sistema. Cuando el nombre de un archivo comienza por /, estamos indicando que es un nombre de archivo absoluto (cuelga del directorio raíz). Cuando encontramos el símbolo / entre el nombre de un archivo o directorio, éste se utiliza simplemente como separador.

Los directorios que aparecen en un sistema Linux pueden variar de unas instalaciones, y versiones, a otras, pero casi siempre encontraremos algunos directorios comunes a todos los sistemas. La mayoría de los sistemas siguen el estándar FHS que determina donde deben colarse los ficheros según su tipo y uso. Por ejemplo, y de forma simplista:

```
/bin      Directorio con programas ejecutables del sistema
/boot     Imagen del sistema
/dev      Directorio de dispositivos
/etc      Directorio de archivos de configuración
/home     De él cuelgan los directorios de los usuarios
/lib      Bibliotecas de programación estándares
/sbin     Ordenes de administración del sistema
/tmp      Directorio para archivos temporales
/usr      Directorio de programas
/usr/bin  Directorio de programas ejecutables de aplicación
/usr/X11  Archivos del sistema de ventanas X
/usr/lib  Archivos de configuración de paquetes y bibliotecas
/usr/local/bin  Ordenes añadidas localmente
/usr/man        Páginas de manual en línea de sistema
/usr/src/linux  Archivos fuentes del kernel de Linux
/var            Desbordamiento para algunos archivos grandes
...
```

Al entrar al sistema, éste nos asigna un directorio por defecto, nuestro directorio **home**. En muchos casos, el nombre de este directorio se hace coincidir con el nombre del usuario y suele estar situado en el directorio /home. Por ejemplo, **/home/i62xxxxx**. En sistemas como el nuestro, con miles de usuarios, los directorios **home** están ubicados en otros directorios.

### 2.3.2.- El shell de entrada

Suponiendo una instalación general, si deseamos trabajar con *bash* deberemos ver cual es el *shell de entrada* del sistema y, si no es el deseado, cambiarlo de forma temporal o permanente.

### Determinación del shell de entrada

Cuando entramos al sistema después de dar nuestro nombre de usuario y nuestra contraseña o palabra clave, Linux inicia un *shell* por defecto que es el que nos muestra el indicador de órdenes de la sesión. Este shell de entrada se denomina *login shell*. Es el administrador del sistema el que establece este shell cuando crea un usuario.

¿Cómo sabemos cual es el shell asignado por defecto? Cuando se inicia el shell, este asigna a una variable el valor del shell actual (en las prácticas siguientes hablaremos detenidamente de las variables). Esta variable se llama $SHELL. Podemos ver el valor de la variable con la orden empotrada echo:

```
echo $SHELL
bin/tcsh
```

```
Averigua cual es el shell asignado a tu sesión de trabajo
```

Podemos variar el shell por defecto con la orden **chsh** (change shell) pero su efecto sólo tiene lugar cuando nos salimos del sistema y volvemos a entrar. Se puede obtener un cambio temporal de shell invocando directamente a un programa shell desde el indicador de ordenes del shell en el que estamos. Este cambio permanecerá mientras no finalicemos el shell actual con la orden **exit**, **Ctrl+D**

```
chsh -s <shell> <user>
```

## 2.4.- Los procesos

El hecho que el sistema operativo sea multitarea implica que podemos lanzar más de un programa a la vez. Un proceso no es más que un programa o aplicación cargado en memoria y ejecutándose. Aunque nuestro ordenador sólo disponga de una CPU, el sistema operativo se encarga de repartir el tiempo de procesamiento de la misma para que varios procesos puedan ir realizando sus operaciones, dando la sensación que se están ejecutando todos a la vez.

### 2.4.1.- Algunas herramientas para vigilar procesos

La forma habitual de ver el estado de nuestro sistema es mediante el uso de herramientas que permiten vigilar la ejecución de los procesos. En Linux, estas herramientas están contenidas en el paquete *procps*. Entre éstas encontramos: **top, vmstat, free, uptime**, **w**, , etc. Estas herramientas utilizan el pseudo-sistema de archivos */proc* para obtener la información. En el caso de Red Hat, algunas herramientas también están disponibles en los entornos de ventanas, como es el caso de ktop, kpm y procinfo en kde, ó gtop en gnome.

#### Comprobación de los dispositivos

El pseudo-sistema de archivos */proc* permite consultar los dispositivos reconocidos por el kernel de Linux. La siguiente orden nos muestra parte de la información que podemos obtener de este tipo:

```
cat /proc/cpuinfo     # información sobre el procesador
cat /proc/devices     # sobre los dispositivos
cat /proc/interrupts # interrupciones
cat /proc/dma         # DMA
cat /proc/ioports     # puertos de entrada/salida de dispositivos
cat /proc/pci         # dispositivos PCI del sistema
ls -l /proc/ide       # dispositivos IDE
ls -l /proc/scsi      # dispositivos SCSI
```

#### Orden ps

La orden **ps** nos da una instantánea de los procesos del sistema actuales (cuando se ejecuta la orden). Si queremos tomar diferentes instantáneas, debemos de utilizar **top**, que nos informa de forma interactiva de los procesos del sistema, del estado de utilización de la CPU, la memoria utilizada y libre, la RAM que utiliza cada proceso, etc.

Algunas de las opciones más utilizadas:

```
ps -e   #muestra todos los procesos
ps -x  #procesos sin terminal de control (tty)
```

```
ps -u  #procesos de un usuario especificado
ps -l  #formato largo
ps -o  #formato definido por el usuario, ...
```

**ps**, sin opciones ni argumentos, da información sobre los procesos de la sesión actual.

```
Muestra los porcentajes de uso de la CPU de los procesos del
sistema
Prueba el comando top
```

El comando **kill** nos permite enviar señales a los procesos que nos interese. Hay muchos tipos diferentes de señales, que podemos ver en el manual de **kill**, aunque las más utilizadas son las que nos sirven para obligar a un proceso a que termine o pause su ejecución. Con CTRL+Z podemos pausar un programa y revivirlo con **fg**. Si queremos que el proceso se ejecute en segundo plano utilizaremos **bg.**

```
comando &
ctr+z
fg <id proceso>
bg <id proceso>
kill <id proceso>
```

### 2.4.2.- Metacaracteres sintácticos

Una vez aprendido como utilizar algunos de los comandos del sistema, es muy probable que en algunos casos nos interese utilizarlos de forma simultánea para agilizar más las acciones que queramos realizar. Los metacaracteres sintácticos nos ayudan en este sentido.

Sirven para combinar varias órdenes de LINUX y construir una única orden lógica. Suministran una forma de ejecución condicional basada en el resultado de la orden anterior. La Tabla 2.1 muestra estos caracteres y da una descripción de su función.

Los metacaracteres sintácticos permiten materializar la filosofía de trabajo de LINUX de caja de herramientas: dadas las ordenes que necesitamos para realizar un trabajo, los metacaracteres sintácticos permiten componerlas ("pegarlas") para resolverlo. Esta filosofía es en gran medida la responsable de que las órdenes de LINUX realicen funciones muy concretas, y sean parcas en dar información al usuario.

| ***Metacarácter*** | ***Descripción de la función*** |
|---|---|
| ; | Separador entre órdenes que se ejecutan secuencialmente |
| \| | Separación entre órdenes que forman parte de un cauce (pipeline). La salida de la orden de la izquierda del separador es la entrada de la orden de la derecha del separador. |
| () | Se usan para aislar ordenes separadas por ; ó \|. Las ordenes dentro de los paréntesis, ejecutadas en su propio shell, son tratadas como una única orden. Incluir un cauce dentro de paréntesis, nos permite a su vez incluirlo en otros cauces. |
| & | Indicador de trabajo en segundo plano (background). Indica al shell que debe ejecutar el trabajo en segundo plano. |
| \|\| | Separador entre órdenes, donde la orden que sigue al \|\| sólo se ejecuta si la orden precedente falla. |
| && | Separador entre ordenes, en la que la orden que sigue al && se ejecuta sólo si la orden precedente tiene éxito. |

*Tabla 2.1: Metacaracteres sintácticos*

### Uniendo órdenes con ;

El uso del punto y coma (**;**) permite escribir dos o más órdenes en la misma línea. Las órdenes se ejecutan secuencialmente, como si se hubiesen dado en líneas sucesivas. En programas shell, se utiliza por razones estéticas (permite una asociación visual entre ordenes relacionadas). En el indicador de órdenes, permite ejecutar varias ordenes sin tener que esperar a que se complete una orden para introducir la siguiente.

En el ejemplo siguiente, utilizamos la orden **pwd** que nos permite ver el camino absoluto del directorio de trabajo actual, y la orden **cd** (para cambiar de directorio).

```
$ pwd
/home/in1xxxxxx
$ cd  so2006  ; ls # Cambiamos de directorio y hacemos ls.
miprograma1
miprograma2
$ pwd
/home/in1xxxxx/so2006
```

### Creando cauces con |

Antes de ver los cauces, debemos comentar algo sobre el entorno de ejecución de los programas en LINUX. Parte del entorno de ejecución de un programa son los archivos que utiliza. Cuando el sistema operativo crea un proceso, le abre tres archivos: la entrada estándar (*stdin*), la salida estándar (*stdout*) y el error estándar (*stderr*), que se corresponden con los descriptores de archivo (o handles) 0, 1 y 2, respectivamente (ver Figura 2.3). Por defecto, estos archivos se corresponden con el teclado para la entrada estándar, y la pantalla para la salida y error estándares, pero como veremos en breve, estas asociaciones se pueden cambiar.


*Figura 2.3: Entrada, salida y error, estándares*

Los cauces son una característica distintiva de LINUX. Un cauce conecta la salida estándar de la orden que aparece a la izquierda del símbolo "|" con la entrada estándar de la orden que aparece a la derecha. El flujo de información entre ambas órdenes se realiza a través del sistema operativo.

Para ver un ejemplo, usaremos la orden **who** que produce un informe de los usuarios que están conectados actualmente al sistema. Cada una de las líneas que muestra hace referencia a un usuario, y da su nombre de usuario, el terminal al que está conectado, y la fecha y hora de entrada al sistema.

```
$ who
maria tty2     May 10 13:35
luis  tty6     May 10 9:00
```

¿Qué debemos hacer si queremos obtener esta misma información pero ordenada alfabéticamente? La solución es sencilla, usar la orden **sort** que sirve para clasificar.

```
$ who | sort  # La salida de who se utiliza como entrada de sort
luis tty6 May 10 9:00
maria tty2 May 10 13:35
```

¿Qué ha ocurrido? La salida de **who** se ha pasado como entrada a la orden **sort**. Esto nos ilustra, además, que las órdenes de Linux están construidas siguiendo el siguiente convenio: cuando en una orden no se especifican archivos de entrada y/o salida, el sistema toma por defecto la entrada o salida estándar, respectivamente. Por ejemplo, ¿qué ocurre si ejecutamos **sort** sin argumentos? Sencillo, se ordena la entrada estándar y se envía a la salida estándar.

**sort <opciones> <fichero>**

```
Prueba el comando sort sin argumentos. Recuerda que Ctrl+d indica
fin de archivo
```

### Combinado ordenes con ()

En ocasiones, necesitaremos aislar un cauce, o una secuencia de punto y coma, del resto de una línea de órdenes. Para ilustrarlo, vamos a usar las ordenes **date**, que nos da la fecha y hora del sistema, y la orden **wc**, que nos permite conocer cuantas líneas, palabras y caracteres, tiene el archivo que se pasa como argumento. Por ejemplo, las siguientes órdenes tienen resultados diferentes:

```
$ date; who | wc #ejecuta date; ejecuta who cuya salida pasa a wc
Wed Oct 11 10:12:04 WET 1995
1 5 31
$ (date; who)|wc  #ejecuta date y who, sus salidas se pasan a wc
```

```
2 12 64
¿Por qué?
```

### Ejecutando ordenes en segundo plano con &

Linux permite dos formas de ejecución de órdenes:

- Ejecución de órdenes en **primer plano** (*foreground*): en este caso, el indicador de órdenes no aparece hasta que ha finalizado la orden que mandamos ejecutar. Por ejemplo, las órdenes que hemos ejecutado hasta ahora.

- Ejecución en **segundo plano** (*background*): en este segundo caso, el indicador reaparece inmediatamente, lo cual nos permite ejecutar otras ordenes aunque la que se lanzó en segundo plano no haya terminado.

Para ejecutar una orden en segundo plano sólo debemos finalizar la línea de ordenes con **&**. Supongamos que deseamos realizar la compilación de un programa escrito en C y que la misma dura algunos minutos, si queremos seguir trabajando mientras el sistema realiza el trabajo, podemos ejecutarla en segundo plano:

```
$ gcc miprograma.c &    # compilación lanzada de fondo
[1] 1305
10% ls    # seguimos mientras otro proceso realiza la compilación
```

Vemos como el shell responde al carácter **&** indicando el número del trabajo que hemos lanzado. Este aparece entre corchetes, seguido del identificador del proceso que ejecuta la orden. El número de trabajo nos servirá para controlar los procesos en segundo plano. Otros ejemplos:

```
$ who  | wc  &              # lazamos de fondo un cauce
[2]   1356 1357
$ (cd  midi rect orio  ; ls)  & # lanzamos de fondo una secuencia de
 [3]   1400                    ordenes
$ cd  midi rect orio &  ls&   # & sirve de separador; perdemos la
                              secuencialidad
[1]   1402
```

```
[2]   1403
```

### Ejecución condicional de ordenes con || y &&

El shell suministra dos metacaracteres que permiten la ejecución condicional de órdenes basada en el estado de finalización de una de ellas. Separar dos ordenes con || o **&&**, provoca que el shell compruebe el estado de finalización de la primera y ejecute la segunda sólo si la primera falla o tiene éxito, respectivamente. Veamos un ejemplo,

```
find *.r || echo "No se encuentra el archivo"
find -name "*.txt" && echo "Archivos encontrados"
```

**find <*ruta*> -name <*nombre*> <*otras opciones*>**

**find <*ruta*> -size <*tamaño*> <*otras opciones*>**

**find <*ruta*> -atime <*numero dias*> <*otras opciones*>**

```
Muestra un mensaje que te indique si esta conectado o no un
usuario. Utiliza la orden echo para mostrar el mensaje
```

**echo <mensaje>**

No hay restricción que limite el número de órdenes que aparecen antes del ||, pero sólo se evalúa el estado de la última.

## 2.4.3.- Órdenes para el control de trabajos

Se denomina trabajo, o job, a cada orden que mandamos a ejecutar al shell. El shell mantiene la pista de todos los trabajos que no han finalizado y suministra mecanismos para manipularlos. En esta sección, vamos a ver esos mecanismos que nos suministra el shell para controlar la ejecución de nuestros trabajos.

### jobs

Un trabajo puede estar en uno de los tres estados siguientes: primer plano, segundo plano, o suspendido (*suspended* o *stopped*, según el sistema.). La orden **jobs** muestra los trabajos actualmente asociados con la sesión del Shell (lanzados en un terminal). Su sintaxis es:

```
jobs [ -l ]
```

Lista los trabajos activos bajo el control del usuario. Con la opción -l se listan, además de la información normal, los identificadores de los procesos.

Prueba lo siguiente:

```
$ gcc –o prueba prueba.c &
$ sort misdatos > misdat.ord &
$ vi texto
  ..
<CTRL>-Z                  # suspendemos la ejecución de vi
$ jobs
¿Qué información proporciona?
```

Si un trabajo contiene un cauce, las ordenes del cauce que han acabado son mostradas en una línea diferente de aquellas que están aún ejecutándose.

Las siguientes órdenes que vamos a ver a continuación nos permiten cambiar el estado de un trabajo. La mayor parte de ellas admiten como argumento un identificador de trabajo.

### fg

La orden **fg** se usa para traer trabajos de fondo, o suspendidos, a primer plano:

```
fg [ % [job ] ]
```

Sin argumentos, o con %, **fg** trae el trabajo actual a primer plano. Con un identificador de trabajo, trae a primer plano el trabajo especificado.

Supongamos que hay dos trabajos en segundo plano, como muestra el ejemplo:

```
$ jobs
```

```
[1] + Stopped (user)            xterm
[2] - Running                   xclock
```

Si deseamos poner en primer plano **xterm**, dado que esta suspendido, podemos ejecutar

```
$ fg %1
 xterm
```

Ahora, el shell no responde hasta que finalicemos el proceso xterm o lo devolvamos a segundo plano. Este tipo de transferencia, nos permite lanzar de fondo programas que necesiten mucho procesamiento inicial - tal como crear una base de datos - y retomarlos cuando necesitemos interactuar con él.

### bg

Contraria a la orden fg, bg envía a segundo plano un programa:

```
bg [ %job ]
```

Ejecuta el trabajo actual, o especificado, en background. Esto se ilustra en el ejemplo.

```
54% jobs
[1]     + Stopped (user)        xterm
[2]     - Running               xclock
55% bg %1
[1]     xterm &
```

### La orden %

La orden **%** se utiliza para cambiar el estado de un trabajo.

```
%[ job ] [ & ]
```

Esta lleva a primer plano el trabajo actual, si no tiene argumento, o el especificado por job. Si al final del orden está presente el &, envía el trabajo a segundo plano.

La orden % es en realidad una abreviación de **fg** y **bg**.

### kill

La orden **kill** envia una señal a un(os) proceso(s). Tiene dos formas:

```
kill [ -sig ] { pid | %job } ...
kill -l
```

La acción por defecto de **kill** es enviar la señal de finalizar al proceso(s) indicado(s) por PID o número de job. Las señales, **-sig**, se indican por números o nombres. Con la opción **-l**, lista los nombres de la señales que pueden ser enviadas.

Las señales son en LINUX el medio básico para notificar de la ocurrencia de eventos a los procesos. Por ejemplo, las pulsaciones <CTRL>-C, <CTRL>-Z, etc. lo que hace en realidad es provocar que el manejador de teclado genere determinadas señales que son enviadas al proceso en ejecución. También, se envían señales cuando un programa referencia una dirección inválida (p.e. un puntero no inicializado) del proceso. En este caso, se genera la señal **SIGSEGV**, que produce el mensaje de sobra conocido "Segmentation fault: core dump".

## 2.5.- Metacaracteres de entrada/salida o de dirección

Los metacaracteres de entradas/salidas son otra de las características distintivas de LINUX. Con ellos, podemos redireccionar (tomar) la entrada de una orden desde un archivo en lugar del teclado, redireccionar (llevar) la salida estándar a un archivo en lugar de a la pantalla, o encauzar su entrada/salida a/desde otra orden. También, podremos mezclar los flujos de información de la salida y el error estándares para que salgan juntos por la salida estándar. Tenemos un resumen en la Tabla 2.2.

| ***Metacarácter*** | ***Descripción de la función*** |
|---|---|
| < *nombre* | Redirecciona la entrada de una orden para leer del archivo *nombre*. |
| >*nombre* | Redirecciona la salida de una orden para escribir en el archivo *nombre*. Si *nombre* existe, lo sobreescribe. |
| 2> *nombre* | Redirecciona el error (*stderr*) a un fichero. Si *nombre* existe, lo |

| *Metacarácter* | *Descripción de la función* |
|---|---|
| | sobreescribe. |
| >& *nombre* | La salida de *stderr* se combina con *stdout*, y se escriben en *nombre*. |
| >> *nombre* | La salida de la orden se añade al final del archivo *nombre*. |
| 2>> *nombre* | La salida de *stderr* se añade al final del archivo *nombre* |
| >>& *nombre* | Añade la salida de *stderr*, combinada con *stdout* y las añade al final de *nombre*. |
| \| | Crea un cauce entre dos órdenes. La salida estándar de la orden a la izquierda de símbolo se conecta a la entrada estándar de la orden de la derecha del signo. |
| \|& | Crea un cauce entre dos ordenes, con las salidas de *stderr* y *stdout* de la orden de la izquierda combinadas y conectadas con la entrada de la orden de la derecha. |

*Tabla 2.2: Metacaracteres de entrada/salida o de dirección*

### Redirección de la entrada estándar con <

Algunas ordenes LINUX toman su entrada de archivos cuyo nombre se pasa como argumento, pero si estos no se especifican, como vimos anteriormente, la lectura se lleva a cabo de *stdin*. Otras órdenes sólo leen de la entrada estándar, por lo que si queremos que lean de archivo debemos utilizar la redirección **<**.

Por ejemplo, la orden **mail** (programa básico de correo electrónico) lee su entrada de *stdin* y no acepta un nombre de archivo como argumento (espera que escribamos el mensaje por teclado). Sin embargo, podemos escribir el mensaje en un archivo y pasárselo a la orden:

```
$ cat >mimensaje
Hola Juan: este es un mensaje de prueba
<CTRL>-d
$ mail x333333 < mimensaje
$ _
```

Como podemos observar, la orden **cat >mimensaje** indica al shell que el contenido de la entrada estándar debe ser incluido en el archivo mimensaje.

### Redirección de salida con >, 2> y >&

Las salidas de las órdenes van normalmente a *stdout*, y muchas órdenes escriben mensajes de error o información adicional en *stderr*. La redirección de la salida permite enviar las salidas a un archivo. En este caso los mensajes que salen por *stderr* siguen saliendo por pantalla. Si queremos redireccionar el error a un fichero independiente utilizamos el metacarácter **2>** y si queremos combinar en un mismo fichero la *stdout* y *stderr*, utilizamos el metacaracter **>&**.

Los ejemplos muestran los diferentes metacaracteres de redirección.

```
$ ls > temporal          # El resultado de ls se almacena en
temporal
$ cat temporal
archiv1
practicas
proyecto
$ cat temporil# Suponemos que temporil no existe
cat: temporil: No existe el fichero o el directorio
$ cat temporil > errores
cat: temporil: No existe el fichero o el directorio
```

Observamos, como sigue saliendo por pantalla el mensaje aunque tenemos redirigida la salida estándar a un archivo. Esto se debe a que, como dijimos antes, algunas órdenes generan información por el error estándar, no sólo por la salida estándar. Por ello, debemos utilizar la redirección **2>** o **&>** para no seguir viéndolo por pantalla.

```
$ cat temporil 2> errores
$ cat errores
ca: temporil: No existe el fichero o el directorio
$ _
```

### Redirección de entrada dentro de programas con <<

El metacarácter **<<** nos permite redireccionar líneas desde un programa a la entrada de una orden dentro del programa. La redirección se realiza mediante un mecanismo conocido como documentación aquí. Esta redirección permite escribir mensajes multilínea desde un programa sin necesidad de usar múltiples órdenes echo.

Para utilizarlo, escribimos a continuación de **<<** un delimitador, es decir, un conjunto único de caracteres que marcarán el fin de las líneas de entrada para la documentación aquí. Cada una de las líneas de entrada que se vaya introduciendo se examina para igualar el delimitador sin realizar ninguna sustitución, o expansión de metacaracteres. Una vez introducido el delimitador, se ejecuta la orden. Esto nos va a permitir ajustar dinámicamente los mensajes u ordenes.

El siguiente ejemplo muestra como usar este mecanismo para generar un mensaje de bienvenida para un usuario adaptándolo a su entorno. Para ello, hacemos uso de las variables **$user** y **$PWD** que nos dan el login de usuario y el directorio de trabajo actual, respectivamente.

```
$ cat << FIN_BIENVENIDA
Hola $user
Su directorio actual es $PWD
No olvide mirar las noticias
FIN_BIENVENIDA
Hola x3333333
Su directorio actual es /d5/home/x3333333
No olvide mirar las noticias
$ _
```

Como vemos, **cat** esta aceptando información hasta que encuentra la palabra clave que le hemos indicado tras el <<, en nuestro ejemplo, FIN_BIENVENIDA; cuando la encuentra entiende que la entrada de información por *stdin* ha finalizado.

```
Genera una secuencia de números. Dicha secuencia acabará cuando
se introduzca un 0.
```

### Algo más sobre cauces y redirecciones

Si deseamos mezclar los mecanismos de cauces y redireccionamiento, encontramos que el único lugar donde la redirección de archivos se acomoda dentro de un cauce es al final de la última orden del cauce. En este caso, podemos usar la redirección de la salida para capturar la salida final en un archivo. Si intentásemos redireccionar la salida de una orden al principio o en medio de un encauzamiento, todos los datos irán al archivo y nada fluirá a través del resto del cauce. Para permitir introducir redireccionamiento en cualquier orden de un cauce, existe la orden **tee** que a partir de un flujo de entrada desdobla éste en dos: uno de ellos puede ir redireccionado a un archivo y el otro al cauce. Con la redirección de *stdin* pasa algo similar, sólo podemos usarla en la primera orden del cauce. Si la usamos en medio, se ignora la información que fluye por el cauce.

**tee <*opciones*> <*fichero*>**

### Archivos de dispositivos y redirecciones

En Linux, el directorio */dev* contiene los archivos de dispositivos y pseudo-dispositivos (dispositivos lógicos que no se corresponden con un dispositivo hardware). Uno de estos pseudo-dispositivos es */dev/null* que actúa como un sumidero de información, de forma que todo lo que se le envía es descartado por el sistema y, por tanto, podemos enviarle la información que no necesitemos. Por ejemplo, la orden **time** ejecuta otra orden pasada como argumento y nos da los tiempos de ejecución de la misma (modo kernel, usuario y tiempo de reloj).

```
$ time ls
COORDINA.PS      arch-survey.ps     complete.wav       woods.ps
real    0m0.003s
user    0m0.001s
sys     0m0.002s
```

Como vemos, **time** muestra el resultado de ejecutar la orden **ls** y el tiempo de ejecución. Si no estamos interesados en el resultado de **ls**, podemos descartarla enviándola a **/dev/null**. Nota: como se puede observar los tiempos de ejecución siguen viéndose en pantalla aunque hemos redireccionado la salida estándar. Esto se debe a que los tiempos salen por el error estándar.

```
% time ls > /dev/null
real    0m0.003s
user    0m0.001s
sys     0m0.002s
```

## 2.6.- Empaquetado y compresión de archivos con tar y gzip

Comprimir un archivo, agrupar varios en uno solo o ver qué contiene un archivo comprimido son tareas que utilizaremos frecuentemente para hacer copias de seguridad, transportar archivos de un sitio a otro, etc. Aunque existen multitud de programas diferentes que nos permiten hacer esta clase de operaciones, generalmente en todos los sistemas UNIX encontraremos la herramienta **tar**. Este programa nos permite manipular de cualquier forma uno o varios archivos para comprimirlos, agruparlos, etc.

Aunque con el mismo **tar** podemos comprimir archivos, la aplicación en sí misma no es de compresión. Como hemos dicho, para ello utiliza programas externos como el **gzip**.

### Compresión de archivos

La compresión de archivos nos permite ahorrar espacio de disco y tiempo en la transmisión de los mismos.

Para comprimir y descomprimir archivos podemos utilizar las órdenes cuya sintaxis aparece a continuación:

```
gzip [opciones] [-S sufijo] [nombre ...]
gunzip [opciones] [-S sufijo] [nombre ...]
bzip2 [opciones] [-S sufijo] [nombre ...]
bunzip2 [opciones] [-S sufijo] [nombre ...]
```

**gzip** y **bzip2** comprimen uno o varios archivos. Cada archivo es sustituido por uno cuya extensión es .gz o .bz, respectivamente. Los archivos son restaurados con las ordenes gzip -d,

gunzip , bzip2 -d, bunzip2,

Veamos algunos ejemplos.

```
$ gzip  archivo1 archivos.gz
$ bzip2 archivo2 archivos2.bz
```

Podemos descomprimir los archivos en la salida estándar en las dos formas siguientes:

```
$ gunzip archivos.gz
$ bzip2 –d archivos2.bz
```

Por último, indicar que si queremos crear un único archivo con múltiples componentes que puedan extraerse posteriormente de forma independiente, es recomendable utilizar **tar**, pues **gzi**p esta pensado para complementar a tar, no para sustituirlo.

## Empaquetado de archivos

La orden **tar** se utiliza para empaquetar un grupo de archivos en un único archivo, que tiene la estructura de un encabezado de 512 bytes y el archivo, tantas veces como archivos empaquetemos. Se utiliza para almacenar jerarquías de directorios, tener copias de seguridad y ahorrar espacio. Es la forma habitual de suministrar parches y distribuciones de sistemas.

La sintaxis de la orden tar es:

```
tar [opciones] [archivos]
```

Las opciones de **tar** pueden ir o no precedidas del -. Si bien tiene numerosas opciones, sólo vamos a ver algunas de ellas:

**c Crea un nuevo paquete para almacenar los archivos.**

**r Graba archivos al final del paquete**

**t Enumera el contenido del paquete.**

**f Especifica el archivo en el que se creará el paquete**

**x Extrae los archivos del paquete**

**v Utiliza en modo verboso**

Vamos a ver un ejemplo de empaquetado en un archivo de disco de una jerarquía de directorios (todos los archivos directorios que hay en el directorio MisPracticasSOI):

```
$ cd MisPracticasSOI
$ tar cvf practicasSoi.tar .
./Sesion1/archivo1
...
./Sesion4/achivon
$ file practicasSoi.tar
practicasSoi.tar: GNU tar archive
```

Para restaurar posteriormente en el directorio actual todos los archivos podemos utilizar la opción x:

```
$ tar xvf practicasSoi.tar
```

Observamos como el archivo tar no desaparece al finalizar el trabajo, pues nos sirve como copia de seguridad. Con las opciones x y t también podemos recuperar archivos concretos: con la primera debemos conocer el camino de acceso al archivo, con la segunda no. Por ejemplo, podemos recuperar una práctica concreta de la forma:

```
$ tar xvf practicasSoi.tar Sesion2/ejercicio1.tcsh
```

### Combinando la compresión y el empaquetado

Podemos combinar la capacidad de compresión con la de empaqueta y así reducir el tamaño. Para ello el **tar** tiene dos opciones más:

**z Comprime utilizando gzip**

**j Comprime utilizando bzip2**

Las extensiones de los archivos son tgz y tbz, respectivamente. A modo de ejemplo:

```
$ tar cvzf misPracticas.tgz mispracticas
```

Empaqueta y comprime el directorio misPracticas y su contenido en un único archivo llamado misPracticas.tgz

```
$ tar xvzf misPracticas.tgz
```

Descomprime y desempaqueta el archivo misPracticas.tgz, manteniendo la estructura de directorios inicial.

## 2.7.- Ejercicios

1.- Desplázate hasta el directorio /bin y genera los siguientes listados de archivos (siempre de la forma más compacta y utilizando los metacaracteres apropiados):

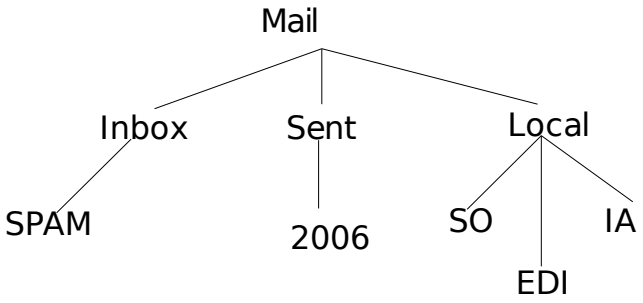
a) Todos los archivos que tengan como tercera letra de su nombre una **a**

b) Todos los archivos que no comiencen por una consonante

c) Todos los archivos que que contengan un número por **da, df**

d) Todos los archivos que comiencen por **b** y acaben en un número

2.- Construye la siguiente estructura de directorios




y los siguientes ficheros de texto con contenido de tu invención.

| *Situado en el directorio* | *Nombre del fichero* |
|---|---|
| SPAM | banco.eml<br>compraTelefono.txt |
| Mail | notas.eml |
| SO | noticiasPracticas.eml |
| EDI | febrero2007_notas.eml |
| IA | anotaciones.txt |
| LOCAL | listaAprobados.txt |
| 2006 | telefonosClase.eml |

Situándote en el directorio *Mail*:

a) Utiliza la orden `find` para encontrar todos los ficheros que acaben en **.txt**

b) Utiliza la orden `find` para encontrar todos los ficheros cuyo nombre contenga la cadena **"fono"**

c) Utiliza la orden `find` para encontrar todos los ficheros cuyo nombre contenga la cadena **"not"**

d) Utiliza la orden `find` y su opción -exec para mostrar el contenido de los ficheros cuyo nombre contenga la cadena **"eml"**.

3.- Crea una línea de órdenes que añada el contenido de todos los archivos contenidos en cualquier subdirectorio de *Mail* a un fichero llamado *copiaSeguridad.bak* situado en el directorio Local

4.- Construye una línea de órdenes que dé como resultado el número de archivos y directorios que hay en el directorio de trabajo y, a continuación, imprima la fecha y hora actual. Para ello necesitarás utilizar las órdenes `wc` (cuenta el número de líneas, palabras y caracteres) y `date` (imprime la hora y la fecha actual).

5.- Construya un archivo llamado *agenda.txt* en el directorio *INBOX* y configure sus privilegios de acceso, de tal modo, que el usuario pueda leer y escribir, el grupo pueda leer y los otros no puedan hacer nada.

6.- Sin utilizar un editor de texto gráfico, crea, en el directorio *SO*, un fichero llamado *notas.txt* con 10 líneas, cada una de las cuales tenga este formato:

Nombre DNI Calificación

7.- Visualice la primera y las siete últimas líneas del archivo *notas.txt*

8.- Crea una línea de órdenes para eliminar todas las líneas duplicadas, menos una, del archivo *telefonosClase.eml*

9.- Escribe una línea de órdenes para visualizar las líneas del archivo *notas.txt* que contengan alumnos con la cadena **zap** en su nombre

10.- Busca archivos cuyo nombre contenga la cadena **"86"** a partir del directorio raíz /. Redirecciona la salida y el error de forma adecuada para enterarte de la salida.

11.- Crea un fichero **.tgz** y otro **.tbz** con la jerarquía de directorios de *Mail*.

12.- Escribe las líneas de órdenes que lleven a cabo las tareas siguientes:

a) Contar el número de caracteres y líneas que hay en el archivo *notas.txt* y mostrar el resultado por pantalla (no hay que mostrar el nombre del archivo)

b) Contar el número de palabras y líneas que hay en el resultado de la orden **ls -1** y mostrar el resultado por pantalla

c) Los mismo que el apartado (b), pero redirigiendo la salida a un archivo llamado *estadísticas.data1*

13.- Construye una tabla ASCII en el archivo *asiapac.temps*, poniendo en cada línea los datos de una ciudad por el orden siguiente: nombre de la ciudad, temperatura máxima y temperatura mínima. La línea siguiente es un ejemplo:

Tokyo 28 10

La información de las ciudades es la siguiente: (Kuala, 28, 13) (Karechi, 30, 12) (Tokyo, 28, 10) (Lahore, 20, 11) (Manila, 28, 17) (NuevaDelhi, 25, 15) , (Jakarta, 30 11), (Pekin, 9, 3) y (HongKong 9, -1)

Escribe la órdenes necesarias para realizar las operaciones siguientes

a) Ordenar la tabla por nombres de ciudades y guardarla en el fichero *ordenNombre.txt*

b) Ordenar la tabla por temperaturas mínimas y guardarla en el fichero *ordenMaxima.txt*

c) Ordenar la tabla empleando la temperatura máxima como clave primaria y la temperatura mínima como clave secundaria y añadirla al fichero *ordenNombre.txt*

14.- Crea un orden que muestre por pantalla todos los archivos ocultos que hay en tu directorio **home.**

15.- Crea una orden que muestre por pantalla el nombre de cinco editores de texto que haya en el sistema.

16.- Inicia un programa interactivo (por ejemplo, `xclock`), y suspéndalo. Revisa la lista de procesos y comprueba que el nuevo proceso está y que su estado sea suspendido. Después reinícialo en segundo plano y finalmente elimínalo.

17.- ¿Que hará el comando *touch *.c*?

# 3.- COMANDOS SHELL II

## 3.1.- Historia de órdenes

Las órdenes que se escriben en el terminal se van guardando en un archivo histórico, es lo que se llama historia de órdenes. Ésta va almacenando las órdenes con sus respectivos parámetros que se le van dando a la shell para que ejecute. El sentido de mantener la historia es facilitar y hacer más rápida la repetición de alguna orden que ya habíamos ejecutado y poder reutilizar sus argumentos.

En este apartado, vamos a estudiar como conseguir que el shell mantenga una historia de las órdenes dadas, y cómo volver a llamar a órdenes dadas con anterioridad, modificarlas y reutilizarlas.

### Asignando la variable historia

La variable $HISTSIZE contiene el número de órdenes que se van a almacenar en la historia antes de descartar la orden más antigua almacenada. El valor predeterminado para esta variable es de 500, si bien podemos cambiarlo asignándole otro valor a la variable.

Para consultar el valor de las variables deberemos teclear **set.**

```
$bash$  set
argv            ()
PWD             /tmp/
HISTSIZE        500
USER            mluque
SHELL           /bin/bash
```

...

La primera columna refleja el nombre de la variable y la segunda su valor.

### Viendo la historia

La orden **history** muestra una lista de las órdenes de la historia de órdenes; cada orden va

precedida de un número que indica la ocurrencia de la misma en la historia, empezando por 1 para la orden más antigua. En nuestro caso

```
$bash$  history
1 echo $SHELL
2 ls
3 ps ax
4 find . -name *.txt
5 history
```

**history [opciones]**

**N : mostrar únicamente las N últimas líneas del histórico**

**-c: borrar la lista histórica**

```
Prueba el comando history. ¿Qué ocurre si lo lanzas en diferentes
terminales
```

### Almacenando la historia de órdenes

Cuando abandonamos una shell (bien utilizando el comando **exit** o cerrando el terminal), las órdenes que se hayan ejecutado (listado histórico) se guardan en el archivo histórico. La ruta de este archivo se encuentra almacenada en la variable $HISTFILE; el archivo histórico predeterminado suele ser *~/.bash_history.*

### Repitiendo órdenes con el editor de órdenes

En el bash, podemos utilizar un editor de órdenes para recuperar y ejecutar órdenes. Con las flechas arriba y abajo del teclado, podemos movernos hacia adelante y hacia atrás en la historia de órdenes. Cada orden va apareciendo en pantalla según la recuperemos con el cursor situado al final. Podemos pulsar **<RETURN>** para ejecutarla **<CONTROL>-C** para cancelarla. También, podemos editar las órdenes para reciclarlas. Utilizando las flechas izquierda y derecha podemos movernos por la línea (orden recuperada) y borrar o insertar para modificar, según nuestra necesidad, la orden recuperada.

### Expansión histórica: repitiendo órdenes con el signo bang (!)

Otra forma para reutilizar órdenes es usar el signo de admiración, denominado *bang*. Las formas para recuperar una orden son:

| | |
|---|---|
| **!numero-orden** | **#Invoca una orden conocida su posición en la historia** |
| **!!** | **#Invoca la última orden ejecutada** |
| **!-n** | **#Invoca la orden situada *n* lugares atrás en la historia** |
| **!cadena_texto** | **#Invoca la orden que comienza por *cadena_texto*** |
| **!?cadena_texto[?]** | **#Invoca la orden que contiene *cadena_texto*** |

Supongamos que estamos en la orden 6, diferentes formas de invocar la orden 4 (*find . -name *.txt*) son:

```
$bash$ !4       # Damos el valor absoluto la posición en la historia
$bash$ !-2      # Damos la posición relativa en la historia
$bash$ !fi      # Ejecutamos la orden que comienza por "fi"
$bash$ !?in     # Ejecutamos la orden que contiene la subcadena "in"
```

En el último ejemplo, indicamos que queremos la orden más reciente que contiene el texto *"in"* en cualquier parte de la cadena de la orden.

Podremos seguir añadiendo texto a una orden anterior si finalizamos con una `?`. Del ejemplo anterior, reutilizamos la escritura de la orden 2 (`ls`) para modificar las opciones `!?ls? -l`

```
ls -l
```

### Modificación de uso común para la expansión del listado histórico

Las palabras de la orden seleccionada se numeran de tal modo que la primera (nombre de la orden) tiene el número 0, y las demás palabras se numeran consecutivamente respecto a la anterior. Se puede emplear un indicador de palabra para identificar la palabra o palabras deseadas dentro de la orden o evento seleccionado. El signo dos puntos (:) se inserta entre el indicador del suceso y el indicador de palabra.

```
! orden [:palabra]
```

La Tabla 3.1 enumera alguno de los indicadores de palabra más frecuentes. Como ejemplo utilizamos la orden *gcc lab1.c -o lab1*

Una vez selecciona una orden, y ciertas palabra dentro de esa orden, se pueden aplicar una o más operaciones a las palabras seleccionadas mediante el uso de una sucesión de operadores llamados modificadores. Lo modificadores van precedidos del signo dos puntos (:)

```
! orden[:palabra][:modificador]
```

| ***Indicador*** | ***Significado*** | ***Ejemplo*** |
|---|---|---|
| o | Palabra 0 (orden) | *o* La orden *gcc* |
| N | La N-ésima palabra | 2 La tercera palabra (*-o*) |
| ^ | La palabra 1 de la orden | ^ La palabra será *lab1.c* |
| $ | La última palabra de la orden | $ La palabra *lab1* |
| * | Toda las palabras salvo el nombre de la orden | * Las palabras *lab1.c*, *-o* y *lab1* |
| N1-N2 | El intervalo de palabras desde el número *N1* al *N2* | 2-3 Las palabras *-o* y *lab1* |

*Tabla 3.1: Indicadores de palabra*

Los modificadores cambian la forma en la que el evento actual es tratado. Algunos de los modificadores más útiles aparecen en la Tabla 3.2 (el resto pueden consultarse en el manual en línea sobre *history*).

| ***Modificador*** | ***Descripción*** |
|---|---|
| p | Imprime el resultado de una orden sin ejecutarla |
| r | Elimina la raíz del nombre de archivo |
| e | Elimina la extensión del nombre de archivo (parte que sigue al punto) |
| h | Se queda con la cabecera de la ruta (todo menos el último componente) |
| t | Se queda con el final de la ruta (último componente) |

*Tabla 3.2: Modificadores de la referencias de la historia*

Vamos a ver algunos ejemplos, :p es útil cuando no estamos seguros si la referencia que vamos a hacer es la que deseamos. Con él, veremos la orden sin ejecutarla, y una vez que estemos seguros sólo tenemos que hacer !!

```
$ !ech:p
echo $SHELL
$ !!
```

## 3.2.- Autocompletar con el tabulador

Una tecla muy conocida en Bash es el tabulador, que nos ayuda a terminar de rellenar una orden con el nombre de un comando, de una variable, de un fichero o directorio, o con el nombre de una función Bash. Para ello, se siguen las siguientes reglas cuando se pulsa el tabulador:

1. Si no hay nada que empiece por el texto de la palabra que precede al cursor se produce un pitido que informa del problema.

2. Si hay un comando (en el PATH), una variable (siempre precedida por $), un nombre de fichero o función Bash que comienza por el texto escrito, Bash completa la palabra.

3. Si hay un directorio que comienza por el nombre escrito, Bash completa el nombre de directorio seguido por una barra de separación de nombres de directorios, /.

4. Si hay más de una forma de completar la palabra, el shell completa lo más que puede y emite un pitido informando de que no la pudo terminar de completar.

5. Cuando Bash no puede completar una cadena (por haber varias posibles que empiezan igual), podemos pulsar dos veces el tabulador y se nos mostrará una lista con las posibles cadenas.

## 3.3.- Metacaracteres

Todos los shells poseen un grupo de caracteres que en diferentes contextos tienen diferentes significados, los **metacaracteres**. Éstos juegan un papel importante cuando el shell está analizando la línea de órdenes, antes de ejecutarla. Vamos a verlos agrupados según la función que realizan.

### 3.3.1.- Metacaracteres sintácticos

Sirven para combinar varias órdenes de LINUX y construir una única orden lógica. Suministran una forma de ejecución condicional basada en el resultado de la orden anterior. La Tabla 3.3 muestra estos caracteres y da una descripción de su función (ver práctica 1XXXXX).

| ***Metacarácter*** | ***Descripción de la función*** |
|---|---|
| ; | Separador entre órdenes que se ejecutan secuencialmente |
| \| | Separación entre órdenes que forman parte de un cauce (pipeline). La salida de la orden de la izquierda del separador es la entrada de la orden de la derecha del separador. |
| () | Se usan para aislar órdenes separadas por ; ó \|. Las órdenes dentro de los paréntesis, ejecutadas en su propio shell, son tratadas como una única orden. Incluir un cauce dentro de paréntesis, nos permite a su vez incluirlo en otros cauces. |
| & | Indicador de trabajo en segundo plano (*background*). Indica al shell que debe ejecutar el trabajo en segundo plano. |
| \|\| | Separador entre órdenes, donde la orden que sigue al \|\| sólo se ejecuta si la orden precedente falla. |
| && | Separador entre órdenes, en la que la orden que sigue al && se ejecuta sólo si la orden precedente tiene éxito. |

*Tabla 3.3: Metacaracteres sintácticos*

### 3.3.2.- Metacaracteres de nombre de archivos

Éstos se utilizan para formar patrones de igualación para la sustitución de nombres de archivos con el fin de poder referenciar de forma abreviada una serie de archivos cuyos nombres siguen un patrón. La Tabla 3.4 muestra estos metacaracteres. Para ver una descripción más detallada ver la práctica XXXX.

| ***Metacarácter*** | ***Descripción de la función*** |
|---|---|
| ? | Comodín a cualquier carácter simple |
| * | Iguala cualquier secuencia de cero o más caracteres |
| [] | Designa un carácter o rango de caracteres que, como una clase, son |

| *Metacarácter* | *Descripción de la función* |
|---|---|
| | igualados por un simple carácter. Para indicar un rango, mostramos el primer y el último carácter separados por un guión (-). Con el símbolo **!** indicamos **negación**. |
| { } | Abreviar conjuntos de palabras que comparten partes comunes |
| ~ | Se usa para abreviar el camino absoluto (path) del directorio home |

*Tabla 3.4: Metacaracteres de nombre de archivos*

### 3.3.3.- Metacaracteres de citación

Los metacaracteres de citación, Tabla 3.5, se utilizan para controlar cuándo deben protegerse el resto de metacaracteres de la expansión o interpretación del shell. Denominamos **expansión** al procedimiento por el cual el shell sustituye la ocurrencia de un carácter especial por una lista.

| *Metacarácter* | *Descripción de la función* |
|---|---|
| \ | Evita que el carácter que le sigue sea interpretado como un metacarácter por el shell. |
| " | Evita que la cadena de caracteres encerrada entre dobles comillas sea interpretada como metacaracteres. |
| ' | Evita que la cadena de caracteres encerrada entre comillas simples sea interpretada como órdenes o metacaracteres. |
| ` | La cadena de caracteres encerrada entre acentos se interpreta como una orden y se ejecuta |

*Tabla 3.5: Metacaracteres de citación*

#### Eludir metacaracteres con \

Cuando el shell procesa una orden, lo primero que hace es sustituir los metacaracteres por sus respectivos valores para que en la ejecución aparezcan los valores reales, no los metacaracteres. "Eludir" metacaracteres significa evitar su interpretación por el shell, de forma que éstos permanezcan en la línea de órdenes para ser procesados en la ejecución.

Supongamos que queremos crear unos archivos con nombres *abc;1* y *Trabajo Para Casa*. Si escribiésemos estos nombres como tal, el shell interpretará el ; y el espacio en blanco como separadores, respectivamente (pruébalo para ver que dice).

Para evitarlo podemos escribir:

```
$bash$ mv abc1 abc\;1
$bash$ mv TrabajoParaCasa Trabajo\ Para\ Casa
```

### Protegiendo metacaracteres con “ ”

La barra invertida sólo protege al carácter que la sigue inmediatamente, por lo que no es cómodo para proteger cadenas largas. En estos casos, podemos proteger la cadena completa usando las comillas dobles. Éstas desactivan el significado especial de los caracteres entre ellas, salvo los de **! evento, $var y `cmd`** que representan la sustitución de la historia, variables y ejecución de órdenes, respectivamente. Ejemplo:

```
$bash$  echo " **** Hola $USER "
**** Hola Pepito
$bash$
```

Observad como los metacaracteres * no se han expandido, pero la variable `$USER` si ha sido sustituida por su valor.

### Protegiendo con ' ' órdenes, variables y metacaracteres

Las comillas simples son parecidas a las comillas dobles, desactivan el significado especial de algunos caracteres salvo la expansión de la historia de órdenes (**!evento**).

```
$bash$  texto="mi variable"
$bash$  dentro="valor de $texto"
$bash$  echo $dentro
valor de  mi va riable
$bash$  set dentro='valor de $texto'
$bash$  echo $dentro
valor de  $texto
```

Tened presente que la acotación de cadenas es compleja, especialmente las cadenas que a su vez contienen metacaracteres de citación. Recordad que los metacaracteres de citación no se utilizan igual que en la escritura normal.

### 3.3.4.- Metacaracteres de entrada/salida o de dirección

Los metacaracteres de entrada/salida son otra de las características distintivas de LINUX. Con ellos, podemos redireccionar (tomar) la entrada de una orden desde un archivo en lugar del teclado, redireccionar (llevar) la salida estándar a un archivo en lugar de a la pantalla, o encauzar su entrada/salida a/desde otra orden. También, podemos mezclar los flujos de información de la salida y el error estándares para que salgan juntos por la salida estándar. Vemos un resumen en la Tabla 3.6.

| ***Metacarácter*** | ***Descripción de la función*** |
|---|---|
| < *nombre* | Redirecciona la entrada de una orden para leer del archivo *nombre*. |
| >*nombre* | Redirecciona la salida de una orden para escribir en el archivo *nombre*. Si nombre existe, lo sobreescribe. |
| &> *nombre* | La salida de *stderr* se combina con *stdout*, y se escriben en *nombre*. |
| 2> *nombre* | La salida de *stderr* se escribe en nombre |
| >> *nombre* | La salida de la orden se añade al final del archivo *nombre*. |
| &>> *nombre* | Añade la salida de *stderr*, combinada con *stdout* al final de *nombre*. |
| \| | Crea un cauce entre dos órdenes. La salida estándar de la orden a la izquierda de símbolo se conecta a la entrada estándar de la orden de la derecha del signo. |

*Tabla 3.6: Metacaracteres de entrada/salida o de dirección*

### 3.3.5.- Metacaracteres de expansión/sustitución

Los metacaracteres de expansión/sustitución actúan como indicadores especiales para el Bash. La Tabla 3.7 muestra los cuatro metacaracteres de este tipo.

| *Metacarácter* | *Descripción de la función* |
|---|---|
| $ | Indica la sustitución de variables. Una palabra precedida de $ se interpreta como variable, y $palabra se sustituye por su valor. |
| ! | Indicador de sustitución de historia (historia de órdenes en bash). |
| : | Precede a los modificadores de sustitución. |

*Tabla 3.7: Metacaracteres de expansión/sustitución*

Estos modificadores nos han ido apareciendo a lo largo de los guiones, veamos que significan: **"$"** sirve para indicar al shell que sustituya una variable por su valor; **"!"** es el indicador de la historia de órdenes; y los **":"** separan la orden de los designadores de palabra en la historia de órdenes.

## 3.4.- Los alias

### 3.4.1.- Definición y eliminación de alias

Un **alias** es un nombre más corto (seudónimo) para un comando que usamos muy a menudo. El mecanismo de alias permite adaptar el comportamiento de las órdenes LINUX cuando se ejecutan. Es similar a la capacidad de definición de macros en otros lenguajes. En Bash, la orden **alias**, se encarga de crear un seudónimo para una orden.

**alias [*nombre_alias*]=[*definición_orden*]**

Asocia un `nombre_alias` con la `definición_orden`.

```
$bash$  alias dir="ls -l"       # asigna a ls -l el nombre de dir
```

Cuando veamos que ya no nos es útil la definición de un alias, podemos borrarla sin salir del sistema. La sintaxis es

**unalias *nombre_alias***

donde *nombre_alias* es el nombre del alias que se va a borrar.

### 3.4.2.- Listar la definición de los alias existentes

En nuestro entorno de prácticas, el sistema define algunos alias de uso común. Se pueden parecer bastante a:

```
$bash$  alias       # lista los alias existentes y su definición
  l.          ls d . [a-zA-z]* --color=tty
  ll          ls l --color=tty
  ls          ls color=tty
  which       alias |/usr/bin/which tty-only read-alias show-do

$bash$  alias    ll    # muestra la definición del alias ll
    ll          ls l --color=tt
```

### 3.4.3.- Renombrar o redefinir una orden existente.

Esto permite ahorrar escritura de órdenes o que nuestro sistema se parezca a otro sistema operativo.

```
$bash$  alias rename=mv          #renombra mv al nombre rename
$bash$  alias ls='ls -l'         #abrevia la escritura de ls -l
```

Cuando redefinimos una orden con un alias, podemos seguir utilizando la orden en su forma original anteponiendo al nombre un \, o bien invocándola con su camino completo. Por ejemplo, definido el alias anterior, para invocar **ls** en su forma normal:

```
$bash$  \ls
$bash$  /bin/ls
```

### 3.4.4.- Crear una nueva orden

Permite abreviar la escritura de las órdenes que usamos frecuentemente:

```
$bash$  alias hp='history|more'   # hp muestra historia paginada
$bash$  alias ht='history|tail'   # ht muestra el final del archivo
                                    de historia
```

**¡Precauciones!**

1. A veces, los alias pueden entrar en conflicto con programas del shell, especialmente si se usan para la redefinición de órdenes. Un programa shell podría no funcionar correctamente porque espera el comportamiento por defecto de una orden.

2. Define siempre los alias multipalabra entre comillas simples.

3. Al crear un alias, debemos tener la precaución de no llamarlos *alia*s, ni crear alias que se referencien a ellos mismos para no crear un bucle en las definiciones de éstos. Ahora bien, sí podemos crear alias que referencien a otros alias.

4. Cuando definimos un alias, éste sólo estará disponible en el terminal en el que se creó y, además, cuando se apague el ordenador se perderá. Si queremos que un alias esté disponible en nuestro sistema siempre que arranquemos, podemos definirlo dentro del fichero **~/.bashrc**. Este fichero se ejecuta siempre que se arranca Bash.

## 3.5.- Variables

El shell, como lenguaje de programación que es, necesita poder definir y usar variables. Una **variable** es una posición de la memoria principal a la cual se le asigna un nombre. Esto nos permite aludir a esta posición de memoria empleando un nombre, en lugar de utilizar su dirección. El nombre de una variable de shell puede estar formado por dígitos, letras y el guión inferior (_), siempre y cuando el primer carácter sera una letra o un guión inferior. La mayúsculas y minúsculas se consideran distintas y la longitud del nombre no está limitada. Como la memoria principal es un tipo de almacenamiento que admite lectura y escritura, se puede leer el valor de la variable, y también se le puede asignar un nuevo valor. Para la shell de Bash, las variables son siempre cadenas de caracteres, aunque se almacenen números en ellas.

### 3.5.1.- Variables de entorno y variables locales

Las variables del shell pueden pertenecer a dos tipos: variables de entorno y variables definidas por el usuario. Las dos pueden albergar valores, cadenas o números, y pueden utilizarse tanto en programas shell como interactivamente. La diferencia entre ambas radica en su alcance o ámbito.

Las **variables de entorno** sirven para personalizar el entorno en el que se ejecuta la shell y para ejecutar correctamente las órdenes del shell. Cuando damos al shell una orden interactiva, Linux inicia un nuevo proceso para ejecutar la orden. El nuevo proceso hereda algunos atributos del entorno en el que se creó. Estos atributos se pasan de unos procesos a otros por medio de las variables de entorno, también denominadas variables globales ($HOME, $SHELL,$USER, ...).

Por contra, las **variables definidas por el usuario o locales** no se pasan de unos procesos a otros y sus contenidos se descartan cuando salimos del shell en el cual se definieron.

El shell trata por igual a todas las variables: las identifica por un prefijo especial, el **signo dolar ($)**, que indica que la palabra que sigue es el nombre de una variable. Por otra parte, existe una convención para la designación de variables: se usan letras mayúsculas para las variables de entorno, y minúsculas para las variables locales. Como Linux es sensible al tipo, es importante el uso de mayúsculas o minúsculas. No son lo mismo $COSA, que $cosa, que $CoSa, etc.

Para visualizar los valores de todas las variables del sistema se utiliza el comando **set** sin argumentos.

### 3.5.2.- Creación de variables

Si queremos darle valor a una variable

```
var=Valor       #asigna Valor a la variable var
var=            #asigna un valor nulo a la variable var
```

Donde ***var*** es el nombre de la variable, compuesto por letras, dígitos y/o subrayado (_). Si ***var*** no existe, se crea, y si no, se cambia el valor de la variable. **No pueden existir espacios antes o después del símbolo =**.

Sin el argumento *valor,* a la variable se le asigna un valor nulo. Si el argumento es ***valor***, entonces se asigna ***valor*** a ***var***. El valor puede ser una palabra o (lista_de_palabras).

Las variables multipalabra son matrices cuyos elementos pueden ser accesibles individualmente, indicando su posición dentro de la matriz con [ ] (comenzando con el elemento número 0).

```
$bash$  varprueba=prueba123
$bash$  matrizpru=(prueba 1 2 3)
$bash$  varvacia
$bash$  matrizpru[3]=dos
$bash$  nombre='Fernando López'
$bash$  nombre2=Fernando\ López
```

Para obtener el valor de una variable debemos de precederla por **$** y utilizar la orden **echo**. La orden **echo** sirve, además, para mostrar una línea de texto por pantalla.

```
echo $variable
echo frase
```

```
$bash$  echo $var prueba
        prueba123
$bash$  echo $no mbre2
        Fernando López
$bash$  echo ${m atrizpr u[0]}
        prueba
$bash$  echo ${m atrizpr u[3]}
        3
$bash$  echo Es to es una  frase
        Esto es una frase
$bash$  echo  El dir ect orio con tiene  `ls  | w c -l` f icher os
        El directorio contiene 12 ficheros.
```

Observad que las comillas o el carácter de escape no forman parte del valor de la variable una vez que se realiza la asignación.

En el último ejemplo, utilizamos el acento para indicarle a la shell que el comando queremos que lo ejecute y no que lo trate como una cadena más.

Para eliminar una variable de entorno podemos usar el comando **unset**. Por ejemplo:

```
$ unset NOMBRE
```

### 3.5.3.- Personalizar el prompt

El *prompt* es el indicador de que la línea de comandos está libre para aceptar una nueva orden. En las aulas de prácticas, el *prompt* por defecto es **$bash$**, sin embargo podemos personalizarlo para que ponga lo que nosotros queramos.

La cadena que se muestra en el *prompt* se almacena en la variable de entorno **PS1**, por lo tanto, bastará con cambiar el valor de esa variable, para cambiar el *prompt* de nuestro sistema.

La Tabla 3.8 muestra algunas opciones que podemos usar para personalizar el *prompt*, veamos algunas de ellas.

| ***Opción*** | ***Descripción*** |
|---|---|
| \A | La hora en formato `HH:MM` |
| \d | La fecha en formato `semana mes dia` |
| \H | Hostname |
| \h | El nombre de la máquina hasta el primer punto |
| \j | Número de jobs hijos del shell |
| \n | Retorno de carro y nueva línea |
| \r | Retorno de carro |
| \s | Nombre del shell |
| \t | Hora en formato `HH:MM:SS` con 12 horas |
| \T | Hora en el formato anterior pero con 24 horas |
| \u | Nombre de usuario |
| \v | Versión de Bash |
| \w | Ruta del directorio actual |
| \W | Nombre del directorio actual (sin ruta) |
| \# | Número de comandos ejecutado |

| *Opción* | *Descripción* |
|---|---|
| | |
| \$ | # si somos root, o $ en caso contrario |
| \nnn | Código de carácter a mostrar en octal |
| \\ | La barra hacía atrás |

*Tabla 3.8: Opciones de personalización del prompt*

### Conocer el directorio de trabajo

Al prompt podemos añadirle muchas opciones que nos ayuden a trabajar; quizás la más usada es **\w** que muestra el directorio donde estamos situados en cada momento.

```
$bash$  PS1='\w->'
~->
```

Donde **~** significa que estamos en el directorio **home**.

### Saber si somos *root* o usuario

Es muy típico querer saber si somos *root* o cualquier otro usuario de la máquina, para lo cual se usa la opción **\$** que muestra una **#** si somos *root,* o un **$** en cualquier otro caso.

```
~-> PS1='\w\$'
~$
```

### Saber en que máquina estamos

Muchas veces el *prompt* nos ayuda a saber en que máquina estamos situados, esto es útil, por ejemplo, cuando tenemos la costumbre de hacer muchos **telnet** o **ssh** a otras máquinas. En este caso, podemos usar la opción **\H** que nos da el nombre de la máquina donde estamos situados, o la opción **\h** que nos da el nombre de la máquina sólo hasta el primer punto:

```
~$ PS1='\h\w\$'
rabinf1 41~$ PS1='\H\w\$'
rabinf1 41.uco. es~$
```

### Mostrar el nombre del usuario

La opción **\u** nos muestra el usuario con el que estamos conectados. Este es un *prompt* bastante útil:

```
rabinf1 41.uco. es~$  PS1='\u@h:\w\$
mluq ue@r abinf1 41:~$
```

Aparte del prompt básico (su valor se obtiene de la variable **PS1**) que nos indica si la línea de comandos está preparada para recibir una nueva orden, existen otros prompts en el sistema. Sus cadenas viene determinadas por los valores de las variables **PS2**, **PS3** y **PS4**.

Así, **PS2** es el *prompt* secundario y por defecto vale >. Se usa cuando escribimos un comando inacabado. Por ejemplo:

```
$bash$  echo En  un lugar de  la Manch a de  cu yo nombre no  \
> q uier o acor dar me  no a  mucho tie mpo  vivía  un \
> hidal go cab aller o.
En un lugar de la Mancha de cuyo nombre no quiero acordar
no a mucho tiempo vivía un hidalgo caballero.
```

**PS3** y **PS4** son *prompts* de programación y depuración.

## 3.5.4.- Variables de entorno internas

Existen una serie de variables de entorno, las cuales se resumen en la Tabla 3.9, que no las fijamos nosotros sino que las fija el shell, y cuyo valor es de sólo lectura.

| *Variable* | *Descripción* |
|---|---|
| SHELL | Path del intérprete de órdenes que estamos usando |
| LINES | Líneas del terminal en caracteres |
| COLUMNS | Columnas del terminal en caracteres |
| HOME | Directorio *home* del usuario |
| PWD | Directorio actual |
| OLDPWD | Anterior directorio actua |

| *Variable* | *Descripción* |
|---|---|
| USER | Nombre de usuario con el que estamos conectados |

*Tabla 3.9: Variables de entorno internas*

Algunas variables se actualizan dinámicamente, por ejemplo, LINES y COLUMNS almacenan en todo momento el número de filas y columnas de la pantalla, y las usan algunos programas como **vi** o **emacs** para modificar su apariencia.

### 3.5.5.- Exportar variables

Cuando definimos una variable, esa variable no es conocida automáticamente por los procesos hijos que lanza la sesión Bash, a no ser que esta se ponga explícitamente a su disposición, con el comando **export**.

Por ejemplo:

```
$bash$  EDITOR=/sw/bin/joe
$bash$  export EDITOR
```

Podemos obtener un listado de las variables exportadas usando el comando **env** o el comando **export** sin argumentos. Para obtener las variables tanto exportadas como no exportadas debemos usar el comando **set** sin argumentos.

## 3.6.- Scripts

En el Capítulo 1(XXX) comentamos que los intérpretes de órdenes o shells eran lenguajes de programación y que nuestro objetivo sería aprender este lenguaje, en concreto, el correspondiente al Bash. Bien, comencemos a conocer este lenguaje.

Cuando programamos en C, creamos un fichero con extensión .c donde escribimos el código del programa y a continuación lo compilamos para obtener el programa ejecutable. De igual forma, en Bash, la forma más común de ejecutar programas es crear ficheros (en este caso con extensión **.sh**) que se llaman **archivos interpretados** o ***scripts*** y que contienen una colección de órdenes Bash que serán ejecutadas por la shell.

Para implementar este mecanismo, los diseñadores de LINUX sobrecargaron la funcionalidad de la llamada al sistema que permite ejecutar un programa, la llamada ***exec***. De esta forma, cuando mandamos a ejecutar un programa, la función ***exec*** comprueba si el archivo es binario o no. Si es un binario, resultado de una compilación, lo ejecuta en la forma usual. Pero si el archivo no es binario, la función ***exec*** está diseñada para suponer que lo que contiene el archivo son órdenes para un intérprete de órdenes, por lo que ejecuta un shell y le pasa el contenido del archivo para que lo interprete. De aquí el nombre de archivos interpretados. Ahora bien, el shell creado por ***exec*** por defecto es un Bourne shell (bash) ¿Qué ocurre si queremos ejecutar un programa escrito en otro lenguaje shell?

En este caso se utiliza el siguiente convenio: la primera línea del archivo interpretado debe contener el programa intérprete al que se le debe pasar el resto del contenido del archivo. El convenio para indicar el intérprete es:

```
#!path_absoluto_del_interprete [argumentos]
```

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

De esta manera, podemos ejecutar programas shell de una manera sencilla. **Esta indicación debe ser siempre la primera línea del *script* y no puede tener blancos.**

### 3.6.1.- Primeros pasos con *scripts*

Para crear un *script* usamos cualquier editor de texto y lo guardamos en un fichero. Una vez creado el *script* existen dos formas de ejecutarlo:

1. La primera forma implica dar al fichero permiso de ejecución (con el comando **chmod**). Una vez dado este permiso, podremos ejecutarlo simplemente poniendo ***./script***. De hecho, muchos comandos comunes (p.e. spell o man) están implementados como *scripts* Bash y no como programas C ejecutables.

   ```
   ./<fichero script>
   ```

2. La segunda consiste en ejecutarlo con el comando **source**, el cual carga el fichero en la

memoria de Bash y lo ejecuta.

```
source <fichero script>
```

Una diferencia entre estas dos formas de ejecutar un *script* es que si lo ejecutamos con **source**, éste se ejecuta dentro del proceso del shell, mientras que si lo ejecutamos como un fichero ejecutable se ejecuta en un subshell. Esto implica que sólo las variables de entorno exportadas son conocidas por el nuevo subshell que ejecuta el programa; y que las variables que sean creadas en la ejecución del programa se conocerán a su finalización.

### Nuestro primer *script*: Hola mundo

Para crear un *script* basta con abrir un fichero con cualquier editor de texto (nedit, pico, vi, gedit) y, a continuación, indicar las acciones que queramos que realice nuestro *script*.

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

Este *script* (*hello.sh*) tiene sólo dos líneas. La primera le indica al sistema qué programa usará para ejecutar el fichero; y la segunda es la única acción realizada por este script, que imprime 'Hola mundo' en la terminal. **Si sale algo como *./hello.sh: Comando desconocido*, probablemente la primera línea, '#!/bin/bash', esté mal escrita. Ejecutad whereis bash para saber donde se encuentra el comando bash y aseguraos de que las barras son las adecuadas y que no habéis dejado espacios.**

### Un *script* de copia de seguridad muy simple

El siguiente *script* (*copiaSegurida.sh*), crea una copia de seguridad del directorio **home**.

```
#!/bin/bash
tar -xvf /var/copia.tar /home/in1lurom
gunzip /var/copia.tar
```

Este script, en vez de imprimir un mensaje en la terminal, empaqueta y comprime el directorio **home** del usuario.

### Comentarios en los *scripts*

En un *script* todo lo que vaya después del símbolo **#** y hasta el siguiente carácter de nueva línea se toma como comentario y no se ejecuta.

```
  echo Hola todos       # comentario hasta fin de línea
```

sólo imprime "Hola todos".

```
  # cat /etc/passwd
```

no ejecuta nada, pues el símbolo **#** convierte toda la línea en comentario.

Los *scripts* suelen encabezarse (**después** de la línea #!/bin/bash) con comentarios que indican el nombre del archivo y lo que hace el *script*. También se suelen colocar comentarios de documentación en diferentes partes del *script* para mejorar la comprensión y facilitar el mantenimiento. Un caso especial es el uso de **#** en la primera línea para indicar el intérprete con que se ejecutará el *script*. Un *script* con comentarios quedaría así:

```
#!/bin/bash
#misdatos.sh#
# muestra datos propios del usuario que lo invoca
#
echo MIS DATOS.
echo Nombre: $LOGNAME
echo Directorio: $HOME
echo -n Fecha:
date # muestra fecha y hora
echo # línea en blanco para presentación
# fin misdatos.sh
```

Este *script* (*misdatos.sh*) muestra por pantalla una serie de datos sobre el usuario. Para ello, utiliza variables de entorno.

## 3.6.2.- Variable en los scripts

Igual que cuando programamos en C, dentro de los *scripts* se pueden utilizar variables para clarificar o facilitar la compresión del programa. Si no se especifica otra cosa, las variables Bash

son de tipo cadena, por lo que pueden contener un número (representado como una cadena), un carácter o una cadena de caracteres. Su creación, consulta y borrado es igual a la vista en el apartado 5; no es necesario declararla, se creará sólo con asignarle un valor a su referencia.

```
#!/bin/bash
#Ejemplo 1 de variables
cad=”Hola Mundo”
echo $cad
```

El *script* anterior (*variables.sh*) crea una variable *cad* con el valor “Hola mundo” y a continuación la muestra por pantalla.

```
#!/bin/bash
#Ejemplo 2
nombre="Fulanito de Copas"
echo "$nombre (asignado entre comillas)"
echo Si lo asigno sin comillas pasa esto:
nombre=Fulanito de Copas
```

El *script* anterior daría como resultado lo siguiente

```
$bash$  ./ejem plo2
Fulanito de Copas (asignado entre comillas)
Si lo asigno sin comillas pasa esto:
./demo04.variables: line 7: de: command not found
```

Al asignar una cadena que contiene espacios a una variable, sin utilizar comillas, nos da un error.

Otro ejemplo con variables

```
#!/bin/bash
#Ejemplo 3: asignación de variables
echo -n "Asigno sin comillas a una variable la cadena
demo* y la escribo"
nombre=demo*
echo $nombre
echo -n “Asigno ahora con comillas la misma cadena y escribo
la variable"
nombre=”demo*”
```

```
echo $nombre
```

El *script* anterior da como resultado del primer echo, un listado con todos los archivos que empiezan con *demo*, y como resultado del segundo echo, la cadena *demo**

```
mluq ue@r abinf1 41:/tm p/bash $ ./ej empl o3
Asigno sin comillas a una variable la cadena demo* y la escribo
demo01.hola_mundo demo02.hola_mundo demo03.entorno
demo04.variables demo04.variables~ demo05.variables
Asigno ahora con comillas la misma cadena y escribo la variable
demo*
```

¿Por qué?

### 3.6.3.- Paso de argumentos a los *scripts*

En ocasiones, puede ser útil que nuestros *scripts* reciban algún tipo de argumento (un directorio sobre el que actuar o un tipo de archivo que buscar) al ser ejecutados. Para hacer referencia a estos argumentos, dentro de los *scripts* se emplean una serie de variables que siempre estarán disponibles, los **parámetros posicionales $1, $2, $3, ...,** , siendo **$0** el nombre del propio programa. Reciben este nombre porque se les reconoce por su ubicación, es decir el primer argumento es **$1**, el segundo **$2** y así sucesivamente.

Por ejemplo imaginemos que creamos el *script* siguiente:

```
#!/bin/bash
# Ejemplo 4: script que recibe parámetros y los imprime
echo "El script $0"
echo "Recibe los argumentos $1 $2"
```

Si lo hemos guardado en un fichero llamado *ejemplo4.sh* con el permiso x activado podemos ejecutarlo así:

```
$bash$  ./ejemplo4.sh hola adiós
El script ./ejemplo4.sh
Recibe los argumentos hola adiós
```

**No se puede modificar el valor de las variables posicionales, sólo se pueden leer; si se intenta asignarles un valor se produce un error**.

Si el argumento posicional al que aludimos no se pasa como argumento, entonces tiene el valor nulo.

```
#!/bin/bash
# Ejemplo 5: script que recibe parámetros y los imprime
echo "El script $0"
echo "Recibe los argumentos $1 $2 $3 $4"
```

Si lo hemos guardado en un fichero llamado *ejemplo5.sh* con el permiso x activado podemos ejecutarlo así:

```
$bash$  ./ejemplo5.sh hola adios
El script ./ejemplo5.sh
Recibe los argumentos hola adios
```

### Las variables $*, $@ y $#

La variable de entorno **$#** almacena el número total de argumentos o parámetros recibidos por el *script* sin contar al **$0**. El valor es de tipo cadena de caracteres, pero más adelante veremos como podemos convertir este valor a número para operar con él. La variables **$*** y **$@** contienen, ambas, los valores de todos los argumentos recibidos por el *script*.

Como ejemplo, el *script* del Ejemplo 4 lo vamos a modificar tal como muestra el Ejemplo 6 para que use estas variables para sacar los argumentos recibidos.

```
#!/bin/bash
# Ejemplo 5: script que recibe parámetros y los imprime
echo "El script $0 recibe $# argumentos:" $*
echo "El script $0 recibe $# argumentos:" $@
```

Al ejecutarlo obtenemos:

```
$bash$  ./ejemplo5.sh hola adios
El script ./ejemplo5.sh recibe 2 argumentos: hola adios
```

```
El script ./ejemplo5.sh recibe 2 argumentos: hola adios
```

Aunque cuando no entrecomillamos **$*** o **$@** no existen diferencias entre usar uno y otro, cuando los encerramos entre comillas dobles (“”), podemos cambiar el símbolo que usa **$*** para separar los argumentos cuando los muestra por pantalla. Por defecto, cuando se imprimen los argumentos, estos se separan por un espacio, pero podemos utilizar otro separador indicándolo en la variable de entorno IFS(Internal Field Separator). La variable **$@** siempre usa como separador un espacio y no se puede cambiar. Por ejemplo si hacemos el siguiente *script*:

```
#!/bin/bash
# Ejemplo 7: script que recibe parámetros y los imprime
IFS=','
echo "El script $0 recibe $# argumentos: $*"
echo "El script $0 recibe $# argumentos: $@"
```

Al ejecutarlo obtenemos:

```
$bash$  ejemplo6 hola adios
El script ./recibe recibe 2 argumentos: hola,adios
El script ./recibe recibe 2 argumentos: hola adios
```

Vemos como, en el caso de **$***, los argumentos se separan por el símbolo coma (,) (separador establecido en la variable IFS), mientras que en el caso de $@ se siguen separando por espacios (no le afecta el valor de la variable IFS).

Siempre conviene entrecomillar las variables **$*** y **$@** para evitar que los argumentos que contengan espacios sean mal interpretados.

### Expansión de variables usando llaves

Realmente la forma que usamos para ver acceder al contenido de una variable, **$variable,** es una simplificación de la forma más general **${variable}**. La simplificación se puede usar siempre que no existan ambigüedades. En este apartado veremos cuando se producen las ambigüedades que nos obligan a usar la forma **${variable}**.

La forma **${variable}** se usa siempre que la variable va seguida por una letra, dígito o guión bajo

(_); en caso contrario, podemos usar la forma simplificada **$variable**. Por ejemplo, si queremos escribir nuestro nombre (almacenado en la variable nombre) y nuestro apellido (almacenado en la variable apellido) separados por un guión podríamos pensar en hacer:

```
$bash$  nombre=Fernando
$bash$  apellido=Lopez
$bash$  echo "$nombre_$apellido"
Lopez
```

Pero esto produce una salida incorrecta porque Bash ha intentado buscar la variable **$nombre_** que al no existir la ha expandido por una cadena vacía.

Para solucionarlo podemos usar las llaves:

```
$bash$  echo "${nombre}_$apellido"
Fernando_Lopez
```

Otro lugar donde necesitamos usar llaves es cuando vamos a sacar el décimo parámetro posicional. Si usamos **$10**, Bash lo expandirá por **$1** seguido de un **0**, mientras que si usamos **${10}** Bash nos dará el décimo parámetro posicional

También necesitamos usar las llaves cuando queremos acceder a los elementos de una variable multipalabra. Por ejemplo:

```
$bash$  matriz=(uno  dos  tres)
$bash$  echo  $matriz
   uno
$bash$  echo  $matriz[2]
   uno[2]
$bash$  echo  ${matriz[2]}
   tres
```

## 3.7.- Ejercicios

1.- Después de ejecutar las siguientes órdenes: **i)** *ls* **ii)** *date* **iii)** *cat > pru.txt* **iv)** *wc -l pru.txt* **v)** *ls -l* **vi)** *chmod 644 pru.txt* **vii)** *echo $PATH* **viii)** *cd ..* **ix)** *pwd* **x)** *echo Hola mundo*

- Visualiza las 5 última órdenes ejecutadas
- Muestra el tamaño máximo del listado histórico
- ¿Cuál es tu archivo histórico? ¿Cómo lo has encontrado?

2.- Suponiendo el listado de órdenes del ejercicio anterior y utilizando el signo bang (**!**):

- Muestra la orden que ocupa la posición 7
- Ejecuta, una orden que comience por la cadena *echo*
- Muestra sin ejecutar la última palabra de la última orden ejecutada

3.- Crea una línea de órdenes que indique mediante un mensaje si la ejecución de la orden ***cat prueba.txt*** tuvo éxito o no. La orden deberá redirigir la salida de error de forma adecuada.

4.- Muestra los siguientes mensajes en pantalla haciendo uso de la orden **echo.** (Las indicaciones en negrita y entre <> indica que hay que ejecutar una orden para obtener ese valor)

- El numero de líneas del archivo pru.txt es **<Numero de líneas de pru.txt>**
- El tamaño del fichero pru.txt es **<Tamaño del fichero pru.txt>**
- La variable $DIR almacena el directorio \home\123

5.- Crea un alias llamado **comprimir** que empaquete y comprima un directorio cualquiera.

6.- Crea un alias llamado **buscarC** que busque todos los archivos con extensión **.c** a partir de tu directorio home.

7.- Crea una variable **fecha** cuyo contenido sea la fecha y hora actual. Comprueba su contenido.

8.- Modifica tu prompt para que muestre el siguiente aspecto `u(h):dr` donde:

- `u` es tu nombre de usuario
- `d` es el nombre del directorio en el que te encuentras (ruta completa)
- `r` indica si eres root o usuario
- `h` es la hora actual

9.- Crea un script de nombre ***datos.sh*** que ejecute las siguientes tareas:

- Liste los ficheros del directorio actual, separándolos por una coma (*ver ayuda del ls*)
- Muestre el número de ficheros (archivos y directorios) que hay en el directorio actual
- Muestre el valor de la variable PATH
- Muestre el número de filas y columnas que acepta la pantalla
- Use comentarios dentro del fichero creado para explicar antes de cada orden su utilidad

10.- Ejecuta el script anterior y almacena su salida en un fichero que se llame **informacion.txt**

11.- Crea un script, de nombre ***copia.sh***, que realice

- Una copia del fichero creado en el ejercicio anterior (**informacion.txt**) en el fichero **informacionDuplicada.txt**
- Añada al final del fichero **informacionDuplicada.txt** el contenido ordenado de **informacion.txt**

12.- Realiza un script, con nombre ***infoDir.sh***, que reciba como primer parámetro el camino completo de un directorio y muestre por pantalla: a) Los ficheros que empiezan por una vocal y b) el número total de ficheros y directorios que hay en ese directorio.

13.- Crea un script, con nombre ***mueveFicherosTexto.sh***, que reciba como primer parámetro el nombre de un directorio que no existe y como segundo parámetro el nombre de un directorio que existe. El script deberá crear el primer directorio y mover todos los ficheros que acaben en .txt del directorio pasado como segundo argumento al directorio pasado como primer argumento.

14.- Crea un script, que se llame ***intercambiar.sh***, que acepte dos argumentos. Los dos argumentos se corresponderán con los nombres de dos archivos existentes cuyos contenidos serán intercambiados al ejecutar el script. Por ejemplo, si el archivo *fich1* contiene la palabra hola y el archivo *fich2* contiene la palabra adiós, al ejecutar el script y darle como argumentos los nombres de estos dos archivos, el resultado será que *fich1* contendrá la palabra adiós y *fich2* contendrá la palabra hola.

15.- Crea un script llamado ***copiaSeguridad.sh*** que reciba como primer parámetro el nombre de un directorio y como segundo parámetro el nombre de un fichero. El script deberá empaquetar y comprimir el directorio pasado como primer parámetro en un archivo con el nombre del segundo parámetro.

16.- Crea un script llamado ***crearScript.sh*** al cual se le pasa como argumento el nombre del script a crear. ***CrearScript.h*** creará otro script, cuyo nombre será el argumento recibido, y una vez creado, le dará permisos de ejecución. El segundo script recibirá como argumentos el nombre de un fichero y el nombre de un directorio; creará el directorio y hará una copia del fichero en el directorio recién creado.

# 4.- PROGRAMACIÓN BASH I

## 4.1.- Variables con tipo

Hasta ahora todas las variables que hemos usado son de tipo cadena de caracteres. Aunque en los primeros shells las variables sólo podían contener cadenas de caracteres, después se introdujo la posibilidad de asignar atributos a las variables que indiquen, por ejemplo, que son enteras o de sólo lectura. Para fijar los atributos de las variables, tenemos el comando interno **declare**, el cual tiene la siguiente sintaxis:

```
declare [±opciones] [nombre[=valor]]
```

La Tabla 4.1 describe las opciones que puede recibir este comando. Una peculiaridad de este comando es que para activar un atributo se precede la opción por un guión -, con lo que para desactivar un atributo decidieron preceder la opción por un +.

| *Opción* | *Descripción* |
|---|---|
| -a | La variable es de tipo array |
| -f | Mostrar el nombre e implementación de las funciones |
| -F | Mostrar sólo el nombre de las funciones |
| -i | La variable es de tipo entero |
| -r | La variable es de sólo lectura |
| -x | Exporta la variable (equivalente a export) |

*Tabla 4.1: Opciones del comando interno declare*

Si escribimos **declare** sin argumentos, nos muestra todas las variables existentes en el sistema. Si usamos la **opción -f**, nos muestra sólo los nombres de funciones (las veremos en la práctica siguiente) y su implementación, y si usamos la **opción -F**, nos muestra sólo los nombres de las funciones existentes.

La **opción -i** declara (Bash no exige que se declaren las variables) la variable de tipo entero, lo cual permite que podamos realizar operaciones aritméticas con ella. Por ejemplo, si usamos variables normales para realizar operaciones aritméticas:

```
$bash var1=5
$bash var2=4
$bash result ado=$var1*$var2
$bash echo $resul  tado
5*4
```

Sin embargo, si ahora usamos variables de tipo entero:

```
$bash declare -i var1=5
$bash declare -i var2=4
$bash declare -i resul  tado
$bash result ado=$var1*$var2
$bash echo $resul  tado
20
```

Para que la operación aritmética tenga éxito, no es necesario que declaremos como enteras a `var1` y `var2`, basta con que recojamos el valor en una variable resultado declarada como entera. Es decir, podríamos hacer:

```
$bash result ado=4*6
$bash echo $resul  tado
24
```

E incluso podemos operar con variables inexistentes:

```
$bash result ado=4*var_ine  xi st ent e
$bash echo $resul  tado
0
```

Podemos saber el tipo de una variable con la **opción -p**. Por ejemplo:

```
$bash declare -p resultado
declare -i resultado="24"
```

La **opción -x** es equivalente a usar el comando *export* sobre la variable. Ambas son formas de

exportar una variable.

La **opción -r** declara una variable como de sólo lectura, con lo que a partir de ese momento no podremos modificarla ni ejecutar **unset** sobre ella. Existe otro comando interno llamado **readonly** que nos permite declarar variables de sólo lectura, pero que tiene más opciones que **declare -r**. En concreto, **readonly -p** nos muestra todas las variables de sólo lectura:

```
$bash readonl y -p
declare -ar BASH_VERSINFO='([0]="2" [1]="05b" [2]="0"
[3]="1" [4]="release" [5]="powerpc-apple-darwin7.0")'
declare -ir EUID="503"
declare -ir PPID="686"
declare -ir UID="503"
```

Además se nos indica si la variable es de tipo array (-a) o entero (-i)

## 4.2.- Expresiones aritméticas

Los valores de todas las variables de la shell de Bash se almacenan en forma de cadenas de caracteres. Aunque esta característica hace que el procesamiento de datos simbólicos sea fácil y agradable, también hace que el procesamiento de datos numéricos presente algunas dificultades. La razón es que los datos enteros se almacenan, en realidad, en forma de cadenas de caracteres. Para poder realizar operaciones aritméticas y lógicas con ellos, es preciso traducirlos a enteros y volver a traducir el resultado a una cadena de caracteres para almacenarla correctamente en una variable de la *shell*.

Afortunadamente, en la *shell* de Bash existen varias maneras de realizar operaciones aritméticas con datos numéricos:

- Utilizando la expansión del shell `$((expresión))`
- Utilizando la orden let

La evaluación de expresiones se efectúa empleando enteros largos y no se hacen comprobaciones de desbordamiento. Cuando se utilizan variables de la shell en un expresión, las variables se

expanden (se sustituyen las variables por sus valores en la expresión) y se traducen al tipo entero largo antes de efectuar la evaluación de la expresión.

### 4.2.1.- Expansión del shell $((expresión))

Para que el shell evalúe una operación aritmética y no la tome como argumentos de un comando, las expresiones aritméticas deberán ir encerradas entre **$((** y **))**.

```
$((expresión))
```

Las expresiones aritméticas, al igual que las variables y la sustitución de comandos, se evalúan dentro de las comillas doble (“ ”).

```
Basándonos en el comando date +%j, que nos devuelve el número de
día Juliano, hacer un script que nos diga cuantos días quedan
hasta el próximo 31 de Diciembre.
El comando se puede implementar restando a 365 días el número de
días transcurridos así:

$bash echo "$(( 365 - $(date +%j) )) dias para el 31 de
Dic"
```

En la Tabla 4.2, se muestran algunos de los operadores aritméticos soportados por Bash.

| ***Operador*** | ***Descripción*** |
|---|---|
| + | Suma |
| * | Multiplicación |
| - | Resta |
| / | División entera |
| % | Resto de la división entera |
| () | Agrupar operaciones |
| ** | Potencia |

*Tabla 4.2: Operadores aritméticos admitidos por Bash*

### 4.2.2.- Similitud con las expresiones aritméticas C

Las expresiones aritméticas de Bash se han diseñado de forma equivalente a las expresiones de C. Por ejemplo `$((x+=2))` añade `2` a `x`.

Aunque algunos operadores (p.e. * o los paréntesis) son caracteres especiales para Bash, no hace falta precederlos por el carácter de escape siempre que estén dentro de **$((...))**. Igualmente, a las variables que están dentro de la expresión aritmética no hace falta precederlas por $ para obtener su valor.

Bash, además, nos permite usar los operadores relacionales (<, >, <= ,>= ,== ,!=) y lógicos de C (!, &&, ||) interpretando, al igual que C, el 1 como cierto y el 0 como falso.

### 4.2.3.- El comando interno let

El comando interno **let** nos permite asignar el resultado de una expresión aritmética a una variable. Tiene la siguiente sintaxis:

```
let var=expresion
```

**expresion** es cualquier expresión aritmética y no necesita estar encerrada entre $((...)).

El comando **let**, a diferencia del comando *declare -i*, no crea una variable de tipo entero, sino una variable de tipo cadena de caracteres normal. Por ejemplo:

```
$bash let a=4 *3
$bash declare -p a
declare -- a="12"
```

Vemos que **a** es de tipo cadena, mientras que si usamos *declare -i* nos la crea de tipo entero:

```
$bash declare -i a=3  *4
$bash declare -p a
declare -i a="12"
```

Al igual que pasa con las variables normales, las declaradas con **let** no pueden tener espacios entre la variable y el signo =, ni entre el signo = y el valor. No obstante, sí pueden tener espacios si encerramos la expresión entre comillas:

```
$bash let x="  (9*5)  / 7  "
$bash echo $x
6
```

## 4.3.- Las sentencias condicionales

### 4.3.1.- Las sentencias if, elif y else

La sentencia condicional tiene el siguiente formato:

```
if condición
then
    sentencias-then-if
elif condición
then
    sentencias-then-elif
else
    sentencias-else
fi
```

El lenguaje nos obliga a que las sentencias estén organizadas con esto saltos de línea, aunque también se pueden poner los **then** en la misma línea que los **if**, para lo cual debemos usar el separador de comandos, que en Bash es el punto y coma (**;**), así:

```
if condición ; then
    sentencias-then-if
elif condición ; then
    sentencias-then-elif
else
    sentencias-else
fi
```

### 4.3.2.- Los códigos de terminación

En UNIX los comandos terminan devolviendo un código numérico al que se llama código de terminación (*exit status*) que indica si el comando tuvo éxito o no.

Aunque no es obligatorio que sea así, normalmente un código de terminación 0 significa que el comando terminó correctamente, y un código entre 1 y 255 corresponde a posibles códigos de error. En cualquier caso siempre conviene consultar la documentación del comando para interpretar mejor sus códigos de terminación. Por ejemplo, el comando **diff** devuelve 0 cuando no encuentra diferencias entre los ficheros comparados, 1 cuando hay diferencias, y 2 cuando se produce algún error (p.e que uno de los ficheros pasados como argumento no se puede leer).

La sentencia **if** comprueba el código de terminación de un comando en la **condición**, si éste es 0 la condición se evalúa como cierta. Luego la forma normal de escribir una sentencia condicional **if** es:

```
if comando ; then
    Procesamiento normal
else
    Procesamos el error
fi
```

**¿Cómo podemos conocer el código de terminación de un comando?**

Para ello podemos usar la variable especial **?**, cuyo valor es **$?**, y que indica el código de terminación del último comando ejecutado por el shell. Por ejemplo:

```
$bash cd dire rroneo
-bash: cd: direrroneo: No such file or directory
$bash echo $?
1
```

Al haber fallado el último comando **$?** vale 1. Sin embargo, si el comando se ejecuta bien **$?** valdrá 0:

```
$bash cd direxi stent e
```

```
$bash echo $?
0
```

La variable **?** debe de ser leída junto después de ejecutar el comando, siendo muy típico guardar su valor en una variable `ct=$?` para su posterior uso.

### 4.3.3.- La sentencia exit

La sentencia **exit** puede ser ejecutada en cualquier sitio, y lo que hace es abandonar el script . Suele usarse para detectar situaciones erróneas que hacen que el programa deba detenerse, aunque a veces se utiliza para "cerrar" un programa.

### 4.3.4.- Operadores lógicos y códigos de terminación

Podemos combinar varios códigos de terminación de comandos mediante los operadores lógicos **and** (representada con **&&**) **or** (representada con ||) y **not** (representada con **!**).

Estas operaciones, al igual que en otros lenguajes como C o Java, funcionan en shortcut, es decir, el segundo operando sólo se evalúa si el primero no determina el resultado de la condición. Según esto la operación:

```
if cd /tmp && cp 001.tmp $HOME ; then
    ....
fi
```

Ejecuta el comando `cd /tmp`, y si éste tiene éxito (el código de terminación es 0), ejecuta el segundo comando `cp 001.tmp $HOME`, pero si el primer comando falla, ya no se ejecuta el segundo porque para que se cumpla la condición ambos comandos deberían de tener éxito (si un operando falla ya no tiene sentido evaluar el otro).

El operador || por contra ejecuta el primer comando, y sólo si éste falla se ejecuta el segundo. Por ejemplo:

```
if cp /tmp/001.tmp ~/d.tmp || cp /tmp/002.tmp ~/d.tmp
then
    ....
```

```
fi
```

Aquí, si el primer comando tiene éxito, ya no se ejecuta el segundo comando (ya que se tiene la condición con que se cumple uno de sus operandos).

Obsérvese, que a diferencia de C, el código de terminación 0 es el que indica verdadero, y cualquier otro código indica falso.

Por último el operador **!** niega un código de terminación. Por ejemplo:

```
if ! cp /tmp/001.tmp ~/d.tmp; then
      Procesa el error
fi
```

### 4.3.5.- Test condicionales

La sentencia **if** lo único que sabe es evaluar códigos de terminación. Pero esto no significa que sólo podamos comprobar si un comando se ha ejecutado bien. El comando interno **test** nos permite comprobar otras muchas condiciones, que le pasamos como argumento, para acabar devolviéndonos un código de terminación.

```
test [condición]
```

Una forma alternativa al comando **test** es el operador "[[ ]]" dentro del cual metemos la condición a evaluar

```
[[ condición ]]
```

es decir, `test cadena1 = cadena2` es equivalente a `[[ cadena1 = cadena2 ]]`. Los espacios entre los corchetes y los operandos, o entre los operandos y el operador de comparación **=** son obligatorios.

Usando los **test** condicionales podemos evaluar atributos de un fichero (si existe, que tipo de fichero es, que permisos tiene, etc.), comparar dos ficheros para ver cual de ellos es más reciente, comparar cadenas, e incluso comparar los números que almacenan las cadenas (comparación numérica).

### Comparación de cadenas

Los operadores de comparación de cadenas se resumen en la Tabla 4.3. Las comparaciones que se realizan con los símbolos **<** y **>** son lexicográficas, es decir comparaciones de diccionario, donde por ejemplo *q* es mayor que *perro*. Obsérvese que no existen operadores **<=** y **>=** ya que se pueden implementar mediante operaciones lógicas.

| ***Operador*** | ***Verdadero si ...*** |
|---|---|
| `str1 = str2` | Las cadenas son iguales |
| `str1 != str2` | Las cadenas son distintas |
| `str1 < str2` | *str1* es menor lexicográficamente a *str2* |
| `str1 > str2` | *str1* es mayor lexicográficamente a *str2* |
| `-n str1` | *str1* es no nula y tiene longitud mayor a cero |
| `-z str1` | *str1* es nula (tiene longitud cero) |

*Tabla 4.3: Operadores de comparación de cadenas*

Cuando se utilizan operadores de comparación de cadenas, las variables que aparezcan en la comparación deben ir encerradas entre comillas dobles.

```
fichero=$1
if [[ -z "$fichero" ]] ; then
   echo 'Use: muestraFichero <fichero>'
else
   cat $fichero
fi
```

Aunque normalmente se considera mejor técnica de programación el encerrar todo el código dentro de un bloque **if** y el comportamiento erróneo dentro de otro bloque **else**, en este caso el bloque **if** corresponde a una situación de error que debería abortar todo el script, con lo que vamos a poner un **exit**, y dejamos el ejemplo como se muestra a continuación:

```
# Script que muestra las líneas de un fichero
# Tiene la forma
#       muestraFichero <fichero>
```

```
fichero=$1
if [[ -z "$fichero" ]] ; then
   echo 'Use: muestraFichero <fichero>'
   exit 1
fi
cat $fichero
```

### Comparación numérica de enteros

El shell también permite comparar variables que almacenen números, para ello se deben de utilizar los operadores de la Tabla 4.4

| *Operador* | *Descripción* |
|---|---|
| -lt | Menor que |
| -le | Menor o igual que |
| -eq | Igual que |
| -ge | Mayor igual que |
| -gt | Mayor que |
| -ne | Diferente |

*Tabla 4.4: Operadores de comparación numérica de cadenas*

### Combinación de condiciones. Operadores lógicos

Los test de condición (los que van entre corchetes [[]]) también se pueden combinar usando los operadores lógicos **&&**, ll y **!**.

```
if [[ condicion] ] && [[ condicion ]]; then
```

También es posible combinar comandos del shell con test de condición:

```
if comando && [[ condicion ]]; then
```

Además, a nivel de test de condición (dentro de los **[[]]**) también se pueden usar operadores lógicos, pero en este caso debemos de usar los operadores **-a** (para and) y **-o** (para or).

Por ejemplo, la siguiente operación comprueba que `$reintegro` sea menor o igual a `$saldo`, y que `$reintegro` sea menor o igual a `$max_cajero`:

```
if [ $reintegro -le $saldo -a  $reintegro -le $max ]
then
     ....
fi
```

Aunque el operador lógico **-a** tiene menor precedencia que el operador de comparación **-le**, en expresiones complejas conviene usar paréntesis que ayuden a entender la expresión. Si usamos paréntesis dentro de un test de condición conviene recordar dos reglas:

- Los paréntesis dentro de expresiones condicionales deben ir precedidos por el carácter de escape **** (para evitar que se interpreten como una sustitución de comandos).
- Los paréntesis, al igual que los corchetes, deben de estar separados por un espacio.

Luego la operación anterior se puede escribir como:

```
if [ \( $reintegr -le $saldo \) -a \( $reintegro -le $max \) ]
then
   ....
fi
```

### Comprobar atributos de ficheros

El tercer tipo de operadores de comparación nos permiten comparar atributos de fichero. Existen 22 operadores de este tipo que se resumen en la Tabla 4.5.

| ***Operador*** | ***Verdadero si ...*** |
|---|---|
| `-a fichero` | *fichero* existe |
| `-b fichero` | *fichero* existe y es un dispositivo de bloque |
| `-c fichero` | *fichero* existe y es un dispositivo de carácter |
| `-d fichero` | *fichero* existe y es un directorio |
| `-e fichero` | *fichero* existe (equivalente a -a) |
| `-f fichero` | *fichero* existe y es un fichero regular |

| *Operador* | *Verdadero si ...* |
|---|---|
| `-g fichero` | *fichero* existe y tiene activo el bit de seguridad |
| `-G fichero` | *fichero* existe y es poseído por el grupo ID efectivo |
| `-h fichero` | *fichero* existe y es un enlace simbólico |
| `-k fichero` | *fichero* existe y tiene el stricky bit activado |
| `-L fichero` | *fichero* existe y es un enlace simbólico |
| `-N fichero` | *fichero* existe y fue modificado desde la última lectura |
| `-O fichero` | *fichero* existe y es poseído por el grupo ID efectivo |
| `-p fichero` | *fichero* existe y es un pipe o named pipe |
| `-r fichero` | *fichero* existe y podemos leerlo |
| `-s fichero` | *fichero* existe y no esta vacío |
| `-S fichero` | *fichero* existe y es un socket |
| `-u fichero` | *fichero* existe y tiene activo el bit de setuid |
| `-w fichero` | *fichero* existe y tenemos permiso de escritura |
| `-x fichero` | *fichero* existe y tenemos permiso de ejecución, o de búsqueda si es un directorio |
| `fich1 -nt fich2` | La fecha de modificación de *fich1* es más moderna que la de *fich2* |
| `fich1 -ot fich2` | La fecha de modificación de *fich1* es más antigua que la de *fich2* |
| `fich1 -ef fich2` | *fich1* y *fich2* son el mismo fichero |

*Tabla 4.5: Operadores para comprobar atributos de un fichero*

El script siguiente muestra el uso de esto atributos en un ejemplo. El script copia un archivo pasado como argumento en un directorio también pasado por argumento. Comprueba que el fichero y el directorio existan y, además, que no exista un fichero con el mismo nombre dentro del directorio.

```
#!/bin/bash

#Declaracion de variables
numArg=$#

if [[ $numArg != 2 ]]
then
  echo $0 "<fichero> <directorio>"
else
  #Comprobaciones sobre el primer argumento: fichero
  if [[ -a $1 ]] #Miramos si existe el primer argumento
  then
    echo "$1 existe"
    if [[ -f $1 ]] # Miramos si el argumento es un fichero
    then
       echo "$1 es un fichero"
    else
       echo "$1 no es un fichero"
       exit
    fi
  else
    echo "$1 no existe"
    exit
  fi

  #Comprobaciones sobre el segundo argumento: directorio
  if [[ -a $2 ]]
  then
     echo "$2 existe"
     if [[ -d $2 ]] #Comprobamos si es un directorio
     then
        echo "$2 es un directorio"
     else
        echo "$2 no es un directorio"
        exit
     fi
  else
    echo "$2 no existe"
```

```
      exit
    fi

    #Comprobamos que no exista un fichero con el mismo nombre en
el directorio
    if [[ -a $2/$1 ]]
    then
      echo "Existe un fichero con nombre $1 en el directorio $2"
    else
      cp $1 $2/.  #Copiamos el fichero en el directorio
    fi
  fi
```

### 4.3.6.- If aritmético

Las expresiones aritméticas pueden usarse en las distintas sentencias de control de flujo, en cuyo caso la expresión va entre dobles paréntesis, pero sin el **$** delante, por ejemplo, el **if** aritmético tendría la forma:

```
if ((expresión aritmética))
then
   cuerpo
fi
```

```
if ((5+7==12))
then
   echo “Son iguales”
fi
```

## 4.4.- El bucle for

El bucle **for** en Bash es un poco distinto a los bucles **for** tradicionales de lenguajes como C o Java; en realidad, se parece más al bucle **for each** de otros lenguajes, ya que aquí no se repite un número fijo de veces, sino que se procesan los elementos de una lista uno a uno.

Su sintaxis es la siguiente:

```
for var [in lista]
do
    .....
   lista de órdenes
    .....
done
```

Ejecuta las órdenes de "*lista de órdenes"* tantas veces como palabras haya en "*lista*". En cada vuelta del bucle, se almacena un elemento de *lista* en *var*.

Si se omite ***in lista***, se recorre el contenido de **$@**, pero aunque vayamos a recorrer esta variable, es preferible indicarla explícitamente por claridad.

Por ejemplo si queremos recorrer una lista de planetas podemos hacer:

```
for planeta in Mercurio Venus Tierra Marte Jupiter Saturno
do
   echo $planeta # Imprime cada planeta en una línea
done
```

La lista del bucle **for** puede contener comodines. Por ejemplo, el siguiente bucle muestra información detallada de todos los ficheros en el directorio actual:

```
for fichero in *
do
   ls -l "$fichero"
done
```

Para recorrer los argumentos recibidos por el script, lo correcto es utilizar **$*. El delimitador que usa el bucle** for **para la variable lista es el que indiquemos en** IFS**, y por defecto este delimitador es el espacio, aunque en ocasiones conviene cambiarlo.**

El ejemplo siguiente recorre dos veces los argumentos de un programa con $*. La primera vez usamos como delimitador el valor por defecto de la variable IFS (el espacio) y la segunda vez cambiamos el valor de la variables IFS para que el delimitador sean las comillas dobles (").

```
  # Bucles que recorren los argumentos
  for arg in $*
  do
       echo "Elemento:$arg"
  done
  IFS='"'          #Cambiamos el delimitador a las comillas
dobles ("")
  for arg in $*
  do
       echo "Elemento2:$arg"
  done
```

Si ahora lo ejecutamos, obtenemos la siguiente salida:

```
$bash recibe.s h "El perro" "La casa"
Elemento:El
Elemento:perro
Elemento:La
Elemento:casa
Elemento2:El perro
Elemento2:La casa
```

### For aritmético

El bucle **for** permite recorrer los elementos de una lista. Sin embargo, existe otro tipo de bucle que se asemeja más a los bucles de C, el **for aritmético**. Éste tiene la siguiente sintaxis:

```
for (( inicialización ; condición ; actualización ))
do
    cuerpo
done
```

En este caso los espacios en `((inicialización;condición;actualización))` no son necesarios.

El ejemplo siguiente muestra por pantalla los 20 primeros números.

```
for((i=1;i<=20;i++))
do
   echo "$i"
done
```

## 4.5.- Bucles while y until

Los bucles **while** y **until** son útiles cuando los combinamos con características como la aritmética con enteros, la entrada y salida de variables, y el procesamiento de opciones de la línea de comandos. Su sintaxis es:

| **while *condición*** | **until *condición*** |
|---|---|
| **do** | **do** |
| **lista de órdenes** | **lista de órdenes** |
| **done** | **done** |

**while** ejecuta las órdenes especificadas en *"lista de órdenes"* mientras el resultado de evaluar *"condición"* siga siendo true. Por su parte, **until** ejecuta las órdenes especificadas en *"lista de órdenes"* mientras el resultado de evaluar *"condición"* sea false.

Un ejemplo del uso del **while** es el siguiente:

```
#!/bin/bash
i=0
while [[ $i -lt 10 ]]
do
   echo El contador es $i
   let i=i+1
done
```

Este script emula la estructura **for** de C, Pascal, ...

Otro ejemplo con **until**:

```
#!/bin/bash
i=20
```

```
until [[ $i -lt 10 ]]
do
   echo Contador $i
   let i-=1
done
```

Tanto en el bucle **for** como en el **while**, podemos usar **break** para indicar que queremos salir del bucle en cualquier momento. El siguiente *script* implementa el juego de adivinar un número.

```
#!/bin/bash
MINUM=8
while [[ 1 ]]; do
        echo "Introduzca un número: "
        read USER_NUM

        if [[ $USER_NUM -lt $MINUM ]]; then
            echo "El número introducido es menor que el mío"
            echo " "
        elif [[ $USER_NUM -gt $MINUM ]]; then
            echo "El número introducido es mayor que el mío"
            echo " "
        elif [[ $USER_NUM -eq $MINUM ]]; then
            echo "Acertaste: Mi número era $MINUM"
            break
        fi
done
# Comandos inferiores...
echo "El script salió del bucle. Terminando..."
exit
```

La condición del bucle **while**, *test* 1, siempre retornará TRUE, y el bucle se ejecutará infinitamente, preguntando en cada vuelta por un nuevo número. Si el número introducido es el mismo que el valor de $MINUM (número a adivinar), **break** hará que se salga del bucle y continúen ejecutándose los comandos inferiores.

### While aritmético

Al igual que ocurría con el ***for*** y el ***if***, también existe el **while aritmético**. Tendría la forma:

```
while ((expresion aritmética))
do
  cuerpo
done
```

## 4.6.- Entrada y salida de texto

En este apartado comentaremos detalladamente los comandos **echo, printf y read**, los cuales nos permiten realizar las operaciones de entrada/salida que requiere un *script*.

### 4.6.1.- El comando interno echo

Como ya sabemos, **echo** simplemente escribe en la salida estándar los argumentos que recibe.

La Tabla 4.6 muestra las opciones que podemos pasarle a **echo**.

| *Opción* | *Descripción* |
|---|---|
| -e | Activa la interpretación de caracteres precedidos por el carácter de escape. |
| -E | Desactiva la interpretación de caracteres precedidos por carácter de escape. Es la opción por defecto. |
| -n | Omite el carácter \n al final de la línea (es equivalente a la secuencia de escape \c). |

*Tabla 4.6: Opciones del comando echo*

La opción **-e** activa la interpretación de las secuencias de escape, que por defecto **echo** las ignora, es decir:

```
$bash echo "Ho la\nAdios"
Hola\nAdios
$bash echo - e "Hola\ nAdios "
Hola
Adios
```

La Tabla 4.7 muestra las secuencias de escape que acepta **echo** (cuando se usa la opción **-e**). Para que éstas tengan éxito deben de encerrarse entre comillas simples o dobles, ya que sino el carácter

de escape \ es interpretado por el shell y no por **echo**.

| *Secuencia de escape* | *Descripción* |
|---|---|
| \a | Produce un sonido “poom” (alert) |
| \b | Borrar hacia atrás (backspace) |
| \c | Omite el \n al final (es equivalente a la opción -n. Debe colocarse al final de la línea |
| \f | Cambio de página de impresora |
| \n | Cambio de línea |
| \r | Retorno de carro |
| \t | Tabulador |

*Tabla 4.7: Secuencias de escape del comando echo*

El carácter de escape **\b** se puede usar para borrar texto escrito. Por ejemplo, el siguiente ejemplo muestra un *script* que permite ir escribiendo el porcentaje realizado de un proceso.

```
for ((i=0;i<10;i++))
do
   echo -n "Procesado $i"
   sleep 1
   echo -ne "\b\b\b\b\b\b\b\b\b\b"
done
```

El bucle se repite de 0 a 9; sin embargo, si queremos hacerlo de 0 a 100, esto se complica ya que estos números tienen distinto número de dígitos. Para solucionarlo podemos usar **\r** que retorna el cursor en el terminal sin introducir un cambio de línea (como hace **\n**). El *script* quedaría de la siguiente forma:

```
for ((i=0;i<10;i++))
do
   echo -n "Procesado $i"
   sleep 1
   echo -ne "\r"
done
```

### 4.6.2.- El comando interno printf

Aunque el comando **echo** es suficiente en la mayoría de los casos, en ocasiones, especialmente a la hora de formatear texto en pantalla, es necesario más flexibilidad. Para estos casos se utiliza el comando **printf**.

Este comando, en principio, puede escribir cadenas igual que echo:

```
$bash printf  "Hola  mundo"
Hola mundo
```

Sin embargo, a diferencia de **echo** no imprime el cambio de línea del final por defecto, sino que debemos indicarlo explícitamente (\n). Además, **printf** por defecto interpreta las secuencias de escape.

```
$bash printf  "Hola  mundo\ n"
Hola mundo
```

El formato de este comando es muy parecido al de la función printf() de C:

**printf *cadenaformato* [argumentos]**

Dentro de la cadena de formato que recibe como primer argumento pueden aparecer especificadores de formato, los cuales empiezan por % y se describen en la Tabla 4.8.

| ***Especificador*** | ***Descripción*** |
|---|---|
| `%c` | Imprime el primer carácter de una variable cadena |
| `%d` | Imprime un número decimal |
| `%i` | Igual que %d |
| `%e` | Formato exponencial b.precisione[+-]e |
| `%E` | Formato exponencial b.precisionE[+-]e |
| `%f` | Número en coma flotante b.precision |
| `%o` | Octal sin signo |
| `%s` | Cadena (string) |
| `%b` | Interpreta las secuencias de escape del argumento cadena |

| *Especificador* | *Descripción* |
|---|---|
| %q | Escribe el argumento cadena de forma que pueda ser usado como entrada a otro comando |
| %u | Número sin signo (unsigned) |
| %x | Hexadecimal con letras en minúscula |
| %X | Hexadecimal con letras en mayúscula |
| %% | % literal |

*Tabla 4.8: Especificadores de formato de printf*

Un ejemplo de uso de los especificadores de formato es el siguiente:

```
$bash n=30
$bash nombre=Fernando
$bash printf "El cliente %d es %s\n" $n $nombre
El cliente 30 es Fernando
```

El especificador de formato tiene el siguiente formato general:

**%[Flag][Ancho][.Precisión]Especificador**

El significado de estos campos depende de si el especificador de formato es para una cadena de caracteres o para un número.

En el caso de que sea una cadena **Ancho** se refiere al ancho mínimo que ocupa la cadena. Por ejemplo:

```
$bash printf "|%10s|\n" Hola
|      Hola|
```

Si queremos indicar un máximo para la cadena debemos de usar el campo **Precisión**:

```
$bash printf "|%-10.10s|\n" "Fernando Lopez"
|Fernando L|
```

El **Flag -** se usa para indicar que queremos alienar a la izquierda:

```
$bash printf "|%-10s|\n" Hola
|Hola      |
```

Dos variantes de **%s** son **%b** y **%q**. El especificador de formato **%b** hace que se interpreten las secuencias de escape de los argumentos. Es decir:

```
$bash printf "%s\n" "Hola \nAdio s"
Hola\nAdios
$bash printf "%b\n" "Hola \nAdio s"
Hola
Adios
```

El especificador de formato **%q** escribe el argumento de forma que pueda ser usado como entrada de otro comando. Por ejemplo:

```
$bash printf "%s\n" "Hola a t odos"
Hola a todos
$bash printf "%q\n" "Hola a t odos"
Hola\ a\ todos
```

Bash no puede manejar expresiones con números en punto flotante, sólo enteros, pero sí puede recibir la salida de un comando que devuelva una cadena con un número en punto flotante. En este caso podemos formatear este número usando **%e, %E, %f.** Entonces **Ancho** es el número total de dígitos a utilizar y **Precisión** indica el número de dígitos a la derecha del punto. Por ejemplo:

```
$bash n=123.4 567
$bash printf "|%9.3 f|\n" $n
| 123.457|
$bash printf "|%1 1.3E|\ n" $n
| 1.235E+02|
```

Respecto a los **Flag** que puede tener un número, estos se resumen en la Tabla 4.9.

| ***Flag*** | ***Descripción*** |
|---|---|
| + | Preceder por + a los números positivos |
| espacio | Preceder por un espacio a las números positivos |
| 0 | Rellenar con 0's a la izquierda |
| # | Preceder por 0 a los números en octal %o y por 0x a los |

| *Flag* | *Descripción* |
|---|---|
| | números en hexadecimal %x y %X |

*Tabla 4.9: Flags de los especificadores de formato para números*

Por ejemplo:

```
$bash printf  "|%+1 0d|\n"  56
|           +56|
$bash printf  "|%0 10d|\n"  56
|0000000056|
$bash printf  "|%x|\ n" 56
|38|
$bash printf  "|%#x| \n"  56
|0x38|
```

### 4.6.3.- El comando interno read

El comando **read** sirve para leer información del teclado y guardarla en variables. Su sintaxis es:

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

Esta sentencia lee una línea de la entrada estándar y la parte en palabras separadas por el símbolo que indique la variable IFS (por defecto espacio o tabulador). Las palabras se asignan a *var1*, *var2*, etc. Si hay más palabras que variables, las últimas palabras se asignan a la última variable. Por ejemplo:

```
$bash read var1  var2  var3
Esto es una prueba
$bash echo $var1
Esto
$bash echo $var2
es
$bash echo $var3
una prueba
```

Si omitimos las variables, la línea entera se asigna a la variable REPLY

Las principales opciones del comando **read**, se resumen en la Tabla 4.10.

| *Opción* | *Descripción* |
|---|---|
| `-a` | Permite leer las palabras como elementos de un array |
| `-d` | Permite indicar un delimitador de fin de línea |
| `-n max` | Permite leer como máximo *max* caracteres |
| `-p` | Permite indicar un texto de prompt |

*Tabla 4.10: Opciones del comando read*

La opción **-d** nos permite indicar un delimitador de fin de línea de forma que la línea se lee hasta encontrar este delimitador.

La opción **-n** nos permite especificar un número máximo de caracteres a leer. Si se intentan escribir más caracteres que los indicados en esta opción, simplemente se acaba la operación de lectura.

La opción **-p** nos permite aportar un texto de prompt al comando, que se imprime antes de pedir el dato:

```
$bash read – p "Cual  es tu  nombre:" nombre
Cual es tu nombre:Fernando
```

## 4.7.- Ejercicios

**Todos los *scripts* planteados deberán tener**

- **Un comentarios al principio que indique lo que hace.**
- **Comentarios a lo largo del *script* para documentar lo que se va haciendo**
- **En el caso de recibir parámetros, comprobación de que el numero de parámetros recibidos es correcto, mostrando en caso contrario un mensaje con la sintaxis.**

1.- Crea un script de nombre **positivoNegativo.sh** que pida un numero por teclado y determine si es positivo, negativo o cero.

2.- Crea un script que se llame **copiar.sh** y que acepte dos argumentos. El primer argumento representa el nombre de un archivo existente cuyo contenido queremos copiar y el segundo argumento es el nombre del fichero donde se copiará el contenido. El scritp copiará el contenido del primer fichero en el segundo fichero, haciendo todas la comprobaciones pertinentes

a) El número de argumentos al llamar al programa debe ser el correcto, si no mandar un mensaje indicando la sintaxis correcta.

b) El fichero o directorio dado como primer argumento debe existir, si no existe debe mandarse un mensaje de error al usuario y finalizar.

c) El segundo argumento no debe corresponder con ningún archivo o directorio existentes, si lo es, se debe mandar un mensaje al usuario y finalizar.

3.- Desarrolla un script de nombre **cal.sh** que implemente una calculadora con las operaciones básicas (suma, resta, multiplicación, división, potencia). Nota: Se puede hacer en menos de cuatro líneas de código.

4.- Escribe un script llamado **sumaN.sh** que pida un número (**N**) e imprima por pantalla la

suma de los **N** primeros números.

5.- Desarrolla un script de nombre **ficheroDirectorio.sh** que reciba un número indeterminado de parámetros. El script deberá clasificar los argumentos según sean ficheros o directorios y guardarlos en dos archivos (fich.txt y dir.txt).

6.- Crea un script de nombre **muestraLinea.sh** que reciba dos argumentos en la línea de órdenes; el primero será un fichero y el segundo un número `n`. El script mostrará por pantalla la línea número `n` del fichero pasado como argumento. Si la línea no existiese, el script deberá mostrar el mensaje correspondiente.

7.- Haz un script llamado **ocupaTxt.sh** que dado un directorio como argumento nos diga el tamaño total de los fichero con extensión **.txt** que haya en él.

# 5.- PROGRAMACIÓN BASH II

## 5.1.- La sentencia case

Mientras que esta sentencia en lenguajes como C o Java se usa para comprobar el valor de una variable simple, como un entero o un carácter, en Bash esta sentencia permite realizar una comparación de patrones con la cadena a examinar.

Su sintaxis es la siguiente:

```
case cadena in
    patron1)
       Lista de órdenes1 ;;
    patron2)
       Lista de órdenes2 ;;
    .....
esac
```

La sentencia ***case*** compara el valor presente en `cadena` con los valores de todos los patrones, uno por uno, hasta que o bien se encuentra un patrón que coincida o ya no quedan más patrones para comparar con `cadena`. Si se halla un coincidencia, se ejecutan las órdenes presentes en la correspondiente `lista-ordenes` y la sentencia case cede el control.

Cada patrón puede estar formado por varios patrones separados por el carácter |.

```
#!/bin/bash
read n
case $n in
*1* | *5* ) echo Uno o Cinco;;
*2*) echo Dos;;
*3*) echo Tres;;
*4*) echo Cuatro;;
esac
```

En una aplicación típica de la sentencia, se incluye un patrón comodín que coincide con cualquier posible valor de `cadena`. Se conoce como caso por omisión y permite la ejecución de un conjunto de órdenes adecuadas para manejar una situación excepcional cuando `cadena` no coincide con ningún patrón.

```
#!/bin/bash
read n
case $n in
*1* | *5* ) echo Uno o Cinco;;
*2*) echo Dos;;
*3*) echo Tres;;
*4*) echo Cuatro;;
*) echo Opción incorrecta;;
esac
```

## 5.2.- La sentencia select

La sentencia **select** nos permite generar fácilmente un menú simple. Su sintaxis es la siguiente:

```
select variable [in lista]
do
    Sentencias que usan $variable
done
```

Vemos que tiene la misma sintaxis que el bucle **for**, excepto por la palabra reservada **select** en vez de **for**. De hecho, si omitimos `in lista,` también se usa por defecto `$@`.

La sentencia genera un menú con los elementos de lista, asigna un número a cada elemento, y pide al usuario que introduzca un número. El valor elegido se almacena en *variable*, y el número elegido en la variable **REPLY**. Una vez elegida una opción por parte del usuario, se ejecuta el cuerpo de la sentencia y el proceso se repite en un bucle infinito.

Aunque el bucle de **select** es infinito (lo cual nos permite volver a pedir una opción cuantas veces haga falta), se puede abandonar usando la sentencia **break**. La sentencia **break**, al igual que en C y Java, se usa para abandonar un bucle y se puede usar en el caso, tanto de **select**, como de los

bucles **for**, **while** y **until**. Pero, a diferencia de C y Java, no sirve para abandonar la sentencia case, sino que ésta se abandona usando dos puntos y comas **;;**.

El **prompt** que usa la función (mensaje que se muestra para pedir la opción del menú) es el definido en la variable de entorno **PS3**, y es habitual cambiar este **prompt** antes de ejecutar **select** para que muestre al usuario un mensaje más descriptivo. Por defecto el valor de **PS3** es `#?`, lo cual no es un mensaje que suela gustar especialmente a los usuarios.

El siguiente script pide dos números y a continuación muestra un menú para elegir que se quiere hacer con ellos (sumar, restar, multiplicar o dividir).

```
#!/bin/bash

#Almacenamos las opciones del menú
OPCIONES="Sumar Restar Multiplicar Dividir Salir"

#Cambiamos el prompt para que sea más descriptivo
PS3="Elija una opción: "
#Leemos dos números
read -p "Numero 1: " n1
read -p "Numero 2: " n2

select opt in $OPCIONES
do
   if [[ $opt = "Sumar" ]]
   then
      echo $((n1+n2))
   elif [[ $opt = "Restar" ]]
   then
      echo $((n1-n2))
   elif [[ $opt = "Multiplicar" ]]
   then
      echo $((n1*n2))
   elif [[ $opt = "Dividir" ]]
   then
      echo $((n1/n2))
   elif [[ $opt = "Salir" ]]
```

```
    then
       echo -----Fin-------
       break
    else
       echo opción errónea
    fi
  done
```

La variable *OPCIONES* se utiliza para almacenar las opciones del menú (la forma de separar las opciones dependerá de la variable IFS). Un ejemplo de ejecución es el siguiente:

```
mluque@tweety:/tmp$ ./pru.sh
Numero 1: 8
Numero 2: 2
1) Sumar
2) Restar
3) Multiplicar
4) Dividir
5) Salir
Elija una opcion: 4
4
Elija una opcion: 5
-----Fin-------
```

## 5.3.- La sentencia shift

La sentencia `shift`, por defecto, desplaza los argumentos de la línea de órdenes una posición a la izquierda, haciendo que $2 pase a ser $1, $3 pase a ser $2, y así sucesivamente. El número de posiciones que se desplazan puede ser mayor que uno y se especifica como argumento de la orden.

Su formato es el siguiente:

```
shift [n]
```

donde **n** es el número de desplazamientos a la izquierda que queremos hacer con los argumentos pasados al script. Si se omite **n**, por defecto se considera que el número de desplazamientos vale 1.

Luego, si ejecutamos el comando **shift 1**, $1 pasará a contener el valor de $2, $2 el de $3 y así sucesivamente. El valor de $1 se pierde.

Un ejemplo clásico del uso de este comando son los scripts con opciones. Las opciones, al igual que los argumentos se reciben en los parámetros posicionales, es decir, si ejecutamos `hacer -o esto.txt aquello.txt`, en $1 recibimos `-o`, en $2 `esto.txt` y en $3 `aquello.txt`. En principio, para tratar las opciones no nos haría falta saber nada más. El problema está en que normalmente las opciones son "*opcionales*", es decir, pueden darse o no, con lo que en función de que la opción se de o no, el script utilizará unos parámetros posicionales u otros

```
if [[ $1 = -o ]]
then
   Ejecuta las operaciones con $2 y $3
else
   Ejecuta las operaciones con $1 y $2
fi
```

Utilizando el comando `shift` podemos resolver esto y hacer el script anterior más sencillo:

```
if [[ $1 = -o ]]; then
    Procesa -o
    shift
fi
Ejecuta la operación con $1 y $2
```

## 5.4.- Listas (Arrays)

Bash admite las listas unidimensionales. Una lista (array) es una colección de elementos dotada de nombre; los elementos son todos del mismo tipo y están almacenados en posiciones consecutivas de memoria. Los elementos de la lista están numerados, y el primer elemento de lista tiene el número 0. No existe límite para el tamaño de una lista, y no se requiere que los elementos de la lista se asignen de forma contigua.

### Creación de listas

Para declarar un lista podemos usar el comando **declare -a** de la siguiente forma:

```
bash$  declare -a A
```

Si ahora preguntamos por la variable A, nos dice que es un array vacío:

```
bash$  declare -p A
declare -a A='()'
```

Realmente no hace falta declarar una lista con **declare -a**, podemos crearla asignándole directamente valores de la siguiente manera:

```
bash$  B=(Jamo n 4  Melon)
bash$  declare -p B
declare -a B='([0]="Jamon" [1]="4" [2]="Melon")'
```

Los elementos en las listas empiezan a contar en 0, aunque podemos cambiar los índices de los elementos indicándolos explícitamente:

```
bash$  C=([5]= Melon  [0]=Ja mon  [3]=400)
bash$  declare -p C
declare -a C='([0]="Jamon" [3]="400" [5]="Melon")'
```

Obsérvese que no hace falta suministrar los elementos en orden, ni suministrarlos todos, los índices donde no colocamos un valor, simplemente valdrán "cadena nula".

Si sólo indicamos algunos índices, los demás los asigna continuando las cuenta desde el último índice asignado:

```
bash$  C=([5]= Cosa  Casa  Perro)
bash$  declare -p C
declare -a C='([5]="Cosa" [6]="Casa" [7]="Perro")'
```

Una forma útil de rellenar una lista es a partir de las líneas devueltas por una sustitución de comandos usando la forma de inicialización B=(...). Dentro de los paréntesis ponemos la sustitución de comandos. Por ejemplo, para obtener una lista con los nombres de ficheros de un directorio podemos usar:

```
bash$  B=(`ls  -1`)
bash$  declare -p B
declare   -a   B='([0]="Desktop"   [1]="Documents"   [2]="Library"
```

```
[3]="Movies" [4]="Music" [5]=Pictures")
```

### Acceso a los elementos de una lista

Para acceder a los elementos, usamos el operador corchete **[]** para indicar el índice del elemento a acceder, siendo obligatorio encerrar entre llaves **{}** la variable:

```
bash$  echo ${B [2]}
Library
```

Si no indicamos índice de elemento, por defecto nos coge el elemento de índice 0:

```
bash$  echo $B
Desktop
```

Esta es la razón por la que hay que encerrar los elementos entre llaves, porque sino siempre nos coge el primer elemento:

```
bash$  echo $B[ 3]
Movies[3]
```

También podemos inicializar una lista introduciendo valores directamente con el operador corchete:

```
bash$  D[2]=Ca sa
bash$  D[0]=A vion
bash$  declare -p D
declare -a D='([0]="Avion" [2]="Casa")'
```

Podemos usar los índices especiales * y @, los cuales retornan todos los elementos de la lista de la misma forma que lo hacen los parámetros posicionales. Cuando no están encerrados entre comillas dobles (“ “) ambos devuelven una cadena con los elementos separados por espacio, pero cuando se encierran entre comillas dobles (“ “) @ devuelve una cadena con los elementos separados por espacio, y * devuelve una cadena con los elementos separados por el valor de IFS. En cualquiera de los casos, la lista se pierde, y lo que recibimos es una cadena de caracteres

```
bash$  IFS=,
bash$  echo ${C [*]}
```

```
Cosa Casa Perro
bash$  echo ${C [@]}
Cosa Casa Perro
bash$  echo "${ C[@]}"
Cosa Casa Perro
bash$  echo "${ C[*]}"
Cosa,Casa,Perro
```

Al igual que en los parámetros posicionales, podemos iterar sobre los elementos de una lista usando un bucle `for`:

```
for e in ${C[*]}
do
   echo $e
done
```

Donde al ser `${C[*]}` una cadena de caracteres hubiéramos también podido usar `${C[@]}` o `"${C[@]}"` pero si hacemos `"${C[*]}"` lo que obtenemos es una cadena con los elementos de ésta separados por el valor de IFS (coma en este ejemplo), y no por espacio. Sin embargo el bucle hubiera funcionado si los elementos están separados por el valor de IFS, y de hecho en ocasiones (p.e. cuando los elementos tienen espacios) conviene utilizar esta última solución.

Los elementos de una lista a los que no se asigna valor tienen una cadena nula, con lo que el bucle anterior sólo imprimiría los elementos que existan en la lista C.

Podemos consultar la longitud de una lista usando la forma **${#lista[@]}**, donde `lista` es el nombre de la lista. Por ejemplo, en el ejercicio anterior:

```
bash$  echo ${# C[@]}
3
```

El ejemplo anterior nos dice que hay tres elementos aunque los índices lleguen hasta el índice 7, ya que existen posiciones sin asignar en la lista. El comando **#** sólo nos devuelve el número de posiciones ocupadas.

También debemos tener en cuenta que dentro de los corchetes va una @, y no un índice. Si

ponemos un índice, nos devuelve la longitud (en caracteres) de ese elemento. Por ejemplo, si en el índice 5 teníamos la cadena "Casa":

```
bash$  echo ${# C[5]}
4
```

Si queremos saber que elementos son no nulos en una lista, podemos usar la forma **${!lista[@]}**, donde `lista` es el nombre de la lista a consultar. Devuelve los índices de los elementos (Esto está disponible a partir de la versión 3 del bash):

```
bash$  echo ${! C[@]}
567
```

Si asignamos una lista a otra existente se pierden los valores existentes, y se asignan los nuevos. Por ejemplo, si a C del ejemplo anterior le asignamos:

```
bash$  C=(Hola  "Que  tal" Adios)
bash$  declare -p C
declare -a C='([0]="Hola" [1]="Que tal" [2]="Adios")'
```

se pierden los elementos 5, 6 y 7 que teníamos asignados.

### Borrar elementos de una lista

Podemos borrar una entrada de una lista usando **unset** sobre ese índice:

```
bash$  uns et C[2]
bash$  declare -p C
declare -a C='([0]="Hola" [1]="Que tal")'
```

O bien borrar todo la lista usando **unset** sobre el nombre del array:

```
bash$  uns et C
bash$  declare -p C
bash: declare: C: not found
```

Obsérvese que este comportamiento es distinto al de asignar un valor al nombre de la lista, que lo que hace es asignárselo al elemento de índice 0.

### Opción -a del comando read

A las opciones del comando **read** vistas en el guión 3 podemos añadir la opción *-a*. Ésta permite leer datos por teclado y almacenarlos como elementos de un array.

```
bash$  read – a f rase
Hola que tal
bash$  declare –p fr  ase
declare –a frase='([0]="Hola" [1]="que" [2]="tal")'
```

## 5.5.- Funciones

Como en casi todo lenguaje de programación, podemos utilizar funciones para agrupar trozos de códigos de una manera más lógica, o practicar el divino arte de la recursividad.

Las funciones de Bash son una extensión de las funciones que existen desde el Bourne Shell.

Éstas, a diferencia de los scripts, se ejecutan dentro de la memoria del propio proceso de Bash, con lo que son más eficientes que ejecutar scripts aparte, pero tienen el inconveniente de que tienen que estar siempre cargadas en la memoria del proceso para poder usarse. Actualmente, debido a la gran cantidad de memoria que tienen los ordenadores, el tener funciones cargadas en la memoria de tiene un coste insignificante, con lo que no importa tener cargadas gran cantidad de éstas en el entorno.

### Definición de funciones

Antes de utilizar una función, es preciso definirla. La forma de definir una función en bash es:

```
nombre_función ( )
{
  comandos bash
}
```

Para borrar una función podemos usar el comando **unset** con la siguiente sintaxis

```
unset -f nombre_función.
```

Cuando definimos una función, se almacena como una variable de entorno. Para ejecutar la función, simplemente escribimos su nombre seguido de argumentos, como cuando ejecutamos un comando. Los argumentos actúan como parámetros de la función.

Podemos ver que funciones tenemos definidas en una sesión usando el comando **declare -f**. El shell imprime las funciones, y su definición, ordenadas alfabéticamente. Si preferimos obtener sólo el nombre de las funciones, podemos usar el comando **declare -F.**

Si una función tiene el mismo nombre que un script o ejecutable, la función tiene preferencia: Esta regla se ha usado muchas veces para, engañando a los usuarios, violar la integridad de su sistema.

Un ejemplo sencillo del uso de funcione es el siguiente script. En él se crean dos funciones, una para salir y otra para mostrar el mensaje hola.

```
#!/bin/bash
#Definimos dos funciones
salir (){
   exit
}
hola () {
   echo < Hola!
}
#Comienza el script
hola   #Llamamos a la funcion hola
salir    #Llamamos a la funcion salir
echo "Fin del script"
```

Las funciones no necesitan ser declaradas en un orden específico. El script al ejecutarse llama primero a la función *hola*, mostrando el mensaje de "Hola!", a continuación llama a la función *salir* y termina el script. El programa nunca llegara a la última línea.

El siguiente script es prácticamente idéntico al anterior. La diferencia está en que la función *imprime* muestra un mensaje que le pasemos como argumento. Los argumentos, dentro de las funciones, son tratados de la misma manera que los argumentos suministrados al script.

```
#!/bin/bash
#Definimos dos funciones
salir () {
   exit
}
imprime (){
   echo $1
}
#Comienza el script
imprime Hola    #Llamamos a la función e con el argumento Hola
imprime Adios  #Llamamos a la función e con el argumento Adios
salir           #Llamamos a la función salir
echo "Fin del script"
```

## 5.6.- Operadores de cadena

Los operadores de cadena nos permiten manipular cadenas sin necesidad de escribir complicadas rutinas de procesamiento de texto. En particular, los operadores de cadena nos permiten realizar las siguientes operaciones:

- Asegurarnos de que una variable existe (que está definida y que no es nula).
- Asignar valores por defecto a las variables.
- Tratar errores que se producen cuando una variable no tiene un valor asignado.
- Coger o eliminar partes de la variable que cumplan un patrón.

Empezaremos viendo los operadores de sustitución, para luego ver los de búsqueda de cadenas.

### 5.6.1.- Operadores de sustitución

La Tabla 5.1 muestra los operadores de sustitución.

| *Operador* | *Descripción* |
|---|---|
| ${var:-valor} | Si `var` existe y no es nula retorna su contenido, si no, retorna `valor`. |
| ${var:+valor} | Si `var` existe y no es nula retorna `valor`, si no, |

| *Operador* | *Descripción* |
|---|---|
| | retorna una cadena nula. |
| ${var:=valor} | Si `var` existe y no es nula retorna su contenido, si no, asigna `valor` a `var` y retorna su contenido. |
| ${var:?mensaje} | Si `var` existe y no es nula retorna su contenido, si no, imprime `var:mensaje` y aborta el script que se esté ejecutando (sólo en shells no interactivos). Si omitimos `mensaje` imprime el mensaje por defecto `parameter null or not set`. |
| ${var:offset:longitud} | Retorna una subcadena de `var` que empieza en `offset` y tiene `longitud` caracteres. El primer carácter de `var` empieza en la posición 0. Si se omite `longitud` la subcadena empieza en `offset` y continuúa hasta el final de `var`. |

*Tabla 5.1: Operadores de sustitución*

Los dos puntos (:) en este operador son opcionales. Si se omiten en vez de comprobar si existe y no es nulo, sólo comprueba que exista (aunque sea nulo).

**${var:-valor}** se utiliza para retornar un valor por defecto cuando el contenido de la variable `var` está indefinido. Por ejemplo, `${nombre:-1}` devuelve 1 si el contenido de *nombre* está indefinido o es nulo.

**${var:+valor}** por contra se utiliza para comprobar que una variable tenga asignado un valor no nulo. Por ejemplo, `${nombre:+1}` retorna 1 (que se puede interpretar como verdadero) si *nombre* tiene un valor asignado.

Los dos operadores de cadena anteriores no modifican el valor de la variable **var**, simplemente devuelven su contenido, si queremos modificar **var** podemos usar **${var:=valor}** que asigna un valor a la variable si ésta está indefinida. Por ejemplo, `${nombre:="Paco"}` asigna "Paco" a la variable *nombre* si ésta no tiene valor.

También podemos querer detectar errores producidos porque una variable no tenga valor

asignado. En este caso, usamos **${var:?mensaje}** que detecta si **var** no tiene valor asignado y produce un mensaje de error. Por ejemplo, `${nombre:?'Eres una persona anónima'}` imprime *nombre: Eres una persona anónima*

Por último, podemos coger partes de una cadena usando **${var:offset:longitud}**. Por ejemplo, si *nombre* vale Fernando López, `${nombre:0:8}` devuelve Fernando y `${nombre:9}` devuelve López.

### Ejemplo

Imaginemos que tenemos un fichero con el saldo y el nombre de un conjunto de clientes con la siguiente forma:

```
bash$  cat cli entes
45340        Jose Carlos Martinez
24520        Mari Carmen Gutierrez
450          Luis Garcia Santos
44           Marcos Vallido Grandes
500          Carlos de la Fuente Lopez
```

Crear un script, de nombre *mejoresClientes.sh,* que imprima los **N** clientes que más saldo tengan. El script recibirá como primer argumento el fichero de clientes y, opcionalmente como segundo argumento, el número **N** de clientes a imprimir. Si no se proporciona **N**, por defecto será 5. La forma del comando podría ser:

```
mejoresclientes.sh fichero [cuantos]
```

Para ello podemos usar el comando **sort** el cual ordena líneas, y después el comando **head** que saca las primeras líneas de la forma:

```
sort -nr $1 | head -${2:-5}
```

La opción **-n** dice a **sort** que ordene numéricamente (no alfabéticamente) y la opción **-r** dice a **sort** que saque los elementos ordenados de mayor a menor. **head** recibe como argumento el número de líneas a mostrar, por ejemplo `head -2` significa coger las primeras 2 líneas. Si $2 es nulo se toma por defecto 5.

Este script, aunque funciona, es un poco críptico; vamos a hacer unos cambios con vistas a mejorar su legibilidad. Por un lado, vamos a poner comentarios (precedidos por #) al principio del fichero, y vamos a usar variables temporales para mejorar la legibilidad del programa. El resultado es el siguiente:

```
# Script que saca los mejores clientes
# Tiene la forma:
#           mejoresclientes.sh <fichero> [<cuantos>]
fichero=$1
cuantos=$2
defecto=5
sort -nr $fichero | head -${cuantos:-$defecto}
```

Estos cambios que hemos hecho mejoran la legibilidad del script, pero no su tolerancia a errores, ya que si por ejemplo no pasamos el argumento **$1**, con el nombre del fichero, se ejecutará el script así:

```
sort -nr | head -$5
```

y **sort** se quedará esperando a que entren datos por su entrada estándar hasta que el usuario pulse Ctrl+D o Ctrl+C. Esto se debe a que, aunque controlamos que el segundo argumento no sea nulo, no controlamos el primero.

En este caso podemos usar el operador de cadena **:?** para dar un mensaje de error cambiando:

```
fichero=$1
```

Por:

```
fichero=${1:?'no suministrado'}
```

Ahora, si no suministramos el argumento se produce el mensaje:

```
./mejoresclientes: 1: no suministrado
```

Podemos mejorar la legibilidad de este mensaje si hacemos:

```
fichero_clientes=$1
fichero_clientes=${fichero_clientes:?'no suministrado'}
```

Ahora si olvidamos proporcionar el argumento vemos el mensaje:

```
./mejoresclientes.sh: fichero_clientes: no suministrado
```

Para ver un ejemplo del operador de cadena **:=** podríamos cambiar:

```
cuantos=$2
```

Por:

```
cuantos=${2:=5}
```

Pero esto no funciona porque estamos intentando asignar un valor a un parámetros posicional (que son de sólo lectura). Lo que sí podemos hacer es:

```
cuantos=$2
.........
sort -nr $fichero_clientes | head -${cuantos:=$defecto}
```

El script más tolerante a fallos sería el siguiente:

```
# Script que saca los mejores clientes
# Tiene la forma
#      mejoresclientes <fichero> [<cuantos>]
fichero_clientes=$1
fichero_clientes=${fichero_clientes:?'no suministrado'}
cuantos=$2
defecto=5
sort -nr $fichero_clientes | head -${cuantos:=$defecto}
```

### 5.6.2.- Operadores de búsqueda de patrones

En la Tabla 5.2, se muestran los operadores de búsqueda de patrones existentes y una descripción de su comportamiento.

| *Operador* | *Descripción* |
|---|---|
| `${var#patron}` | Si **patron** coincide con el inicio del valor de **var,** borra la parte más pequeña que coincide y retorna el resto. |
| `${var##patron}` | Si **patron** coincide con el inicio del valor de **var**, borra la parte más grande que coincide y retorna el resto. |
| `${var%patron}` | Si **patron** coincide con el final del valor de **var**, borra la parte más pequeña que coincide y |

| *Operador* | *Descripción* |
|---|---|
| | retorna el resto. |
| `${var%%patron}` | Si **patron** coincide con el final del valor de **var**, borra la parte más grande que coincide y retorna el resto. |
| `${var/patron/cadena}`<br>`${var//patron/cadena}` | La parte más grande de **patron** que coincide en **var** es reemplazada por **cadena**. La primera forma sólo reemplaza la primera ocurrencia, y la segunda forma reemplaza todas las ocurrencias. Si **patron** empieza por **#** debe producirse la coincidencia al principio de **var**, y si empieza por **%** debe producirse la coincidencia al final de **var**. Si **cadena** es nula, se borran las ocurrencias. En ningún caso **var** se modifica, sólo se retorna su valor con modificaciones. |

*Tabla 5.2: Operadores de búsqueda de patrones*

Un uso frecuente de los operadores de búsqueda de patrones es eliminar partes de la ruta de un fichero, como pueda ser el directorio o el nombre del fichero. Algunos ejemplos de cómo funcionan estos operadores son:

Supongamos que tenemos la variable *ruta* cuyo valor es:

```
ruta=/usr/local/share/qemu/bios.core.bin
```

Si ejecutamos los siguientes operadores de búsqueda de patrones, entonces los resultados serían los siguientes:

| **Operador** | **Resultado** |
|---|---|
| `${ruta##/*/}` | `bios.core.bin` |
| `${ruta#/*/}` | `local/share/qemu/bios.core.bin` |
| `${ruta%.*}` | `/usr/local/share/qemu/bios.core` |
| `${ruta%%.*}` | `/usr/local/share/qemu/bios` |

En la búsqueda de patrones se pueden usar tanto los comodines tradicionales como los extendidos.

### Ejemplo

En el mundo de GNU existen unas herramientas llamadas NetPBM1 que permiten convertir entre muchos formatos gráficos. La herramienta suele convertir de formatos conocidos (gif, bmp, jpg) a un formato interno, o bien del formato interno a los formatos conocidos. Los formatos internos que utiliza son **.ppm** (Portable Pixel Map) para imágenes en color, .**pgm** (Portable Gray Map) para imágenes en escala de grises, y **.pbm** (Portable Bit Map) para imágenes formadas por bits de blanco y negro. A veces estos formatos aparecen bajo la extensión general **.pnm**, que abarca a todos ellos. Nuestro objetivo es hacer un script llamado **bmptojpg** que reciba uno o dos nombres de fichero, el primero de tipo .**bmp** y el segundo de tipo **.jpg**. Si no se suministra el segundo argumento, el nombre del fichero será el mismo que el del primer argumento pero con la extensión **.jpg**. Para realizar las conversiones usaremos los comandos de NetPBM **bmptoppm** y **ppmtojpeg**.

Los comandos reciben como argumento el fichero origen y emiten por la salida estándar el fichero en el formato destino.

Para obtener el primer argumento, o dar error si no se nos suministra, podemos usar el operador **:?** así:

```
fichero_entrada=${1:?'falta argumento'}
```

Para obtener los nombres de fichero intermedio y de salida usamos:

```
fichero_intermedio=${fichero_entrada%.bmp}.ppm
fichero_salida=${2:-${fichero_intermedio%.ppm}.jpg}
```

Obsérvese que para el nombre de salida usamos el operador :- para que si no se ha suministrado el segundo argumento usemos el nombre del fichero de entrada, pero con la extensión cambiada.

Luego el script que realiza esta operación es el siguiente:

```
# Convierte un .bmp en un .jpg
fichero_entrada=${1:?'falta argumento'}
fichero_intermedio=${fichero_entrada%.bmp}.ppm
fichero_salida=${2:-${fichero_intermedio%.ppm}.jpg}
bmptoppm $fichero_entrada > $fichero_intermedio
ppmtojpeg $fichero_intermedio > $fichero_salida
```

### 5.6.3.- El operador longitud

El operador longitud nos permite obtener la longitud (en caracteres) del valor de una variable. Tiene la forma **${#var}** donde **var** es la variable a cuyo contenido queremos medir la longitud. Por ejemplo, si la variable *nombre* vale Fernando, **${#nombre}** devolverá 8.

## 5.7.- Sustitución de comandos

La sustitución de comandos permite usar la salida de un comando como si fuera el valor de una variable.

La sintaxis de la sustitución de comandos es:

```
$(comando)
```

También se puede hacer utilizando las comillas francesas, aunque esta nueva forma permite anidar sustituciones de comandos.

Un ejemplo de uso de la sustitución de comandos es **$(pwd)**, que nos devuelve el directorio actual, y es equivalente a leer la variable de entorno **$PWD**.

Otro ejemplo es el uso de **$(ls $HOME)**, esta sustitución de comandos nos devuelve una variable con todos los ficheros del directorio home:

```
bash$  midir=$ (ls  $HOME)
bash$  echo $mi  dir
Desktop Documents Library Movies Music Pictures Public
Sites autor jdevhome tmp xcode
```

También podemos cargar el contenido de un fichero en una variable de entorno usando **$(<fichero)**, donde fichero es el fichero a cargar. Por ejemplo, para cargar el contenido del fichero *copyright.txt* en la variable *copyright* hacemos:

```
bash$  copyright=$(< copyright.tx t)
```

Si tenemos un conjunto de ficheros de la forma *tema*.txt,* podemos usar la sustitución de comandos para abrir todos los que traten de Bash así:

```
bash$  nedit  $(g rep  -l  'bas  h'  t ema*.txt  )
```

La opción -l hace que **grep** devuelva sólo los nombres de fichero donde ha encontrado la palabra 'bash'.

## 5.8.- Los ficheros de configuración de Bash

Cada vez que entramos a nuestra cuenta se ejecuta el contenido del fichero **/etc/profile**, y luego se mira a ver si en el directorio **home** existe el fichero .**bash_profile**, de ser así se ejecuta su contenido para personalizar aspectos de nuestra cuenta.

Cualquier configuración que añadamos a **.bash_profile** no será efectiva hasta que salgamos de la cuenta y volvamos a conectarnos, si hacemos cambios en este fichero y queremos verlos sin salir de la cuenta podemos usar el comando **source**, el cual ejecuta el contenido del fichero que le digamos:

```
bash$  sour ce  .ba sh_pr ofile
```

Alternativamente al comando **source** está el comando punto (**.**), con lo que el contenido de .**bash_profile** también se puede ejecutar así:

```
bash$  . .bash _p rofile
```

Bash permite usar dos nombres alternativos para **.bash_profile** por razones de compatibilidad histórica: **.bash_login**, nombre derivado del fichero .login del C Shell, y **.profile** nombre usado por el Bourne Shell y el Korn Shell. En cualquier caso, sólo uno de estos ficheros será ejecutado. Si .**bash_profile** existe los demás serán ignorados, sino Bash comprueba si existe **.bash_login** y, sólo si éste tampoco existe, comprueba si existe **.profile**. La razón por la que se eligió este orden de búsqueda es que podemos almacenar en **.profile** opciones propias del Bourne Shell, y añadir opciones exclusivas de Bash en el fichero **.bash_profile** seguido del comando **source .profile** para que Bash también cargue las opciones del fichero **.profile.**

**.bash_profile** se ejecuta sólo al entrar por primera vez, si abrimos otro shell (ejecutando bash) desde la línea de comandos de Bash lo que se intenta ejecutar es el contenido de **.bashrc**. Si **.bashrc** no existe, no se ejecutan configuraciones adicionales al abrir un nuevo shell. Este esquema

nos permite separar configuraciones que se hacen una sola vez, al conectarnos, de configuraciones que se cambian cada vez que se abre un nuevo shell. Si hay configuraciones en **.bashrc** que también queremos ejecutar al conectarnos, podemos poner **source .bashrc** dentro del fichero **.bash_profile.**

Por último, el fichero **.bash_logout** es un fichero que, de existir, contiene órdenes que se ejecutarán al abandonar la cuenta, por ejemplo eliminar ficheros temporales o almacenar datos de actividad de usuarios en un fichero de log.

## 5.9.- Ejercicios

**Todos los scripts planteados deberán tener**

- **Un comentario al principio que indique lo que hace.**
- **Comentarios a lo largo del script de lo que se va haciendo**
- **En el caso de recibir parámetros, comprobación de que el numero de parámetros recibidos es correcto, mostrando en caso contrario un mensaje con la sintaxis.**

1.- Crea un script de nombre **opciones.sh** que pueda tener una de estas dos sintaxis:

```
opciones.sh -o fichero1 fichero2
opciones.sh fichero1 fichero2
```

El script, en cualquiera de los dos casos, copiará el *fichero1* en el *fichero2*, comprobando que *fichero1* existe y que *fichero2* no existe. Si se utiliza para su ejecución la primera sintaxis (con la opción -o), el script copiará *fichero1* en *fichero2* pero ordenando previamente *fichero1*. Utiliza la sentencia shift para controlar la opción.

2.- Desarrolla el script **mezclaFicheros.sh** que reciba un número indeterminado de parámetros. Los **n** primeros serán ficheros con el mismo número de líneas, y el último parámetro será el fichero de salida. El script deberá mezclar las líneas de los **n** primeros ficheros y guardarlas en el fichero de salida. El número mínimo de parámetros a procesar por el script será de 3, no estando limitado el número máximo. Si fuera invocado con menos parámetros mostrará el correspondiente mensaje de error.

3.- Crea un script, de nombre **ordenaParametros.sh**, que envíe ordenados a la salida estándar los parámetros que recibe por la línea de órdenes. El número mínimo de parámetros a procesar por el script será de 2, no estando limitado el número máximo. Si fuera invocado con menos parámetros mostrará en la salida de error estándar un mensaje en el que se indique la

forma de invocarlo.

4.- Realiza un script llamado **menuFicheros.sh** que reciba como argumento un fichero o directorio. El script, a través de un menú, realizará las siguientes operaciones sobre el argumento recibido:

a) Contar el número de ficheros que hay en el directorio. Si el argumento que se le pasa es un fichero, deberá mostrar el correspondiente mensaje de error.

b) Mostrar el tamaño del fichero

c) Crear el fichero (antes deberá asegurarse de que no existe).

d) Mostrar información sobre el fichero (nombre, permisos que tiene, propietario, ...)

5.- Crea un script de nombre **cambiaExtension.sh** que reciba como argumentos dos extensiones. El script deberá cambiar la extensión de todos los ficheros que tengan la extensión pasada como primer argumento por la extensión pasada como segundo argumento.

6.- Los directorios de búsqueda que almacena la variable de entorno PATH a veces resultan difíciles de ver debido a que cuesta encontrar el delimitador dos puntos (:). Escribe un script llamado **verpath.sh** que muestre los directorios del PATH uno en cada línea.

7.- Haz una función que reciba como parámetro el nombre de un fichero y devuelva el número de palabras que empiezan por A en él. Utiliza esta función en un script llamado **funciones.sh.** El script permitirá al usuario introducir nombres de ficheros por teclado hasta que se meta uno vacío; para cada fichero, mostrará el número de palabras que comienzan con A utilizando la función implementada anteriormente.