man es el paginador del manual del sistema.

* Control + F: avanzar página
* Control + B: retrasar página
* Control + P: moverse una línea hacia arriba
* Control + N: moverse una línea hacia abajo
* /texto : busca el texto que se escribe a continuación desde la primera línea hacia adelante, marcando las coincidencias encontradas.
* ¿texto: busca el texto que se escribe a continuación desde la pimera línea mostrada hacia delante, marcando las coincidencias encontradas
* n : siguiente elemento en la búsqueda
* N : elemento previo en la búsqueda
* v : lanza el editor por defecto para editar el fichero que estamos viendo q: sale del man. También puede ser Q, :q, :Q y zz. ls lista los contenidos de un directorio-a, –-all no ignora las entradas que empiecen con .
* -A, –-almost-all no lista los implicados en . y ..
* -b, --escape imprime C-style salidas para caracteres no graficos.
* -C lista las entradas por columnas
* -d lista los directorios en si mismos y no sus contenidos
* -l usa un formato de listado más largo
* -o como –l, pero no lista la información del grupo
* -r lista en orden inverso
* -R lista subdirectorios recursivamente, además del directorio actual
* -t muestra por tiempo de modificación, los nuevos primeros
* -x lista las entradas por líneas en lugar de por columnas
* -X lista alfabéticamente por la extensión de la entrada

cd Cambia de directorio de trabajo. Las abrevaiaciones . y .. se pueden utilizar como referencia de los directorios actual y padre, respectivamente. El símbolo

~ es el directorio HOME del usuario y el símbolo / al inicio de un camino es el directorio raíz del sistema

* -L fuerza a seguir enlaces simbólicos: resuleve enlaces simbólicos en DIR después de procesar instancias de <<..>>
* -P usa la estructura del directorio físico sin seguir enlaces simbólicos: resuelve enlaces simbólicos: resuelve enlaces simbólicos en DIR antes de procesar instancias de <<..>>
* -e si se proporciona la opción –P, y el directorio de trabajo actual no se puede determinar correctamente, sale con un estado distinto de cero
* -@ en sistemas que lo soportan, presenta un archivo con atributos extendidos como un directorio que contiene los atributos del archivo

pwd Imprime el camino absoluto del directorio actual

* -L usa pwd desde el entorno, incluso si este contiene enlaces simbólicos
* -P imprime el directorio físico, sin ningún enlace simbólico

Por defecto, pwd se comporta como si se hubiera especificado –L mkdir *directorio* Crea un directorio a partir del *nombre* dado como argumento

* -p si no existe error, crea directorios padre
* -v imprime un mensaje para cada directorio creado
* -m establece el modo de archivo (como en chmod)

rmdir *directorio* borra un directorio existente (si está vacío)

* -p elimina un directorio y sus ancestros; ejemplo ‘rmdir –p a/b/c ‘ es similar a ‘rmdir a/b/c a/b a’
* -v salida de un diagnostica para cada directorio procesado

cat *archivo(s)* orden multipropósito: muestra el contenido de un archivo o varios, concatena archivos, copia un archivo, crea un archivo de texto o muestra caracteres invisibles de control

* -A muestra todo
* -b numera las líneas no vacías de salida, se sobrepone a –n
* -e equivalente a –vE
* -E muestra $ al final de cada línea
* -n numera todas las líneas de salida
* -s suprime los líneas de salidas repetidas
* -t equivalente a –vT
* -T muestra las tabulaciones como ^I

cp *archivo1 archivo2*  Copia el archivo1 en el archivo2. Si archivo2 no existe, se crea.

mv *fuente destino*  Renombra archivos (el archivo fuente puede ser archivo o directorio, al al igual que el destino) y puede mover de lugar un archivo o directorio)

* -f no pregunta antes de sobrescribir

file *archivo(s)*  Muestra el tipo de archivo dado como argumento.

More *archivo(s)* Visualiza un archivo fraccionándolo una pantalla cada vez (existen otros paginadores como *page*, *pg*, etc.). Antes de usar esta orden es conveniente usar la orden *file* para comprobar que se tratar de un archivo ASCII. (Muy similar a lo que realiza la orden cat).

* -*number* El tamaño de pantalla a utilizar, in numero de líneas
* +*number* Empieza a mostrar cada archivo desde la línea indicada en numero.

+/*string* La cadena a ser buscada en cada archivo antes de empezar a mostrarlo

rm *directorio\_archivos* Borra archivos y directorios con contenido

* -f ignora los archivos y argumentos no existentes, nunca pregunta
* -i pregunta antes de cada borrado
* -I pregunta una vez antes de remover mas de tres archivos, o cuando se borra re cursivamente; menos intrusivo que –i, mientras que sigue dando protección contra muchos errores.
* -r, -R elimina directorios y sus contenidos recursivamente
* -d elimina directorios vacíos

touch *archivo(s)* Si existen los archivos dados como argumentos se modifican su fecha y hora. En caso contrario, se crean con la fecha actual del sistema.

* -a cambia solo la hora de acceso
* -C no crea ningún archivo
* -m cambia solo la hora de modificación

clear Borra el contenido del terminal actual

tail *[archivos(s)]* Muestra la parte final del contenido de un archivo dado como argu- mento. Por defecto muestra 10 líneas

* -n sale el numero de líneas indicado, en lugar de las diez por defecto
* -c sale el numero de bytes indicado.

head *[archivo(s)]* Muestra la parte inicial del contenido de un archivo dado como ar- gumento. Por defecto muestra diez líneas.

* -c sale el numero de bytes indicado, en lugar de las diez por defecto
* -n sale el numero de líneas en lugar de las diez por defecto

*sort [archivo(s)]* Ordena, según un criterio elegido, el contenido de los archivos dados como argumentos.

* -b ignore los espacios en blanco
* -d considera solo los espacios en blanco y los caracteres alfanuméricos
* -f casos bajos a casos mayores de caracteres
* -g compara en función del valor general numérico
* -i considera solo caracteres que se puedan mostrar Campos informativos:**Tipo de objeto:** Un carácter indica el tipo de objeto que represente esa entrada: un – (guión) indica que es un archivo plano o regular; una d que es un directorio; la c indica que es un archivo de dispositivo orientado a carácter; la b, dispositivo orientado a bloques; la letra l, archivo de tipo enlace simbólic.
* **Bits de protección:** Parte de la protección en Linux descansa en la protección de los archivos. Cada archivo tiene asociados 12 bits de protección que indican qué operaciones podemos realizar sobre él y quien puede hacerlas. La orden ls muestra sólo 9 bits a través de una cadena de la forma genérica rwxrwxrwx donde cada letra representa un bit (un permiso). La pertenencia a un grupo la establece el administrador del sistema cuando lo crea.

# Permisos:

* r lectura (read) Se puede listar su contenido
* w escritura (write) Podemos modificarlo
* x ejecución (execute) Podemos acceder a él
* - no hay permiso

# Grupos de permisos:

* Propietario del archivo (user)
* Grupo de usuarios (group)
* Resto de usuarios (others)

Si tras intentar ejecutar un ejecutable con la orden cat y la visualización del mismo deja el terminal con caracteres extraños, entonces empleamos la orden **setterm –r**.

# Metacaracteres de archivo

? Representa cualquier carácter simple en la posición en la que se indique

* Representa cualquier secuencia de cero o más caracteres

[] Designan un carácter o rango de caracteres que representan un carácter simple a través de una lista de caracteres o mediante un rango, en cuyo caso, mostramos el primer y último carácter del rango separados por un guión “-“.

{} Sustituyen conjuntos de palabras separadas por comas que comparten partes comune

~ Se usa para abreviar el camino absoluto (path) del directorio HOME

# Modificación de los permisos de acceso a archivos.

* Simbólico, para poder cambiar uno o varios bits de protección sin modificar el resto
* Absoluto, cambia todos los permisos, expresándolos como tres cifras en base 8 (octales). En este caso, los valores tanto para el propietario, grupo o resto de usuarios oscilan entre el 0 (ningún privilegio) hasta el 7 (todos los privilegios). Si dicho valor se codifica en binario, se necesitan tres bits que se corresponden con cada permiso (lectura, escritura y ejecución):

En el modo simbólico, se debe indicar primero a qué grupo de usuarios se va a aplicar el cambio con una letra minúscula:

* u: propiertario
* g: grupo
* o: resto de usuarios
* a: todos los grupos de usuarios

Se debe indicar si va a permitir el acceso **+** o se va a denegar el acceso  **-**. Por último, se indica qué tipo de permiso es el que estamos modificando **r,w,x** y el archivo al que se le van a modificar los permisos.

chmod con esta orden se modifican los permisos que tiene el usuario, el grupo o el resto de usuarios sobre un archivo, añadiendo o quitando permisos.

Se pueden combinar las letras para que se aplique la modificación de permisos en varios grupos , o se modifique varios permisos a la vez

# Metacarecteres de redirección

**< nombre**  Redirecciona la entrada de una orden para que la obtenga el archivo *nombre.*

* **nombre** Redirige la salida de una orden para que la escriba en el archivo *nombre.* Si dicho archivo ya existe, lo sobreescribe.

**&> nombre**  La salida estándar se combina con la salida de error estándar y ambas se escriben en el archivo *nombre.*

**>> nombre** Funciona igual que le metacarácter “>” pero añade la salida estándar al final del contenido del archivo *nombre.*

**&>> nombre** Igual que el metacarácter “&>”, pero añadiendo las dos salidas combinadas al final del archivo *nombre.*

**2> nombre**  Redirige la salida de error estándar a un archivo (sólo funciona en shells se “bash”).

| Crea un cauce entre dos órdenes. La salida de una de ellas se utiliza como entrada de la otra.

**|&** Crea un cauce entre dos órdenes utilizando las dos salidas (estándar y error) de una de ellas como entrada de la otra.

## Metacaracteres sintácticos

**;** Separador entre órdenes que se ejecutan secuencialmente.

**()** Se usan para aislar órdenes separadas por “;” o por “|”. Las órdenes dentro de los paréntesis son tratadas como una única orden.

**&&** Separador entre órdenes, en la que la orden que sigue al metacarácter “&&” se ejecuta sólo si la orden precedente ha tenido éxito (no ha habido errores).

**||** Separador entre órdenes, en la que la orden que sigue al metacarácter “||” se ejecuta sólo si la orden precedente falla.

## Variables.

El bash contempla dos tipos de variables. Las **variables de entrono** o variables globales son aquellas que son comunes a todos los shells. Se pueden visualizar con env o printenv. Por convenio, se usan letras mayúsculas para los nombres de dichas variables.

Las **variables locales,** son sólo visibles en el Shell donde se definen y les da valor. Para ver las variables locales se puede usar la orden set.

## Variables según su contenido.

* Cadenas: su valor es una secuencia de caracteres.
* Números: se podrán usar en operaciones aritméticas.
* Constantes: su valor no puede ser alterado.
* Vectores o arrays: conjunto de elementos a los cuales se puede acceder mediante un índice. Normalmente el índice es un número entero y el primer elemento es el 0.

## Variables importantes.

**$BASH** Contiene la ruta de acceso completa usada para ejecutar la instancia actual de bash

**$HOME**  Almacena el directorio raíz del usuario; se puede emplear junto con la orden cd sin argumentos para ir al directorio raíz del usuario.

**$PATH** Guarda el camino de búsqueda de las órdenes, este camino está formado por una lista de todos los directorios en los que queremos buscar una orden.

**$?** Contiene el código de retorno de la última orden ejecutada, bien sea una instrucción o un guión.

Para borrar una variable se usa la orden **unset** junto con el nombre de la variable (o variables) a eliminar.

Si deseamos crear una variable con ciertos atributos, utilizaremos la orden  **declare**, cuya sintaxis completa se puede ver con help declare. Podemos indicar que una variable es numérica con la opción –i y ver los atributos con la opción –p

**declare** Sirve para crear una variable con ciertos atributos.

* -a para hacer matrices indexadas de NOMBREs
* -A para hacer matrices asociativas de NOMBREs
* -i para hacer que NOMBREs tenga el atributo <<integer>>
* -l para convertir NOMBREs a minúsculas al asignar
* -n hacer NOMBRE una referencia de la variable nombrada por su valor
* -p para ver los atributos de una variable establecida previamente
* -r para hacer NOMBREs de solo lectura
* -t para hacer que NOMBRESs tenga el atributo <<trace>>
* -u para convertir NOMBREs a mayúsculas al asignar
* -x para hacer exportación de NOMBREs

## Exportar variables

Para que las variables locales sean visibles en otros shell debemos exportar su valor para que el sistema lo reconozca, sino las variables tomaran su valor original al terminar el guion o cambiar de shell

### Para ello utilizamos la orden **export**

Con la orden export también se pueden declarar variables al mismo tiempo que las exportarmos a otros shells. **export variable=valor.**

## Significado de las diferentes comillas en las órdenes.

La sustitución de ordenes permite la ejecución de una orden, con o sin argumentos, de forma que su salida se trata como si fuese el valor de una variable. Se puede hacer poniendo **$(orden *argumentos*)**, o usando los apóstrofos inversos, **‘orden *argumentos*‘.**

Ejemplos:

* $ echo “Los archivos que hay en el directorio son: **$(ls –l)**”
* $ echo “Los archivos que hay en el directorio son: **‘ls –l‘**”

En los casos anteriores, las comillas dobles se utilizan como mecanismo de **acotación débil** para proteger cadenas desactivando el significado de los caracteres especiales que hayan entre ellas, salvo los caracteres **!**, **$**, **\** y **‘** , que no quedan protegidos. También se pueden proteger cadenas usando comillas simples como mecanismo de **acotación fuerte**, aunque en este caso se protegen los caracteres especiales salvo **!**; en consecuencia, las comillas simples serán útiles cuando se quiera proteger una variable o orden.

## Asignación de resultados de órdenes a variables.

Podemos asignar el resultado de una orden a una variable a través del operador **‘. variable=‘orden‘**

Orden expr para operaciones matemáticas. Hay que fijarse en que la orden **expr** y sus argumentos se encuentran entre apóstrofes inversos, de tal forma que a la variable *numero* no se asigne la palabra expr, sino el resultado de la orden expr.

## La orden empotrada printf.

**printf** formato [argumentos]

Donde formato es una cadena que describe cómo se deben imprimir los elementos del mensaje.

* \b espacio atrás
* \n nueva línea
* \t tabulador
* \’ carácter comilla simple
* \\ barra invetida
* \0n n = número en octal que representa un carácter ASCII de 8 bits.

Codigo de formato:

* %d un número con signo
* %f un número en coma flotante (decimal) sin notación exponenial
* %q entrecomilla una cadena
* %s muestra una cadena sin estrecomillar
* %x muestra un número en hexadecimal
* %o muestra un número en octal

## Alias

Se crean con la orden empotrada alias y se eliminan con la orden unalias. La orden la definimos entre comillas simples ‘.

### Ejemplo

$ alias dir=’ls –l’

$ dir

Para ignorar un alias y ejecutar la orden original se antepone barra invertida **\** al nombre del alias.

$ \ls –l $ HOME

#### Órdenes de búsqueda: find y grep, egrep, fgrep

#### Orden find

Se utiliza para buscar por la estructura de directorios los archivos que satisfagan los criterios especificados.

***find*** *lista-de-directorios [expresiones]*

En las expresiones es posible utilizar los metacaracteres de archivo dados en la práctica nº2.

Los criterios se especifican mediante una palabra precedida por un guion, seguida de un espacio y por una palabra o número entero precedido o no por un + o un -.

* -name seguida por el nombre deseado
* -atime 7 busca los archivos a los que se accedió hace 7 días
* -atime -2 busca los archivos a los que se accedió hace menos de 2 días
* -atime +5 busca los archivos a los que se accedió hace más de 5 días
* -type f busca los archivos regulares
* -type d busca directorios
* -size (+ o -) busca los archivos con más o menos de los bloques establecidos. Si el numero va seguido de la letra cien el tamaño dado es en bytes
* -user busca los archivos que pertenezcan al usuario
* -o con este operador se conectan dos expresiones y se seleccionarán aquellos archivos que cumplan una de las dos expresiones
* -print visualiza los nombres de camino de cada archivo que se adapta al criterio de búsqueda. Es la opción por defecto.
* -exec permite añadir una orden que se aplicará a los archivos localizados. La orden se situará a continuación de la opción y debe terminarse con un espacio, un carácter \ y a continuación un ;. Se utiliza {} para representar el nombre de archivos localizados.
* -ok es similar a –exec, con la excepción de que solicita confirmación en cada archivo localizado antes de ejecutar la orden.

#### Órdenes grep, egrep y fgrep.

La orden permite buscar cadenas en archivos utilizando patrones para especificas dicha cadena. Esta orden lee de la entrada estándar o de una lista de archivos especificados como argumentos y escribe en la salida estándar aquellas líneas que contengan la cadena. Su formato es:

***grep*** *opciones patrón archivos*

Algunas opciones que se pueden utilizar con la orden **grep** son:

* -x localiza líneas que coincidan totalmente, desde el prinicpio hasta el final de la línea, con el patrón especificado.
* -v selecciona todas las líneas que no contengan el patrón especificado
* -c produce solamente un recuento de las líneas coincidentes
* -i ignora las distinciones entre mayúsculas y minúsculas
* -n añade el número de línea en el archivo fuente a la salida de las coincidencias.
* -l selecciona sólo los nombres de aquellos archivos que coincidan con el patrón búsqueda
* -e especial para el uso de múltiples patrones e incluso si el patrón comienza por el carácter (-)

Variaciones: **fgrep** acepta sólo una cadena simple de búsqueda en vez de una expresión regular. La orden **egrep** permite un conjunto más complejo de operadores en expresiones regulares. Usando man comprueba las diferencias entre estas tres órdenes.

##### Guiones

Un guion del shell es un archivo de texto que contiene órdenes del shell y del sistema operativo. Este archivo es utilizado por shell como guía para saber que órdenes ejecutar.

$0 Nombre del guión o scrit que se ha llamado. Sólo se emplea dentro del guion

$1…$9 Son los distintos argumentos que se pueden facilitar al llamar a un guion. Los nueve primero se referencian con $1, $2, …, $9, y a partir de ahí es necesario encerrar el número entre llaves, es decir, ${n}, para n>9

$\* Contiene todos los argumentos que se le han dado. Cuando va entre comillas dobles es equivalente a “$1 $2 $3 … $n”:

$@ Contiene todos los argumentos que se le han dado. Cuando va entre comillas dobles es equivalente a “$1” “$2” … “$n”.

$# Contiene el número de argumentos que se han pasado al llamar al guion.

${arg: -val} Si el argumento tiene valor y es no nulo, continua con su valor, en caso contrario se le asigna el valor indicado por val.

${arg:?val} Si el argumento tiene valor y es no nulo, sustituye a su valor; en caso contrario imprime el valor de val y sale del guion. Si val es omitida, imprime un mensaje indicando que el argumento es nulo o no está asignado.

Las variables numéricas nos permiten pasar argumentos a un guion para adaptar su comportamiento, son los parámetros del guion.

Podemos depurar un guion con la orden bash y empleando las siguientes opciones:

* -n chequea errores sintácticos pero sin ejecutar el guion
* -v visualiza cada orden del guion antes de ejecutarla
* -x actúa igual que –v sólo que sustituyendo, en su caso, las variables por los valores que tienen en ese instante.

##### Normas de estilo

Es buena costumbre comentar los guiones para conocer siempre quien lo ha escrito, en qué fecha, qué hace, etc. Para ello, utilizaremos el símbolo **#**, bien al inicio de una línea o bien tras una orden.

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

**# Titulo: prueba**

**# Fecha: 20/10/2018**

**# Autor: Daniel Monjas Miguélez**

**# Versión: 1.0**

**# Descripción: Muestra de las normas de estilo necesarias para un guión**

**# Opciones: Ninguna**

**# Uso: Mostrar las normas de estilo**

##### Expresiones con variables

El shell bash ofrece dos posibles sintaxis para manejar expresiones aritméticas haciendo uso de lo que se denomina *expansión aritmética,* o *sustitución aritméticas,* que evalúa una expresión aritmética y sustituye el resultado de la expresión en el lugar donde se utiliza. Ambas posibilidades son:

**$(( … ))**

**$[ … ]**

En estos casos, lo que se ponga en lugar de los puntos suspensivos se interpretará como una expresión aritmética, no siendo necesario dejar huecos entre los paréntesis más internos y la expresión contenida en ellos, ni entre los corchetes y la expresión que contengan. Además hay que tener en cuenta que las variables que se usen en una expresión aritmética no necesitan ir precedidas del símbolo **$** para ser sustituidas por su valor, aunque si lo llevan no será causa de error, y que cualquier expresión aritmética puede contener otras expresiones aritméticas, es decir, las expresiones aritméticas se pueden anidar.

##### Orden date.

**Date** permite consultar o establecer la fecha y la hora del sistema

* +a% muestra el dia de la semana abreviado
* +A% muestra el dia de la semana completo
* +%b muestra el mes del año abreviado
* +%B muestra el mes del año completo
* +%c muestra la fecha y hora local
* +%C muestra el siglo actual
* +%d muestra el dia del mes
* +%D lo mismo que +%m%+%d%+%y
* +%F fecha compelta +%Y-+%m-+%d

##### Operadores aritméticos

+ - Suma y resta, o más unario y menos unario.

\* / % Multiplicación, división (truncando decimales), y resto de la división.

\*\* Potencia

++ Incremento en una unidad. Puede ir como prefijo o sufijo de una variable: si se usa como prefijo (++variable), primero se incrementa la variable y luego se hace lo que se desee con ella; si se utiliza como sufijo (variable++), primero se hace lo que se desee con la variable y después se incrementa su valor.

* Decremento en una unidad. Actúa de forma análoga al caso anterior, pudiendo usarse como prefijo o como sufijo de una variable (--variable o variable--).

() Agrupación para evaluar conjuntamente; permite indicar el orden en el que se evaluarán las subexpresiones o partes de una expresión.

, Separador entre expresiones con evaluación secuencial.

= x=expresión, asigna a x el resultado de evaluar la expresión (no puede haber huecos en blanco a los dos lados del símbolo “=”)

**+=**; **-=**; **\*=**; **/=**; **%=**

* x+=y equivale x=x+y; x-=y equivale a x=x-y;
* x\*=y equivale x=x\*y; x/=y equivale a x=x/y;
* x%=y equivale x=x%y;

En el resultado del cálculo de expresiones aritméticas, bash solamente trabaja con números enteros, por lo que si se necesitase calcular un resultado con decimales, habría que utilizar una forma alternativa, como puede ser la ofrecida por la orden **bc,** cuya opción –l, permite hacer algunos cálculo matemáticos (admite otras posibilidades que pueden verse mediante man).

##### Asignación y variables aritméticas.

Otra forma de asignar valor a una variable entera es utilizar la orden **let** de la shell bash. Aunque esta orden se usa para evaluar expresiones aritméticas, en su forma más habitual su sintaxis es:

**let** variableEntera=expresión

Hemos de indicar que (( <expresión> )) equivale a la orden **let** y presenta ventaja como por ejemplo a la hora de hacer comparaciones numérica para usarlas en ejecuciones condicionales

##### Operadores relacionales

A veces es necesario poder relacionar dos expresiones aritméticas, A y B, o negar una expresión aritmética, de forma que se pueda evaluar si se da o no cierta relación. La evaluación de una relación entre expresiones tomará finalmente un valor numérico, de manera que el 1 representa una evaluación “verdadera” (true), mientras que el 0 indica que la evaluación ha sido “falsa”(false). Observe que esta forma de evaluar resulta un poco discordante respecto a lo que sucede cuando se evalúa la variable $? que se mencionaba en l practica anterior.

##### Operadores.

* **A = B; A == B; A –eq B** A es igual que B
* **A ¡= B; A –ne B** A es distinta de B
* **A < B; A –lt B** A es menor que B
* **A > B; A –gt B** A es mayor que B
* **A <= B; A –le B** A es menor o igual que B
* **A >= B; A –ge B** A es mayor o igual que B
* **¡ A** A es falsa; representa al operador NOT (negación lógica):
* **A && B** A es verdadera y B es verdadera; es el operador AND (conjunción lógica).
* **A || B** A es verdadera o B es verdadera; es el operador OR (disyunción lógica).

##### Operadores de consulta de archivos.

A veces es necesario conocer características específicas de los archivos o directorios para saber cómo tratarlos.

Para aplicar los operadores de consulta de archivos haremos uso de dos órdenes nuevas, **test** e **if**

La sintaxis de la orden **test** es:

***test*** *expresión*

Esta orden evalúa una expresión condicional y da como salida el estado 0, en caso de que expresión se haya evaluado como verdadera, o el estado 1, si la evaluación ha resultado false o se le dio algún argumento no válido.

**test** evalúa una expresión condicional.

* -b *archivo existe y es un dispositivo de bloques*
* *-c archivo existe y es un dispositivo de caracteres*
* *-d archivo existe y es un directorio*
* *-e archivo existe*
* *-f archivo existe y es un archivo plano o regular*
* *-G archivo existe y es propiedad del mismo grupo del usuario*
* *-h archivo existe y es un enlace simbólico*
* *-L archivo existe y es un enlace simbólico. Es igual que –h*
* *-O archivo existe y es propiedad del usuario.*
* *-r archivo existe y el usuario tiene permiso de lectura sobre él*
* *-s archivo existe y es no vacío*
* *-w archivo existe y el usuario tiene permiso de escritura sobre él*
* *-x archivo existe y el usuario tiene permiso de ejecución sobre él, o es un directorio y el usuario tiene permiso de búsqueda en él*
* *Archivo1 –nt archivo2 archivo1 es más reciente que archivo2, según la fecha de modificación, o si archivo1 existe y archivo2 no*
* *Archivo1 –ot archivo2 archivo1 es más antiguo que archivo2, según la fecha de modificación, o si archivo2 existe y archivo1 no.*
* *Archivo1 –ef archivo2 archivo1 es un enlace duro al archivo2, es decir, si ambos se refieren a los mismo números de dispositivo e inode.*

##### Orden if / else

La orden if / else ejecuta una lista de declaraciones dependiendo de si se cumple o no cierta condición, y se podrá utilizar tanto en la programación de guiones, como en expresiones más simples.

###### Estructura de la orden **if**

**if** condición;

then

*declaraciones*

*[****elif*** *condición;*

***then*** *declaraciones ]*

###### [else

*declaraciones ]*

**fi**

La principal diferencia de este condicional respecto a otros lenguajes es que cada condición representa realmente una lista de declaraciones, con órdenes, y no una simple expresión booleana. De esta forma, como las órdenes terminan con un estado de finalización, condición se considera true si su estado de finalización es 0, y false en caso contrario. Al igual que en otros lenguajes, en cualquiera de las declaraciones puede haber otra orden if, lo que daría lugar a un anidamiento.

El funcionamiento es: se comienza haciendo la ejecución de la lista de órdenes contenidas en la primera condición; si su estado de salida es 0, entonces se ejecuta la lista de declaraciones que sigue a la palabra then y se temina la ejecución del if; si el estado de salida fuese 1, se comprueba si hay un bloque que comience por elif. En caso de haber varios bloque elif, se evalua la condición del primero de ellos de forma que si su estado de salida es 0, se hace la parte then correspondiente y el if, pero si su estado de salida es 1, se conitnúa comprobando de manera análoga el siguiente bloque elif, si es que existe. Si el estado de salida de todas las condidicones existentes es 1, se comprueba si hay un bloque else, en cuyo caso se ejecutarían las declaraciones asociadas a él, y termina el if.

La orden if puede expresarse utilizando la orden test para hacer una comprobación. De forma análoga se puede utilizar [ … ]; si se usa “if [ expresión ];” expresión puede ser una expresión booleana y puede contener órdenes, siendo necesarios los huecos en blanco entre los corchetes y expresión.

##### Comparaciones aritméticas.

La comparación entre valores aritméticos se puede realizar empleando los operadores = + -. A la hora de efectuar comparaciones con valores numéricos es posible tratar dichos valores como cadenas de caracteres o como su número.

También podemos realizar comparaciones aritméticas utilizando el doble paréntesis. Cuando se emplea el doble paréntesis, no es posible utilizar los operadores –eq, -ne, -le, -lt, -ge, y –gt. En su lugar se utilizarán los operadores equivalentes mostrados en la tabla. Además , la última comparación obliga a usar el operador de igualdad ==. Si se emplea el = responderá con un mensaje de error.

##### Comparaciones entre cadenas de caracteres.

En bash, los operadores == y ¡=, también pueden utilizarse para comparar si dos cadenas de caracteres A y B coinciden o no, respectivamente; además, en la versión 2.0 las posteriores < y > permiten comparar si una cadena de caracteres A se clasifica antes o después de otra cadena B, respectivamente, siguiendo el orden lexográfico. Además, si se deja un espacio en blanco antes y después del operador =, también se realiza la comparación en lugar de una asignación si se hace sin los pertinentes espacios en blanco.

Cuando la variable de caracteres que se desea comparar pueda contener algún espacio en blanco, es obligatorio poner la variable entre comillas para realizar correctamente la comparación (cuando la variable contiene un valor numérico o una palabra, no es necesario representarla entre comillas). En el siguiente ejemplo se ilustra tal necesidad.

##### Expresiones regulares

Una expresión regular es un patrón que describe un conjunto de cadenas y que se puede utilizar para búsquedas dentro de una cadena o archivo. Un usuario avanzado del sistema que desee obtener la máxima potencia de ciertas órdenes debe conocer y manejar expresiones regulares. Las expresiones regulares se construyen de forma similar a las expresiones aritméticas, utilizando diversos operadores para combinar expresiones sencillas. Las piezas fundamentales para la construcción de expresiones regulares son las que representan a un carácter simple. La mayoría de los caracteres, incluyendo las letras y los dígitos, se consideran expresiones regulares que se representan a sí mismos. Para comparar con el asterisco (\*), éste debe ir entre comillas simples (‘\*‘).

En las expresiones regulares se puede utilizar una barra inclinada invertida (\), denominada a veces como barra de escape, para modificar la forma en la que se interpretará el carácter que le siga. Cuando los metacaracteres “?”, “+”, “{“, “}”, “|”, “(“ y “)” aparecen en una expresión regular no tienen un significado especial, salvo que vayan precedidos de una barra de escape; si se utilizan anteponineo es barra, es decir “\?”, “\+”, “\{“, “\}”, “\|”, “(“ o “\)”, su significado será el que corresponda según la tabla.

##### Patrones en expresiones regulares

La tabla muestra patrones simples que pueden utilizarse en expresiones regulares. Una expresión regular estará formado por uno o varios de estos patrones.

Dos expresiones regulares también pueden concatenarse; la expresión que se obtiene representa a cualquier cadena formada por la concatenación de las dos subcadenas dadas por las subexpresiones concatenadas. Esto no debe confundirse con el uso del operador OR

Por defecto, las reglas de precedencia indican que primero se trata la repetición, luego la concatenación y después la alteración, aunque el uso de paréntesis permite considerar subexpresiones y cambiar esas reglas.

##### Expresiones regulares con órdenes de búsqueda.

Recordar que find sirve para buscar a nivel de características generales de los archivos, como, por ejemplo nombre o condiciones de acceso, pero sin entrar en su contenido; por el contrario, grep y egrep examinan examinan la cadena que se le dé mediante la entrada estándar o el contenido de los archivos que se le pongan como argumento.

##### Patrones:

* **\** la barra de escape; si en un patrón se quiere hacer referencia a este mismo carácter, debe ir predecido por él mismo y ambos entre comillas simples
* **.** cualquier carácter en la posición en la que se encuentre el punto cuando se usa en un patrón con otras cosas; si se usa solo, representa a cualquier cadena; si se quiere buscar un punto como parte de un patrón, debe utilizarse \. Entre comillas simples o dobles.
* **()** un grupo; los caracteres que se pongan entre los paréntesis serán considerados conjuntamente como si fuesen un único carácter.(Hay que usar \)
* **?** que el carácter o grupo al que sigue puede aparecer una vez o no aparecer ninguna vez (Hay que usar \)
* **\*** que el carácter o grupo al que sigue puede no aparecer o aparecer varias veces seguidas. (No hay que usar \)
* **+** que el carácter o grupo previo debe aparecer una o más veces seguidas
* **{n}** que el carácter o grupo previo debe aparecer exactamente n veces.(usar \)
* **{n,}** que el carácter o grupo previo debe aparecer n ves o más veces seguidas (usar \)
* **{n,m}** que el carácter o grupo previo debe aparecer de n a m veces seguidas; al menos n veces, pero no más de m veces. (usar \)
* **[]** una lista de caracteres que se tratan uno a uno como caracteres simples; si el primer carácter de la lista es “^”, entonces representa a cualquier carácter que no esté en esa lista
* **-** un rango de caracteres cuando el guion no es el primero o el último en una lista; si el guion aparece el primero o el último de la lista, entonces se trata como él mismo, no como rango; en los rangos de caracteres, el orden es el alfabético, pero intercalando minúsculas y mayúsculas – es decir: aAbB…-; en los rangos de dígitos el orden es 012… También es posible describir rangos parciales omitiendo el inicio o el final del rango.
* **^** indica el inicio de una línea; como se ha dicho anteriormente, cuando se usa al comienzo de una lista entre corchetes, representa a los caracteres que no están en la lista. Situando a continuación de ^ un carácter, filtrará todas aquellas líneas que comiencen por ese carácter.
* **$** indica el final de una línea. Situando un carácter antes del $, filtrará todas aquellas líneas que terminen por ese carácter.
* **\b** el final de una palabra. (Debe utilizarse entre comillas simples o dobles)
* **\B** que no está al final de una palabra.(Debe utilizarse entre comillas simples o dobles)
* **\<** el comienzo de una palabra.(Debe utilizarse entre comillas simples o dobles)
* **\>** el final de una palabra.(Debe utilizarse entre comillas simples o dobles)
* **\|** el operador OR para unir dos expresiones regulares, de forma que la expresión regular resultante representa a cualquier cadena que coincida con al menos una de las dos subexpresiones. (La expresión global debe ir entre comillas simples o dobles; además, cuando se usa con grep, esta orden debe ir acompañada de la opción –E)

Se utiliza la orden find con la opción –regex para especificar que se buscan nombres de archivos que satisfagan la expresión regular que se pone a continuación de esta opción.

Si no se usa esta opción se interpretarán los símbolos como metacaracteres y no como parte de una expresión regular.

##### Ordenes para guiones más complejos

* **for** ejecuta una lista de declaraciones un número fijo de veces
* **while** ejecuta una lista de declaraciones cierto número de veces mientras cierta condición se cumple
* **until** ejecuta una lista de declaraciones repetidamente hasta que se cumpla cierta condición
* **case** ejecuta una de varias lista de declaraciones dependiendo del valor de una variable

##### Lectura del teclado

En nuestro guiones podemos leer desde el teclado usando la orden **read,** que detiene la ejecución del guion y espera a que el usuario teclee algo de forma que el texto tecleado es asignado a la(s) varaible(s) que acompañan a la orden. Su sintaxis es:

*Read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-p prompt] [-t timeout] [-u fd] [name …]*

* a array asigna las palabras a índices secuenciales de la variable de matriz ARRAY, empezando por cero
* d delim continuar hasta que el primer carácter de DELIM sea leído, en vez de una nueva línea
* e usar readline para obtener la línea en un intérprete de ordenes interactivo
* i text usar TEXT como texto inicial para readline
* n nchars devolver después de leer los caracteres NCHARS en vez de esperar una nueva línea, pero incorporar un delimitador si menos de nchars caracteres se leen antes del delimitador.
* p prompt generar la cadena prompt sin arrastrar una nueva línea antes de intentar leer
* r no permitir barras invertidas para escapar caracteres
* s no repetir le entrada que viene de la terminal
* **t timeout** terminar el plazo y devolver un fallo si una línea completa de entrada no se lee en timeout segundos. El valor de la variable timeout es el tiempo de espera pre determinado. Timeout puede ser un número fraccionario. Si TIMEOUT es 0, leer devuelve inmediatamente, sin intentar leer datos, devolviendo éxito solo si hay entrada disponible en el descriptor de archivo especificado. El estado de salida es mayor que 128 si se excede el tiempo de espera
* **u fd** lees desde el descriptor de archivo FD en vez de la entrada estándar.

###### Orden for

El bucle for permite repetir cierto número de veces las declaraciones especificadas. Su sintaxis es:

for nombre [in lista]

do

*declaraciones que pueden usar $nombre*

*done*

orden **seq**, genera una secuencia de números desde el valor indicado como primer argumento, hasta el valor final dado como tercer argumento, tomando como incremento el segundo.

**Orden case**

La orden **case** compara un variable con una serie de valores o patrones. Si el valor o el patrón coinciden, se ejecutan las declaraciones correspondientes. La sintaxis para esta orden es:

*Case expresión in*

*Patron1 )*

*Declaraciones;;*

*Patron2 )*

*Declaraciones;;*

*…*

*Esac*

Cuando se escribe como pratron ‘\*’ representa cualquier otro caso distinto a los ya descritos anteriormente.

**Ordenes while y until**

Las órdenes while y until permiten repetir una serie de declaraciones. La orden while ejecuta las declaraciones comprendidas entre do y done mientras que le expresión sea true. Si la expresión falla en el primer intento, nunca se ejecutan las declaraciones del cuerpo while. La sintaxis para while es:

while expresión;

do

declaraciones …

done

Es importante que comprenda que este bucle encontrará false su condición cuando demos **control –d**.

La orden break sirve para terminar un bucle y seguir ejecutando la siguiente orden del script. La orden break puede venir seguida de un número que indica cuántos bucles anidados romper, por ejemplo, break 2.

La orden until es similar a while excepto que se repite el cuerpo hasta que la condición sea cierta(mientras que while se repite hasta que la condición sea falsa). En este caso, la condición se comprueba antes de cada iteración, incluso antes de la primera iteración. La sintaxis para until es:

*until expresión;*

*do*

*declaraciones…*

*done*

La orden sleep seguida de un numero indica al sistema el tiempo a esperar para ejecutar la siguiente orden. Por ejemplo, sleep 2, pausa el sistema durante 2 segundos.

**Funciones**

Definimos una función con la orden function y luego el nombre de la función o, del mismo modo, sin tener que escribir la palabra anterior pero añadiendo el paréntesis abierto y cerrado junto al nombre. Se puede apreciar su sintaxis a continuación:

*Function nombre\_fn {*

*Declaraciones*

*}*

*nombre\_fn() ´{*

*declaraciones*

*}*

Podemos borra una función con **unset –f nombre\_fn**. Podemos ver qué funciones tenemos definidas junto con su definición con **declare –f**, y solo su nombre con **declare –F**.

El orden de preferencia del shell a la hora de resolver un símbolo es:

1. Alias
2. Palabra clave (como if, function, etc.)
3. Funciones
4. Órdenes empotradas
5. Guiones y programas ejecutables

Como norma de estilo, podemos nombrar las funciones con un signo “\_” delante del nombre (por ejemplo, \_mifuncion).

**Variables locales en funciones. Parámetros.**

Una función puede tener variables locales. Para ello, debemos declararlas dentro de la función con el modificador **local**.

**Archivos de configuración**

Existen diferentes archivos de configuración que son leídos por shell cuando se lanza, son los archivos de arranque. Estos guiones tienen como objetivo establecer la configuración del shell definiendo variables, funciones, alias, etc.

**Caracteristicas de depuración en bash**

Un guión puede ser erróneo por errores sintácticos, porque no funciona como se desea que lo haga, o porque funcionando correctamente posee efectos secundarios indeseados. Para depura un guion defectuoso, se puede usar:

* echo en puntos críticos del guion para seguir el rastro de las variables más importantes
* las opciones –n, -v y –x de bash
* la orden trap de depuración
* funciones de “aserción”.

**Opciones de depuración en bash.**

-n Chequea errores sintácticos sin ejecutar el guion.

-v Visualiza cada orden antes de ejecutarla

-x Actúa igual que –v pero de forma más abreviada

**Realizar la traza con la orden trap.**

La orden trap sirve para especificar una acción a realizar cuando se recibe una señal. Las señales son en Linux un mecanismo de comunicación entre procesos que permite la notificación de que ha ocurrido un determinado evento (suceso) a los procesos.

Uno de los usos de esta orden es asistir a la depuración de guiones. Para ello, se invoca esta orden al comienzo del guión pasándole el argumento DEBUG. Esto hace que la acción especificada como argumento de la orden trap se ejecuta después de cada orden de un guion, cuando la opción xtrace de bash (- x) esté activada.

Un aspecto importante que se debe recordar sobre DEBUG es que no es heredado por las funciones invocadas desde el shell en el que está definido.

DEBUG es un ejemplo de lo que se denomina señales falsas que se utilizan en las declaraciones trap para que el shell actúa bajo ciertas condiciones. Éstas actúan como las señales, pero son generadas por el propio shell. Otros ejemplo **: EXIT, ERR y RETURN.** La señal falsa EXIT ejecutará su código cuando el shell en la que está activa finalice. Solamente podemos atrapar finalizaciones de un guion.

La señal ERR se activa cuando una orden devuelve un código de finalización distinto de cero. Podemos construir una función para imprimir un mensaje cuando alguna orden no finaliza correctamente.

RETURN, se utiliza cuando se regresa tras la ejecución de órdenes o guiones que se han ejecutado con source. Esta orden permite ejecutar un guion dentro de la Shell actual y no como un proceso aparte, de forma que si se crean o modifican variables permanecerán en la Shell después de la ejecución del guion.

A partir de la versión 3 de la shell de Bash, existen algunas variables que facilitan la depuración, además de las disponibles $LINENO y $FUNCNAME, como son: $BASH\_ARGC, $BASH\_ARGV, $BASH\_SOURCE, $BASH\_LINENO, $BASH\_SUBSHELL, $BASH\_EXECUTION\_STRING, y $BASH\_COMMAND. Entre las facilidades que suministra Linux para la depuración, podemos incluir la orden script que crea una copia de la sesión de un terminal en un archivo para su posterior revisión.

**Aserciones**

Una función de aserción comprueba una variable o condición en puntos críticos del guion.

**Control de trabajos en bash.**

Las órdenes que se mandan ejecutar al shell reciben el nombre de trabajos o Jobs. Un trabajo puede están en primer plano (**foreground),** en segundo plano (**background)** o suspendido (detenido). Durante una sesión, la shell almacena una lista de los trabajos no finalizados. Estos trabajos no finalizados pueden ser aquellos trabajos que se ejecutan en segundo plano o/y trabajos que han sido suspendidos. Una orden o trabajo se ejecuta en segundo plano cuando incluimos un & (**ampersand)** al final de la orden.

Para introducir una pausa se introduce la orden sleep junto con el tiempo que quiere pausar el programa.

Los trabajos se pueden manipular usando órdenes de la shell. Estas órdenes permiten hacer referencia a un trabajo de varias formas. Una forma de hacerlo es mediante el carácter ‘%’, como se muestra:

* % trabajo actual (%+ y %% son sinónimos de este especificador)
* %- trabajo previo al actual
* %n trabajo número n
* %<cadena> trabajo cuya línea de órdenes comienza por <cadena>
* %?<cadena> trabajo cuya línea de órdenes contiene <cadena>

La siguiente clasificación recoge las órdenes más frecuentes de control de trabajos:

* Jobs lista los trabajos activos bajo el control del usuario
* Fg trae a primer plano un trabajo que se encuentra suspendido o en segundo plan
* Bg envía a segundo plano un trabajo
* % permite cambiar el estado de un trabajo
* Wait espera la finalización de procesos en segundo plano
* Disown suprime un trabajo de la lista de trabajos activos
* Kill envía una señal a un/os proceso/s. Por defecto, finaliza la ejecución de un proceso
* Ps muestra el estado de los procesos actuales en el sistema
* Top muestra los procesos en ejecución con actualización de su información en tiempo real.

**La orden Jobs**

Permite ver el estado de los trabajos suspendidos y en segundo plano. La opcioón –l permite visualizar, junto con la información normal, el identificador de proceso asociado al trabajo.

**Las órdenes fg, bg y %**

La orden fg seguida de un especificador trae a primer plano el trabajo especificado (trabajo de fondo o suspendido). Esta orden, usada sin argumentos, lleva el trabajo actual a primer plano.

La orden bg es la opuesta de fg. Esta orden seguida de un especificador envía a segundo plano el trabajo especificado. Esta orden, usada sin argumentos, lleva el trabajo actual a segundo plano (background).

**Esperando a procesos en segundo plano**

La orden wait produce una espera hasta que el trabajo especificado cmo argumento haya finalizado su ejecución. El argumento también puede ser un identificador de proceso (PID), en cuyo caso, se esperaría la finalización de dicho proceso.

**Eliminando procesos con las órdenes disown y kill/killall**

La orden disown elimina el trabajo cuyo identificador se da como argumento siempre y cuando dicho trabajo esté activo, es decir, no tiene efecto sobre trabajos que estén suspendidos. Si no se aportasen argumentos, suprimiría el trabajo actual. Examina las opciones de disown usando la orden man.

La orden kill también permite eliminar procesos, pero es mucho más general que disown ya que actúa tanto sobre procesos activos como suspendidos. La orden kill sirve para enviar a un proceso una señal. La acción por omisión al ejecutar kill es finalizar el proceso (enviar la señal SIGTERM) o procesos indicados con identificadores o especificadores de trabajo (ver opción -s de la orden kill).

A veces la señal que envía kill a un proceso puede no finalizarlo debido a que el proceso la ignore o realice una acción distitna a la especificada por defecto. Para forzar la terminación de un proceso en estas circunstancias se debe invocar las orden kill -9.

**Examinando el estado de los procesos con ps**

Para obtener una información más detallada sobre los procesos del sistema se puede usar la orden ps. Esta orden muestra un listado de información sobre los procesos actuales y tiene muchas opciones.

Las opciones más comunes de esta orden son:

* -e muestra información de todos los procesos
* -l muestra la información sobre los procesos en formato largo.
* -u <nombre> muestra el estado de los procesos del usuario <nombre>
* -o <formato> permite usar un formato definido por el usuario para el listado
* -aux

**Examinando los procesos en ejecución con top**

La orden topo muestra una lista de procesos en ejecución de forma actualizada en tiempo real. La lista mostrará los procesos ordenados en función del consumo de CPU e información descriptiva de los mismos como el PID, usuario al que pertenece, consumo de CPU, consumo de memoria RAM, etc.

**Introducción a la compilación de programas con gcc/g++**

El compilador analiza el código fuente preprocesado y lo traduce a un código objeto que se almacena en un archivo archivoinc.o, conocido como módulo objeto. Para generar un archivo ejecutable es necesario que uno de los archivos .o que se están enlazando contenga la función main().

La función g++:

* -c indica a g++ que solo realice la etapa de preprocesado y compilación.
* -o se usa para dar un nombre distinto al que utilizada g++ por omisión.
* -L(directorio) permite especificar directorios en donde g++ puede buscar las bibliotecas necesarias. Por omisión g++ las busca en los directorios /lib y /usr/lib.
* -l<nombre> busca aquellas bibliotecas cuyo nombre va precedido por lib y su extensión es .a
* -I permite especificar directorios en donde g++ puede buscar archivos de cabecera.

El enlazador de GNU, id, debe tener los módulos objeto que contengan la definición de las referencias externas.

**Introducción a bibliotecas**

Si disponemos de un conjunto de módulos objeto podemos generar una biblioteca utilizando la orden ar:

* -r inserta los archivos miembro dentro del archivo. Esta operación difiere de q in que cualquier miembro existente previamente son borrados si sus nombres coinciden con aquellos añadidos.
* -v este modificador requiere la versión verbal de una operación. Varias operaciones muestran muestran información adicional, como los nombres de archivos procesados, cuando el modificador v es añadido
* -s escribe un archivo objeto índice dentro del archivo, o aumenta uno existente, incluso si ningún otro cambio es hecho al archivo. Puedes usar este modificador tanto con otra operación como solo. Es equivalente a ejecutar ranlib.

**Uso de archivos de tipo makefile**

Existe una utilidad make que permite gestionar las dependencias (y otras muchas cosas), comprobando qué archivos se han modificado desde la última vez que se ejecutó para construir el archivo ejecutable y, en su caso, vuelve a construirlo haciendo de nuevo sólo lo que sea necesario, es decir, compilando exclusivamente aquellos archivos que hubieran sido modificados.

**Ejecución de la utilidad make**

La utilidad make admite entre sus opciones la especificación del nombre de un archivo tipo makefile. Esa opción es –f y, a continuación, el nombre del archivo. Si el nombre para el archivo makefileGNU, makefile o Makefile, entonces no es necesario especificar la opción anterior junto al nombre del archivo, es decir, bastará con ejecutar la utilidad make sin argumentos.

**Estructura de un archivo makefile**

Un archivo makefile está compuesto por una o varias reglas cada una de las cuales estará asociada a la consecución de un objetivo concreto. Las reglas están formadas por un objetivo, una lista de dependencias (posiblemente vacía) y las acciones u órdenes que son necesarias para alcanzar ese objetivo.

La sintaxis básica para una reglas es como sigue:

TABULADOR orden1

TABULADOR orden2

····

TABULADOR orden

El objetivo ha de ser un nombre que resulte característico para la acción que representará y ha de escribirse comenzando desde la primera columna de línea. En la mayoría de los casos, el objetivo coincide con el nombre de un archivo, tal y como se podrá comprobar en los sucesivos ejemplos.

Una regla que permite alcanzar ese objetivo según la estructura de un archivo makefile sería la siguiente:

Programa1:

G++ -I./includes –o programa1 main.cpp hello.cpp factorial.cpp

Si hay una lista de dependencias, éstas deberán estar separadas por un espacio en blanco. Cuando se ejecuta la utilizada make, ésta comprueba si ha habido algún cambio en alguna de sus dependencias; si es así, se busca el objetivo correspondiente de las dependencias modificadas y se ejecuta su lista de órdenes asociadas.

Las órdenes son un conjunto de una o más líneas de orden del shell y siempre deben tener un tabulador al principio de la línea de orden.

Es posible construir una reglas sin órdenes asociadas. En este caso, cuando make trate de construir el objetivo de esta regla simplemente comprobará que los archivos de la lista dependencias están actualizados. Si es necesario construir alguno, pasa a ejecutar la regla que tiene este archivo como objetivo.

En general, la utilidad make, permite la ejecución de una regla cualquiera invocando el nombre del objetivo de la regla como argumento del make.

En un archivo de tipo makefile, también se pueden añadir comentarios anteponiendo el símbolo # en su primer columna y se puede extender a lo largo de toda una línea de texto. Si deseamos varias líneas de comentario cada una deberá comenzar por #.

**Uso de variables**

La utilidad make permite la definición de variables de igual forma que podíamos hacerlo en los guiones de bash.

Estas variables se pueden usar en las declaraciones de las reglas simplemente incluyéndolas entre paréntesis o llaves y anteponiéndoles el signo $. Además existen variables especiales que actúan cuando make procesa cada regla. Ejemplos:

* $@ Representa el nombre del objetivo de la regla en la que nos encontramos
* $< Representa la primera dependencia de la regla en la que nos econtramos
* $? Representa las dependencias de la presente regla que hayan sido actualizadas

(modificadas) dentro del objetivo de la regla y separadas por un espacio en blanco

* $^ Represneta todas las dependencias separadas por un espacio en blanco

**Uso de $@**

El valor de $@ en la regla sustituirá al objetivo.

**Uso de $<**

Esta variables se utiliza para representar el archivo aportado como primera dependencia de una regla.

**Uso de $?**Cuando se desea referencia a varias de las dependencias a la hora de actuar en consecuencia con las órdenes de una regla, es posible usar esta variable para indicarlas y, en concreto, referenciaría aquellas dependencias que se hubieran actualizado:

lpr – esta orden imprime archivos.

**Uso de $^**

Cuando se desea referenciar a todos los archivos indicados en las dependencias de una regla, lo cual supondría el ahorro de tener que escribirlos en varios lugares del archivo makefile, se utiliza la variable $^.

**Introducción a la depuración de programas con gdb**

La utilidad gdb o GNU debugger es el depurador estándar para el sistema operativo GNU. Funciona para varios lenguajes de programación como ensamblador, C, C++ o FORTRAN.

Ofrece la posibilidad de trazar y modificar la ejecución de un programa. El usuario puede controlar y alterar los valores de las variables internas del programa.

El esquema normal del funcionamiento es:

1. Compilar el programa g++ con la opción –g que añade la información necesarioa para gdb de cara a poder depurar el programa. Esta opción –g se ha de situar en la fase de compilación, aunque si con una sola orden se efectúa la compilación y el enlazado, entonces se deberá incluir igualmente en esa orden.
2. Ejecutar el depurador gdb
3. Dentro del intérprete del depurador, ejecutar el programa con la orden run
4. Mostrar el resultado que aparece tras la ejecución.

**Comprobación de ayuda y listar código.**

La orden help de gdb permite obtener ayuda genérica del programa. También se puede buscar ayuda de una orden, por ejemplo, help run.

La orden apropos busca en todo el manual por si hubiera ayuda asociada a un término, por ejemplo, apropos run.

Para listar código se utiliza la orden list.

**Comprobación de variables y estados**

Una de las ventajas de los depuradores es que podemos visualizar la información de las variables y su estado, así como información del contexto del programa. Para ello pueden ser interesantes las siguientes órdenes:

* display <variable> Muestra el valor de la variable durante la ejecución cada vez que el programa se detiene en un punto de ruptura. A cada orden display se le asigna un valor numérico que permite referenciarla
* print <variable> Muestra el valor de una variable únicamente en el punto de ruptura en el que se da esta orden. Se puede aplicar tanto a variables de área global o de alcance local.
* delete display id Elimina el efecto de la orden display sobre una variable, donde id representa el valor numérico asociado a la orden display correspondiente. Ese valor toma 1 para la primer orden display, 2 para la siguiente y así sucesivamente.
* examine dirección Examina el contenido de una dirección de memoria. La dirección siempre se expresa en hexadecimal.
* Show values Muestra la historia de valores de las variables impresas.
* p/x $pc Muestra el contador de programa usando su dirección lógica
* x/i $pc Muestra la siguiente instrucción que se ejecutará usando el contador de programa
* disassemble Muestra el código ensamblador de la parte que estamos depurando
* whatis variables devuelve el tipo de una variable
* info locals lsita todas las variables locales

**Puntos de ruptura simples**

En gdb se pueden añadir puntnos de ruptura simple que permiten examinar qué hace el programa en un determinado lugar. Para ello se puede utilizar la orden break. Esta orden puede tomar como parámetro el nombre de una función, la dirección lógica donde parar, o un número de línea. Para continuar el programa hasta el final o hasta el próximo punto de ruptura, se puede utilizar la orden **continue**.

Una vez detenidos a causa de un punto de ruptura, podemos avanzar a la siguiente instrucción de programa con la orden **next** o con **step**.

Los puntos de ruptura activos pueden verse con **info breakpoints**. Podemos eliminar un punto de ruptura con la orden **delete**.

**Ejecución de guiones**

Para no tener que escribir las acciones en la propia interfaz de gdb, podemos hacer uso de los guiones de gdb.

Los guiones de tipo gdb se invocan de la siguiente forma cuando se desee aplicar la depuración establecida en el guion al programa en cuestión:

gdb –x guion.gdb ejemplo1

**Depuración avanzada de programas con gdb: marcos (frames)**

Un programa contiene información acerca de las direcciones donde se van a ejecutar determinadas funciones del mismo. A esta información la pila de llamadas.

La pila de llamadas se divide en secciones contiguas llamadas pila de marcos (stack frames) o simplemente marcos(frames).

Cuando empieza el programa, la pila solo contiene un marco. Cada vez que se llama a una función, se crea un nuevo marco. Cada vez que la función devuelve algo, el marco asignado a dicha función se elimina.

La utilidad gdb emplea marcos en la depuración de un programa. La instrucción info frame muestra información acerca del marco actual. Mientras que backtrace full nos muestra la información referente a las variables locales y el reto de información asociada al marco.

Las órdenes next y step funcionan de forma distinta. Si se ha detenido la ejecución en una instrucción de llamaba a un subprograma, la orden step entrará en el marco donde se encuentra ese subprograma ejecutando sus instrucciones paso a paso, mientras que la orden next ejecuta el subprograma como si se tratase de una instrucción simple todo él.

Con down o up podemos elegir subir o bajar en la pila de marcos, de tal forma que podemos ir a al función más interna o subir a donde se hizo la última llamada a dicha función

**Puntos de ruptura condicionales**

La utilidad gdb permite añadir comprobaciones en los puntos de ruptura, de tal forma que sólo habilitar el punto de ruptura si se cumple la condición. Para ello utiliza la sintaxis del lenguaje depurado, así, si el programa estaba compilado para C++, la sintaxis utilizada será la de C++.

**Cambio de valores en variables**

Una de las grande ventajas en gdb es que se permite cambiar el valor de una variable mientras se está depurando el programa. Para ello puede ser útil la orden set, cuya sintaxis es:

set variable variable = valor.

**Depurar programas que se están ejecutando**

Gdb permite depurar programas que ya se encuentran ejecutándose en el sistema opertaivo.

La orden sería:

attach PID

donde PID es el identificador del proceso que se encuentra en ejecución y se desea depurar.

**Funcionalidad adicional del gdb**

La utilidad gdb permite integrarse con los editores del sistema, de tal forma que podemos lanzar el editor en cualquier momento desde éste. Para ello hay que cambiar la variable EDITOR del sistema:

$EDITOR=/usr/bin/gedit

$export EDITOR

Ahora podemos escribir:

edit numero\_linea

edit nombre\_funcion

El primero nos permite la edición del número de línea indicada y el segundo la función.

Desde gdb también podemos hacer llamadas a la shell. Basta con realizar lo siguiente

Shell orden