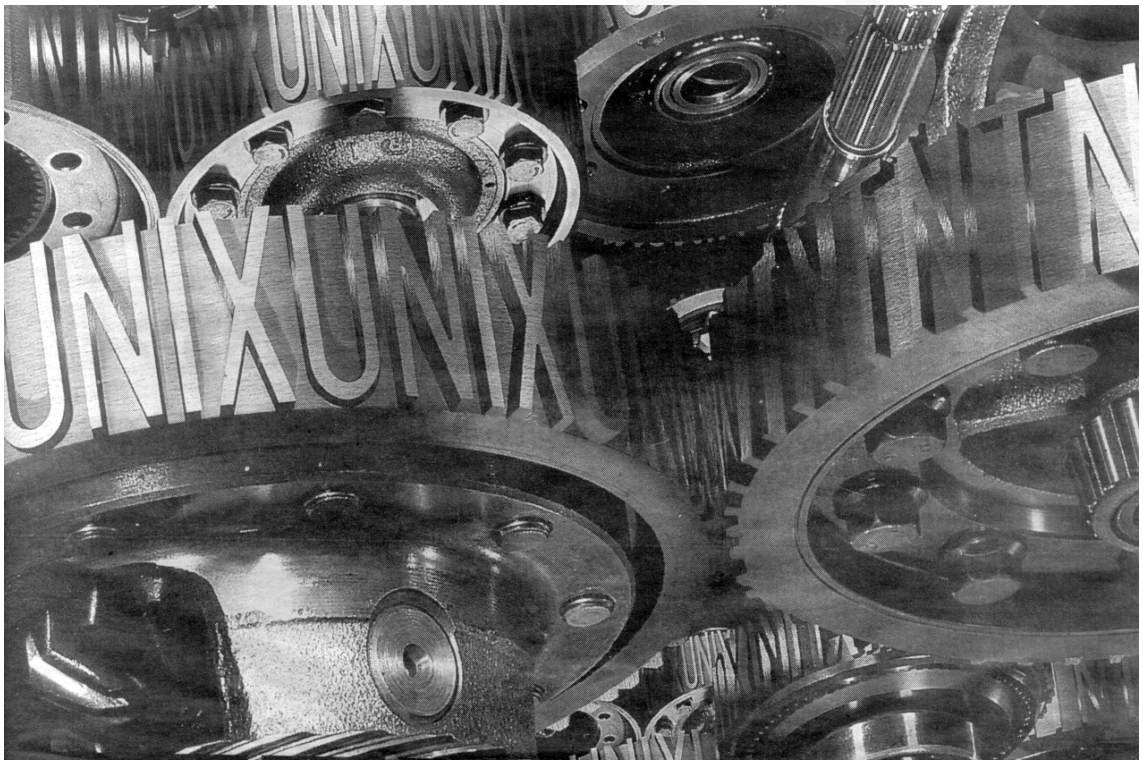
Universidad de Cantabria
Grado de Ingeniería Informática

Sistemas Operativos

Guion Práctica 1



Rafael Menéndez de Llano Rozas

Guion

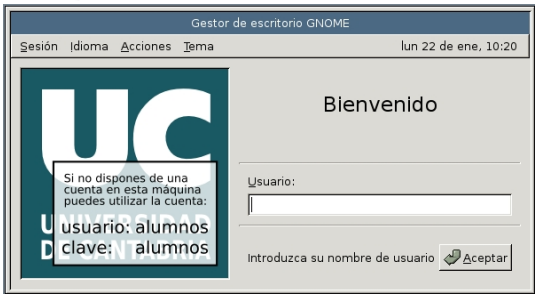
1

Uso de la Shell y del entorno de desarrollo del sistema

Visión del sistema operativo como usuario: Entrarás al sistema con tu usuario / password y utilizarás los comandos más comunes, así como sus distintas modalidades de uso como comodines, concatenación o desvío. Además, manejarás distintas características de la Shell como la posibilidad de cambiar sus variables o la de realizar macro comandos. Las preguntas que se te hacen en este formato sombreado las deberás responder y anotar para realizar la memoria final junto con pantallazos o código realizado.

Entrada y comandos básicos

Una vez arrancado el computador tendrás que elegir el tipo de sistema operativo con el que trabajar que en nuestro caso será Linux.



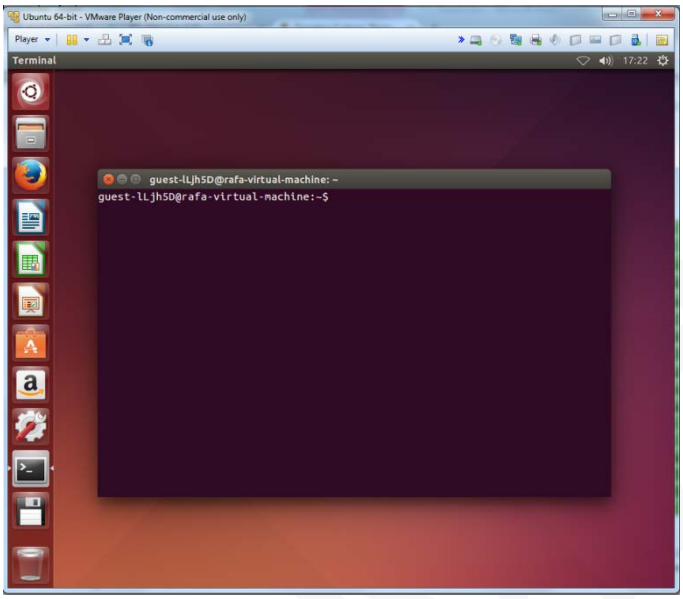
A continuación, se te pedirá el usuario/password. Si no te acuerdas de cuál era pregunta al profesor. En esta fase puedes elegir el idioma y el tipo de entorno gráfico que prefieres. Por defecto es Gnome.

Si estás usando Linux, tienes que tener en cuenta que dispones de varios terminales en uno. La mayoría son de texto y el último es el gráfico (arranque por defecto). Para acceder a ellos tienes que pulsar la combinación Ctrl-Alt-F1 a Ctrl-Alt-F7 (si fueran siete terminales). En el caso de máquinas virtuales suele funcionar además la combinación con *shift*.

1.	Entra al sistema y comprueba si te funcionan los distintos tipos de terminales. Deberás identificarte al menos en uno de ellos. Indica cuál es el gráfico.

Cuando abandones la sesión (con el comando `exit` o control-d), debes salir de todos los terminales abiertos.

Vete al terminal gráfico y explora el entorno, en cada versión puede variar. Por ejemplo, también tienes varios escritorios en uno sólo (pruébalo). Para acceder al uso de la Shell debes abrir un terminal que no es más que una aplicación del sistema sin ningún privilegio especial equivalente a los terminales de texto anteriores. Debes saber que Windows también dispone de su propia Shell para administración (Power Shell) y que recientemente se puede instalar una Shell bash de Ubuntu en el propio Windows (probado en 8,1 y 10).



Hay muchos tipos de shell: sh, csh, ksh, tcsh, bash, … Nosotros vamos a emplear la bash.

Todas las shell, una vez están en funcionamiento, esperan la introducción de comandos con un símbolo que se llama inductor (*prompt*), que es configurable como veremos posteriormente. El diálogo con la shell se basa en el uso de comandos que trabajan sobre el sistema de ficheros. Éste tiene una serie de propiedades que son las habituales actualmente:

- Es jerárquico, de tal manera que los usuarios pueden agrupar la información relacionada en una unidad y manejarla eficientemente (directorios).
- Aumento dinámico del tamaño del fichero para que contenga el tamaño necesario para almacenar su información, sin necesidad de que intervenga el usuario.
- Ficheros no estructurados, el Linux/UNIX no impone una estructura interna al fichero, por lo que el usuario es libre de interpretar el contenido de los mismos.
- Los ficheros pueden ser protegidos de accesos no autorizados.
- Existe un tratamiento idéntico de ficheros y dispositivos de entrada/salida. Así, los mismos programas pueden utilizar indistintamente tanto ficheros como dispositivos.

La estructura jerárquica es en forma de árbol. Partiendo de un directorio (carpeta en el ambiente gráfico) raíz conocido por "/" del que cuelgan subdirectorios. La separación entre directorios se hace con "/" (a diferencia de Windows que es con "\").

El acceso a esos ficheros se realiza de dos formas:

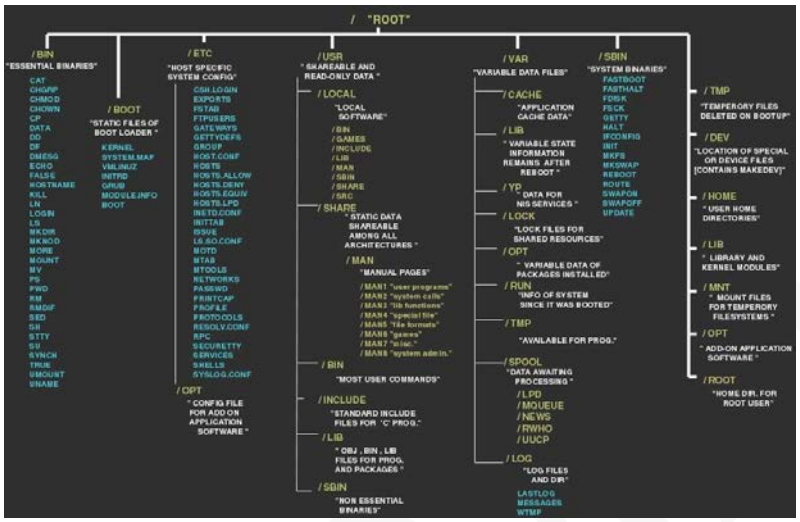
- Absoluta, partiendo del raíz y dando todo el camino: `/home/pepe/directorio/hola.c`
- Relativa. De forma automática, en cada directorio que se crea, aparecen dos subdirectorios virtuales que son el "." y el "..". Con ellos podemos ir al directorio actual o a su directorio padre.

A diferencia de Windows, en UNIX/Linux no existen unidades nombradas con una letra como A: , B:, C: , etc. sólo existe una unidad principal que mantiene el sistema de ficheros central donde está el directorio raíz y a partir de ese sistema de ficheros, en un proceso de montaje, se solapan los directorios raíces de las unidades secundarias en directorios (vacíos -si no lo están si contenido se

oculta-) de la principal. De esta manera sólo existe un único sistema de ficheros, aunque pertenezca a varias unidades.

Además, UNIX/Linux trabaja de la misma forma con ficheros y directorios que con dispositivos. De esta manera, donde se puede poner el nombre de un fichero, también se puede usar un dispositivo.

El árbol del sistema de ficheros puede variar ligeramente de uno a otro sistema, pero existen colgados del directorio raíz una serie de directorios típicos:



El que más nos interesa es el `/home` donde están los directorios de usuarios (cada usuario tiene el suyo propio con su nombre y está protegido de otros usuarios). Recuerda que en el laboratorio tu directorio estará en `/remotehome` al estar contra un servidor.

Conociendo ya la estructura del sistema de ficheros, podemos trabajar con ella a través de los comandos más habituales (los comandos pueden ser externos –se puede ver su tipo con `file`– o internos –se puede ver con `type`–). Todos los comandos tienen una estructura típica: nombre (en negro), opciones (en verde) y argumentos (en azul), resultado después. En el caso de directorios son:

```
pwd nada nada pinta directorio de trabajo
$ pwd
/usr/ramon/trabajos

cd nada nada|directorio cambia de directorio (sin arg va al directorio home)
$ cd ../textos
$ cd -                  cambia al directorio de donde veníamos
```

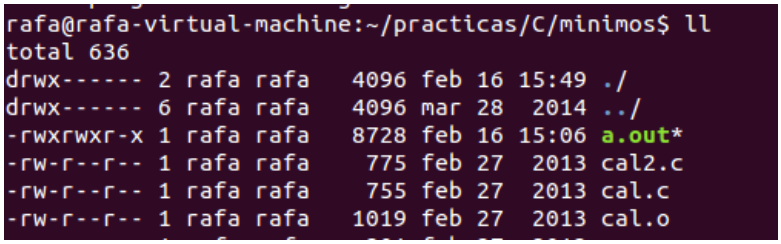
También se puede usar "~" para acceder a home. En Windows se puede obtener con la combinación Alt+126, o mediante la combinación AltGr+4 (el 4 de encima de las letras no el del teclado numérico); en Linux con la tecla AvPág o mediante la combinación AltGr+ñ; en Mac OS X con Alt+ñ.

```
mkdir nada directorio hace un nuevo directorio
$ mkdir /usr/ramon/graficos

rmdir nada directorio remueve (borra) un directorio un directorio vacío
$ rmdir graficos

ls -a,c,l,r,s,... nada|directorio lista directorios
$ ls –alrt           muestra todos los archivos (incluidos los ocultos) de forma
extendida y ordenados por fecha de última modificación en orden creciente
```

En este comando el nombre del directorio es opcional y las opciones más usadas son: a para dar todos los ficheros, c para ordenar por fecha, s para mostrar tamaño, `r` para invertir el orden de salida y `l` para dar formato largo a la salida del directorio, esta opción se usa tanto que hay una orden que sustituye a `ls -l` y es `ll` (si no existe se puede crear como veremos).



En Linux incluso se pueden presentar los distintos tipos de ficheros por colores (verde para ejecutables, azul oscuro para directorios blanco para ficheros regulares, hay más), donde en total nos dice el número de bloques de espacio listados (636), después en la siguiente línea y en el primer carácter nos dice el tipo de fichero que es (por ejemplo d para directorios o – para ficheros regulares), y a continuación los permisos del mismo (lectura, escritura o ejecución), el número de enlaces, el propietario del fichero, el grupo al que pertenece, el tamaño en bytes, la fecha del último cambio y por último el nombre del fichero.

Y para ficheros:

```
cp varias f1 f2 fn directorio copia ficheros origen destino
$ cp origen destino
$ cp f1 f2 f3 directorio

cat sin ficheros (concatenar) muestra un fichero
$ cat pepe.c
main ()
{
...

more varias ficheros muestra por páginas un fichero
$ more pepe.c
```

tiene su propio lenguaje de comandos, principalmente se puede dar un número positivo (+) o negativo (-) de pantallas para ir hacia adelante o hacia atrás, si se da el número sin signo nos posicionamos en una determinada pantalla, si pulsamos [enter] avanzamos una pantalla adelante, con b iremos para atrás o con espacio una línea para adelante y si pulsamos q nos saldremos del programa, (en algunos sistemas UNIX se utiliza el comando pg de similares características).

Existe la versión `less` más avanzada que no necesita leer el fichero completo para presentarlo.

```
less varias ficheros muestra por páginas un fichero
$ less pepe.c
```

también tiene su propio lenguaje de comandos (además del `more`): ctrl+u página arriba, ctrl-d página abajo, ctrl-p línea arriba, ctrl-n línea abajo, g al principio, G al final, y / para buscar.

```
head varias fichero muestra las primeras líneas de un fichero
$ head pepe.c
main ()
...

tail varias fichero muestra las últimas líneas de un fichero
$ tail pepe.c
...
}

mv varias origen destino mueve o renombra ficheros a un directorio
$ mv registro almacen
$ mv reg1 reg2 reg3 muestras

ln varias 4 formas enlaza (link) un fichero a otro o a un directorio
$ ln /usr/ramon/ventas /usr/ramon/graficos
```

Primera forma: dos nombres de ficheros (crea un enlace con otro nombre en el mismo directorio), Segunda un único nombre (crea un enlace en el directorio actual de un fichero en otro directorio), tercera forma: fichero y directorio (crea un enlace en el directorio indicado), cuarta forma (*target*) con –t directorio fichero (equivalente a la 3ª solo que se pone antes el directorio). En este comando la forma de enlazado

se puede hacer de dos maneras: *duro (*enlace *hardware* o físico) es el por defecto, crea una "puerta" a un fichero existente, pero sólo existe un fichero y las dos puertas tienen los mismos privilegios; *lógico (*enlace *software*, blando o simbólico), se crea un fichero especial (acceso directo) que simplemente guarda información sobre como llegar al original (se usa –s y se puede leer con `readlink`). Los directorios sólo se pueden enlazar de esta manera. El número de enlaces aparece al hacer un `ls –l`.

```
rm -i fichero elimina un fichero
rm -r directorio elimina un directorio y sus ficheros
$ rm fich
$ rm -r /usr/ramon/ventas     Borra recursivamente sin preguntar
$...
```

Hay que tener mucho cuidado con este comando, al igual que con `rmdir`, ya que no existe la vuelta atrás y si se borra no se podrá recuperar lo eliminado (no hay papelera).

Siempre que tengamos alguna duda sobre el uso de un comando se puede usar este otro comando:

```
man [seccion] comando
```

Nos presentará la información que hay disponible sobre un determinado comando o aplicación. La información del manual está dividida en secciones y deberíamos saber a qué sección pertenece lo que intentamos buscar (si es un comando normalmente la 1, que es tomada por defecto), pero también podríamos buscar una llamada al sistema (la 2) u otra entidad del sistema. El formato de salida será el del comando `less`. Entre las muchas opciones que tiene, la -k es posiblemente la más interesante y en Linux es equivalente al comando `apropos` que busca el *string* indicado en la base de datos de búsqueda del sistema. Hay que destacar que es diferente a -K que busca el *string* en todo el manual y puede ser muy lento. Un comando para buscar de forma sumarizada en que sección está algo es `whatis`.

```
man -k comando ⇔ apropos comando   /     whatis comando
```

Las secciones del comando suelen residir en el directorio `/usr/man` y son: 1. Comandos de usuario. 2. Llamadas al sistema. 3. Funciones y rutinas de librería. 4. Configuración y formato de ficheros. 5. Miscelánea. 6. Ficheros especiales y hardware. 7. Ficheros especiales y hardware. 8. Comandos de mantenimiento. 9. Drivers.

Ahora que ya conocemos el comando `man` (suele ser interesante colocar otro terminal para el `man`), podemos describir la arquitectura de los comandos y como vienen descritos en él:

- **[ ]** indica que hay argumentos opcionales.
- **…** indica que se pueden expresar múltiples valores del argumento.
- **|** indica que los argumentos a la izquierda o derecha son excluyentes.
- **{ }** indica argumentos mutuamente exclusivos pero uno obligatorio.

Debes saber que hay cientos de comandos, la gran mayoría comunes a todas las Shell. Una referencia para la `bash` la puedes encontrar por ejemplo en *http://es.wikipedia.org/wiki/Comandos_Bash*.

2. Vete a tu directorio de usuario, compruébalo con `pwd`. Crea un directorio a partir de tu directorio llamado temporal, entra en él, y compruébalo de nuevo con `pwd`. Intenta hacer lo mismo desde el directorio `/remotehome` ¿Puedes hacerlo? Di porqué. Averigua si `cd` es interno o externo. ¿Y `pwd`?

3. Ve a ~/temporal y con el comando `touch` (si no sabes que es prueba el `man`) crea estos ficheros: "pepe1", "pepe2", "pepe3", "pepe12" y "pipe3". Copia, mueve y enlaza de dos formas (hardware y software) algún fichero de `temporal` en su directorio padre. Si haces el enlace blando de forma relativa ¿qué ocurre? ¿Cómo lo solucionarías? Explora las opciones de los comandos. Localiza en que sección están del manual.

4. Haz un `ls` con diversas opciones (a, l, s, r, y R) y describe las diferencias. Cuando termines borra el directorio `temporal` con `rmdir`. Indica si puedes hacerlo o no. Si es que no, ¿Cómo lo harías (pero no lo hagas)?

Los comandos que hemos introducido en una sesión de trabajo siempre se pueden recuperar (se guardan en el fichero `.bash_history`), en lo que se conoce como historial (lo que simplifica el trabajo diario) con el comando `history`:

```
$ history
```

Las flechas arriba y abajo nos permitirán recorrer el historial. Otra forma de aprovecharlo es usando el comodín "`!`" seguido de alguna de las letras por la que empezaba el comando almacenado, recuperarlo sin necesidad de escribirlo de nuevo. La búsqueda se hace en sentido inverso, es decir, se busca primero el más reciente. También se puede usar `control+r` (`ctrl+r` presionar la tecla control y r) seguido de los caracteres de algún comando anterior para que la *Shell* lo localice.

También se pueden autocompletar las líneas de comando usando el tabulador. Si existen ambigüedades nos lo dirá y deberemos continuar escribiendo. Sabed que `ctrl+a` nos mandará al comienzo de una línea y `control+e` al final, `alt+f` y `alt+b` mueven adelante y atrás una palabra, `ctrl+k` y `ctrl+u` borran desde el cursor hasta el final o hasta el principio, y `alt+d` y `ctrl+w` borran la palabra de adelante o de atrás. El uso de estas facilidades hace que se queden en tu memoria "muscular". Se pueden cambiar con `set –o vi / set –o emacs` si estás acostumbrado a otros editores.

Por último y para acabar con este apartado, podemos encadenar varios comandos en una sola línea de comandos simplemente separándoles con un ";". También se pueden encerrar varios comandos entre paréntesis para ejecutarlos dentro de una *subshell*:

```
(cd; pwd) ; pwd      pintará el directorio home, y después el actual, dejándonos en él.
cd ; pwd ; pwd       pintará dos veces el directorio home y se quedará en él.
```

Existen otras variantes de cadenas de ejecución, pero en estos casos condicionales. Cada vez que se ejecuta un proceso, devuelve a la *Shell* un valor lógico (*exit status*) que la indica si este proceso se ha ejecutado bien o no. El estatus valdrá cero (al contrario que en lenguaje C) cuando el comando se haya ejecutado con éxito. Se puede utilizar este valor para encadenar la ejecución de varios comandos (date cuenta que esto no es canalización, se verá después) de forma condicional, de tal manera que, si se ha ejecutado el primero bien, se ejecute el segundo o no. Para ello existen dos operadores principales: el AND, &&, y el OR, ||:

```
comando1 && comando2      se ejecutará el 2 si el 1 es correcto (status 0)
comando1 || comando2      se ejecutará el 2 si el 1 es no correcto (status <> 0)
```

5. Usa el comando `history` y el comodín "!". Prueba el `ctrl+r` (búsqueda reversa) y el resto de combinaciones de teclas para editar líneas. Comprueba como cuando se hace uso de los paréntesis, se ejecuta lo encerrado en una *subshell*. Prueba el concatenado condicional para copiar un fichero que exista o no, si es así, que se edite. Haz lo mismo compilando un programa con y sin errores y ejecutándolo.

Comodines

Una característica que está en otros sistemas operativos es la posibilidad de utilizar caracteres especiales que sirven de comodines (*Pattern Matching*) para referenciar un grupo de ficheros.

En la *Shell* existen dos caracteres especiales el "*" que sirve para sustituir a una secuencia de 0 o más caracteres y el "?" que sólo sustituye uno. A diferencia de otros sistemas operativos el punto en un nombre de un fichero no actúa como un carácter especial separando nombre y extensión, sino que forma parte del nombre completo, por lo cual el "*" también lo sustituye.

Algunos ejemplos de utilización son:

```
rm * borrará el directorio actual.
rm pep* borrará todos los ficheros que sean o empiecen por pep
rm reg? borrará los ficheros reg1, reg2 y reg3, pero no reg12
```

Hay una modificación del "?" y es cuando se especifican entre corchetes los caracteres que se van a sustituir, así:

```
rm reg[12] sólo borrará los ficheros reg1 y reg2
```

También se pueden utilizar otros caracteres especiales: { } para replicar (expansión) un grupo de nombres encerrados entre las llaves y separados por comas:

```
a{b,c,d}e se referirá a abe, ace y ade
```

Además en la `tcsh` existe el carácter '^' (94) para negar cualquier selección hecha con los comodines. Por ejemplo, si tengo los ficheros `bang`, `crash`, `cruch`, y `out`, `ls ^cr*` dará como resultado `bang out`.

Por último, existe un carácter especial más que es el '\', que antepuesto, sirve para interpretar literalmente (*quoting*) algún carácter especial como puede ser un comodín o un espacio. Esto es útil cuando por error (por estupidez o capricho) hemos introducido uno de estos caracteres en el nombre de un archivo e intentamos acceder a él, piensa, por ejemplo, como podrías borrar un fichero que tiene en medio del nombre el carácter espacio. También se utiliza para escribir una orden en varias líneas, acabando cada una de éstas con un '\' para que la *shell* sepa que no termina.

6. Vete a temporal y prueba con los ficheros creados los caracteres comodines ("*", "?", "[ ]"). Utiliza también el comodín de negación (es necesario ejecutar la `tcsh`). Crea un fichero con un espacio en su nombre y bórralo. Vete a tu directorio con "~". Usa el comando `echo` con un argumento que no quepa en una línea (uso de \).

Redirección y Canalización

Otra característica importante de las *Shell* es la redirección de los canales de entrada/salida estándar. Cuando se ejecuta un programa, normalmente la *Shell* le adjudica al proceso tres canales o ficheros estándar, por un lado, un canal de entrada (`stdin`), de donde el programa toma los datos; por otro lado un canal de salida (`stdout`), donde se dejan los resultados; y por último un canal de errores (`stderr`) que recibirá los mensajes de error que se produzcan durante la ejecución del proceso (como curiosidad, en la *Shell* hay hasta 9 canales). Los dos primeros canales están asignados normalmente al terminal donde se trabaja (teclado y pantalla) y el tercero a la pantalla, pero es distinto de forma lógica al canal de salida. Estos tres canales tienen 3 descriptores de fichero asignados: 0 para entrada, 1 para salida y 2 para errores.

Desde la línea de comandos se puede cambiar la asignación a estos canales a través de un proceso que se llama redirección y que utiliza dos caracteres especiales el ">" y el "<", el primero para la salida y el segundo para la entrada, estos símbolos pueden ser precedidos por el descriptor de fichero, aunque si son los normales (0 y 1) son omitidos, no así el de error.

Si tenemos un programa que nos pide datos desde la línea de comandos y produce un resultado en la pantalla, podemos hacer que esos datos los tome desde un fichero en vez de escribirlos uno a uno con el teclado, para ello utilizaremos la redirección de entrada:

```
$ programa < datos
```

Donde `datos` es un fichero de texto con la información necesaria para que el programa funcione. De igual manera podemos hacer que la salida de ese programa no se pierda en la pantalla, sino que se escriba en un fichero, entonces tendremos que utilizar la redirección de salida:

```
$ programa > salida
```

e incluso podemos hacer que tome los datos de un fichero y los saque en otro:

```
$ programa < datos > salida
```

Cada vez que usamos la redirección de salida, si el fichero existe se sobrescribe (podemos cambiar este comportamiento con una opción de la *Shell*: `set –o noclobber`). Si queremos añadir los nuevos datos usaremos ">>":

```
$ programa >> salida
```

Y como hemos dicho, la equivalencia entre un fichero y un dispositivo en UNIX/Linux es total, por lo que podemos redirigir la entrada o la salida hacia un dispositivo (en el directorio `/dev` están representados todos los dispositivos del sistema) como la impresora o nulo:

```
$ programa > /dev/lp    $progr > /dev/null
```

Incluso si no queremos que aparezca nada se puede eliminar la salida con el dispositivo especial `/dev/null`, que hace que todo lo que le llegue desaparezca.

También se puede redirigir la salida de error con "2>":

```
$ cat pepe 2> juan      $cat pepe > aaa 2< bbb
```

de tal manera que si no existe el archivo `pepe` se escribirá el error en el archivo `juan`.

También se puede hacer que la salida de errores y la normal (o viceversa) vayan hacia el mismo canal. Para ello existe la combinación &> o >&, aunque se recomienda la primera forma. En el primer caso (salida normal desviada a errores) deberíamos usar 1&>2, en el segundo caso 2&>1. En las *shell* modernas como bash sólo se usa &>:

```
$ programa &> fichero
```

Es conveniente saber que las redirecciones que hemos utilizado aquí son para las *shell* derivadas de `sh` directamente: `ksh` y `bash`. Para la `tcsh` también sirven con dos salvedades: que no se usan los descriptores de ficheros, lo que implica que la salida de errores se redirecciona con ">&·"; y que la salida normal y la de errores no se pueden usar simultáneamente, con lo cual tenemos que utilizar la ejecución en una sub-shell para corregirlo:

```
$ (programa > salida ) >& error
```

7. Usando la redirección de salida crea un fichero `calendario` con el comando `cal`, primero de 2017 y después añadiendo el 2016. Mira el contenido del fichero con `cat`, con more y con `less` usando sus subcomandos (retorno, espacio, b, q, etc.).

8. Usa el comando `ls –l` con el directorio `/root` (o algún otro que no tenga permisos) y desvía la salida hacia un fichero ¿sale algo por pantalla? ¿A qué es debido? ¿Cómo se podría evitar que salga? Desvía de forma unificada tanto la salida normal como la de errores en un único fichero.

La canalización está relacionada con la comunicación entre procesos y aunque se utilice también en la línea de comandos y junto con la redirección, no tiene nada que ver con ésta. La canalización consiste en transmitir datos directamente de un proceso (comando o programa) a

otro (la salida del primero sirve de entrada al segundo) para lo que se utiliza el símbolo "|" (*pipe*, *pipeline* o tubería) que los comunica. Por lo tanto, mientras la redirección comunica procesos con ficheros/dispositivos, la canalización comunica programas. No hay límite entre el número de procesos que se pueden encadenar:

```
$ programa1 < entrada | programa2 | programa3 > salida
```

Existe un comando especial `tee` que hace que lo que pasa entres dos programas se guarde en un fichero:

```
$ programa1 | tee fichero | programa2
```

La canalización se puede combinar con la concatenación de comandos condicional como se ha hecho en los dos ejemplos siguientes:

```
echo hola|write rafa||mail rafa        Manda hola a un usuario o bien por pantalla
                                       si está en el sistema o bien por mail.
cat fich|write usuario&&echo si        Manda un fichero por pantalla a un usuario y
                                       pone si solo si realmente funcionó.
```

También se puede unificar la salida de dos programas con llaves (se debe acabar cada orden con "; " y empezar con " " ojo con los espacios que son ␣).

```
{␣echo comienzo;␣cat fichero;␣} | wc –l    unifica la salida de los dos primeros
```

Hay comandos que no leen nada de la entrada estándar (`ls`) y otros de tratamiento de texto que usan mucho con la canalización: `head` para mostrar las primeras líneas de un fichero, `tail` las últimas, `nl` para poner números de líneas, `wc` cuenta caracteres, palabras y líneas, `uniq` para eliminar líneas repetidas, `sort` para ordenar líneas (se puede usar –n y –r), `cut` para trocear líneas, `fmt` formatea texto a un número de caracteres por línea, etc. Existen muchos más.

9. Realizar un pipeline (tubería) con el comando `echo, tee y wc`. Haz una concatenación de comandos con llaves de el comando `echo` y `more` de un fichero y entúbalo al comando `nl`, date cuenta de la diferencia de usar y no usar llaves.

Expresiones regulares y búsquedas

Algunos comandos del sistema (`awk, ed, find, grep, sed, vi, …`) funcionan como motores de búsqueda y usan lo que se conoce como expresiones regulares (pueden ser básicas BRE o extendidas ERE) para indicar lo que estamos buscando (no confundir con comodines –metacaracteres de shell– ya vistos). Una expresión regular es un conjunto de reglas que se emplean para especificar uno o más elementos dentro de una cadena de caracteres. Veamos algunos de los más usados (hay bastantes posibilidades, ver http://www.gnu.org/software/grep/manual/grep.html#Regular-Expressions):

- Cualquier carácter **individual** con "**.**". Ejemplo l..a puede ser lapa, lana, laca, …
- **Conjuntos** de caracteres con "**[ ]**". Ejemplo expresi[oó]n puede ser expresion o expresión. Se puede usar el carácter "-" para expresar rangos de caracteres. También hay algunos predefinidos: [:alnum:], [:alpha:], [:cntrl:], [:digit:], [:graph:], [:lower:], [:print:], [:punct:], [:space:], [:upper:], y [:xdigit:]. Por ejemplo [[:alnum:]] es cualquier alfanumérico.
- **Anclajes**: Comienzo con "**^**". Ejemplo ^x todo lo que empieza por x. Fin con "**$**". Ejemplo n$ todo lo que acabe en n.
- **Repetición**. Lo anterior a "*****" estará repetido 0 ó más veces. Ejemplo xy* puede ser x, xy, xyy, xyyy, … Lo anterior a "**?**" es opcional y estará repetido una vez. Ejemplo xy? Puede ser x o xy. Lo anterior a "**+**" estará repetido 1 ó más veces. Ejemplo xy+ puede ser xy, xyy, xyyy, … Con las llaves "**{n}**" lo precedente n veces (número exacto). Con "**{n,}**" n o más veces, con "**{,m}**" al menos m veces, con "**{n,m}**" al menos n veces pero no más de m.

- **Grupos**. Con "**( )**". Ejemplo (xy)+ sería xy, xyxy, xyxyxy, …
- Las expresiones pueden ser concatenadas.
- Las expresiones pueden ser alternativas con "|". Ejemplo x|y|z es x o y o z.
- Secuencia de escape con "****". Sirve para dar un significado especial al carácter que viene detrás. Por ejemplo: \. \$ * .

Si las usamos con `grep` para construir patrones de búsqueda, estos suelen estar encerrados entre apóstrofes (o comillas). `grep` puede tomar opcionalmente un argumento que es un fichero donde buscar, o sin él, hacer la búsqueda en la entrada estándar. Si queremos usar ERE debemos usar la versión extendida –E o `egrep`). Hay una tercera versión para los familiarizados con perl `–P`.

Ejemplos:

```
egrep '(192.168.0.30|192.168.0.40)' test      Busca las dos cadenas en test
egrep '192.168.0.[30,40] ' test               Busca las que tengan 30 o 40
egrep '192.168.0.[0-2]0' test                 Busca entre 0 y 2
egrep '^hostname' test                        Las que empiecen por hostname
egrep 'node$' test                            Las que terminen en node
```

10. Crea un fichero con el comando `cal`. Busca dentro del fichero creado con el comando `grep` las ocurrencias de la palabra febrero.

11. Realizar un pipeline (tubería) con el comando `ls` (en forma recursiva del directorio /) y el comando `grep` para buscar los ficheros que tengan "pepe" en su nombre y acaben en un digito.

Hay una forma de búsqueda en el sistema de ficheros y ejecución combinada de lo buscado, se realiza con el comando `find`, muy útil para usuarios y sobre todo para labores administrativas. La búsqueda siempre se realiza de forma recursiva salvo que se indiquen directorios donde buscar como primer argumento opcional. Toma como resto de argumentos expresiones que pueden ser de tres tipos: opciones, test y acciones. Dada la complejidad del comando es mejor ver algunos ejemplos y revisar el `man` para otros usos:

1. Buscar un fichero por su nombre:
```
a. find  -name abc.c
b. find  -name abc*            // con comodines
c. find  -iname aBc*           // sin tener en cuenta mayusculas
d. find  -not -name *.php      // invirtiendo criterio
e. find  -type f -name abc*    // con tipo fichero
f. find  ./d1 ./d2 -name abc*  // en dos directorios solo
```

2. Buscar por alguna cualidad (los permisos se verán en otro apartado):
```
a. find  -perm 0664            // con ese permiso
b. find  ! -perm 0777          // sin ese permiso
c. find  -perm /u=r            // permiso de lectura
d. find  -user luis            // archivos de luis
e. find  -mtime 50             // de los últimos 50 dias
f. find  -atime 7              // accedidos los últimos 7 dias
g. find -mtime +30 -mtime -60      // entre 30 y 60 dias
h. find -cmin -60           // cambiados –estatus- hace 60 minutos
i. find -mmin -60             // modificados hace 60 minutos
j. find -amin -60             // accedidos hace 60 minutos
k. find . –size +40M          // ficheros de más de 40 MB.
l. find –user pepito          // buscar ficheros de pepito
```

3. Buscar y realizar acciones (`exec`) con lo encontrado:
```
a. find -type f -name "*.txt" -exec rm -f {} \;    // los borra
b. find  -perm 0777 -print -exec chmod 644 {} \;   // cambia permiso
```

Advierte como si usas la tercera versión con `exec` debes respetar el curioso formato del comando `{}␣\;` (ojo al espacio ␣) que indica que se ejecute lo obtenido exclusivamente en lo encontrado por `find`.

12. Averigua los cinco ficheros más grandes de todo tu directorio de dos maneras diferentes: usando `find` y usando directamente `ls`. Pista: deberás usar los comandos `sort` y `head` canalizados.

Hay otros dos comandos que se usan para localizar ficheros, el primero es `locate`, usa una base de datos donde busca el fichero y por tanto es más rápido pero esa base de datos debe estar actualizada, para lo cual se usa `updatedb` como superusuario y el segundo `whereis` que busca comandos y nos dice la ubicación, los ficheros fuentes y su manual (`-bsm`).

```
$ locate pepe1
$ whereis ls
```

Ejecución en segundo plano (background)

Hay dos formas de ejecutar un programa: una es la usual en primer plano "*foreground*" y la otra es en segundo "*background*". En la primera, el programa una vez ejecutado toma el control de la entrada/salida, en la segunda el control vuelve a la Shell desde donde lo hemos ejecutado (hay que tener en cuenta que las salidas del programa se podrán mezclar en el terminal con el *prompt* y lo que escribamos a la Shell). Para realizar la ejecución de esta manera basta con poner un "&" detrás del comando:

```
$ programa > salida &
```

Cuando se ejecutan varios procesos en *background* se puede ver si realmente se están ejecutando con el comando `ps` (estatus de proceso) que tiene varias opciones que nos dicen que procesos representar (todos los del sistema, los de un usuario, etc.) y su formato. La forma más sencilla es:

```
$ ps
PID TT TIME COMMAND
3243 30 0:11 bash
3254 30 0:01 ps
```

la información que aparece se refiere a la identificación del proceso (PID), al número de terminal, al tiempo empleado en la ejecución y al nombre del proceso.

El directorio `/proc` (pseudo sistema de ficheros de sólo lectura) hace de interfase con el núcleo del sistema (más información con `man proc`). En él aparecen una serie de "directorios" con nombre numérico (PID) que corresponden a los procesos activos en ese momento en el sistema. En estos directorios podemos encontrar toda la información de un proceso, como por ejemplo los ficheros que tiene abiertos (`fd`), el mapa de memoria (`maps`), o el estado del mismo (`status`).

Otra forma de ver la jerarquía de los procesos es el comando `pstree` que nos da una imagen en forma de árbol de los procesos que se están ejecutando actualmente.

Conociendo el PID de un proceso siempre lo podemos abortar con el comando `kill`, al cual daremos como argumento ese PID. En Linux muchas veces resulta más cómodo dar el nombre del proceso (o procesos si tienen el mismo nombre como `a.out`) que queremos eliminar, esto se puede hacer con el comando `killall`. En ambos casos los `kill` pueden tomar una opción numérica que es la señal (evento del sistema) que queremos enviar, en el caso de no poner nada, la señal enviada es la número 15 (se pueden ver todas con `kill -l`). Los procesos pueden estar protegidos contra la recepción de señales, en ese caso el comando `kill` no les

afectaría. Por seguridad hay una señal que nunca puede ser evitada que es la número 9 (`SIGKILL`). Por eso en muchos casos se usa:

```
$ kill -9 12038     o      $ killall -9 ./proceso
```

Los procesos son generales al sistema, pero la *shell* permite tratar a los procesos creados en una sesión de una forma más sencilla, son los *jobs*. No hay que pensar que los *jobs* son diferentes a los procesos, simplemente es otra forma de utilizarlos y no tener en cuenta al resto de procesos creados en el sistema. De esta manera, nosotros podemos suspender un proceso que se esté ejecutando con `ctrl-z` (no confundir con ejecución en *background*, que si se ésta ejecutando en otro plano) y al tener el control de la *shell* de nuevo, mandar un nuevo proceso a ejecutar en *foreground* o *background*. Para ver la lista de procesos de la sesión, *jobs*, tenemos en comando del mismo nombre `jobs`, que nos dará una lista de los mismos entre corchetes:

```
$ jobs
[1]   Detenido                 vi
[2]-  Detenido                 vi
[3]+  Detenido                 vi
[6]   Hecho                    ls --color=auto -R &> /dev/null
```

Para ejecutarles sólo tenemos que utilizar los comandos *`fg`* y *`bg`*, que ponen respectivamente al *job* en ejecución *foreground* o *background*. El + indica el que se toma por defecto, el – el siguiente.

```
$ bg 3
$ fg 1
```

13.	Indica que dos procesos están siempre cuando se ejecuta ps. Averigua las opciones del comando `ps`. Ejecuta en *background* el editor de textos `vi`. Mira si el proceso se ha creado realmente. Si es así mátalo (`kill`). Repítelo con ejecución en *foreground* y para al proceso (debes saber cómo hacerlo). Después, pásalo de nuevo a *foreground*, páralo de nuevo, pásalo a *background* (indica la diferencia entre este estado y el anterior) y finalmente mátalo con `killall`. Piensa en una forma de averiguar la shell que estás ejecutando.

Fin de la primera sesión

Ambiente multiusuario

Como se ha dicho, el sistema UNIX es multiusuario y multiproceso, esto significa que varios usuarios pueden estar ejecutando varios procesos a la vez. Para poder hacer esto, el sistema tiene que tener unas determinadas características:

- Sistema de seguridad de acceso. Se debe garantizar que un usuario tiene a salvo su trabajo libre de accesos no permitidos. En el caso del sistema Linux/UNIX a través de un password.
- Seguridad de ficheros. Se debe garantizar que los usuarios no accedan y por tanto cambien o destruyan ficheros del sistema o de otros usuarios. En Linux/UNIX se utilizan tres tipos de permisos y tres modos de acceso que se verán posteriormente.

- Compartición de recursos. Los recursos comunes deben ser repartidos equitativamente entre todos los usuarios (ver capítulo de gestión de recursos). En el Luinux/UNIX se utilizan sistemas de *spooling.*
- Control de recursos. También se debe llevar la cuenta de los gastos de los distintos usuarios (uso de CPU, papel, etc.). Se llama *accounting*.
- Administración de sistemas. El sistema debe garantizar mecanismos para hacer *backups* de los trabajos de los usuarios y la posible recuperación de errores.

Como hemos dicho, la seguridad del sistema se lleva a cabo en dos niveles, por un lado pidiendo el *password* al usuario (cuando *root* ha creado un usuario con el comando `adduser` o `useradd`) le pone una palabra de paso temporal en `/etc/passwd` —actualmente se usa /etc/shadow por seguridad— que debería ser cambiada[1]), y por otro lado y una vez dentro del sistema, no permitiendo el acceso a ficheros no autorizados. El *password* es una palabra secreta que sólo conoce el propietario de la misma y que debe introducir en el sistema cuando entra al mismo. Para proteger el secreto cuando se introduce esta palabra, el terminal se pone en blanco de tal manera que los caracteres introducidos no se vean reflejados en la pantalla. El usuario siempre puede cambiar su *password* a través de un comando del sistema operativo. En nuestro caso no conviene hacerlo ya que son usuarios por red.

El segundo nivel de seguridad es el acceso a los ficheros. Cada fichero del sistema, incluyendo los directorios, tienen un usuario propietario y por tanto un grupo (cada usuario pertenece a un grupo). De esta manera, el acceso al mismo dependerá de si se es el propietario, de si se es del mismo grupo de usuarios, o de si se es de otro grupo cualquiera. Y es el propietario del fichero el que puede cambiar estos tres permisos de entrada. Los grupos se pueden crear por razones de utilidad (proyectos, intereses comunes, mismo nivel de jerarquía, …) o por razones de acceso hardware (quien puede acceder al sonido, al CD, al bluetooth, …).

A la vez existen tres tipos de acceso al fichero: para leerlo (r), para cambiarlo (w), o para ejecutarlo (x); (en el caso de un directorio hacer un `ls`, incluir un fichero dentro o hacer un `cd` a él) pudiéndose cambiar estos accesos a cada tipo de usuario. Teniendo en cuenta esto, existen en total 9 permisos que el propietario del fichero puede cambiar y que se pueden ver al hacer un listado de un directorio en modo largo, es decir, con la opción -l (recuerda que puede tener un alias ll):

```
-rw-rw-rw- 2 grupo 7 alumnos 3654 Nov 24 12:17 fichero
 uuugggooo
```

El primer trío de la izquierda rwx corresponde a los permisos del usuario (uuu), cuando aparezca una letra significará que se tiene ese permiso y cuando aparezca un "-" que no, los otros dos tríos son los del grupo (ggg) y los de los otros usuarios (ooo).

La primera letra que falta indicará, como ya hemos visto, el tipo de fichero, siendo el "-" el correspondiente a un fichero regular y "d" a un directorio (existen más tipos).

Los permisos de un determinado fichero se pueden cambiar a través de la orden `chmod` (cambio de modo), que es un poco especial ya que no tiene opciones, pero si los permisos que se quieren cambiar, y toma como argumentos los ficheros o directorios afectados. La forma de poner los permisos se compone de tres partes, por un lado, a que tipo de usuarios afecta el cambio, simbolizados por tres letras: la u para el usuario, la g para el grupo y la o para los otros usuarios, por otro lado, si se da o se quita el permiso, simbolizado por + ó -, y por último el tipo de permiso de lectura (r), escritura (w) y ejecución (x). Así, para quitar los permisos de lectura y escritura a todos menos el propietario del fichero *nominas* habría que hacer:

```
chmod go-rw nominas
```

[1] En la página os he dejado un manual básico de administración para que lo tengáis de referencia.

Otra forma de utilizar este comando es dando como permiso un número octal de tres cifras, donde cada cifra es un tipo de usuario y donde cada dígito binario del octal es un tipo de permiso, por ejemplo 777 (sería 111 111 111) daría todos los permisos a todo el mundo. Si quisiéramos hacer ejecutable una macro y poder ejecutarla sin `sh`, podríamos hacerlo de dos formas:

```
chmod +x macro
chmod 755 macro
```

Hay que tener en cuenta que el *inodo* del fichero contiene 12 bits de control de acceso de los cuales 9 corresponden a estos permisos. Los otros 3 bits son conocidos como *setuid*, *setgid* y *sticky bit*. El primero de ellos servirá para que un ejecutable tome como usuario el propietario del ejecutable y no al usuario que lo ha ejecutado. Por ejemplo el fichero `/etc/passwd` sólo puede ser editado por el *root*, sin embargo cuando ejecutamos `passwd` lo podemos cambiar, esto es debido a que `passwd` tiene activado el *setuid*. De forma análoga para el grupo. El tercer bit (el bit pegajoso) se usa también en ejecutables cuando queremos que permanezcan permanentemente en memoria secundaria. Actualmente en Linux sólo se utiliza en directorios cuando queremos que sólo el propietario pueda renombrar/borrar los ficheros. Hay que tener en cuenta que la manipulación de estos bits es peligrosa ya que ciertos agujeros de seguridad se basan en su manipulación.

14.	Elige un fichero (o créalo) y cámbiale los permisos de lectura y escritura para que nadie pueda acceder a él. Intenta mostrar su contenido por pantalla, ¿qué ocurre?

Por defecto, todos los accesos están permitidos, a no ser que esté definido algún otro acceso por el comando **`umask`**. Este comando toma como argumento una máscara (código de inhibición) que es restado al 666 (acceso total sin ejecución ya que los ejecutables lo tienen activado por defecto) cada vez que se crea un fichero (no aplicable a directorios). Normalmente se ejecuta cuando arranca una shell `umask 022` lo que equivale a unos permisos 644 (`rw-r--r--`).

Como hemos dicho todos los ejecutables tienen activados los bits de ejecución, por lo que no son necesarias las extensiones tipo .exe o .com. Date cuenta que también se pueden poner permisos de ejecución a las macros por lo que ya no sería necesario anteponer el "`sh`".

Está claro el significado de los permisos en los ficheros regulares, pero ¿qué significa en los directorios?:

- Ejecución: permite acceder al directorio (cd).
- Lectura: permite listar el contenido del mismo.
- Escritura: permite la creación de nuevos ficheros en su interior.

Para poder aplicar los permisos, en la meta información del fichero (se ve con `ls` en formato largo) debe aparece el propietario y el grupo al que pertenece el fichero. Tanto propietario (usuario) como grupo de usuarios, están codificados por un entero que se llama UID y GID.

Estos se pueden cambiar con los correspondientes comandos:

```
chown [-R] nuevo_usuario fichero (Cambia el UID).
chgrp [-R] nuevo_grupo fichero (Cambia GID).
```

En la distribución de Ubuntu en vez de crear dos usuarios (uno el *root* y otro el personal de usuario) como es recomendable, han optado por sólo crear un usuario por defecto. Cada vez que ese usuario tiene que hacer alguna labor de administración, tiene que anteponer el comando `sudo` a la instrucción (pide el *pasword*) para que funcione. En otras distribuciones se usa el comando `su` para cambiar a administrador temporalmente. Otros comandos para manejarse en el sistema multiusuario son:

```
who  o w              para decir que usuarios están en el sistema.
su usuario            para cambiar temporalmente de usuario.
login usuario         para cambiar de usuario sin retorno.
passwd [usuario]      para cambiar el password de un usuario.
```

Si queremos comunicar usuarios que estén en ese momento en el sistema, usaremos el comando `write usuario` (este comando se puede confundir con la llamada al sistema `write`, de ahí la necesidad de `whatis` y las secciones del manual ya vistas). Con `write` conseguiremos que le aparezca en la pantalla a otro usuario el mensaje que le hemos escrito (terminará en control-d). Si no queremos que nadie nos escriba (puede ser molesto), usaremos `mesg n`. Esto se hace modificando el acceso (`rw`) del dispositivo que estamos usando de terminal.

```
write usuario [terminal]    para mandar un mensaje a otro usuario
mesg y, n                   para permitir o no que nos escriban
```

Hay una forma más sofisticada de comunicarse que es el comando `talk usuario` que divide la pantalla en dos partes una para el emisor y otra para el receptor (no instalada en el sistema). La tercera forma de comunicación off-line sería el conocido `mail`.

Si la comunicación es entre máquinas distintas se usa el protocolo `rsh`, actualmente `ssh` para arrancar una Shell remota en otro computador. Hace falta que en el destino exista el servicio (demonio) de `rsh` - `ssh`.

```
rsh [-l usuario] host.maquina.com "mkdir testdir"
```

15. Crea dos terminales sobre ventanas o usa alguno de texto. Comprueba que estás en el sistema con `who` (`who am i`). Utiliza el comando `write` contigo mismo. Prueba `mesg`. Con el comando `ps` mira el dispositivo que estás usando en la shell. Observa el contenido del directorio `/dev` hasta encontrarlo. Usa la redirección para mostrar con `cat` el contenido de un fichero en el dispositivo que usas (puede ser `tty` o `pts/x`) de forma equivalente a como lo harías con un fichero. Hazlo también para que el contenido salga en el otro terminal.

Hay una excepción en la utilización de los ficheros en los sistemas UNIX, ya que existe un encargado del mismo que tiene privilegios sobre el resto de los usuarios, es el llamado superusuario. Por convención, el nombre del superusuario es *root* y como este nombre indica su directorio de trabajo es el directorio `/root` y tendrá los suficientes privilegios como para acceder a cualquier fichero del sistema, lo único que no puede conocer es nuestro *password*. Cuando el *root* quiere comunicarse con los usuarios utiliza el comando `wall`.

Por último, para compartir los recursos comunes del sistema entre varios usuarios, en concreto la impresora, lo que se utiliza es el mecanismo de *spooling*, de tal manera que cuando varios usuarios intentan acceder a la impresora a la vez, este mecanismo almacena los requerimientos e impone un orden de envío a la misma (se forma una cola de impresión).

Para utilizar el spooling existen varios comandos en UNIX, los más usados son:

```
lp ficheros      para enviar a la impresora un fichero(s) (lpr en LINUX).
lpstat           para ver la cola de impresión (lpq en LINUX).
cancel trabajo   para cancelar un envío anterior (lprm en LINUX).
```

Las variables de la Shell

Otra característica potente de las *shell* es la definición de variables que servirá para guardar datos del usuario y también para modificar el comportamiento de la propia *shell*. Esto se hace o bien a través de la creación de variables de usuario, o bien modificando el valor de las definidas por la propia *shell*.

La definición de las variables depende del tipo de Shell. En el caso de tener la *Shell* `bash` se utiliza su nombre seguido de "=" y el nuevo valor. Hay que recordar que en Linux/UNIX es distinto una letra mayúscula que una minúscula y que las variables se suelen poner en mayúsculas. Si el nombre tienes espacios se debe encerrar entre apóstrofes. Una vez definida la variable, se podrá utilizar en cualquier comando del sistema simplemente anteponiendo a su nombre el carácter "$" (expansión de variable). También se puede poner su nombre entre llaves y si hay espacios con comillas simples o dobles.

Para ver variables en la *Shell* se utiliza el comando interno `echo` (también existe `printf`). Por ejemplo (ten en cuenta que en los ejemplos que te pongo el *prompt* está simbolizado con "$", no confundirlo ahora con el "$" de la expansión de variables) podemos definir y ver variables:

```
$ DIRECTORIO=/home/pepe/proyecto/sesion/trabajo/grupo
$ DIRS='. ~'
$ echo $DIRECTORIO $DIRS
$ echo mis directorios ${DIRS} definidos
```

En cuanto a las variables definidas por la *shell*, las más usadas son (hacer `man bash` para ver todas también con el comando `set` se pueden ver incluso las definidas por nosotros):

```
PATH    Donde se guardan los directorios donde se pueden encontrar ficheros ejecutables.
        PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
HOME    El directorio de trabajo (home) del usuario.        HOME=/home/rafa
PS1     El string que se utiliza como inductor prompt.
        \[\e]0;\u@\h: \w\a\]${debian_chroot:+($debian_chroot)}\u@\h:\w\$
USR     El usuario.                                         USER=rafa
PWD     El directorio donde estamos.                        PWD=/home/rafa
TERM    Tipo de terminal desde el que estamos conectados al sistema. TERM=xterm
SHELL   Tipo de shell que ejecutamos.                       SHELL=/bin/bash
```

En el primer caso la definición de la variable será el conjunto de directorios donde la *shell* va a buscar ejecutables (en otros directorios no se podrá ejecutar nada) separados por ":". Antiguamente se ponía también el directorio "." para poder ejecutar en cualquier directorio, pero era un riesgo de seguridad y se quitó, por eso estamos obligados a poner "./" antes del ejecutable. La segunda variable se puede usar como hemos visto para obtener el directorio `home` de un usuario. La tercera es el *prompt*, alguno de los modificadores que existen (hacer `man bash` para ver todos) para variar su valor son:

```
\d          La fecha.
\H          El nombre de máquina.
\j          El número de jobs.
\s          El nombre de la shell.
\t          La hora.
\u          El usuario.
\w          El directorio actual.
\!          El número del comando dentro de la historia de comandos.
\#          El número de comando.
\$          Será $ en caso de usuario normalo # en caso de root.
```

un caso habitual puede ser: `[\u@\h \W]\$`, que pondría como *prompt* el usuario@ordenador, directorio de trabajo y el $ de siempre.

Debemos tener en cuenta que la definición de una variable sólo tiene efecto en esa sesión de *shell*, si queremos que sirvan para *shell* o programas creados a partir de ella, hay que utilizar el comando `export` (realmente es un comando interno [built-in] de la *shell*) para convertir variables de *shell* en variables de entorno o *environment variables*. Esto es importante para realizar macros (*Shell*

scripts) ya que las macros se ejecutan en una sub-shell; o programas (desde un programa en C se puede tener acceso a las variables de entorno como tercer argumento de *main*).

Para ver las variables de entorno se utiliza el comando `printenv o env` que nos pondrá en pantalla las que tenemos asignadas (si sólo queremos conocer una podremos hacer `echo $variable`). Para borrar una variable se utiliza el comando `unset`.

Si necesitáramos realizar operaciones con las variables, se pueden asignar valores con el comando interno `typeset –i` (i para *integers*):

```
$ typeset –i numero=1
```

En el caso de la `tcsh`, para crear variable se utiliza `set` y para ver las variables de entorno el comando `setenv`.

16. Si estás ejecutando una `bash`, mira el valor de las variables de shell `PS1` y `PATH` (entorno) de dos formas; utilizando `echo` y una canalización con el `grep`. Arranca una shell de tipo `tcsh` (si está instalada) y mira el valor esta vez, si están indefinidas piensa el porqué. Sal de la `tcsh`. Mira todas las variables que tienes definidas, averigua la shell que estás usando y el tipo de terminal. Cambia el valor de PS1 para que te ponga un mensaje personal terminado en $.

17. Crea una variable de shell para asignar el nombre de un directorio a esa variable, comprueba su contenido y después úsala con el comando `cd`. Comprueba con grep si existe como variable de entorno, conviértela a entorno y comprueba de nuevo. Por último, elimínala.

El comando `set` también se utiliza para ver y cambiar las opciones definidas en la *Shell,* por ejemplo `set –o` pondrá las opciones actuales y para ponerlas o quitarlas se usa `–o` o `+o` (ojo + para quitar), algunas ya las hemos visto como `set –o emacs` o `vi` para recupera y editar el histórico de órdenes.

Ficheros de configuración

Cuando arrancamos un terminal y sobre él se ejecuta automáticamente la Shell, antes ésta ejecuta una serie de macros que definirán su comportamiento. Estás macros normalmente están en el directorio home de cada usuario y empiezan por punto, para estar ocultas.

Normalmente la secuencia de carga es:

- bash:
  `/etc/bashrc` → `/etc/profile` → `$HOME/.bashrc` → `$HOME/.bash_profile`

- tcsh (csh)

`/etc/csh.cshrc` → `/etc/csh.login` → `$HOME/.tcshrc(.cshrc)` → `$HOME/.login`

Pero en algunos sistemas esto no es así por tanto siempre conviene revisarlo, por ejemplo pueden no existir los ficheros de usuario, pero haber un `.profile`.

Debemos saber que por ejemplo las variables de Shell sólo están definidas en la sesión de trabajo, si queremos crear una variable nueva o modificar las existentes de Shell para siempre debemos definirla o redefinirla en esos ficheros. También se usa para renombrar nuevos comandos con `alias` (ver apartado siguiente).

18. Averigua los ficheros de configuración que tienes ocultos en tu directorio personal. Examina el aspecto del que tienes que modificar para crear variables. Modifícalo (al final) para cambiar la variable adecuada para redefinir tu inductor (*prompt*).

Creación de macrocomandos (macros): shell scripts

Un aspecto importante de las shell de UNIX, es que se pueden programar y hacer macrocomandos compuestos de la ejecución secuencial de otros comandos más simples. Es lo que se llama en UNIX *shell scripts* (guion) o normalmente macros. Esta característica es muy útil cuando se utiliza a menudo una secuencia de comandos, ya que nos permitirá repetirla fácilmente con un sólo nombre. Como se ha visto en teoría esto constituye el actual procesamiento *batch*.

Básicamente, un macrocomando es un fichero de texto que tiene el atributo de ejecutable (se verá como cambiarlo) o puede ser ejecutado con `sh macro` y en el que hemos escrito una serie de comandos que queremos ejecutar secuencialmente, puede incluir redirección, canalización y *background*. La estructura de las macros puede llegar a la misma complejidad que la de un programa de alto nivel, de hecho, dentro de una se pueden utilizar las mismas estructuras de control (selección e iteración) que en un programa y también se puede hacer uso de variables, tanto definidas por el usuario como por la propia *shell*.

Una vez escrita la macro con el editor de textos, podremos ejecutarla, para ello tendremos que arrancar una mini-shell `sh` (Bourne) a la que le damos su nombre. Cuando termine de ejecutar la macro también terminará la nueva sub-shell. Otra forma de ejecutarla es cambiando el modo del fichero haciéndolo ejecutable como se verá en el siguiente apartado, en este caso se utiliza una *Shell* como la invocante. Otra forma es poniendo punto para que no se llame a otra *Shell*.

```
$ sh macro    si es que hemos llamado a la macro macro
$ . .bashrc   es útil para ejecutar los ficheros de configuracion
```

Además, una macro puede recibir datos desde el exterior en la propia línea de comandos, estos datos son conocidos como argumentos posicionales y pueden ser usados dentro de la macro antecediendo su posición con el símbolo $ (ya que realmente son variables), de tal manera que $0 es el nombre de la macro, $1 el primer argumento, etc. No es usual poner más de 10 argumentos, aunque si se diera el caso, se podrían tomar con el comando interno `shift`.

Las variables más comunes son (antecediéndolas el $ para obtener su valor):

```
$1 a 9   Variables de argumentos posicionales.
$0       El nombre de la propia macro.
$#       El número de argumentos dados.
$?       El código exit del último comando ejecutado, 0 sin errores.
$!       El identificador de PID del último proceso ejecutado.
$$       El identificador de la Shell ejecutante
$*       Un string conteniendo todos los argumentos posicionales.
```

Las macros se han utilizado algunas veces para redefinir comandos de otros sistemas operativos o para simplificar los existentes. Por ejemplo, podríamos hacernos nuestro propio comando `ls` como `dir` (estilo MS-DOS) escribiendo un fichero de texto que se llamara `dir`. Para ejecutarlo haríamos `dir directorio` y su contenido sería:

```
ls -l $1
```

Para estos casos triviales, no se usan macros, sino el comando interno `alias`:

```
$ alias cd.. = 'cd ..'      para incluir la eliminación del espacio
$ alias dir = 'ls –l'       para tener un dir de MSDOS
```

en estos dos ejemplos hemos redefinido el comando `ls` como `dir` y hemos evitado un error frecuente que se comete al no dejar espacio entre `cd` y el directorio "..". Si quisiéramos eliminarlos utilizaríamos el comando interno de la *Shell* `unalias`.

19. Vete a los ficheros de configuración de la Shell y comprueba si hay alias definidos. Si no es así crea un alias que se llame `ll`, si existe crea uno que te parezca útil.

Dentro una macro se puede acceder a las variables de entorno y también definir variables locales y leerlas desde teclado. Para ello se usa el comando interno `read` (si hacéis un `man` de `read` os daréis cuenta de la importancia de poner la sección adecuada en el `man`, ya que existen dos, el perteneciente a la shell y la llamada al sistema `read`, ojo, en sistemas Ubuntu actuales no aparece).
El siguiente ejemplo nos demuestra cómo crear variables y utilizar comandos dentro de una macro (un # indica que la línea es un comentario):

```
# Esta macro usa variables y los comandos echo y read
echo "Por favor entra tu apellido"
echo seguido de tu nombre:
read nombre1 nombre2
echo "Bienvenido a UNICAN $nombre2 $nombre1"
printf "otra forma de pintar \n"
```

El comando `echo` (existe también el comando `print y printf`) sirve para poner un mensaje en pantalla, tanto con comillas como sin ellas, `read` leerá desde teclado los valores de variables, en este caso `nombre1` y `nombre2` y por último se pintarán estos valores usando el $ delante de la variable.

Si nos interesa especificar la *Shell* con la que se va a ejecutar la macro y que no coja una por defecto, como primera línea se puede poner `#!` y el nombre absoluto para llegar:

```
#!/bin/sh     #!/bin/bash    …
```

También se pueden hacer evaluaciones numéricas (recuerda el comando `typeset`, será más rápido) y operar con las variables con el operador doble paréntesis "`(())`". La *Shell* maneja valores enteros por defecto, si queremos trabajar con decimales se usa el comando `bc`. Los operadores numéricos son: " `+ - * / %` ". Veamos un ejemplo de evaluación:

```
EDAD=22                    o   typeset –i EDAD=22
MAYOR=$EDAD+25             el valor será literal
echo $MAYOR
MAYOR=$(($EDAD+25))        el valor será numérico
printf "$MAYOR \n"         otra forma de pintar
echo $((1+1))
echo 3/4|bc -l             otra operacion echo "1.2 * 2.4" | bc -l
```

Sabiendo usar variables, la entrada/salida y la evaluación, pasamos al siguiente aspecto que son las sentencias selectivas e iterativas. Como en C, el primer paso para hacerlo es poder construir expresiones **condicionales** y lógicas, para ello tenemos el comando `test` (los valores de las expresiones condicionales son contrarios a los del lenguaje C, en la *shell* 0 es TRUE y distinto de 0 es FALSE, esto es debido a que cuando un programa funciona bien devuelve un valor de *exit* de 0 que puede ser obtenido con la variable $?). `test` adopta la forma de cualquier comando: el nombre, las opciones y los argumentos (más información con `man test`), las opciones nos dirán lo que tenemos que comprobar (testear). En lo que se refiere a ficheros como argumentos, las principales opciones son: -e para existencia, -d para directorio y -f para fichero regular (hay muchas más http://wiki.bash-hackers.org/commands/classictest).

```
test -e $fichero    Comprueba si el fichero $fichero (variable) existe o no.
```

En la mayoría de las *Shell,* en una macro en vez de utilizar `test` se puede usar [ ] (¡ojo con los espacios!) que es equivalente: [-e fichero]:

```
[ -e $fichero ]   Comprueba si el fichero $fichero (variable) existe o no.
```

También admite argumentos de tipo cadena y enteros sobre los que puede establecer comparaciones. Para la `bash` (ojo con los espacios):

```
(cadenas)  =  !=
(números) -eq –ge –ne –gt –le –lt
```

Y poder usar expresiones lógicas entre ellos: AND (-a), OR (-o) y NOT (!)

```
[ cadenas  = otra_cadena ] –a ![ tercera != cuarta ]
[ entero -eq otro entero ] –o [ otro –ne otromas ]
```

Además, si usamos la `tcsh` (es la única ventaja de usar esta *Shell*), en las expresiones también se pueden utilizar los mismos operadores lógicos que en el lenguaje C e incluso se pueden utilizar operadores aritméticos. Los más importantes para `tcsh` son:

```
== != <= >= < > || && !       | & ^ ~ >> << + * - / % ( )
```

Las expresiones construidas con `test` o `[ ]`, o la ejecución de cualquier comando (que como hemos visto también es una expresión lógica) son utilizadas para realizar las sentencias de selección e iteración:

```
if [ expresión ]
then
     código si 'expresión' es verdadera
fi
```

Existe la variante de ponerla sentencia de selección como `if [ expresión ]; then` . Unos ejemplos de utilización serían:

```
FILE=~/.bashrc        # definicion de variable
if [ -f $FILE ]       # si es fichero
then
   echo el fichero $FILE existe
fi

if test -e $FILE       # otra forma
then
   echo el fichero $FILE existe
fi

echo $#              # pinto los argumentos pasados a la macro
if [ $# -eq 0 ]      # si no hay
then
     printf “no has pasado argumentos \n”
fi

if [ rafa = rafa ]; then  echo si; fi

# ejecutado con dos argumentos: sh macro argu1 argu2
#dentro de la macro argu1 es $1 y argu2 es $2, $0 es el nombre macro
echo $1 y $2
if [ ! $1 = $2 ]          #o if ! [ $1 = $2 ]
then
   echo son distintos
fi
```

En este caso estamos usando el comando `test` combinado con la sentencia `if` (terminado con `fi`). También lo hacemos con `[ ]`. Comprobamos el número de argumentos numéricamente y miramos si dos cadenas son distintas con `not`.

Como ocurre en un lenguaje de alto nivel las sentencias se pueden anidar y puede haber `if` dentro de `if`.

También existe la combinación de la sentencia `if` con la `else`, al igual que estas estructuras se pueden anidar (`else if` o su contracción `elif`) su estructura sería:

```
if condicion
then
     comandos (condición es true)
else
     comandos (condición es false)
fi
```

Se pueden hacer comprobaciones más complejas utilizando los comandos existentes en la *shell*, ya que sabemos que la ejecución de un comando devuelve un código exit que es lógico:

```
#!/bin/sh    # se ejecuta con la sh
#USER=rafa
if who | grep $USER > /dev/null         # no me interesa la salida
then
    echo $?                             #pondrá 0
    echo $USER esta en el sistema       #si funciona el grep
else
    echo $?                             #pondrá 1
    echo no esta                        #si no funciona
fi
```

En este caso utilizamos el comando `who` (del ambiente multiusuario) combinado con el comando de búsqueda `grep` para saber si un usuario está en el sistema, la salida estándar se desvía a `null` para que no aparezca en pantalla.

Existe también la construcción `case`, que compara una palabra dada con los patrones de la sentencia, como siempre termina con la sentencia inversa `esac`:

```
case palabra in
     patron1) comando(s)  ;;
     patron2) comando(s)  ;;
     patronN) comando(s)  ;;
esac
```

Ejemplo:

```
#!/bin/sh
echo $1
case $1 in
       [0-9] ) echo "numero" ;;
       [a-z] ) echo "minuscula" ;;
       "" ) echo "no hay argumento" ;;
       * ) echo "fallo de argumento" ;;
esac
```

También existen sentencias iterativas (bucles), la primera de ellas es la sentencia `for` que utiliza una variable de índice que se puede mover entre los valores de una lista de valores, por ejemplo $* (todos los argumentos posicionales) que será tomado por defecto. La estructura es (como en el `if`, la primera línea también puede acabar en `; do`):

```
for var in lista de palabras     # ejemplo 1 2 3 4 5 o uno dos tres
do
     comandos $var
done
```

Un ejemplo de utilización del bucle `for` se ve a continuación, donde pasamos a una macro una serie de nombres de usuarios para saber si están en el sistema o no están:

```
for i in $*    #se podría poner for i solamente en el método abreviado
do
     if who | grep $i > /dev/null
     then
          echo $i esta en el sistema
     else
          echo $i no esta
     fi
done
```

La lista de nombres también se puede obtener de la ejecución de un comando con "$( )" o con "``" como en estos dos ejemplos:

```
for i in $( cat num ); do  #lo cogera de lo escrito en num
     echo item: $i
done

for i in `ls`              #lo cogera de los ficheros del directorio
do
     echo fichero: $i
done

for i in `seq 0 9`         #commando seq o for i in 0 1 2 3 4 5 6 7 8 9
do
     echo entrada$i
done
```

20. Las macros también funcionan igual en línea de comando, separando cada línea por ";". Crea con una única línea (y un bucle `for`) cinco ficheros vacíos que vayan de vacio1 a vacio5.

Otros tipos de bucle son el `while` y el `until` cuyo significado es claro (uno de permanencia otro de salida) y cuya estructura aparece a continuación:

```
while comando                    until comando
do                               do
    comandos                          comandos
done                             done
```

En los siguientes ejemplos hacemos un bucle infinito y podemos esperar a que un usuario salga del sistema (`sleep` 60 parará la ejecución durante un minuto):

```
while :          # : es un comando interno que devuelve TRUE (0)
do
     echo siempre
done

while who |grep $1 >/dev/null
do
     sleep 60
done
echo "$1 se ha ido"
```

También se pueden emplear expresiones numéricas (en muchos sistemas no existe en comando interno `let` pero se puede sustituir por:

```
limite=5
i=0
while [ $limite -ge $i ]
do
     echo Acción $i ejecutada
     #let i=$i+1
     i=$(($i+1))    #sustituye al let de arriba
done
```

Por último, para acabar el apartado de las macros, debemos saber que también podemos hacer funciones con `function nombre { }` y variables locales con `local` , pero como en el caso del `let`, no en todos los sistemas existen.

21. Edita con `vi` un fichero de texto. `vi` o `vim` es un editor de texto de pantalla básico, tienes un pequeño manual en la página. El uso de `vi` a pesar de ser un editor primario, está justificado por su carácter universal y no depender de las características de un teclado particular. Al menos una vez en la vida es necesario su uso. Incluso, si te acostumbras a él, será más eficiente que cualquier otro editor gráfico).

22. Ese fichero será una macro que servirá para saber el tamaño en Kb o Mb de una serie de ficheros de un directorio. La macro tomará estos ficheros por argumento y pedirá el directorio con lectura. Sólo mostrará el tamaño de los ficheros si existen y si está definida una variable MENSAJE puesta a SI. La variable MENSAJE en un caso estará definida desde fuera de la macro y en el otro caso de forma interna ¿puedes acceder a la variable interna definida en la macro desde la shell?

Certificaciones (solo info)

Debes saber que existe un instituto el LPI (Linux Professional Institute ─ http://www.lpi-spain.es ─) que ofrece certificaciones profesionales que pueden ser de ayuda para el mercado laboral. LPI tiene dos tipos de certificaciones: **Linux Essentials** y **LPIC** (1,2,3):

- **Linux Essentials** está pensado para novatos y aquellos que quieran comenzar su andadura en el mundo del Sofware Libre. Al aprobar el examen se obtiene el "LPI Linux Essentials Professional Development Certificate".
- **LPIC** Estas certificaciones LPI han sido diseñadas para certificar la capacitación de los profesionales de las Tecnologías de la Información usando el Sistema Operativo Linux y herramientas asociadas a este sistema.

Ha sido diseñado para ser independiente de la distribución y siguiendo la Linux Standard Base y otros estándares relacionados. Estas certificaciones LPI están orientadas al puesto de trabajo a desempeñar utilizando para ello procesos de Psicometría para garantizar la relevancia y calidad de la certificación.

Actualmente existen tres niveles de certificación profesional:

- LPIC-1 Linux Server Professional
- LPIC-2 Linux Network Professional
- LPIC-3 Linux Enterprise Professional (especialidades 300, 303 y 304)

Volviendo al Linux Essentials que es el que está relacionado con esta parte, comprende cinco tópicos (en inglés https://www.lpi.org/study-resources/linux-essentials-exam-objectives/):

1. The Linux community and a career in open source
2. Finding your way on a Linux system
3. The power of the command line
4. The Linux operating system
5. Security and file permissions

El primero es literatura que se puede estudiar a solas, el segundo, el tercero y el quinto los has visto en las sesiones precedentes y el cuarto está dedicado al hardware.

Fin de la segunda sesión

Entorno de desarrollo

Un entorno de desarrollo se compone fundamentalmente de tres tipos de herramientas:

1. Editor de texto para la edición de código fuente. Puede ser:
    - De texto. De pantalla como `vi` (`vim`) o de línea como `ex` o `ed`.
    - Gráfico. Dependerá del entorno de ventanas en uso o será independiente:
        1. `kedit` , `gedit` o `sublime`.
        2. `emacs` o `nedit`. Cuyas propiedades principales son:
        3. Marcado sintáctico (muy útil para programar). También el `vim`, el `gedit` o `sublime` lo tienen si están configurados.
            - Compilación integrada.
            - Altamente configurable.

2. Compilador y afines: Herramienta que permite convertir secuencias de control en alto nivel a lenguaje máquina.

    - `gcc` en modo línea de comando.
    - `makefile` para compilación automatizada.

3. Depurador. Herramienta que permite comprobar en tiempo de ejecución porque un programa no funciona. Puede ser en:

    - Modo línea de comando: `gdb`.
    - Modo gráfico: `ddd`.

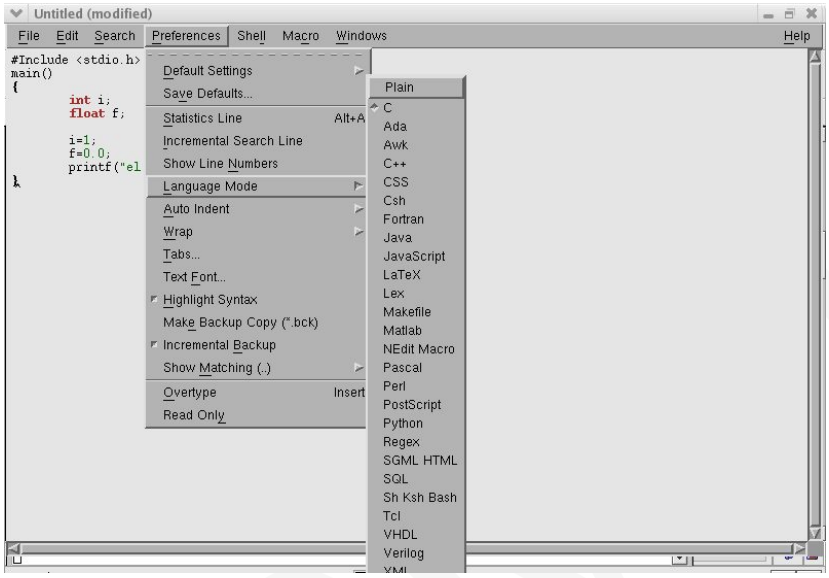
Incluso existen otros tipos de herramientas que permiten la colaboración entre usuarios para la realización de un proyecto como por ejemplo el control de versiones –`git`– y repositorios que serán tratados en otras asignaturas.

También podemos disponer de entornos integrados de desarrollo (IDE) que agrupan todas estas herramientas en una única y pueden estar dedicados a varios lenguajes de programación como

Eclipse, ActiveState Komodo, IntelliJ IDEA, Oracle JDeveloper, NetBeans, Codenvy o Microsoft Visual Studio; o a uno cerrado.

Hay otros, que estando basados en un lenguaje concreto, soportan varios a través de plug-in como GNU Emacs basado en C también para ADA, Lisp, PHP, Perl, …; o Eclipse o NetBeans, basados en Java, o MonoDevelop, basados en C# sirven para otros tantos.

También existen IDE que vienen con el propio gestor de ventanas como Kdevelop, Builder, Anjuta, …



En nuestro caso vamos a trabajar con las herramientas de forma independiente. Haciendo una analogía, vamos a aprender a sumar antes de usar la calculadora.

De los editores no hay mucho que decir salvo que ya se ha utilizado el `vi - vim` para la escritura de macros y se han visto sus carencias a pesar de su universalidad. Por eso a partir de aquí se podrá usar cualquiera de los editores del sistema. Pasemos por tanto a la siguiente herramienta.

Compilador

El compilador de C que vamos a utilizar en el laboratorio es el `gcc` (g++). Este compilador realiza automáticamente toda la cadena de operaciones para producir un fichero ejecutable, es decir:

1. Llama al preprocesador. `cpp` es el preprocesador.
2. Compila el programa, produce un fichero en ensamblador.
3. Ensambla el programa, `as` es el ensamblador.
4. Lo enlaza (*linka*), `ld` es el enlazador (*linker*), y produce el ejecutable.

El compilador siempre espera que se le dé un nombre de fichero que contenga un programa fuente de lenguaje C, para distinguir estos ficheros de código fuente del resto de ficheros del sistema, es obligatorio que todos terminen en ".c". Si no hacemos esto, el compilador nos responderá con: `"nombre: file not recognized: No se reconoce el formato del fichero"`. Otras terminaciones comunes son:

- .C, .cc, y .cxx para ficheros en c++.
- .h, .i para ficheros del preprocesador.
- .s, .S para ficheros en ensamblador.
- .o para ficheros objeto (sin enlazar).

Como cualquier otro comando del sistema, el formato para ejecutarlo será nombre [-opciones] y argumentos. La forma más sencilla de ejecutarlo para producir un fichero ejecutable con nombre `a.out` sería la siguiente:

```
gcc programa.c
```

Como hemos dicho, el compilador es un comando más, por lo que también admite opciones, éstas usualmente se pondrán delante del nombre del fichero fuente y deberán estar separadas, por ejemplo, no es lo mismo poner `gcc -dr` que `gcc -d -r`. Las opciones están divididas en varios grupos: globales, del lenguaje, de alerta, de depuración, de optimización, de preprocesado, de ensamblador o linkador, de directorios y de dependencias del hardware.

Las opciones más habituales (las puedes encontrar con `man gcc`) son:

Globales:

-c	Compila pero no llama al linker, generando un .o.
-o ejecutable	Cambia el nombre del fichero ejecutable (por defecto `a.out`) al expresado en la opción.

Del lenguaje:

-ansi	Compila siguiendo las reglas del ANSI C.
-std=std	Compila siguiendo el estándar *std*, por ejemplo c99.
-E	Ejecuta sólo el preprocesador.

Del linker:

-llibreria	Incluye la librería libreria. Se pone al final de la línea.

De directorios:

-Ldirectorio	Añade el directorio directorio a los directorios donde se buscará lo dado por -l (anterior opción).
-Idirectorio	Añade el directorio a la lista de directorios donde están los *includes*.

De alerta:

-w	Inhibe los mensajes de *warning*. Los mensajes de *warning* indicarán algo a tener en cuenta, pero que no es un error, es decir, produce el ejecutable.
-Wall	Pone todos los mensajes de *warning*.

De depuración:

-g	Introduce información adicional en el ejecutable para poder utilizar alguno de los depuradores (*debuggers*) del sistema (`gdb`).

De optimización:

-O	-O0 Sin optimizar, -O1 Optimización razonable, -O2 mejor, -Os mejor sin incrementar tamaño, -O3 agresiva.

Una vez que se ha compilado un programa con la opción -g, puede ser ejecutado con el *debugger* del sistema (gdb –línea– o ddd –gráfico–). Esto permitirá ejecutarle de forma controlada (paso a paso, línea a línea, hasta un punto de ruptura, …), ver los valores de las variables, poner guardianes para cambio de variables, etc. Para más información se puede utilizar el comando *help* del mismo. También se puede utilizar el depurador para obtener información de los ficheros *cores* del sistema. Cuando un programa se aborta en ejecución en algunos casos se produce un fichero de nombre `core` que puede ser analizado con `gdb core`. Se puede forzar su aparición con `ulimit -c unlimited`.

El compilador no sólo toma como argumentos ficheros fuente, si hemos hecho el programa en módulos (módulos objeto compilados, ficheros cabecera, ficheros del preprocesador, …). Un ejemplo de compilación de este tipo sería:

```
gcc modulo1.c modulo2.c prepro.i programa.c objeto.o -o ejecutable -lm
```

23. Vete a la asignatura virtual y bájate el fichero comprimido de ejemplos para el sistema de desarrollo. Selecciona el fichero `min.c`. Ábrelo con el editor de textos y revísalo. Después compílalo con el gcc. Usa a continuación varias opciones para ver el resultado como `-ansi, -std=c99` o `-Wall`.

24. Ejecuta el programa. Verás que te pide una serie de datos. Bájate el fichero `datos` y utilízalo con la redirección de entrada. Usa también la redirección de salida. Introduce un error en el programa e indica hacia que canal va la salida del compilador. Usa la opción `-o` para generar un programa ejecutable distinto de `a.out`. Con las opciones `-g` y `-static` e indica cómo cambia el tamaño del ejecutable.

25. Selecciona los ficheros `main.c, cap.c y cal.c`. Ábrelos con el editor de textos y revísalos. Crea una línea de compilación para producir un ejecutable con estos ficheros. Usa la opción `-c` con `cal.c` para producir un fichero objeto y después repite la línea de compilación con este objeto. Revisa el fichero `partes.c` e indica como lo compilarías.

26. Selecciona el fichero `cal2.c`. Ábrelo con el editor de textos y revísalo. Indica la diferencia con el anterior. Crea una línea de compilación para producir un ejecutable (usa la librería matemática, revisa también las pruebas comentadas en `main.c`). Indica que es `math.h` y la opción `-lm`. Usa la opción `-c` con `cal2.c` para producir un fichero objeto y después repite la línea de compilación con este objeto.

Cuando el programa está compuesto de varios ficheros fuente es muy engorroso y poco eficiente usar la línea de comando. Por ello se utilizan otras herramientas más sofisticadas como el `make`.

La herramienta *make*

Otra herramienta interesante cuando se hacen grandes programas que están compuestos de muchos ficheros es `make`. Esta se utiliza con un fichero de comandos donde decimos de que ficheros está compuesto nuestro programa, que librerías utiliza y como se pueden obtener los ejecutables. Hay que recordar que un programa en C puede estar compuesto de varios ficheros, uno de ellos tendrá la función `main()` y los otros serán otras funciones ya compiladas con la opción -c (sin llamar al *linker*), código fuente, librerías o ficheros de cabecera `.h`.

La ventaja de utilizar el `make`, es que nos compilará sólo los ficheros que sean necesarios según la fecha y hora del último cambio y que no tendremos que poner una complicada línea de comando para realizar la compilación, sólo `make,` el cual leerá el fichero de órdenes `makefile` (es el nombre por defecto), donde encontrará como hacer la compilación.

El fichero de órdenes se compondrá de una serie de *targets* (objetivos) a cumplir (indicación para el usuario) y de la descripción de cómo realizarlos (indicación para la máquina), las líneas de realización siempre empiezan con el carácter `tabulador` (ojo que nadie ponga tab), pudiendo haber varias para un solo objetivo.

```
programa : main.o   lib_uno.o   lib_dos.o
tab      gcc -o programa main.o lib_uno.o  lib_dos.o

main.o : main.c
tab      gcc -c main.c

lib_uno.o : lib_uno.c
tab      gcc -c lib_uno.c

lib_dos.o : lib_dos.c    incluido.h
tab      gcc -c lib_dos.c
```

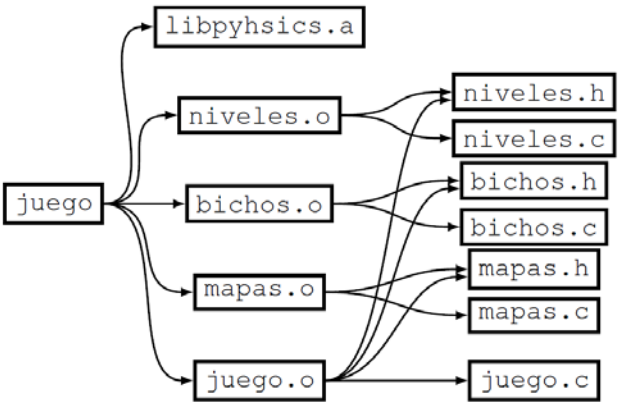
En este ejemplo el fichero `makefile` estará compuesto de cuatro objetivos: `programa, main.o, lib_uno.o, y lib_dos.o`. El primer objetivo es el fichero ejecutable `programa` cuyas dependencias aparecen a continuación de los ":". En la siguiente línea (empieza por el carácter tab) se dice como se puede obtener, en este caso compilando e incluyendo los ficheros objeto. A continuación se indica

como obtener estos objetivos secundarios, en este caso compilando con la opción -c para producir ficheros objeto. Se observa que el objetivo lib_dos.o tiene dos dependencias: lib_dos.c e incluido.h, ya que lib_dos.c tiene una instrucción de tipo #include <incluido.h>.

Como hemos dicho si el fichero se llama `makefile` bastará con escribir `make` para hacer la compilación, si lo hemos llamado de otra manera deberemos usar la opción `-f`:

```
make –f ficherodemake
```

27. Dada la figura posterior, escribe una línea de comando para realizar la compilación del programa juego. Después escribe un fichero *make* atendiendo a las dependencias descritas.



Al igual que en las macros, en los ficheros *make* también se pueden usar variables, incluso emplear las variables de entorno definidas en la Shell. Hay bastantes definidas por defecto que se pueden redefinir, por ejemplo, `CC`, `CFLAGS` u `OBJECTS` (en algunos casos se utiliza := en vez de =):

```
CC = gcc –O2
OBJECTS = main.o lib_uno.o lib_dos.o
programa : $(OBJECTS)
      $(CC) -o programa $(OBJECTS)
main.o : main.c
      $(CC) -c main.c
lib_uno.o : lib_uno.c
      $(CC) -c lib_uno.c
lib_dos.o : lib_dos.c    incluido.c
      $(CC) -c lib_dos.c
```

O usar las variables de entorno:

```
SRC = $(HOME)/src
juego:
tab       gcc $(SRC)/*.c –o juego
```

Al estilo de las variables de la *bash* existen las siguientes variables automáticas en *make* que también se pueden usar (hay muchas más):

```
$@: Se sustituye por el nombre del objetivo de la presente regla.
$<: Se sustituye por la primera dependencia de la presente regla.
$^: Se sustituye por una lista de dependencias de la presente regla.
$?: Se sustituye por una lista de dependencias de recientes.
```

Así, se podría convertir una regla como esta:

```
CC=gcc
CFLAGS=-Wall -g

hola: hola.o auxhola.o
tab    $(CC) -o hola hola.o auxhola.o

#convertida con variables automaticas
hola: hola.o auxhola.o
tab    $(CC) -o $@ $^
```

También hay una serie de reglas (*rules*) predefinidas de sólo una línea, así se podría sustituir el tercer objetivo por esto:

```
lib_uno.o :
```

28. Haz un fichero make para la compilación automatizada de los tres ficheros `main.c`, `cap.c` y `cal.c`. Pruébalo. Vuelve a ejecutarlo ¿qué ocurre? Cambia la fecha de algunos de los ficheros fuente, ejecútalo y observa que ocurre. Cambia tu fichero `makefile` (haz una copia) para usar en este caso `cal2.c` Cópialo de nuevo y produce una versión para depuración. Copia de nuevo el fichero y haz una versión con las reglas por defecto y variables automáticas.

Por último, existen una serie de reglas virtuales que complementan a las reglas normales que sirvan para realizar una determinada acción dentro de nuestro proyecto. El ejemplo más típico de este tipo de reglas es la regla *clean*, utilizada para "limpiar" de ficheros objeto los directorios que haga falta, con el propósito de rehacer todo la próxima vez que se llame a *make* o para borrar ficheros no deseados como los *core*. Sólo se ejecutará si hacemos `make clean`.

```
clean:    # para cores y objetos
tab     rm core
tab     rm –f juego *.o
```

Otro objetivo virtual bastante utilizado es:

```
print:
tab     lp *.c
```

que sirve para imprimir los ficheros que interesen.

Como norma general, se puede poner un objetivo que sea una combinación de comandos, que sólo se ejecutarán cuando se haga explícitamente `make objetivo` (si no se pone nada se ejecutan las reglas normales).

29. Añade al fichero *make* la regla virtual *clean* y ejecútala. Comprueba después que al realizar *make* se realiza de nuevo la complicación desde cero.

Programación separada (modular) - No evaluable, Opcional.

Hasta ahora hemos visto que todos nuestros programas estaban en un fichero que se edita, compila y ejecuta. Pero normalmente, en programas grandes, esto no se hace así, sino que se construye el programa de forma modular en varios ficheros, aplicándose el principio de *divide y vencerás*, ya que los módulos del programa serán más fáciles de entender y depurar (algo parecido a la división de un

programa en funciones, pero a otro nivel más elevado). Con esta forma de trabajar conseguimos algunas ventajas:

1. Obviamente los módulos tienen una extensión menor que el programa completo. Por lo tanto, éstos serán más fáciles de analizar.
2. Cada módulo se puede compilar por separado (más rápido) y será más fácil de depurar, ya que no se tendrán que tener en cuenta influencias externas.
3. La división del trabajo entre varios programadores es más sencilla y limpia de realizar.

Esto conlleva que sea conveniente aplicar ciertos criterios a la hora de construir ese programa utilizando la estructura en árbol de directorios y ficheros:

1. Se puede utilizar un directorio (en vez de un fichero como antes) para contener los ficheros de los que va a estar constituido el programa.

2. Dentro de ese directorio general se puede crear varios subdirectorios donde se sepa que se va a encontrar lo que estamos buscando, como por ejemplo:
   2.1. Un directorio para los ficheros fuente.
   2.2. Un directorio para los ficheros de cabecera del preprocesado (normalmente "include"). Ver punto 4. Los del sistema están en el directorio "/usr/include".
   2.3. Un directorio de librerías (normalmente "lib").
   2.4. Un directorio de ejecutables (normalmente "bin").
   2.5. Un directorio de documentación (normalmente "doc").

3. Los módulos tienen que ser construidos (división del programa) teniendo en cuenta principios semánticos, ofreciendo servicios a otros módulos externos (cajas negras), de tal manera que se garantice el perfecto funcionamiento de los mismos de forma aislada. También se deberá tener en cuenta que[1]:
   3.1. Hay partes dependientes del hardware que deben ser señaladas como tal. De hecho, por definición no son transportables a otros sistemas y deben estar separadas del resto del programa.
   3.2. Otras dependerán de algo específico como llamadas a un sistema operativo concreto y deberán ser tratadas de la misma forma.

4. Los ficheros de cabecera "*.h" están destinados normalmente a contener las definiciones comunes a varios módulos. En ellos suelen aparecer distintos tipos de información:
   4.1. Definición de constantes.
   4.2. Definición de tipos de datos.
   4.3. Definición de prototipos de funciones.

   Suele ser una mala práctica de programación incluir las propias definiciones (reservas de espacio) de variables.

Gestión de librerías

Anteriormente se ha comentado que podemos incluir en nuestro programa, código objeto realizado en otros lenguajes o en el mismo C, así la línea de compilación que veíamos en el anterior apartado podría complicarse:

```
gcc modulo1.c modulo2.c prepro.i programa.c objeto.o -o eje -lm
```

donde hemos incluido al compilar dos módulos de código fuente C, un fichero de preprocesado, el programa principal fuente, un código objeto (no necesariamente C) y una librería, en este caso la librería matemática (para ello deberemos haber usado en alguna parte un `#include de math.h`).

Una pregunta que podemos hacernos es por qué hemos incluido esa librería. La respuesta es porque con el fichero de cabecera `math.h` sólo hemos incluido definiciones de constantes, tipos de datos y

[1] Gran parte de los inconvenientes del código dependiente se pueden solventar con la compilación condicionada que nos proporciona el preprocesador. Referencias más amplias de él las podemos encontrar en el libro de Kernigham.

prototipos de funciones, pero no el cuerpo compilado de estas funciones, que está contenido precisamente en esa librería (`libm.a`), ya que estas funciones no son de uso general. Ocurre lo contrario con las librerías de manejo de la entrada/salida y funciones comunes, que sí que se incluyen por defecto al realizar los ejecutables, estas librerías son `crt0.o` y `libc.a` (podemos considerar una librería como un conjunto de ficheros que contienen código objeto, las propiedades de esos ficheros son asimiladas en la propia librería que los mantiene a través de un registro índice).

De la misma manera que existen librería predefinidas, podemos crear las nuestras. Para ello existe el comando `ar` , que nos permite añadir módulos, modificarlas o eliminarlas (quitar módulos). Como cualquier comando, la sintaxis del mismo incluye opciones y argumentos:

```
ar -[opciones] [módulos] librería [ficheros]
```

Las opciones más habituales son:

Opción	Significado
d	Borrar módulos a través de los ficheros indicados
m	Cambia de orden (mueve) un módulo en la librería
p	Pinta en pantalla un módulo a través de su fichero
q	Añade de forma rápida módulos (sin registro índice) al final
r	Reemplaza módulos a través de su fichero
t	Muestra el contenido de la librería
x	Extrae módulos a través de su fichero
o	Preserva la fecha original del módulo en la extracción
s	Crea o actualiza el registro índice
u	Reemplaza teniendo en cuenta la fecha
Modificador	
a	Lo coloca detrás de un módulo existente
b, i	Añade delante de un módulo existente
c	Crea una librería
v	Modo "verbose"

De esta manera si tenemos una librería que se llama `libre.a` y tres módulos `mod1.o`, `mod2.o` y `mod3.o` podemos hacer:

Ejemplo	Acción
`ar c libre.a`	Crea la librería
`ar r libre.a mod1.o`	Añade el módulo y crea la librería si no existe
`ar tv libre.a`	Muestra el contenido de la librería
`ar q libre.a mod2.o mod3.o`	Coloca al final de forma rápida el módulo
`ar s libre.a`	Actualiza el registro índice
`ar x libre.a mod3.o`	Extrae el tercer módulo

A la hora de usar la librería creada tenemos que tener en cuenta las siguientes reglas:

1. Las librerías se buscarán en los directorios por defecto que son `lib, lib64` y `/usr/lib`. Si no ponemos nuestra librería ahí (no nos dejaran), tendremos que utilizar la opción del compilador -L para indicarlo.
2. Lo mismo tendremos que hacer con los ficheros incluidos de cabecera, en este caso la opción es -I. El directorio por defecto es `/usr/include.`

3. Todas las librerías que creemos empezaran con la palabra `lib` a la que seguirá el nombre propio de la librería (en el caso de las matemáticas `m` y en el anterior `re`).
4. Al compilar tendremos que invocar al enlazador con la opción -l para que incluya la librería creada (en el ejemplo `-lre`).

Existe un comando relacionado con las librerías, comando `nm`, para ver el contenido de sus módulos:

```
nm -[opciones] [ficheros]
```

donde el fichero puede ser un módulo o una librería, en este último caso se puede usar la opción -s para ver el índice.

Depuración

Una vez compilado el programa con éxito, tendremos que ejecutarlo. Lo normal es que existan errores de ejecución. Para depurarlos se pueden realizar impresiones de variables por nuestra cuenta (ojo, recuerda que el `printf` sin `\n` no imprimirá un mensaje inmediatamente por la salida estándar) o preferentemente utilizar un depurador si el programa es secuencial (no paralelo).

Siguiendo la filosofía de que tenéis que conocer las herramientas fundamentales de desarrollo incluidas en el `GNU toolchain` (ya habéis usado entornos integrados y los seguiréis usando en un futuro):

- GNU Compiler Collection (GCC): compiladores para varios lenguajes.
- GNU Binutils: enlazador, ensamblador y otras herramientas.
- GNU C Library (glibc): librería de C, incluyendo cabeceras, librerías y el cargador dinámico.
- GNU Debugger (GDB): un depurador interactivo.
- GNU make: automatización de la estructura y de la compilación.

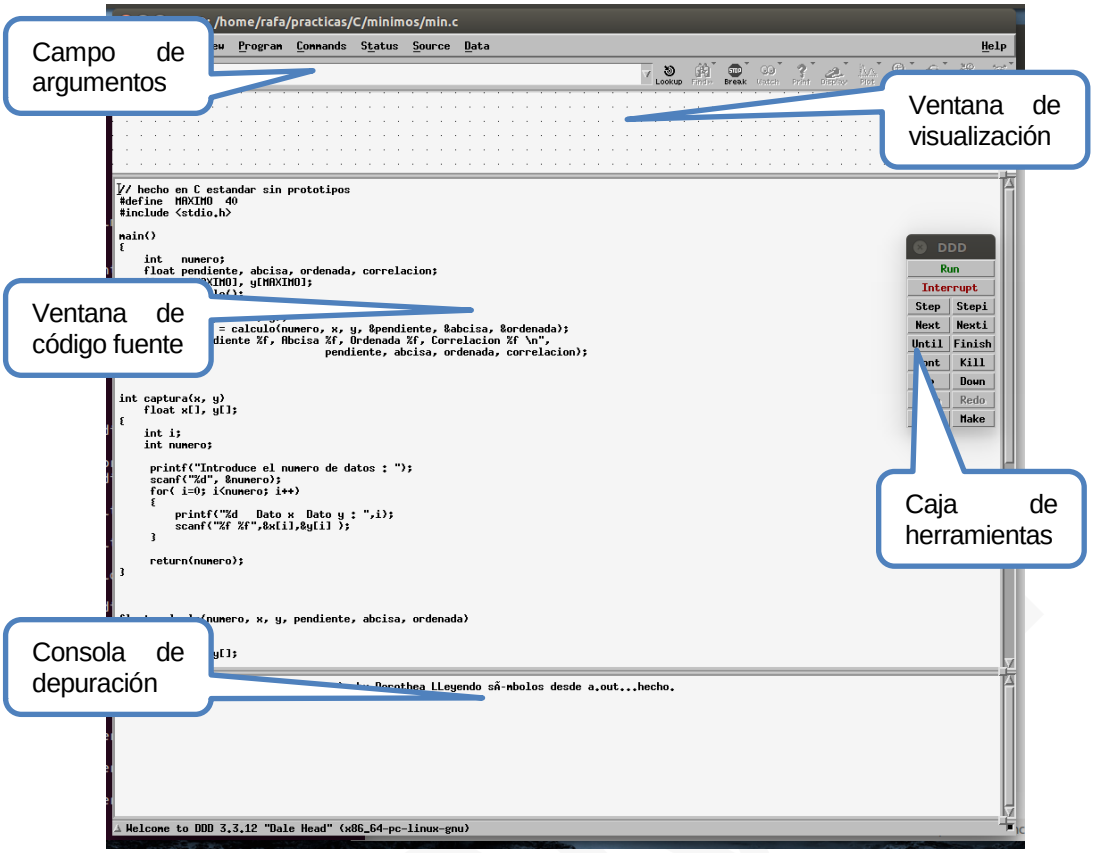
Nuestra herramienta de depuración será el `gdb`. Es un depurador en línea bastante incómodo de manejar, por lo que tiene como cliente gráfico el programa `ddd`. Este depurador nos permitirá:

- Tener varias ventanas de visualización (variables, depuración ejecución, …).
- Usar el botón izquierdo y derecho del ratón.
- Poner puntos de ruptura (break points).
- Utilizar el modo de ejecución paso a paso (step).
- Visualización de variables (print, display, …)
- Avisos de cambio (watch) de variables.
- …

Para ejecutarlo podemos usar varios modos: `ddd`, `ddd core`, `ddd ejecutable`. Pero el ejecutable a depurar siempre deberá haber sido compilado con la opción –g.

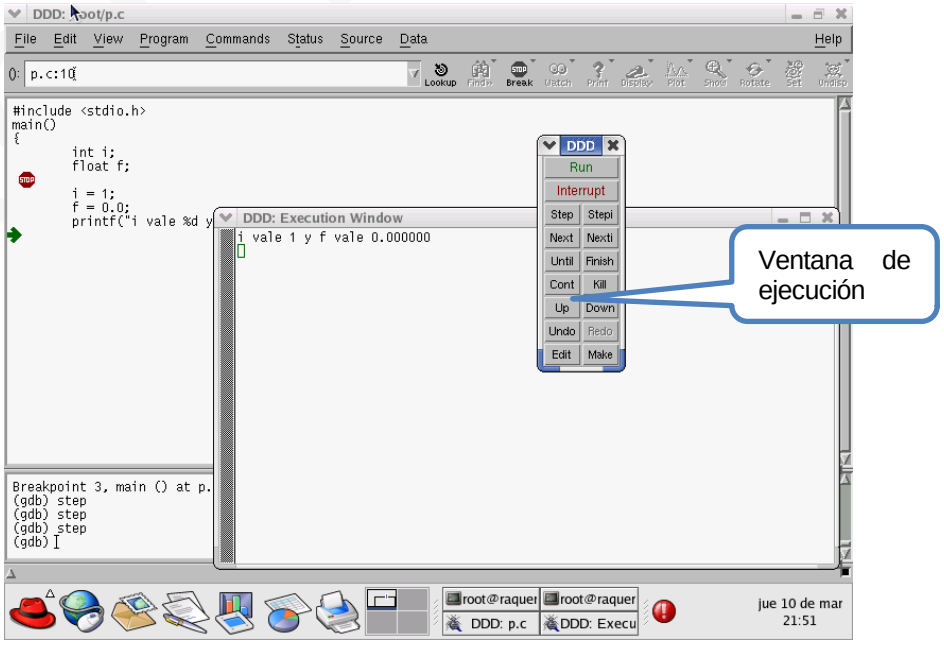
30. Ejecuta el `ddd` con el programa de mínimos cuadrados. Indica que ocurre. Cambia la opción de compilación para que incluya la depuración. Prueba ahora el `ddd`.

Una vez cargado el programa a depurar se podría tener algo como esto:



Arriba un campo para introducir los argumentos al programa, el menú disponible, una ventana de visualización de variables, otra del código, una caja de herramientas y abajo una ventana de introducción de órdenes (por ejemplo, para atender el `scanf`) y depuración.

También se puede arrancar una ventana de ejecución en vez de usar la de órdenes. Se puede forzar desde el menú *view* o con alt-9.



Una sesión de trabajo normal podrá incluir:

- Poner un *breakpoint* con el botón derecho del ratón en una zona de código fuente.
- Ejecutar el programa con la orden *Run* de la caja. Aparecerá una flecha verde por dónde se está ejecutando el programa y las primeras líneas en la consola de depuración. También podemos ir al menú o poner F2 para ejecutarlo con argumentos si es necesario.
- Salir del punto de ruptura dando a la orden *Next* de la caja, lo que también nos llevará a través de subrutinas.
- Ejecutar paso a paso con la orden *Step*.
- Si hemos pasado por la inicialización de variables, Señalar las con el botón derecho del ratón y ver sus valores arriba (*display*) de forma permanente o abajo (*print*). En la parte de visualización también se pueden mover con el ratón, redimensionar la ventana, ver arrays, etc.

31.	Con una versión del programa para depuración utiliza el ddd para: poner puntos de ruptura (botón derecho o menú), ejecutar, llegar al punto de ruptura, ver valores de variables (con print y display), tanto de tipo no estructurado (incluyendo punteros) como estructurado, ejecutar paso a paso e introducir valores en la ventana de entrada o la de ejecución.

Si utilizas la compilación separada (modular) debes tener presente que todas las funciones deben incorporar la opción de compilación con depuración, ya que si no el `ddd` no podrá mostrar su código fuente ni depurarlas, sólo ejecutarlas de un solo paso.

Empaquetado

Por último, y dado que lo vas a usar para mandar las prácticas, existen un par de parejas de comandos para empaquetar/desempaquetar y comprimir y descomprimir ficheros. Para lo primero se utiliza el comando `tar` con diferentes funciones. para lo segundo los comandos `gzip` y `gunzip`.

La arquitectura del primero es la función a realizar, las opciones y los ficheros afectados. Las funciones puedes cualquiera de las letras Acdrtux : concAtenar, crear, diferenciar, añadir, listar, update, extraer (tienen sus variantes largas y más claras). Hay muchas opciones (puedes ver el manual). En el 90% de los casos se utilizan estas versiones de `tar`:

`tar -cvf archivo.tar pepe juan`	crea (función) en el archivo (opción f) los ficheros pepe y juan
`tar --create -f z.tar nano*`	crea z.tar con los archivos que empiezan en nano
`tar -xvf archivo.tar`	extrae los ficheros que hay en archivo.tar
`tar -tvf archive.tar`	lista los ficheros que hay en archivo.tar

El segundo es más sencillo basta con poner el nombre del fichero a comprimir/descomprimir. Por defecto creará un fichero que empieza en z.nombre. Dos ejemplos ilustrativos:

`gzip archivo.tar`	convierte a z.archivo.tar comprimido
`gunzip z.archivo.tar`	convierte de nuevo a archivo.tar descomprimido