Introducción al entorno de trabajo

Área de Arquitectura y Tecnología de Computadores – Versión 1.2, 11/09/2017

Índice

Objetivos de la sesión

En esta sesión se introduce al alumno el entorno de trabajo que se utilizará a lo largo de gran parte de las prácticas de laboratorio. Este entorno se basa en el sistema operativo GNU/Linux, el lenguaje de programación C y el control de versiones distribuido con *git*. Por tanto, los principales objetivos de esta sesión son:

- Preparar el entorno e introducir el flujo de trabajo.

- Introducir el sistema operativo GNU/Linux y sus comandos básicos.
- Aprender a conectarse a máquinas GNU/Linux remotas y transferir ficheros.
- Introducir los conceptos básicos del lenguaje C.
- Conocer el funcionamiento de los punteros en C.

Para conseguir estos objetivos se arrancará un sistema operativo Linux en el que se realizarán pequeños programas escritos en lenguaje C.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Disponer de una máquina, física o virtual, con un sistema operativo Ubuntu Server 16.04 LTS sobre el que se trabajará a lo largo del curso. En la parte inicial de la práctica se explica cómo crear la máquina virtual.
- Durante la sesión se plantearán una serie de preguntas que puedes responder en el correspondiente cuestionario (https://www.campusvirtual.uniovi.es/mod/quiz/view.php?id=252790) en el campus virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

1. Preparando el entorno

Inicialmente debemos realizar unos preparativos para poder realizar las prácticas de la asignatura. Estos comprenden la instalación de la máquina GNU/Linux que vamos a utilizar y la definición del flujo de trabajo a seguir.

1.1. Creación de la máquina virtual

Para el desarrollo de las prácticas utilizaremos un sistema GNU/Linux. Puede servir tanto una máquina física como virtual, por lo que utilizaremos esta última opción por simplicidad.

Los profesores de la asignatura hemos preparado una máquina virtual con todas las herramientas necesarias que utilizaremos a lo largo de las prácticas. Concretamente, contiene dos *drivers* desarrollados específicamente.

- Descarga el fichero con la máquina virtual (http://www.atc.uniovi.es/grado/2ac/ubuntu-16.04-server-32bits-2ac-2.0.ova) a tu ordenador.
- Abre VirtualBox y elije la opción Archivo ▸ Importar servicio virtualizado y elige el fichero que acabas de descargar.

> Es posible que tengas que modificar antes la configuración de VirtualBox para crear la máquina en un directorio en el que tengas permiso de escritura. Comprueba dónde se guardan las máquinas usando la opción Archivo ▸ Preferencias.

- Establece el nombre de la máquina virtual a `2AC` y deja el resto de opciones por defecto. Activa la opción ***Reinicializar la dirección MAC de todas las tarjetas de red***.
- Pulsa [**Importar**].

1.2. Arrancando el sistema

GNU/Linux es un sistema operativo multitarea y multiusuario, lo que indica que puede ejecutar varias tareas de distintos usuarios al mismo tiempo. Este operativo pertenece a la familia de operativos UNIX, por lo que presenta grandes similitudes con otros sistemas operativos como BSD o MAC OS X.

- Arranca la máquina con el sistema GNU/Linux instalado.

GNU/Linux ofrece dos modos de trabajo al usuario: modo gráfico y modo texto. En el modo gráfico se puede trabajar sobre el equipo a través de un escritorio similar al que puede encontrarse en otros sistemas operativos de ventanas. El modo texto ofrece la misma funcionalidad que el modo gráfico, con la desventaja de que todas las operaciones sobre el equipo deben realizarse a través de una interfaz de comandos. Este último modo es el utilizado habitualmente cuando nos conectamos de forma remota al equipo o bien cuando el sistema no tiene configurado el modo gráfico, típicamente en servidores, para ahorrar recursos en la máquina.

Dada la gran flexibilidad que ofrece la interfaz de comandos del operativo, y que no siempre estará disponible el modo gráfico cuando trabajemos con máquinas UNIX, este será nuestro entorno de trabajo. A continuación se introducirán los comandos básicos de la interfaz de comandos.

2. Interfaz de comandos

Cuando accedemos al sistema utilizando el modo texto de UNIX directamente aparece la interfaz de comandos del sistema operativo.

Puedes acceder a través de la consola de la máquina si estás ante ella (ya sea física o virtual), o bien a través de la red utilizando SSH. Necesitarás conocer la IP de la máquina para conectarte usando, por ejemplo, el programa PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html). En estas prácticas se asumirá este último caso, pues siempre podrá seguirse. En la consola de PuTTY puedes copiar texto seleccionando con el botón izquierdo del ratón y pegar pulsando el botón derecho.

UNIX permite utilizar diferentes interfaces de comandos, aunque la más común es la interfaz `bash`. Esto tiene sus implicaciones a la hora de desarrollar tareas *batch*, es decir, cuando programamos para la propia interfaz de comandos a través de *scripts*.

Un *script* es un fichero de texto que contiene comandos.

Dentro de la interfaz de comandos puedes utilizar la tecla `Tab` para autocompletar comandos (y nombres de ficheros) y las flechas de arriba y abajo para acceder al histórico de comandos introducidos en la consola.

- Entra en sesión usando PuTTY con el nombre de usuario `student` y la contraseña `student`. Cambia la contraseña siguiendo las instrucciones. Apunta la contraseña utilizada (idealmente, en tu gestor de contraseñas). La máquina virtual se configura en modo NAT e incluye una regla que redirige el tráfico SSH, por lo que debes utilizar la dirección `localhost` y el puerto `62000` para conectarte a la máquina virtual con PuTTY.

Ten cuidado cuando entres en sesión por primera vez para cambiar la contraseña. Inicialmente, se te pedirá la contraseña actual (`student`) para después introducir la nueva contraseña dos veces. Además, como medida de seguridad, la consola de Linux no muestra el eco local cuando introduces contraseñas, esto es, no verás los caracteres que escribes (no te aparecerán asteriscos). Después de cambiar la contraseña se cerrará la conexión. Podrás conectarte de nuevo, pero ya utilizando la nueva contraseña.

2.1. Directorio de trabajo

Una vez abierta la interfaz de comandos se muestra el *prompt*, que es el texto que aparece al principio de la línea y que, por defecto, suele indicar el directorio de trabajo actual. Todos los usuarios tienen asignado un directorio de trabajo por defecto denominado `home` , que aparece representado en el *prompt* a través del carácter ~. El comando `pwd` [1] sirve para obtener el directorio actual de trabajo:

```
$> pwd
/home/student
```

El concepto de directorio de trabajo es de vital importancia, pues sobre él se realizarán por defecto todas las operaciones de acceso a ficheros, por ejemplo, crear un fichero de texto, eliminar un fichero, etc., cuando no se especifiquen rutas absolutas en los nombres de ficheros.

Al igual que otros sistemas de ficheros, en UNIX los ficheros se agrupan en directorios organizados de forma jerárquica[2]. A diferencia de los sistemas Windows, en UNIX no existen letras de unidad como `C:` o `D:` . En su lugar, todos los directorios cuelgan del directorio raíz, referenciado por `/` . Para cambiar el directorio de trabajo se utiliza el comando `cd` .

- Cambia el directorio de trabajo al directorio raíz.

```
$> cd /
```

- Baja un nivel en la jerarquía de directorios al directorio `/etc` .

```
$> cd etc
```

- Vete al directorio `/usr/include` especificando una ruta absoluta.

```
$> cd /usr/include
```

- Retrocede al directorio anterior.

```
$> cd -
```

- Sube un nivel en la jerarquía de directorios. Ten cuidado, hay un espacio entre el nombre del comando y los dos puntos.

```
$> cd ..
$> pwd
/
```

- Vete al directorio `home`, es decir, al directorio de trabajo por defecto cuando arrancas la interfaz de comandos. Esto puede hacerse de varias formas.

```
$> cd
$> cd ~
$> cd $HOME
```

Hay dos formas de especificar una ruta a través de la jerarquía de directorios: de forma absoluta o de forma relativa. Se puede especificar una ruta absoluta indicando el conjunto de directorios a través del que hay que bajar desde el raíz hasta llegar al fichero deseado. Por ejemplo, el siguiente comando utiliza la orden `cat`, que muestra el contenido de un fichero, especificando su ruta absoluta y, por lo tanto, funcionará independientemente del directorio donde esté situado el usuario.

```
$> cat /etc/hostname
2ac
```

También pueden utilizarse los directorios especiales `.` y `..` que referencian al directorio actual y al padre respectivamente para especificar una ruta relativa, es decir, a partir del directorio de trabajo actual. Observa que una ruta relativa nunca comienza por el directorio raíz `/` al contrario de lo que ocurría con una ruta absoluta.

```
$> cd
$> pwd
/home/student
$> cat ../../etc/hostname
2ac
$> cd /etc
$> cat hostname
2ac
$> cd ..
$> cat ./etc/hostname
2ac
$> cat etc/hostname
2ac
```

El contenido de un directorio puede listarse con el comando `ls`.

```
$> ls
bin   cdrom  etc   initrd.img  media  opt   rofs  sbin     srv  tmp  var
boot  dev    home  lib         mnt    proc  root  selinux  sys  usr  vmlinuz
```

La gran mayoría de sistemas UNIX organizan sus sistemas de ficheros de forma parecida, de tal forma que presentan directorios colgando del raíz con misiones específicas:

- `/home`. Es el directorio donde se ubican los directorios de los usuarios. Típicamente existe un directorio por usuario.
- `/bin`. Contiene comandos que pueden invocarse, tales como el comando `pwd`.
- `/dev`. En él se ubican los ficheros especiales que permiten acceder a los dispositivos del computador y periféricos.
- `/etc`. Contiene la mayor parte de los ficheros para la configuración del sistema y de los programas instalados.
- `/proc`. Contiene ficheros con información del sistema.

2.2. Editando ficheros

Para editar ficheros de texto a través de la consola podemos utilizar editores como `nano`.

Vamos a editar un fichero de texto utilizando el editor en modo consola. Ten cuidado con el directorio de trabajo, pues es donde se crea el fichero resultante.

- Abre el editor indicándole que vamos a editar un nuevo fichero de nombre `1-1texto.txt`. La opción `-c` hace que el editor muestre los números de línea en la parte inferior.

```
$> cd
$> nano -c 1-1texto.txt
```

- Escribe algo en el fichero, como se muestra en la figura siguiente.

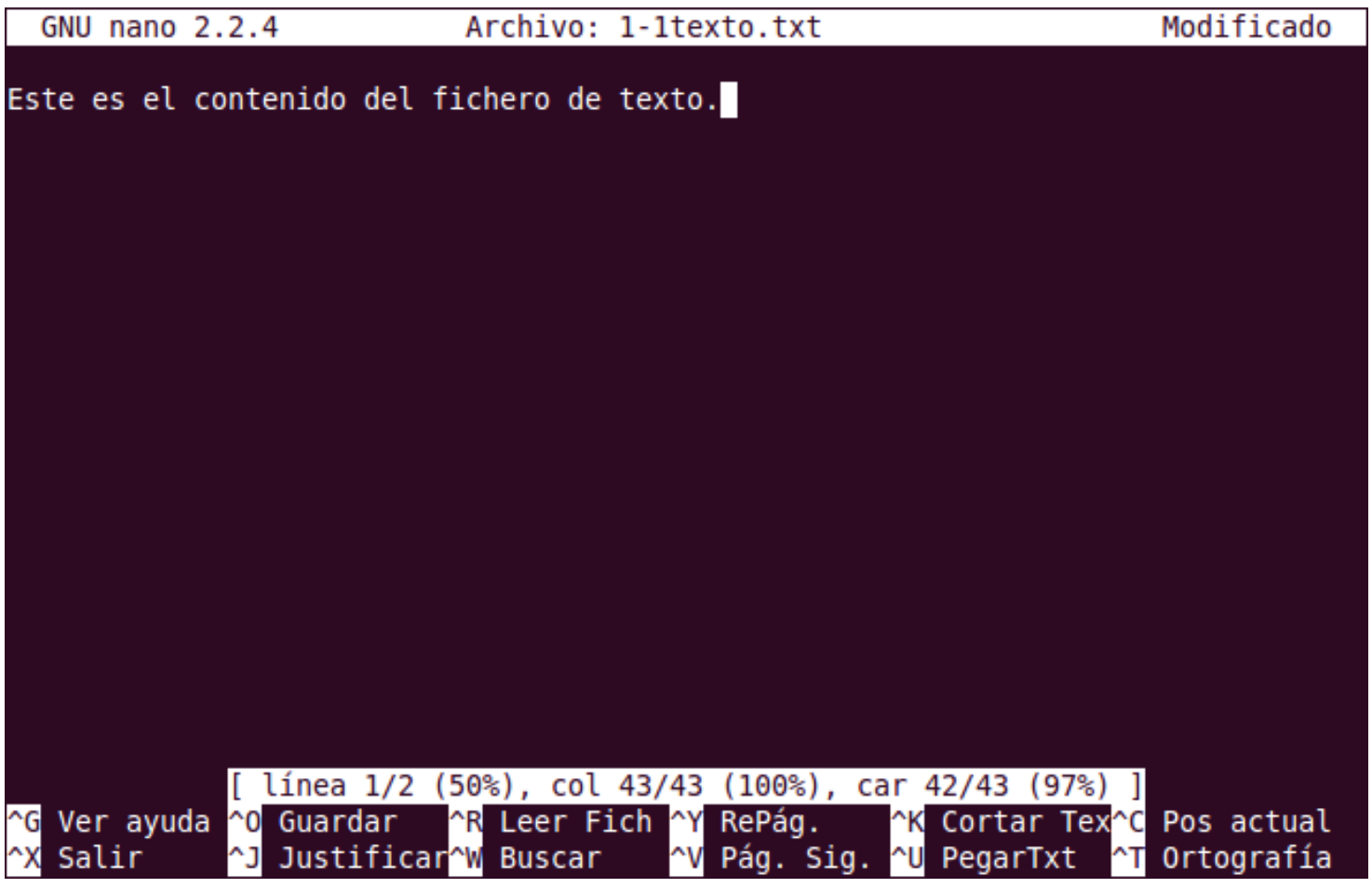

Figura 1. Editor de texto nano

- Guarda el fichero. Para ello utiliza la combinación `Ctrl` + `o` y pulsa `ENTER` para confirmar el nombre del fichero.
- Sal del editor utilizando la combinación `Ctrl` + `x`. Como se ve en la figura, este editor muestra también en la parte inferior una ayuda con el significado de diferentes combinaciones de teclas, donde `^` indica la tecla `Ctrl`.

Como cualquier otro comando, se le puede indicar al editor un nombre de fichero incluyendo una ruta relativa o absoluta. En el ejemplo anterior se especificó una ruta relativa, es decir, se le indicó al editor que creara el fichero en el directorio de trabajo

actual, pero también puede especificarse una ruta absoluta con el fichero.

- Abre el fichero recién creado especificando su ruta absoluta.

```
$> nano -c /home/student/1-1texto.txt
```

- Sal del editor.

2.3. Permisos

Como ocurre en otros sistemas operativos multiusuario, UNIX etiqueta cada entrada en el sistema de ficheros (fichero, directorio o enlace simbólico) con una serie de permisos. En concreto, existen tres tipos de permisos, representados por distintos caracteres cuando se hace un listado de un directorio: permiso de lectura (`r`), de escritura (`w`) y de ejecución (`x`). Además, estos permisos se asignan en tres niveles distintos: al usuario propietario del fichero (directorio o enlace simbólico), al grupo propietario y al resto de usuarios del sistema.

Veámoslo con un ejemplo que utiliza la opción `-l` del comando `ls` para mostrar, además del nombre de los ficheros, sus atributos (permisos, propietario, etc.):

```
$> ls -l
total 12
drwxr-xr-x 2 student student 40 2011-06-10 10:42 directorio
-rw-rw-r-- 1 student student 43 2011-06-10 10:07 1-1texto.txt
-rw-r--r-- 1 student root   16 2011-06-10 10:40 otro.txt
-rwxr-xr-x 1 student student 18 2011-06-10 10:39 script
```

La información que se muestra en el listado está organizada en columnas con el siguiente significado:

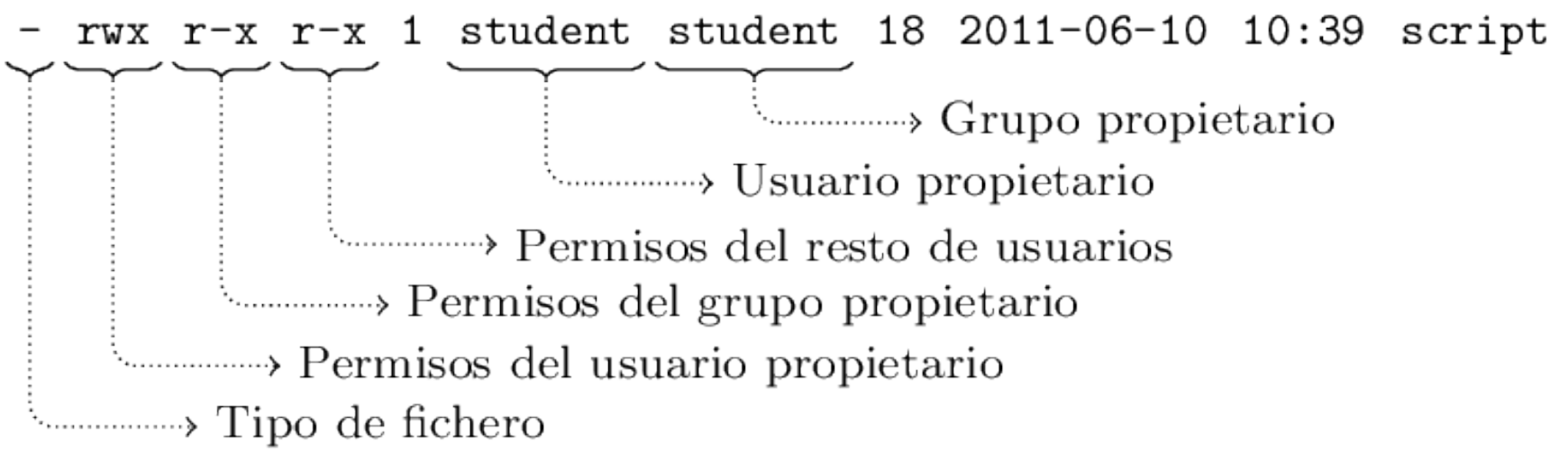

Figura 2. Significado de los permisos en UNIX

La primera entrada del listado se corresponde con un directorio, de ahí el carácter `d`, el resto de entradas son ficheros regulares, de ahí que muestren el carácter `-` en el tipo de fichero. Los enlaces simbólicos muestran el carácter `l`.

El usuario `student` es el propietario de las cuatro entradas que aparecen en el listado anterior. Además, este usuario tendrá permisos de lectura y escritura sobre todos los ficheros, mientras que tendrá además permisos de ejecución sobre el directorio y el fichero `script`.

Los permisos de un fichero pueden cambiarse utilizando el comando `chmod`. Por ejemplo, se puede eliminar el permiso de escritura sobre el fichero `1-1texto.txt` de la siguiente forma:

```
$> chmod -w 1-1texto.txt
$> ls -l 1-1texto.txt
-r--r--r-- 1 student student 43 2011-06-10 10:07 1-1texto.txt
```

También pueden modificarse todos los permisos considerando cada tripleta como un número binario. De esta forma, se especifica un número formado por 3 bits, cada uno de los cuales equivale a la representación en binario de los permisos `rwx`, donde habría un `1` para cada permiso activo. Así, si queremos asignar los siguientes permisos:

- Usuario propietario: todos los permisos ⇒ `rwx` ⇒ `111b` ⇒ `7d`.
- Grupo propietario: lectura y escritura ⇒ `rw-` ⇒ `110b` ⇒ `6d`.
- Resto de usuarios: solo lectura ⇒ `r--` ⇒ `100b` ⇒ `4d`.

```
$> chmod 764 1-1texto.txt
$> ls -l 1-1texto.txt
-rwxrw-r-- 1 student student 43 2011-06-10 10:07 1-1texto.txt
```

En ocasiones, igual que ocurre en otros sistemas operativos, es necesario tener los privilegios de administrador para realizar una tarea o ejecutar un comando. El usuario administrador en sistemas UNIX es habitualmente el usuario `root` (o superusuario). Sin embargo, cuando entramos en sesión rara vez lo hacemos como usuario administrador.

- Comprueba con qué usuario estás dentro del sistema. Utiliza el comando `whoami`.

```
$> whoami
student
```

Los sistemas UNIX ofrecen la posibilidad de ejecutar cualquier comando o programa con los privilegios del usuario `root` utilizando el comando `sudo`. Por ejemplo, para reiniciar la red se puede utilizar el *script* `/etc/init.d/networking`. Aunque este *script* puede ser ejecutado por cualquier usuario, los comandos que contiene requieren permisos de `root` para ejecutarse. Vamos a parar la red y volver a arrancarla.

- Prueba a reiniciar la red invocando al *script* `/etc/init.d/networking`. Verás que ocurre un error, pues no tienes los permisos adecuados.

```
$> /etc/init.d/networking restart
```

> En algunas versiones modernas de Linux es posible que el sistema detecte que se intenta ejecutar un comando sin los necesarios privilegios y se realice un `sudo` de forma transparente. En otros, es posible que, tras ejecuctar el comando, la consola de PuTTY deje de mostrar el eco local, es decir, los caracteres que se pulsan. En ese caso, cierra la conexión y conéctate de nuevo.

- Invoca ahora el *script* con permisos de superusuario.

```
$> sudo /etc/init.d/networking restart
```

Recuerda el uso del comando `sudo`, ya que se utilizará a menudo a lo largo del curso.

2.4. Operaciones básicas con ficheros

Hasta ahora hemos visto cómo crear y editar ficheros de texto, listar el contenido de los directorios del sistema o cambiar el directorio actual de trabajo. Ahora veremos operaciones básicas como copiar, mover, renombrar o eliminar ficheros.

- Crea un directorio de nombre `dir` que cuelgue de tu directorio `home`. Para ello debes utilizar el comando `mkdir`, que como el resto de comandos puede recibir una ruta relativa o absoluta.

```
$> mkdir ~/dir
```

- Haz una copia del fichero `1-1texto.txt` dentro del directorio recién creado que tenga el nombre `1-1copia1.txt`. Para ello debes utilizar el comando `cp`. Observa el ejemplo de invocación del comando e intenta entender qué significan los parámetros que se le pasan. Si tienes alguna duda pregúntale al profesor.

```
$> cd ~/dir
$> cp ../1-1texto.txt ./1-1copia1.txt
```

- Renombra el fichero `1-1copia1.txt` a `1-1copia2.txt` al tiempo que lo mueves a tu directorio `home` utilizando el comando `mv`. Este comando sirve tanto para mover como para renombrar ficheros.

```
$> mv 1-1copia1.txt ../1-1copia2.txt
```

- Elimina el fichero `1-1copia2.txt` utilizando el comando `rm`. Mucho cuidado con este comando, pues no solicita confirmación cuando se borran los ficheros.

```
$> cd ..
$> rm 1-1copia2.txt
```

- Elimina el directorio `dir` que creamos al principio con el comando `rmdir`. Este comando requiere que el directorio esté vacío.

```
$> rmdir dir
```

2.5. Ayuda en línea

GNU/Linux es un sistema operativo eminentemente orientado a usuarios desarrolladores de aplicaciones en C. De hecho, las distribuciones suelen incluir el compilador y el entorno de desarrollo, entre otras ayudas a la programación. Una de ellas es la ayuda en línea, que cubre aspectos de programación en C, así como comandos básicos de la interfaz de comandos.

La ayuda en línea de GNU/Linux y, en general, de los sistemas operativos UNIX se ofrece a través de manuales organizados en categorías. Cada una de estas categorías se numera desde el 1 hasta el 8 y comprenden comandos generales de línea de comandos (categoría 1) o funciones de biblioteca invocables desde un programa escrito en el lenguaje C (categoría 3).

Puedes acceder a la ayuda de cualquier comando utilizando la orden `man` seguida del nombre del comando.

- Consulta el manual acerca del comando `ls`. Puedes salir del manual pulsando `Q`.

```
$> man ls
```

- También puedes especificar directamente la categoría donde consultar el manual. Consulta ahora el manual de la función `printf` de C. Esta función sirve para mostrar información en pantalla tal como se verá más adelante.

```
$> man 3 printf
```

En el manual se muestra información útil sobre cómo invocar la función y qué ficheros de cabecera hay que incluir para poder invocarla.

- Sal del manual.

La indicación explícita de la categoría a consultar resulta útil en el caso anterior, pues también existe un comando de la línea de comandos de nombre `printf`. Si no se hubiese especificado la categoría, se mostraría la ayuda del comando de la línea de comandos en lugar de la función de C. Puedes probar a invocar el comando anterior sin indicar la categoría y verás el resultado.

La ayuda se puede consultar también a través de buscadores de internet, que habitualmente indexan los manuales. Generalmente, el resultado de consultar la ayuda desde la línea de comandos o desde un buscador lleva a un resultado equivalente.

2.6. Deteniendo el sistema

Como con cualquier otro sistema operativo, al acabar nuestro trabajo debemos detener el sistema.

- Apaga la máquina virtual usando el siguiente comando.

```
$> poweroff
```

> Siempre debes utilizar esta comando para apagar la máquina virtual. Nunca lo hagas a través de la opción *Apagar máquina virtual* de VirtualBox, pues equivaldría a desconectar la máquina de la alimentación eléctrica. Sí podrías hacerlo con la opción *Enviar señal de apagado*.

- Vuelve a arrancar la máquina virtual.

> Puede utilizarse el comando `reboot` para reiniciar la máquina, esto es, apagarla y volver a arrancarla.

3. Control de versiones

Cuando se trabaja con código fuente (y en general con cualquier tipo de documento que sufre una cierta evolución) se antoja muy recomendable el uso de los sistemas de control de versiones. En los casos de proyectos software de envergadura, resulta entonces imprescindible.

Estos sistemas permiten llevar registro de todos los cambios que se producen en el código (quién, qué y cuándo) de forma automática. A lo largo de esta asignatura utilizaremos *git* como sistema de control de versiones, si bien un flujo de trabajo mínimo. Concretamente, utilizaremos un repositorio[3] *git* en la nube como sistema de *backup* de los ficheros que vayamos generando.

- Configura *git* con tu nombre de usuario y tu correo electrónico. Estos serán los datos que aparecerán en el historial de cambios.

```
$> git config --global user.name "<usuario>"
$> git config --global user.email "<correo>"
```

No pongas los símbolos "<" y ">": se utilizan para indicar que no debes escribir esa palabra literalmente.

- Regístrate en Bitbucket (https://bitbucket.org/), que es un servicio de repositorios *git* gratuito. Puedes utilizar tu *UO* como nombre de usuario.
- Crea un nuevo repositorio eligiendo la opción Repositorios ▸ Crear repositorio. Dale como nombre `2ac-uo<xxxxxx>`, dejando el resto de opciones por defecto.
- Dale a tu profesor de prácticas acceso al repositorio de lectura y escritura en la sección de Settings ▸ Access management. El profesor te indicará su nombre de usuario.
- Pincha en la opción Acciones ▸ Clonar. Copia el texto que aparece en la ventana emergente (es un comando).
- Desde la interfaz de comandos de Linux colócate en tu directorio `home` y ejecuta el comando que acabas de copiar para clonar el repositorio de Bitbucket en tu máquina. Debería ser parecido al siguiente, donde `<usuario>` representa tu nombre de usuario en Bitbucket.

```
$> git clone https://<usuario>@bitbucket.org/<usuario>/2ac-
uo<xxxxxx>.git
```

- Cambia al directorio `2ac-uo<xxxxxx>` donde se ha copiado el repositorio.

El flujo de trabajo que seguiremos siempre será:

1. Al comienzo de cada sesión de prácticas traer del repositorio de Bitbucket los cambios que se hubiesen realizado con anterioridad.

No ejecutes estas órdenes todavía. Esto es sólo una explicación.

```
$> git pull
```

2. Crear o modificar los ficheros que desarrollemos.
3. Añadir los ficheros o los cambios que hayamos realizado que queremos que estén bajo el control de versiones (de los que queremos *backup*) con la siguiente orden, donde `<fichero>` es el nombre del fichero añadido o modificado.

   ```
   $> git add <fichero>
   ```

4. Confirmar los cambios realizados, de forma que se registren en el sistema de control de versiones. Se solicitará un mensaje en el que indicarás una breve descripción de los cambios realizados. Normalmente realizaremos varios *commits* en cada sesión de prácticas para crear sucesivas versiones en el repositorio.

   ```
   $> git commit
   ```

Se abrirá un editor para que puedas introducir la descripción de los cambios realizados. Guarda el fichero para confirmar el *commit*.

5. Cuando acabemos de trabajar, enviar los cambios hacia el repositorio de Bitbucket.

   ```
   $> git push
   ```

Resultaría conveniente crear un subdirectorio para cada sesión de prácticas dentro del directorio con el respositorio *git*.

4. Programación en C

El lenguaje de programación que se utilizará a lo largo de la asignatura será el lenguaje C. El lenguaje C es el lenguaje de programación de sistemas por excelencia y se encuentra íntimamente relacionado con los sistemas operativos UNIX. En esta sección repasaremos algunos conceptos sobre C y veremos algunos nuevos como los punteros.

- Crea un subdirectorio de nombre `sesion1-1` en el subdirectorio que contiene el repositorio de *git* `2ac`.

```
$> mkdir sesion1-1
```

- Entra en el nuevo directorio.

Deberás crear todos los programas que desarrolles a partir de ahora en el subdirectorio que acabas de crear.

4.1. Primer programa

Vamos a editar nuestro primer programa en C y ejecutarlo para ilustrar cuál es el procedimiento que debe seguirse en el desarrollo de programas.

- Edita el fichero `1-1hello.c` para que tenga el contenido que se muestra a continuación:

```c
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

En esta asignatura se adopta el convenio de programar en inglés. Cualquier graduado en informática debe ser capaz de leer y escribir código escrito en inglés, lo que le permitirá integrarse en proyectos internacionales.

Para generar el ejecutable es necesario realizar dos pasos: compilar el fuente y enlazar el ejecutable.

- Compilación: consiste en convertir los ficheros fuentes en lenguaje C a su equivalente código objeto (código máquina). La compilación se realiza invocando al compilador pasándole la opción `-c` junto con los ficheros fuente a compilar. En este caso:

```
$> gcc -c 1-1hello.c
$> ls *.o
1-1hello.o
```

- Enlazado: consiste en la generación del ejecutable a partir de los códigos objeto y las bibliotecas necesarias. Para realizar el enlazado se invoca al mismo compilador, que este caso actúa de enlazador, pasándole los ficheros con el código objeto. Opcionalmente, se le puede indicar el nombre del ejecutable deseado utilizando la opción `-o`. En otro caso, el compilador siempre generará un ejecutable de nombre `a.out`. Habría que indicarle también al compilador las bibliotecas con las que enlazar, pero en este caso no es necesario, pues no se necesitan bibliotecas adicionales. Por defecto el compilador enlaza el ejecutable con la biblioteca estándar de C, que contiene las funciones definidas por el estándar tales como `printf`.

```
$> gcc 1-1hello.o -o 1-1hello
$> ls -l 1-1hello
-rwxr-xr-x 1 student student 7155 2011-06-10 16:26 1-1hello
```

Como ves, se genera un fichero ejecutable de nombre `1-1hello`. Puedes invocar el ejecutable indicando la ruta en la que se encuentra, ya que por defecto el directorio de trabajo no está dentro de los directorios que consulta la interfaz de comandos para la ejecución de comandos y programas.

- Invoca el programa especificando únicamente su nombre. Verás que te aparece un mensaje indicativo de que no se encontró el comando.

```
$> 1-1hello
1-1hello: orden no encontrada
```

- Invoca el programa especificando ahora la ruta donde se encuentra, en este caso en el directorio actual. De esta forma, se le indica a la interfaz de comandos dónde debe buscar el comando.

```
$> ./1-1hello
Hello, World!
```

- Borra los ficheros `1-1hello.o` y `1-1hello`. Usa el comando `rm`.

Recuerda especificar la ruta cuando ejecutes programas creados durante las prácticas.

Por último, vamos a incorporar el fichero recién creado al repositorio *git*.

- Añade el fichero que acabas de crear al control de versiones.

```
$> git add 1-1hello.c
```

- Confirma los cambios.

```
$> git commit
```

4.2. Invocando funciones de biblioteca

Cuando un programador desarrolla programas en C, y en general en cualquier lenguaje de programación, utiliza funciones que se repiten entre programa y programa. Tal es el caso de las funciones que muestran información en pantalla, solicitan memoria de forma dinámica, trabajan con cadenas, etc. No tiene sentido que el programador tenga que escribir estas funciones una y otra vez, por lo que el lenguaje define un mecanismo para reutilizar estas funciones.

Estas funciones comunes pueden implementarse en ficheros separados que serán enlazados con el programa a desarrollar en la fase de enlazado. Estos ficheros pueden ser ficheros objetos como hemos visto o bibliotecas de funciones. No obstante, el compilador necesita conocer estas funciones que el programa utiliza pero que no define, para lo cual se utilizan ficheros de cabecera. Estos ficheros contienen declaraciones de funciones, es decir sus prototipos, que únicamente le indican al compilador cómo se llaman las funciones, cuántos parámetros reciben y de qué tipos.

En la fase de enlazado hay que especificarle al enlazador cuáles son los ficheros objetos o bibliotecas donde se encuentran estas funciones. La figura siguiente ilustra todo este proceso.

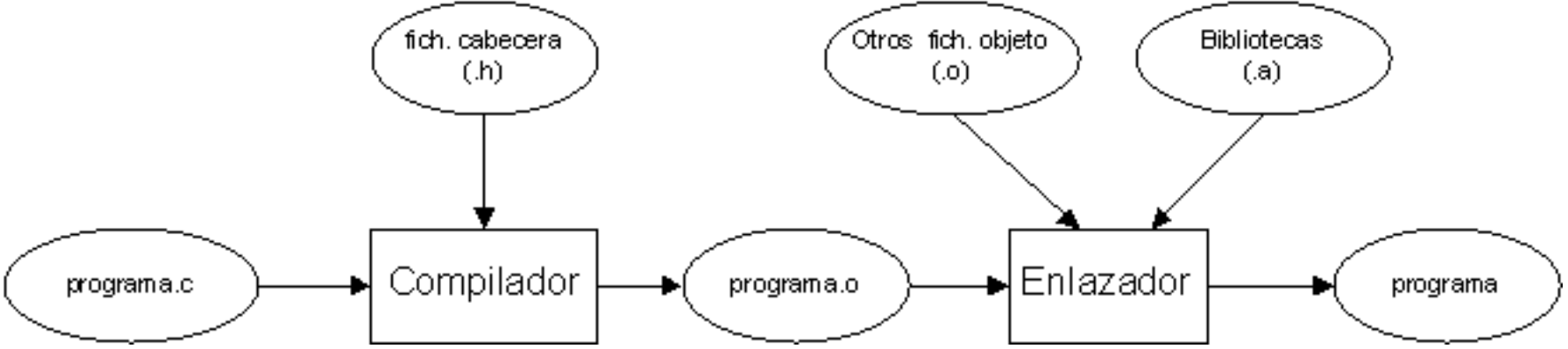

Figura 3. Ciclo de desarrollo de programas en C

Vamos a verlo con un ejemplo. Crearemos una función `circleArea` que devuelva el área de un círculo a partir de su radio. Esta función sería reutilizable por más programas, por lo que se definirá en un fichero distinto a donde definimos el programa principal.

- Edita el fichero `1-1circle.c` para que tenga el siguiente contenido:

```c
#include "1-1circle.h"

double circleArea(double radius)
{
    const double PI = 3.1415;
    return PI * radius * radius;
}
```

- Edita el fichero de cabecera `1-1circle.h` donde se definirá el prototipo de la función `circleArea`.

```c
// Use always this statement in header files
#pragma once

// Function prototype
double circleArea(double radius);
```

La línea `#pragma once` hace que el compilador añada a la compilación el código del fichero de cabecera una sola vez. Debe hacerse siempre para evitar errores de compilación por duplicidad de identificadores al hacer referencia al fichero de cabecera desde dos o más ficheros fuente.

- Edita el fichero `1-1program.c` con el programa principal que hace uso de la función círculo:

```c
#include <stdio.h>

// Include the header file
#include "1-1circle.h"

int main()
{
    double radius = 3.0;
    double area = circleArea(radius);
    printf("Circle area (radius=%f) is %f\n", radius, area);
    return 0;
}
```

- Compila ambos ficheros para obtener el código objeto.

```
$> gcc -c 1-1circle.c 1-1program.c
```

- Enlaza el programa indicando el fichero objeto donde está la función `circleArea`.

```
$> gcc 1-1circle.o 1-1program.o -o 1-1program
```

- Ejecuta el programa resultante e incorpora los tres ficheros fuente a tu repositorio (`git add` y `git commit`).

El estándar que describe el lenguaje C define adicionalmente una biblioteca de funciones que deben incluir todos los compiladores denominada la biblioteca estándar de C. Esta biblioteca incluye el código de las funciones definidas en el estándar de C.

Un ejemplo de función incluida en la biblioteca estándar de C es la función `printf` antes citada. Puedes consultar el manual de la función para ver cómo se invoca y qué fichero de cabecera es necesario incluir.

- Abre el manual de la función `printf`.

```
$> man 3 printf
```

- ¿Qué fichero de cabecera es necesario incluir? Responde en el cuestionario (https://www.campusvirtual.uniovi.es/mod/quiz/view.php?id=252790): pregunta 1.

El enlazador por defecto enlaza todos los programas con la biblioteca estándar de C, por lo que solo debes preocuparte de incluir el fichero de cabecera correspondiente.

Sin embargo, hay algunas funciones que se definen en otras bibliotecas. Tal es el caso de la función `sqrt`, que calcula la raíz cuadrada de un número real.

- Consulta el manual de la función `sqrt`.

```
$> man 3 sqrt
```

- ¿Qué fichero de cabecera es necesario incluir? Responde en el cuestionario (https://www.campusvirtual.uniovi.es/mod/quiz/view.php?id=252790): pregunta 2.
- ¿Es necesario indicarle alguna opción al enlazador? Responde en el cuestionario (https://www.campusvirtual.uniovi.es/mod/quiz/view.php?id=252790): pregunta 3.
- Edita el fichero `1-1root.c`.

```c
#include <stdio.h>
// Add the required #include directive

int main()
{
    double num = 9.0;
    double root = sqrt(num);
    printf("The square root of %f is %f\n", num, root);
    return 0;
}
```

- Compila el programa.
- Enlaza el programa, verás que te aparece un error pues la función `sqrt` no está definida. Recuerda que solo has incluido el fichero de cabecera, que contiene únicamente el prototipo de la función, no su código.

```
$> gcc 1-1root.o -o 1-1root
1-1root.o: In function 'main':
1-1root.c:(.text+0x2d): undefined reference to 'sqrt'
collect2: ld returned 1 exit status
```

- Enlaza ahora el programa incluyendo en la línea de comandos la opción que te indicaba la página del manual de `sqrt`.
- Ejecuta el programa e incorpora el fuente al repositorio *git*.

Es de vital importancia que sepas diferenciar los errores de compilación de los errores de enlazado.

4.3. Automatizando la generación del ejecutable

Los sistemas UNIX suelen incluir una herramienta, denominada `make`, que permite automatizar la generación de ejecutables. Para ello, utiliza un fichero que habitualmente se llama `Makefile` y que contiene una serie de reglas que indican cómo debe generarse el ejecutable teniendo en cuenta qué ficheros deben regenerarse cada vez que cambien otros. Cuando un fichero A debe regenerarse cuando cambie un fichero B, se dice que B es una dependencia de A.

La gran ventaja de la herramienta `make` es que automáticamente comprueba si los ficheros que se especifican como dependencias han sido modificados, y genera solo los ficheros que dependan de dichos ficheros modificados, con lo que se ahorra tiempo al generar los programas.

Las reglas de un fichero `Makefile` asocian un objetivo con las dependencias y los comandos que son necesarios ejecutar para, a partir de las dependencias, obtener el objetivo. Son de la forma:

```
objetivo : dependencias
<Tabulador> comandos
```

Antes de indicar el comando que crea el objetivo a partir de la lista de dependencias hay que utilizar un carácter de tabulación. A continuación se muestra un ejemplo de `Makefile` que puede utilizarse para compilar y enlazar el programa `1-1program` visto anteriormente.

MAKEFILE

```
# Rule for generating the main program
1-1program : 1-1program.o 1-1circle.o
    gcc 1-1program.o 1-1circle.o -o 1-1program

# Compile source files
1-1program.o : 1-1program.c 1-1circle.h
    gcc -c 1-1program.c

1-1circle.o : 1-1circle.c 1-1circle.h
    gcc -c 1-1circle.c

# Clean object files
clean :
    rm -f 1-1program.o 1-1circle.o
```

- Edita el fichero `Makefile` en el mismo directorio que los ficheros `1-1program.c`, `1-1circle.c` y `1-1circle.h` de forma que contega el listado mostrado.

  

  > Ten cuidado al copiar y pegar en la consola el listado del fichero, pues deberás sustituir los espacios por tabulaciones.

- Para que veas cómo compila make, borra todos los ficheros objeto con `rm *.o` y borra también `1-1program`.

- Genera el ejecutable invocando al comando `make`. Puedes hacerlo indicando el nombre del ejecutable a generar o bien sin parámetros, ya que la regla que genera el ejecutable es la primera del fichero `Makefile`.

```
$> make
gcc -c 1-1program.c
gcc -c 1-1circle.c
gcc 1-1program.o 1-1circle.o -o 1-1program
```

- Intenta volver a generar el ejecutable. El comando `make` comprobará que las dependencias del objetivo `1-1program` no han cambiado, por lo que inferirá que el ejecutable está actualizado.

```
$> make
make: '1-1program' está actualizado.
```

- Elimina los ficheros temporales generados invocando la regla `clean` del `Makefile`.

```
$> make clean
rm -f 1-1program.o 1-1circle.o
```

- Incorpora el fichero `Makefile` al repositorio *git*.

Puedes utilizar este fichero `Makefile` como patrón para generar el resto correspondientes a los programas que desarrollarás a lo largo de las prácticas. Debes tener cuidado de modificar apropiadamente los nombres de fichero que aparecen en el `Makefile` para adecuarlos al programa a generar.

4.4. Salida por pantalla

Para mostrar información por pantalla utilizaremos la función `printf`, que se encuentra declarada en el fichero de cabecera `stdio.h`. Para imprimir por pantalla una cadena de texto se podría utilizar la siguiente sentencia:

```c
printf("This string is shown on the screen");
```

Esta función recibe un número variable de parámetros, de los cuales el primero es una cadena de formato que puede contener:

- Texto. Este texto se mostrará en la pantalla tal cual.
- Secuencias de escape. Comienzan por el carácter `\` y van seguidas por otro carácter. Las secuencias `\n` y `\t` representan un salto de línea y una tabulación respectivamente.
- Especificadores de conversión. Se utilizan para mostrar el valor de variables. Comienzan por el carácter `%`:
  - `%c`: carácter (tipo `char`).
  - `%d`: entero decimal con signo (tipo `int` o `short`).
  - `%f`: real (tipo `float` o `double`).
  - `%s`: cadena de caracteres (tipo `char *`).
  - `%x`: entero en formato hexadecimal.

- `%p` : puntero.

El resto de parámetros de la función son las variables a mostrar que deben aparecer en el mismo orden que aparecen los especificadores de conversión.

- Edita el fichero `1-1output.c` con el contenido que se muestra a continuación.

```c
#include <stdio.h>

int main()
{
    int i = 3;
    float f = 2.4;
    char * s = "Content";

    // Print the content of i

    // Print the content of f

    // Print the content of the string


    return 0;
}
```

- Completa las líneas comentadas que faltan de tal forma que al ejecutar el programa lo que se muestre por pantalla sea lo siguiente.

```
i content: 3
f content: 2.400000
s content: Content
```

- Compila y enlaza el programa. En este punto deberías ser capaz de conocer los comandos necesarios para llevar a cabo ambas tareas usando un `Makefile`. Si tienes alguna duda pregúntale al profesor.
- Ejecuta el programa.

```
$> ./1-1output
```

- Incorpora el fichero fuente `1-1output.c` al repositorio *git*.

4.5. Punteros

Los punteros y su aritmética es con mucho la parte más complicada del lenguaje C, pero también la que le da la capacidad de manejar la memoria a un nivel muy bajo, básica para programación de sistemas. Un puntero es una variable que guarda una dirección de memoria, o lo que es lo mismo, que apunta a un dato en memoria, de ahí su nombre. Para declarar un puntero es necesario indicar el tipo de datos al que apunta junto con el carácter `*`. Algunas definiciones de punteros son las siguientes.

```c
char * pChar;  // Refers to a piece of data in memory 1-byte wide
int * pInt;    // Refers to an int in memory (usually 4-bytes wide)
double * pDouble1, * pDouble2;  // Refer to float numbers with double
                                // precision (8-bytes wide)
```

Un puntero definido de esta forma no puede ser directamente utilizado, ya que apunta a una zona de memoria desconocida. En lugar de ello, hay que hacer que el puntero apunte a una zona de memoria válida para poder leer y escribir en las posiciones de memoria a las que apunta. Esto puede hacerse de dos formas: haciendo que el puntero apunte a la zona de memoria de una variable del programa (utilizando el operador `&` para obtener la dirección de una variable), o haciendo que el puntero apunte a una zona de memoria reservada de forma dinámica, habitualmente en el montículo o *heap*.

```c
int main()
{
    int x;
    int * pInt;
    char * pChar;

    x = 3;
    pInt = &x;
    pChar = (char *)malloc(20 * sizeof(char));
    free(pChar);
    return 0;
}
```

Observa en el ejemplo anterior cómo se reserva memoria de forma dinámica en el montículo con la función `malloc`. El operador `sizeof` devuelve el tamaño en bytes de un tipo. Además, es necesario hacer una conversión (*casting*) al asignar la dirección al puntero, ya que la función retorna `void *`, que quiere decir «puntero a un tipo indeterminado».

A diferencia de lenguajes que incorporan recolector de basura como Java, en C es necesario liberar la memoria reservada de forma explícita con la función `free` cuando ya no se use.

Para utilizar los punteros se usa el operador indirección `*`. Cuando este operador precede al nombre de una variable puntero, significa que se hace referencia al valor de la memoria a la que el puntero apunta. Si a continuación del código anterior se añade la siguiente sentencia justo antes del `return`, la salida por pantalla sería 3 3.

```c
printf("%d %d", x, *pInt);
```

Un uso muy común de los punteros es en el paso de parámetros a una función. En C los parámetros se pasan por valor[4]. Esto quiere decir que si se utiliza una variable como parámetro de una función, se copia su valor a una nueva variable dentro de la función. Si se cambia este valor dentro de la función, el valor de la variable que está fuera de la función no cambiará.

Aunque una función siempre recibe los parámetros por valor (recibe una copia del valor de la variable), en la práctica se puede conseguir el paso por referencia si se le pasa la dirección de la variable, es decir, un puntero. En este último caso la función recibe una copia del puntero, con lo que es posible modificar los datos a los que apunta el puntero usando el operador de indirección pero no la dirección (el puntero en sí).

El siguiente código muestra un ejemplo de paso de una variable a un procedimiento por valor.

```c
#include <stdio.h>

void f(double d)
{
    d = 4;
}

int main(int argc, char* argv[])
{
    double d;
    d = 3;

    f(d);
    printf("%f\n", d);
    return 0;
}
```

- Genera el programa `1-1pointer1.c` a partir del listado anterior y comprueba cuál es el valor mostrado por pantalla.
- Ahora modifica el programa, llamándolo `1-1pointer2.c`, para que la función `f` en lugar de recibir un `double` reciba un puntero a un `double`. Además, tendrás que modificar la asignación para modificar

el contenido de la dirección apuntada por el puntero (utiliza el operador de indirección). Por último, deberás modificar la llamada a la función para que reciba la dirección de la variable `d`. Comprueba que el programa imprime ahora el valor `4`. Si no es así, consulta a tu profesor.

- Incorpora ambos ficheros fuente al repositorio *git*.

4.6. Estructuras

Las estructuras son agrupaciones de tipos básicos en memoria, de tal forma que se ubican en memoria varios elementos de tipos básicos uno a continuación de otro. El siguiente programa muestra cómo definir y utilizar una estructura denominada `Person` que incluye una cadena de texto, un entero y un real de precisión doble.

- Edita el fichero `1-1person.c` con el contenido que se muestra a continuación.

```c
#include <stdio.h>
#include <string.h>

struct _Person
{
    char name[30];
    int heightcm;
    double weightkg;
};
// This abbreviates the type name
typedef struct _Person Person;

int main(int argc, char* argv[])
{
    Person Peter;
    strcpy(Peter.name, "Peter");
    Peter.heightcm = 175;

    // Assign the weight

    // Show the information of the Peter data structure on the screen

    return 0;
}
```

- Completa las líneas comentadas que faltan de tal forma que al ejecutar el programa se muestre por pantalla lo siguiente.

```
Peter's height: 175 cm; Peter's weight: 78.7 kg
```

- Genera el ejecutable y ejecútalo.
- Incorpora el fichero al repositorio *git*.

También es posible crear punteros a estructuras. Volviendo a la estructura `Person` que creamos anteriormente, podemos definir un nuevo tipo puntero a dicha estructura.

```c
// Type name abbreviations
typedef struct _Person Person;
typedef Person* PPerson;

int main(int argc, char* argv[])
{
    Person Javier;
    PPerson pJavier;

    // Memory location of Javier variable is assigned to the pointer
    pJavier = &Javier;
    Javier.heightcm = 180;
    Javier.weightkg = 84.0;
    pJavier->weightkg = 83.2;
    return 0;
}
```

Fíjate cómo para acceder a los campos de una estructura a través del puntero se debe hacer uso del operador `->`. Por ejemplo, `pJavier->weightkg` es equivalente a `(*pJavier).weightkg`, es decir, acceder a donde apunta `pJavier` (que es una estructura de tipo `Person`) y luego a su campo `weightkg`.

- Modifica el fichero `1-1person.c` para añadir el tipo puntero a persona `PPerson` tal como se indicó anteriormente.
- Incorpora los cambios sobre el fichero al repositorio *git*.

Archivos de la práctica

Debes guardar todos los ficheros sobre los que has trabajado durante la práctica. Si has incorporado los cambios realizados a tu repositorio, basta con sincronizar tu repositorio con el almacenado en Bitbucket.

- Sincroniza tu repositorio en Bitbucket con el repositorio local. El siguiente comando sube a Bitbucket los cambios que has incorporado a tu repositorio local.

```
$> git push
```

- Comprueba en Bitbucket que están todos los ficheros.

Ejercicios

Se proponen los siguientes ejercicios adicionales:

- Haz un programa de nombre `1-1add.c` que defina una función `add` que reciba por parámetro un vector de enteros y su tamaño. Dicha función debe devolver como resultado la suma de todos los elementos del vector. La función `main` de dicho programa será la siguiente.

```c
#include <stdio.h>

#define NUM_ELEMENTS 7

int main()
{
    int vector[NUM_ELEMENTS] = { 2, 5, -2, 9, 12, -4, 3 };
    int result;

    result = add(vector, NUM_ELEMENTS);
    printf("The addition is: %d\n", result);
    return 0;
}
```

- Haz un programa de nombre `1-1copy.c` que copie cadenas de texto a través de la función `copy` que tendrá el siguiente prototipo.

```c
int copy(char * source, char * destination, unsigned int lengthDestination);
```

  En el primer parámetro recibe la cadena a copiar. Recuerda que las cadenas en C terminan con `0`. El segundo parámetro es un puntero a la zona de memoria donde copiar la cadena, mientras que el tercero indica el tamaño de esta zona, para evitar que la función escriba fuera de esta zona.

---

1. Ten cuidado al introducir comandos porque la interfaz de comandos distingue entre mayúsculas y minúsculas.
2. En Windows es común denominar carpetas a los directorios.
3. Un repositorio es un contenedor con el historial de cambios de los ficheros a él asociados.
4. En C++ las funciones pueden recibir referencias a variables (por ejemplo, el tipo de un parámetro puede ser `int&`, es decir, una referencia a entero). Sin embargo, C no tiene referencias.