# Entorno de desarrollo C para GNU/Linux

## Índice



## 1 Introducción

Uno de los objetivos de la asignatura *Sistemas Operativos* es profundizar en el conocimiento del comportamiento básico del sistema operativo LINUX ejercitando para ello los servicios que el *kernel* ofrece a través de la interfaz de las llamadas al sistema relacionadas con sus principales áreas de gestión: ficheros y directorios, procesos, memoria, señales y tiempo, y mecanismos de sincronización y comunicación.

Instrumentalmente necesitaremos emplear el lenguaje C desde el cual invocar las llamadas, y también necesitaremos disponer de un entorno de desarrollo que nos permita codificar programas de prueba de tamaño pequeño o mediano, así como asistir en la creación de proyectos formados por uno o varios ficheros fuente, con su correspondiente compilación y enlace hasta obtener un fichero ejecutable. Emplearemos además utilidades para ayudar a depurar los programas en ejecución, analizar el comportamiento de la ejecución y extraer información de los ficheros generados.

Existen disponibles entornos de desarrollo gráficos llamados genéricamente IDEs. En el laboratorio y en la máquina virtual de la asignatura está instalado VSCode pero en este manual hemos optado por explicar individualmente las herramientas de desarrollo desde terminales de texto por un doble motivo. Primero, hacer consciente al alumno de

los distintos pasos que se requieren en la preparación, construcción y ejecución de un proyecto de programación. En segundo lugar, deseamos prescindir de las necesidades administrativas de configuración que el IDE demanda.

Esencialmente las herramientas de desarrollo necesarias para generar y probar proyectos escritos en lenguaje C en el contexto de esta asignatura son un editor de texto –como `gedit` o el proporcionado por Visual Studio Code–, un compilador de C –como `gcc`–, la herramienta GNU `make`, y un depurador con soporte de C –como `gdb`–.

Nótese que todos los códigos de ejemplo a los que hará referencia en este documento se encuentran dentro de un fichero comprimido que puede descargarse usando este enlace.

## 2 Primeros pasos

Lo mínimo que necesitaremos para empezar a trabajar es arrancar el puesto de ordenador con el sistema operativo GNU/Linux[1] y abrir una ventana de terminal. De esta manera entramos a relacionarnos con GNU/Linux mediante el intérprete de órdenes (*shell*) asociado al terminal; se trata de un programa (generalmente corresponde a `/bin/bash`), que se encarga de solicitar líneas de órdenes al usuario y las manda ejecutar de modo interactivo tras un procesamiento previo que distingue entre órdenes locales, órdenes externas, variables, estructuras de programación y otras directivas.

Vamos a preparar un sencillo programa en C, a compilarlo, enlazarlo y ejecutarlo. No explicaremos las características del lenguaje C; en la carpeta "Material Adicional" disponible en este enlace el alumno dispone de breves introducciones, enlaces a tutoriales y referencias bibliográficas sobre el lenguaje C, que se deberán consultar y estudiar para obtener un nivel medio de conocimiento que permita elaborar los programas propuestos para ejercitar el uso de la API del sistema objetivo de la asignatura.

El alumno debe elegir un editor de los propuestos clase o algún otro de su preferencia; supondremos que el elegido es `gedit`[2]. Desde el terminal tecleamos:

```
$ gedit greetings.c &
```

e introducimos en el área de texto del editor el siguiente programa (también podemos acceder a los editores a través del entorno de escritorio instalado en la máquina virtual):

```
#include <stdio.h>

int main(void) {
        char name[100];

        printf("Enter your name: ");
        if (scanf("%s", name) != 1) {
                printf("Error/EOF\n");
                return 1;
        } else {
                printf("Hi %s!!\n", name);
                return 0;
        }
}
```

A continuación lo mandamos compilar con la siguiente orden

```
$ gcc -c greetings.c
```

que genera el fichero objeto de nombre `greetings.o`. Este fichero tiene formato ELF pero no está completo. Lo podemos comprobar con la siguiente orden:

[1] en las instalaciones de laboratorio de la FDI puede utilizarse la distribución Ubuntu instalada de forma nativa o la máquina virtual de Linux de la asignatura disponible tanto en Windows como Linux nativo

[2] Si este editor no estuviera disponible en la máquina virtual del laboratorio, bastaría instalarlo con el comando `sudo apt install gedit`, y teclear si la solicita la contraseña del usuario.

```
$ file greetings.o
greetings.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV),
 not stripped
not stripped
```

que nos dice que greetings.o contiene código ELF reubicable, pero no especifica que sea un fichero ejecutable. Faltan por resolver las referencias a funciones de biblioteca y la incorporación de código preparatorio al principio y al final, que lo conviertan en un fichero ejecutable definitivo. De ello se encarga la siguiente orden:

```
$ gcc -o greetings greetings.o
```

De nuevo lo comprobamos:

```
$ file greetings
greetings: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.26,
BuildID[sha1]=0xeb23a88b880589e556896f4d9412d5a67a3a83fa, not stripped
```

que informa que greetings es ya un ejecutable final, enlazado, eso sí, con bibliotecas dinámicas (compartidas) que se agregarán a la imagen del proceso en tiempo de ejecución.

Podemos ejecutar el programa a continuación:

```
$ ./greetings
Enter your name: Mike
Hi Mike!!
```

# 3 Compilación de proyectos en Linux

En programación consideramos un proyecto el conjunto de ficheros necesarios para producir una aplicación. Para cada aplicación emplearemos un directorio donde se alojarán todos los ficheros relacionados con ella, que básicamente son de 6 tipos:

1. Uno o más ficheros fuente escritos en lenguaje C (con extensión .c).
2. Uno más ficheros objeto generados por compilación de cada fichero fuente (con extensión .o).
3. Uno o más ficheros de cabecera referenciados mediante sentencias #include en los ficheros fuente (con extensión .h).
4. Un fichero de nombre Makefile que contendrá las reglas de construcción de la aplicación y que la utilidad make consulta, habitualmente para generar el ejecutable final.
5. Opcionalmente, uno o más ficheros de biblioteca que contendrán código objeto.
6. Un fichero ejecutable.

## 3.1 Compilador GCC

Para compilar los ficheros fuente utilizaremos el compilador de C de GNU, gcc. En realidad gcc es una herramienta que sirve de interfaz común para el uso de distintas herramientas del proceso de compilación, como son el preprocesador, el compilador propiamente dicho (cc), el ensamblador (as) y el enlazador (ld). El proceso que normalmente llamamos compilación es una secuencia de los siguientes pasos:

1. Un preprocesador (`cpp`), que toma como entrada un fichero fuente `archi.c` y lo transforma en un fichero intermedio archi.i mediante la realización de las operaciones de pre-proceso textual demandadas por las instrucciones `#xxx` que aparecen en el código fuente, denominadas macros del preprocesador. Las principales macros son

`#include`, `#ifdef`, `#ifndef`, `#else`, `#endif`, `#define`,... (Consultar un tutorial de Lenguaje C para conocer el significado de tales instrucciones).

2. Un compilador de C (`cc`) que toma como entrada el fichero intermedio `archi.i` y produce código textual en lenguaje ensamblador propio de la arquetectura destino, que almacena en un fichero `archi.s`.
3. Un ensamblador (`as`), que toma como entrada el fichero `archi.s` y genera código máquina que almacena en un fichero con formato ELF reubicable de nombre `archi.o`.
4. Un enlazador (`ld`) que monta uno o varios ficheros .o, resuelve las referencias cruzadas entre ellos y con bibliotecas de código objeto, especialmente la biblioteca básica de C de nombre `libc.a` (versión de enlace estático) o `libc.so` (versión de enlace dinámico). Si no queda ninguna referencia por resolver, el enlazador genera un fichero ELF ejecutable, de nombre `a.out` por defecto, utilizable como punto de partida para crear una imagen de proceso.

La herramienta `gcc` admite una serie de opciones al ser invocada; las más usadas se muestran en la siguiente tabla:

| Opción | Explicación |
|---:|---|
| -c | Realiza sólo los tres primeros pasos: preproceso, compilación y ensamblaje |
| -g | Durante la compilación crea una tabla de símbolos (asociación entre nombres de funciones o variables y direcciones de referencia) que almacena en los ficheros objeto y ejecutable generados y que puede ser consultada por una utilidad de depuración para permitir depuración simbólica |
| -Ox | Demanda aplicar distintos niveles de optimación durante el proceso de compilación y generación de código máquina. Los niveles más usuales son **s** (optimización en tamaño), **0** (anular optimización), **1**, **2** y **3** (distintos grados cada vez más exigentes de optimización) |
| -o nombre | Propone nombre como nombre del fichero de salida resultante en vez del nombre utilizado por defecto (que, en caso del ejecutable final, es a.out) |
| -I directorio | Instruye al preprocesador para que busque los ficheros de cabecera referenciados en el directorio designado además de los directorios por defecto /include y /usr/include |
| -D xxx | Define una variable de preproceso de nombre `xxx`. También se le puede dar un valor si se utiliza `-D xxx=valor` |
| -L directorio | Instruye al enlazador a que busque las bibliotecas propuestas, en el directorio designado además de los directorios por defecto `/lib` y `/usr/lib` |
| -lxxx | Instruye al enlazador para que utilice la biblioteca de nombre `libxxx.a` (versión estática) o `libxxx.so` (versión dinámica) existente en algunos de los directorios especificados implícita o explícitamente |
| -static o -dynamic | Escoge entre enlace con bibliotecas estáticas o dinámicas (si no se indica nada, se aplica enlace dinámico) |
| -v | Muestra información sobre cada uno de los pasos dados durante el proceso de compilación general |
| -Wall | Habilita los avisos del compilador para múltiples errores comunes, es muy recomendable usarlo siempre |

Vamos a ver el uso de algunas opciones de `gcc` y su resultado en la compilación, y examinaremos con algún detalle qué ocurre durante el proceso de compilación. Comenzaremos con un ejemplo muy simple, una aplicación formada por dos ficheros: `archi.c` (fuente) y `archi.h` (cabecera):

```
$ cat archi.c
#include "archi.h"
```

```
int main(void)
{
       printf("Hello ...%d\n", VAR);
}

$ cat archi.h
#define VAR 100
```

Como ya dijimos, la primera etapa de compilación es el preprocesado. El preprocesador examina los ficheros .c en busca de declaraciones que comiencen por #; cuando son del tipo #include localiza el fichero designado y lo añade al programa. A continuación el preprocesador busca y sustituye todas las macros (#define) y realiza las sustituciones de texto indicadas; el compilador propiamente dicho (etapa segunda), no compila el fichero .c original sino la suma del .c y .h con las macros sustituidas. Esto puede producir algunos problemas serios si el fichero de cabecera contiene errores. Cuando se emplean macros que se comportan como funciones, y existe algún error en la macro, puede resultar difícil detectar la causa del error; el compilador no sabe de cabeceras, y cuando ve un error, informa acerca del número de línea que él ve. Al buscar tal línea en el fichero .c original puede que nos encontremos una sentencia aparentemente inocente.

Vamos a pedir a gcc que genere un fichero intermedio como salida de cada una de sus 4 etapas:

```
$ gcc --save-temps archi.c
archi.c: In function ‘main’:
archi.c:5:2: warning: implicit declaration of function ‘printf’ [-Wimplicit-function-declaration]
  printf("Hello ...%d\n", VAR);
  ^~~~~~
archi.c:5:2: warning: incompatible implicit declaration of built-in function ‘printf’
archi.c:5:2: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
archi.c:2:1:
+#include <stdio.h>

archi.c:5:2:
  printf("Hello ...%d\n", VAR);
  ^~~~~~
$ ls -l a.out archi.*
-rwxr-xr-x 1 501 dialout 16608 jul 22 19:40 a.out
-rw-r--r-- 1 501 dialout    71 jul 22 16:38 archi.c
-rw-r--r-- 1 501 dialout    17 jul 22 16:38 archi.h
-rw-r--r-- 1 501 dialout   234 jul 22 19:40 archi.i
-rw-r--r-- 1 501 dialout  1544 jul 22 19:40 archi.o
-rw-r--r-- 1 501 dialout   483 jul 22 19:40 archi.s
```

(Olvidemos, de momento, el aviso producido referente a la línea 5). El fichero generado por el preprocesador es archi.i:

```
$ cat archi.i
# 1 "archi.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "archi.c"
# 1 "archi.h" 1
# 2 "archi.c" 2

int main(void)
{
 printf("Hello ...%d\n", 100);
}
```

Se ha producido la sustitución de la macro VAR. También podemos ver el resultado de la segunda etapa, la compilación de C, que genera un fichero fuente en código ensamblador:

```
$ cat archi.s
        .file   "archi.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello ...%d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $100, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 8.3.0-6) 8.3.0"
        .section        .note.GNU-stack,"",@progbits
```

El siguiente paso, el ensamblado, genera un fichero objeto con formato de código binario reubicable accesible como archi.o; podemos comprobar su formato con la orden

```
$ file archi.o
archi.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV),
not stripped
```

El último paso, el enlazado, agrega fundamentalmente código de biblioteca para generar un fichero ejecutable final listo para servir como origen de ejecución de procesos; por defecto, gcc le da el nombre a.out. También está en formato ELF:

```
$ file a.out
a.out: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=71b3e2d6ee99b470f5254e40ff5360ec0dd0b8d7, not stripped
```

Si nos interesa podemos aplicar una o varias de las etapas de compilación separadamente. Por ejemplo, podemos generar ficheros objeto (tres primeras etapas) y luego enlazar separadamente.

A modo de ejemplo, procederemos a generar el fichero objeto a partir del programa `archi.c`.

```
$ gcc -c archi.c -o archi.o
archi.c: In function ‘main’:
archi.c:5:2: warning: implicit declaration of function ‘printf’ [-Wimplicit-function-declaration]
  printf("Hello ...%d\n", VAR);
```

```
  ^~~~~
archi.c:5:2: warning: incompatible implicit declaration of built-in function ‘printf’
archi.c:5:2: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
archi.c:2:1:
+#include <stdio.h>

archi.c:5:2:
  printf("Hello ...%d\n", VAR);
  ^~~~~
```

Con la opción `-c` indicamos a `gcc` que construya únicamente el fichero objeto, cuyo nombre se especifica con la opción `-o`; es decir, en este caso prescinde de la operación de enlazado. A pesar de que el fichero objeto se ha generado correctamente, el compilador genera una advertencia sobre el desconocimiento de la función `printf()`. Se deja como ejercicio realizar las modificaciones pertinentes en el programa para eliminar esta advertencia. **Nota**: Para averiguar qué fichero(s) de cabecera se ha(n) de incluir en nuestro programa al usar una función estándar no definida en el mismo se ha de consultar página de manual de la función en cuestión.

Para llevar a cabo la etapa final de enlace podría invocarse manualmente al enlazador (comando `ld`). Sin embargo, esto nos obligaría a proporcionar al enlazador información adicional acerca del tipo de enlace del programa (estático o dinámico), así como sobre la ubicación de las bibliotecas y ficheros auxiliares donde se encuentran definidos los simbolos (funciones y variables) que el programa utiliza pero no define. El comando `gcc`, que puede emplearse también como *frontend* del enlazador, simplifica enormemente esta labor. En particular, con `gcc` podemos proceder al enlace empleando simplemente el siguiente comando:

```
$ gcc archi.o -o archi
```

En el ejemplo anterior, el programa solo utiliza símbolos de la biblioteca estándar de C, por lo que no se precisan opciones adicionales a `gcc` para llevar a cabo el enlace satisfactoriamente. Sin embargo, cuando se emplean símbolos de otras bibliotecas, es necesario especificar explíctamente el nombre de cada biblioteca dependiente empleando la opción `-l<nombre_biblioteca>` de `gcc`. No hacerlo provoca errores de enlazado. Por ejemplo, a la hora de crear programas multihilo en Linux, utilizaremos la biblioteca POSIX Threads. Para el correcto enlazado de estos programas es necesario indicar la opción `-lpthread`.

## 3.2 GNU Make

Es frecuente durante el desarrollo del proyecto que haya que compilar dicho proyecto multitud de veces. Además, el proyecto va cambiando, añadiéndose ficheros, bibliotecas con las que se debe enlazar, ejecutables y bibliotecas que se deben generar, etc. Repetir en cada ocasión las llamadas a `gcc` para compilar cada objetivo del proyecto es ineficiente y propenso a errores. La popular herramienta Make, diseñada para entornos UNIX, tiene como objetivo facilitar y, en cierta medida, automatizar este proceso de compilación del proyecto.

Hay distintas versiones de la herramienta Make, sin embargo todas ellas funcionan de un modo muy parecido. Cuando se invoca, Make interpreta un fichero de texto con una sintaxis especial y nombre preestablecido (p.ej. *makefile*), que le indica los pasos que debe seguir para compilar el proyecto. Una vez confeccionado el *makefile* repetir la compilación es tan sencillo como ejecutar `make` desde el terminal.

En esta asignatura emplearemos la implementación de Make de GNU, que se encuentra instalada en el laboratorio. Cuando el usuario invoca el comando `make` desde el *shell*, la herramienta busca en primer lugar un fichero que se llama `GNUmakefile`. Si no se encuentra se busca un fichero llamado `makefile`, y si por último no se encontrase, se buscaría el fichero `Makefile`. Si no se encuentra en el directorio actual ninguno de esos tres ficheros, se producirá un error y make no continuará:

```
$ make
make: *** No targets specified and no makefile found.  Stop.
```

En caso contrario, `make` procesará el archivo de texto e invocará el conjunto mínimo de reglas requeridas para construir nuestro proyecto. Para ilustrar el flujo de trabajo de make, consideremos el siguiente ejemplo. Supongamos que tenemos un proyecto C que consta de un *Makefile* correctamente construido y dos archivos .c, que deben compilarse y enlazarse para construir el ejecutable de un programa. Normalmente, las reglas del fichero *Makefile* garantizan que si solo se ha modificado un archivo C desde que se invocó `make` por última vez, una nueva invocación de `make` realizará la compilación solo para el archivo C modificado y la etapa final de enlazado. Por tanto, el fichero objeto intermedio asociado al archivo C sin modificar no se volverá a generar.

En este curso, la mayor parte de los proyectos C de ejemplo proporcionados con las distinas prácticas se acompañan de un fichero *Makefile* específico para el proyecto. Por lo general, para compilar el proyecto asociado bastará teclear `make` en la terminal. Además, la mayoría de los *Makefiles* contarán también con una regla de limpieza para eliminar todos los archivos intermedios generados durante el proceso de compilación. En particular, al ejecutar el comando `make clean` desde el directorio principal del proyecto, los archivos de objeto y ejecutables correspondientes se eliminarán automáticamente.

Cabe destacar que durante el desarrollo de la asignatura los estudiantes no tendrán que elaborar ficheros *Makefile* de cero, sino adaptar mínimamente ejemplos ya proporcionados. Por lo tanto en este documento no procederemos a introducir la sintaxis de GNU make. No obstante, esta sintaxis está descrita en el tutorial disponible en este enlace.

# 4 Depuración

La utilidad de depuración en ejecución de programas C en GNU/Linux es, por antonomasia, el depurador `gdb` de GNU. Existen interfaces gráficas que facilitan su utilización bien de forma autónoma (p.ej., la aplicación `ddd` — *Data Display Debugger*), bien incluido dentro de un IDE (p.ej. Visual Studio Code).

## 4.1 Primeros pasos con `gdb`

Para ilustrar como lanzar un programa con el depurador usaremos el programa `example.c`:

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

int main(int argc, char *argv[]) {

  if (argc<3){
        fprintf(stderr,"Usage: %s <first-name> <last-name>\n",argv[0]);
        return 1;
  }

  printf("Hi %s %s!\n",argv[1],argv[2]);
  return 0;
}
```

Este programa acepta dos parámetros (un nombre y apellido), e imprime un mensaje de saludo construido en base a esos parámetros. Compilaremos el programa en modo depuración, usando la opción `-g` de `gcc`, junto con `-Wall` para mostrar todas las advertencias (*warnings*):

```
$ gcc -g -Wall example.c -o example
```

Procederemos entonces a lanzar el depurador, pasando el nombre del ejecutable como argumento del comando `gdb`. Esto nos dará acceso al *prompt* de GDB:

```
$ gdb example
GNU gdb (Debian 8.2.1-2+b3) 8.2.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

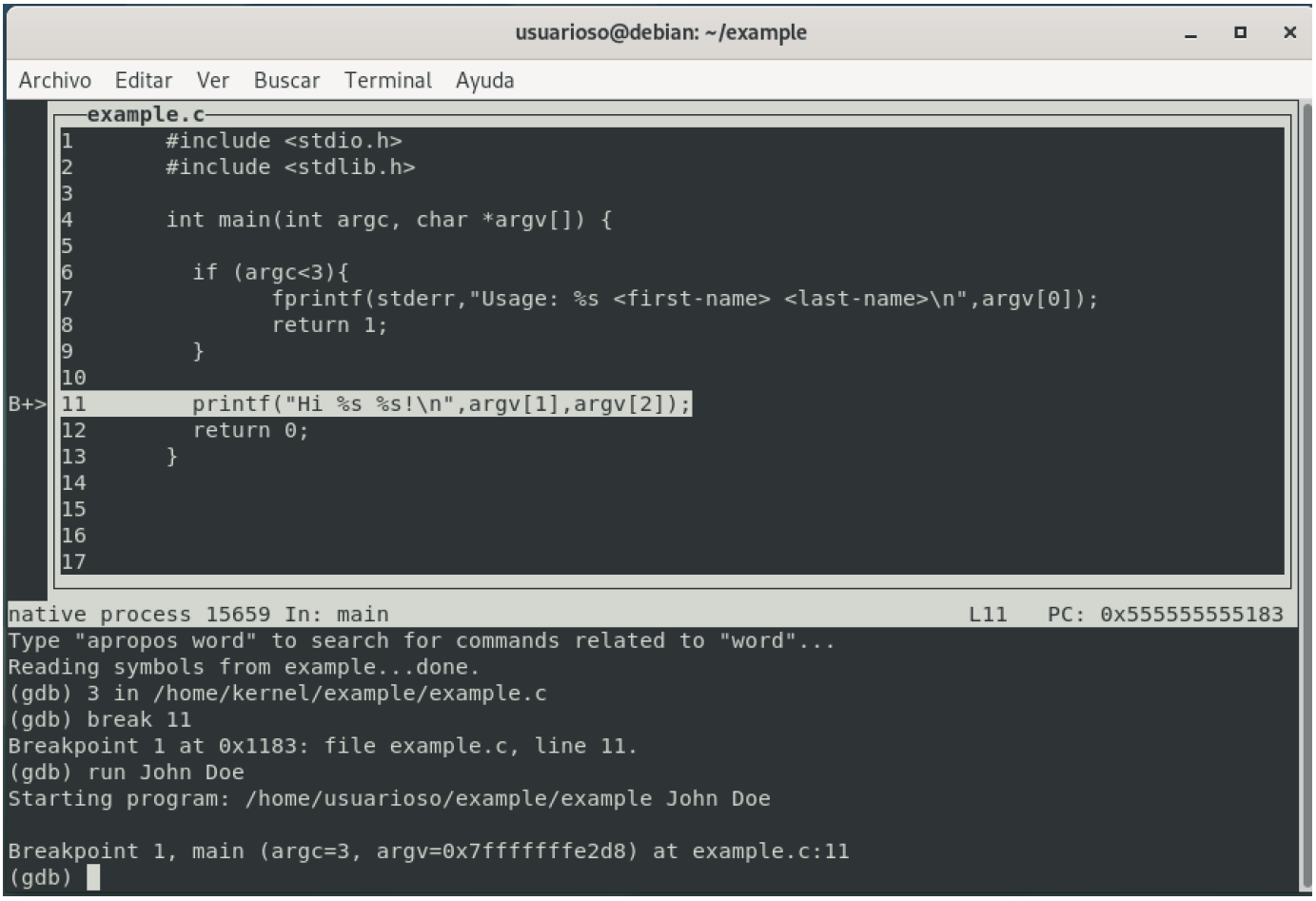
```
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from example...done.
(gdb)
```

Para lanzar el programa con el depurador usaremos el comando `run` de GDB, que acepta como parámetros los argumentos de línea de comandos que acepte el programa (dos en este caso). Como no hemos establecido *breakpoints* en la sesión de depuración, el comando `run` ejecutará el programa de principio a fin, y mostrará la salida junto con los mensajes del depurador:

```
(gdb) run John Doe
Starting program: /home/usuarioso/example/example John Doe
Hi John Doe!
[Inferior 1 (process 14153) exited normally]
```

Una de las limitaciones de la depuración usando la interfaz de texto básica de GDB es el hecho de que visualizar el punto actual de ejecución del programa requiere ejecutar comandos como `list` o `backtrace`, que se presentan en la siguiente sección. El modo TUI de GDB ofrece una interfaz alternativa, que en su versión por defecto divide la pantalla en dos partes, mostrando el código en la parte superior, y el *prompt* de gdb en la parte inferior, como en la siguiente figura:




Cuando se alcanza un *breakpoint*, o se ejecuta el programa paso a paso la línea actual queda resaltada como se muestra en la figura, haciendo la depuración más amigable.

Existen dos opciones para activar la interfaz TUI de `gdb`:

1. Lanzar el depurador con la opción `-tui`. Ejemplo: `gdb -tui example`.
2. Utilizar el comando de GDB `tui enable`.

Análogamente el modo TUI puede desactivarse en cualquier momento con el comando `tui disable`. Para más información sobre GDB TUI puede consultarse la documentación oficial. Asimismo resulta de gran interés la visualización del vídeo de Jacob Sorber sobre la potencia de los distintos layouts de GDB TUI.

## 4.2 Depuración de un programa de ejemplo

Procedamos ahora a utilizar `gdb` para depurar un programa que se ejecuta incorrectamente, y así poner de manifiesto las principales facilidades de que dispone.

El programa se llama `badsort.c` y realiza una ordenación por el método de la burbuja aplicada a un *array* de datos inicializado dentro del programa. El código se suministra como parte de los ficheros de la primera práctica.

Un aviso (*warning*) del compilador nos ayudará a detectar el primer error. Le pedimos a `gcc` que nos informe de cualquier circunstancia sospechosa del código aún cuando no sea necesariamente un error fatal. Para ello utilizaremos la opción `-Wall`. También emplearemos la opción `-g` para permitir la depuración simbólica del programa en ejecución. Escribimos pues:

```
jcsaez@debian:~/Tests$ gcc -Wall -g -o badsort badsort.c
badsort.c: In function 'sort':
badsort.c:20:17: warning: suggest parentheses around comparison
in operand of '&' [-Wparentheses]
```

En el mensaje de compilación nos llaman la atención sobre la línea 20 que contiene
`/* 20 */ for(; i < n & s != 0; i++) {`

El aviso nos sugiere colocar paréntesis para dejar claro qué comparaciones se pretenden realizar. Una forma podría ser `(i < n) & (s != 0)`. Con un poco de atención podríamos darnos cuenta de que `&` es la operación AND bit-a-bit en C y no la operación lógica Y, que se escribe `&&`. Por lo que la sentencia está mal escrita y debería ser:
`/* 20 */ for(; (i < n) && (s != 0); i++) {`

Corregimos, volvemos a compilar y esta vez no recibimos ninguna indicación de error ni aviso.
`jcsaez@debian:~/Tests$ gcc -Wall -g -o badsort badsort.c`

Mandamos ejecutar el programa y observamos la siguiente salida:
```
jcsaez@debian:~/Tests$ ./badsort
Segmentation fault
```

Se ha producido una violación de segmento, indicativo de que el programa ha intentado acceder a una dirección de memoria que no es válida. Para averiguar la causa de este error recurrimos a ejecutar de nuevo el programa, esta vez bajo el control del depurador `gdb` sin el modo TUI (para centrarnos en el uso de comandos del depurador):

```
jcsaez@debian:~/Tests\$ gdb ./badsort
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/jcsaez/Tests/badsort...done.
```

Ordenamos a `gdb` que ejecute el programa con `run` (en este caso el programa no acepta argumentos):

```
(gdb) run
Starting program: /home/jcsaez/Tests/badsort

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400595 in sort (a=0x600c20, n=5) at badsort.c:23
23  /*  23  */                      if(a[j].key > a[j+1].key) {
```

Cuando el programa falla, `gdb` muestra la razón y la posición donde se produjo el fallo. Podemos investigar ahora la causa. El fallo aparece en la línea 23. Podemos indagar qué funciones había activas en el momento del fallo con la orden `backtrace` (`bt` en corto):

```
(gdb) bt
#0  0x0000000000400595 in sort (a=0x600c20, n=5) at badsort.c:23
#1  0x000000000040078a in main () at badsort.c:37
```

Podemos ver que la función `sort()` había sido invocada desde la función `main()` en la línea 37 del mismo fichero `badsort.c` con dos argumentos, la dirección inicial del array de datos por ordenar(`0x600c20`) y `n`, el número de

entradas del *array* (5).

La línea culpable del fallo es una comparación de un elemento del *array* con otro:
`if(a[j].key > a[j+1].key)`

Podemos examinar el contenido de los parámetros de las funciones, de las variables locales y de los datos globales con la orden `print`. Por ejemplo:

```
(gdb) print j
$2 = 4
```

El hecho de que `j` tenga el valor 4 significa que el programa ha intentado ejecutar la sentencia:
`if(a[4].key > a[5].key)`

El *array* que se ha pasado a `sort()` tiene sólo 5 elementos, por lo que los índices válidos van de 0 a 4. La sentencia pretende acceder a un elemento a[5] inexistente. Eso significa que la variable `j` ha tomado un valor incorrecto.

Otros ejemplos de órdenes para examinar variables:

```
(gdb) print a[3]
$3 = {data = "alex", '\0' <repeats 4091 times>, key = 1}
(gdb) print j
$4 = 4
(gdb) print a[$-1].key
$5 = 1
```

`$` hace referencia al último valor examinado, y `print` admite casi cualquier expresión con sentido en lenguaje C.

Una orden muy útil es `list` que permite ver el código fuente del programa alrededor de la posición actual. También se le puede dar como argumento el nombre de una función, o un par de números que representen el rango de líneas de interés.

```
(gdb) list
18      /*  18  */      int s = 1;
19      /*  19  */
20      /*  20  */      for(; (i < n) && (s != 0); i++) {
21      /*  21  */              s = 0;
22      /*  22  */              for(j = 0; j < n; j++) {
23      /*  23  */                    if(a[j].key > a[j+1].key) {
24      /*  24  */                            item t = a[j];
25      /*  25  */                            a[j] = a[j+1];
26      /*  26  */                            a[j+1] = t;
27      /*  27  */
```

Podemos ver en la línea 22 que el bucle está preparado para ejecutarse mientras la variable `j` sea menor que `n`. En este caso `n` vale 5 por lo que `j` tendrá 4 como valor final, pero ese valor es excesivo. Una solución a este problema particular es corregir la condición de terminación del bucle para que sea `j < (n-1)`.

Abandonamos el depurador, corregimos la línea 22 con el editor, recompilamos y probamos de nuevo.

```
jcsaez@debian:~/Tests$ gcc -g -o badsort badsort.c
jcsaez@debian:~/Tests$ ./badsort
array[0] = {john, 2}
array[1] = {alex, 1}
array[2] = {bill, 3}
array[3] = {neil, 4}
array[4] = {rick, 5}
```

El programa sigue sin funcionar; la lista no está bien ordenada. Volvamos a usar `gdb`, en este caso, para seguir su

ejecución paso a paso. Colocaremos puntos de ruptura (*breakpoints*) para detener la ejecución en las sentencias que interesen. La función `sort()` tiene dos bucles. El bucle exterior, con variable de control `i`, se ejecuta una vez por cada elemento del *array*. El bucle interior intercambia el elemento con los que se encuentran por debajo en la lista. Esto tiene el efecto de hacer emerger los elementos más pequeños a la cima de la lista. Después de cada pasada del bucle exterior, el elemento con mayor valor habrá descendido al fondo de la lista. Podemos confirmar esta forma de actuar deteniendo el programa en el bucle exterior y examinando el estado del *array*.

Comenzamos colocando un *breakpoint* en la línea 21 y ejecutando el programa:

```
jcsaez@debian:~/Tests$ gdb ./badsort
GNU gdb 7.4.1-debian
..
(gdb) break 21
Breakpoint 1 at 0x40053e: file badsort.c, line 21.
(gdb) run
Starting program: /home/jcsaez/Tests/badsort

Breakpoint 1, sort (a=0x600c20, n=5) at badsort.c:21
21  /*  21  */              s = 0;
(gdb)
```

Podemos imprimir el valor del *array* y dejar luego que el programa continúe con la orden `cont`. El programa seguiría así hasta el próximo punto de ruptura, en este caso, hasta que se ejecute la línea 21 de nuevo. Se pueden tener varios puntos de ruptura activos al mismo tiempo.

```
(gdb) print array[0]  # prints the first array item
$1 = {data = "bill", '\0' <repeats 4091 times>, key = 3}
(gdb) print array[0]@5  # prints the first five array items
$2 = {{data = "bill", '\0' <repeats 4091 times>, key = 3}, {
    data = "neil", '\0' <repeats 4091 times>, key = 4}, {
    data = "john", '\0' <repeats 4091 times>, key = 2}, {
    data = "rick", '\0' <repeats 4091 times>, key = 5}, {
    data = "alex", '\0' <repeats 4091 times>, key = 1}}
(gdb) cont      # resumes the execution
Continuing.

Breakpoint 1, sort (a=0x600c20, n=4) at badsort.c:21
21      /*  21  */              s = 0;
(gdb) print array[0]@5  # The biggest item is located at the end of the array
$3 = {{data = "bill", '\0' <repeats 4091 times>, key = 3}, {
    data = "john", '\0' <repeats 4091 times>, key = 2}, {
    data = "neil", '\0' <repeats 4091 times>, key = 4}, {
    data = "alex", '\0' <repeats 4091 times>, key = 1}, {
    data = "rick", '\0' <repeats 4091 times>, key = 5}}
```

Se puede usar la orden `display` para visualizar el *array* automáticamente cada vez que el programa se detenga en un punto de ruptura. Y se puede cambiar el punto de ruptura para que, en vez de detener el programa, visualice los datos y continúe; para ello se usa la orden `commands`:

```
(gdb) display array[0]@5
1: array[0] @ 5 = {{data = "bill", '\0' <repeats 4091 times>, key = 3}, {
    data = "john", '\0' <repeats 4091 times>, key = 2}, {
```

```
    data = "neil", '\0' <repeats 4091 times>, key = 4}, {
    data = "alex", '\0' <repeats 4091 times>, key = 1}, {
    data = "rick", '\0' <repeats 4091 times>, key = 5}}
(gdb) commands
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>cont
>end
```

Cuando ahora se deje al programa que continúe, se ejecutará hasta el final mostrando el valor del *array* cada vez que efectúe una pasada por el bucle exterior.

```
(gdb) cont
Continuing.

Breakpoint 1, sort (a=0x600c20, n=3) at badsort.c:21
21      /*  21  */              s = 0;
2: array[0] @ 5 = {{data = "john", '\0' <repeats 4091 times>, key = 2}, {
    data = "bill", '\0' <repeats 4091 times>, key = 3}, {
    data = "alex", '\0' <repeats 4091 times>, key = 1}, {
    data = "neil", '\0' <repeats 4091 times>, key = 4}, {
    data = "rick", '\0' <repeats 4091 times>, key = 5}}
array[0] = {john, 2}
array[1] = {alex, 1}
array[2] = {bill, 3}
array[3] = {neil, 4}
array[4] = {rick, 5}

Program exited normally.
```

El programa no parece ejecutar el bucle exterior tantas veces como cabía esperar. Podemos ver que el valor del parámetro `n`, usado en la terminación del bucle, se va reduciendo en cada punto de ruptura. Esto significa que el bucle no se ejecuta suficientes veces. El culpable es la sentencia que decrementa `n` en la línea 30.

```
30      /*  30  */              n--;
```

Se ha tratado de un intento de optimizar el programa aprovechando el hecho de que al final de cada bucle exterior el mayor valor del *array* se habrá colocado al final de la lista, y por tanto habrá quedado un elemento menos que ordenar. Pero, como se ha visto, esto interfiere con el bucle exterior y provoca problemas. La solución más simple (aunque hay otras) es eliminar la línea en cuestión. Vamos a probar si este cambio arregla el problema usando el depurador para aplicar un parche.

Necesitamos parar el programa en la línea 30 e incrementar la variable `n` para contrarrestar el decremento y así dejar su valor inalterado. Tendremos que reiniciar el programa desde el principio, deshabilitando primero los *breakpoints* y *display* activos (también se pueden eliminar pero de este modo retenemos la posibilidad de activarlos más tarde).

```
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
1:   y  array[0] @ 5
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000000000040053e in sort at badsort.c:21
        breakpoint already hit 3 times
```

```
        cont
(gdb) disable break 1
(gdb) disable display 1
(gdb) break 30
Breakpoint 2 at 0x400754: file badsort.c, line 30.
(gdb) commands 2
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>set variable n = n+1
>cont
>end
```

Mandamos ejecutar el programa en estas nuevas condiciones:

```
(gdb) run
Starting program: /home/jcsaez/Tests/badsort

Breakpoint 2, sort (a=0x600c20, n=5) at badsort.c:30
30      /*  30  */              n--;

Breakpoint 2, sort (a=0x600c20, n=5) at badsort.c:30
30      /*  30  */              n--;

Breakpoint 2, sort (a=0x600c20, n=5) at badsort.c:30
30      /*  30  */              n--;

Breakpoint 2, sort (a=0x600c20, n=5) at badsort.c:30
30      /*  30  */              n--;

Breakpoint 2, sort (a=0x600c20, n=5) at badsort.c:30
30      /*  30  */              n--;
array[0] = {alex, 1}
array[1] = {john, 2}
array[2] = {bill, 3}
array[3] = {neil, 4}
array[4] = {rick, 5}

Program exited normally.
```

Ahora sí funciona bien el programa y proporciona el resultado correcto. Podemos incorporar el cambio al fichero fuente (eliminar la línea 30) y quizás probar más exhaustivamente con un mayor número de datos para convencernos de la corrección del programa.

### 4.3 Depuración de un proceso existente

Es posible entrar a depurar una aplicación que ya está ejecutándose mediante el recurso de asociar (*attach*) `gdb` al proceso. Todo lo que se necesita es conocer el PID (identificador numérico único) del proceso que depurar y emplearlo en la invocación del depurador:

```
jcsaez@debian:~/Tests$ gdb programa PID
o con la orden attach de gdb
(gdb) attach PID
```

En cualquier caso el programa pasa a situación de "detenido" y el depurador se hace cargo de él, pudiéndose listar la posición actual de ejecución, examinar variables, colocar puntos de ruptura, ejecutar paso a paso, etc.. Cuando se ha finalizado la sesión de depuración se puede devolver la autonomía al proceso con la orden `detach`.

### 4.4 Depuración post-mortem de un proceso

Cuando una aplicación aborta y genera un volcado en un fichero de memoria (*core*), `gdb` puede servir para identificar qué ha ocurrido. Primero hay que preparar el sistema para que se pueda generar el fichero *core*, con la orden de *shell*:
`jcsaez@debian:~/Tests$ ulimit -c unlimited`

Al abortar el programa, ahora genera un fichero imagen de memoria que suele denominarse `core` o `core.pid`. Le aplicamos el depurador con la orden:
`jcsaez@debian:~/Tests$ gdb programa core`
lo que nos permitirá examinar la situación en que el programa finalizó abruptamente. No es posible ejecutar partes del programa o fijar puntos de ruptura, pero sí imprimir variables, conocer el estado de la pila, etc.

### 4.5 Otras ordenes útiles de gdb

| Comando | Descripción |
|---:|---|
| `quit` | Finaliza `gdb` |
| `step` | Ejecuta la sentencia siguiente. Si contiene la llamada a una función, entra en ella. Admite un parámetro `n`, para ejecutar `n` sentencias. |
| `next` | Ejecuta la sentencia siguiente. Si contiene la llamada a una función, la ejecuta completamente y sigue. Admite un parámetro `n`, para ejecutar `n` sentencias |
| `set follow-fork-mode child` | Al ejecutar un *fork* el depurador sigue al proceso hijo |
| `set follow-fork-mode parent` | Al ejecutar un *fork* el depurador sigue al proceso padre |

## 5 Páginas de manual

La mayoría de los sistemas LINUX suministran páginas de manual junto con sus aplicaciones, accesibles en línea. La orden `man` es el recurso más habitual para consultar información respecto al uso y finalidad de las órdenes, funciones y ficheros disponibles en el sistema. El software GNU ofrece un sistema de documentación en línea alternativo llamado `info`. La ventaja de `info` es que se puede navegar por la documentación utilizando enlaces y referencias cruzadas para saltar directamente a las secciones relevantes.

Si quisiéramos conocer las opciones que el compilador `gcc` admite podríamos hacerlo con:

```
jcsaez@debian:~/Tests$ man gcc
```

que nos mostraría

```
GCC(1)                          GNU                                GCC(1)

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
```

```
            [-Idir...] [-Ldir...]
            [-Dmacro[=defn]...] [-Umacro]
            [-foption...] [-mmachine-option...]
            [-o outfile] [@file] infile...

        Only the most useful options are listed here; see below for the
        remainder.  g++ accepts mostly the same options as gcc.

DESCRIPTION
        When you invoke GCC, it normally does preprocessing, compilation,
```

O podríamos hacerlo con

```
jcsaez@debian:~/Tests$ info gcc
```

También se puede emplear `xman` que es una interfaz de ventana para consultar las páginas de manual.

Las páginas de manual están divididas en secciones, siendo la clasificación más típica la siguiente:

| **Secciones** | **Contenidos** |
|---|---|
| 1 | Ordenes de usuario |
| 3 | Llamadas al sistema |
| 3 | Funciones de biblioteca C |
| 4 | Dispositivos y ficheros especiales |
| 5 | Formatos de fichero |
| 6 | Juegos |
| 7 | Convenciones y misceláneas |
| 8 | Ordenes de administración y mantenimiento |

Las páginas están organizadas en apartados; algunos de los más comunes son los siguientes:

| **Secciones** | **Contenidos** |
|---|---|
| NOMBRE | Resumen en una línea |
| SINOPSIS | Descripción de uso |
| DESCRIPCION | Discusión sobre lo que la orden o la función hace |
| VALOR DEVUELTO | Posibles valores de retorno |
| ERRORES | Resumen de valores de errno y condiciones de error |
| FICHEROS | Lista de ficheros del sistema que la orden o función utiliza |
| VEASE TAMBIEN | Lista de ordenes o funciones relacionadas con la presente |
| ENTORNO | Lista de variables de entorno relevantes |
| OBSERVACIONES | Información sobre uso no habitual o peculiaridades de la implementación |

Dos formas habituales de uso de la orden man son:

**man # nombre** , que consulta `nombre` en la sección indicada por el argumento #
(Ej., `man 2 write`)

**man -k nombre** , que muestra las entradas del manual que pueden tener relación con `nombre`
(Ej., `man -k editor`)

Otra forma de acceder a la información de manual es recurriendo a Google en un navegador Web, pidiendo que busque "`man orden`"

# 6 Biblioteca estándar de C

El lenguaje C no tiene soporte directo para Entada/Salida (escritura y lectura de ficheros, o dispositivos como la consola), gestión del Heap, etc. Este soporte se proporciona a través de una biblioteca de funciones denominada biblioteca (o librería) estándar de C. La implementación de esta biblioteca es dependiente del sistema, sin embargo su API es un estándar ANSI/ISO, lo que facilita la portabilidad de los programas escritos en C entre sistemas. Una descripción completa de la biblioteca queda fuera del alcance de este documento. En esta sección veremos sólo con cierto detalle las funciones básicas de gestión del Heap, el soporte para E/S y algunas funciones de gestión de errores. Para ampliar esta documentación se recomienda consultar un manual de C.

## 6.1 Gestión del Heap

En el mapa de memoria de un proceso, suele reservarse una zona de memoria conocida como Heap. Esta zona de memoria se utiliza para que el programador pueda ir solicitando trozos de distinto tamaño dinámicamente, que suele utilizar para alojar variables creadas en tiempo de ejecución. Es decir, en lugar de reservar espacio para una variable global en tiempo de compilación, o reservarla en la pila como variable local de alguna función, el programador solicita al sistema la memoria que necesita para alojar la variable, y el sistema le reserva un trozo del heap.

En C la gestión del Heap se realiza principalmente a través de dos funciones: **malloc**, para solicitar memoria, y **free**, para liberarla. Las cabeceras de estas funciones se declaran en el fichero *stdlib.h*, y son:

- ```
  void* malloc( size_t size );
  ```

  La función **malloc** toma como parámetro el número de bytes que se desea reservar, y devuelve un puntero con la dirección de memoria del bloque reservado, o **NULL** en caso de que no pudiese reservar esta memoria (por ejemplo, que no quede espacio en el heap).

- ```
  void free( void* ptr );
  ```

  La función **free** toma una dirección previamente devuelta por **malloc** y la vuelve a añadir como memoria libre/disponible al heap. Es importante que no se intente liberar con free memoria a partir de una dirección no devuelta por malloc, esto provocaría excepción, haciendo que el programa finalice abruptamente con error.

Existen otras funciones para la gestión del Heap, como **calloc**, **realloc**, **valloc** y **reallocf**. Se recomienda al alumno consultar sus páginas de manual.

## 6.2 Funciones de E/S estándar

Las principales funciones para escribir sobre el terminal o leer lo que introduce el usuario desde el teclado son: **printf** y **scanf**. Las cabeceras de estas funciones se declaran en el fichero **stdio.h**, y son:

- ```
  int printf(const char * restrict format, ...);
  ```

  La función **printf** permite mostrar por la salida estándar el valor de cualquier variable de tipo nativo (**int**, **char**, etc), con un formato especificado por una cadena de caracteres. La cadena de caracteres de formato puede incluir cero o más directivas, caracteres normales distintos de % que son escritos tal cual en el terminal, y especificadores de conversión. Cada especificación de conversión comienza con el carácter %, seguido de uno o más caracteres que describen la conversión. Su función es obtener un texto que describa el contenido de una variable nativa, que será insertado en esa misma posición en el flujo de salida (terminal).
  ```

Por ejemplo, supongamos que tenemos una variable entera *n*, y queremos mostrar por terminal la cadena de texto: "n = *valor_de_n*". Usaríamos la función **printf** de la siguiente manera:

```
printf("n = %d", n );
```

También podemos añadir a la cadena de formato secuencias de escape, que comienzan por el carácter `'\'`, seguido de otro carácter. Estas secuencias sirven para incluir, por ejemplo, saltos de línea (`\n`).

El formato concreto de todos los especificadores de conversión y las secuencias de escape puede encontrarse consultando la página de manual de **printf** o consultando cualquier manual de C.

- ```
  int scanf(const char *restrict format, ...);
  ```

  La función **scanf** sirve para leer de la entrada estándar. Al igual que **printf** toma una cadena de caracteres para especificar el formato de lo que se espera leer de la entrada estándar, que incluye especificadores de conversión. Cada especificador de conversión se corresponde con una variable que debe ser pasada a **scanf** por referencia (pasamos la dirección de la variable), de forma que al hacer la conversión de su valor en texto a su valor numérico el resultado se almacenará en dicha variable.

  Siguiendo con el ejemplo, si esperamos leer una cadena como la del caso anterior, y asignar el valor a la variable n, escribiríamos:

  ```
  scanf("n = %d", &n );
  ```

  La función devuelve el número de asignaciones realizadas (conversiones). Se recomienda al alumno consultar su página de manual para completar esta descripción.

## 6.3 Funciones de E/S sobre ficheros

La biblioteca estándar de C ofrece una serie de funciones que permiten el acceso a ficheros. Las cabeceras de estas funciones se declaran en el fichero **stdio.h**. En esta sección vamos a describir sólo algunas de las funciones más importantes, que son:

- ```
  FILE * fopen(const char *restrict filename, const char *restrict mode);
  ```

  La función **fopen** abre el fichero de nombre indicado por la cadena de caracteres que se pasa como primer argumento. El nombre puede ser una ruta absoluta o relativa. El segundo argumento de la función es una cadena de caracteres que indicará el modo en que debe abrirse el fichero:

  **"r"** Abre el fichero en lectura, apuntando el indicador de posición al comienzo del fichero.

  **"r+"** Abre el fichero en lectura y escritura, apuntando el indicador de posición al comienzo del fichero.

  **"w"** Trunca a tamaño cero el fichero si existe y si no existe lo crea. El fichero se abre en escritura apuntando el indicador de posición al comienzo del fichero.

  **"w+"** Trunca a tamaño cero el fichero si existe y si no existe lo crea. El fichero se abre en lectura y escritura, apuntando el indicador de posición al comienzo del fichero.

  **"a"** Abre el fichero en escritura. El fichero se crea si no existe. El indicador de posición se hace apuntar al final del fichero, de forma que las sucesivas escrituras añadan contenido al fichero.

  **"a+"** Abre el fichero en lectura y escritura. El fichero se crea si no existe. El indicador de posición se hace apuntar al final del fichero, de forma que las sucesivas escrituras añadan contenido al fichero.

  Si tiene éxito, la función devuelve la dirección de una estructura `FILE` que se usa como descriptor del fichero, y deberá pasarse al resto de funciones para trabajar con dicho fichero. Si se produce algún error, **fopen** devolverá `NULL`.

- `int fclose(FILE *stream);`

  La función **fclose** cierra un fichero abierto. Toma como argumento la dirección del descriptor (estructura `FILE`).

  Si tiene éxito, la función devuelve 0. En caso de error, devolverá `EOF` y asignará a la variable global errno un valor que describa el motivo del error.

- ```
  size_t fread(void *restrict ptr, size_t size, size_t nitems,
          FILE *restrict stream);
  ```

  La función **fread** permite leer `nitems` elementos de `size` bytes del fichero asociado al descriptor cuya dirección se pasa como último argumento. Los bytes leídos se escribirán en memoria a partir de la dirección que se pasa como primer argumento. Es responsabilidad del programador asegurarse de que hay espacio suficiente en dicho buffer.

  Si tiene éxito, **fread** devuelve el número de elementos (*items*) leídos correctamente. En caso de error, o si se alcanza el final de fichero, el valor devuelto será menor que el número de elementos que se intentó leer (tercer parámetro en la llamada). La función avanzará el indicador de posición tantos bytes como los bytes leídos.

- ```
  size_t fwrite(const void *restrict ptr, size_t size, size_t nitems,
          FILE *restrict stream);
  ```

  La función **fwrite** permite escribir `nitems` elementos de `size` bytes en el fichero asociado al descriptor cuya dirección se pasa como último argumento, leyendo los valores a escribir a partir de la dirección que se pasa como primer argumento.

  Si tiene éxito, **fwrite** devuelve el número de elementos (*items*) escritos. En caso de error, o si se alcanza el final de fichero, el valor devuelto será menor que el número de elementos que se intentó escribir (tercer parámetro en la llamada). La función avanzará el indicador de posición tantos bytes como los bytes escritos.

- `int fseek(FILE *stream, long offset, int whence);`

  La función **fseek** permite desplazar el indicador de posición del fichero asociado al descriptor cuya dirección se pasa como primer argumento. La nueva posición, medida en bytes, se obtiene sumando `offset` bytes a la posición indicada por el tercer argumento `whence`. Éste último puede tomar tres posibles valores:

  **SEEK_SET** comienzo del fichero

  **SEEK_CUR** posición actual

  **SEEK_END** final del fichero

  En caso de éxito, **fseek** devuelve 0. En caso de error, la función devuelve -1 y asigna a la variable global `errno` un valor que describe el motivo del error.

- `int putc(int c, FILE *stream);`

  La función **putc** escribe el valor entero `c` convertido a `unsigned char` en el fichero asociado al descriptor cuya dirección se pasa como segundo argumento. Si tiene éxito devuelve el carácter escrito. En caso de error devuelve `EOF`.

- `int getc(FILE *stream);`

  La función **getc** lee un carácter (1 byte) del fichero asociado al descriptor cuya dirección se pasa como argumento. Si tiene éxito devuelve el carácter leído como `unsigned char` transformado a entero. En caso de error devuelve `EOF`.

- `int feof(FILE *stream);`

  La función **feof** comprueba si se ha activado el indicador de final de fichero en el descriptor cuya dirección se pasa como argumento, devolviendo un valor no nulo si está activo. Este indicador se puede borrar explícitamente invocando la función `clearerr()` o como efecto lateral del uso de `fseek()`.

- `char* fgets(char *str, int size, FILE *stream);`

  La función **fgets** lee como mucho *size-1* caracteres del fichero cuyo descriptor se pasa como parámetro (*stream*), y los almacena en el buffer apuntado por `str`. La lectura finaliza cuando se llega a fin de fichero o se procesa un salto de línea. En caso de que se detecte un salto de línea, éste se almacena en el buffer, que en cualquier caso incluye el caracter terminador al final.

- `int fprintf(FILE  *stream, const char *format, ...)`

  Se trata de una variante de la función **printf** donde el mensaje con formato se imprime en el fichero cuyo descriptor se pasa como primer parámetro (*stream*).

## 6.4 Comprobación de errores

Como hemos visto en las secciones anteriores, muchas funciones de la biblioteca estándar de C asignan un valor determinado a la variable global `errno` para indicar la causa por la que han finalizado con error. La biblioteca ofrece también una serie de funciones que permiten interpretar estos valores y comunicarlos al usuario de forma más amigable. Una de las funciones más utilizadas para ello es:

- `void perror(const char *s);`

  La función **perror** muestra por pantalla una cadena de caracteres terminada en final de línea, con un mensaje que describe el error indicado en ese momento por la variable `errno`. Además, acepta como argumento la dirección de una cadena de caracteres. Si éste argumento no es `NULL`, dicha cadena se pone delante del mensaje descriptivo, separando ambos por el carácter '`:`'.