# 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, como Eclipse. En el laboratorio está instalado Eclipse, 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; y segundo, 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 son las siguientes:

1. **Editor**: para preparar los ficheros fuente en C. Se recomienda uno de los siguientes
   - **gedit**: editor gráfico con reconocimiento de sintaxis C
   - **kwrite**: otro editor gráfico
   - **kate**: otro editor gráfico
   - **nedit**: otro editor gráfico
   - **vim**: editor desde terminal
   - **emacs**: otro editor desde terminal
2. **Compilador**: para generar código objeto a partir del programa fuente en C o C++
   - **gcc** (lenguaje C)
   - **g++** (lenguaje C++)
3. **Gestión de Bibliotecas**
   - **ar**: creación y mantenimiento de bibliotecas estáticas
   - **gcc**: creación y mantenimiento de bibliotecas dinámicas
   - **ldd**: lista las bibliotecas dinámicas de las que depende un ejecutable
   - **nm**: lista la tabla de símbolos de un fichero objeto o ejecutable
4. **Automatización de proyectos**
   - **make**: gestiona las relaciones entre ficheros fuentes y bibliotecas que gobiernan los pasos necesarios para construir un proyecto
5. **Depurador**: permite hacer el seguimiento de la ejecución de un programa, paso a paso o mediante puntos de ruptura
   - **gdb**: depurador con interfaz de caracteres
   - **ddd**: depurador con interfaz gráfica, que utiliza `gdb` como base de operación
6. **Otras utilidades misceláneas**
   - **strace**: traza las llamadas al sistema invocadas por un programa durante su ejecución
   - **ltrace**: traza las funciones de biblioteca invocadas por un programa durante su ejecución
   - **objdump**: extrae información de un fichero objeto en formato ELF
   - **readelf**: extrae información de un fichero objeto en formato ELF

- **size**: informa sobre las necesidades de memoria anotadas en un fichero ejecutable
- **cflow**: informa sobre las relaciones entre funciones deducidas de los ficheros fuente de un programa
- **cdecl**: ayuda a interpretar el significado de una declaración en lenguaje C
- **pmap**: muestra la ocupación de memoria de un programa en ejecución

## 2 Primeros pasos

Lo mínimo que necesitaremos para empezar a trabajar es arrancar el puesto de ordenador con el sistema operativo LINUX (en las instalaciones de laboratorio de la FDI se utiliza la distribución Debian de LINUX instalada de forma nativa o una máquina virtual de Debian 7 que debemos arrancar desde Windows o Linux) y abrir una ventana de terminal. De esta manera entramos a relacionarnos con 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 material adicional 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 interfaz API del Sistema objetivo de la asignatura.

El alumno debe elegir un editor de los propuestos en la "introducción" o algún otro de su preferencia; supondremos que el elegido es gedit. Desde el terminal tecleamos
$ gedit greetings.c &
y tecleamos el siguiente programa (también podemos acceder a los editores a través del menú gráfico *gnome*):

```
#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:

```
$ 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. 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, 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 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: incompatible implicit declaration of
built-in function 'printf' [enabled by default]
$ ls -l a.out archi.*
-rwxr-xr-x 1 501 dialout 6769 Sep  7 10:41 a.out
-rw-r--r-- 1 501 dialout   71 Sep  7 10:38 archi.c
-rw-r--r-- 1 501 dialout   17 Dec  1  2014 archi.h
-rw-r--r-- 1 501 dialout  132 Sep  7 10:41 archi.i
-rw-r--r-- 1 501 dialout 1488 Sep  7 10:41 archi.o
-rw-r--r-- 1 501 dialout  452 Sep  7 10:41 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 "<command-line>"
# 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.i
# 1 "archi.c"
# 1 "<command-line>"
# 1 "archi.c"
# 1 "archi.h" 1
# 2 "archi.c" 2

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

```
}
$ cat archi.s
    .file   "archi.c"
    .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
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
    .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 executable, x86-64, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.26,
BuildID[sha1]=0x714b1d029926c1d48631903bc3c89b139b05983a, 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.

```
$ gcc -c archi.c
archi.c: In function 'main':
archi.c:5:2: warning: incompatible implicit declaration
of built-in function 'printf' [enabled by default]
```

Con la opción -c obtenemos el fichero objeto archi.o; es decir, prescinde de la operación de enlazado. La etapa final de enlace para producir el ejecutable la hacemos a continuación; el enlazador, invocado independientemente, se llama ld.

```
$ ld archi.o -o archi
ld: warning: cannot find entry symbol _start;
defaulting to 00000000004000b0
archi.o: In function `main':
archi.c:(.text+0x14): undefined reference to `printf'
```

Aparecen dos mensajes. Primero, un aviso nos informa que falta un símbolo _start que representa el punto de inicio de la ejecución del programa; y segundo, aparece un error debido a que hemos utilizado la función de biblioteca printf sin informar al enlazador dónde está localizada (en el archivo libc.so.6 del directorio /lib/x86_64-linux-gnu). Resolvemos primero el segundo error con una u otra de las siguientes órdenes, equivalentes:

```
$ ld archi.o -o archi -lc
ld: warning: cannot find entry symbol _start;
defaulting to 0000000000400270
$ ld  archi.o -o archi /lib/x86_64-linux-gnu/libc.so.6
ld: warning: cannot find entry symbol _start;
defaulting to 0000000000400270
```

Olvidamos de momento el aviso e intentamos ejecutar el programa de nombre archi (opción -o).

```
$ ./archi
-bash: ./archi: No such file or directory
```

El *shell* (bash) se queja indicando que no encuentra el fichero ejecutable, aunque éste existe en realidad. No es un problema del *shell*; lo que ocurre es que los servicios de carga y ejecución de ejecutables para ficheros ELF los aporta ld-linux-x86-64.so.2 y esta información hay que proporcionársela al enlazador así:

```
$ ld archi.o -o archi -lc -dynamic-linker /lib64/ld-linux-x86-64.so.2
ld: warning: cannot find entry symbol _start;
defaulting to 0000000000400280
```

Cuando se ejecuta un programa de C, no es main la primera entrada de código a la que se invoca. Primero se ejecuta un código de arranque que a su vez invoca a main. La primera línea ejecutable dentro de un programa está etiquetada como _start, por convenio. Podemos forzar a que el programa empiece directamente en main, indicándoselo expresamente al enlazador, ld como sigue:

```
$ ld archi.o -o archi -lc -e main -dynamic-linker /lib64/ld-linux-x86-64.so.2
$ ./archi
Hello ...100
Segmentation fault
```

El enlazado ya no dio error, y el programa arranca correctamente pero la ejecución finaliza con un fallo de direccionamiento de memoria. El error se debe a que, cuando el programa termina, debe devolver control al sistema; de ello se encarga la llamada exit() que no hemos empleado en nuestro código. Podemos incluir la línea exit(1) al final del código de archi.c para solucionar este problema; y de paso incluir también las cabeceras stdio.h y stdlib.h que contienen las definiciones de los prototipos de las funciones printf() y exit(), respectivamente — así silenciamos avisos del compilador.

```
$ cat archi_n.c
//Add new lines to archi.c
#include <stdio.h>
#include <stdlib.h>
#include "archi.h"
```

```
int main(void)
{
        printf("Hello ...%d\n", VAR);
        exit(0);
}
```

Volvemos a compilar con gcc -c y a enlazar como en el ejemplo anterior usando el nuevo fichero objeto creado, archi_n.o. Una vez hecho esto, ejecutamos el nuevo programa y mostramos en pantalla el valor devuelto por el programa al *shell* a través del comando exit:

```
$ ./archi_n; echo $?
Hello ...100
0
```

Por fin, todo funcionó correctamente. Sin embargo esto no es correcto. Al quitar el código de inicialización, el sistema no está correctamente preparado, porque se asume que antes de entrar en la función main se ejecuta un código de preparación proporcionado con el compilador, así como un código de terminación. Por ejemplo con el siguiente código:

```
$ cat archi_n2.c
//Add new lines to archi.c
#include <stdio.h>
#include <stdlib.h>
#include "archi.h"

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

  printf("Hello ...%d\n", VAR);

  if( argc >= 1 )
    for( i=0; i < argc; i++)
        printf("Argument #%d: %s\n",i,argv[i]);

  exit(0);
}
```

una compilación como la anterior no da error, pero en la ejecución se produce una violación de segmento. En realidad, nunca se hace el enlazado invocando directamente a ld, sino que se utiliza gcc como frontend, y podemos pedirle que nos muestre lo que hace para el enlazado con el flag -v:

```
$ gcc -c archi_n.c
$ gcc -v -o archi_n archi_n.o
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/4.7/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Debian 4.7.2-5'
--with-bugurl=file:///usr/share/doc/gcc-4.7/README.Bugs --enable-languages=c,c++,
go,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-4.7 --enable-shared
--enable-linker-build-id  --with-system-zlib --libexecdir=/usr/lib
--without-included-gettext --enable-threads=posix
```

```
--with-gxx-include-dir=/usr/include/c++/4.7 --libdir=/usr/lib --enable-nls --with-sysr
--enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu
--enable-plugin --enable-objc-gc --with-arch-32=i586 --with-tune=generic --enable-chec
--build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 4.7.2 (Debian 4.7.2-5)
COMPILER_PATH=/usr/lib/gcc/x86_64-linux-gnu/4.7/:/usr/lib/gcc/x86_64-linux-gnu/4.7/:
/usr/lib/gcc/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/4.7/:/usr/lib/gcc/x86_64-
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/4.7/:
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../x86_64-linux-gnu/:
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../../lib/:/lib/x86_64-linux-gnu/:
/lib/../lib/:/usr/lib/x86_64-linux-gnu/:/usr/lib/../lib/:
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-v' '-o' 'archi_n' '-mtune=generic' '-march=x86-64'
/usr/lib/gcc/x86_64-linux-gnu/4.7/collect2 --sysroot=/ --build-id --no-add-needed
--eh-frame-hdr -m elf_x86_64 --hash-style=both -dynamic-linker /lib64/ld-linux-x86-64.
-o archi_n /usr/lib/gcc/x86_64-linux-gnu/4.7/../../../x86_64-linux-gnu/crt1.o
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/4.7/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/4.7
-L/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../x86_64-linux-gnu
-L/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../
-L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib
-L/usr/lib/gcc/x86_64-linux-gnu/4.7/../../.. archi_n.o -lgcc --as-needed
-lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
/usr/lib/gcc/x86_64-linux-gnu/4.7/crtend.o
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../x86_64-linux-gnu/crtn.o
```

Como vemos, utilizar gcc es mucho más fácil, tenemos que pasar muchas menos opciones. Además enlaza de forma automática junto con nuestro programa una serie de ficheros objeto: crt1.o, crti.o, crtbegin.o, crtend.o y crtn.o; y enlaza (dos veces) con las librerías libgcc y libgcc_s.

### 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, generalmente llamado *makefile* o *Makefile*, que le indica los pasos que debe seguir para compilar el proyecto. La sintaxis de este fichero permite expresar los pasos de compilación de forma sencilla y compacta, es sencillo añadir nuevos ficheros, objetivos de compilación, bibliotecas de enlace, etc. Una vez confeccionado el *makefile* repetir la compilación es tan sencillo como ejecutar make desde el terminal.

La estructura de un *makefile* se compone principalmente de tres cosas:

1. **Comentarios**. Los desarrolladores pueden facilitar la comprensión de los `makefiles` mediante la inclusión de comentarios. Todo lo que esté escrito desde el carácter # hasta el final de la línea será ignorado por `make`. Las líneas que comiencen por el carácter # serán tomadas a todos los efectos como líneas en blanco.

2. **Definiciones de variables**. Una variable es un identificador que será sustituido por una cadena de texto, de forma que podemos utilizar el identificador y make lo sustituirá por el texto de su definición.
3. **Reglas**. Una regla indica los pasos a seguir para construir un objetivo.

Los siguientes apartados describen con algo de detalle la sintaxis de las definiciones de variables y las reglas, así como la invocación del comando make. Para obtener información adicional se recomienda al alumno que consulte la documentación oficial de GNU Make.

### Variables

Es muy habitual que los *makefiles* utilicen variables para facilitar su portabilidad a diferentes plataformas y entornos. La forma de definir una variable es muy sencilla, basta con indicar el nombre de la variable (típicamente en mayúsculas) y su valor, de la siguiente forma:

```
CC = gcc -Wall -pthread
```

Cuando queramos acceder al contenido de esa variable, lo haremos así:

```
$(CC) partial_sum1.c -o sum1
```

Es necesario tener en cuenta que la expansión de variables puede dar lugar a problemas de expansiones recursivas infinitas, por lo que a veces se emplea esta sintaxis:

```
CC := gcc
CC := $(CC) -Wall
```

Empleando := en lugar de = evitamos la expansión recursiva y por lo tanto todos los problemas que pudiera acarrear.

Además de las variables definidas en el propio `Makefile`, es posible hacer uso de las variables de entorno, accesibles desde el intérprete de comandos. Esto puede dar pie a formulaciones de este estilo:

```
SRC = $(HOME)/src
sum1:
  gcc $(SRC)/*.c -o sum1
```

Un tipo especial de variables lo constituyen las variables automáticas, aquellas que se evalúan en cada regla. Estas variables recuerdan a las variables usadas en los scripts de `bash`. Los más importantes son:

- **$@ :** Se sustituye por el nombre del objetivo de la presente regla.
- **$*** : Se sustituye por la raíz de un nombre de fichero.
- **$<** : Se sustituye por la primera dependencia de la presente regla.
- **$^~**: Se sustituye por una lista separada por espacios de cada una de las dependencias de la presente regla.
- **$? :** Se sustituye por una lista separada por espacios de cada una de las dependencias de la presente regla que sean más nuevas que el objetivo de la regla.
- **$(@D):** Se sustituye por la parte correspondiente al subdirectorio de la ruta del fichero correspondiente a un objetivo que se encuentre en un subdirectorio.
- **$(@F):** Se sustituye por la parte correspondiente al nombre del fichero de la ruta del fichero correspondiente a un objetivo que se encuentre en un subdirectorio.

### Reglas

El formato de una regla es el siguiente:

```
objetivo: prerrequisitos
          comandos
```

Un *objetivo* es generalmente un fichero a crear, normalmente un ejecutable, una biblioteca o un fichero objeto. Los *prerrequisitos* son una serie de ficheros (que pueden ser objetivos de otras reglas) que deben estar presentes y actualizados para que make pueda crear el *objetivo*. Para construir el *objetivo* make ejecutará los *comandos* indicados en el cuerpo de la regla.

Make procesa una regla de la siguiente manera:

1. Primero comprueba los *prerrequisitos*. Cada *prerrequisito* es el nombre de un fichero que debe estar actualizado.
    - Si hay alguna regla para construir el *prerrequisito*, make procesará dicha regla (recursivamente) para actualizar el fichero.
    - Si no hay ninguna regla para crear el *prerrequesito*, pero hay un fichero con ese nombre, el *prerrequisito* queda cumplido.
    - Si no hay ninguna regla para crear el *prerrequisito* y el fichero no existe, make no puede completar la compilación y termina con error.
2. Make determina si debe aplicar la regla o no.
    - Si no existe ningún fichero con el nombre del *objetivo*, o existe pero es más antiguo que alguno de los *prerrequisitos*, entonces la regla debe aplicarse para actualizar el *objetivo*.
    - Si por el contrario el fichero existe y no es más antiguo que ninguno de los *prerrequisitos*, make no necesita aplicar la regla.
3. Si hay que aplicar la regla, make ejecutará los *comandos* escritos en el cuerpo de la regla. Es muy importante que los comandos estén separados por un tabulador del comienzo de línea. Algunos editores cambian los tabuladores por 8 espacios en blanco, y esto hace que los `makefiles` generados así no funcionen.

Un ejemplo sencillo de *makefile* con tres reglas podría ser el siguiente:

```
partial_sum1.o: partial_sum1.c
    gcc -Wall -g -c partial_sum1.c -o partial_sum1.o

partial_sum2.o: partial_sum2.c
    gcc -Wall -g -c partial_sum2.c -o partial_sum2.o

sum : partial_sum1.o partial_sum2.o
    gcc sum -o partial_sum1.o partial_sum2.o
```

En este ejemplo la regla principal es la que nos dice como crear el objetivo sum, que tiene como prerrequisitos partial_sum1.o y partial_sum2.o. Cada uno de estos tiene su propia regla, que indica cómo construirlos a partir de los ficheros fuente partial_sum1.c y partial_sum2.c respectivamente. Al procesar las reglas, y asumiendo que ninguno de los ficheros objetos está actualizado, make ejecutaría las órdenes:

```
gcc -Wall -g -c partial_sum1.c -o partial_sum1.o
gcc -Wall -g -c partial_sum2.c -o partial_sum2.o
gcc sum -o partial_sum1.o partial_sum2.o
```

El orden en que aparecen las reglas en el *makefile* no tiene importancia, salvo para determinar la predeterminada, es decir, el objetivo que `make` construirá si no se le especifica ninguno en la línea de órdenes. Esta regla predeterminada es la primera que aparece en el `makefile` que no empiece por un punto. Por lo tanto, se escribirá el `makefile` de

forma que la primera regla que se ponga sea la encargada de compilar el programa entero, o todos los programas que se describan. Es costumbre llamar a tal regla all.

### Reglas virtuales

Es relativamente habitual que además de las reglas normales, los ficheros `Makefile` puedan contener reglas virtuales, que no generen un fichero en concreto, sino que sirvan para realizar una determinada acción dentro de nuestro proyecto software. Normalmente estas reglas suelen tener un objetivo, pero ninguna dependencia. El ejemplo más típico de este tipo de reglas es la regla clean que incluyen casi la totalidad de `Makefiles`, utilizada para "limpiar" de ficheros ejecutables y ficheros objeto los directorios que haga falta, con el propósito de rehacer todo la próxima vez que se llame a `make`:

```
clean:
  rm -f sum *.o
```

Esto provocaría que cuando alguien ejecutase make clean, el comando asociado se ejecutase y borrase el fichero Suma y todos los ficheros objeto (la opción -f hace que clean se ejecute sin pedir confirmación de borrado, generar mensajes de diagnóstico o retornar con un error en caso de que los ficheros a borrar no existan). Sin embargo, como ya hemos dicho, este tipo de reglas no suelen tener dependencias, por lo que si existiese un fichero que se llamase clean dentro del directorio del `Makefile`, make consideraría que ese objetivo ya está realizado, y no ejecutaría los comandos asociados:

```
$ touch clean
$ make clean
make: 'clean' is up to date.
```

Para evitar este extraño efecto, podemos hacer uso de un objetivo especial de make, .PHONY. Todas las dependencias que incluyamos en este objetivo obviarán la presencia de un fichero que coincida con su nombre, y se ejecutarán los comandos correspondientes. Así, si nuestro anterior `Makefile` hubiese añadido la siguiente línea:

```
.PHONY : clean
```

habría evitado el anterior problema de manera limpia y sencilla. Si además queremos evitar que errores en la ejecución de un comando paren la ejecución del `Makefile`, incluiremos un guión antes de los mismos:

```
clean:
  -rm sum *.o

$ make clean
rm sum
rm: cannot remove «sum»: No such file or directory
rm: cannot remove «*.o»: No such file or directory
make: [clean] Error 1 (no effect)
```

### Reglas implícitas

No todos los objetivos de un `Makefile` tienen por qué tener una lista de comandos asociados para poder realizarse. En ocasiones se definen reglas que sólo indican las dependencias necesarias, y es el propio make quien decide cómo se lograrán cada uno de los objetivos. Veámoslo con un ejemplo:

```
sum1 : partial_sum1.o
partial_sum1.o : partial_sum1.c
```

Con un `Makefile` como este, make verá que para generar Suma1 es preciso generar previamente sumaparcial1.o y para generar sumaparcial1.o no existen comandos que lo puedan realizar, por lo tanto, make presupone que para generar

un fichero objeto basta con compilar su fuente, y para generar el ejecutable final, basta con enlazar el fichero objeto. Así pues, implícitamente ejecuta las siguientes reglas:

```
cc -c partial_sum1.c -o partial_sum1.o
cc partial_sum1.o -o sum1
```

Generando el ejecutable, mediante llamadas al compilador estándar.

### Reglas patrón

Las reglas implícitas que acabamos de ver, tienen su razón de ser debido a una serie de "reglas patrón" que implícitamente se especifican en los `Makefiles`. Nosotros podemos redefinir esas reglas, e incluso inventar reglas patrón nuevas. He aquí un ejemplo de cómo redefinir la regla implícita anteriormente comentada:

```
%.o : %.c
  $(CC) $(CFLAGS) $< -o $@
```

Es decir, para todo objetivo que sea un .o y que tenga como dependencia un .c, ejecutaremos una llamada al compilador de C ($(CC)) con los modificadores que estén definidos en ese momento ($(CFLAGS)), compilando la primera dependencia de la regla ($<, el fichero .c) para generar el propio objetivo ($@, el fichero .o).

### Invocando al comando make

Cuando nosotros invocamos al comando make desde la línea de comandos, lo primero que se busca es 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.
```

Existen además varias maneras de llamar al comando make con el objeto de hacer una traza o debug del `Makefile`. Las opciones -d, -n, y -W están expresamente indicadas para ello. Algunos de los argumentos más habituales empleados con la utilidad `make`, son los mostrados en la siguiente tabla:

| ORDEN | EXPLICACIÓN |
|---|---|
| `make -f fichero` | El fichero de reglas es `fichero` en vez de `Makefile` o `makefile`, que son los nombres por defecto |
| `make -C directorio` Cambiar al directorio y aplicar `make` al `makefile` que allí exista. Permite aplicar `make` recursivamente | |
| `make -n` | Mostrar las órdenes que se mandarían ejecutar, pero sin ejecutarlas realmente. Es una ayuda para depuración |
| `make -d` | Muestra información de depuración además del procesamiento normal. Esta información dice qué ficheros están siendo considerados para ser rehechos, qué tiempos de ficheros están siendo comparados y con qué resultados, qué ficheros necesitan realmente ser rehechos, qué reglas implícitas están siendo tenidas en cuenta y cuáles se están aplicando: o sea, todo lo interesante sobre cómo `make` decide las cosas que tiene que hacer |

| ORDEN | EXPLICACIÓN |
|---|---|
| `make -W fichero` | Pretende que el objetivo *fichero* acaba de ser modificado. Cuando se emplea con la opción -n, esto nos enseña lo que pasaría si fuéramos a modificar ese fichero. Sin -n, es casi lo mismo que ejecutar la orden *touch* en el fichero dado antes de dar la orden `make`, salvo en que el tiempo de modificación se cambia solamente en la imaginación de *fichero*. |
| `make objetivo` | Comenzar activando el objetivo especificado en vez del primero que se encuentre en `makefile` |
| `make var=valor` | Incluir la declaración de la variable `var` especificada; tiene preferencia sobre una posible declaración interna en `makefile` |

# 4 Depuración

La utilidad de depuración en ejecución de programas C en 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. `Eclipse` o `KDevelop`).Vamos a ver la utilización de `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 material de la 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`: (La figura [fig:ddd] muestra la ventana obtenida al ejecutar `ddd badsort &`)

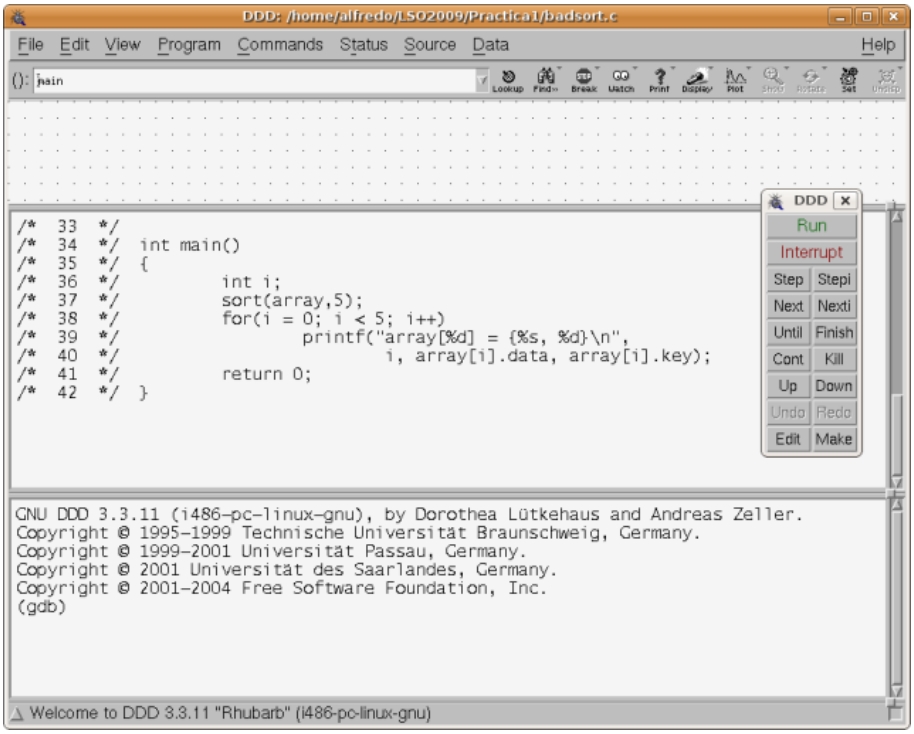


Figure 1: Ejecución de badsort bajo el control de ddd[fig:ddd]

```
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:

```
(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, a 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   0x0000000000040053e 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.

### 4.1 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 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`.

### 4.2 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.

### 4.3 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 |

| Secciones | Contenidos |
|---|---|
| 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:

- `{basicstyle="\ttfamily\small"}` `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).

- `{basicstyle="\ttfamily\small"}` `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

  Por ejemplo, supongamos que tenemos una variable entera *n*, y queremos mostrar po

  ```
  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

  Siguiendo con el ejemplo, si esperamos leer una cadena como la del caso anterior,

  ```
  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`.

- `{basicstyle="\ttfamily\small"}` `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.

- `{basicstyle="\ttfamily\small"}` `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.

- `{basicstyle="\ttfamily\small"}` `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.

- `{basicstyle="\ttfamily\small"}` `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.

- `{basicstyle="\ttfamily\small"}` `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`.

- `{basicstyle="\ttfamily\small"}` `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`.

- `{basicstyle="\ttfamily\small"}` `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()`.

### 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:

- `{basicstyle="\ttfamily\small"}` `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 '`:`'.

## 7 Bibliografía


- Marquez. *UNIX Programación avanzada* RA-MA. 3edición, 2004. Apéndices A y B
- N.Matthew, R.Stones, *Beginning LINUX Programming* Wrox, 4th edition, 2008.
- M.T.Jones. *GNU/LINUX Aplication Programming*. Thompson, 2004.
- M.Mitchell, *Advanced Linux Programming* New Riders Pub., 2001.
  (libro on-line: http://www.advancedlinuxprogramming.com/)
- *http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html*
- *http://es.wikipedia.org/wiki/Make*