

Software en formato fuente

Eugenia Damonte, Ariel Fideleff y Martín Goñi

Índice

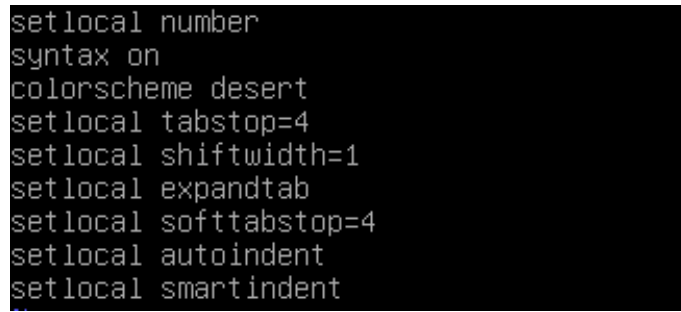
1. Configuración previa	1
2. Uso básico de gcc	2

1. Configuración previa

Antes de comenzar a resolver los ejercicios configuramos `vim` para editar archivos en C. Para hacer esto abrimos el archivo `~/.vimrc` (en todo caso de no existir, hay que crearlo) que es el archivo de configuración de `vim`. Estaba vacío por lo que le le añadimos dos líneas: `set nocp` y `filetype plugin on`. Lo que hace el primer comando es desactivar el modo de compatibilidad. Éste hace que algunas de las funciones de `vim` sean deshabilitadas o modificadas para que se comporte de manera similar a `vi`, el antecesor de `vim`. La segunda permite utilizar el plugin `filetype`.

Luego para asegurarnos de tener todos los paquetes de `vim` utilizamos el comando `sudo apt-get install vim-gui-common vim-runtime`. El primer paquete tuvo que instalarse demorando varios minutos por la velocidad de descarga abismal de los repositorios. El segundo, por el otro lado ya estaba instalado en nuestro caso.

Finalmente creamos el archivo de configuración para los archivos con extensión `.c`, llamado `c.vim`. Para poder crearlo primero tuvimos que crear la carpeta `~/.vim/ftplugin`, que es donde se ponen los archivos de configuración. Luego abrimos el mismo con `vim` y escribimos las configuraciones que queríamos usar.



```
setlocal number
syntax on
colorscheme desert
setlocal tabstop=4
setlocal shiftwidth=1
setlocal expandtab
setlocal softtabstop=4
setlocal autoindent
setlocal smartindent
```

Figura 1.1: Las configuraciones para los archivos `.c`

2. Uso básico de gcc

Antes de comenzar con el proyecto en sí, decidimos asegurarnos de que `gcc` funcionaba correctamente y que sabíamos usarlo. Para hacer esto copiamos el programa de ejemplo, `circulo.c`, que se encuentra en el apunte provisto.

```
1 #define PI 3.1416
2
3 main() {
4     float area, radio;
5
6     radio = 10;
7     area = PI * (radio * radio);
8     printf("Circulo.\n");
9     printf("%s%f\n\n", "Área de círculo de radio 10: ", area);
10 }
```

Figura 2.1: El programa de ejemplo `circulo.c`

2.1. Compilación directa

Una vez copiado el programa realizamos una compilación directa para asegurarnos de que el programa funcionase correctamente. Para hacer esto usamos el comando `gcc -o circulo circulo.c`. Lo que hace el argumento `-o` es permitirnos especificar el nombre del archivo de salida, pues si no, el archivo se nombra por defecto `a.out`.

```
martin@DebianPC:~/C$ gcc -o circulo circulo.c
martin@DebianPC:~/C$ ./circulo
Circulo.
Area de círculo de radio 10: 314.160004
martin@DebianPC:~/C$ rm circulo
```

Figura 2.2: Muestra del funcionamiento de `circulo.c`

2.2. Compilación compleja

Habiendo comprobado que `gcc` funcionaba correctamente decidimos intentar compilar el mismo archivo, `circulo.c`, de manera compleja. Es decir, haciendo cada uno de los pasos que realiza el compilador a la hora de transformar un archivo en C en un programa ejecutable, manualmente uno por uno.

2.2.1. Preprocesamiento

El preprocesado o preprocesamiento es la primera etapa de modificación del código fuente. Sirve para que, en la fase de compilación, que es la siguiente, el compilador pueda leer correctamente el código.

El trabajo del preprocesador consiste en llevar a cabo las instrucciones dadas por las *directivas* dirigidas al mismo (que, en el caso de C y C++, son las que comienzan con un numeral, como `#define`, `#include`, `#ifdef` y `#error`, entre otros).

En el archivo `circulo.c` podemos encontrar la directiva `#define PI 3.1416`, que justamente define una constante de nombre `PI` y valor `3.1416`. Esta constante es llamada en la función `main` de manera que, en vez de escribir `3.1416`, escribimos simplemente `PI`.

En la figura 2.3, se observa el código preprocesado que obtuvimos como salida del comando `gcc -E circulo.c` (también se puede usar `cpp circulo.c`, que hace referencia directamente al preprocesador). Si prestamos atención, la directiva antes mencionada no figura en esta salida. Además, en la línea que en el código fuente decía `area = PI * (radio * radio)`, ahora dice `area = 3.1416 * (radio * radio)`.

En resumen, lo que hizo el preprocesador fue tomar esa definición que le indicamos en la directiva y colocó el valor de la constante en las partes del código en donde se hacía referencia a ella.

```
# 1 "circulo.c"
# 1 "<command-line>"
# 1 "circulo.c"

main() {
    float area, radio;

    radio = 10;
    area = 3.1416 * (radio * radio);
    printf("Circulo.\n");
    printf("%s%f\n\n", "Area de circulo de radio 10: ", area);
}
```

Figura 2.3: Resultado del preprocesado de `circulo.c`

2.2.2. Compilación

La compilación es el proceso donde se transforma el código antes pre-procesado (en nuestro caso en C), a assembler propio del procesador de la computadora (en español, *lenguaje ensamblador*). Para hacer esto usamos el comando `gcc -S circulo.c`. Notar que directamente nos referimos al archivo con el código fuente `circulo.c`, pues el argumento `-S` ya de por medio realiza el preprocesado antes explicado. Finalmente verificamos que haya funcionado el comando, mostrando las primeras líneas del archivo `circulo.s`, que es donde `gcc` almacena el compilado.

```
martin@DebianPC:~/C$ gcc -S circulo.c
martin@DebianPC:~/C$ head circulo.s
.file "circulo.c"
.section .rodata
.LC2:
.string "Circulo."
.LC3:
.string "Area de circulo de radio 10: "
.LC4:
.string "%s%f\n\n"
.text
.globl main
```

Figura 2.4: Primeras 10 líneas del resultado de la compilación

2.2.3. Ensamblado

Una vez realizado el compilado, procedemos a ensamblar el archivo. Es decir, transformar el archivo de assembler a código objeto, un archivo binario en lenguaje máquina. Hicimos esto con el comando `as -o circulo.o circulo.s`. Luego verificamos que haya funcionado revisando qué tipo de archivo era `circulo.o` haciendo uso del comando `file`.

Aclarar que otra forma por la cual podríamos haber obtenido el archivo en cuestión sería el comando `gcc -c circulo.c`, pero como se puede ver, toma directamente desde el código fuente, ya que además realiza todas las etapas previas explicadas.

```
martin@DebianPC:~/C$ as -o circulo.o circulo.s
martin@DebianPC:~/C$ file circulo.o
circulo.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

Figura 2.5: Ejecución del ensamblado y detalles del archivo generado

Al estar el archivo conseguido en lenguaje máquina, no es texto en sí que podamos ver con facilidad. En todo caso, podemos hacer uso de un comando como lo es `objdump -d circulo.o`, que interpreta y permite ver las instrucciones a la computadora contenidas en el archivo. Hay que aclarar que las posiciones de memoria a las que se refiere probablemente no sean correctas, ya que éstas deben ser relacionadas en el próximo paso, el enlazado, con las librerías externas utilizadas.

2.2.4. Enlazado

Finalmente enlazamos el archivo. El enlazado es el proceso mediante el cual se vincula y se incorpora al programa, las librerías que requiere para poder funcionar. Estas librerías están compuestas de código con funciones que uno utiliza en dicho programa. Por ejemplo, en `circulo.c` usamos funciones como `printf`, el cual hace referencia a la librería `stdio.h` (típicamente se definiría al comienzo del programa de la forma `#include<stdio.h>` pero, al ser comúnmente utilizado, es incorporado automáticamente por el compilador).

En este paso es donde nos encontramos con problemas. El comando que se da en el apunte no funciona. Al usarlo da varios errores indicando que las librerías usadas como argumentos no existen, así como también con algunas de las opciones del comando.

```
martin@DebianPC:~/C$ ld -o circulo /usr/lib/gcc-lib/i386-linux/2.95.2/collect2 -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -d
circulo /usr/lib/crti.o /usr/lib/crti.o /usr/lib/gcc-lib/i386-linux/2.95.2/crtbegin.o -L/usr/lib/gcc-lib/i386-linux/2.95.2 circ
ulo.o -lgcc -lc -lgcc /usr/lib/gcc-lib/i386-linux/2.95.2/crtend.o /usr/lib/crtn.o
ld: cannot find /usr/lib/gcc-lib/i386-linux/2.95.2/collect2: No such file or directory
ld: cannot find /usr/lib/crti.o: No such file or directory
ld: cannot find /usr/lib/crti.o: No such file or directory
ld: cannot find /usr/lib/gcc-lib/i386-linux/2.95.2/crtbegin.o: No such file or directory
ld: cannot find -lgcc
```

Figura 2.6: El primer intento de usar `ld`, siguiendo el apunte

Dado todos los errores que habían, decidimos borrar todas las opciones innecesarias y probar nuevamente. Al hacer esto, los errores anteriores desaparecieron a cambio de uno nuevo. Éste decía “cannot find entry symbol `_start`”, el cual se traduce como “no se encuentra el símbolo de entrada `_start`”. Luego de algo de investigar descubrimos que este error se debe a que el verdadero punto de entrada¹ de un programa es `_start` y no `main`, siendo que `_start` simplemente redirige a él. Para solucionar esto usamos el argumento `--entry main` para especificar la función `main` como el punto de entrada del programa.

Una vez hechos estos cambios la función no daba mas errores y el programa parecía estar listo para usar. A la hora de ejecutarlo, sin embargo, éste no era reconocido como un programa ejecutable. Para asegurarnos de haber hecho todo correctamente revisamos que el archivo existiese así como también sus permisos, siendo éstos correctos.

```
martin@DebianPC:~/C$ ld -o circulo circulo.o -lc
ld: warning: cannot find entry symbol _start; defaulting to 000000000080481e0
martin@DebianPC:~/C$ ld -o circulo circulo.o -lc --entry main
martin@DebianPC:~/C$ ./circulo
-bash: ./circulo: No such file or directory
martin@DebianPC:~/C$ ls -l circulo
-rwxr-xr-x 1 martin martin 2245 Jul 29 16:20 circulo
```

Figura 2.7: El segundo intento de usar ld

Dado que no podíamos ejecutar el programa decidimos intentar volver a añadir algunas de las opciones que no causaban ores. Primero volvimos a añadir las opciones `-m elf_i386` y `--dynamic-linker /lib/ld-linux.so.2`. La primera opción define el objetivo del compilador². La segunda define la ubicación del enlazador dinámico³ a usar.

Luego de hacer estos cambios logramos ejecutar el programa, que parecía funcionar correctamente. Sin embargo, al final de éste tuvimos el error **Segmentation fault**. Para tratar de averiguar de donde venía el error decidimos debuggear el programa utilizando `gdb`.

¹El punto de entrada de un programa es donde se ejecutan las primeras instrucciones y se pasa control al programa.

²El objetivo del compilador es lo que determina que tipo de código objeto debe producir la función.

³Un enlazador dinámico o *dynamic linker* es una forma de enlazar los archivos binarios que se necesitan para que el programa funcione. En este caso el código de las funciones se mantiene en la biblioteca y la hora de ejecutar el programa se cargan en memoria.


```

martin@DebianPC:~/C$ ld -o circulo -m elf_i386 --dynamic-linker /lib/ld-linux.so.2 circulo.o -lc --entry main
martin@DebianPC:~/C$ ./circulo
Circulo.
Area de círculo de radio 10: 314.160004
Segmentation fault

```

(a) Error al correr el programa

```

martin@DebianPC:~/C$ gdb circulo
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 "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/martin/C/circulo...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/martin/C/circulo
Circulo.
Area de círculo de radio 10: 314.160004

Program received signal SIGSEGV, Segmentation fault.
0x00000001 in ?? ()

```

(b) Nuestro intento de debuggear el programa


Figura 2.8: El tercer intento de usar ld

Cuando intentamos debuggear el programa nos encontramos con algo extraño, y es que `gdb` no sabía de qué línea provenía el error. Esto nos llevó a creer que provenía del enlazado del programa, y no del programa en sí. Luego de buscar más todavía, encontramos el problema, y es que nos faltaba incluir las librerías que requería el enlazador dinámico. Para entender por qué pasa esto hay que entender cómo funciona el comando.

Lo primero que hace el comando es especificar la ubicación del enlazador dinámico que requieren las demás librerías para acceder a las funciones dinámicas de C. Luego se incluyen otras tres librerías `/usr/lib/i386-linux-gnu/crt1.o`, `/usr/lib/i386-linux-gnu/crti.o` y `/usr/lib/i386-linux-gnu/crtn.o`. La primera es la librería que tiene referencias a los archivos que requiere el enlazador (`/lib/libc.so.6` y `/usr/lib/libc_nonshared.a`). Las otras dos se encargan de que existan `_init` y `_fini`, que son el código de inicialización y finalización. Algo importante de recordar es que la ubicación de las librerías puede cambiar dependiendo del sistema y la instalación específica. En nuestro caso, las encontramos buscando en `/usr/lib` y revisando todas las carpetas que parecían tener alguna relación.

Es importante notar la posición de las librerías, `crti.o` debe ir después de `crt1.o`. Esto es porque el segundo hace referencia al primero. Además

ambas deben ir antes del archivo que se está enlazando. Finalmente `crtn.o` va al final del comando, después de todos los demás argumentos.



```
martin@DebianPC:~/C$ ld -o circulo -m elf_i386 --dynamic-linker /lib/ld-linux.so.2 /usr/lib/i386-linux-gnu/crti.o /usr/lib/i386-linux-gnu/ctfn.o circulo.o -lc /usr/lib/i386-linux-gnu/crtn.o
martin@DebianPC:~/C$ ./circulo
Circulo.
Area de circulo de radio 10: 314.160004
martin@DebianPC:~/C$ _
```

Figura 2.9: El cuarto y último intento de usar `ld`

Luego de hacer todo esto, el programa finalmente funcionó y se ejecutó de manera correcta y sin errores. Habiendo terminado decidimos que ya teníamos el suficiente conocimiento para intentar compilar un programa usando `make`.