

Herramientas de desarrollo

Alejandro Furfaro

4 de abril de 2019

Temario

1 Lenguajes de programación

- Primeros conceptos
- Lenguaje Ensamblador
- Lenguajes de alto nivel

2 Primeros pasos en lenguaje C

- Primer ejemplo: Hola Mundo (poco original. . .)

3 Toolchain

- Hilando un poco mas fino
- De que se ocupa cada herramienta
- Avanzando un poco mas con las herramientas de desarrollo
- Necesidad de formato de objetos

4 Formato ELF

- Introducción

1 Lenguajes de programación

- Primeros conceptos
- Lenguaje Ensamblador
- Lenguajes de alto nivel

2 Primeros pasos en lenguaje C

3 Toolchain

4 Formato ELF

¿Que lenguaje hablan los microprocesadores?

- En los modelos originales las CPU fueron pensadas para tratar con valores que pueden tomar solo dos estados:
 - Verdadero - Falso
 - 1 - 0
 - Tensión +V - Tensión 0.
- Entonces un Microprocesador solo “habla” en binario.

El problema

A los seres humanos no nos resulta “natural” hablar este lenguaje. Si bien podemos hacerlo, nos es engorroso, y por otra parte es muy fácil cometer un error. Basta para ello permutar un 1 con un 0. Y, una vez cometido, es sumamente arduo de encontrar.

Programando en el lenguaje del Microprocesador

El listado de la izquierda es el original. El de la derecha es una copia y tiene un error ¿donde está?

01101011	11011111	01101100	01101011	11011111	01101100
01000110	01110111	10001010	01000110	01110111	10001010
11101010	10010011	01101011	11101010	10010011	01101011
10100100	11010101	00110100	10100100	11010101	00110100
01100001	00010000	01101010	01100001	00010000	01101010
00011110	10001010	01011010	00011110	10001010	01011010
11010111	11010011	10100101	11010111	11010011	10100101
10001001	10010111	10011000	10001001	10010111	10011000
10001101	10100101	01111001	10001101	10100101	01111001
11000010	10010110	01101011	11000110	10010110	01101011
10110011	00101001	01111111	10110011	00101001	01111111
00101001	00010100	01101101	00101001	00010100	01101101
01010110	10010100	01100101	01010110	10010100	01100101

Programando en el lenguaje del Microprocesador

Y ? ... ¿lo encontraste?

mmmm..... ¿estás seguro?

01101011	11011111	01101100	01101011	11011111	01101100
01000110	01110111	10001010	01000110	01110111	10001010
11101010	10010011	01101011	11101010	10010011	01101011
10100100	11010101	00110100	10100100	11010101	00110100
01100001	00010000	01101010	01100001	00010000	01101010
00011110	10001010	01011010	00011110	10001010	01011010
11010111	11010011	10100101	11010111	11010011	10100101
10001001	10010111	10011000	10001001	10010111	10011000
10001101	10100101	01111001	10001101	10100101	01111001
11000010	10010110	01101011	11000110	10010110	01101011
10110011	00101001	01111111	10110011	00101001	01111111
00101001	00010100	01101101	00101001	00010100	01101101
01010110	10010100	01100101	01010110	10010100	01100101

1 Lenguajes de programación

- Primeros conceptos
- **Lenguaje Ensamblador**
- Lenguajes de alto nivel

2 Primeros pasos en lenguaje C

3 Toolchain

4 Formato ELF

Necesitamos un lenguaje mas “humano”

```
section .data
    msg: DB 'Hola Mundo', 10
    largo EQU $ - msg
```

```
global main
```

```
extern printf
```

```
section .text
```

```
main:
```

```
    push rbp
    mov rbp, rsp
    mov rdi, msg
    mov eax, 1
    call printf
    mov eax, 0
    pop rbp
    ret
```


1º paso: Una sentencia = una instrucción

- Este es el lenguaje llamado Ensamblador, también conocido como “lenguaje de máquina”.
- Cada instrucción tiene un nombre alusivo a la operación que realiza (en inglés), y se lo representa por su abreviatura. Ej: **MOV**, por Move, **ADD** por Addition, etc.
- Cada sentencia en el programa corresponde a una y solo una instrucción de la CPU.
- Con ayuda de un programa llamado Ensamblador, se convierte este texto escrito de manera mas legible para un programador, a la secuencia de números binarios para que lo entienda el Microprocesador.
- Al texto original del programa escrito en lenguaje “humano” se lo conoce como ***código fuente***.

1 Lenguajes de programación

- Primeros conceptos
- Lenguaje Ensamblador
- Lenguajes de alto nivel

2 Primeros pasos en lenguaje C

3 Toolchain

4 Formato ELF

2º paso: Una sentencia = varias instrucciones

- Cada sentencia del programa se compone de varias instrucciones del procesador.
- Ventaja: permite escribir aplicaciones de mayor complejidad con menos texto.
- El programa se escribe en un archivo de texto plano, (igual que los programas en Assembler).
- Un programa llamado Compilador convierte este texto a números binarios, explotando cada sentencia en una o mas instrucciones del microprocesador.
- Al igual que el caso del programa escrito en Assembler, el texto escrito en lenguajes de alto nivel se denomina programa fuente.

1 Lenguajes de programación

2 Primeros pasos en lenguaje C

- Primer ejemplo: Hola Mundo (poco original...)

3 Toolchain

4 Formato ELF

El mismo programa anterior escrito en lenguaje C

```
/* Esta secuencia es para iniciar un comentario.  
El comentario puede ocupar cuantas lineas quieras  
Y al final .....  
Esta secuencia es para cerrar un comentario*/
```

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

```
int      main ()  
{  
    printf("Hola Mundo!!\n");  
    return 0;  
}
```

¿Que contiene este simple programa?

- 1 En primer lugar lo mas fácil. Todo texto encerrado entre `/*` y `*/`, es tratado como un comentario. Significa que el compilador no va a generar código alguno con este texto.
- 2 Un programa C, se compone de dos elementos lógicos básicos: ***funciones*** y ***variables***.
- 3 Las ***funciones*** contienen sentencias que definen las diferentes operaciones que se ejecutan una a una,
- 4 Las ***variables*** contienen los datos que el programa mantiene almacenados, y que eventualmente modificará como consecuencia de su operación.
- 5 Las funciones pueden llevar el nombre que mejor nos parezca, pero hay una función “obligatoria”: **`main`**.
- 6 Un programa comienza su ejecución en el inicio de la función **`main`**.

¿Que contiene este simple programa?

- 7 **main** para organizar el trabajo llama a otras funciones que como veremos van componiendo las partes que solucionan el problema completo (esto es programación modular).
- 8 Las funciones invocadas por **main** pueden estar:
 - En el mismo archivo del programa,
 - En otro archivo fuente, que junto con el que contiene a **main** compone el proyecto de software,
 - O puede tratarse de funciones externas a nuestro programa que ya traducidas a números binarios (compiladas) y que están almacenadas en archivos que llamaremos bibliotecas de código.

¿Que contiene este simple programa?

9

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

#include es una directiva para el compilador. Le indica que debe incluir el archivo cabecera **<stdio.h>** que contiene las definiciones de las funciones de E/S standard que están almacenadas en la biblioteca **libc**.

Concepto Importante

¡stdio.h no contiene el código de la biblioteca! . Es un archivo **de texto** en el que solamente se declaran las funciones cuyos archivos objeto están almacenados en una biblioteca.

Su misión es que el compilador conozca la sintaxis correcta para invocar alas funciones allí declaradas.

La biblioteca de código está en otro archivo (también binario). El código fuente de las funciones que componen esta biblioteca, tampoco está en **stdio.h**. **No olvidar este concepto** .

¿Que contiene este simple programa?

- 10 Toda función puede recibir una lista de valores que se denominan **argumentos**.
- 11 En el caso de `main` , en esta aplicación simple no recibe argumentos. Mas adelante en el curso veremos que puede recibirlos y como tratarlos en tal caso.
- 12 Luego entre los caracteres `{` y `}` se encierran las sentencias que componen el cuerpo de la función.
- 13 En el caso de este sencillo ejemplo el cuerpo de `main` solo contiene las sentencias:

```
printf( "Hola Mundo!!\n" );  
return 0;
```

¿Que es printf?

- 14 Una función. Eso.
- 15 Tal como explicamos recibe un argumento, en este caso el puntero al área de memoria que almacena un texto: "Hola Mundo!!\n"
- 16 Lo que hace **printf** es imprimir en pantalla el texto cuyo puntero le pasamos como argumento.
- 17 `\n` es una secuencia de escape que utiliza el lenguaje C para representar el caracter Nueva Línea.
- 18 El comportamiento de nuestro programa será imprimir en pantalla en el renglón siguiente al comando que lo ejecute, el mensaje Hola Mundo!!, y luego saltar a la línea siguiente como si se pulsase la tecla <Enter>
- 19 El tipo de argumento es una cadena de caracteres en forma de constante, por eso va encerrada entre comillas dobles.

¿Donde está printf?

- 20 En nuestro archivo fuente, evidentemente no está.
- 21 De modo que solo cabe una posibilidad: La función es externa.
- 22 **printf** está contenida en una de las bibliotecas mas utilizadas en C: La libc. Los prototipos de las funciones de Input Output de esta biblioteca, estan en el archivo header stdio.h.
- 23 Comprobémoslo:

Tipear en la consola

```
\      grep printf /usr/include/stdio.h
```

- 24 Alguno de uds. estará preguntándose como se logra que el programa acceda al código de **printf** si ésta no es parte de programa sino que está afuera de él ¿verdad?
- 25 Quienes aun no se lo preguntaron... deberían hacerlo ;)

1 Lenguajes de programación

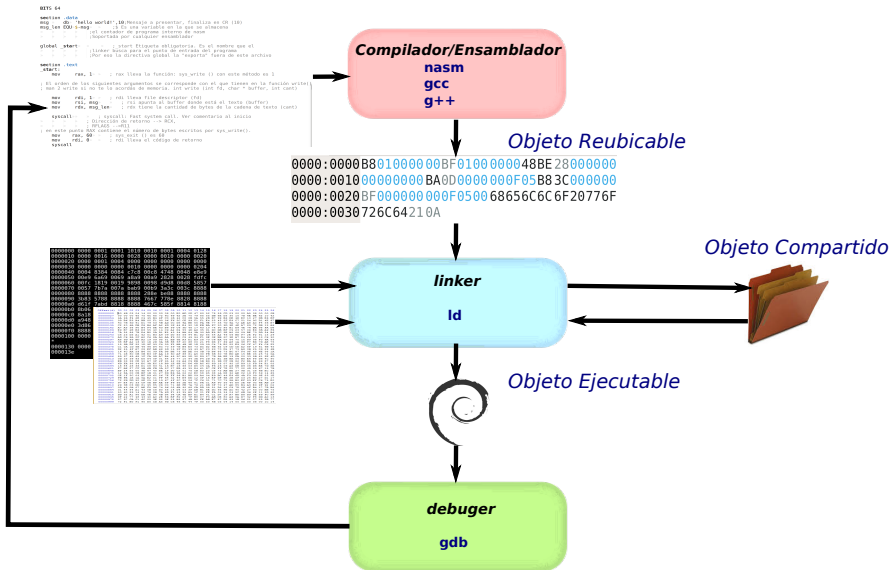
2 Primeros pasos en lenguaje C

3 Toolchain

- Hilando un poco mas fino
- De que se ocupa cada herramienta
- Avanzando un poco mas con las herramientas de desarrollo
- Necesidad de formato de objetos

4 Formato ELF

Toolchain



Archivos objeto

Tenemos tres clases de archivos objeto

Relocatable files Son archivos compilados que contienen código y datos aptos para combinarse con otros archivos objeto para construir otro archivo que puede ser **executable** o **shared object**.

Executable files Contiene un programa listo para ser ejecutado por el sistema operativo

Shared Object files Contienen código y datos adecuados para ser procesados en dos contextos:

- El Link editor lo combina con otros **shared objects** o **relocatables objects** para generar otro archivo objeto (**shared object** o un **executable**)
- El Dynamic linker lo procesa con un **executable** y eventualmente otros **shared objects** para construir la imagen de un proceso en memoria

1 Lenguajes de programación

2 Primeros pasos en lenguaje C

3 Toolchain

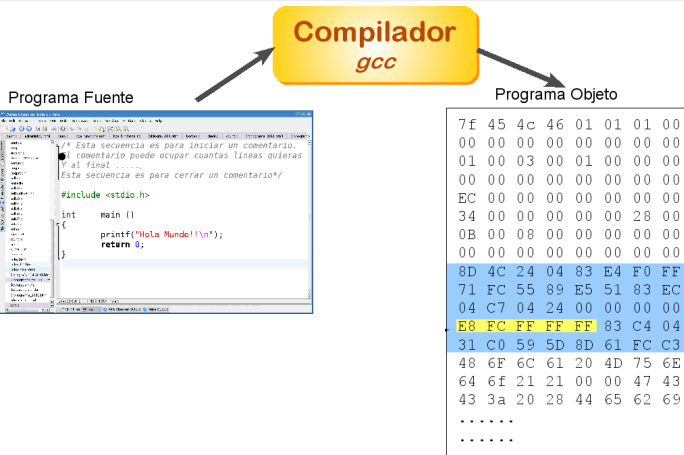
- Hilando un poco mas fino
- De que se ocupa cada herramienta
- Avanzando un poco mas con las herramientas de desarrollo
- Necesidad de formato de objetos

4 Formato ELF

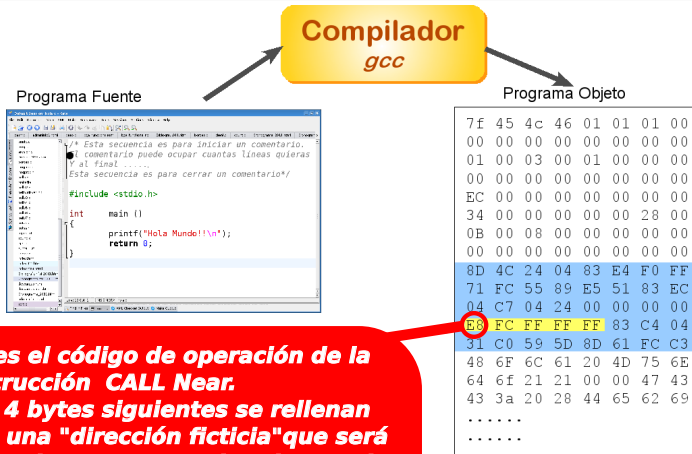
El compilador

- Es un programa capaz de analizar sintácticamente un archivo de texto que contiene un programa fuente.
- Si éste está escrito de manera correcta, respetando la semántica del lenguaje para el cual compila, genera un código binario adecuado para ser ejecutado por el Microprocesador que obra como CPU en el sistema.
- Además de analizar las operaciones reemplaza los nombres lógicos que adoptemos en nuestro programa para variables o funciones por las direcciones de memoria en donde se ubican las mismas.
- No puede resolver referencias a funciones exteriores al archivo fuente que analiza. Por ejemplo, no puede resolver por que valor numérico reemplazar a la etiqueta `printf`, ya que no tiene visibilidad de la misma. Habrá que esperar a la siguiente fase para resolver este tema.

Cuando se dejan referencias por resolver



Cuando se dejan referencias por resolver



E8 es el código de operación de la instrucción CALL Near. Los 4 bytes siguientes se rellenan con una "dirección ficticia" que será posteriormente completada por el linker o (como veremos...) por el propio Sistema Operativo al momento de cargar el programa en memoria para su ejecución.

El compilador

- Antes de hacer su trabajo, invoca a un programa denominado preprocesador, que se encarga de eliminar los comentarios, incluir otros archivos (la línea `#include <stdio.h>`, es reemplazada por contenido del archivo `stdio.h`), y reemplaza las macros (la sentencia para el preprocesador en este caso es `#define`).
- Si genera errores el programa está mal escrito y debe ser revisado.
- Si no genera errores solo significa que el programa está correctamente escrito. De allí a que funcione correctamente es otra cuestión. . .
- Una vez que compiló, su producto es un programa objeto. Este es un binario pero que aún no está listo para poderse ejecutar.
-

Para generar el programa objeto, tipear en la consola

```
gcc -c hola.c -ohola.o -Wall
```

El Linker

- Es un programa capaz de tomar el programa objeto generado recién por el compilador, enlazarlo (“linkearlo”) con otros programas objeto y con otras biblioteca de código y generar un programa ejecutable por el Sistema Operativo sobre el cual estamos desarrollando nuestro programa.
- Muchas cosas juntas ¿verdad?
- Enlazar significa:
 - Poner todos los bloques de código juntos y ordenar código y datos en secciones comunes para luego guardar ese conjunto en un único archivo ejecutable.
 - Una vez ordenado, resolver cada referencia a una variable o función que en la fase de compilación eran externas. En nuestro caso el linker resolverá la referencia a `printf`.
 - Identificar y marcar el punto de entrada del programa (la dirección que se le asignará a `main`).

El linker

- Parece poco relevante. Sin embargo es crucial esta fase de la generación de nuestro programa

Para generar el programa ejecutable podríamos, tipear en la consola

```
COLLECT.GCC_OPTIONS='-o' 'hola' '-v' '-mtune=generic' '-march=x86-64'
/usr/lib/gcc/x86_64-linux-gnu/7/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/7/liblto_plugin
.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper -plugin-opt=-fresolution=/
tmp/ccLkRp8F.res -plugin-opt=-pass-through=lgcc -plugin-opt=-pass-through=lgcc.s -
plugin-opt=-pass-through=lc -plugin-opt=-pass-through=lgcc -plugin-opt=-pass-through=
lgcc.s --sysroot=/ --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed
-dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -z now -z relro -o hola /usr/lib/gcc/
x86_64-linux-gnu/7/../../../../x86_64-linux-gnu/Scrt1.o /usr/lib/gcc/x86_64-linux-gnu
/7/../../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/7/crtbeginS.o -L/usr/
lib/gcc/x86_64-linux-gnu/7 -L/usr/lib/gcc/x86_64-linux-gnu/7/../../../../x86_64-linux-gnu
-L/usr/lib/gcc/x86_64-linux-gnu/7/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/./lib -
L/usr/lib/x86_64-linux-gnu -L/usr/lib/./lib -L/opt/intel/compilers_and_libraries_2018
.3.222/linux/ipp/lib/intel64 -L/opt/intel/compilers_and_libraries_2018.3.222/linux/
compiler/lib/intel64_lin -L/opt/intel/compilers_and_libraries_2018.3.222/linux/mkl/lib/
intel64_lin -L/opt/intel/compilers_and_libraries_2018.3.222/linux/tbb/lib/intel64/gcc4.7
-L/opt/intel/compilers_and_libraries_2018.3.222/linux/tbb/lib/intel64/gcc4.7 -L/opt/
intel/compilers_and_libraries_2018.3.222/linux/daal/lib/intel64_lin -L/opt/intel/
compilers_and_libraries_2018.3.222/linux/daal/./tbb/lib/intel64_lin/gcc4.4 -L/usr/lib/
gcc/x86_64-linux-gnu/7/../../../../hola.o -lgcc --push-state --as-needed -lgcc.s --pop-
state -lc -lgcc --push-state --as-needed -lgcc.s --pop-state /usr/lib/gcc/x86_64-linux-
gnu/7/crtendS.o /usr/lib/gcc/x86_64-linux-gnu/7/../../../../x86_64-linux-gnu/crtn.o
```

El linker

- Hay involucrados unos cuantos objetos como vemos que son relevantes:
 - `/usr/lib/x86_64-linux-gnu/Scrt1.o`
 - `/usr/lib/x86_64-linux-gnu/crti.o`
 - `/usr/lib/x86_64-linux-gnu/crtbeginS.o`
 - `/usr/lib/x86_64-linux-gnu/crtn.o`
- Y algún que otro componente adicional.
- Engorroso, imposible de memorizar, y sobre todo, sujeto a cuestiones internas del sistema.

El linker

- Por eso, gcc sabe llamar al linker y nos evita este engorroso trámite a nosotros

Para generar el programa ejecutable tipeamos en la consola

```
gcc -ohola hola.o -Wall
```

- Para saber como el gcc arma el llamado usamos la opción -v (verbose)

Tipear en la consola

```
gcc -ohola hola.o -v
```

Warning: Prestale atención a los warnings

- Los Warnings que arrojan tanto el Compilador como el linker, son como su nombre lo indica Advertencias.
- No impiden que el compilador genere el archivo con el objeto reubicable ni que el linker genere el archivo ejecutable.
- No por eso debemos ignorarlos.
- Por el contrario debe prestarse especial atención ' a los Warnings
- La experiencia indica que terminan transformándose en errores de lógica.
- La opción **-Wall** en ambas líneas de compilación y linkeo, indica a ambas herramientas que presenten Todos los Warnings (**Warning all**). Aun los mas insignificantes

Buenas Costumbres

Buena Práctica de Desarrollo

Siempre incluir la opción **-Wall** en las líneas de compilación y linkeo, trabajando sobre el código para eliminar **TODOS** los warnings.

Algo es 100 % seguro: ***El Warning se transforma en un bug mas tarde o mas temprano.***

Por lo tanto se deberá incluir en cada proyecto **-Wall** en las líneas de compilación y linkeo **SIEMPRE**.

1 Lenguajes de programación

2 Primeros pasos en lenguaje C

3 Toolchain

- Hilando un poco mas fino
- De que se ocupa cada herramienta
- Avanzando un poco mas con las herramientas de desarrollo
- Necesidad de formato de objetos

4 Formato ELF

Agreguemos alguna función de cálculo

```
/* Programa sqrt.c:
 * Su función es calcular la raíz cuadrada de un número
 * predefinido en su código y mostrar su resultado en
 * la pantalla del computador.
 * Para compilarlo: gcc -c sqrt.c -o sqrt.o
 * Para linkearlo: gcc sqrt.o -o sqrt -lm
 */

#include <stdio.h>
#include <math.h>

#define N 1234567890

int main ()
{
    double result;
    result = sqrt(N);
    printf ("La raíz cuadrada de %d es:%10.7f\n",N,result);
    return 0;
}
```

Linkeando con una Biblioteca

- Si observamos el comentario que encabeza el listado del programa del slide anterior, vemos que al linker se le provee una opción adicional: `-lm`
- `-l` sirve para especificar el nombre de una Biblioteca (l por library)
- `m` es el nombre de la biblioteca: `m` es math, cuyos prototipos, macros y constantes están definidos en `math.h` (entre ellos la función `sqrt`)
- Pregunta: ¿Porque no hubo que especificar la librería que contiene `printf` ?
- El compilador “conoce” la ubicación de las bibliotecas mas comunes para evitar que debamos especificar permanentemente librerías de uso casi tan común como la propia función `main`

¿Porque motivo tantos objetos?

Luego del compilador/ensamblador se dispone del binario necesario para que la CPU pueda utilizar los datos y ejecutar la secuencia de instrucciones...

- ¿es así?...veamos.

0000:0000B801000000BF0100000048BE28000000¿.....H ³ / ₄ (...
0000:001000000000BA0D000000F05B83C000000 ⁰<...
0000:0020BF00000000F050068656C6C6F20776F	¿.....hello wo
0000:0030726C64210A	rld!.

- En un binario crudo hay varias incertezas:
- ¿como sabe la CPU donde comienza el programa? (es decir, en donde está la primer instrucción)
- ¿Como puede determinar donde finaliza el programa y comienzan los datos?
- ¿Y si los datos están al comienzo del archivo?

1 Lenguajes de programación

2 Primeros pasos en lenguaje C

3 Toolchain

- Hilando un poco mas fino
- De que se ocupa cada herramienta
- Avanzando un poco mas con las herramientas de desarrollo
- Necesidad de formato de objetos

4 Formato ELF

El binario crudo no es suficiente

El formato binario es útil cuando nos proponemos desarrollar un kernel, o un firmware de arranque de un sistema, ya éstos se ejecutan en el arranque del sistema y por lo general obedecen a requerimientos particulares, como por ejemplo que la primer instrucción esté en una dirección de memoria bien específica.

Una vez que arrancó el computador, el sistema operativo ubicará los diferentes objetos en memoria en donde le resulte mas conveniente.

En este escenario el formato binario no tiene la información completa que se necesita para continuar adelante con la cadena de desarrollo.

Formatos de objetos

- Cada sistema operativo definen un formato propio para los diferentes objetos que soporta.
- En Linux se soportan algunos formatos que a esta altura podemos calificar como legacy: *Common Object File Format (coff)*, *Minix/Linux as86 Object Files*, *Relocatable Dynamic Object File Format (rdf)*.
- El formato mas actualizado y vigente es ELF (**elf16**, **elf32**, o **elf64** según la CPU que soporte Linux)

- 1 Lenguajes de programación
- 2 Primeros pasos en lenguaje C
- 3 Toolchain
- 4 Formato ELF**
 - **Introducción**
 - ELF Header
 - SECTIONS

¿Que es ELF?

- ELF es la abreviatura de **E**xecutable and **L**inkable **F**ormat.
- Define la estructura para archivos binarios, bibliotecas, y archivos core.
- Su especificación es formal. Esto permite al sistema operativo interpretar correctamente las características de la máquina en que se basa el computador.
- Los archivos ELF típicamente son salidas del compilador o del linker con lo cual ya contienen el binario del programa.
- Con las herramientas apropiadas podemos analizarlo y entender su estructura.

Formato de un archivo objeto

Linking View

ELF Header
Program Header Table <i>optional</i>
Section 1
...
Section <i>n</i>
...
...
Section Header Table

Execution View

ELF Header
Program Header Table
Segment 1
Segment 2
...
Section Header Table <i>optional</i>

Formato de un archivo ELF

Formato

- El Header contiene el roadmap de la organización del archivo.
- Las secciones (***sections***) contienen los bloques de información para la vista del linker, entre otras cosas:
 - Los bloques de código de las diferentes subrutinas y funciones
 - Los datos
 - Las tablas de símbolos
 - La información de relocación
- Los segmentos contienen los bloques para la vista de ejecución.
- La ***Program Header Table*** (si existiere), indica como crear la imagen del proceso en memoria. Los **relocatables objects** no la necesitan.
- La ***Section Header Table*** tiene información de las diferentes secciones que componen el archivo. Cada sección tiene una entrada en esta tabla.

Representación de datos

Se define un set de tipos de datos para los anchos de palabra para los diferentes datos de cada encabezado en diferentes arquitecturas.

Nombre	Tamaño	Alineación	Define
Elf32_Addr	4	4	Unsigned program direction
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

En código:

```
# include <stdint.h>
typedef uint16_t Elf32_Half;    // Unsigned half int
typedef uint32_t Elf32_Off;    // Unsigned offset
typedef uint32_t Elf32_Addr;    // Unsigned address
typedef uint32_t Elf32_Word;    // Unsigned int
typedef int32_t  Elf32_Sword;    // Signed int
```

- 1 Lenguajes de programación
- 2 Primeros pasos en lenguaje C
- 3 Toolchain
- 4 Formato ELF**
 - Introducción
 - ELF Header**
 - SECTIONS

ELF Header

```
# define ELF_NIDENT      16

typedef struct {
    unsigned char    e_ident[ELF_NIDENT];
    Elf32_Half       e_type;
    Elf32_Half       e_machine;
    Elf32_Word       e_version;
    Elf32_Addr       e_entry;
    Elf32_Off        e_phoff;
    Elf32_Off        e_shoff;
    Elf32_Word       e_flags;
    Elf32_Half       e_ehsize;
    Elf32_Half       e_phentsize;
    Elf32_Half       e_phnum;
    Elf32_Half       e_shentsize;
    Elf32_Half       e_shnum;
    Elf32_Half       e_shstrndx;
} Elf32_Ehdr;
```

Campos del ELF Header

```
unsigned char    e_ident[ELF_NIDENT];
```

Arreglo de 16 bytes encargado de definir al archivo como un **object file** y especificar sus características.

```
enum Elf_Ident {  
    EI_MAG0          = 0, // 0x7F  
    EI_MAG1          = 1, // 'E'  
    EI_MAG2          = 2, // 'L'  
    EI_MAG3          = 3, // 'F'  
    EI_CLASS         = 4, // Architecture (32/64)  
    EI_DATA          = 5, // Byte Order  
    EI_VERSION       = 6, // ELF Version  
    EI_OSABI         = 7, // OS Specific  
    EI_ABIVERSION    = 8, // OS Specific  
    EI_PAD           = 9  // Padding  
};
```


Campos del ELF Header

```
# define ELF_MAG0 0x7F // e_ident[EI_MAG0]
# define ELF_MAG1 'E' // e_ident[EI_MAG1]
# define ELF_MAG2 'L' // e_ident[EI_MAG2]
# define ELF_MAG3 'F' // e_ident[EI_MAG3]

// e_ident[EI_CLASS]
# define ELF_CLASS_NONE 0 // Arquitectura Invalida
# define ELF_CLASS_32 1 // Arquitectura de 32 bits
# define ELF_CLASS_64 2 // Arquitectura de 32 bits

// e_ident[EI_DATA]
# define ELF_DATA_2LSB 0 // Orden de datos Invalido
# define ELF_DATA_2LSB 1 // Little Endian
# define ELF_DATA_2LSB 2 // Big Endian

// e_ident[EI_VERSION]
# define ELF_VERSION_NONE 0 // Version invalida
# define ELF_VERSION_CURRENT 1 // Version actual
```

ELF Header

```
Elf32_Half      e_type;
```

Name	Value	Meaning
ET_NONE	0	No file type
ET_REL	1	Relocatable file
ET_EXEC	2	Executable file
ET_DYN	3	Shared object file
ET_CORE	4	Core file
ET_LOPROC	0xff00	Processor-specific
ET_HIPROC	0xffff	Processor-specific

ELF Header

```
Elf32_Half      e_machine;
```

Name	Value	Meaning
ET_NONE	0	No machine
EM_M32	1	AT&T WE 32100
EM_SPARC	2	SPARC
EM_386	3	Intel Architecture
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000 Big-Endian
EM_MIPS_RS4_BE	10	MIPS RS4000 Big-Endian
RESERVED	11-16	Reserved for future use

ELF Header

```
Elf32_Word    e_version;
```

Name	Value	Meaning
EV_NONE	0	Invalid version
EV_CURRENT	1	Current version

```
Elf32_Word    e_entry;
```

Si el objeto no tiene un punto de entrada asociado, este campo no tiene significado. Si posee punto de entrada (es el objeto obtenido a partir de un .c por ejemplo que tiene el main) este campo almacena la dirección virtual del punto de entrada del objeto

ELF Header

```
Elf32_Off e_phoff;
```

Contiene el offset en bytes de la **Program Header Table**. Si el archivo no tiene este tipo de tablas este offset es 0.

```
Elf32_Off e_shoff;
```

Contiene el offset en bytes de la **Section Header Table**. Si el archivo no tiene este tipo de tablas este offset es 0.

```
Elf32_Word e_flags;
```

Almacena flags específicos del procesador si es que el archivo los contiene.

```
Elf32_Half e_ehsize;
```

Mantiene el tamaño en bytes del header ELF

```
Elf32_Half e_phentsize;
```

Mantiene el tamaño en bytes de una entrada de la **Program Header Table**. (Las entradas de esta tabla tienen todas el mismo tamaño)

ELF Header

```
Elf32_Half      e_phnum;
```

Contiene la cantidad de entradas de la **Program Header Table**. Si no hay tal tabla este campo es 0. De este modo el producto entre `e_phnum` y `e_phentsize` da el tamaño en bytes de la **Program Header Table**.

```
Elf32_Half      e_shentsize;
```

Contiene el tamaño en bytes de una **Section Header**, es decir, de cada entrada en la **Section Header Table**. (Todas éstas entradas son del mismo tamaño)

```
Elf32_Half      e_shnum;
```

Contiene el número de entradas de la **Section Header Table**. De este modo el producto entre `e_shnum` y `e_shentsize` da el tamaño en bytes de esta Tablea. Si no llegase a existir la **Section Header Table** este valor es 0.

```
Elf32_Half      e_shstrndx;
```

Contiene el índice de la **Section Header Table** de la entrada asociada con la Tabla de string de nombres de sección. Si no hay Tabla de Strings este campo mantiene el valor `SHN_UNDEF`.

Ejemplo

Consideremos el siguiente código

```
int sumar(const int, int)

    const int a=2678;
    int b=7903;
    int r;

int main (void)
{
    r=sumar(a,b);
    return r;
}

int sumar (const int i, int j)
{
    return (i+j);
}
```

Compilar (solamente) y ver el encabezado

Para compilar y obtener un objeto se utilizan las siguientes opciones en el compilador

```
$ gcc -c -m32 -o suma.o suma.c
```

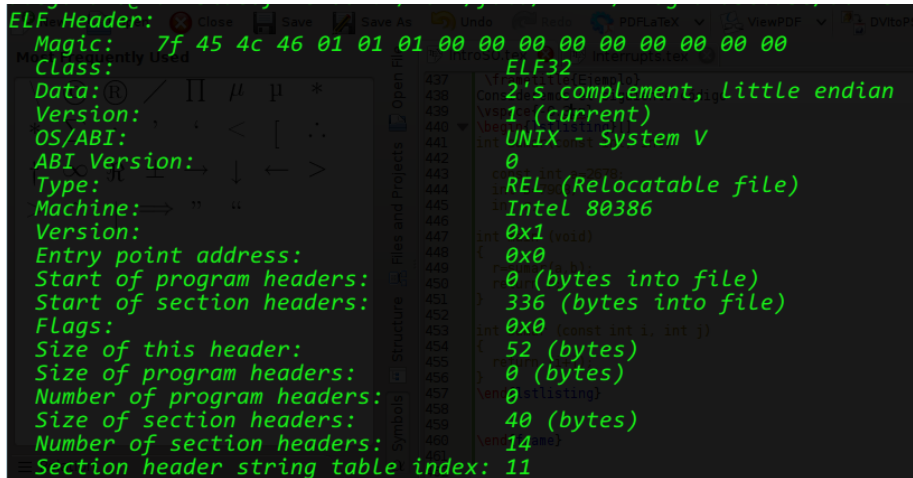
- c Indica al gcc que no debe invocar al linker. Solo debe compilar el fuente y generar el archivo objeto **relocatable**.
- m32 indica al gcc que genere código de 32 bits. No se espera que invoque a **collect2** para que arme la llamada al linker

Compilar (solamente) y ver el encabezado

Para visualizar el encabezado el comando es

```
$ readelf -h suma.o
```

que resulta en



```

ELF Header:
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX System V
ABI Version: 0
Type: REL (Relocatable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x0
Start of program headers: 0 (bytes into file)
Start of section headers: 336 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 0 (bytes)
Number of program headers: 0 (bytes)
Size of section headers: 40 (bytes)
Number of section headers: 14
Section header string table index: 11
  
```

Compilar (solamente) y ver el encabezado

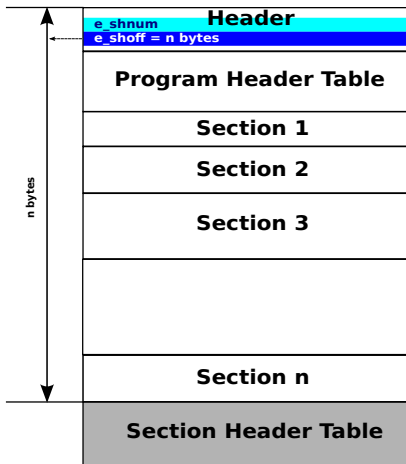
Lo que se ve en la pantalla es el contenido de la instancia de esta estructura correspondiente al archivo objeto **suma.o**

Header
Program Header Table
Section 1
Section 2
Section 3
Section n
Section Header Table

```
typedef struct {  
    unsigned char e_ident[ELF_NIDENT];  
    Elf32_Half e_type;  
    Elf32_Half e_machine;  
    Elf32_Word e_version;  
    Elf32_Addr e_entry;  
    Elf32_Off e_phoff;  
    Elf32_Off e_shoff;  
    Elf32_Word e_flags;  
    Elf32_Half e_ehsize;  
    Elf32_Half e_phentsize;  
    Elf32_Half e_phnum;  
    Elf32_Half e_shentsize;  
    Elf32_Half e_shnum;  
    Elf32_Half e_shstrndx;  
} Elf32_Ehdr;
```

- 1 Lenguajes de programación
- 2 Primeros pasos en lenguaje C
- 3 Toolchain
- 4 Formato ELF**
 - Introducción
 - ELF Header
 - SECTIONS**

Section Header Table



```

Sección 1
typedef struct {
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
} Elf32_Shdr;

Sección 2
typedef struct {
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
} Elf32_Shdr;

...

Sección e_shnum
typedef struct {
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
} Elf32_Shdr;

```

Section Header Table: Contenido

Conclusión La **Section Header Table** es un array de `e_shnun` elementos de la siguiente estructura:

```
typedef struct {  
    Elf32_Word      sh_name;  
    Elf32_Word      sh_type;  
    Elf32_Word      sh_flags;  
    Elf32_Addr      sh_addr;  
    Elf32_Off       sh_offset;  
    Elf32_Word      sh_size;  
    Elf32_Word      sh_link;  
    Elf32_Word      sh_info;  
    Elf32_Word      sh_addralign;  
    Elf32_Word      sh_entsize;  
} Elf32_Shdr;
```

Además en el Header ELF el parámetro `e_shentsize` contiene la cantidad de entradas (miembros) de esta tabla (estructura)

Contenido de una entrada de la tabla

- sh_name** Índice en la Tabla de Strings de Sección del Header de la Sección en donde se guardan los Nombres de cada sección (Ej: .text, .data, etc), como strings terminadas en '0'.
- sh_type** Caracteriza contenido y semántica de la sección.
- sh_flags** Cada sección tiene un grupo de flags de 1 bit que especifican determinados atributos
- sh_addr** Si la sección formará parte de la imagen de memoria del proceso, este miembro contiene la dirección de memoria en la que debe estar el primer byte de la sección. De otro modo es 0.
- sh_offset** Contiene la distancia en bytes desde el principio del archivo objeto, hasta el primer byte de la sección.

Contenido de una entrada de la tabla

- sh_size** Contiene el tamaño de la sección expresado en bytes. Si el tipo expresado por sh_type es SHT_NOBITS la sección no consume ni un solo byte en la imagen del proceso aunque sh_size sea diferente de cero
- sh_link** Mantiene un link al Índice al Header de la Tabla de Sección.
- sh_info** Contiene información adicional que depende del tipo de sección que se trate
- sh_addralign** En caso que la sección tenga requerimientos de alineación (por ejemplo una que contenga doble words que requiera alinear a doble word), los expresa en este campo. Almacena el módulo en que debe estar alineada (ej: 2 indica doublewords ya que es 2^2)
- sh_entsize** En caso en que la sección tenga entradas de tamaño fijo este campo expresa el tamaño de las entradas.

Tipos de sección

Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	see below
.hash	SHT_HASH	SHF_ALLOC
.line	SHT_PROGBITS	none
.note	SHT_NOTE	none
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	see below
.symtab	SHT_SYMTAB	see below
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

Visualizando las secciones de nuestro objeto

La opción -S del comando readelf, muestra las secciones

```

alejandro@DarkSideOfTheMoon:~/work/facu/TDIII/ProgramasClase/Linker-scripting$ readelf -S suma.o
There are 14 section headers, starting at offset 0x150:

Section Headers:
 [Nr] Name              Type             Addr             Off              Size              ES Flg Lk Inf AL
 [ 0] SECTIONS             NULL             00000000          000000           000000           00  0  0  0
 [ 1] .text                 PROGBITS         00000000          000034           00002c           00 AX  0  0  4
 [ 2] .rel.text            REL              00000000          000048c          000028           08 12  1  4
 [ 3] .data                 PROGBITS         00000000          000060           000004           00 WA  0  0  4
 [ 4] .bss                  NOBITS           00000000          000064           000000           00 WA  0  0  4
 [ 5] .rodata               PROGBITS         00000000          000064           000004           00 A  0  0  4
 [ 6] pepe                 PROGBITS         00000000          000068           00000d           00 AX  0  0  1
 [ 7] .comment              PROGBITS         00000000          000075           00001d           01 MS  0  0  1
 [ 8] .note.GNU-stack       PROGBITS         00000000          000092           000000           00 non 0  0  1
 [ 9] .eh_frame             PROGBITS         00000000          000094           000058           00 A  0  0  4
[10] .rel.eh_frame         REL              00000000          0004b4           000010           08 12  0  9  4
[11] .shstrtab             STRTAB           00000000          0000ec           000064           00 SHF 0 ALL 0  1
[12] .symtab               SYMTAB           00000000          000380           0000f0           10 13 10  4
[13] .strtab               STRTAB           00000000          000470           000019           00 non 0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

```

Puede probar con -t para ver detalles de las secciones, o directamente con -a para ver ... ¡todo!