Introducción al Lenguaje C

> Tutorial extraído del blog **C para Operativos** (autor: Matías García Isaia)

Video: C - Una charla minimalista

> Presentación del video: **C - Una charla minimalista**

Arrancando

Vamos a hacer un programa en C, por lo que empezamos con una función:

```c
int main(void) {
  return 0;
}
```

Este es (aproximadamente) el programa más chico que podamos hacer en C. `main()` es la función que se ejecuta al ejecutar un programa C. En este caso, el prototipo de la función es `int main(void)` : nuestro programa no recibirá parámetros (`void`), y devolverá a quién lo ejecute un entero signado (`int`).

Nuestro programa tiene una única instrucción: devolver (`return`) 0, un código de salida que, por convención, indica que el programa ejecutó correctamente.

Guardémoslo en un archivo `ok.c` y compilemos nuestro programa:

```sh
$ gcc ok.c -o ok
$ ls
ok.c ok
```

`gcc` es el compilador más usado de C. Parte de la GNU Compiler Collection, `gcc` es el compilador específico de C (el proyecto se llamaba GNU C Compiler, pero cambiaron el nombre por soportar también C++, Java y tantos otros lenguajes). Como casi todo comando en Linux, Unix y derivados, podemos leer su manual haciendo `man gcc`.

`gcc` recibe como parámetro (entre tantos otros) el archivo fuente a compilar (`ok.c`), y el parámetro `-o NOMBRE` indica qué nombre queremos darle al binario resultante (`ok`). De no indicarlo, `gcc` elije uno *hermoso*: `a.out`.

Ejecutemos:

```sh
$ ./ok
$ echo $?
0
```

Hay un 0, así que debemos estar *no-tan-mal*, como mínimo. ¿Qué pasó acá?

En UNIX, la forma de ejecutar un programa es escribiendo como orden la ruta completa al mismo y, luego, separados por espacios, todos sus parámetros. Por ejemplo:

```sh
$ /bin/ps --version
procps-ng version 3.3.3
```

Ejecutamos el programa `/bin/ps` con el parámetro `--version`. `ps` nos contesta la versión que tenemos instalada. En nuestro caso anterior, ejecutamos `./ok`.

`.` y `..` son dos enlaces especiales que hay en todo directorio: `.` enlaza al directorio actual (el propio directorio que contiene a `.`), y `..` enlaza al directorio padre del actual (osea, al directorio que contiene al directorio que contiene a `..`).

Entonces, al escribir `.` ya estamos referenciando toda la ruta al directorio actual.

Si estamos ubicados en `/home/utnso` , `.` y `/home/utnso` se refieren al mismo directorio. Agregándole `/ok` queda `./ok` , que equivale a `/home/utnso/ok` , la ruta completa a nuestro programa. Hell yeah, ruta completa => ¡ejecutamos el programa!

> "Che, pero... ¡No me mostró nada! ¿Dónde está mi 0?"

Bueno, sí, es cierto. No muestra nada porque no le pedimos que muestre nada: nuestro programa sólo devuelve un 0, y nuestra consola sólo ejecuta las instrucciones que le damos. Entonces, pidámosle que muestre el resultado: `echo $?` .

> "¡Que te recontra!"

Bueno, sí. **El amigo Bourne** había faltado a la clase de nombres descriptivos. `echo` es un comando de las consolas que imprime en pantalla[1] lo que sea que le pasemos por parámetro.

Por ejemplo, `echo Hola mundo` imprime `Hola mundo`. `bash` (el lenguaje que interpreta nuestra consola) posee variables, y para dereferenciarlas (leerlas) hay que anteponerle un `$` al nombre de la variable. Por ejemplo:

sh

```
$ nombre = "Mundo" # asigno "Mundo" a la variable nombre, creandola si no existe
$ echo $nombre # imprimo el contenido de la variable llamada nombre
Mundo
$ echo "Hola $nombre"
Hola Mundo
```

(`#` es el caracter de comentario)

En particular, nosotros le habíamos pedido mostrar una variable: `$?`. `?` es una variable manejada automáticamente por Bash. Cada vez que ejecuta una instrucción, Bash almacena en `?` el código de salida del programa ejecutado. Por eso, al pedirle que imprima la variable `?` (`echo $?`), Bash nos mostró el 0 que nuestro programa había devuelto.

Variables

Bien. Escribimos, compilamos y corrimos nuestro primer programa. Pero es como bastante aburrido, ¿no? Vamos a ponerle onda: declaremos una variable (¡iupi! (¿?)).

c

```
1  int main(void) {
2      int exit_status = 0;
3      return exit_status;
4  }
```

Guau. Me la jugué 😃 Anoche no dormí porque me quedé debuggeando un error en este programa (?).

¿Qué cambió? Bueno, entre las llaves hay dos instrucciones ahora. En principio, donde antes decía `return 0;`, ahora dice `return exit_status;`.

> "Así que seguramente `exit_status` sea una variable mágica de C, como `$?` en bash"

Pendorcho. `exit_status` es una variable, sí. O sea, es un identificador de un *cacho'e memoria*. Puedo guardar *cosas* ahí, y luego leerlas. Pero antes necesito declararla, para decir:

1. que existe; y
2. qué tipo de cosas va a manejar esa variable.

Y eso es lo que hicimos antes: `int exit_status = 0;`.

Para declarar una variable, especificamos su tipo de dato, seguido por su nombre. En nuestro caso, `int exit_status` crea una variable de tipo `int` llamada `exit_status`.

Declaraciones válidas son `int hola;` o `int hola, chau;`, por ejemplo: la primera declara una variable `hola` de tipo `int`, mientras que la segunda crea `hola` y `chau`, dos variables de tipo `int`, ambas totalmente independientes entre sí.

Pero con declarar la variable no alcanza: si queremos devolverla o leerla, primero tenemos que darle un valor ("inicializarla", para los amigos). En C, las asignaciones son del estilo `variable = expresion;`,

donde, en nuestro caso original, `expresion` es un triste `0` constante. Y ahí tenemos nuestra primera línea: `int exit_status = 0;` .

> "Che, y, entonces, si no es una variable mágica de C, ¿por qué se llama `exit_status` y no, por ejemplo, `a` , `bleh` o `code` ?"

Bueno, porque nosotros **sí** fuimos a la clase de nombres bonitos y representativos 😃 Y si esa variable representa nuestro estado de salida, así la llamaremos[2]. Podríamos haberla llamado `a` , `bleh` , `code` o `__a256723b` , pero preferimos reservarnos los nombres horribles para las PPT 😃

Buen, a ver qué hace este programa:

```sh
$ gcc ok.c -o ok
$ ./ok
$ echo $?
0
```

Compilamos y ejecutamos, y vemos que sigue sin mostrar nada. Hacemos el `echo` y vemos nuestro hermoso 0.

> "Che, para mí que éste nos está chamuyando y el 0 está hardcodeado por ahí"

OK, cambiémoslo:

```c
int main(void) {
    int exit_status = 1;
    return exit_status;
}
```

Y probemos:

```sh
$ gcc ok.c -o ok
$ ./ok
$ echo $?
1
```

¡Touché!

> "OK, ganaste. Ahora, si necesito recompilarpara cambiar el valor de una variable, muy variable no me parece. Y podría cambiar el 0 por un 1 en la primer versión de `ok.c` , y no tengo que andar haciendo tanta parafernalia. ¿Por qué se llaman *variables*?"

Buen, sí, justamente, porque podés cambiarles el valor durante una misma ejecución del programa. Así como inicializamos `exit_status` en 0 o en 1, podríamos después de esa incialización *asignarle* un nuevo valor. Desde que se ejecute esa instrucción en adelante, cada vez que se lea el contenido de la variable obtendremos el nuevo valor, como si nunca hubiera tenido un valor distinto:

```c
int main(void) {
    int exit_status = 0;
    exit_status = 1;
    return exit_status;
}
```

```sh
$ gcc ok.c -o ok
$ ./ok
$ echo $?
1
```

`=` es el operador de asignación. El resultado de evaluar lo que esté a su derecha (ya veremos alternativas, pero por ahora quedémonos con que los números evalúan a sí mismos) se almacena en el espacio de memoria referido a la izquierda.

Y, ¿qué pasó con el 0? Se perdió. El 0 sigue existiendo y valiendo 0, como siempre. Sólo que el contenido de la variable `exit_status` se sobreescribe con `1`: la asignación es *destructiva*.

Sigamos jugando con esto:

```c
int main(void) {
    int exit_status = 0;
    int a_number = 1;
    exit_status = a_number;
    a_number = 3;
    return exit_status;
}
```

```sh
$ gcc ok.c -o ok
$ ./ok
$ echo $?
1
```

> "¡Eh! ¿¿Qué onda?! Si `exit_status` es igual a `a_number`, y a `a_number` le asigno `3`, ¿por qué el estado de salida es `1`?"

Bueno, porque te olvidaste lo que dije de la asignación: en lo que está a la izquierda del `=` guardo el resultado de evaluar lo que está a *la otra izquierda* del mismo (comunmente conocida como "derecha").

Y nada más que eso: las variables no se ligan, ni quedan relacionadas, ni nada. Las variables se evalúan a su contenido del momento en que se ejecuta la instrucción, por lo que al hacer `exit_status = a_number;` estamos diciendo "en `exit_status` guardame lo que `a_number` valga en ese momento".

Como `a_number` venía valiendo 1, `exit_status` pasa a valer 1 también. Que después modifiquemos `a_number` es otra canción, y no tiene ninguna relación con esa asignación que ya se hizo: lo hecho, hecho está, y si al evaluar la variable ésta valía 1, los posibles valores que tenga después no importan, porque ya se realizó la asignación.

Bajando el nivel

> "Che, me acabás de decir que el 0 sigue existiendo por más que *lo pise* con otro valor. ¿El 0 es un objeto al que la VM le mantiene referencias y por eso no se lo lleva el Garbage Collector?"

What!? No way, papá. Por un cuatrimestre, las palabras "objeto" y "Garbage Collector" dejalas en la oficina, y VM = VirtualBox 😃

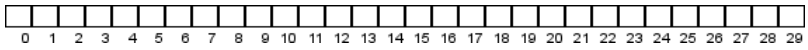
En C no hay objetos.

Perdón si fui duro, pero es necesario: en C no hay objetos. Y no hay GC.

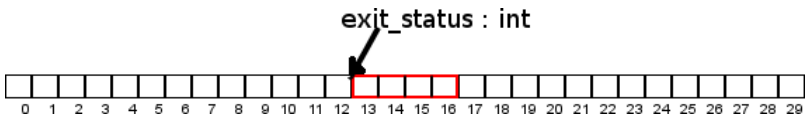
"Entonces, ¿qué es el 0?"

El 0 es una constante. Si recordás, en Arquitectura (bazinga) vimos que los números se representan en binario en la PC. Las variables de C son meras referencias a algún bloque de memoria, siendo el tipo de la variable el que anuncia qué tamaño tiene esa referencia.

Digamos, esto es una memoria de 30 bytes[3] (éramos tan pobres):



Esa es toda la memoria de esta computadora hipotética. Si en mi programa declaro una variable `int exit_status;`, podríamos pensarlo como que C hará algo así:



En algún lugar de la memoria (en este caso, a partir del byte 13), C reservó[4] unos bytes para nuestra variable.

> "¿Cuántos bytes reservó?"

Eso depende del tipo de dato que le declaramos a la variable. En el caso de un `int`, la variable va a ocupar 4 bytes[5].

Entonces, al hacer algo como `exit_status = 0;` (y sabiendo que `exit_status` es una variable entera de 4 bytes), el compilador sabe que tiene que hacer que los 4 bytes que están a partir del byte en que empieza `exit_status` valgan un 0.

En Arquitectura aprendimos que los enteros con signo se representan usando el **complemento a 2**, por lo que el 0 en 32 bits es simplemente `0x0000 0000` (4 bytes en 0).

Entonces, ¿dónde está ese 0? **Hardcodeado** en el binario. Compilar un programa es pedirle al compilador (`gcc` en nuestro caso) que traduzca todo ese C hermoso que escribimos en las instrucciones de Assembler que nuestro procesador tiene que ejecutar para que el código funcione con nuestro sistema operativo.

De todas esas, alguna va a ser algo como `mov eax, 0x00000000` [6]: **ese** es tu 0.

> "¿Y a mí cuál?"

Meh, no mucho. Sólo quería dejar en claro cómo funciona esto: C es una mini abstracción de la programación en Assembler, por lo que no está tan lejos. No tenés que ser un capo de Assembler para programar C[7] (no necesitás saber Assembler, siquiera), pero tenés que tener un claro entendimiento de cómo funciona la computadora a bajo nivel para entender qué es lo que está haciendo tu código.

---

1. En realidad, imprime en la salida estándar, ya lo veremos más adelante. Por ahora, creeme que es la pantalla. ↩

2. Hay miles de debates sobre escribir código en inglés vs español. No prometo ser constante durante la guía, pero tiendo a codificar en inglés. Pero, si escribir en inglés hará que no entiendas lo que representa cada cosa (cargándote la clase de nombres descriptivos), dale con el español. El piso mínimo que deberías tener es que todo el equipo (sea del TP, o de un trabajo *de posta*) use la misma convención. Y atenti con el tema de caracteres acentuados y eso, que a ¡¡50!! años de inventarse **el código ASCII** seguimos encontrándonos caracteres mochos por ahí, y, en el peor de los casos, puede hacernos fallar la compilción por problemas de encoding. Nadie quiere eso. ↩

3. La memoria de una PC tiene muchísimo más tamaño, pero dejame hacerla dibujable. ↩

4. C es un lenguaje. Decir "C hizo tal cosa" es medio vago: probablemente sea algo que hizo el compilador, o el sistema operativo, o alguna biblioteca más o menos estándar. Cuando decimos "C hizo tal cosa" es porque: a) no nos interesa mucho quién lo hizo (importa que lo hizo *otra persona*, y que es más o menos lo mismo para cualquier programa en C), o b) no sabemos quién lo hizo (y nos da fiaca averiguarlo, de momento). ↩

5. En realidad, en arquitecturas más antiguas este tipo de dato ocupaba un tamaño menor, que dependía del **tamaño de palabra del procesador**. Si te interesa saber toda la historia, te dejo un link **acá**, pero, a modo de resumen, en arquitecturas de 32 o 64 bits siempre va a ocupar 4 bytes 😃 ↩

6. No sé Assembler, creo que el parámetro no va en la misma línea que el `mov`, pero anda por ahí cerca. ↩

7. Lo aclaré en la anterior: no sé Assembler. ↩