Guía de C

Arquitectura y Organización del Computador

Introducción

¿Por qué queremos aprender C?

- C es de bajo nivel, ideal para **system programming**
- C es **pequeño** en comparación con otros lenguajes de programación,
- C permite hacer uso de los recursos subyacentes de manera mucho más sencilla.

Es un lenguaje compilado que corre directamente en el sistema host y permite manipulación directa de memoria. Es un lenguaje de programación de propósito general, que ha sido ampliamente utilizado en sistemas operativos, compiladores, interpretes, etc.

Puntos fuertes:

- **Eficiencia**. Las abstracciones que C permite están fuertemente arraigadas a los tipos de datos y operaciones que ofrecen las computadoras convencionales.[1]
- **Portabilidad**. Teniendo ciertas precauciones, C es un lenguaje portable, lo que significa que el código fuente puede ser compilado en diferentes arquitecturas.
- **Ampliamente utilizado** en sistemas operativos, compiladores, interpretes, etc.
- Permite manipulación directa de **memoria**, lo que lo hace ideal para tareas de bajo nivel.
- Tiene una estructura **simple y clara**.

Puntos débiles:

- C es permisivo con las operaciones que se pueden realizar, por lo que se asume que aquel que programa *sabe lo que está haciendo.*
- Al ser un lenguaje que carece de verificación de errores automáticos (como *index out of array* o *null pointer reference*), los programas son más propensos a errores.
- Los programas en C pueden ser difíciles de entender, ya que el código permite ser muy obtuso en su escritura. De hecho, existe una competencia llamada "Obfuscated C Code Contest" que premia al código más difícil de entender [2]
- Los programas en C pueden ser más difíciles de mantener. Esto se asocia estrechamente con el ítem anterior.

[1] https://dl.acm.org/doi/abs/10.1145/154766.155580
[2] Ver, por ejemplo: https://github.com/ioccc-src/winner/blob/master/1986/bright/bright.orig.c

Recomendaciones para usar esta guía

Para que puedan afianzar de manera correcta los conceptos, es importante que sigan las recomendaciones que se detallan a continuación.

- Empiecen con un **directorio completamente vacío**. Inicialicen un repositorio de git y hagan un commit de un archivo `.gitginore` que puede ser como https://github.com/github/gitignore/blob/main/C.gitignore
- **No copien y peguen el código**. Tipeenlo ustedes. Esto indefectiblemente introducirá errores y eso es **bueno**. Familiarizarse con los mensajes de error es parte fundamental del proceso de aprendizaje.
- Lean los mensajes de error atentamente. En la mayoría de los casos, el mensaje indica exactamente donde está el error y qué es lo que está mal.
- No se queden sólo con los ejemplos. Experimenten con el código, cambien cosas, rómpanlo y vean los mensajes de error.

El aprendizaje de un lenguaje rara vez es lineal. Es un proceso iterativo y es normal que haya que volver atrás para entender algo. Esperamos que esta guía les sirva para aprender los rudimentos de C y que puedan usarlo en de manera efectiva en la materia y en sus proyectos. Si tienen consultas, no duden en preguntar. Si les parece que existe un error o que algo se puede explicar mejor, no duden en hacérnoslo saber.

Ahora sí, ¡a programar!

Hola Orga!

- What's in the box?
- Pain.

Vaya, parece que tenemos un programa en C que nos saluda. Veamos que hace.

Snippet 1: Hola Orga!

```c
#include <stdio.h>              // Incluye la biblioteca stdio.h

int main() {                    // main es el punto de entrada
    printf("Hola Orga!\n");     // imprime en pantalla
    return 0;                   // devuelve un 0 al SO
}
```

Este primer código en C escribe en pantalla el texto `Hola, Orga!` en la consola y termina. Hace uso de la función `printf` de la biblioteca estándar de C que imprime en pantalla. La función `main` es el punto de entrada de cualquier programa en C y devuelve un valor al sistema operativo. Notar que devuelve un tipo `int`.

Lo primero que encontramos es un `#include <stdio.h>`. `#include` es una directiva de preprocesador (porque empieza con `#` que agrega las declaraciones al comienzo del archivo (funciona como un *copy paste*). En particular, el archivo `stdio.h` es un *header file* o encabezado que se encuentra en un directorio estándar del sistema y contiene declaraciones de funciones que interactúan con entrada/salida, como lo es `printf`.

Como resumen llevémonos los siguientes conceptos básicos de un programa en C:

- Todo programa en C tiene una función `main()`, la cual es la primer función que se ejecuta al correr un binario.
- Todo programa en C (todo programa en realidad) devuelve un valor al sistema operativo. En este caso, el valor 0 indica que el programa terminó correctamente.
- Las funciones en C se declaran con el tipo de dato que devuelven, seguido del nombre de la función y los parámetros que recibe.
- Los string literales en C se escriben entre comillas dobles.

Para compilar el programa anterior, se puede hacer uso de un compilador de C, como `gcc`[3]. En la terminal, escribimos:

```
gcc -c hola.c -o hola.o
gcc hola.o -o hola
```

De esta manera, el código fuente es transformado en un código objeto .o, y luego ese archivo objeto se *linkea* para formar un archivo ejecutable para el sistema operativo. Notar que no es necesario hacer un llamado a gcc dos veces, si no que se puede hacer el llamado solo con el .c y generar el binario:

```
gcc hola.c -o hola
```

Pero en general se hace en dos pasos para separar la compilación de la generación del binario.

[3]Para instalar el compilador y algunas utilidades, en Ubuntu, instalar el paquete `build-essential` haciendo `sudo apt install build-essential`.

Ambos comandos generan un archivo con el nombre `hola`, que puede ser ejecutado desde la terminal con el comando

```
./hola
```

En general, usaremos la extensión `.c` para los archivos fuente de C, y `.h` para los archivos de cabecera, de los cuales hablaremos más adelante. Los binarios en linux no tienen extensión. Para compilar, usaremos algunos flags extra que nos permiten tener más información en caso de errores, como `-Wall`, `-Wextra` y `-pedantic`

```
gcc -Wall -Wextra -pedantic -c hola.c -o hola.o
gcc -Wall -Wextra -pedantic hola.o -o hola
```

Ejercicio 1:

Realizar el programa que imprima en pantalla el texto `Hola, Orga!`. Compilar con todos los flags mencionados.

Para no estar escribiendo siempre los mismos comandos, se puede usar un `Makefile`. Empecemos con lo básico de `make`

Make

It's dangerous to go alone, take this!

`make` es una herramienta en la cual podemos especificar cómo se compila un programa. Un archivo `Makefile` contiene reglas que le dicen a `make` cómo compilar un programa.

Qué es make? De acuerdo a Wikipedia:

> En el contexto del desarrollo de software, Make es una herramienta de gestión de dependencias; típicamente, las que existen entre los archivos que componen el código fuente de un programa, para dirigir su recompilación o "generación" automáticamente. (...) La herramienta make se usa para las labores de creación de fichero ejecutable o programa, para su instalación, la limpieza de los archivos temporales en la creación del fichero, todo ello especificando unos parámetros iniciales (que deben estar en el makefile) al ejecutarlo.
>
> Todos los Makefiles están ordenados en forma de reglas, especificando qué es lo que hay que hacer para obtener un módulo en concreto. El formato de cada una de esas reglas es el siguiente:
>
> ```
> target: dependencias
>   commandos
> ```
>
> En *target* definimos el módulo o programa que queremos crear, después de los dos puntos y en la misma línea podemos definir qué otros módulos o programas son necesarios para crear el target. Por último, en la línea siguiente y sucesivas (el cuerpo del *target*) indicamos los *comandos* necesarios para llevar esto a cabo. Es muy importante que los *comandos* estén separados por un tabulador del comienzo de línea (no espacios).

Los makefiles permiten definir dependencias entre archivos (se necesita X para compilar Y) y resolverlas en caso de ser necesario. Además, `make` es una herramienta muy útil para compilar programas grandes, ya que permite compilar solo los archivos que han cambiado y no todo el programa. Esto ahorra tiempo y recursos al compilar.

Definiendo targets y dependencias, construimos las "recetas" para compilar binarios y ejecutar código. Vamos a crear un archivo `Makefile`[4] para compilar el programa `hola.c` que hicimos antes. El archivo `Makefile` se ve de la siguiente manera:

```
all: hola

hola: hola.o
	gcc -Wall -Wextra -pedantic hola.o -o hola

hola.o: hola.c
	gcc -Wall -Wextra -pedantic -c hola.c -o hola.o

clean:
	rm *.o hola

.PHONY: all clean
```

Vemos que en el `Makefile` se definen una serie de targets. Hay ciertas convenciones en el uso de `Makefile` que son importantes tener en cuenta. Por ejemplo, el target `all` es el que se ejecuta por defecto si no se especifica ningún target. Al ejecutar `make` en el directorio actual, se intentará construir `all`. Por defecto, `make` supone que un `target` es un archivo en el disco. La línea `.PHONY:`

[4] Es importante que el archivo tenga este nombre para que el comando `make` lo reconozca

`all clean` indica que estos targets no son archivos, sino que son simplemente comandos que se pueden ejecutar. Esos targets, al ser `.PHONY`, siempre se consideran desactualizados. El target `clean`, por su parte, es un target especial que se usa para limpiar los archivos generados por la compilación. En este caso, se eliminan todos los archivos `.o` y el binario `hola`.

Si creamos el archivo `Makefile` en el directorio y ejecutamos el comando `make` vemos lo siguiente en la terminal:

```
$ make
gcc -Wall -Wextra -pedantic -c hola.c -o hola.o
gcc -Wall -Wextra -pedantic hola.o -o hola
```

¿Cómo se da cuenta `make` lo que tiene que ejecutar? La respuesta es que arma un **árbol de dependencias**. En este caso particular, busca el target `all` y ve que depende de `hola`. Como `hola` no se encuentra en el directorio, busca la manera de construirlo y ve que entre sus recetas tiene una para construirlo, y que depende de `hola.o`. Como `hola.o` no se encuentra en el directorio, busca la manera de construirlo y ve que su construcción depende de `hola.c`. Como `hola.c` es un archivo fuente que se encuentra en el directorio, ejecuta los comandos definidos en esa receta. Eso termina ejecutando `gcc -Wall -Wextra -pedantic -c hola.c -o hola.o`. Al terminar de hacer esto, `make` empieza a volver sobre sus pasos, y entonces ahora usa ese archivo para generar el binario `hola`, ya que ahora posee sus dependencias (`hola.o`). Continúa entonces ejecutando `gcc -Wall -Wextra -pedantic hola.o -o hola`. Con eso quedan satisfechas las dependencias de `all` y por lo tanto `make` termina su trabajo.

> Los *comandos* podrían ser lo que nosotros queramos, `make` es completamente agnóstico de lo que hacen. Se supone, que deben construir el `target` (por ejemplo `hola.o`), pero eso no depende de `make`. Depende exclusivamente de lo que nosotros pongamos ahí. Lo único que `make` nos pide es que pongamos el bendito tabulador al principio de la línea.

Podemos mejorar nuestro `Makefile` para que use variables. Por ejemplo, podemos definir una variable `CC` que contenga el compilador a usar y `CFLAGS` para que contenga los flags. De esta manera, si queremos cambiar de compilador o cambiar algún flag, solo tenemos que cambiar la variable y no todas las recetas.

```
CC = gcc
CFLAGS = -Wall -Wextra -pedantic

all: hola

hola: hola.o
	$(CC) $(CFLAGS) hola.o -o hola

hola.o: hola.c
	$(CC) $(CFLAGS) -c hola.c -o hola.o

clean:
	rm *.o hola

.PHONY: all clean
```

Todavía podemos mejorar más nuestro `Makefile`. Podemos usar lo que se llaman variables automáticas. Estas variables son variables que `make` define automáticamente y que contienen información sobre el target actual. Por ejemplo, `$@` contiene el nombre del target actual, `$<` contiene el primer prerequisito (dependencia) y `$^` contiene todos los prerequisitos. También

podemos definir una variable `TARGET` que contenga el nombre del binario a generar. De esta manera, si queremos cambiar el nombre del binario, solo tenemos que cambiar la variable y no todas las recetas.

```
CC = gcc
CFLAGS = -Wall -Wextra -pedantic
TARGET = hola

all: $(TARGET)

$(TARGET): hola.o
	$(CC) $(CFLAGS) $^ -o $@

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

clean:
	rm *.o $(TARGET)

.PHONY: all clean
```

Por ahora nos vamos a conformar con este Makefile. Cuando tengamos que compilar varios archivos, veremos algunos trucos para no ser repetitivos en las recetas.

Ejercicio 2:

Realizar un Makefile para compilar nuestro programa de Hola Orga! De ahora en más, todos los ejemplos que veamos en la guía, deben tener un Makefile para compilar el programa.

Tipos de datos

Data is like garbage. You'd better know what you are going to do with it before you collect it.

C es un lenguaje de tipado estático. ¿Qué quiere decir esto? Que el tipo de dato de una variable se define en tiempo de compilación y no puede ser cambiado en tiempo de ejecución. Esto implica que el programador debe especificar el tipo de dato de cada variable que declare. Una variable puede hacer referencia a un número entero, un número de punto flotante, un carácter, a un puntero, etc.

Tipos nativos del lenguaje

Los tipos de datos (y sus modificadores) que maneja C nativamente son los siguientes:

- **Enteros**: `char`, `short`, `int`, `long`, `long long`
- **Reales**: `float`, `double`, `long double`
- **Caracteres**: `char`
- **Punteros**: `int*`, `char*`, `void*`, etc.
- **Void**: `void`
- **Booleanos**: `bool` (no es nativo, pero se puede usar con `#include <stdbool.h>`, desde C99)

Para tener idea de los tamaños típicos en una arquitectura de 64 bits, podemos ver la siguiente tabla:

Tipo	Bytes	Rango
char	1	-128 a 127 -2^7 a $2^7 - 1$
unsigned char	1	0 a 255 0 a $2^8 - 1$
short	2	-32768 a 32767 -2^{15} a $2^{15} - 1$
unsigned short	2	0 a 65535 0 a $2^{16} - 1$
int	4	-2147483648 a 2147483647 -2^{31} a $2^{31} - 1$
unsigned	4	0 a 4294967295 0 a $2^{32} - 1$
long	8	$-9{,}2 \times 10^{18}$ a $9{,}2 \times 10^{18}$ -2^{63} a $2^{63} - 1$
unsigned long	8	0 a $1{,}84 \times 10^{19}$ 0 a $2^{64} - 1$

¿Por qué hablamos de tamaños típicos? Porque el tamaño de los tipos de datos en C no está definido por el estándar, sino que depende de la arquitectura y del compilador. Por ejemplo, en una arquitectura de 32 bits, un `int` puede ser de 4 bytes, mientras que en una arquitectura de 64 bits, puede ser de 8 bytes. El estándar sólo define límites mínimos para los tamaños de los

tipos de datos. A continuación, se muestra un ejemplo de cómo se pueden imprimir los tamaños de los tipos de datos en C:

Snippet 2: sizeof

```c
#include <stdio.h>

int main() {
        char c = 100;
        short s = -8712;
        int i = 123456;
        long l = 1234567890;

        printf("char(%lu): %d \n", sizeof(c),c);
        printf("short(%lu): %d \n", sizeof(s),s);
        printf("int(%lu): %d \n", sizeof(i),i);
        printf("long(%lu): %ld \n", sizeof(l),l);

        return 0;
}
```

Este código permite imprimir por pantalla los tamaños de algunos tipos de datos. Para ello disponemos del operador `sizeof()` que devuelve el tamaño en bytes de un tipo de datos o de una variable. Este operador es muy útil para saber cuánta memoria ocupa un tipo de dato en particular. Se resuelve en tiempo de compilación.

Además, este código usa `printf` de manera que se pueden observar algunas cosas:

- El uso de `printf` con el modificador `%lu` (`long unsigned`) para imprimir el tamaño de los tipos de datos.
- El uso de `%d` para imprimir un `char` y un `short`, y `%ld` para imprimir un `long`.
- El orden de los argumentos de `printf`: primero se pone el string (con los correspondientes *especificadores de conversión*) y luego las variables a imprimir, en el orden en que aparecen.[5]

Ejercicio 3:

Realizar un programa que imprima por pantalla todos los tamaños de los tipos de datos (con sus modificadores) vistos hasta el momento. Mirar atentamente la tabla de especificadores de conversión y los `length modifiers` para saber cómo imprimir los distintos tipos de datos. Traten de fixear todos los warnings que les tire el compilador.

Enteros de ancho fijo

Varias veces podemos olvidarnos del tamaño de los tipos de datos al programar, e incluso cuál es el rango de valores que puede representar un tipo en particular. Pero muchas veces, no saber esto nos puede traer problemas. Por ejemplo, si estamos programando un sistema embebido y necesitamos que un entero sea de 16 bits, pero no sabemos si `int` es de 16 bits o de 32 bits, podemos tener problemas. Para no generar confusiones, existe una biblioteca llamada `stdint.h` que define nuevos tipos de datos que resuelven la ambigüedad de los tamaños.

Más información sobre los tipos de datos de ancho fijo en https://en.cppreference.com/w/c/types/integer.

[5] Para más información sobre los especificadores de conversión de `printf`, ver https://en.cppreference.com/w/c/io/fprintf

	8 bits/1 byte	16 bits/2 bytes	32 bits/4 bytes	64 bits/8 bytes
Signed	`int8_t`	`int16_t`	`int32_t`	`int64_t`
Unsigned	`uint8_t`	`uint16_t`	`uint32_t`	`uint64_t`

Ejercicio 4:

Realizar un programa que imprima por pantalla los tamaños de los tipos de datos de ancho fijo de la biblioteca `stdint.h`.

Constantes enteras

Las constantes en C se usan para representar valores directamente en el código. Se pueden escribir de diferentes maneras, dependiendo del tipo de dato que queramos representar: **decimal**, **octal** o **hexadecimal**.

- **Decimal:** se escriben como números enteros, por ejemplo: `1234`
- **Octal:** se escriben con un 0 al principio, por ejemplo: `01234`
- **Hexadecimal:** se escriben con un `0x` o `0X` al principio, por ejemplo: `0x1234`

Los literales enteros en C son de tipo `int` por defecto, pero se pueden especificar otros tipos de datos usando sufijos. Por ejemplo, para especificar un literal de tipo `long` se puede usar el sufijo `l` o `L`, y para especificar un literal de tipo `unsigned` se puede usar el sufijo `u` o `U`. También se pueden combinar los sufijos, por ejemplo: `1234UL` es un literal de tipo `unsigned long`.

Snippet 3: sufijos en literales enteros

```c
unsigned int ui = 71U;
signed long int sli = 9223372036854775807L;
unsigned long long int ui = 18446744073709551615ULL;
```

Más información sobre los literales en C se puede encontrar en https://en.cppreference.com/w/c/language/integer_constant.

Importante: En C, los literales enteros son de tipo `int` por defecto.

Números en punto flotante

Los números en punto flotante son aquellos que tienen una parte decimal. En C, los números en punto flotante se representan con los tipos `float`, `double` y `long double`. La diferencia entre ellos es el tamaño y la precisión. El tipo `float` ocupa 4 bytes y tiene una precisión de 6-7 dígitos decimales, el tipo `double` ocupa 8 bytes y tiene una precisión de 15-16 dígitos decimales, y el tipo `long double` ocupa 10 bytes (o más) y tiene una precisión de 18-19 dígitos decimales.

Constantes en punto flotante

Las constantes en punto flotante se escriben como números decimales, por ejemplo: `3.14` o `2.71828`. También se pueden escribir en notación científica, por ejemplo: `1.23e4` (que equivale a `12300`) o `1.23E4`.

Los literales en punto flotante son de tipo `double` por defecto, pero se pueden especificar otros tipos de datos usando sufijos. Por ejemplo, para especificar un literal de tipo `float` se puede usar el sufijo `f` o `F`, y para especificar un literal de tipo `long double` se puede usar el sufijo `l` o `L`.

Snippet 4: sufijos en literales en punto flotante

```c
10.0   /* type double */
10.0F  /* type float */
10.0L  /* type long double */
```

const

En C, se puede declarar una variable como `const` para indicar que su valor no puede ser modificado después de su inicialización. Esto es útil para definir constantes que no deben cambiar durante la ejecución del programa. Por ejemplo:

Snippet 5: const

```c
const int MAX = 100;
const float PI = 3.14159;
const int i = 1; // const-qualified int
i = 2;           // error: i is const-qualified
```

Enumeraciones

Las enumeraciones son un tipo de dato que permite definir un conjunto de constantes enteras con nombre. Se declaran con la palabra clave `enum` y se pueden usar para mejorar la legibilidad del código. Por ejemplo:

Snippet 6: enum

```c
enum Color { RED, GREEN, BLUE };
enum Color c = RED;
printf("%d\n", c); // imprime 0
printf("%d\n", GREEN); // imprime 1
printf("%d\n", BLUE); // imprime 2
```

Las enumeraciones son útiles para definir constantes que representan estados, opciones o categorías. Por ejemplo, se pueden usar para definir los días de la semana, los meses del año, los colores, etc. Las enumeraciones también se pueden usar para mejorar la legibilidad del código y evitar el uso de números mágicos.

Se puede especificar los valores que toman las enumeraciones. Por ejemplo:

Snippet 7: enum con valores

```c
enum Color { RED = 1, GREEN = 2, BLUE = 4 };
enum Color c = RED;
printf("%d\n", c); // imprime 1
printf("%d\n", GREEN); // imprime 2
printf("%d\n", BLUE); // imprime 4
```

En general, el tipo subyacente de una enumeración es un `int`, pero puede ser otros tipos enteros también. para más información: https://en.cppreference.com/w/c/language/enum.

Conversiones de tipos

Dado una variable en C, puedo realizar una conversion de tipos de manera manual denominada *casting*. Lo que permite el casting (o *casteo*, como se le suele decir), es reinterpretar un valor de cierto tipo como otro.

Veamos un programa un poco más avanzado que incorpora operaciones de casteo explícito:

Snippet 8: Casting

```c
#include <stdio.h>

int main() {
  int mensaje_secreto[] = {116, 104, 101, 32, 103, 105, 102, 116, 32, 111,
  102, 32, 119, 111, 114, 100, 115, 32, 105, 115, 32, 116, 104, 101, 32,
  103, 105, 102, 116, 32, 111, 102, 32, 100, 101, 99, 101, 112, 116, 105,
  111, 110, 32, 97, 110, 100, 32, 105, 108, 108, 117, 115, 105, 111, 110};

  size_t length = sizeof(mensaje_secreto) / sizeof(int);
  char decoded[length];

  for (int i = 0; i < length; i++) {
    decoded[i] = (char) (mensaje_secreto[i]); // casting de int a char
  }

  for (int i = 0; i < length; i++) {
    printf("%c", decoded[i]);
  }
}
```

En este código, se realizan casts de `int` a `char` para poder imprimir el mensaje secreto. El cast se realiza con el operador `(tipo)`.

La sintaxis general para hacer un casting es:

```
(type) expression
```

Este tipo de cast se denomina explícito, ya que el programador está indicando explícitamente que quiere que la variable sea interpretada como otro tipo. Existen otros tipos de cast, como el cast implícito, que se realiza automáticamente por el compilador. Las conversiones implícitas, suceden automáticamente en expresiones a medida que son requeridas. Las reglas son *bastante complejas*, pero en general, el compilador intenta hacer la conversión más segura posible. Por ejemplo, si se suma un `int` y un `float`, el compilador convertirá el `int` a `float` antes de realizar la suma. Esto se denomina promoción de tipo. Para más información ver https://en.cppreference.com/w/c/language/conversion.

Los cast son una herramienta poderosa, pero también peligrosa, ya que se pueden perder datos si no se hace correctamente. Por ejemplo, si se castea un `int` a un `char` y el valor del `int` es mayor a 127, se perderán bits de información.[6]

Dependiendo del cast, pueden pasar dos cosas:

- se reinterpretan el tipo de datos, sin modificar la representación binaria

  `int* i = (int*) un_int;`

- se modifica la representación binaria del dato, por ejemplo en

  `int i = (int) un_float;`

En general, si tenemos activados los warnings del compilador, vamos a tener un warning si se realiza un cast implícito que pueda perder información. Si realizamos un cast explícito, el compilador no nos va a avisar si estamos perdiendo información, ya que asume que el programador sabe lo que está haciendo.

[6] Sólo si el char es `signed` que es lo típico. Si es `unsigned`, el rango es de 0 a 255. Ah sí, el tipo `char` "a secas" puede ser `signed` o `unsigned` dependiendo de la implementación.

Ejercicio 5:

Realizar un programa que imprima el valor de `0.1` como `float` y como `double`. Luego, realizar un cast de `float` a `int` y de `double` a `int`. ¿Qué sucede?

Ejercicio 6:

Compilar y ejecutar el programa del mensaje secreto. ¿Qué mensaje se imprime? ¿Por qué les parece que `length` se calcula de esa manera? ¿Qué pasaría si el mensaje secreto tuviera un tamaño distinto? ¿Por qué se usa `size_t` para calcular el tamaño del mensaje secreto? Ver https://en.cppreference.com/w/c/types/size_t para más información.

Operadores

Language itself is an operator on the mind, shaping what we see and how we interpret the world.

En C, disponemos de varios operadores que nos permiten realizar operaciones aritméticas, lógicas, de comparación, etc. A continuación, se muestra una tabla con los operadores más comunes en C:

operador	tipo
`++, --`	incremento, decremento
`+, -, *, /, %`	aritméticos
`<, <=, >, >=, ==, !=`	comparación
`&&, ||, !`	lógicos
`&, <<, >>, |, ~, ^`	binarios
`=, +=, -=, *=, /=, %=`	asignación
`?:`	condicional

Se encuentran agrupados por tipo de operación, no por precedencia. La precedencia de los operadores en C se puede ver en la siguiente tabla: Tabla de precedencia de operadores en C

Asimismo, en esa tabla podemos observar la asociatividad de los operadores. Por ejemplo, es equivalente escribir `a * b / c` que `(a * b) / c`, ya que el operador `*` y `/` tienen asociatividad de izquierda a derecha.

Asimismo, operaciones como `a = b = c` son posibles en C porque el operador de asignación tiene asociatividad de derecha a izquierda.

Algunas cosas a tener en cuenta:

- `++i` incrementa `i` y luego devuelve el valor de `i`.
- `i++` devuelve el valor de `i` y luego incrementa `i`.
- `a = b = c = 0` asigna 0 a las variables `a`, `b` y `c`.
- `a = b = c` asigna el valor de `c` a `a` y `b`.
- los operadores de comparación devuelven 1 si la condición es verdadera y 0 si es falsa.
- un entero es considerado verdadero si es distinto de 0.

Ejercicio 7:

Realizar un programa que imprima por pantalla el resultado de las siguientes operaciones:

- `a = 5, b = 3, c = 2, d = 1`
- `a + b * c / d`
- `a % b`
- `a == b, a != b`
- `a & b, a | b`
- `~a`
- `a && b, a || b`
- `a << 1`
- `a >> 1`
- `a += b, a -= b, a *= b, a /= b, a%= b`

Importante: Interpretar los resultados de las operaciones y ver si coinciden con lo esperado. Para imprimir los valores de las operaciones binarias, usar `%x` o también `%X` en `printf`.

Ejercicio 8:

Realizar un programa que muestre la diferencia entre `++i` y `i++`.

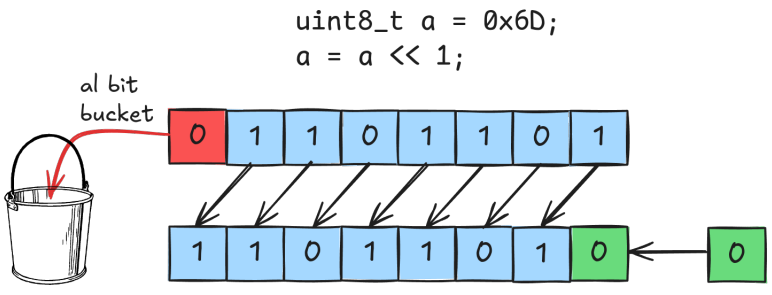
Veamos ahora algunas manipulaciones de bits las cuales tienen especial relevancia en la materia: **shifting** y **máscaras**.

Shifting

Repasemos antes las operaciones de shift a izquierda y derecha. Estas operaciones son muy utilizadas en C para manipular bits.

Algunas limitaciones a tener en cuenta en las operaciones de shifteo:

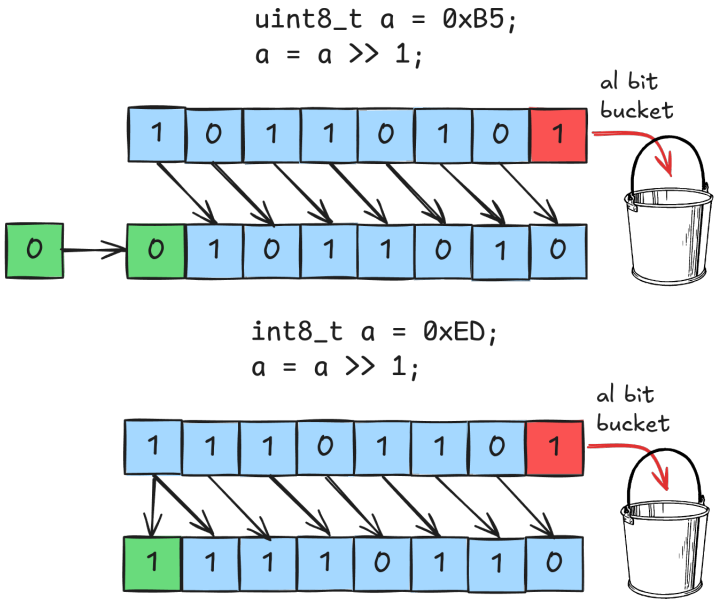
- En `a << b`, los bits que se desplazan a la izquierda se pierden.
- En `a >> b`, los bits que se desplazan a la derecha se pierden.
- En `a << b` y en `a >> b`, el tipo de `b` se promociona al tipo de `a`.
- `b` debe estar entre 0 y el tamaño en bits del tipo de `a` menos 1. Por ejemplo, si `a` es un `uint32_t`, `b` puede valer entre 0 y 31. En otro caso, el comportamiento es indefinido.
- Si `b` es negativo, el comportamiento es indefinido.



Shifteo a izquierda

En relación al shift a derecha:

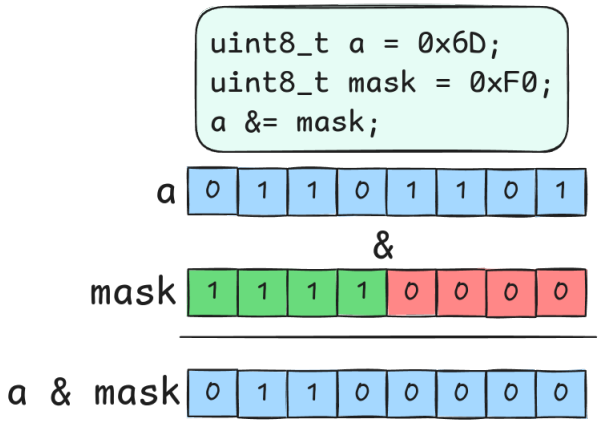
- Cuando shifteamos a derecha un entero con signo, se realiza un shifteo aritmético, es decir, se rellena con el bit de signo.[7]
- Cuando shifteamos a derecha un entero sin signo, se realiza un shifteo lógico, es decir, se rellena con ceros a la izquierda.



Shifteo a derecha

Máscaras

La operación de máscara es una operación muy común en C. Consiste en aplicar una operación binaria de AND u OR bit a bit entre dos números, de manera de afectar ciertos bits de una *palabra*[8] con 0 o con 1, dejando el resto de los bits intactos. Por eso se llama máscara, porque enmascaramos a los bits que no queremos modificar.

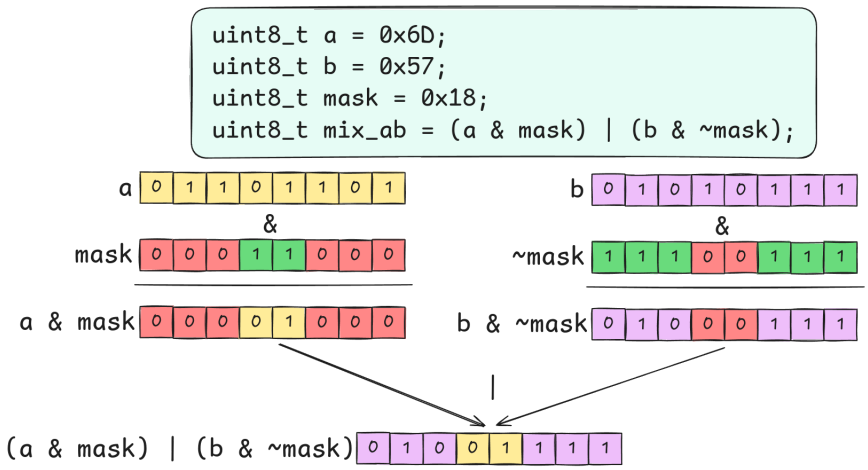


Operación de máscara

[7] En realidad, en este caso el comportamiento es *implementation defined* según el estándar, pero en la práctica, en la mayoría de las arquitecturas, se realiza un shifteo aritmético.

[8] Nos referimos a *palabra* de manera genérica para hablar de una secuencia de bits que tenga sentido tratar como una unidad, por ejemplo 32-bits

Poner un bit en 1 en una palabra se llama *set* y poner un bit en 0 se llama *clear*. Para *setear* un bit, se hace un OR bit a bit con 1, y para *clearear* un bit, se hace un AND bit a bit con 0. Por ejemplo, si queremos combinar dos palabras de 8 bits, de manera que algunos bits sean de una palabra y el resto de la otra, podemos hacer dicha mezcla de la siguiente manera:



Mezcla de palabras de 8 bits

Las operaciones de máscara se utilizan en el paradigma de procesamiento SIMD (Single Instruction Multiple Data) para procesar varios datos al mismo tiempo. En este paradigma, se utilizan registros de 128 bits o más para procesar varios datos al mismo tiempo. Las operaciones de máscara permiten seleccionar qué datos se dejan pasar y cuáles no, lo que permite hacer mezclas de datos de manera eficiente.

Ejercicio 9:

Realizar un programa que compare si los 3 bits más altos de una palabra de 32 bits son iguales a los 3 bits más bajos de otra palabra de 32 bits. Si son iguales, informarlo por pantalla.

Estructuras de control

I believe in creative control. No matter what anyone makes, they should have control over it.

En las siguientes estructuras de control, usaremos una `condition` para decidir si se ejecuta un bloque de código o no. La `condition` puede ser cualquier expresión que devuelva un valor entero. Esto es algo sumamente utilizado en C, ya que el valor 0 es considerado falso y cualquier otro valor es considerado verdadero. Si bien se dispone de tipos booleanos desde C99, el uso de enteros para representar verdadero o falso es una práctica extremadamente común y aceptada idiomáticamente en C.

If-else

Ejemplo de uso de estructura `if-else`:

Snippet 9: if-else

```c
#include <stdio.h>

int main() {
  int a = 5;
  if (a % 2) {
    printf("a es impar\n");
  } else {
    printf("a es par\n");
  }
  return 0;
}
```

switch

La estructura `switch` es una forma de controlar el flujo de un programa en función del valor de una variable. La sintaxis de un `switch` es la siguiente:

Snippet 10: switch

```c
switch (expression) {
  case constant1:
    // código
    break;
  case constant2:
    // código
    break;
  default:
    // código
}
```

El `switch` evalúa la expresión y compara su valor con los valores de cada `case`. Si encuentra una coincidencia, ejecuta el bloque de código correspondiente. Si no encuentra ninguna coincidencia, ejecuta el bloque de código del `default` (si existe). El `break` se utiliza para salir del `switch` una vez que se ha ejecutado el bloque de código correspondiente. Si no se incluye un `break`, el programa continuará ejecutando los bloques de código de los siguientes `case` hasta encontrar un `break` o llegar al final del `switch`. Esto se llama *fall-through*. La expresión del

`switch` debe ser de un tipo entero o un enumerador. Los valores de los `case` deben ser constantes enteras y deben ser únicos dentro del mismo `switch`. No se pueden usar variables como valores de `case`.

Snippet 11: switch

```c
#include <stdio.h>

int main() {
  int a = 2;
  switch (a) {
    case 1:
      printf("a es 1\n");
      break;
    case 2:
      printf("a es 2\n");
      break;
    default:
      printf("a no es ni 1 ni 2\n");
  }
  return 0;
}
```

Es común usar un `switch` en combinación con un `enum` para definir los valores de los `case`. Esto permite tener un código más legible y fácil de mantener. Por ejemplo:

Snippet 12: switch con enum

```c
#include <stdio.h>

enum Color { RED, GREEN, BLUE };

int main() {
  enum Color c = GREEN;
  switch (c) {
    case RED:
      printf("c es rojo\n");
      break;
    case GREEN:
      printf("c es verde\n");
      break;
    case BLUE:
      printf("c es azul\n");
      break;
    default:
      printf("c no es un color válido\n");
  }
  return 0;
}
```

Nota: En este caso, el `switch` es más legible que un `if-else` anidado.

ciclos

Un ciclo `while` se ejecuta mientras la condición sea verdadera. La sintaxis de un ciclo `while` es la siguiente:

Snippet 13: while

```
while (condition) {
  // código
}
```

Analicemos el siguiente ejemplo:

Snippet 14: while

```
#include <stdio.h>

int main() {
  int i = 10;

  while(i--){
    printf("i = %d\n",i); // imprime o no el 0?
  }
}
```

Nota: para responder la pregunta, recordar como funciona el operador post-decremento.

La sintaxis de un ciclo for es la siguiente:

Snippet 15: for

```
for (initialization; condition; update) {
  // código
}
```

La `initilization` se ejecuta una sola vez al principio del ciclo, mientras que `condition` se evalúa en cada iteración y si es verdadera se ejecuta el bloque de código, y `update` se ejecuta al final de cada iteración.

Snippet 16: for example (pun intended)

```
#include <stdio.h>

int main() {
  for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
  }
  return 0;
}
```

Notar como podemos declarar variables en la inicialización del ciclo `for`. El scope de dichas variables es local al ciclo.

Todo ciclo puede ser interrumpido con la sentencia `break`, que termina el ciclo en el que se encuentra, y `continue`, que saltea el resto del bloque de código y pasa a la siguiente iteración. Generalmente, si sabemos la cantidad de veces que vamos a iterar, usamos un ciclo `for`, y si

no, usamos un ciclo `while`. Pero es una cuestión de estilo y preferencia. Ambas estructuras se pueden usar indistintamente y se pueden reemplazar una por la otra.

Ejercicio 10:

Reemplazar el ciclo `while` del snippet anterior por un ciclo `for`.

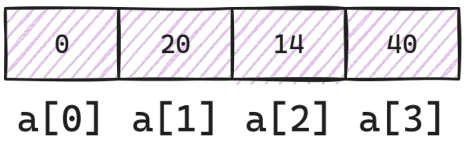
Y ya que estamos con ciclos, pasemos a ver `arrays` y `strings`.

Arrays

Un array es una tira contigua en memoria de elementos del mismo tipo.

Snippet 17:

```
#define N 4

uint32_t a[N];
a[0] = 0;
a[1] = 20;
a[2] = 14;
a[3] = 40;
```



Array

Dependiendo de donde esté definido, un array puede estar sin inicializar. En ese caso, los valores de los elementos del array son basura. Veamos como se inicializa típicamente un array:

Snippet 18:

```
#define N 100

uint32_t a[N];
int i = 0;
while(i < N){
  a[i] = i;
  i++;
}
```

Snippet 19:

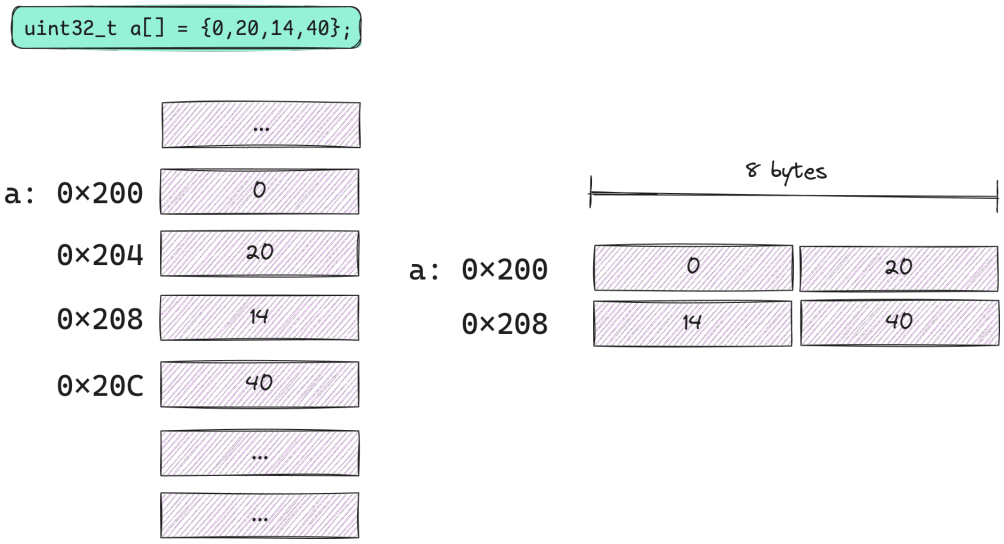
```
#define N 100

uint32_t a[N];
for (int i = 0; i < N; i++){
  a[i] = i;
}
```

También se pueden inicializar los elementos de un array al momento de declararlo:

Snippet 20:

```c
#define N 4
uint32_t a[N] = {0, 20, 14, 40};
uint32_t b[] = {0, 20, 14, 40};
uint32_t c[N] = {0}; // c es {0, 0, 0, 0}
uint32_t d[] = {[1] = 20, [2] = 14, [3] = 40};
```

Ahora, ¿Cómo es que se guardan los valores en memoria? Recordar que dijimos que un array es una tira contigua de elementos de un mismo tipo. Por lo tanto, los elementos de un array se guardan en memoria de manera contigua. A continuación, se muestra un ejemplo de cómo se guarda en memoria un array de 4 elementos de tipo `uint32_t`:



Array: Memoria

En este ejemplo, se supone que el array se ubica en la posición de memoria `0x200`. Cada elemento del array ocupa 4 bytes, ya que es un `uint32_t`. Por lo tanto, el primer elemento del array se ubica en la posición de memoria `0x200`, el segundo en `0x204`, etc.

No hay nada que indique dónde termina un array. Depende de nosotros como programadores saber dónde termina.

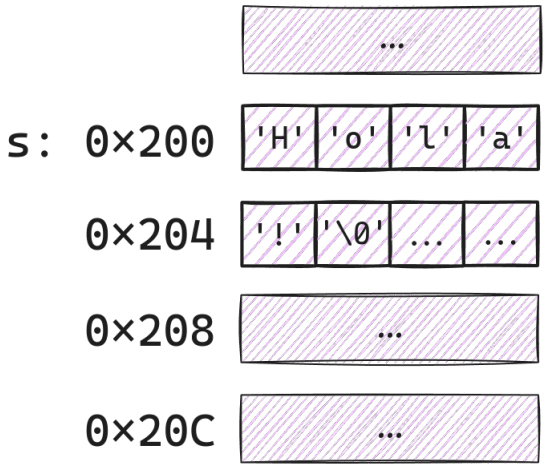
Strings

C no contiene un tipo de dato nativo *strings*, pero sí tiene un tipo *char* que sirve para representar caracteres en formato ASCII. Con este tipo, podemos definir strings como un array de caracteres. La convención en C es que todo string termina en un carácter nulo (`'\0'`), y con esto se puede determinar el final del string. Por ejemplo, el string "Hola!" se define de la siguiente manera:

Snippet 21: strings

```c
char s[] = "Hola!";                          // string literal
char u[] = {'H', 'o', 'l', 'a', '!', '\0'}; // char literals
```

Ambas definiciones son equivalentes. La representación de un string en memoria es la siguiente:



String

Hay una sutileza en la definición de un string literal. Un string literal es un array de caracteres constante. Por lo tanto, no se puede modificar. Si se intenta modificar un string literal, el comportamiento es indefinido. Analicemos el siguiente código[9]:

Snippet 22: string literal

```c
int main(){
  char s[] = "Hola!"; // s es el nombre del array
  char *u = "string"; // u es un puntero a un char

  printf("s = %s\n", s);
  printf("u = %s\n", u);
  s[0] = 'h'; // s = "hola!"
  u[0] = 'S'; // ERROR
}
```

Nota: Si se intenta modificar un string literal, el compilador no necesariamente va a tirar un error. En general, el compilador va a reservar un espacio de memoria para el string literal y va a intentar escribir en ese espacio. Si el espacio de memoria es de solo lectura, el sistema operativo va a arrojar un error de segmentación.

```
$ gcc -Wall -Wextra -pedantic strings.c -o strings
$ ./strings
s = Hola!
u = string
[1] 65377 segmentation fault (core dumped) ./strings
```

Nota: El error de segmentación (segmentation fault) es un error que ocurre cuando un programa intenta acceder a una dirección de memoria que no tiene permiso para acceder. Esto puede ocurrir por varias razones, como intentar escribir en un string literal, o intentar acceder a un puntero nulo.

[9] Vemos que aparece un puntero a un char, lo veremos más adelante.Tomarlo simplemente como una variable que guarda la dirección de memoria de un char.

Ejercicio 11:

Realizar un programa que rote un arreglo de números enteros a la izquierda. El arreglo puede estar *hardcodeado*. Por ejemplo, si el arreglo es `[1, 2, 3, 4]`, el resultado debe ser `[2, 3, 4, 1]`. Cuando veamos punteros, podremos hacer una función de rotación genérica.

Ejercicio 12:

Generalizar el ejercicio anterior para que la rotación sea un parámetro de entrada. Por ejemplo, si el arreglo es `[1, 2, 3, 4]` y la rotación es 2, el resultado debe ser `[3, 4, 1, 2]`.

Ejercicio 13:

Realizar un programa que tire un dado de 6 caras 60 millones de veces y cuente la cantidad de veces que salió cada número. Para esto, usar un array de 6 elementos. Luego imprimir el resultado por pantalla. Para tirar el dado aleatoriamente, usar la función `rand()` de la librería `stdlib.h`. Ver el ejemplo de uso provisto en https://en.cppreference.com/w/c/numeric/random/rand.

Declaración y definición

The gift of words is the gift of deception and illusion

Para seguir avanzando en nuestro aprendizaje de C, tenemos que hacer una distinción clara entre dos conceptos: *declaración* y *definición*. Veamos su definición (*pun not intended*):

Definición se refiere al lugar en donde la variable es creada o donde se le asigna un espacio en memoria (*storage*). En el caso de funciones, la definición es el lugar en donde se le asigna un procedimiento a la función.

Declaración se refiere a los lugares en donde la naturaleza de la variable es anunciada, pero no se le asigna un espacio en memoria.

La **declaración** especifica el tipo de una variable/función. Esta es usada por el compilador para asegurar ciertas propiedades de tipos (Por ejemplo, que dos números de tipo `int` puedan ser sumados, o que el tipo que recibe una función sea el esperado). Toda variable en C debe ser declarada antes de que pueda ser usada. La forma en la que lo hacemos es escribir el tipo de la variable y su nombre.

Por otro lado, la **definición** corresponde a darle un valor concreto al identificador declarado. Para el caso de variables se le asignan valores del tipo, mientras que para funciones, se define al asignarle el procedimiento a efectuar. Tener en cuenta que en muchas situaciones, aunque no estemos explícitamente dando un valor a una variable, el compilador lo hace por nosotros, y esto representa entonces una definición.

El lenguaje nos permite declarar y definir al mismo tiempo.

Snippet 23:

```c
extern int mi_int;                    // decl

int sumarUno(int numero);             // decl

double sumar(double num1, double num2){ // decl y def
  return num1 + num2;
}

int main(){
  int i = 0;                          // decl y def

  char line[1000];                    // decl y def
  const double e = 2.71828182845905; // decl y def

  int count;                          // decl y def
  count = 10;
}
```

Veamos línea por línea:

- `extern int mi_int` **declara** una variable de tipo `int`. Su definición puede estar en otro archivo.[10]
- `int sumarUno(int numero)` **declara** una función que toma un valor de tipo `int`, y devuelve `int`.

[10] Veremos el uso de la palabra clave `extern` más adelante.

- `double sumar(double num1, double num2)` **declara** y **define** una función que toma dos valores de tipo `double`, y devuelve `double`.
- `int i = 0` **declara** y **define** una variable con su tipo (`int`) y nombre (`i`) y la **define** con un valor concreto (`0`).
- `char line[1000]` **declara** una variables con su tipo (`char[]`) y nombre (`line`) y la **define** con valores basura.
- `const double e = 2.71828182845905` **declara** y **define** una variable con su tipo (`double`) y nombre (`e`) y la **define** con un valor concreto (`2.71828182845905`).
- `int count` **declara** una variable con su tipo (`int`) y nombre (`count`) y la **define** con valores basura.
- `count = 10` simplemente asigna el valor `10` a la variable count. Se trata de una **asignación**.

Hay una regla en C, llamada *One Definition Rule* (ODR), que dice que:

Una variable o función puede ser declarada múltiples veces en distintos lugares pero solo puede ser definida una vez.

Scope y duración de un identificador

En C, un identificador es un nombre que se le da a una variable, función, estructura, etc. Un identificador tiene dos propiedades importantes: *scope* y *duración*.

Scope

El *scope* de un identificador es el lugar en el código donde este puede ser accedido. Este concepto es alcanzado tanto para variables como para nombres de funciones. Los scopes pueden ser[11]:

- **Block scope:** El identificador puede ser accedido en cualquier lugar posterior a su declaración en el mismo bloque donde se define. Para que un identificador tenga *block scope*, debe ser declarado dentro de un bloque de código (por ejemplo, dentro de una función o un bucle).
- **File scope:** El identificador puede ser accedido en cualquier lugar posterior a su declaración dentro del mismo archivo. Para que un identificador tenga *file scope*, debe ser declarado fuera de cualquier función. Se les llama también **variables globales**.

Duración

La duración de una variable es el tiempo durante el cual existe. En ese tiempo, la variable tiene un espacio de memoria asignado. No quiere decir que pueda ser accedida, eso depende del scope. quiere decir que la variable tiene un espacio de memoria asignado. Las duraciones pueden ser:

- **Estática:** La variable existe durante toda la ejecución del programa. Toda variable global es estática. Se destruye cuando el programa termina.
- **Automática:** La variable existe durante el llamado a una función. Se destruye cuando la función termina. Esto es lo que sucede con las variables locales a una función.

[11] Hay más, pero estos son los más comunes

- **Dinámica:** La variable existe durante el tiempo que se le asigna memoria. Esto es controlado por el programador. Se crea, por ejemplo, cuando se llama a la función `malloc` y se destruye cuando se llama a la función `free`. Lo veremos más adelante.

Toda variable local a una función tiene, por defecto, una duración *automática* y un *block scope*. Por otro lado, toda variable global tiene una duración *estática* y un *file scope*.

Además, toda variable global no inicializada explícitamente es inicializada a 0. Esto no sucede con las variables locales. Por lo tanto, si una variable local no es inicializada explícitamente, su valor es basura.

Se conoce como *shadowing* a la situación en la que una variable local tiene el mismo nombre que una variable global. En este caso, la variable local .ºculta.ª la variable global. Esto puede llevar a confusiones y errores, por lo que se recomienda evitarlo.

Veamos un ejemplo de *shadowing*:

Snippet 24:

```c
#include <stdio.h>

int i; // inicializada a 0

void funcion() {
  int i = 1; // variable local
  printf("i = %d\n", i); // imprime 1
}

int main() {
  funcion();
  printf("i = %d\n", i); // imprime 0
  return 0;
}
```

En este ejemplo, la variable global `i` es inicializada a 0. Luego, en la función `funcion`, se declara una variable local `i` que es inicializada a 1. Cuando se imprime el valor de `i` dentro de la función, se imprime 1. Sin embargo, cuando se imprime el valor de `i` en la función `main`, se imprime 0, ya que se está accediendo a la variable global.

Ejercicio 14:

Realizar un programa que declare una variable global y una variable local. Luego, imprimir el valor de ambas variables en la función `main`. Luego, probar darles el mismo nombre. ¿Qué sucede? ¿Por qué?

Funciones

Words can carry any burden we wish. All that's required is agreement and a tradition upon which to build.

En C, una función esta conformada por el nombre que lleva, el tipo, orden y número de parámetros que recibe y el tipo de retorno. La sintaxis de una función es la siguiente:

```
return-type function-name (parameters){
  declarations
  statements
}
```

Las funciones pueden retornar un valor de un tipo en particular (por ejemplo, `int`), o si no devuelven nada, esto se indica con el tipo especial `void`. Si la función no recibe nada, también se indica con `void`.[12]. Es necesario declarar toda función antes de ser usada[13]. Asi también pueden haber múltiples declaraciones de una misma función, pero solo puede haber una única definición de la misma (ODR).

Pasaje de parámetros

En C, los parámetros de una función son pasados por valor. Esto significa que se crea una copia de la variable que se pasa como argumento. Por lo tanto, si se modifica el valor de un parámetro dentro de una función, no se modifica el valor de la variable original. Veamos un ejemplo:

Snippet 25: Pasaje por valor

```c
#include <stdio.h>

void duplicar(int n) {
  n = n * 2;
}

int main() {
  int n = 5;
  duplicar(n);
  printf("%d\n", n); // imprime 5
  return 0;
}
```

Para poder modificar el valor de una variable, se puede pasar un puntero a la función. Un puntero es una variable que guarda la dirección de memoria de otra variable. Lo veremos más adelante.

Entonces recordar:

Los parámetros de una función son pasados por valor. Siempre

Además, tener en cuenta que una función puede devolver un valor y eso se indica con la palabra clave `return`. Si la función no devuelve nada y queremos terminar la función, usamos `return;`. Tener en cuenta que en C no podemos devolver más de un valor. Tampoco podemos

[12] Notar que en C, no es lo mismo declarar la función sin argumentos que especificar `void`

[13] Recordemos que una definición puede servir como declaración, de la misma forma que en definiciones de variables

declarar una función que devuelva un array. Tenemos formas de hacer ambas cosas, lo veremos más adelante.

Otro concepto importante es el de *prototipo de función*. Un prototipo de función es una declaración de la función que le dice al compilador el nombre de la función, el tipo de retorno y los tipos de los parámetros. Un prototipo de función se ve de la siguiente manera:

```
return-type function-name (parameters);
```

En general, dichas declaraciones se encuentran en un archivo de cabecera o encabezado (*header file*, con extensión `.h`) que se incluye en el archivo fuente (*source file*, con extensión `.c`) donde se define la función.

Veamos un ejemplo:

Snippet 26: número primo

```c
#include <stdio.h>

int es_primo(int n) {      // definición de función
  if (n <= 1) {
    return 0;
    }


  for (int divisor = 2; divisor * divisor <= n; divisor++) {
    if (n % divisor == 0) {
      return 0;
    }
  }

  return 1;
}

int main(void) {
  int n;
  printf("Ingrese un numero: ");
  scanf("%d", &n);                // se usa para leer un número del teclado

  if (es_primo(n)) {
    printf("Primo\n");
  } else {
    printf("No es primo\n");
  }

  return 0;
}
```

En este ejemplo tenemos dos funciones. La función `main` donde empieza a ejecutar nuestro programa, y una función `es_primo` que toma un número y devuelve 1 si es primo o 0 si no lo es.

También se podría haber escrito de la siguiente manera, que es más común en C:

Snippet 27: Analisis de numero primo

```c
#include <stdio.h>

int es_primo(int n);    // prototipo o declaración de función

int main(void) {
  int n;
  printf("Ingrese un numero: ");
  scanf("%d", &n);      // se usa para leer un número del teclado

  if (es_primo(n)) {
    printf("Primo\n");
  } else {
    printf("No es primo\n");
  }

  return 0;
}

int es_primo(int n) {     // definición de función
  // cuerpo de la función igual que antes
}
```

La función `es_primo` es definida después de la función `main`. Esto no es un problema, ya que el compilador sabe que existe una función llamada `es_primo` porque la hemos declarado antes de usarla, con su prototipo. Si no lo hubiéramos hecho, el compilador no sabría que existe y nos arrojaría un error.

Ejercicio 15:

Realizar un programa que calcule el factorial de un número entero positivo. Para esto, usar una función que reciba el número y devuelva el resultado.

- El factorial de un número entero positivo n es el producto de todos los números enteros positivos menores o iguales a n. Por ejemplo, el factorial de 5 es $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.
- El factorial de 0 es 1.
- El factorial de un número negativo no está definido.

Se puede realizar utilizando recursión o iteración.

Veamos, a continuación, una serie de ejemplos. Estudiados los mismos debería quedar claro el concepto de prototipo de función y la diferencia entre scope y duración. A su vez, se introduce el keyword `static`.

Snippet 28: Scope y duración

```c
#include <stdio.h>
#define FELIZ 0
#define TRISTE 1

void ser_feliz(int estado);
void print_estado(int estado);

int main(){
  int estado = TRISTE; // automatic duration. Block scope
  ser_feliz(estado);
  print_estado(estado); // qué imprime?
}

void ser_feliz(int estado){
  estado = FELIZ;
}

void print_estado(int estado){
  printf("Estoy %s\n", estado == FELIZ ? "feliz" : "triste");
}
```

Ejercicio 16:

¿Qué imprime el programa anterior? Traten de entender cómo se manifiestan el scope y la duración de cada variable.

Ahora consideremos el mismo programa pero con una modificación:

Snippet 29: Scope y duración. v2

```c
#include <stdio.h>
#define FELIZ 0
#define TRISTE 1

int estado = TRISTE; // static duration. File scope

void ser_feliz();
void print_estado();

int main(){
  print_estado();
  ser_feliz();
  print_estado(); // qué imprime?
}

void ser_feliz(){
  estado = FELIZ;
}

void print_estado(){
  printf("Estoy %s\n", estado == FELIZ ? "feliz" : "triste");
}
```

Ejercicio 17:

¿Qué imprime el programa en su versión modificada?

Por último, veamos un ejemplo de uso de la palabra clave `static` en una función.

Snippet 30: static

```c
#include <stdio.h>
#define FELIZ 0
#define TRISTE 1

int estado = TRISTE; // static duration. File scope

void alcoholizar();
void print_estado();

int main(){
  print_estado();
  alcoholizar();
  print_estado();
  alcoholizar();alcoholizar();alcoholizar();
  print_estado(); // que imprime?
}
void alcoholizar(){
  static int cantidad = 0; // static duration. block scope
  cantidad++;
  if(cantidad < 3){
    estado = FELIZ;
  }else{
    estado = TRISTE;
  }
}

void print_estado(){
  printf("Estoy %s\n", estado == FELIZ ? "feliz" : "triste");
}
```

Ejercicio 18:

¿Qué imprime el programa en su versión con el keyword `static`? ¿Qué pasa si se quita la palabra clave `static` en la función `alcoholizar`?

Ejercicio 19:

Analizar el siguiente programa. ¿Qué imprime? Compilarlo y ejecutarlo para verificar el resultado.

```c
#include <stdio.h>

int g = 10;

void functionA() {
    int a = 20;
    static int b = 30;

    printf("Dentro de functionA:\n");
    printf("  g = %d\n", g);
    printf("  a = %d\n", a);
    printf("  b = %d\n", b);

    // Modificación de las variables
    g += 5;
    a += 10;
    b += 5;
}

void functionB() {
    int a = 40;
    static int c = 50;

    printf("\nDentro de functionB:\n");
    printf("  g = %d\n", g);
    printf("  a = %d\n", a);
    printf("  c = %d\n", c);

    // Modificación de las variables
    g += 5;
    a += 10;
    c += 5;
}

int main() {
    printf("Dentro de main:\n");
    printf("  g = %d\n", g);

    functionA();
    functionB();
    functionA();
    functionB();

    printf("\nFinal en main:\n");
    printf("  g = %d\n", g);

    return 0;
}
```

Ejercicio 20:

El siguiente esquema de programa muestra solo definiciones de funciones y de variables. Indicar el scope y duración de cada variable.

```
int b, c;
void f(void)
{
  int b, d;
}

void g(int a)
{
  int c;
  {
    int a, d;
  }
}
```

Etapa de desarrollo

A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.

Compilación

C es un lenguaje *compilado*. El resultado de compilar un programa deriva en un archivo ejecutable que corre sobre la máquina. Hay que tener en cuenta que en C *cada archivo se compila por separado*. Por lo tanto, si se tienen varios archivos fuente, cada uno de ellos se compila en un archivo objeto. Luego, se los enlaza ("linkea") para generar el archivo ejecutable. ¿Y los archivos de cabecera? Estos se incluyen en los archivos fuente (con la directiva `#include`), y el preprocesador los reemplaza por el contenido del archivo de cabecera (como si fuera un copy-paste). Es decir, que el compilador de C en sí, no "ve" los archivos de cabecera, sino que ya los ve incluidos o expandidos, como se suele decir, en el archivo fuente. Cada una de estas unidades de compilación se llama *unidad de traducción* o en inglés *translation unit*.

El proceso de compilación se divide en varias etapas:

- **Preprocesamiento:** En esta etapa, el preprocesador de C toma el código fuente y realiza una serie de transformaciones. Los archivos fuentes suelen tener extensión `.c` y los headers `.h`. Por ejemplo, se incluyen los archivos de cabecera, se reemplazan los macros, se eliminan los comentarios, etc. Para invocar únicamente al preprocesador en gcc se puede hacer:

  ```
  gcc -E archivo.c -o archivo.i
  ```

  El resultado de esta etapa es un archivo con extensión `.i`. Esto no se suele hacer, sólo es útil para diagnosticar errores en la etapa de preprocesamiento.

- **Compilación:** En esta etapa, el compilador de C toma el archivo generado por el preprocesador y lo convierte en un archivo en lenguaje ensamblador. Para generar el archivo en assembly con gcc se puede hacer:

  ```
  gcc -S archivo.c -o archivo.s
  ```

  El resultado de esta etapa es un archivo con extensión `.s`. Esto tampoco se suele hacer, pero puede ser instructivo para entender el trabajo hecho por el compilador. Existe un sitio excelente donde podemos ver esta salida para diferentes compiladores: [compiler explorer]

- **Ensamblado:** En esta etapa, el ensamblador toma el archivo generado por el compilador y lo convierte en un archivo objeto.

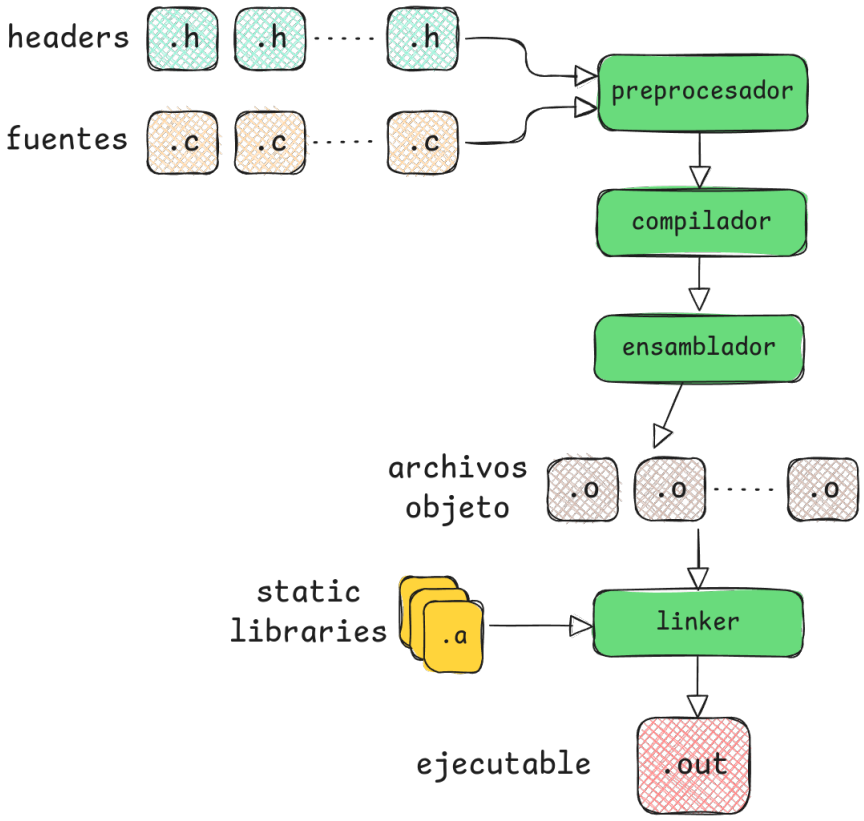
  ```
  gcc -c archivo.c -o archivo.o
  ```

  Este archivo tiene extensión `.o` y contiene el código máquina correspondiente al código fuente. Lo más común es invocar de esta manera al compilador de C, que realiza todas las etapas anteriores en un solo comando.

- **Linking:** En esta etapa, el linker toma los archivos objeto generados por el ensamblador y los combina en un solo archivo ejecutable. El resultado de esta etapa es un archivo ejecutable. Se lo puede invocar de la siguiente manera:

```
gcc archivo.o -o binario
```

Además, en esta etapa se resuelven las referencias a funciones y variables globales. Si una función o variable global es declarada en un archivo fuente y definida en otro, el linker se encarga de resolver esta referencia. Si no se encuentra la definición de una función o variable global, el linker arroja un error. Si una definición de una función o variable global aparece en más de un archivo objeto, el linker arroja un error (ODR). Las bibliotecas estáticas y dinámicas que contienen funciones ya compiladas de la biblioteca estándar o de terceros también se linkean en esta etapa.



Compilación en C

Header files

Existe una convención en C de separar la declaración de las definiciones. Las definiciones se ponen en un archivo fuente (`.c`) y las declaraciones en un archivo de cabecera (`.h`). Esto se hace para separar la interfaz de la implementación. La interfaz es lo que el usuario de la biblioteca ve, y la implementación es cómo se hace. Todo lo que necesita el compilador para compilar un archivo fuente es la declaración de las funciones y las variables globales (la interface). Por lo tanto, se incluyen los archivos de cabecera en los archivos fuente. Luego, será el linker el que se encargue de resolver las referencias a las definiciones y verificar que efectivamente, todo lo declarado tenga una definición (y sólo una) en algún lugar. Los errores más comunes que arroja el linker son: *multiple definition* (definición múltiple) y *undefined reference* (referencia indefinida). El primero se da cuando existe más de una definición de una función o variable. El segundo se da cuando no puede encontrar ninguna definición.

Compilación separada. Primer ejemplo

Veamos un ejemplo sencillo de compilación separada:

Snippet 31: funca.h

```c
/* funca.h */
#ifndef FUNCA_H
#define FUNCA_H

void a();

#endif // FUNCA_H
```

En este archivo `funca.h` se declara la función `a`. La directiva `#ifndef` es una directiva de preprocesador que se lee como *if not defined*. Si la macro `FUNCA_H` no está definida, se define y se incluye el contenido del archivo. Si ya está definida, no se incluye el contenido del archivo. Esto se hace para evitar la inclusión múltiple de un archivo de cabecera en una misma unidad de traducción. Esto puede traer problemas en inclusiones recursivas (A incluye a B, B incluye a C, C incluye a A) y también para incrementar la velocidad de compilación.

Nuestro archivo fuente sería el siguiente:

Snippet 32: funca.c

```c
/* funca.c */
#include "funca.h"
#include <stdio.h>

void a(){
  printf("Hola, soy A!\n");
}
```

Y el archivo principal sería:

Snippet 33: main.c

```c
/* main.c */
#include "funca.h"

int main(){
  a();
}
```

Para compilar, sería:

```
$ gcc -c funca.c -o funca.o
$ gcc -c main.c -o main.o
$ gcc funca.o main.o -o binario
$ ./binario
Hola, soy A!
```

Compilación separada. Segundo ejemplo

Ahora veamos un ejemplo con 3 archivos fuente y 2 archivos de cabecera.

Snippet 34: funca.h

```c
/* funca.h */
#ifndef FUNCA_H
#define FUNCA_H

void a();

#endif // FUNCA_H
```

Snippet 35: funcb.h

```c
/* funcb.h */
#ifndef FUNCB_H
#define FUNCB_H

void b();

#endif // FUNCB_H
```

Snippet 36: funca.c

```c
/* funca.c */
#include "funca.h"
#include <stdio.h>

void a(){
  printf("Hola, soy A!\n");
}
```

Snippet 37: funcb.c

```c
/* funcb.c */
#include "funcb.h"
#include <stdio.h>

void b(){
  printf("Hola, soy B!\n");
}
```

Snippet 38: main.c

```c
/* main.c */
#include "funca.h"
#include "funcb.h"

int main(){
  a();
  b();
}
```

Para compilar, sería:

```
$ gcc -c funca.c -o funca.o
$ gcc -c funcb.c -o funcb.o
$ gcc -c main.c -o main.o
$ gcc funca.o funcb.o main.o -o binario
$ ./binario
Hola, soy A!
Hola, soy B!
```

Este es un buen momento para ver cómo se actualizaría el `Makefile` para compilar varios archivos. Un primer intento podría ser:

```
CC = gcc
CFLAGS = -Wall -Wextra -pedantic
TARGET = binario

all: $(TARGET)

$(TARGET): funca.o funcb.o main.o
	$(CC) $(CFLAGS) $^ -o $@

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

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

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

clean:
	rm *.o $(TARGET)

.PHONY: all clean
```

Sin embargo, vemos que las reglas para compilar los archivos `main.o`, `funca.o` y `funcb.o` son iguales. Podemos usar un patrón para definir una regla genérica. Esto se hace con el símbolo `%`. El símbolo `%` es un comodín que representa cualquier cadena de caracteres. Por lo tanto, podemos definir una regla genérica para compilar cualquier archivo fuente en un archivo objeto. Este tipo de regla se llama *pattern rule*. Más info en https://www.gnu.org/software/make/manual/html_node/Pattern-Rules.html.

Por otro lado, como los nombres de los archivos fuentes generan archivos objetos con el mismo nombre, también podemos usar una variable para definir el nombre de los archivos fuentes. Luego, con una *substitution reference* podemos generar los nombres de los archivos objetos. Esto se hace en la línea `OBJ = $(SRC:.c=.o)`. Esta línea le dice a `make` que genere una lista de archivos objetos a partir de la lista de archivos fuentes. Más información en https://www.gnu.org/software/make/manual/html_node/Substitution-Refs.html.

```
CC = gcc
CFLAGS = -Wall -Wextra -pedantic
TARGET = binario

all: $(TARGET)

SRCS = funca.c funcb.c main.c
OBJS = $(SRCS:.c=.o)

$(TARGET): $(OBJS)
        $(CC) $(CFLAGS) $^ -o $@

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

clean:
        rm *.o $(TARGET)

.PHONY: all clean
```

Bueno, esto ya luce bastante bien. Pero falta algo. Ahora tenemos headers. Al ser modificados, es necesario disparar el proceso de compilación de todos los archivos que incluyen a dichos headers. Pero como los archivos `.c` en sí no cambian al modificar los headers, `make` no los recompila.

Las dependencias que un módulo `.c` puede tener con los headers `.h` pueden ser complicadas de escribir y mantener a mano. Básicamente, tendríamos que poner todos los `.h` que en caso de ser modificados, necesitan disparar una recompilación del módulo. Esto se haría por ejemplo de la siguiente manera:

```
module1.o: module1.c module1.h
module2.o: module2.c module2.h
module3.o: module3.c module3.h
main.o: main.c module1.h module2.h
```

Esto es un problema, porque si tenemos muchos módulos y muchos headers, se hace muy difícil mantenerlo. Además, si un header incluye a otro header, y este último cambia, el primero también debería recompilarse. Pero no lo sabemos a menos que mantengamos el árbol de dependencias a mano. Los include guards justamente evitan duplicaciones en los include, pero cuando tenemos que saber quién incluye a quién y tenemos que además mantener el árbol de dependencias a mano, nos estamos plantando bombas en el camino. Por eso existen unos flags de compilación que se le pueden pasar a gcc: `-MMD -MP`. Esto hace que genere unos archivos de dependencias con extensión `.d`. Se pueden chusmear, porque se van a generar en el directorio de trabajo. Ya que el compilador estudia todas las dependencias entre los archivos, como parte de su proceso de compilación habitual, incluyendo este flag le decimos que nos genere targets armados de Makefile que ya nos sirvan. Es un ejemplo de cooperación entre el compilador y make. Finalmente, lo que hacemos con el último `-include` es justamente "pegar.esas reglas para informarle a make que tenga en cuenta esas dependencias y dispare la compilación en caso de cambios en los headers. El `-include` le dice a make que incluya el archivo de dependencias, pero no arroje un error si no existe. Alta black magic.


```
CC = gcc
CFLAGS = -Wall -Wextra -pedantic -MMD -MP
TARGET = binario

all: $(TARGET)

SRCS = funca.c funcb.c main.c
OBJS = $(SRCS:.c=.o)

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) $^ -o $@

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

-include $(OBJS:.o=.d)

clean:
    rm *.o $(TARGET) *.d

.PHONY: all clean
```

Ejercicio 21:

Compilar con ese Makefile y ver qué pasa. Modificar un header y ver qué sucede.

Resolución de variables globales

En C, las variables globales son visibles en todos los archivos fuente. Por lo tanto, si se define una variable global en un archivo fuente, esta es visible en todos los archivos fuente. Si se define una variable global en un archivo fuente y se declara en otro, el linker se encarga de resolver esta referencia.

Analicemos el siguiente ejemplo:

Snippet 39:

```c
// file1.h
#ifndef FILE1_H
#define FILE1_H

#include <stdio.h>
#include "file1.h"

int count;
void print_count();

#endif // FILE1_H
```

Snippet 40:

```c
// file1.c
#include "file1.h"

void print_count(){
  printf("Count: %d\n", count);
}
```

El archivo principal sería:

Snippet 41:

```c
// main.c
#include <stdio.h>
#include "file1.h"

void increment_count() {
  count++;
}

int main() {
  count = 10;
  print_count();
  increment_count();
  printf("Count en main: %d\n", count);
  return 0;
}
```

Esto parece tener mucho sentido. ¿Qué sucede si intentamos compilarlo?

```
$ gcc -Wall -c file1.c -o file1.o
$ gcc -Wall -c main.c -o main.o
$ gcc -Wall file1.o main.o -o binario
/usr/bin/ld: main.o:(.bss+0x0): multiple definition
of `count'; file1.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

¡Changos! ¿Qué pasó? El linker nos está diciendo que hay una definición múltiple de la variable `count`. Esto es porque la variable `count` está definida en el archivo fuente `file1.c` y en el archivo fuente `main.c`. Para resolver esto, se puede usar la palabra clave `extern` en la declaración de la

variable `count` en el header `file1.h`.

Snippet 42:

```
// file1.h
#ifndef FILE1_H
#define FILE1_H

#include <stdio.h>
#include "file1.h"

// declaración de la variable count como extern. Esto no es una definición!
extern int count;

void print_count();

#endif // FILE1_H
```

Necesitamos una definición (ODR) de la variable `count`. Esto se hace en un solo archivo fuente. En este caso, lo hacemos en `main.c`.

Snippet 43:

```
// main.c
#include <stdio.h>
#include "file1.h"

int count; // definición de la variable count

void increment_count() {
  count++;
}

int main() {
  count = 10;
  print_count();
  increment_count();
  printf("Count en main: %d\n", count);
  return 0;
}
```

Compilamos:

```
$ gcc -Wall -c file1.c -o file1.o
$ gcc -Wall -c main.c -o main.o
$ gcc -Wall file1.o main.o -o binario
$ ./binario
Count: 10
Count en main: 11
```

Referencias

Muad'Dib learned rapidly because his first training was in how to learn. And the first lesson of all was the basic trust that he could learn. It's shocking to find how many people do not believe they can learn, and how many more believe learning to be difficult.

- Kernighan, Ritchie - The C Programming Language, 2nd Edition.
- Seacord, Robert - Effective C: An Introduction to Professional C Programming
- C reference: https://en.cppreference.com/w/c
- GNU C reference: https://gcc.gnu.org/onlinedocs/gcc/
- GNU Make reference: https://www.gnu.org/software/make/manual/html_node/
- C programming: https://www.learn-c.org/