# Beej’s Guide to C Programming

Traducción Gassmann Stefano Alejandro

Brian “Beej Jorgensen” Hall

# Contents

# Chapter 1

# Prefacio

> *C no es un gran lenguaje, y no está bien servido por un gran libro.*
>
> –Brian W. Kernighan, Dennis M. Ritchie

No tiene sentido desperdiciar palabras aquí, saltemos directamente al código C:

```
E((ck?main((z?(stat(M,&t)?P+=a+'{'?0:3:
execv(M,k),a=G,i=P,y=G&255,
sprintf(Q,y/'@'-3?A(*L(V(%d+%d)+%d,0)
```

Y vivieron felices para siempre. Fin! :D

¿Qué? ¿Algo aún no está claro sobre todo este asunto del lenguaje de programación C?

Siendo honesto, ni siquiera estoy seguro de qué hace el código anterior. Es un fragmento de uno de los participantes del Concurso internacional de código C ofuscado[1] de 2001, una maravillosa competencia en la que los participantes intentan escribir el código C más ilegible posible, con resultados a menudo sorprendentes.

La mala noticia, es que, si eres un principiante en todo esto, ¡todo el código C que veas probablemente parezca ofuscado! La buena noticia, es que no va a ser así por mucho tiempo.

Lo que intentaremos hacer a lo largo de esta guía es guiarte desde la completa, total y masiva confusión, hasta la iluminación, que sólo se puede obtener, a través de pura programación en C.

En los viejos tiempos, C era un lenguaje más simple. Un buen número de las características contenidas en este libro y *muchas* de las características en el Libro de Referencias, no existían, cuando K&R escribieron la segunda famosa edición de su libro, en 1988. Sin embargo, en su esencia, el lenguaje sigue siendo pequeño, y espero presentarlo de una manera que comience con ese núcleo principal simple y se desarrolle hacia afuera.

Y ése es mi pretexto para escribir un libro tan graciosamente grande, para un lenguaje tan pequeño y conciso.

[1] https://www.ioccc.org/

## 1.1 Importante!

Esta guía asume que ya tienes algunos conocimientos de programación provenientes de otro lenguaje, como Python[2], JavaScript[3], Java[4], Rust[5], Go[6], Swift[7], etc. ¡Los desarrolladores de (Objective-C[8] lo tendrán bastante fácil!)

Vamos a suponer que sabes qué son las variables, qué hacen los bucles, cómo funcionan las funciones, etc.

Si ese no es tu caso, lo mejor que puedo ofrecerte es un entretenimiento honesto para tu placer lector. Lo único que puedo prometer, es que esta guía no terminará en un suspenso… ¿o *SI* ?

## 1.2 Cómo leer este libro

La guía está dividida en dos volúmenes, y este es el primero: ¡el volumen del tutorial!

El segundo volumen se llama biblioteca de referencia[9], y es mucho más una referencia que un tutorial.

Si eres nuevo, siga el tutorial en orden. En general, cuanto más avanzas en los capítulos, menos importante es ir en orden.

Y sin importar tu nivel de habilidad, la parte de referencia está ahí, con ejemplos completos de las llamadas a funciones de librerías estándar, para ayudarte a refrescar la memoria cuando sea necesario. Excelente lectura para acompañar un plato de cereal o en otro momento.

Finalmente, al ojear el índice (si estás leyendo la versión impresa, las entradas de la sección de referencia están en cursiva).

## 1.3 Plataforma y Compilador

Trataré de apegarme a la antigua y clásica Norma ISO C[10]. Bueno, la mayor parte. Podré alocarme un poco y empezar a hablar de POSIX[11] o algo así, pero ya veremos

Los usuarios de **Unix** (por ejemplo, Linux, BSD, etc.) intenten ejecutar `cc` o `gcc` desde la línea de comandos; es posible que ya tengan un compilador instalado. Si no, busquen en su distribución cómo instalar `gcc` o `clang` .

Los usuarios **de Windows** deberían consultar Visual Studio Community[12]. O, si está buscando una experiencia más similar a Unix (¡recomendado!), instalar WSL[13] y `gcc` .

Los usuarios **de Mac** querrán instalar XCode[14] y, en particular, las herramientas de línea de comandos.

Hay muchos compiladores por ahí, y todos ellos funcionarán para este libro. Y un compilador de C++ compilará la mayoría (¡pero no todo!) del código C. De ser posible, es mejor usar un compilador propiamente de C.

[2] https://en.wikipedia.org/wiki/Python_(programming_language)
[3] https://en.wikipedia.org/wiki/JavaScript
[4] https://en.wikipedia.org/wiki/Java_(programming_language)
[5] https://en.wikipedia.org/wiki/Rust_(programming_language)
[6] https://en.wikipedia.org/wiki/Go_(programming_language)
[7] https://en.wikipedia.org/wiki/Swift_(programming_language)
[8] https://en.wikipedia.org/wiki/Objective-C
[9] https://beej.us/guide/bgclr/
[10] https://en.wikipedia.org/wiki/ANSI_C
[11] https://en.wikipedia.org/wiki/POSIX
[12] https://visualstudio.microsoft.com/vs/community/
[13] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[14] https://developer.apple.com/xcode/

## 1.4 Página Web Oficial

La ubicación oficial de este documento es https://beej.us/guide/bgc/[15]. Tal vez esto cambie en el futuro, pero es más probable que todas las demás guías se migren desde las computadoras de Chico State.[Traductor: "Chico State" se refiere a California State University, conocida como Chico State, que es una universidad pública ubicada en Chico, California.]

## 1.5 Política de Email

Por lo general, estoy disponible para ayudar con preguntas por correo electrónico, así que no dudes en escribirme, pero no puedo garantizar una respuesta. Llevo una vida bastante ocupada y hay momentos en los que simplemente no puedo responder a una pregunta que tengas. Cuando es ese el caso, generalmente solo borro el mensaje. No es nada personal; Simplemente nunca tendré tiempo para dar la respuesta detallada que necesitas.

Como regla general, cuanto más compleja es la pregunta, menos probable es que responda. Si puedes reducir tu pregunta antes de enviarla por correo y asegurarte de incluir cualquier información pertinente (como plataforma, compilador, mensajes de error que recibes y cualquier otra cosa que creas que podría ayudarme a solucionar el problema), es mucho más probable que obtengas una respuesta.

Si no obtienes una respuesta, inténtalo un poco más, trata de encontrar la respuesta y, si aún te resulta difícil, escríbeme de nuevo con la información que encontraste y, con suerte, será suficiente para ayudarte.

Ahora que te he molestado acerca de cómo escribir y cómo no escribirme, me gustaría que sepas que aprecio *completamente* todos los elogios que ha recibido la guía a lo largo de los años. ¡Es un verdadero impulso moral y me alegra saber que se está utilizando para bien! `:-)` ¡Gracias!

## 1.6 Duplicación

Eres más que bienvenido a duplicar este sitio, ya sea pública o privadamente. Si reflejas públicamente el sitio y quieres que lo enlace desde la página principal, escríbeme a `beej@beej.us` .

## 1.7 Nota para traductores

Si deseas traducir la guía a otro idioma, escríbeme a `beej@beej.us` y enlazaré tu traducción desde la página principal. Siéntete libre de agregar tu nombre e información de contacto a la traducción.

[Traductor: Hola, soy el traductor de este maravilloso libro. Mi nombre es Stefano Alejandro Gassmann. Aquí te dejo mis datos de contacto por si encuentras algún error en la traducción. ¡Muchas gracias y espero que disfrutes tanto este libro como yo! :)]

Ten en cuenta las restricciones de licencia en la sección "Derechos de Autor y Distribución", a continuación.

## 1.8 Derechos de autor y distribución



[15] https://beej.us/guide/bgc/

Una excepción específica a la parte de la licencia "Sin obras derivadas" es la siguiente: esta guía se puede traducir libremente a cualquier idioma, siempre que la traducción sea precisa y la guía se reimprima en su totalidad. Se aplican las mismas restricciones de licencia a la traducción que a la guía original. La traducción también puede incluir el nombre y la información de contacto del traductor.

El código fuente C presentado en este documento se otorga al dominio público y está completamente libre de cualquier restricción de licencia.

Se alienta libremente a los educadores a recomendar o proporcionar copias de esta guía a sus alumnos.

Póngase en contacto `beej@beej.us` para obtener más información.

## 1.9 Dedicatoria

Las cosas más difíciles de escribir esta guía son:

- Aprender el material con suficiente detalle para poder explicarlo.
- Descubrir la mejor manera de explicarlo de forma clara, un proceso iterativo aparentemente interminable.
- Exponerme como una supuesta *autoridad*, cuando en realidad solo soy un ser humano normal que trata de encontrarle sentido a todo, como todos los demás.
- Continuar cuando tantas otras cosas me llaman la atención.

Mucha gente me ha ayudado a través de este proceso, y quiero reconocer a aquellos que han hecho posible este libro.

- Todos los usuarios de Internet que decidieron ayudar a compartir sus conocimientos de una forma u otra. El intercambio gratuito de información instructiva es lo que hace que Internet sea el gran lugar que es.
- Los voluntarios de cppreference.com[16] que proporcionan el puente que lleva de la especificación al mundo real.
- Las personas útiles y conocedoras de comp.lang.c[17] y r/C_Programming[18] que me ayudó a superar las partes más difíciles del lenguaje.
- Todos los que enviaron correcciones y los pedidos de incorporación (pull-requests), desde instrucciones confusas hasta errores tipográficos.

¡Gracias! ♥

[16] https://en.cppreference.com/
[17] https://groups.google.com/g/comp.lang.c
[18] https://www.reddit. com/r/C_Programming/

# Chapter 2

# Hello, World!

## 2.1 Qué esperar de C

> *"¿A dónde llevan estas escaleras?"*
> *"Llevan hacia arriba."*
>
> —Ray Stantz y Peter Venkman, Cazafantasmas

C es un lenguaje de bajo nivel.

No solía ser así. En aquellos días, cuando las personas tallaban tarjetas perforadas de granito, C era una manera increíble de liberarse de la tediosa tarea de usar lenguajes de bajo nivel como ensamblador[1].

Pero, en estos tiempos modernos, los lenguajes de la generación actual, ofrecen todo tipo de características que no existían cuando se inventó C (1972). Esto significa que C es un lenguaje bastante básico, con pocas características. Puede hacer *cualquier cosa*, pero puede hacerte trabajar por ello.

Entonces, ¿por qué lo usaríamos hoy en día?

- Como herramienta de aprendizage: C no es solo una pieza venerable de la historia de la informática, está conectado a nivel de hardware[2] de una manera en la que los lenguajes actuales no lo están. Cuando aprendes C, entiendes cómo el software interactúa con la memoria de la computadora a bajo nivel. No hay medidas de seguridad. Te garantizo, que escribirás software que fallará. ¡Y eso es parte de la diversión!
- Como herramienta útil: C todavía se utiliza para ciertas aplicaciones, como la construcción de sistemas operativos[3] o en sistemas embebidos[4]. (¡Aunque el lenguaje de programación Rust[5] está observando ambos campos!)

Si estás familiarizado con otro lenguaje, muchas cosas en C te resultarán fáciles. C ha inspirado muchos otros lenguajes, y verás fragmentos de él en Go, Rust, Swift, Python, JavaScript, Java y todo tipo de otros lenguajes. Esos aspectos te resultarán familiares.

Lo único que confunde a la gente en C son los *punteros*. Prácticamente todo lo demás es conocido, pero los punteros son lo peculiar. Es probable que ya conozcas el concepto detrás de los punteros, pero C te obliga a ser explícito al respecto, utilizando operadores que probablemente nunca hayas visto antes.

[1] https://en.wikipedia.org/wiki/Assembly_language
[2] https://en.wikipedia.org/wiki/Bare_machine
[3] https://en.wikipedia.org/wiki/Operating_system
[4] https://en.wikipedia.org/wiki/Embedded_system
[5] https://en.wikipedia.org/wiki/Rust_(programming_language)

Es especialmente insidioso porque una vez que *comprendes* los punteros, de repente se vuelven fáciles. Pero hasta ese momento, son como anguilas resbaladizas.

Todo lo demás en C es simplemente memorizar otra forma (¡o a veces la misma forma!) de hacer algo que ya has hecho antes. Los punteros son la parte extraña. Y, se podría argumentar, que incluso los punteros, son variaciones de un tema con el que probablemente estás familiarizado.

Así que.. prepárate! para una emocionante aventura, tan cerca del núcleo de la computadora como puedas estar, sin llegar al lenguaje ensamblador, en el lenguaje de computadora más influyente de todos los tiempos[6]. ¡Agárrate fuerte!

## 2.2 Hello, World!

Este es el ejemplo canónico de un programa en C. Todo el mundo lo utiliza. (Nota: los números a la izquierda son solo para referencia del lector y no forman parte del código fuente.)

```
/* Programa Hola Mundo */

#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");  // En realidad se hace el trabajo aquí
}
```

Vamos a ponernos nuestros guantes de goma resistentes y de manga larga, agarrar un bisturí y abrir esto para ver cómo funciona. Así que lávate bien, porque allá vamos. Cortando muuuy suavemente…

Vamos a quitar lo fácil del camino: cualquier cosa entre los dígrafos `/*` y `*/` es un comentario y será completamente ignorado por el compilador. Lo mismo ocurre con cualquier cosa en una línea después de un `//`. Esto te permite dejar mensajes para ti y para otros, de modo que cuando vuelvas y leas tu código en el futuro lejano, sabrás qué demonios estabas tratando de hacer. Créeme, lo olvidarás; Sucede.

Ahora, ¿Que es ese? `#include`? ¡QUÉ ASCO! Bueno, le dice al preprocesador de C que extraiga el contenido de otro archivo y lo inserte en el código justo *ahí*.

Espera!… ¿qué es el Preprocesador de C? Buena pregunta. Hay dos etapas[7] en la compilación: el preprocesador y el compilador. Cualquier cosa que comiense con el signo, almohadilla o "numeral" (`#`) es algo para que el preprocesador opere, antes de que la compilación siquiera comiece. Comunmente las *directivas del preprocesador*, son llamadas por `#include` y `#define`. Pero hablaremos de ello más adelante.

Antes de continuar, ¿por qué me molestaría en señalar que el signo de numeral se llama almohadilla? La respuesta es simple: creo que la palabra almohadilla es tan extremadamente divertida[Traductor: me recurda a las patitas de un gato] que tengo que difundir su nombre gratuitamente siempre que tenga la oportunidad. Almohadilla. Almohadilla, Almohadilla, Almohadilla.

Entonces, *de todas formas*. Después de que el preprocesador de C haya terminado de preprocesar todo, los resultados están listos para que el compilador los tome y produzca código ensamblador[8], código máquina[9], o lo que sea que esté a punto de hacer. El código máquina es el "lenguaje" que entiende la CPU, y lo puede entender *muy rápidamente*. Esta es una de las razones por las que los programas en C tienden a ser rápidos.

Por ahora, no te preocupes por los detalles técnicos de la compilación; solo debes saber que tu código fuente pasa por el preprocesador, la salida de eso pasa por el compilador, y luego eso produce un ejecutable para

[6] Sé que alguien me discutirá esto, pero debe estar al menos entre los tres primeros, ¿verdad?
[7] Bueno, técnicamente hay más de dos, pero bueno, finjamos que hay dos, ¿verdad?
[8] https://en.wikipedia.org/wiki/Assembly_language
[9] https://en.wikipedia.org/wiki/Machine_code

que lo ejecutes.

¿Qué hay del resto de la línea? ¿Qué es `<stdio.h>`? Eso es lo que se conoce como un *archivo de encabezado*. Es el ".h" al final lo que lo delata. De hecho, es el archivo de encabezado de "Entrada/Salida Estándar" (`stdio` : **ST**an**D**ar **I**nput/**O**utput) que llegarás a conocer y amar. Nos da acceso a un montón de funcionalidades de E/S [10]. Para nuestro programa de demostración, estamos mostrando la cadena "¡Hola, Mundo!", por lo que en particular necesitamos acceso a la función `printf()` para hacer esto. El archivo `<stdio.h>` nos proporciona este acceso. Básicamente, si intentáramos usar `printf()` sin `#include <stdio.h>`, el compilador se nos habría quejado.

¿Cómo supe que necesitaba `#include <stdio.h>`para `printf()`? Respuesta: está en la documentación. Si estás en un sistema Unix, `man 3 printf` te dirá justo al principio de la página del manual qué archivos de encabezado se requieren o consulta la sección de referencia en este libro. `:-)`

¡Santo cielo! Todo eso fue para cubrir la primera línea. Pero, seamos sinceros, ha sido completamente diseccionada. ¡No quedará ningún misterio!

Así que toma un respiro… repasa el código de muestra. Solo quedan un par de líneas fáciles.

¡Bienvenido de nuevo de tu descanso! Sé que realmente no tomaste un descanso; solo te estaba haciendo una broma.

La siguiente linea es `main()`. Esta es la definición de la función `main()`; todo lo que está entre las llaves (`{` y `}`) es parte de la definición de la función.

(¿Cómo se *llama* a una función? La respuesta está en la línea `printf()`, pero llegaremos a ella en un minuto).

La función main, es muy especial, se destaca sobre las demás ya que es la función que se llamará automáticamente cuando tu programa comienza a ejecutarse. Nada de tu código se llama antes de `main()`. En el caso de nuestro ejemplo, esto funciona bien, ya que todo lo que queremos hacer es imprimir una línea y salir.

Otra cosa: una vez que el programa se ejecute, más allá del final de `main()` y por debajo de la llave de cierre, el programa terminará y volverás a tu símbolo del sistema / Terminal / Consola.

Así que, sabemos que ese programa ha traído un fichero de cabecera, `stdio.h`, y ha declarado una función `main()` que se ejecutará cuando se inicie el programa. ¿Cuáles son las bondades de `main()`?

Me alegra mucho que lo hayas preguntado. ¡De verdad! Solo tenemos una ventaja: una llamada a la función `printf()`. Puedes darte cuenta de que esto es una llamada a una función y no una definición de función de varias maneras, pero un indicador es la falta de llaves después de ella. Y terminas la llamada a la función con un punto y coma para que el compilador sepa que es el final de la expresión. Verás que estarás poniendo puntos y comas después de casi todo.

Estás pasando un argumento a la función `printf()`: una cadena que se imprimirá cuando la llames. Oh, sí, ¡estamos llamando a una función! ¡Somos geniales! Espera, espera, no te pongas arrogante. ¿Qué es ese loco `\n` al final de la cadena? Bueno, la mayoría de los caracteres en la cadena se imprimirán tal como están almacenados. Pero hay ciertos caracteres que no se pueden imprimir bien en pantalla y que están incrustados como códigos de barra invertida de dos caracteres. Uno de los más populares es `\n` (se lee "barra invertida-N" o simplemente "nueva línea") que corresponde al carácter *nueva línea*. Este es el carácter que hace que la impresión continúe al principio de la siguiente línea, en lugar de la actual. Es como presionar enter al final de la línea.

Así que copia ese código en un archivo llamado `hello.c` y compílalo. En una plataforma similar a Unix (por ejemplo, Linux, BSD, Mac o WSL), desde la línea de comandos lo compilarás con un comando como este:

```
gcc -o hello hello.c
```

[10] Técnicamente, contiene directivas de preprocesador y prototipos de funciones (más sobre eso adelante) para necesidades comunes de entrada y salida.

(Esto significa “compilar `hello.c` y generar un ejecutable llamado `hello`”.)

Después de eso, deberías tener un archivo llamado `hello` que puedes ejecutar con este comando:

```
./hello
```

(El `./` inicial le indica al shell que “ejecute desde el directorio actual”.)

Y esto es lo que pasa:

```
Hello, World!
```

¡Está hecho y probado! ¡Envíalo!

## 2.3 Detalles de la Compilación

Hablemos un poco más sobre cómo compilar programas en C y qué sucede detrás de escena en ese proceso.

Como otros lenguajes, C tiene un *código fuente*. Sin embargo, dependiendo del lenguaje del que provengas, es posible que nunca hayas tenido que *compilar* tu código fuente en un *ejecutable*.

La compilación es el proceso de tomar código fuente en C y lo convertirlo en un programa que tu sistema operativo puede ejecutar.

Los desarrolladores de JavaScript y Python no están acostumbrados a un paso de compilación separado en absoluto, ¡aunque detrás de escena eso está sucediendo! Python compila tu código fuente en algo llamado *bytecode*, que la máquina virtual de Python puede ejecutar. Los desarrolladores de Java están acostumbrados a la compilación, pero eso produce bytecode para la Máquina Virtual de Java.

Cuando se compila en C, se genera *código máquina*. Estos son los unos y ceros que pueden ser ejecutados directa y rápidamente por la CPU.

> Los lenguajes que típicamente no se compilan se llaman lenguajes *interpretados*. Pero como mencionamos con Java y Python, también tienen un paso de compilación. Y no hay ninguna regla que diga que C no puede ser interpretado. (¡Existen intérpretes de C por ahí!) En resumen, hay muchas áreas grises. La compilación en general simplemente toma código fuente y lo convierte en otra forma más fácil de ejecutar.

El compilador de C es el programa que realiza la compilación.

Como ya mencionamos, `gcc` es un compilador que está instalado en muchos Sistemas operativos similares a Unix[11]. Y comúnmente se ejecuta desde la línea de comandos en una terminal, pero no siempre. También puedes ejecutarlo desde tu entorno de desarrollo integrado (IDE).

Entonces, ¿cómo realizamos compilaciones desde la línea de comandos?

## 2.4 Construyendo con `gcc`

Si tienes un archivo fuente llamado `hello.c` en el directorio actual, puedes compilarlo en un programa llamado `hello` con este comando que se escribe en una terminal:

```
gcc -o hello hello.c
```

[11] https://en.wikipedia.org/wiki/Unix

El `-o` significa "salida a este archivo" (Output)[12]. Y al final está `hello.c`, que es el nombre del archivo que queremos compilar.

Si tu código fuente está dividido en varios archivos, puedes compilarlos todos juntos (casi como si fueran un solo archivo, aunque las reglas son más complejas que eso) colocando todos los archivos `.c` en la línea de comandos:

```
gcc -o awesomegame ui.c characters.c npc.c items.c
```

y todos serán compilados juntos en un único ejecutable grande.

Eso es suficiente para empezar. Más adelante hablaremos sobre detalles como múltiples archivos fuente, archivos objeto y muchas otras cosas divertidas.

## 2.5 Construyendo con `clang`

En Macs, el compilador estándar no es `gcc`, es `clang`. Sin embargo, también se instala un envoltorio para que puedas ejecutar `gcc` y que funcione de todos modos.

También puedes instalar el compilador `gcc` de forma adecuada a través de Homebrew[13] u otros medios.

## 2.6 Construyendo con IDEs

Si estás utilizando un *Entorno de Desarrollo Integrado* (IDE), probablemente no necesites compilar desde la línea de comandos.

Con Visual Studio, con `CTRL-F7` puedes compilar, y con `CTRL-F5` puedes ejecutar.

Con VS Code, puedes presionar `F5` para ejecutar a través del depurador. (Tendrás que instalar la extensión C/C++ para esto).

Con Xcode, puedes compilar con `COMMAND-B` y ejecutar con `COMMAND-R`. Para obtener las herramientas de línea de comandos, busca en Google "Xcode command line tools" y encontrarás instrucciones para instalarlas.

Para comenzar, te animo también a intentar compilar desde la línea de comandos, ¡es historia!

## 2.7 Versiones de C

C ha recorrido un largo camino a lo largo de los años, y ha tenido muchas versiones numeradas para describir el dialecto del lenguaje estás utilizando.

Estos generalmente se refieren al año de la especificación.

Los más famosos son C89, C99, C11 y C2x. Nos centraremos en este último en el libro.

Pero aquí tienes una tabla más completa:

[12] Si no le proporcionas un nombre de archivo de salida, por defecto se exportará a un archivo llamado `a.out`—este nombre de archivo tiene sus raíces en la historia profunda de Unix.

[13] https://formulae.brew.sh/formula/gcc

| Version | Descripción |
|---|---|
| K&R C | En 1978, la versión original. Nombrada en honor a Brian Kernighan y Dennis Ritchie. Ritchie diseñó y codificó el lenguaje, y Kernighan coescribió el libro sobre él. Hoy en día rara vez se ve código original de K&R. Si lo ves, se verá extraño, como el inglés medio, luce extraño para los lectores de inglés moderno. |
| **C89**, ANSI C, C90 | En 1989, el American National Standards Institute (ANSI) produjo una especificación del lenguaje C que marcó el tono para C que persiste hasta hoy. Un año después, la responsabilidad pasó a la Organización Internacional de Normalización (ISO), que produjo el estándar C90, idéntico al de ANSI. |
| C95 | Una adición mencionada raramente a C89 que incluía soporte para caracteres. |
| **C99** | La primera gran revisión con muchas adiciones al lenguaje. Lo que la mayoría de la gente recuerda es la adición de los comentarios de estilo `//`. Esta es la versión más popular de C en uso hasta la fecha de esta escritura. |
| **C11** | Esta actualización mayor incluye soporte para Unicode y multi-threading. Ten en cuenta que si comienzas a usar estas características del lenguaje, podrías estar sacrificando la portabilidad en lugares que aún están usando C99. Sin embargo, honestamente, 1999 ya fue hace un tiempo. |
| C17, C18 | Actualización de corrección de errores para C11. C17 parece ser el nombre oficial, pero la publicación se retrasó hasta 2018. Según tengo entendido, ambos términos son intercambiables, prefiriéndose C17. |
| C2x | Lo que viene a continuación se espera que eventualmente se convierta en C23. |

Puedes forzar a GCC a usar uno de estos estándares con el argumento de línea de comandos `-std=`. Si quieres que sea estricto con el estándar, añade `-pedantic`

Por ejemplo:

```
gcc -std=c11 -pedantic programa.c
```

Para este libro, compilo programas para C2x con todas las advertencias activadas:

```
gcc -Wall -Wextra -std=c2x -pedantic programa.c
```

# Chapter 3

# Variables y Declaraciones

> *"¿Para hacer un mundo se necesitan de todo tipo de personas, ¿no es así, Padre?"*
> *"Así es, hijo mío, así es."*
>
> —El capitán pirata Thomas Bartholomew Red al Padre, Piratas

Puede haber muchas cosas en un programa en C.

Sí.

Y por varias razones, será más fácil para todos nosotros si clasificamos algunos de los tipos de cosas que puedes encontrar en un programa, para que podamos ser claros sobre lo que estamos hablando.

## 3.1 Variables

Se dice que "las variables contienen valores". Pero otra manera de pensarlo es que una variable es un nombre legible por humanos que se refiere a algún dato en la memoria.

Vamos a tomarnos un momento para echar un vistazo a los punteros. No te preocupes por esto.

Puedes pensar en la memoria como un gran array de bytes[1]. Los datos se almacenan en este "array"[2]. Si un número es más grande que un solo byte, se almacena en múltiples bytes. Debido a que la memoria es como un array, cada byte de memoria puede ser referido por su índice. Este índice en la memoria también se llama una *dirección*, o una *ubicación*, o un *puntero*.

Cuando tienes una variable en C, el valor de esa variable está en la memoria en *algún lugar*, en alguna dirección. Por supuesto. Después de todo, ¿dónde más estaría? Pero es un dolor referirse a un valor por su dirección numérica, así que le damos un nombre en su lugar, y eso es lo que es una variable.

La razón por la que menciono todo esto es doble:

1. Va a hacer que sea más fácil entender las variables de puntero más adelante: ¡son variables que contienen la dirección de otras variables!
2. También va a hacer que sea más fácil entender los punteros más adelante.

Así que una variable es un nombre para algunos datos que están almacenados en la memoria en alguna dirección.

[1] Un "byte" es típicamente un número binario de 8 bits. Piensa en ello como un entero que solo puede contener valores del 0 al 255, inclusive. Técnicamente, C permite que los bytes sean de cualquier número de bits y si quieres referirte inequívocamente a un número de 8 bits, deberías usar el término *octeto*. Pero los programadores asumirán que te refieres a 8 bits cuando dices "byte" a menos que especifiques lo contrario.

[2] Estoy simplificando mucho cómo funciona la memoria moderna aquí. Pero el modelo mental funciona, así que por favor perdóname.

### 3.1.1 Nombres de las variables

Puedes usar cualquier carácter en el rango 0-9, A-Z, a-z, y guión bajo para los nombres de variables, con las siguientes reglas:

- No puedes empezar una variable con un dígito del 0-9.
- No puedes empezar un nombre de variable con dos guiones bajos.
- No puedes empezar un nombre de variable con un guión bajo seguido de una letra mayúscula de la A-Z.

Para Unicode, simplemente pruébalo. Hay algunas reglas en la especificación en §D.2 que hablan sobre qué rangos de puntos de código Unicode están permitidos en qué partes de los identificadores, pero eso es demasiado para escribir aquí y probablemente sea algo en lo que nunca tendrás que pensar de todas formas.

### 3.1.2 Tipos de variables

Dependiendo de los lenguajes que ya tengas en tu repertorio, puedes estar familiarizado o no con la idea de los tipos. Pero C es bastante estricto con ellos, así que deberíamos hacer un repaso.

Algunos ejemplos de tipos, de los más básicos:

| Tipo | Ejemplo | Tipo en C |
|---|---|---|
| Entero | `3490` | `int` |
| Punto flotante | `3.14159` | `float`[3] |
| Caracter (uno solo) | `'c'` | `char` |
| Texto | `"Hola, mundo!"` | `char *`[4] |

C hace un esfuerzo por convertir automáticamente entre la mayoría de los tipos numéricos cuando se lo pides, pero, aparte de eso, todas las conversiones son manuales, en particular entre cadenas y números.

Casi todos los tipos en C son variantes de estos tipos básicos.

Antes de poder usar una variable, debes *declarar* esa variable y decirle a C qué tipo de datos contiene. Una vez declarada, el tipo de variable no puede cambiarse más tarde durante la ejecución. Lo que estableces es lo que es, hasta que salga de alcance y sea reabsorbida en el universo.

Tomemos nuestro código anterior de “Hola, mundo” y agreguemos un par de variables a él:

```
#include <stdio.h>

int main(void)
{
    int i;    // Almacena enteros con signo, por ejemplo, -3, -2, 0, 1, 10.
   float f;  // Almacena números de punto flotante con signo, por ejemplo, -3.1416.

    printf("Hello, World!\n");  // Ah, bendita familiaridad
}
```

¡Listo! Hemos declarado un par de variables. Aún no las hemos usado y ambas están sin inicializar. Una contiene un número entero y la otra contiene un número de punto flotante (un número real, si tienes conocimientos de matemáticas).

[3] Estoy siendo un poco impreciso aquí. Técnicamente, `3.14159` es del tipo `double`, pero aún no hemos llegado allí y quiero que asocies `float` con “Punto Flotante”, y C convertirá ese tipo felizmente en un `float`. En resumen, no te preocupes por ello hasta más adelante.

[4] Lee esto como “puntero a un char” o “char pointer”. “Char” por carácter. Aunque no puedo encontrar un estudio, parece anecdóticamente que la mayoría de las personas pronuncian esto como “char”, una minoría dice “car”, y algunos pocos dicen “care”. Hablaremos más sobre los punteros más adelante.

Las variables no inicializadas tienen un valor indeterminado[5]. Deben ser inicializadas o de lo contrario debes asumir que contienen algún número absurdo.

> Este es uno de los puntos donde C puede "atraparte". En mi experiencia, la mayor parte del tiempo, el valor indeterminado es cero… ¡pero puede variar de ejecución en ejecución! Nunca asumas que el valor será cero, incluso si ves que lo es. *Siempre* inicializa explícitamente las variables a algún valor antes de usarlas[a].
>
> [a] Esto no es estrictamente 100% cierto. Cuando aprendamos sobre la duración de almacenamiento estática, descubrirás que algunas variables se inicializan automáticamente a cero. Pero lo seguro es siempre inicializarlas.

¿Qué es esto? ¿Quieres almacenar algunos números en esas variables? ¡Locura!

Vamos a hacer eso ahora mismo:

```
int main(void)
{
    int i;

    i = 2; // Asigna el valor 2 dentro de la variable i

    printf("Hello, World!\n");
}
```

Perfecto. Hemos almacenado un valor. Vamos a imprimirlo.

Lo vamos a hacer pasando dos argumentos asombrosos a la función `printf()`. El primer argumento es una cadena que describe qué imprimir y cómo imprimirlo (llamado *cadena de formato* [*format string*]), y el segundo es el valor a imprimir, es decir, lo que sea que esté en la variable `i`.

`printf()` busca a través de la cadena de formato en busca de diversas secuencias especiales que comienzan con un signo de porcentaje (`%`) y que le indican qué imprimir. Por ejemplo, si encuentra `%d`, busca el siguiente parámetro que se le pasó y lo imprime como un entero. Si encuentra `%s`, imprime el valor como un flotante. Si encuentra `%s`, imprime una cadena.

De esta manera, podemos imprimir el valor de varios tipos de la siguiente manera:

```
#include <stdio.h>

int main(void)
{
    int i = 2;
    float f = 3.14;
   char *s = "Hello, world!";  // char * ("Puntero a caracter") es del tipo String

    printf("%s  i = %d y f = %f!\n", s, i, f);
}
```

Y la salida será:

```
Hello, world!  i = 2 y f = 3.14!
```

De esta manera, `printf()` podría ser similar a varios tipos de cadenas de formato o cadenas parametrizadas en otros lenguajes con los que estás familiarizado.

[5] Coloquialmente, decimos que tienen valores "aleatorios", pero no son realmente—ni siquiera pseudo-realmente—números aleatorios.

### 3.1.3 Tipo booleano

¿C tiene tipos booleanos, verdadero o falso?

`1`!

Históricamente, C no tenía un tipo booleano, y algunos podrían argumentar que todavía no lo tiene.

En C, `0` significa "falso", y cualquier valor no-cero significa "verdadero".

Entonces `1` es verdadero. Y `-37` es verdadero. Y `0` es falso.

Puedes simplemente declarar tipos booleanos como `int`:

```
int x = 1;

if (x) {
    printf("x es verdadero!\n");
}
```

Si incluyes `#include <stdbool.h>`, también obtienes acceso a algunos nombres simbólicos que podrían hacer que las cosas parezcan más familiares, específicamente un tipo `bool` y los valores `true` y `false`:

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

int main(void) {
    bool x = true;

    if (x) {
        printf("x es verdadero!\n");
    }
}
```

Pero estos son idénticos a usar valores enteros para verdadero y falso. Son solo una fachada para que las cosas luzcan bien.

## 3.2 Operadores y Expresiones

Los operadores de C deberían resultarte familiares si has trabajado con otros lenguajes. Vamos a repasar algunos de ellos aquí.

(Hay muchos más detalles que estos, pero vamos a cubrir lo suficiente en esta sección para comenzar).

### 3.2.1 Operadores aritméticos

Espero que estos te resulten familiares:

```
i = i + 3;  // Operadores de adición (+) y asignación (=), suma 3 a i
i = i - 8;  // Resta, resta 8 a i
i = i * 9;  // Multiplicación
i = i / 2;  // División
i = i % 5;  // Módulo (resto de la división)
```

Hay variantes abreviadas para todo lo anterior. Cada una de esas líneas podría escribirse de manera más concisa como:

```
i += 3;  // Es igual que "i = i + 3", Suma 3 a i
i -= 8;  // Es igual que "i = i - 8"
i *= 9;  // Es igual que "i = i * 9"
i /= 2;  // Es igual que "i = i / 2"
i %= 5;  // Es igual que "i = i % 5"
```

No hay operador de exponenciación en C. Tendrás que usar una de las variantes de la función `pow()` de `math.h`.

¡Vamos a adentrarnos en algunas cosas más extrañas que es posible que no encuentres en tus otros lenguajes!

### 3.2.2 Operador ternario

C también incluye el *operador ternario*. Se trata de una expresión cuyo valor depende del resultado de una condición incluida en ella.

```
// Si x > 10, suma 17 a y. De lo contrario suma 37 a y.

y += x > 10? 17: 37;
```

¡Qué lío! Te acostumbrarás a medida que lo leas. Para ayudar un poco, reescribiré la expresión anterior usando declaraciones `if`:

```
// Esta expresión

y += x > 10? 17: 37;

// es equivalente a esta no-expresión:

if (x > 10)
    y += 17;
else
    y += 37;
```

Compara esos dos hasta que veas cada uno de los componentes del operador ternario.

Otro ejemplo, el cual imprime si un número almacenado en `x` es par o impar sería:

```
printf("El número %d es %s.\n", x, x % 2 == 0? "Par :)": "Impar :(");
```

El especificador de formato `%s` en `printf()` significa imprimir una cadena. Si la expresión `x % 2` se evalúa a `0`, el valor de toda la expresión ternaria se evalúa a la cadena `"Par"`. De lo contrario, se evalúa a la cadena `"Impar"`. ¡Bastante genial!

Es importante señalar, que el operador ternario no es de control de flujo, como lo es la declaración `if` . Es simplemente una expresión que se evalúa a un valor.

### 3.2.3 Pre-y-Post Incremento-y-Decremento

Ahora, vamos a *jugar* con otra cosa que quizás no hayas visto.

Estos son los legendarios operadores de post-incremento y post-decremento:

```
i++;        // Suma uno a i (post-incremento)
i--;        // Resta uno a i (post-decremento)
```

Muy comúnmente, estos se utilizan simplemente como versiones más cortas de:

```
i += 1;        // Suma 1 a i
i -= 1;        // Resta 1 a i
```

pero los astutos bribones son un poco más sutilmente diferentes que eso.

Echemos un vistazo a esta variante, pre-incremento y pre-decremento:

```
++i;        // Suma 1 a i (pre-incremento)
--i;        // Resta 1 a i (pre-decremento)
```

Con el pre-incremento y pre-decremento, el valor de la variable se incrementa o decrementa *antes* de evaluar la expresión. Luego, la expresión se evalúa con el nuevo valor.

Con el post-incremento y post-decremento, primero se calcula el valor de la expresión con el valor actual, y *luego* se incrementa o decrementa el valor después de que se haya determinado el valor de la expresión.

De hecho, puedes incluirlos en expresiones, así:

```
i = 10;
j = 5 + i++;  // Calcula 5 + i, _luego_ incrementa i

printf("%d, %d\n", i, j);  // Imprime 11, 15
```

Comparemos esto con el operador de pre-incremento:

```
i = 10;
j = 5 + ++i;  // Incrementa i, _luego_ calcula 5 + i

printf("%d, %d\n", i, j);  // Imprime 11, 16
```

Esta técnica se usa frecuentemente con el acceso y la manipulación de arreglos y punteros. Te da una manera de usar el valor en una variable y también incrementar o decrementar ese valor antes o después de que se use.

Pero, con mucho, el lugar más común donde verás esto es en un bucle `for`:

```
for (i = 0; i < 10; i++)
    printf("i es %d\n", i);
```

Pero hablaremos más sobre eso más adelante.

### 3.2.4 El operador coma

Esta es una forma poco común de separar expresiones que se ejecutarán de izquierda a derecha:

```
x = 10, y = 20;  // Primero asigna 10 a x, luego 20 a y
```

Parece un poco tonto, ¿verdad? Porque podrías simplemente reemplazar la coma, con un punto y coma, ¿no es así?

```
x = 10; y = 20;  // Primero asigna 10 a x, luego 20 a y
```

Pero eso es un poco diferente. El segundo caso son dos expresiones separadas, mientras que el primero es una sola expresión.

Con el operador coma, el valor de la expresión coma es el valor de la expresión más a la derecha:

```
x = (1, 2, 3);

printf("x is %d\n", x);  // Imprime 3, porque 3 es el que está más a la derecha
                         // de la lista separada por comas
```

Pero incluso eso es bastante forzado. Un lugar común donde se usa el operador coma es en los bucles `for` para hacer múltiples cosas en cada sección de la declaración:

```
for (i = 0, j = 10; i < 100; i++, j++)
    printf("%d, %d\n", i, j);
```

Volveremos sobre eso más tarde.

### 3.2.5 Operadores condicionales

Para valores booleanos, tenemos una serie de operadores estándar:

```
A == B;  // Verdadero si A es equivalente a B
A != B;  // Verdadero si A no es equivalente a B
A < B;   // Verdadero si A es menor que B
A > B;   // Verdadero si A es más grande que B
A <= B;  // Verdadero si A es menor o igual que B
A >= B;  // Verdadero si A es mayor o igual que B
```

¡No mezcles la asignación `=` con la comparación `==`! Usa dos signos iguales para comparar y uno para asignar.

Podemos usar las expresiones de comparación con declaraciones `if`:

```
if (a <= 10)
    printf("Exito!\n");
```

### 3.2.6 Operadores Booleanos

Podemos encadenar o combinar expresiones condicionales con operadores booleanos para *y* (and), *o* (or), y *no* (not).

| Operador | Significado booleano |
|---|---|
| `&&` | Y (and) |
| `\|\|` | O (or) |
| `!` | Negación (not) |

Un ejemplo de “y” booleano (and):

```
// Haz algo si 'x' es menor que 10 e 'y' es mayor que 20:

if (x < 10 && y > 20)
    printf("¡Haciendo algo!\n");
```

Un ejemplo de "no" booleano (not):

```
if (!(x < 12))
    printf("x no es menor que 12"\n");
```

`!` tiene mayor precedencia que los otros operadores booleanos, por lo que debemos usar paréntesis en ese caso.

Por supuesto, eso es simplemente lo mismo que:

```
if (x >= 12)
    printf("x no es menor que 12\n");
```

Pero necesitaba el ejemplo!

### 3.2.7 El operador `sizeof`

Este operador le indica el tamaño (en bytes) que una variable o tipo de datos en particular utiliza en la memoria.

Más particularmente, le indica el tamaño (en bytes) que *el tipo de una expresión particular* (que podría ser solo una variable) usa en la memoria.

Esto puede ser diferente en distintos sistemas, a excepción de `char` (que siempre son de 1 byte).

Y puede que esto no parezca muy útil ahora, pero haremos referencias a ello aquí y allá, por lo que vale la pena cubrirlo.

Dado que esto calcula la cantidad de bytes necesarios para almacenar un tipo, se podría pensar que devolvería un `int`. O… dado que el tamaño no puede ser negativo, tal vez un `unsigned`?

Pero resulta que C tiene un tipo especial para representar el valor de retorno de `sizeof`. Es `size_t`, pronunciado "*size tee*"[6]. Todo lo que sabemos es que es un tipo de entero sin signo que puede contener el tamaño en bytes de cualquier cosa que le des a `sizeof`.

`size_t` aparece en muchos lugares diferentes donde se pasan o devuelven conteos de cosas. Piénsalo como un valor que representa un conteo.

Puedes tomar el `sizeof` de una variable o expresión:

```
int a = 999;

// %zu es el especificador de formato para el tipo size_t

printf("%zu\n", sizeof a);      // Imprime 4 en mi sistema
printf("%zu\n", sizeof(2 + 7)); // Imprime 4 en mi sistema
printf("%zu\n", sizeof 3.14);   // Imprime 8 en mi sistema

// Si necesitas imprimir valores negativos de size_t, usa %zd
```

[6] El `_t` es abreviatura de `type` (Tipo).

Recuerda: es el tamaño en bytes del *tipo* de la expresión, no el tamaño de la expresión en sí. Es por eso que el tamaño de `2+7` es el mismo que el tamaño de `a`—ambos son de tipo `int`. Revisaremos este número `4` en el próximo bloque de código…

…Donde veremos que puedes obtener el `sizeof` de un tipo (nota que los paréntesis son requeridos alrededor del nombre de un tipo, a diferencia de una expresión):

```
printf("%zu\n", sizeof(int));   // Imprime 4 en mi sistema
printf("%zu\n", sizeof(char));  // Imprime 1 en todos los sistemas
```

Es importante tener en cuenta que `sizeof` es una operación en *tiempo de compilación* [7]. El resultado de la expresión se determina completamente en tiempo de compilación, no en tiempo de ejecución.

Más adelante haremos uso de esto.

## 3.3 Control de flujo

Los booleanos son buenos, pero por supuesto, no llegaríamos a ninguna parte si no pudiéramos controlar el flujo del programa. Echemos un vistazo a varios constructos: `if`, `for`,`while`, y `do-while`.

Primero, una nota general anticipada sobre declaraciones y bloques de declaraciones, cortesía de tu amigable desarrollador de C local:

Después de algo como una declaración `if` o `while`, puedes colocar ya sea una sola declaración que se ejecutará, o un bloque de declaraciones que se ejecutarán en secuencia.

Comencemos con una sola declaración:

```
if (x == 10) printf("x es 10\n");
```

Esto también a veces se escribe en una línea separada. (Los espacios en blanco son en gran medida irrelevantes en C—no es como en Python.)

```
if (x == 10)
    printf("x es 10\n");
```

Pero ¿qué pasa si quieres que ocurran varias cosas debido a la condición? Puedes usar llaves para marcar un *bloque* o *declaración compuesta*.

```
if (x == 10) {
    printf("x es 10\n");
    printf("Y también esto ocurre cuando x es 10\n");
}
```

Es un estilo muy común *siempre* usar llaves incluso si no son necesarias:

```
if (x == 10) {
    printf("x es 10\n");
}
```

Algunos desarrolladores sienten que el código es más fácil de leer y evita errores como este, donde visualmente parece que las cosas están dentro del bloque `if`, pero en realidad no lo están.

[7] Excepto con arreglos de longitud variable—pero eso es una historia para otro momento.

```
// EJEMPLO DE ERROR GRAVE

if (x == 10)
    printf("Esto sucede si x es 10\n");
    printf("Esto sucede SIEMPRE\n"); // ¡Sorpresa! ¡Incondicional!
```

`while`, `for` y los demás constructos de bucles funcionan de la misma manera que los ejemplos anteriores. Si deseas hacer múltiples cosas en un bucle o después de un `if`, envuélvelas en llaves.

En otras palabras, el `if` ejecutará lo que esté después de él. Y eso puede ser una sola declaración o un bloque de declaraciones.

### 3.3.1 El estado `if`-`else`

Ya hemos estado usando `if` en varios ejemplos, ya que es probable que lo hayas visto en algún lenguaje antes, pero aquí tienes otro:

```
int i = 10;

if (i > 10) {
    printf("Sí, i es mayor que 10.\n");
    printf("Y esto también se imprimirá si i es mayor que 10.\n");
}

if (i <= 10) printf("i es menor o igual que 10.\n");
```

En el código de ejemplo, el mensaje se imprimirá si `i` es mayor que 10, de lo contrario, la ejecución continúa en la siguiente línea. Observa las llaves después de la instrucción `if`; si la condición es verdadera, se ejecutará la primera instrucción o expresión justo después del if, o bien, se ejecutará el conjunto de código dentro de las llaves después del if. Este tipo de comportamiento de *bloque de código* es común en todas las instrucciones.

Por supuesto, dado que C es divertido de esta manera, también puedes hacer algo si la condición es `else` con una cláusula `else` en tu `if`:

```
int i = 99;

if (i == 10)
    printf("i es 10!\n");
else {
    printf("i definitivamente no es 10.\n");
    printf("Que, francamente, me irrita un poco.\n");
}
```

Y puedes incluso encadenar estos para probar una variedad de condiciones, como esto:

```
int i = 99;

if (i == 10)
    printf("i es 10!\n");

else if (i == 20)
    printf("i es 20!\n");
```

```
else if (i == 99) {
    printf("i es 99! Mi favorito\n");
    printf("No puedo decirte lo feliz que estoy.\n");
    printf("En serio.\n");
}

else
    printf("i es algún número loco que nunca he escuchado antes.\n");
```

Si vas por ese camino, asegúrate de revisar la declaración `switch` para una solución potencialmente mejor. La única limitación es que `switch` solo funciona con comparaciones de igualdad con números constantes. La cascada `if`-`else` anterior podría verificar desigualdades, rangos, variables o cualquier otra cosa que puedas crear en una expresión condicional.

### 3.3.2 La declaración `while`

La declaración `while` es simplemente un bucle promedio y corriente. Realiza una acción mientras una expresión de condición sea verdadera.

¡Hagamos uno!

```
// Imprime la siguiente salida:
//
// i es ahora 0!
// i es ahora 1!
// [ más de lo mismo entre 2 y 7 ]
// i es ahora 8!
// i es ahora 9!

int i = 0;

while (i < 10) {
    printf("i es ahora %d!\n", i);
    i++;
}

printf("¡Todo hecho!\n");
```

Así se obtiene un bucle básico. C también tiene un bucle `for` que habría sido más limpio para ese ejemplo.

Un uso no poco común de `while` es para bucles infinitos donde se repite mientras es verdadero:

```
while (1) {
    printf("1 es siempre cierto, así que esto se repite para siempre.\n");
}
```

### 3.3.3 La sentencia `do-while`

Ahora que ya tenemos la sentencia `while` bajo control, echemos un vistazo a su prima, `do-while`.

Básicamente son lo mismo, excepto que si la condición del bucle es falsa en el primer paso, `do-while` se ejecutará una vez, pero `while` no se ejecutará en absoluto. En otras palabras, la prueba para ver si se debe ejecutar o no el bloque ocurre al *final* del bloque con `do-while`. Ocurre al *principio* del bloque con `while`.

Veámoslo con un ejemplo:

```
// Utilizar una sentencia while:

i = 10;

// Esto no se ejecuta, porque i no es menor que 10:
while(i < 10) {
    printf("while: i es %d\n", i);
    i++;
}

// Utilizar una sentencia do-while:

i = 10;

// Se ejecuta una vez, porque la condición del bucle no se comprueba hasta
// después de que se ejecute el cuerpo del bucle:

do {
    printf("do-while: i es %d\n", i);
    i++;
} while (i < 10);

printf("¡Todo hecho!\n");
```

Observa que en ambos casos, la condición del bucle es falsa inmediatamente. Así que en el `while`, el bucle falla, y el siguiente bloque de código nunca se ejecuta. Con el `do-while`, sin embargo, la condición se comprueba *después* de que se ejecute el bloque de código, por lo que siempre se ejecuta al menos una vez. En este caso, imprime el mensaje, incrementa `i`, falla la condición y continúa con la salida "¡Todo hecho!

La moraleja de la historia es la siguiente: si quieres que el bucle se ejecute al menos una vez, sin importar la condición del bucle, usa `do-while`.

Todos estos ejemplos podrían haberse hecho mejor con un bucle `for`. Hagamos algo menos determinista: ¡repetir hasta que salga un cierto número aleatorio!

```
#include <stdio.h>   // Para printf
#include <stdlib.h>  // Para rand

int main(void)
{
    int r;

    do {
        r = rand() % 100; // Obtener un número aleatorio entre 0 y 99
        printf("%d\n", r);
    } while (r != 37);    // Repetir hasta que aparezca 37
}
```

Nota al margen: ¿lo has hecho más de una vez? Si lo hiciste, ¿te diste cuenta de que volvió a aparecer la misma secuencia de números? Y otra vez. ¿Y otra vez? Esto se debe a que `rand()` es un generador de números pseudoaleatorios que debe ser *sembrado* con un número diferente para generar una secuencia

diferente. Busque el `srand()`[8] para más detalles.

### 3.3.4 La sentencia 'for

¡Bienvenido a uno de los bucles más populares del mundo! ¡El bucle `for`!

Este es un gran bucle si sabes de antemano el número de veces que quieres hacer el bucle.

Podrías hacer lo mismo usando sólo un bucle `while`, pero el bucle `for` puede ayudar a mantener el código más limpio.

Aquí hay dos trozos de código equivalente—note cómo el bucle `for` es sólo una representación más compacta:

```
// Imprime los números entre 0 y 9, ambos inclusive...

// Usando una sentencia while:

i = 0;
while (i < 10) {
    printf("i es %d\n", i);
    i++;
}

//Haz exactamente lo mismo con un bucle for:

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

Así es, hacen exactamente lo mismo. Pero puedes ver cómo la sentencia `for` es un poco compacta y agradable a la vista. (Los usuarios de JavaScript apreciarán plenamente sus orígenes en C en este punto).

Está dividida en tres partes, separadas por punto y coma. La primera es la inicialización, la segunda es la condición del bucle, y la tercera es lo que debe ocurrir al final del bloque si la condición del bucle es verdadera. Estas tres partes son opcionales.

```
for (inicializar cosas; bucle si esto es cierto; hacer esto después de cada bucle)
```

Tenga en cuenta que el bucle no se ejecutará ni una sola vez si la condición del bucle comienza siendo falsa.

> **Curiosidad del bucle 'for**
>
> Puedes usar el operador coma para hacer múltiples cosas en cada cláusula del bucle `for`.
>
> ```
> for (i = 0, j = 999; i < 10; i++, j--) {
>     printf("%d, %d\n", i, j);
> }
> ```

Un `for` vacío se ejecutará eternamente:

```
for(;;) {  // "for-ever" (para-siempre)
    printf("Imprimiré esto una y otra y otra vez\n" );
    printf("Por toda la eternidad hasta la muerte por calor del universo.\n");
```

[8] https://beej.us/guide/bgclr/html/split/stdlib.html#man-srand

```
    printf("O hasta que pulses CTRL-C.\n");
}
```

### 3.3.5 Declaración `switch`

Dependiendo del lenguaje del que vengas, puede que estés o no familiarizado con `switch`, o incluso puede que la versión de C sea más restrictiva de lo que estás acostumbrado. Esta es una sentencia que te permite tomar una variedad de acciones dependiendo del valor de una expresión entera.

Básicamente, evalúa una expresión a un valor entero, salta al `case` que corresponde a ese valor. La ejecución se reanuda desde ese punto. Si se encuentra una sentencia `break`, la ejecución salta fuera del `switch`.

He aquí un ejemplo en el que, para un número determinado de cabras, imprimimos una intuición de cuántas cabras son.

```
#include <stdio.h>

int main(void)
{
    int goat_count = 2; // goat_count = contador de cabras

    switch (goat_count) {
        case 0:
            printf("No tienes cabras :(\n");
            break;

        case 1:
            printf("Solo tienes una cabra\n");
            break;

        case 2:
            printf("Tienes un par de cabras\n");
            break;

        default:
            printf("¡Tienes una gran cantidad de cabras!\n");
            break;
    }
}
```

En ese ejemplo, el `switch` saltará al `case 2` y ejecutará desde allí. Cuando (si) llega a un `break`, salta fuera del `switch`.

Además, puede que veas la etiqueta `default` en la parte inferior. Esto es lo que ocurre cuando ningún caso coincide.

Cada `case`, incluyendo `default`, es opcional. Y pueden ocurrir en cualquier orden, pero es realmente típico que `default`, si lo hay, aparezca último.

Así que todo actúa como una cascada `if`-`else`:

```
if (goat_count == 0)
    printf("No tienes cabras\n");
else if (goat_count == 1)
```

```
    printf("Tienes solo una cabra.\n");
else if (goat_count == 2)
    printf("Tienes un par de cabras.\n");
else
    printf("Tienes una gran cantidad de cabras!\n");
```

Con algunas diferencias clave:

- A menudo, `switch` es más rápido para saltar al código correcto (aunque la especificación no lo garantiza).
- `if-else` puede hacer cosas como condicionales relacionales como `<` y `>=` y punto flotante y otros tipos, mientras que `switch` no puede.

Hay una cosa más sobre switch que a veces se ve y que es bastante interesante: *falla de salida* (*fall through*).

¿Recuerdas que `break` nos hace saltar fuera del switch?

Bueno, ¿qué pasa si *no se utiliza* `break`?

¡Resulta que seguimos con el siguiente `case`! ¡Demo!

```
switch (x) {
    case 1:
        printf("1\n");
        // Falla el salto! Sigue ejecutando!
    case 2:
        printf("2\n");
        break;
    case 3:
        printf("3\n");
        break;
}
```

Si `x == 1`, el `switch` irá primero al `caso 1`, imprimirá el `1`, pero luego simplemente continúa con la siguiente línea de código… ¡que imprime `2`!

Y entonces, por fin, llegamos a un `break` así que saltamos del `switch`.

si `x == 2`, entonces simplemente entramos dentro del `case 2`, imprimimos `2`, y `break` como es normal.

Al no tener un `break` se *falla la salida*.

Consejo de experto: *Siempre* ponga un comentario en el código en el que tiene intención de fallar la salida, como he hecho yo más arriba. Evitará que otros programadores se pregunten si realmente querías hacer eso.

De hecho, este es uno de los lugares comunes para introducir errores en los programas en C: olvidar poner un `break` en tu `case`. Tienes que hacerlo si no quieres simplemente pasar al siguiente caso[9].

Antes dije que `switch` funciona con tipos enteros– mantenlo así. No utilices tipos de coma flotante o cadenas. Una laguna legal aquí es que puedes usar tipos de caracteres porque estos son secretamente números enteros. Por lo tanto, esto es perfectamente aceptable:

```
char c = 'b';
```

[9]Esto se consideró tal peligro que los diseñadores del Lenguaje de Programación Go hicieron `break` por defecto; tienes que usar explícitamente la sentencia `fallthrough (fallar la salida)` de Go si quieres pasar al siguiente caso

```
switch (c) {
    case 'a':
        printf("La letra es 'a'!\n");
        break;

    case 'b':
        printf("La letra es 'b'!\n");
        break;

    case 'c':
        printf("La letra es 'c'!\n");
        break;
}
```

Finalmente, puedes usar `enum` en `switch` ya que también son tipos enteros. Pero más sobre esto en el capítulo `enum`.

# Chapter 4

# Funciones

> "Señor, no en un ambiente como éste. Por eso también he sido programado para más de treinta funciones secundarias que…"_>
>
> —C3PO, antes de ser interrumpido bruscamente, informando de un número ya poco impresionante de funciones adicionales, *Star Wars* script

Muy parecido a otros lenguajes a los que estás acostumbrado, C tiene el concepto de *funciones*.

Las funciones pueden aceptar una variedad de *argumentos* y devolver un valor. Sin embargo, hay algo importante: los tipos de argumentos y valores de retorno están predeclarados,—¡porque así lo prefiere C!

Veamos una función. Esta es una función que toma un `int` como argumento, y devuelve un `int`.

```
#include <stdio.h>

int plus_one(int n)  // La "Definición"
{
    return n + 1;
}
```

El `int` antes del `plus_one` indica el tipo de retorno.

El `int n` indica que esta función toma un argumento `int`, almacenado en el *parámetro* `n`. Un parámetro es un tipo especial de variable local en la que se copian los argumentos.

Voy a insistir en que los argumentos se copian en los parámetros. Muchas cosas en C son más fáciles de entender si sabes que el parámetro es una *copia* del argumento, no el argumento en sí. Más sobre esto en un minuto.

Continuando el programa hasta `main()`, podemos ver la llamada a la función, donde asignamos el valor de retorno a la variable local `j`:

```
int main(void)
{
    int i = 10, j;

    j = plus_one(i);  // La "llamada"

    printf("i + 1 es %d\n", j);
```

```
}
```

> Antes de que se me olvide, fíjate en que he definido la función antes de usarla. Si no lo hubiera hecho, el compilador aún no la conocería al compilar `main()` y habría dado un error de llamada a función desconocida. Hay una forma más adecuada de hacer el código anterior con *prototipos de función*, pero hablaremos de eso más adelante.

Observa también que `main()` ¡es una función!

Devuelve un `int`.

¿Pero qué es eso de `void`? Es una palabra clave para indicar que la función no acepta argumentos.

También puede devolver `void` para indicar que no devuelve ningún valor:

```
#include <stdio.h>

// Esta función no toma argumentos y no devuelve ningún valor:

void hello(void)
{
    printf("Hello, world!\n");
}

int main(void)
{
    hello();  // Imprime "Hello, world!"
}
```

## 4.1 Transmisión por valor

He mencionado antes que cuando pasas un argumento a una función, se hace una copia de ese argumento y se almacena en el parámetro correspondiente.

Si el argumento es una variable, se hace una copia del valor de esa variable y se almacena en el parámetro.

De forma más general, se evalúa toda la expresión del argumento y se determina su valor. Ese valor se copia en el parámetro.

En cualquier caso, el valor del parámetro es algo propio. Es independiente de los valores o variables que hayas utilizado como argumentos al llamar a la función.

Veamos un ejemplo. Estúdielo y vea si puede determinar la salida antes de ejecutarlo:

```
#include <stdio.h>

void increment(int a)
{
    a++;
}

int main(void)
{
    int i = 10;
```

```
    increment(i);

    printf("i == %d\n", i);  // ¿Qué imprime esto?
}
```

A primera vista, parece que `i` es `10`, y lo pasamos a la función `increment()`. Allí el valor se incrementa, así que cuando lo imprimimos, debe ser `11`, ¿no?

> "Acostúmbrate a la decepción."
>
> —El temible pirata Roberts, La princesa prometida

Pero no es `11`… ¡imprime `10`! ¿Cómo?

Se trata de que las expresiones que pasas a las funciones se *copian* en sus parámetros correspondientes. El parámetro es una copia, no el original

Así que `i` es `10` en `main()`. Y se lo pasamos a `increment()`. El parámetro correspondiente se llama `a` en esa función.

Y la copia ocurre, como si fuera una asignación. Más o menos, `a = i`. Así que en ese punto, `a` es `10`. Y en `main()`, `i` es también `10`.

Entonces incrementamos `a` a `11`. ¡Pero no estamos tocando `i` en absoluto! Sigue siendo `10`.

Finalmente, la función está completa. Todas sus variables locales se descartan (¡adiós, `a`!) y volvemos a `main()`, donde `i` sigue siendo `10`.

Y lo imprimimos, obteniendo `10`, y hemos terminado.

Por eso en el ejemplo anterior con la función `plus_one()`, `devolvíamos` el valor modificado localmente para poder verlo de nuevo en `main()`.

Parece un poco restrictivo, ¿no? Como si sólo pudieras recuperar un dato de una función, es lo que estás pensando. Hay, sin embargo, otra forma de recuperar datos; la gente de C lo llama *pasar por referencia* y esa es una historia que contaremos en otra ocasión.

Pero ningún nombre rimbombante te distraerá del hecho de que *TODO* lo que pasas a una función *SIN EXCEPCIÓN* se copia en su parámetro correspondiente, y la función opera sobre esa copia local, *NO IMPORTA QUÉ*. Recuérdalo, incluso cuando estemos hablando del llamado paso por referencia.

## 4.2 Prototipos de funciones

Así que si recuerdas en la edad de hielo hace unas secciones, mencioné que tenías que definir la función antes de usarla, de lo contrario el compilador no lo sabría de antemano, y bombardearía con un error.

Esto no es estrictamente cierto. Puedes notificar al compilador por adelantado que vas a utilizar una función de un tipo determinado que tiene una lista de parámetros determinada. De esta forma, la función puede definirse en cualquier lugar (incluso en un fichero diferente), siempre que el *prototipo de función* haya sido declarado antes de llamar a esa función.

Afortunadamente, el prototipo de función es realmente sencillo. Es simplemente una copia de la primera línea de la definición de la función con un punto y coma al final. Por ejemplo, este código llama a una función que se define más tarde, porque primero se ha declarado un prototipo:

```
#include <stdio.h>

int foo(void);  // Esto es el prototipo!
```

```
int main(void)
{
    int i;

    // Podemos llamar aquí a foo() antes de su definición porque el
    // prototipo ya ha sido declarado, ¡arriba!

    i = foo();

    printf("%d\n", i);  // 3490
}

int foo(void)  // ¡Esta es la definición, igual que el prototipo!
{
    return 3490;
}
```

Si no declaras tu función antes de usarla (ya sea con un prototipo o con su definición), estás realizando algo llamado *declaración implícita*.Esto estaba permitido en el primer estándar C (C89), y ese estándar tiene reglas al respecto, pero ya no está permitido hoy en día. Y no hay ninguna razón legítima para confiar en ello en código nuevo.

Puede que notes algo en el código de ejemplo que hemos estado utilizando… Es decir, ¡hemos estado usando la vieja función `printf()` sin definirla ni declarar un prototipo! ¿Cómo nos libramos de esta ilegalidad? En realidad, no lo hacemos. Hay un prototipo; está en ese fichero de cabecera `stdio.h` que incluimos con `#include`, ¿recuerdas? ¡Así que seguimos siendo legales, oficial!

## 4.3 Listas de parámetros vacías

Es posible que los veas de vez en cuando en código antiguo, pero nunca deberías usar uno en código nuevo. Usa siempre `void` para indicar que una función no toma parámetros. Nunca hay[1] una razón para omitir esto en código moderno.

Si eres bueno recordando poner `void` para listas de parámetros vacías en funciones y prototipos, puedes saltarte el resto de esta sección.

Hay dos contextos para esto:

- Omitir todos los parámetros donde se define la función
- Omitir todos los parámetros en un prototipo

Veamos primero una posible definición de función:

```
void foo()  // Realmente debería tener un `void` ahí
{
    printf("Hello, world!\n");
}
```

Aunque la especificación dice que el comportamiento en este caso es *como si* hubieras indicado `void` (C11§6.7.6.3¶14), el tipo `void` está ahí por una razón. Utilícelo.

Pero en el caso de un prototipo de función, hay una diferencia *significativa* entre usar `void` y no:

[1] Nunca digas “nunca”.

```
void foo();
void foo(void);  // ¡No es lo mismo!
```

Dejar `void` fuera del prototipo indica al compilador que no hay información adicional sobre los parámetros de la función. De hecho, desactiva toda la comprobación de tipos.

Con un prototipo **definitivamente** use `void` cuando tenga una lista de parámetros vacía.

# Chapter 5

# Punteros… ¡Poder con miedo!

> *"¿Cómo llegas al Carnegie Hall?"*
> *"Practica!"*
>
> —Chiste del siglo XX de origen desconocido

Los punteros son una de las cosas más temidas del lenguaje C. De hecho, son lo único que hace que este lenguaje sea todo un reto. ¿Por qué?

Porque, sinceramente, pueden provocar descargas eléctricas que salgan por el teclado y te *suelden* los brazos de forma permanente, ¡maldiciéndote a una vida al teclado en este idioma de los años 70!

¿De verdad? Bueno, en realidad no. Sólo estoy tratando de prepararte para el éxito.

Dependiendo del lenguaje del que provengas, puede que ya entiendas el concepto de *referencias*, donde una variable hace referencia a un objeto de algún tipo.

Esto es muy parecido, salvo que tenemos que ser más explícitos con C sobre cuándo estamos hablando de la referencia o de la cosa a la que se refiere.

## 5.1 Memoria y variables

La memoria del ordenador contiene datos de todo tipo, ¿verdad? Contendrá `float`s, `int`s, o lo que tengas. Para facilitar el manejo de la memoria, cada byte de memoria se identifica con un número entero. Estos enteros aumentan secuencialmente a medida que avanzas en la memoria[1]. Puedes pensar en ello como un montón de cajas numeradas, donde cada caja contiene un byte[2] de datos. O como un gran array donde cada elemento contiene un byte, si vienes de un lenguaje con arrays. El número que representa cada casilla se llama *dirección*.

Ahora bien, no todos los tipos de datos utilizan sólo un byte. Por ejemplo, un `int` suele tener cuatro bytes, al igual que un `float`, pero en realidad depende del sistema. Puedes utilizar el operador `sizeof` para determinar cuántos bytes de memoria utiliza un determinado tipo.

```
// %zu es el especificador de formato para el tipo size_t

printf("Un int utiliza %zu bytes de memoria\n", sizeof(int));
```

[1] Típicamente. Estoy seguro de que hay excepciones en los oscuros pasillos de la historia de la informática

[2] Un byte es un número formado por no más de 8 dígitos binarios, o *bits* para abreviar. Esto significa que en dígitos decimales como los que usaba la abuela, puede contener un número sin signo entre 0 y 255, ambos inclusive

```
// A mí me imprime "4", pero puede variar según el sistema.
```

> **Datos curiosos sobre la memoria**: Cuando tienes un tipo de datos (como el típico `int`) que utiliza más de un byte de memoria, los bytes que componen los datos son siempre adyacentes en memoria. A veces están en el orden que esperas, y a veces no[a]. Aunque C no garantiza ningún orden de memoria en particular (depende de la plataforma), en general es posible escribir código de forma independiente de la plataforma sin tener que tener en cuenta estos molestos ordenamientos de bytes.
>
> [a] El orden en que vienen los bytes se denomina *endianidad* del número. Los sospechosos habituales son *big-endian* (con el byte más significativo primero) y *little-endian* (con el byte más significativo al final), o, ahora poco común, *mixed-endian* (con los bytes más significativos en otro lugar)

Así que, de todos modos, si podemos ponernos manos a la obra y poner un redoble de tambores y algo de música premonitoria para la definición de puntero, *un puntero es una variable que contiene una dirección*. Imagina la partitura clásica de 2001: Una Odisea del Espacio en este punto. Ba bum ba bum ba bum ¡BAAAAH!

Vale, quizás un poco exagerado, ¿no? No hay mucho misterio sobre los punteros. Son la dirección de los datos. Al igual que una variable `int` puede contener el valor `12`, una variable puntero puede contener la dirección de los datos.

Esto significa que todas estas cosas son lo mismo, es decir, un número que representa un punto en la memoria:

- Índice en memoria (si piensas en la memoria como una gran matriz)
- Dirección
- Ubicación

Voy a usarlos indistintamente. Y sí, acabo de incluir *localización* porque nunca hay suficientes palabras que signifiquen lo mismo.

Y una variable puntero contiene ese número de dirección. Al igual que una variable `float` puede contener `3.14159`.

Imagina que tienes un montón de notas Post-it® numeradas en secuencia con su dirección. (La primera está en el índice numerado `0`, la siguiente en el índice `1`, y así sucesivamente).

Además del número que representa su posición, también puedes escribir otro número de tu elección en cada uno. Puede ser el número de perros que tienes. O el número de lunas alrededor de Marte…

…O, *podría ser el índice de otra nota Post-it*

Si has escrito el número de perros que tienes, eso es sólo una variable normal. Pero si has escrito ahí el índice de otro Post-it, *eso es un puntero*. ¡Apunta a la otra nota!

Otra analogía podría ser con las direcciones de las casas. Puedes tener una casa con ciertas cualidades, patio, tejado metálico, solar, etc. O puedes tener la dirección de esa casa. La dirección no es lo mismo que la casa en sí. Una es una casa completa, y la otra son sólo unas líneas de texto. Pero la dirección de la casa es un *puntero* a esa casa. No es la casa en sí, pero te dice dónde encontrarla.

Y podemos hacer lo mismo en el ordenador con los datos. Puedes tener una variable de datos que contenga algún valor. Y ese valor está en la memoria en alguna dirección. Y puedes tener una variable *puntero* diferente, que contenga la dirección de esa variable de datos.

No es la variable de datos en sí, pero, como con la dirección de una casa, nos dice dónde encontrarla.

Cuando tenemos eso, decimos que tenemos un "puntero a" esos datos. Y podemos seguir el puntero para acceder a los datos en sí.

(Aunque todavía no parece especialmente útil, todo esto se vuelve indispensable cuando se utiliza con llamadas a funciones. Ten paciencia conmigo hasta que lleguemos allí).

Así que si tenemos un `int`, digamos, y queremos un puntero a él, lo que queremos es alguna forma de obtener la dirección de ese `int`, ¿verdad? Después de todo, el puntero sólo contiene la *dirección de* los datos. ¿Qué operador crees que usaríamos para encontrar la *dirección* del `int`?

Pues bien, por una sorpresa que debe resultarle chocante a usted, amable lector, utilizamos el operador `dirección` (que resulta ser un ampersand: "`&`") para encontrar la dirección de los datos. Ampersand.

Así que para un ejemplo rápido, introduciremos un nuevo *especificador de formato* para `printf()` para que puedas imprimir un puntero. Ya sabes cómo `%d` imprime un entero decimal, ¿verdad? Pues bien, `%p` imprime un puntero. Ahora, este puntero va a parecer un número basura (y podría imprimirse en hexadecimal[3] en lugar de decimal), pero es simplemente el índice en memoria en el que se almacenan los datos. (O el índice en memoria en el que se almacena el primer byte de datos, si los datos son multibyte). En prácticamente todas las circunstancias, incluyendo ésta, el valor real del número impreso no es importante para usted, y lo muestro aquí sólo para la demostración del operador `de dirección`.

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    printf("El valor de i es %d\n", i);
    printf("Y su dirección es %p\n", (void *)&i);
```

> **El código anterior contiene un *cast*** donde coaccionamos el tipo de la expresión `&i` para que sea del tipo `void*`. Esto es para evitar que el compilador arroje una advertencia aquí. Esto es todo lo que no hemos cubierto todavía, así que por ahora ignora el `(void*)` en el código de arriba y finge que no está ahí.

En mi computadora, se imprime esto:

```
El valor de i es 10
Y su dirección es 0x7ffddf7072a4
```

Si tienes curiosidad, ese número hexadecimal es 140.727.326.896.068 en decimal (base 10 como la que usaba la abuela). Ese es el índice en memoria donde se almacenan los datos de la variable `i`. Es la dirección de `i`. Es la ubicación de `i`. Es un puntero a `i`.

> **Espera-¿Tienes 140 terabytes de RAM?** ¡Sí! ¿No? Pero me causa gracia, por supuesto que no (ca. 2024). Los ordenadores modernos usan una tecnología milagrosa llamada memoria virtual[a] que hace que los procesos piensen que tienen todo el espacio de memoria de tu ordenador para ellos solos, independientemente de cuánta RAM física lo respalde. Así que aunque la dirección era ese enorme número, está siendo mapeada a alguna dirección de memoria física más baja por el sistema de memoria virtual de mi CPU. Este ordenador en particular tiene 16 GB de RAM (de nuevo, ca. 2024, pero uso Linux, así que es suficiente). ¿Terabytes de RAM? Soy profesor, no un multimillonario punto-com. Nada de esto es algo de lo que que preocuparse, excepto la parte en la que no soy fenomenalmente rico.
>
> [a] https://en.wikipedia.org/wiki/Virtual_memory

Es un puntero porque te permite saber dónde está `i` en la memoria. Al igual que una dirección escrita en un trozo de papel te dice dónde puedes encontrar una casa en particular, este número nos indica en qué parte de la memoria podemos encontrar el valor de `i`. Apunta a `i`.

[3] Es decir, base 16 con dígitos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, y F.

Una vez más, no nos importa cuál es el número exacto de la dirección, por lo general. Sólo nos importa que es un puntero a "i".

## 5.2 Tipos de puntero

Así que… todo esto está muy bien. Ahora puede tomar con éxito la dirección de una variable e imprimirla en la pantalla. Hay algo para el viejo currículum, ¿verdad? Aquí es donde me agarras por el cuello y me preguntas amablemente ¡¡Para qué sirven los punteros!!

Excelente pregunta, y llegaremos a ella justo después de estos mensajes de nuestro patrocinador.

> `SERVICIOS DE LIMPIEZA ROBOTIZADA DE VIVIENDAS. SU VIVIENDA SERÁ DRÁSTICAMENTE MEJORADA O SERÁ DESPEDI`

Bienvenidos a otra entrega de la Guía de Beej. La última vez que nos vimos estuvimos hablando de cómo hacer uso de los punteros. Pues bien, lo que vamos a hacer es almacenar un puntero en una variable para poder utilizarlo más adelante. Puedes identificar el *tipo de puntero* porque hay un asterisco (`*`) antes del nombre de la variable y después de su tipo:

```
int main(void)
{
    int i;  // El tipo de i es "int"
    int *p; // El tipo de p es "puntero a un int", o "int-pointer".
}
```

Así que.. aquí tenemos una variable, que es de tipo puntero, y puede apuntar a otros `int`s. Es decir, puede contener la dirección de otros `int`s. Sabemos que apunta a `int`s, ya que es de tipo `int*` (léase "int-pointer").

Cuando haces una asignación a una variable puntero, el tipo de la parte derecha de la asignación tiene que ser del mismo tipo que la variable puntero. Afortunadamente para nosotros, cuando tomas la `dirección de` (`address-of`) de una variable, el tipo resultante es un puntero a ese tipo de variable, por lo que asignaciones como la siguiente son perfectas:

```
int i;
int *p;  // p es un puntero, pero no está inicializado y apunta a basura

p = &i;  // a p se le asigna la dirección de i--p ahora "apunta a" i
```

A la izquierda de la asignación, tenemos una variable de tipo puntero a `int` (`int*`), y a la derecha, tenemos una expresión de tipo puntero a `int` ya que `i` es un `int` (porque la dirección de `int` te da un puntero a `int`). La dirección de una cosa puede almacenarse en un puntero a esa cosa.

¿Lo entiendes? Sé que todavía no tiene mucho sentido ya que no has visto un uso "real" para la variable puntero, pero estamos dando pequeños pasos aquí para que nadie se pierda. Así que ahora, vamos a presentarte el operador `anti-dirección-de`. Es algo así como lo que sería `address-of` en Bizarro World.

## 5.3 Desreferenciación

Una variable puntero puede considerarse como *referida* a otra variable apuntando a ella. Es raro que oigas a alguien en la tierra de C hablar de "referir" o "referencias", pero lo traigo a colación sólo para que el nombre de este operador tenga un poco más de sentido.

Cuando tienes un puntero a una variable (más o menos "una referencia a una variable"), puedes usar la variable original a través del puntero *referenciando* el puntero. (Puedes pensar en esto como "despointerizar" el puntero, pero nadie dice nunca "despointerizar").

Volviendo a nuestra analogía, esto es vagamente como mirar la dirección de una casa y luego ir a esa casa.

Ahora bien, ¿qué quiero decir con "acceder a la variable original"? Bueno, si tienes una variable llamada `i`, y tienes un puntero a `i` llamado `p`, ¡puedes usar el puntero desreferenciado `p` *exactamente como si fuera la variable original* `i`!

Casi tienes conocimientos suficientes para manejar un ejemplo. El último dato que necesitas saber es el siguiente: ¿qué es el operador de desreferencia? En realidad se llama *operador de dirección*, porque estás accediendo a valores indirectamente a través del puntero. Y es el asterisco, otra vez: `*`. No lo confundas con el asterisco que usaste antes en la declaración del puntero. Son el mismo carácter, pero tienen significados diferentes en contextos diferentes[4].

He aquí un ejemplo en toda regla:

```
#include <stdio.h>

int main(void)
{
    int i;
    int *p;  // esto NO es una desreferencia--esto es un tipo "int*"

    p = &i;  // p apunta ahora a i, p tiene la dirección de i

    i = 10;  // i es ahora 10
    *p = 20; // lo que p señala (es decir, i!) es ahora 20!!

    printf("i es %d\n", i);   // Imprime "20"
    printf("i es %d\n", *p);  // ¡"20"! ¡Dereferencia-p es lo mismo que i!
}
```

Recuerda que `p` contiene la dirección de `i`, como puedes ver donde hicimos la asignación a `p` en la línea 8. Lo que hace el operador de indirección es decirle al ordenador que *utilice el objeto al que apunta el puntero* en lugar de utilizar el propio puntero. De esta manera, hemos convertido `*p` en una especie de alias para `i`.

Genial, pero *¿por qué?* ¿Por qué hacer algo de esto?

## 5.4 Pasar punteros como argumentos

Ahora mismo estarás pensando que tienes muchísimos conocimientos sobre punteros, pero absolutamente ninguna aplicación, ¿verdad? Quiero decir, ¿para qué sirve `*p` si en su lugar puedes decir simplemente `i`?

Pues bien, amigo mío, el verdadero poder de los punteros entra en juego cuando empiezas a pasarlos a funciones. ¿Por qué es esto tan importante? Tal vez recuerdes que antes podías pasar todo tipo de argumentos a las funciones los cuales se copiarían en parámetros, que luego podías manipular en copias locales de esas variables desde dentro de la función, y así devolver un único valor.

¿Qué pasa si quieres devolver más de un dato de la función? Es decir, sólo puedes devolver una cosa, ¿verdad? ¿Y si respondo a esa pregunta con otra pregunta? …Er, ¿dos preguntas?

¿Qué ocurre cuando se pasa un puntero como argumento a una función? ¿Se coloca una copia del puntero en el parámetro correspondiente? ¿Recuerdas que antes he divagado sobre cómo *CADA ARGUMENTO* se copia en los parámetros y la función utiliza una COPIA del argumento? Pues aquí ocurre lo mismo. La función obtendrá una copia del puntero.

[4] ¡Eso no es todo! Se usa en `/*comentarios*/` y en multiplicaciones, ¡y en prototipos de funciones con matrices de longitud variable! Es el mismo `*`, pero el contexto le da un significado diferente

Pero, y esta es la parte inteligente: habremos configurado el puntero de antemano para que apunte a una variable… ¡y entonces la función puede desreferenciar su copia del puntero para volver a la variable original! La función no puede ver la variable en sí, ¡pero sí puede desreferenciar un puntero a esa variable!

Esto es análogo a escribir la dirección de una casa en un papel y luego copiarla en otro papel. Ahora tienes *dos* punteros a esa casa, y ambos son igualmente buenos para llevarte a la casa misma.

En el caso de una llamada a una función, una de las copias se almacena en una variable puntero fuera del ámbito de llamada, y la otra se almacena en una variable puntero que es el parámetro de la función.

Ejemplo: Volvamos a nuestra vieja función `increment()`, pero esta vez hagámosla de modo que realmente incremente el valor en el ámbito de la llamada.

```
#include <stdio.h>

// Nota: ten en cuenta que acepta un puntero a un int
void increment(int *p)
{
    *p = *p + 1;        // Añade uno a la cosa a la que p apunta
}

int main(void)
{
    int i = 10;
   int *j = &i;  // Nota: la dirección de [address-of (&)]; lo convierte en un puntero a i

    printf("i es %d\n", i);        // Imprime "10"
    printf("i es también %d\n", *j);  // Imprime "10"

    increment(j);                  // j es un int*--a i

    printf("i es %d\n", i);        // Imprime "11"!
}
```

¡Ok! Hay un par de cosas que ver aquí… la menor de ellas es que la función `increment()` toma un `int*` como argumento. Le pasamos un `int*` en la llamada cambiando la variable `int i` a un `int*` usando el operador `address-of (&)`. (Recuerda, un puntero contiene una dirección, así que hacemos punteros a variables pasándolas por el operador `address-of`).

La función `increment()` obtiene una copia del puntero. Tanto el puntero original `j` (en `main()`) como la copia de ese puntero `p` (el parámetro en `increment()`) apuntan a la misma dirección, la que contiene el valor `i`. (De nuevo, por analogía, como dos trozos de papel con la misma dirección escrita en ellos). Si desreferenciamos cualquiera de las dos, podremos modificar la variable original `i`. La función puede modificar una variable en otro ámbito. ¡Muévete!

El ejemplo anterior a menudo se escribe de forma más concisa en la llamada simplemente utilizando address-of en la lista de argumentos:

```
printf("i es %d\n", i);  // Imprime "10"
increment(&i);
printf("i es %d\n", i);  // Imprime "11"!
```

Como regla general, si quieres que la función modifique la cosa que estás pasando para que veas el resultado, tendrás que pasar un puntero a esa cosa.

## 5.5 El puntero `NULL`

Cualquier variable puntero de cualquier tipo de puntero puede establecerse a un valor especial llamado `NULL`. Esto indica que este puntero no apunta a nada.

```
int *p;

p = NULL;
```

Dado que no apunta a un valor, su desreferencia es un comportamiento INDEFINIDO y probablemente provoque un fallo:

```
int *p = NULL;

*p = 12;  // FALLA o ALGO PROBABLEMENTE MALO. LO MEJOR ES EVITARLO.
```

A pesar de ser llamado el error del millon de dolares por su creador[5], el puntero `NULL` es un buen sentinela[6] e indicador general de que un puntero aún no ha sido inicializado.

(Por supuesto, al igual que otras variables, el puntero apunta a basura a menos que le asignes explícitamente que apunte a una dirección o a `NULL`).

## 5.6 Nota sobre la declaración de punteros

La sintaxis para declarar un puntero puede ser un poco extraña. Veamos este ejemplo:

```
int a;
int b;
```

Podemos condensarlo en una sola línea, ¿verdad?

```
int a, b;  // Es lo mismo
```

Así que `a` y `b` son ambas `int`s. No hay problema.

Pero, ¿y esto?

```
int a;
int *p;
```

¿Podemos convertirlo en una línea? Sí, podemos. ¿Pero dónde va el `*`?

La regla es que el `*` va delante de cualquier variable que sea de tipo puntero. Es decir, el `*` no es parte del `int` en este ejemplo. es parte de la variable `p`.

Con eso en mente, podemos escribir esto:

```
int a, *p;  // Es lo mismo
```

Es importante notar que la siguiente línea *no* declara dos punteros:

[5] https://en.wikipedia.org/wiki/Null_pointer#Historia
[6] https://en.wikipedia.org/wiki/Sentinel value

```
int *p, q;  // p es un puntero a un int; q es sólo un int.
```

Esto puede ser particularmente insidioso si el programador escribe la siguiente línea de código (válida) que es funcionalmente idéntica a la anterior.

```
int* p, q;  // p es un puntero a un int; q es sólo un int.
```

Así que echa un vistazo a esto y determina qué variables son punteros y cuáles no:

```
int *a, b, c, *d, e, *f, g, h, *i;
```

Dejaré la respuesta en una nota al pie[7]..

## 5.7 `sizeof` y punteros

Sólo un poco de sintaxis aquí que puede ser confusa y que puedes ver de vez en cuando.

Recuerda que `sizeof` opera sobre el *tipo* de la expresión.

```
int *p;

// Imprime el tamaño de un 'int
printf("%zu\n", sizeof(int));

// p es de tipo 'int *', por lo que imprime el tamaño de 'int*'
printf("%zu\n", sizeof p);

// *p es de tipo 'int', por lo que imprime el tamaño de 'int'
printf("%zu\n", sizeof *p);
```

Usted puede ver el código en la naturaleza con ese último `sizeof` allí. Recuerda que `sizeof` se refiere al tipo de expresión, no a las variables de la expresión.

[7] Las variables de tipo puntero son `a`, `d`, `f` e `i`, porque son las que tienen `*` delante

# Chapter 6

# Arrays

> "¿Los índices de las matrices deben empezar en 0 o en 1?
> Mi compromiso de 0.5 fue rechazado sin, pensé, la debida consideración."_
>
> —Stan Kelly-Bootle, informático

Por suerte, C tiene matrices. Ya sé que se considera un lenguaje de bajo nivel [1], pero al menos incorpora el concepto de arrays. Y como muchos lenguajes se inspiraron en la sintaxis de C, probablemente ya estés familiarizado con el uso de `[` y `]` para declarar y usar matrices.

Pero C apenas tiene arrays. Como veremos más adelante, los arrays, en el fondo, son sólo azúcar sintáctico en C—en realidad son todo punteros. Pero por ahora, usémoslos como arrays. *Phew*.

## 6.1 Ejemplo sencillo

Pongamos un ejemplo:

```
#include <stdio.h>

int main(void)
{
    int i;
    float f[4];  // Declara un array de 4 floats

    f[0] = 3.14159;  // La indexación empieza en 0, por supuesto.
    f[1] = 1.41421;
    f[2] = 1.61803;
    f[3] = 2.71828;

    // Imprímelos todos:

    for (i = 0; i < 4; i++) {
        printf("%f\n", f[i]);
    }
}
```

[1] Hoy en día, por lo menos

Cuando declaras un array, tienes que darle un tamaño. Y el tamaño tiene que ser fijo [2].

En el ejemplo anterior, hicimos un array de 4 `float`s. El valor entre corchetes de la declaración nos lo indica.

Más tarde, en las líneas siguientes, accedemos a los valores de la matriz, estableciéndolos u obteniéndolos, de nuevo con corchetes.

Espero que le suenen de alguno de los idiomas que ya conoce.

## 6.2 Obtener la longitud de una matriz

No puedes…ish. C no registra esta información[3]. Tienes que gestionarlo por separado en otra variable.

Cuando digo "no se puede", en realidad quiero decir que hay algunas circunstancias en las que *se puede*. Hay un truco para obtener el número de elementos de un array en el ámbito en el que se declara un array. Pero, en general, esto no funcionará como quieres si pasas el array a una función[4].

Veamos este truco. La idea básica es que usted toma el `sizeof` de la matriz, y luego se divide por el tamaño de cada elemento para obtener la longitud. Por ejemplo, si un `int` es de 4 bytes, y la matriz es de 32 bytes de largo, debe haber espacio para $\frac{32}{4}$ o $8$ `int`s allí.

```
int x[12];  // 12 ints

printf("%zu\n", sizeof x);     // 48 bytes totales
printf("%zu\n", sizeof(int));  // 4 bytes por int

printf("%zu\n", sizeof x / sizeof(int));  // 48/4 = 12 ints!
```

Si es un array de `char`s, entonces `sizeof` del array *es* el número de elementos, ya que `sizeof(char)` está definido como 1. Para cualquier otro tipo, tienes que dividir por el tamaño de cada elemento.

Pero este truco sólo funciona en el ámbito en el que se definió el array. Si pasas el array a una función, no funciona. Incluso si lo haces "grande" en la firma de la función:

```
void foo(int x[12])
{
    printf("%zu\n", sizeof x);     // ¡8?! ¿Qué ha sido del 48?
    printf("%zu\n", sizeof(int));  // 4 bytes por int

    printf("%zu\n", sizeof x / sizeof(int));  // 8/4 = 2 ints?? INCORRECTO.
}
```

Esto se debe a que cuando "pasas" arrays a funciones, sólo estás pasando un puntero al primer elemento, y eso es lo que mide `sizeof`. Más sobre esto en la sección, Pasar arrays unidimensionales a funciones. más abajo.

Otra cosa que puedes hacer con `sizeof` y arrays es obtener el tamaño de un array de un número fijo de elementos sin declarar el array. Es como obtener el tamaño de un `int` con `sizeof(int)`.

Por ejemplo, para ver cuántos bytes se necesitarían para un array de 48 `dobless`, puedes hacer esto:

[2] De nuevo, en realidad no, pero las matrices de longitud variable -de las que no soy muy fan- son una historia para otro momento

[3] Dado que los arrays son sólo punteros al primer elemento del array bajo el capó, no hay información adicional que registre la longitud

[4] Porque cuando pasas un array a una función, en realidad sólo estás pasando un puntero al primer elemento de ese array, no el array "entero"

```
sizeof(double [48]);
```

## 6.3 Inicializadores de matrices

Puedes inicializar un array de antemano:

```
#include <stdio.h>

int main(void)
{
    int i;
    int a[5] = {22, 37, 3490, 18, 95};  // Inicializar con estos valores

    for (i = 0; i < 5; i++) {
        printf("%d\n", a[i]);
    }
}
```

Nunca debes tener más elementos en tu inicializador de los que caben en el array, o el compilador se pondrá de mal humor:

```
foo.c: In function ‘main’:
foo.c:6:39: warning: excess elements in array initializer
    6 |     int a[5] = {22, 37, 3490, 18, 95, 999};
      |                                       ^~~
foo.c:6:39: note: (near initialization for ‘a’)
```

Pero (¡dato curioso!) puedes tener *menos* elementos en tu inicializador de los que caben en el array. Los elementos restantes de la matriz se inicializarán automáticamente con cero. Esto es cierto en general para todos los tipos de inicializadores de matrices: si tienes un inicializador, todo lo que no se establezca explícitamente a un valor se establecerá a cero.

```
int a[5] = {22, 37, 3490};

// Es lo mismo que:

int a[5] = {22, 37, 3490, 0, 0};
```

Es un atajo común ver esto en un inicializador cuando quieres poner un array entero a cero:

```
int a[100] = {0};
```

Lo que significa, “Haz el primer elemento cero, y luego automáticamente haz el resto cero, también”.

También puedes establecer elementos específicos del array en el inicializador, especificando un índice para el valor. Cuando haces esto, C seguirá inicializando los valores subsiguientes por ti hasta que el inicializador se agote, llenando todo lo demás con `0`.

Para hacer esto, pon el índice entre corchetes con un `=` después, y luego establece el valor.

Aquí hay un ejemplo donde construimos un array:

```
int a[10] = {0, 11, 22, [5]=55, 66, 77};
```

Como hemos puesto el índice 5 como inicio para `55`, los datos resultantes en el array son:

```
0 11 22 0 0 55 66 77 0 0
```

También puedes introducir expresiones constantes sencillas.

```
#define COUNT 5

int a[COUNT] = {[COUNT-3]=3, 2, 1};
```

que nos da:

```
0 0 3 2 1
```

Por último, también puedes hacer que C calcule el tamaño del array a partir del inicializador, simplemente dejando el tamaño desactivado:

```
int a[3] = {22, 37, 3490};

// Es lo mismo que:

int a[] = {22, 37, 3490};  // ¡Dejé el tamaño!
```

## 6.4 ¡Fuera de los límites! (Out of Bounds!)

C no te impide acceder a matrices fuera de los límites. Puede que ni siquiera te avise.

Robemos el ejemplo de arriba y sigamos imprimiendo el final del array. Sólo tiene 5 elementos, pero vamos a tratar de imprimir 10 y ver lo que sucede:

```
#include <stdio.h>

int main(void)
{
    int i;
    int a[5] = {22, 37, 3490, 18, 95};

   for (i = 0; i < 10; i++) {  // MALAS NOTICIAS: ¡imprime demasiados elementos!
        printf("%d\n", a[i]);
    }
}
```

Ejecutándolo en mi computadora imprime:

```
22
37
3490
18
95
```

```
32765
1847052032
1780534144
-56487472
21890
```

¡Caramba! ¿Qué es esto? Bueno, resulta que imprimir el final de un array resulta en lo que los desarrolladores de C llaman *comportamiento indefinido*. Hablaremos más sobre esta bestia más adelante, pero por ahora significa: “Has hecho algo malo, y cualquier cosa podría pasar durante la ejecución de tu programa”.

Y por cualquier cosa, me refiero típicamente a cosas como encontrar ceros, encontrar números basura, o bloquearse. Pero en realidad la especificación de C dice que en estas circunstancias el compilador puede emitir código que haga *cualquier cosa*[5].

Versión corta: no hagas nada que cause un comportamiento indefinido. Nunca [6].

## 6.5 Matrices multidimensionales

Puede añadir tantas dimensiones como desee a sus matrices.

```
int a[10];
int b[2][7];
int c[4][5][6];
```

Se almacenan en memoria en row-major order[7].Esto significa que en una matriz 2D, el primer índice de la lista indica la *fila* y el segundo la *columna*. También puedes utilizar inicializadores en matrices multidimensionales anidándolos:

```
#include <stdio.h>

int main(void)
{
    int row, col;

    int a[2][5] = {      // Inicializar una matriz 2D
        {0, 1, 2, 3, 4},
        {5, 6, 7, 8, 9}
    };

    for (row = 0; row < 2; row++) {
        for (col = 0; col < 5; col++) {
            printf("(%d,%d) = %d\n", row, col, a[row][col]);
        }
    }
}
```

Para la salida de:

[5] En los viejos tiempos de MS-DOS, antes de que existiera la protección de memoria, yo escribía un código C particularmente abusivo que deliberadamente tenía todo tipo de comportamientos indefinidos. Pero sabía lo que hacía, y las cosas funcionaban bastante bien. Hasta que cometí un error que causó un bloqueo y, como descubrí al reiniciar, borró todas mis configuraciones de BIOS. Fue divertido. (Un saludo a @man por esos momentos de diversión)

[6] Hay un montón de cosas que causan un comportamiento indefinido, no sólo los accesos a arrays fuera de los límites. Esto es lo que hace al lenguaje C tan *excitante*

[7] https://en.wikipedia.org/wiki/Row-_and_column-major_order

```
(0,0) = 0
(0,1) = 1
(0,2) = 2
(0,3) = 3
(0,4) = 4
(1,0) = 5
(1,1) = 6
(1,2) = 7
(1,3) = 8
(1,4) = 9
```

Y se puede inicializar con índices explícitos:

```
// Hacer una matriz de identidad 3x3
int a[3][3] = {[0][0]=1, [1][1]=1, [2][2]=1};
```

que construye un array 2D como este:

```
1 0 0
0 1 0
0 0 1
```

## 6.6 Matrices y punteros

Así que… "*Casualmente*" ¿podría haber mencionado que los arrays eran punteros, en el fondo? Deberíamos hacer una inmersión superficial en eso ahora para que las cosas no sean completamente confusas. Más adelante veremos cuál es la relación real entre arrays y punteros, pero por ahora sólo quiero pasar arrays a funciones.

### 6.6.1 Obtener un puntero a una matriz

Quiero contarte un secreto. En general, cuando un programador de C habla de un puntero a un array, está hablando de un puntero *al primer elemento* del array [8].

Obtengamos un puntero al primer elemento de un array.

```
#include <stdio.h>

int main(void)
{
    int a[5] = {11, 22, 33, 44, 55};
    int *p;

    p = &a[0];  // p apunta a la matriz
                // Bueno, al primer elemento, en realidad

    printf("%d\n", *p);  // Imprime "11"
}
```

Esto es tan común de hacer en C que el lenguaje nos permite una forma abreviada:

[8] Esto es técnicamente incorrecto, ya que un puntero a un array y un puntero al primer elemento de un array tienen tipos diferentes. Pero podemos quemar ese puente cuando lleguemos a él

```
p = &a[0];  // p apunta a la matriz

// Es lo mismo que:

p = a;      // p apunta a la matriz, ¡pero es mucho más bonito!
```

Hacer referencia al nombre del array de forma aislada es lo mismo que obtener un puntero al primer elemento del array. Vamos a utilizar esto ampliamente en los próximos ejemplos.

Pero espera un segundo… ¿no es `p` un `int*`? ¿Y `*p` nos da `11`, lo mismo que `a[0]`? Sí. Estás empezando a ver cómo se relacionan las matrices y los punteros en C.

### 6.6.2 Paso de matrices unidimensionales a funciones

Hagamos un ejemplo con un array unidimensional. Voy a escribir un par de funciones a las que podemos pasar el array para que hagan cosas diferentes.

¡Prepárate para algunas firmas de funciones alucinantes!

```
#include <stdio.h>

// Pasar como puntero al primer elemento
void times2(int *a, int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 2);
}

// Lo mismo, pero utilizando la notación de matriz
void times3(int a[], int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 3);
}

// Lo mismo, pero utilizando la notación de matriz con tamaño
void times4(int a[5], int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 4);
}

int main(void)
{
    int x[5] = {11, 22, 33, 44, 55};

    times2(x, 5);
    times3(x, 5);
    times4(x, 5);
}
```

Todos esos métodos de enumerar el array como parámetro en la función son idénticos.

```
void times2(int *a, int len)
void times3(int a[], int len)
void times4(int a[5], int len)
```

En el uso por parte de los habituales de C, la primera es la más común, con diferencia.

Y, de hecho, en la última situación, el compilador ni siquiera le importa qué número le pasas (aparte de que tiene que ser mayor que cero[9]). No impone nada en absoluto.

Ahora que lo he dicho, el tamaño del array en la declaración de la función realmente *importa* cuando pasas arrays multidimensionales a funciones, pero volveremos a eso.

### 6.6.3 Modificación de matrices en funciones

Ya hemos dicho que las matrices son punteros disfrazados. Esto significa que si pasas un array a una función, probablemente estés pasando un puntero al primer elemento del array.

Pero si la función tiene un puntero a los datos, ¡puede manipular esos datos! Así que los cambios que una función hace a un array serán visibles de nuevo en el invocador.

He aquí un ejemplo en el que pasamos un puntero a un array a una función, la función manipula los valores de ese array, y esos cambios son visibles en la llamada.

```
#include <stdio.h>

void double_array(int *a, int len)
{
    // Multiplica cada elemento por 2
    //
    // Esto duplica los valores en 'x' en main() ya que 'x' y 'a' apuntan
    // ¡Al mismo array en memoria!

    for (int i = 0; i < len; i++)
        a[i] *= 2;
}

int main(void)
{
    int x[5] = {1, 2, 3, 4, 5};

    double_array(x, 5);

    for (int i = 0; i < 5; i++)
        printf("%d\n", x[i]);  // 2, 4, 6, 8, 10!
}
```

Aunque pasamos el array como parámetro `a` que es de tipo `int*`, ¡mira cómo accedemos a él usando la notación array con `a[i]`! Vaya. Esto está totalmente permitido.

Más adelante, cuando hablemos de la equivalencia entre arrays y punteros, veremos que esto tiene mucho más sentido. Por ahora, es suficiente saber que las funciones pueden hacer cambios a los arrays que son

[9] C11 §6.7.6.2¶1 requiere que sea mayor que cero. Pero puede que veas código por ahí con arrays declarados de longitud cero al final de `struct`s y GCC es particularmente indulgente al respecto a menos que compiles con `-pedantic`. Este array de longitud cero era un mecanismo para hacer estructuras de longitud variable. Desafortunadamente, es técnicamente un comportamiento indefinido acceder a un array de este tipo aunque básicamente funcionaba en todas partes. C99 codificó un reemplazo bien definido llamado *flexible array members*, del que hablaremos más adelante

visibles en el llamador.

### 6.6.4 Paso de matrices multidimensionales a funciones

La historia cambia un poco cuando hablamos de matrices multidimensionales. C necesita conocer todas las dimensiones (excepto la primera) para saber en qué parte de la memoria debe buscar un valor.

He aquí un ejemplo en el que somos explícitos con todas las dimensiones:

```
#include <stdio.h>

void print_2D_array(int a[2][3])
{
    for (int row = 0; row < 2; row++) {
        for (int col = 0; col < 3; col++)
            printf("%d ", a[row][col]);
        printf("\n");
    }
}

int main(void)
{
    int x[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    print_2D_array(x);
}
```

Pero en este caso, estos dos[10] son equivalentes:

```
void print_2D_array(int a[2][3])
void print_2D_array(int a[][3])
```

En realidad, el compilador sólo necesita la segunda dimensión para poder calcular la distancia de memoria que debe saltarse en cada incremento de la primera dimensión. En general, necesita conocer todas las dimensiones excepto la primera.

Además, recuerda que el compilador hace una comprobación mínima de los límites en tiempo de compilación (si tienes suerte), y C no hace ninguna comprobación de los límites en tiempo de ejecución.¡Sin cinturones de seguridad! No te estrelles accediendo a elementos del array fuera de los límites.

[10] Esto también es equivalente: `void print_2D_array(int (*a)[3])`, pero eso es más de lo que quiero entrar ahora

# Chapter 7

# Strings (“Cadenas” de caracteres)

¡Por fin! ¡Cadenas! ¿Qué podría ser más sencillo?

Bueno, resulta que las cadenas en realidad no son cadenas en C. ¡Así es! ¡Son punteros! Por supuesto que lo son.

Al igual que las matrices, las cadenas en C *apenas existen*.

Pero vamos a comprobarlo… en realidad no es para tanto.

## 7.1 Literales de cadena

Antes de empezar, hablemos de los literales de cadena en C. Son secuencias de caracteres entre comillas *dobles* (`"`). (Las comillas simples encierran caracteres, y son un animal completamente diferente).

Por ejemplo:

```
"Hello, world!\n"
"This is a test."
"When asked if this string had quotes in it, she replied, \"It does.\""
```

El primero tiene una nueva línea al final, algo bastante común.

La última tiene comillas incrustadas, pero cada una está precedida por (decimos «escapada por») una barra invertida (`\`) indicando que una comilla literal pertenece a la cadena en este punto. Así es como el compilador de C, puede diferenciar entre, imprimir una comilla doble y la comilla doble al final de la cadena.

## 7.2 Variables de cadena

Ahora que sabemos cómo hacer un literal de cadena, asignémoslo a una variable para poder hacer algo con él.

```
char *s = "Hello, world!";
```

Fíjate en el tipo: puntero a un `char`. La variable de cadena `s` es en realidad un puntero al primer carácter de esa cadena, concretamente la `H`.

Y podemos imprimirlo con el especificador de formato `%s` (de **S**tring «cadena»):

```
char *s = "Hello, world!"; // "Hola, mundo!"

printf("%s\n", s);  // Hello, world!
```

## 7.3 Variables de cadena como matrices

Otra opción es ésta, casi equivalente al uso anterior de `char*`:

```
char s[14] = "Hello, world!";

// o, si fuéramos perezosos y dejáramos que el compilador
// calculara la longitud por nosotros:

char s[] = "Hello, world!";
```

Esto significa que puedes utilizar la notación de matrices para acceder a los caracteres de una cadena.Hagamos exactamente eso para imprimir todos los caracteres de una cadena en la misma línea:

```
#include <stdio.h>

int main(void)
{
    char s[] = "Hello, world!";

    for (int i = 0; i < 13; i++)
        printf("%c\n", s[i]);
}
```

Tenga en cuenta que estamos utilizando el especificador de formato `%c` para imprimir un solo carácter.

Además, fíjate en esto. El programa seguirá funcionando bien si cambiamos la definición de `s` para que sea de tipo `char*`:

```
#include <stdio.h>

int main(void)
{
    char *s = "Hello, world!";   // char* aqui

    for (int i = 0; i < 13; i++)
        printf("%c\n", s[i]);    // ¿Pero seguir usando arrays aquí...?
}
```

Y aún podemos utilizar la notación de matrices para imprimirlo. Esto es sorprendente, pero sólo porque aún no hemos hablado de la equivalencia matriz/puntero. Pero esto es otra pista de que los arrays y los punteros son la misma cosa, en el fondo.

## 7.4 Inicializadores de cadenas

Ya hemos visto algunos ejemplos con la inicialización de variables de cadena con literales de cadena:

```
char *s = "Hello, world!";
char t[] = "Hello, again!";
```

Pero estas dos inicializaciones son sutilmente diferentes. Un literal de cadena, similar a un literal de número entero, tiene su memoria gestionada automáticamente por el compilador. Con un entero, es decir, un dato de tamaño fijo, el compilador puede gestionarlo con bastante facilidad. Pero las cadenas son una bestia de bytes variables que el compilador domestica lanzándolas a un trozo de memoria, y dándote un puntero a él.

Esta forma apunta al lugar donde se colocó esa cadena. Típicamente, ese lugar está en una tierra lejana del resto de la memoria de tu programa – memoria de sólo lectura – por razones relacionadas con el rendimiento y la seguridad.

```
char *s = "Hello, world!";
```

Entonces, si intentas mutar esa cadena con esto:

```
char *s = "Hello, world!";

s[0] = 'z';  // MALAS NOTICIAS: ¡intentó mutar una cadena literal!
```

El comportamiento es indefinido. Probablemente, dependiendo de su sistema, se producirá un fallo.

Pero declararlo como un array es diferente. El compilador no guarda esos bytes en otra parte de la ciudad, están al final de la calle. Esta es una *copia* mutable de la cadena – una que podemos cambiar a voluntad:

```
char t[] = "Hello, again!";  // t es una copia de la cadena
t[0] = 'z'; //  No hay problema

printf("%s\n", t);  // "zello, again!"
```

Así que recuerda: si tienes un puntero a un literal de cadena, ¡no intentes cambiarlo! Y si usas una cadena entre comillas dobles para inicializar un array, no es realmente un literal de cadena.

## 7.5 Obtención de la longitud de la cadena

No puedes, ya que C no lo rastrea por ti. Y cuando digo «no puede», en realidad quiero decir «puede»[1]. Hay una función en `<string.h>` llamada `strlen()` que puede usarse para calcular la longitud de cualquier cadena en bytes[2].

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

int main(void)
{
    char *s = "Hello, world!";

    printf("La cadena tiene %zu bytes de longitud.\n", strlen(s));
}
```

[1] Aunque es cierto que C no rastrea la longitud de las cadenas

[2] Si estás usando el juego de caracteres básico o un juego de caracteres de 8 bits, estás acostumbrado a que un carácter sea un byte. Sin embargo, esto no es cierto en todas las codificaciones de caracteres

La función `strlen()` devuelve el tipo `size_t`, que es un tipo entero por lo que se puede utilizar para matemáticas de enteros. Imprimimos `size_t` con `%zu`.

El programa anterior imprime:

```
La cadena tiene 13 bytes de longitud.
```

Estupendo. ¡Así que *es* posible obtener la longitud de la cadena!

Pero… si C no rastrea la longitud de la cadena en ninguna parte, ¿cómo sabe cuán larga es la cadena?

## 7.6 Terminación de la cadena

C hace las cadenas de forma un poco diferente a muchos lenguajes de programación, y de hecho de forma diferente a casi todos los lenguajes de programación modernos.

Cuando estás haciendo un nuevo lenguaje, tienes básicamente dos opciones para almacenar una cadena en memoria:

1. Almacenar los bytes de la cadena junto con un número que indica la longitud de la cadena.
2. Almacenar los bytes de la cadena, y marcar el final de la cadena con un byte especial llamado *terminador*.

Si desea cadenas de más de 255 caracteres, la opción 1 requiere al menos dos bytes para almacenar la longitud. Mientras que la opción 2 sólo requiere un byte para terminar la cadena. Así que se ahorra un poco.

Por supuesto, hoy en día parece ridículo preocuparse por ahorrar un byte (o 3: muchos lenguajes te permiten tener cadenas de 4 gigabytes de longitud). Pero en su día, era un problema mayor.

Así que C adoptó el enfoque nº 2. En C, una «cadena» se define por dos características básicas:

- Un puntero al primer carácter de la cadena.
- Un byte de valor cero (o carácter `NUL`[3]) en algún lugar de la memoria después del puntero que indica el final de la cadena.

Un carácter `NUL` puede escribirse en código C como `\0`, aunque no es necesario hacerlo a menudo.

Cuando incluyes una cadena entre comillas dobles en tu código, el carácter `NUL` se incluye automática e implícitamente.

```
char *s = "Hello!";  // En realidad «Hola!\0» entre bastidores
```

Así que con esto en mente, vamos a escribir nuestra propia función `strlen()` que cuenta `caracteres` en una cadena hasta que encuentra un `NUL`.

El procedimiento es buscar en la cadena un único carácter `NUL`, contando a medida que avanzamos[4]:

```
int my_strlen(char *s)
{
    int count = 0;

    while (s[count] != '\0')  // Comillas simples para caracteres simples
        count++;

    return count;
```

[3]Esto es diferente del puntero `NULL`, y lo abreviaré `NUL` cuando hable del carácter frente a `NULL` para el puntero

[4]Más adelante aprenderemos una forma más ordenada de hacerlo con aritmética de punteros

```
}
```

Y así es como la función `strlen()` hace su trabajo.

## 7.7 Copiar una cadena

No se puede copiar una cadena mediante el operador de asignación (`=`). Todo lo que hace es hacer una copia del puntero al primer carácter… por lo que terminas con dos punteros a la misma cadena:

```
#include <stdio.h>

int main(void)
{
    char s[] = "Hello, world!";
    char *t;

    // Esto hace una copia del puntero, ¡no una copia de la cadena!
    t = s;

    // Modificamos t
    t[0] = 'z';

    // ¡Pero imprimir s muestra la modificación!
    // ¡Porque t y s apuntan a la misma cadena!

    printf("%s\n", s);  // "zello, world!"
}
```

Si quieres hacer una copia de una cadena, tienes que copiarla byte a byte—pero esto es más fácil con la función `strcpy()`[5].

Antes de copiar la cadena, asegúrate de que tienes espacio para copiarla, es decir, la matriz de destino que va a contener los caracteres debe ser al menos tan larga como la cadena que estás copiando.

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

int main(void)
{
    char s[] = "Hello, world!";
    char t[100];  // Cada char es un byte, así que hay espacio de sobra

    // ¡Esto hace una copia de la cadena!
    strcpy(t, s);

    // Modificamos t
    t[0] = 'z';

    // Y s no se ve afectada porque es una cadena diferente
    printf("%s\n", s);  // "Hello, world!"
```

[5]Hay una función más segura llamada `strncpy()` que probablemente deberías usar en su lugar, pero llegaremos a eso más tarde

```
    // Pero t ha cambiado
    printf("%s\n", t);  // "zello, world!"
}
```

Observe que con `strcpy()`, el puntero de destino es el primer argumento, y el puntero de origen es el segundo. Una mnemotécnica que uso para recordar esto es que es el orden en el que habrías puesto `t` y `s` si una asignación `=` funcionara para cadenas, con el origen a la derecha y el destino a la izquierda.

# Chapter 8

# Estructuras (Structs)

En C, tenemos algo llamado `struct`, que es un tipo definible por el usuario, el cual, contiene múltiples piezas de datos, potencialmente de diferentes tipos.

Es una forma conveniente de agrupar múltiples variables en una sola. Esto puede ser beneficioso para pasar variables a funciones (así sólo tienes que pasar una en lugar de muchas), y útil para organizar datos y hacer el código más legible.

Si vienes de otro lenguaje, puede que estés familiarizado con la idea de *clases* y *objetos*. Estos no existen en C, de forma nativa[1]. Puedes pensar en una `struct` como una clase con sólo miembros de datos, y sin métodos.

## 8.1 Declaración de una estructura

Puedes declarar una `struct` en tu código de la siguiente manera:

```
struct car {
    char *name;
    float price;
    int speed;
};
```

Esto se hace a menudo en el ámbito global, fuera de cualquier función, para que la «estructura» esté disponible globalmente.

Cuando haces esto, estás creando un nuevo *tipo*. El nombre completo del tipo es `struct car`. (No sólo `car`—eso no funcionará).

Todavía no hay variables de ese tipo, pero podemos declarar algunas:

```
struct car saturn;  // Variable «saturn» de tipo «struct car»
```

Y ahora tenemos una variable no inicializada `saturn`[2] de tipo `struct car`.

Deberíamos inicializarlo. Pero, ¿cómo establecemos los valores de cada uno de esos campos?

[1] Aunque en C los elementos individuales en memoria como `int`s se denominan «objetos», no son objetos en el sentido de la programación orientada a objetos

[2] El Saturn fue una popular marca de coches económicos en los Estados Unidos hasta que fue sacada del negocio por el crack de 2008, tristemente para nosotros los fans.

Como en muchos otros lenguajes que lo robaron de C, vamos a usar el operador punto (`.`) para acceder a los campos individuales.

```
saturn.name = "Saturn SL/2";
saturn.price = 15999.99;
saturn.speed = 175;

printf("Name:           %s\n", saturn.name);
printf("Price (USD):    %f\n", saturn.price);
printf("Top Speed (km): %d\n", saturn.speed);
```

Allí en las primeras líneas, establecemos los valores en la `struct car`, y luego en la siguiente parte, imprimimos esos valores.

## 8.2 Inicializadores de estructuras

El ejemplo de la sección anterior era un poco difícil de manejar. Tiene que haber una forma mejor de inicializar esa variable `struct`.

Puedes hacerlo con un inicializador, poniendo valores en los campos *en el orden en que aparecen en la `struct`*, cuando defines la variable. (Esto no funcionará después de que la variable haya sido definida - tiene que ocurrir en la definición).

```
struct car {
    char *name;
    float price;
    int speed;
};

// ¡Ahora con un inicializador! Mismo orden de campos que en la declaración struct:
struct car saturn = {"Saturn SL/2", 16000.99, 175};

printf("Name:      %s\n", saturn.name);
printf("Price:     %f\n", saturn.price);
printf("Top Speed: %d km\n", saturn.speed);
```

El hecho de que los campos del inicializador tengan que estar en el mismo orden, es un poco raro. Si alguien cambia el orden en `struct car`, ¡podría romper el resto del código!

Podemos ser más específicos con nuestros inicializadores:

```
struct car saturn = {.speed=175, .name="Saturn SL/2"};
```

Ahora, es independiente del orden, en la declaración `struct`. Lo que sin duda es un código más seguro.

De forma similar a los inicializadores de array, cualquier designador de campo que falte, se inicializa a cero (en este caso, sería `.price`, que he omitido).

## 8.3 Paso de estructuras a funciones

Puedes hacer un par de cosas para pasar una `struct` a una función.

1. Pasar la `struct`.
2. Pasar un puntero a la `struct`.

Recuerda que cuando pasas algo a una función, se hace una *copia* de esa cosa para que la función opere sobre ella, ya sea una copia de un puntero, un `int`, una `struct`, o cualquier otra cosa.

Hay básicamente dos casos en los que querrías pasar un puntero a la `struct`:

1. Necesitas que la función sea capaz de hacer cambios a la `struct` que fue pasada, y que esos cambios se muestren en la llamada.
2. La `struct` es algo grande y es más caro copiarla en la pila que copiar un puntero[3].

Por estas dos razones, es mucho más común pasar un puntero a una `estructura` es una función, aunque no es ilegal pasar solamente la `estructura`.

Intentemos pasar un puntero, haciendo una función que nos permita establecer el campo `.price` de la `struct car`:

```
#include <stdio.h>

struct car {
    char *name;
    float price;
    int speed;
};

int main(void)
{
    struct car saturn = {.speed=175, .name="Saturn SL/2"};

    // Pasar un puntero a este coche struct, junto con un nuevo,
    // más realista, precio:
    set_price(&saturn, 799.99);

    printf("Price: %f\n", saturn.price);
}
```

Usted debe ser capaz de llegar a la firma de la función para `set_price()` con sólo mirar los tipos de los argumentos que tenemos.

`saturn` es un `struct car`, así que `&saturn` debe ser la dirección del `struct car`, es decir, un puntero a un `struct car`, un `struct car*`.

Y `799.99` es un `float`.

Así que la declaración de la función debe tener este aspecto:

```
void set_price(struct car *c, float new_price)
```

Sólo tenemos que escribir el cuerpo. Un intento podría ser:

```
void set_price(struct car *c, float new_price) {
    c.price = new_price;  // ERROR!!
}
```

Eso no funcionará porque el operador punto sólo funciona en `struct`s… no funciona en *punteros* a `struct`s.

[3]Un puntero es probablemente de 8 bytes en un sistema de 64 bits

Entonces podemos desreferenciar la variable `c` para des-apuntarla y llegar a la propia `struct`. Dereferenciar una `struct car*` resulta en la `struct car` a la que apunta el puntero, sobre la que deberíamos poder usar el operador punto:

```
void set_price(struct car *c, float new_price) {
    (*c).price = new_price;  // Funciona, pero es feo y no idiomático :(
}
```

Y funciona. Pero es un poco engorroso teclear todos esos paréntesis y el asterisco. C tiene un azúcar sintáctico llamado, operador *flecha (arrow)* que ayuda con eso.

## 8.4 El operador Arrow / flecha (->)

El operador flecha ayuda a referirse a campos en punteros a `struct`s.

```
void set_price(struct car *c, float new_price) {
    // (*c).price = new_price;  // Funciona, pero no es idiomático :(
    //
    // La línea de arriba es 100% equivalente a la de abajo:

    c->price = new_price;  // ¡Ese es!
}
```

Así que.. cuando accedemos a campos, ¿cuándo usamos punto, y cuándo usamos flecha?

- Si tienes una `struct`, usa punto (`.`).
- Si tienes un puntero a una `struct`, usa arrow/flecha (`->`).

## 8.5 Copiar y devolver `structs`

Aquí tienes una fácil.

¡Sólo tienes que asignar de uno a otro!

```
struct car a, b;

b = a;  // Copiar la estructura
```

Devolver una `estructura` (en lugar de un puntero a una) desde una función, también hace una copia similar a la variable receptora.

Esto no es una «copia profunda»[4]. Todos los campos se copian tal cual, incluyendo los punteros a cosas.

## 8.6 Comparación de `structs`

Sólo hay una forma segura de hacerlo: comparar cada campo de uno en uno.

Podrías pensar que podrías utilizar `memcmp()`[5], pero eso no maneja el caso de los posibles bytes de relleno que pueda haber.

[4] Una *copia profunda* sigue a los punteros en la `struct` y copia también los datos a los que apuntan. Una *copia superficial* sólo copia los punteros, pero no las cosas a las que apuntan. C no viene con ninguna funcionalidad de copia profunda incorporada

[5] https://beej.us/guide/bgclr/html/split/stringref.html#man-strcmp

Si primero borras la `struct` a cero con `memset()`[6], entonces *podría* funcionar, aunque podría haber elementos extraños que puede que no se compare como usted espera[7].

[6] https://beej.us/guide/bgclr/html/split/stringref.html#man-memset
[7] https://stackoverflow.com/questions/141720/how-do-you-compare-structs-for-equality-in-c

# Chapter 9

# Archivo de Entrada/Salida (Input/Output)

Ya hemos visto algunos ejemplos de E/S con `printf()` para hacer E/S en la consola.

Pero llevaremos estos conceptos un poco más lejos en este capítulo.

## 9.1 El tipo de dato `FILE*`.

Cuando hacemos cualquier tipo de E/S en C, lo hacemos a través de un dato que se obtiene en forma de un tipo `FILE*`. Este `FILE*` contiene toda la información necesaria, para comunicarse con el subsistema de E/S acerca de qué fichero tienes abierto, en qué parte del fichero te encuentras, etc.

La especificación se refiere a estos como *streams*, es decir, un flujo de datos de un archivo o de cualquier fuente. Voy a utilizar «archivos (File)» y «flujos (streams)» indistintamente, pero en realidad deberías pensar en un «archivo (File)» como un caso especial de un «flujo (Stream)». Hay otras formas de introducir datos en un programa además de leerlos de un fichero.

Veremos en un momento, cómo pasar de tener un nombre de fichero, a obtener un `FILE*` abierto para él, pero primero quiero mencionar tres flujos que ya están abiertos para ti y listos para usar.

| `FILE*` nombre | Descripción |
|---|---|
| `stdin` | Entrada estándar, generalmente por defecto es el teclado |
| `stdout` | Salida estándar, generalmente por defecto es la pantalla |
| `stderr` | Error estándar, generalmente por defecto es la pantalla |

Resulta que ya los hemos estado utilizando implícitamente. Por ejemplo, estas dos llamadas son iguales:

```
printf("Hello, world!\n");
fprintf(stdout, "Hello, world!\n");  // printf a un fichero
```

Pero hablaremos de ello más adelante.

También notarás que tanto `stdout` como `stderr` van a la pantalla. Aunque al principio esto parece un descuido o una redundancia, en realidad no lo es. Los sistemas operativos típicos, te permiten *redirigir* la salida de cualquiera de ellos a archivos diferentes, y puede ser conveniente poder separar los mensajes de error, de la salida normal que no es de error.

Por ejemplo, en un shell POSIX (como sh, ksh, bash, zsh, etc.) en un sistema tipo Unix, podríamos ejecutar un programa y enviar sólo la salida no error (`stdout`) a un fichero, y toda la salida error (`stderr`) a otro fichero.

```
./foo > output.txt 2> errors.txt   # Este comando es específico de Unix
```

Por este motivo, debe enviar los mensajes de error graves a `stderr` en lugar de a `stdout`.

Más adelante se explica cómo hacerlo.

## 9.2 Lectura de archivos de texto

Los flujos se clasifican en dos categorías diferentes: *texto* y *binario*.

A los flujos de texto, se les permite hacer traducciones significativas de los datos, sobre todo, traducciones de nuevas líneas a sus diferentes representaciones[1]. Los archivos de texto son lógicamente una secuencia de *líneas* separadas por nuevas líneas. Para que sean portables, los datos de entrada deben terminar siempre con una nueva línea.

Pero la regla general, es que si puedes editar el archivo en un editor de texto normal, es un archivo de texto. En caso contrario, es binario. Hablaremos más sobre binario en un momento.

Así que manos a la obra: ¿cómo abrimos un archivo para leerlo y extraer datos de él?

Creemos un archivo llamado `hello.txt` que contenga esto:

```
Hello, world!
```

Y vamos a escribir un programa para abrir el archivo, leer un carácter fuera de él, y luego cerrar el archivo cuando hayamos terminado. ¡Ese es el plan!

```
#include <stdio.h>

int main(void)
{
    FILE *fp;                   // Variable para representar el archivo abierto

    fp = fopen("hello.txt", "r");  // Abrir archivo para lectura

    int c = fgetc(fp);             // Leer un solo carácter
    printf("%c\n", c);             // Imprimir char en stdout

    fclose(fp);                    // Cierre el archivo cuando haya terminado
```

Mira como, cuando abrimos el fichero con `fopen()`, nos devolvió el `FILE*` para que pudiéramos usarlo más tarde.

(Lo estoy omitiendo por brevedad, pero `fopen()` devolverá `NULL` si algo va mal, como file-not-found (archivo no encontrado), ¡así que deberías comprobar el error!)

Fíjate también en la `«r»` que pasamos—esto significa «abrir un flujo de texto para lectura». (Hay varias cadenas que podemos pasar a `fopen()` con significado adicional, como escribir, o añadir, etc.).

[1] Solíamos tener tres nuevas líneas diferentes en amplio efecto: Retorno de carro (CR, usado en los viejos Macs), Salto de línea (LF, usado en sistemas Unix), y Retorno de carro/Salto de línea (CRLF, usado en sistemas Windows). Afortunadamente, la introducción de OS X, al estar basado en Unix, redujo este número a dos

Después, usamos la función `fgetc()` para obtener un carácter del flujo. Te estarás preguntando, por qué he hecho que `c` sea un `int` en lugar de un `char`… ¡espera un momento!

Por último, cerramos el flujo cuando hemos terminado con él. Todos los flujos se cierran automáticamente cuando el programa se cierra, pero es de buena educación y buena limpieza cerrar explícitamente cualquier archivo cuando se termina con ellos.

El `FILE*` mantiene un registro de nuestra posición en el fichero. Así, las siguientes llamadas a `fgetc()` obtendrían el siguiente carácter del fichero, y luego el siguiente, hasta el final.

Pero eso parece complicado. Veamos si podemos hacerlo más fácil.

## 9.3 Fin de fichero: `EOF`

Existe un carácter especial definido como macro: `EOF`. Esto es lo que `fgetc()` devolverá cuando se haya alcanzado el final del fichero y haya intentado leer otro carácter.

Qué tal si comparto ese Fun Fact™(Hecho divertido / Hecho curioso), ahora. Resulta que `EOF` es la razón por la que `fgetc()` y funciones similares devuelven un `int` en lugar de un `char`. `EOF` no es un carácter propiamente dicho, y su valor probablemente cae fuera del rango de `char`. Dado que `fgetc()` necesita ser capaz de devolver cualquier byte **y** `EOF`, necesita ser un tipo más amplio que pueda contener más valores, así que será `int`. Pero a menos que estés comparando el valor devuelto con `EOF`, puedes saber, en el fondo, que es un `char`.

¡Muy bien! ¡Volvemos a la realidad! Podemos usar esto para leer todo el archivo en un bucle.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int c;

    fp = fopen("hello.txt", "r");

    while ((c = fgetc(fp)) != EOF)
        printf("%c", c);

    fclose(fp);
}
```

(Si la línea 10 es demasiado rara, basta con descomponerla empezando por los paréntesis más internos. Lo primero que hacemos es asignar el resultado de `fgetc()` a `c`, y *luego* comparamos *eso* con `EOF`. Lo hemos metido todo en una sola línea. Esto puede parecer difícil de leer, pero estúdialo—es C idiomático).

Y ejecutando esto, vemos:

```
Hello, world!
```

Pero aún así, estamos operando carácter por carácter, y muchos archivos de texto tienen más sentido a nivel de línea. Vamos a cambiar a eso.

### 9.3.1 Leer línea a línea

Entonces, ¿cómo podemos obtener una línea entera de una vez? `fgets()` ¡al rescate! Como argumentos, toma un puntero a un buffer `char` para almacenar bytes, un número máximo de bytes a leer, y un `FILE*`

del que leer. Devuelve `NULL` al final del archivo o en caso de error. `fgets()` es incluso lo suficientemente amable como para terminar con NUL la cadena cuando ha terminado[2].

Vamos a hacer un bucle similar al anterior, excepto que vamos a tener un fichero multilínea y lo vamos a leer línea a línea.

Aquí hay un archivo `quote.txt`:

```
Un hombre sabio puede aprender más de
una pregunta tonta que un tonto
puede aprender de una respuesta sabia.
                  --Bruce Lee
```

Y aquí hay algo de código que lee ese archivo línea por línea e imprime un número de línea antes de cada una:

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char s[1024];  // Suficientemente grande para cualquier línea
                   // que encuentre este programa.

    int linecount = 0;

    fp = fopen("quote.txt", "r");

    while (fgets(s, sizeof s, fp) != NULL)
        printf("%d: %s", ++linecount, s);

    fclose(fp);
}
```

Lo que da la salida:

```
1: Un hombre sabio puede aprender más de
2: una pregunta tonta que un tonto
3: puede aprender de una respuesta sabia.
4:                   --Bruce Lee
```

## 9.4 Entrada con formato

¿Sabes cómo puedes obtener una salida formateada con `printf()` (y, por tanto, `fprintf()` como veremos, más adelante)? Puede hacer lo mismo con `fscanf()`.

> Antes de empezar, deberías saber que usar funciones del estilo de `scanf()` puede ser peligroso con entradas no confiables. Si no especifica anchos de campo con tu `%s`, podrías desbordar el buffer. Peor aún, una conversión numérica inválida puede resultar en un comportamiento indefinido. Lo más seguro es usar `%s` con un ancho de campo, luego usar funciones como `strtol()` o `strtod()` para hacer las conversiones.

[2] Si el buffer no es lo suficientemente grande como para leer una línea entera, se detendrá la lectura a mitad de línea, y la siguiente llamada a `fgets()` continuará leyendo el resto de la línea

Dispongamos de un fichero con una serie de registros de datos. En este caso, ballenas, con nombre, longitud en metros y peso en toneladas. `ballenas.txt`:

```
blue 29.9 173
right 20.7 135
gray 14.9 41
humpback 16.0 30
```

Sí, podríamos leerlos con `fgets()` y luego analizar la cadena con `sscanf()` (y en eso es más resistente contra archivos corruptos), pero en este caso, vamos a usar `fscanf()` y sacarlo directamente.

La función `fscanf()` se salta los espacios en blanco al leer, y devuelve `EOF` al final del fichero o en caso de error.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char name[1024];  // Suficientemente grande para cualquier
                      //línea que encuentre este programa.
    float length;
    int mass;

    fp = fopen("whales.txt", "r");

    while (fscanf(fp, "%s %f %d", name, &length, &mass) != EOF)
        printf("%s whale, %d tonnes, %.1f meters\n", name, mass, length);

    fclose(fp);
}
```

Lo que da el resultado:

```
blue whale, 173 tonnes, 29.9 meters
right whale, 135 tonnes, 20.7 meters
gray whale, 41 tonnes, 14.9 meters
humpback whale, 30 tonnes, 16.0 meters
```

## 9.5 Escribir archivos de texto

Del mismo modo que podemos usar `fgetc()`, `fgets()` y `fscanf()` para leer flujos de texto, podemos usar `fputc()`, `fputs()` y `fprintf()` para escribir flujos de texto.

Para ello, tenemos que `fopen()` el archivo, en modo de escritura pasando `«w»` como segundo argumento. Abrir un fichero existente en modo `«w»` truncará instantáneamente ese fichero a 0 bytes para una sobreescritura completa.

Vamos a montar un programa sencillo que da salida a un archivo `output.txt` usando una variedad de funciones de salida.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int x = 32;

    fp = fopen("output.txt", "w");

    fputc('B', fp);
    fputc('\n', fp);   // Salto de linea
    fprintf(fp, "x = %d\n", x);
    fputs("Hello, world!\n", fp);

    fclose(fp);
}
```

Y esto produce un archivo, `output.txt`, con el siguiente contenido:

```
B
x = 32
Hello, world!
```

Dato curioso: como `stdout` es un archivo, podrías sustituir la línea 8 por:

```
fp = stdout;
```

y el programa habría dado salida a la consola en lugar de a un archivo. Pruébalo.

## 9.6 E/S de archivos binarios

Hasta ahora sólo hemos hablado de archivos de texto. Pero existe esa otra bestia que mencionamos al principio llamada archivos *binarios*, o flujos binarios.

Funcionan de forma muy similar a los archivos de texto, excepto que el subsistema de E/S no realiza ninguna traducción de los datos como haría con un archivo de texto. Con los archivos binarios, se obtiene un flujo de bytes sin procesar, y eso es todo.

La gran diferencia al abrir el fichero es que tienes que añadir una `«b»` al modo. Es decir, para leer un fichero binario, ábralo en modo «rb». Para escribir un fichero, ábrelo en modo «wb».

Como son flujos de bytes, y los flujos de bytes pueden contener caracteres NUL, y el carácter NUL es el marcador de fin de cadena en C, es raro que la gente use las funciones `fprintf()` y amigas para operar con ficheros binarios. En cambio, las funciones más comunes son `fread()` y `fwrite()`. Las funciones leen y escriben un número especificado de bytes en el flujo.

Para la demostración, escribiremos un par de programas. Uno escribirá una secuencia de valores de bytes en el disco de una sola vez. Y el segundo programa leerá un byte a la vez y los imprimirá[3].

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

[3]Normalmente el segundo programa leería todos los bytes a la vez, y *entonces* los imprimiría en un bucle. Eso sería más eficiente. Pero vamos para el valor de demostración, aquí

```
int main(void)
{
    FILE *fp;
    unsigned char bytes[6] = {5, 37, 0, 88, 255, 12};

    fp = fopen("output.bin", "wb");  // ¡modo wb para "escribir binario"!

    // En la llamada a fwrite, los argumentos son:
    //
    // * Puntero a los datos a escribir
    // * Tamaño de cada «pieza» de datos
    // * Recuento de cada «pieza» de datos
    // * ARCHIVO

    fwrite(bytes, sizeof(char), 6, fp);

    fclose(fp);
}
```

Esos dos argumentos centrales de `fwrite()` son bastante extraños. Pero básicamente lo que queremos decirle a la función es: «Tenemos elementos que son *así* de grandes, y queremos escribir *así* muchos de ellos». Esto hace que sea conveniente si usted tiene un registro de una longitud fija, y usted tiene un montón de ellos en una matriz. Sólo tienes que decirle el tamaño de un registro y cuántos escribir.

En el ejemplo anterior, le decimos que cada registro es del tamaño de un `char`, y tenemos 6 de ellos.

Ejecutando el programa obtenemos un fichero `output.bin`, pero al abrirlo en un editor de texto no aparece nada amigable. Son datos binarios, no texto. Y datos binarios aleatorios que me acabo de inventar.

Si lo paso por un programa hex dump[4], podemos ver la salida como bytes:

```
05 25 00 58 ff 0c
```

Y esos valores en hexadecimal coinciden con los valores (en decimal) que escribimos.

Pero ahora vamos a intentar leerlos de nuevo con un programa diferente. Este abrirá el fichero para lectura binaria (modo `«rb»`) y leerá los bytes de uno en uno en un bucle.

La función `fread()` devuelve el número de bytes leídos, o `0` en caso de EOF. Así que podemos hacer un bucle hasta que veamos eso, imprimiendo números a medida que avanzamos.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    unsigned char c;

    fp = fopen("output.bin", "rb"); // ¡rb para «leer binario»!

    while (fread(&c, sizeof(char), 1, fp) > 0)
        printf("%d\n", c);
```

[4] https://en.wikipedia.org/wiki/Hex_dump

```
    fclose(fp);
}
```

Y, al ejecutarlo, ¡vemos nuestros números originales!

```
5
37
0
88
255
12
```

Woo hoo!

### 9.6.1 `struct` y advertencias sobre números

Como vimos en la sección `struct`s, el compilador es libre de añadir relleno a una `struct` como considere oportuno. Y diferentes compiladores pueden hacer esto de manera diferente. Y el mismo compilador en diferentes arquitecturas podría hacerlo de forma diferente. Y el mismo compilador en las mismas arquitecturas podría hacerlo de manera diferente.

A lo que quiero llegar es a esto: no es portable simplemente `fwrite()` una `struct` entera a un fichero cuando no sabes dónde acabará el relleno.

¿Cómo solucionarlo? Espera un momento… veremos algunas formas de hacerlo después de analizar otro problema relacionado.

Números.

Resulta que no todas las arquitecturas representan los números en memoria de la misma manera.

Veamos una simple `fwrite()` de un número de 2 bytes. Lo escribiremos en hexadecimal para que cada byte sea claro. El byte más significativo tendrá el valor `0x12` y el menos significativo tendrá el valor `0x34`.

```
unsigned short v = 0x1234;  // Dos bytes, 0x12 y 0x34

fwrite(&v, sizeof v, 1, fp);
```

¿Qué termina en el flujo?

Bueno, parece que debería ser `0x12` seguido de `0x34`, ¿no?

Pero si ejecuto esto en mi máquina y volcado hexadecimal el resultado, me sale:

```
34 12
```

¡Están al revés! ¿Por qué?

Esto tiene algo que ver con lo que se llama el *endianess*[5] de la arquitectura. Algunas escriben primero los bytes más significativos y otras los menos significativos.

Esto significa que si escribes un número multibyte directamente desde la memoria, no puedes hacerlo de forma portable[6].

Un problema similar existe con el punto flotante. La mayoría de los sistemas usan el mismo formato para sus números en coma flotante, pero algunos no. No hay garantías.

[5] https://en.wikipedia.org/wiki/Endianess

[6] Y esta es la razón por la que usé bytes individuales en mis ejemplos `fwrite()` y `fread()`, arriba, astutamente

Entonces… ¿cómo podemos solucionar todos estos problemas con números y `structs` para que nuestros datos se escriban de forma portable?

El resumen es *serializar* los datos, que es un término general que significa tomar todos los datos y escribirlos en un formato que controlas, que es bien conocido, y programable, para funcionar de la misma manera en todas las plataformas.

Como puede imaginar, se trata de un problema resuelto. Hay un montón de librerías de serialización que puedes aprovechar, como *búferes de protocolo*[7] de Google, ahí fuera y listas para usar. Se encargarán de todos los detalles por ti, e incluso permitirán que los datos de tus programas en C interoperen con otros lenguajes que soporten los mismos métodos de serialización.

Hágase un favor a sí mismo y a todo el mundo. Serializa tus datos binarios cuando los escribas en un flujo. Esto mantendrá las cosas bien y portátiles, incluso si transfiere archivos de datos de una arquitectura a otra.

[7] https://en.wikipedia.org/wiki/Protocol_buffers

# Chapter 10

# `typedef`: Creación de nuevos tipos

Bueno, no tanto crear *nuevos* tipos como obtener nuevos nombres para tipos existentes. Suena un poco inútil en la superficie, pero realmente podemos utilizar esto para hacer nuestro código más limpio.

## 10.1 `typedef` en Teoría

Básicamente, se toma un tipo existente y se hace un alias para él con `typedef`.

Así:

```
typedef int antelope;  // Hacer de «antelope» un alias de «int»

antelope x = 10;       // El tipo «antelope» es el mismo que el tipo «int»
```

Puede tomar cualquier tipo existente y hacerlo. Usted puede incluso hacer un número de tipos con una lista de comas:

```
typedef int antelope, bagel, mushroom;  // Estos son todos «int»
```

Eso es muy útil, ¿verdad? ¿Que puedas escribir «mushroom» en lugar de «bagel»? Debes de estar muy emocionado con esta función.

De acuerdo, Profesor Sarcasmo… llegaremos a algunas aplicaciones más comunes de esto en un momento.

### 10.1.1 Alcance

`typedef` sigue las reglas de ámbito habituales.

Por esta razón, es bastante común encontrar `typedef` en el ámbito del archivo («global») para que todas las funciones puedan utilizar los nuevos tipos a voluntad.

## 10.2 `typedef` en la práctica

Así que renombrar `int` a otra cosa no es tan emocionante. Veamos dónde suele aparecer `typedef`.

### 10.2.1 `typedef` y `struct`s

A veces, una `struct` «estructura» se `typedef` «tipifica» con un nuevo nombre para que no tengas que escribir la palabra `struct` una y otra vez.

```
struct animal {
    char *name;
    int leg_count, speed;
};

// Nombre Original  Nuevo Nombre
//            |         |
//            v         v
//      |-----------| |----|
typedef struct animal animal;

struct animal y;  // Esto funciona
animal z;         // Esto también funciona porque «animal» es un alias
```

Personalmente, no me gusta esta práctica. Me gusta la claridad que tiene el código cuando añades la palabra `struct` al tipo; los programadores saben lo que obtienen. Pero es muy común, así que lo incluyo aquí.

Ahora quiero ejecutar exactamente el mismo ejemplo de una manera que se puede ver comúnmente. Vamos a poner el `struct animal` *en* el `typedef`. Puedes mezclarlo todo así:

```
//      Nombre Original
//            |
//            v
//      |-----------|
typedef struct animal {
    char *name;
    int leg_count, speed;
} animal;                       // <-- Nuevo nombre

struct animal y;  // Esto funciona
animal z;         // Esto también funciona porque «animal» es un alias
```

Es exactamente igual que el ejemplo anterior, pero más conciso.

Pero eso no es todo. Hay otro atajo común, que puedes ver en el código, usando lo que se llaman *estructuras anónimas*[1]. Resulta que en realidad, no necesitas nombrar la estructura en una variedad de lugares, y con `typedef` es uno de ellos.

Hagamos el mismo ejemplo con una estructura anónima:

```
//  ¡Estructura anónima! ¡No tiene nombre!
//         |
//         v
//      |----|
typedef struct {
    char *name;
    int leg_count, speed;
} animal;                       // <-- Nuevo nombre
```

[1] Hablaremos más de ellas más adelante

```
//struct animal y;  // ERROR: esto ya no funciona--¡no existe tal estructura!
animal z;           // Esto funciona porque «animal» es un alias
```

Como otro ejemplo, podríamos encontrar algo como esto:

```
typedef struct {
    int x, y;
} point;

point p = {.x=20, .y=40};

printf("%d, %d\n", p.x, p.y);  // 20, 40
```

### 10.2.2 `typedef` y otros tipos

No es que usar `typedef` con un tipo simple como `int` sea completamente inútil… te ayuda a abstraer los tipos para que sea más fácil cambiarlos después.

Por ejemplo, si tienes `float` por todo tu código en 100 zillones de sitios, va a ser doloroso cambiarlos todos a `double` si descubres que tienes que hacerlo más tarde por alguna razón.

Pero si te preparas un poco con:

```
typedef float app_float;

// y

app_float f1, f2, f3;
```

Si más tarde quieres cambiar a otro tipo, como `long double`, sólo tienes que cambiar el `typedef`:

```
//        voila!
//      |---------|
typedef long double app_float;

// y no es necesario cambiar esta línea:

app_float f1, f2, f3;  // Ahora todos estos son long double
```

### 10.2.3 `typedef` y punteros

Puedes hacer un tipo que sea un puntero.

```
typedef int *intptr;

int a = 10;
intptr x = &a;  // «intptr» es tipo «int*»
```

Realmente no me gusta esta práctica. Oculta el hecho de que `x` es un tipo puntero porque no se ve un `*` en la declaración.

En mi opinión, es mejor mostrar explícitamente que estás declarando un tipo puntero para que otros desarrolladores puedan verlo claramente y no confundan `x` con un tipo no puntero.

Pero en el último recuento, digamos, 832.007 personas tenían una opinión diferente.

### 10.2.4 `typedef` y mayúsculas

He visto todo tipo de mayúsculas en `typedef`.

```
typedef struct {
    int x, y;
} my_point;          // lower snake case

typedef struct {
    int x, y;
} MyPoint;          // CamelCase

typedef struct {
    int x, y;
} Mypoint;          // Leading uppercase

typedef struct {
    int x, y;
} MY_POINT;          // UPPER SNAKE CASE
```

La especificación C11 no dicta un modo u otro, y muestra ejemplos en mayúsculas y minúsculas.

K&R2 utiliza predominantemente las mayúsculas, pero muestra algunos ejemplos en mayúsculas y minúsculas (con `_t`).

Si utiliza una guía de estilo, cíñase a ella. Si no, hazte con una y cíñete a ella.

## 10.3 Arrays y `typedef`

La sintaxis es un poco extraña, y en mi experiencia esto se ve raramente, pero usted puede utilizar `typedef` en una matriz, de algún número de elementos.

```
// Hacer del tipo five_inst un array de 5 ints
typedef int five_ints[5];

five_ints x = {11, 22, 33, 44, 55};
```

No me gusta porque oculta la naturaleza de la variable array, pero se puede hacer.

# Chapter 11

# Punteros II: Aritmética

¡Es hora de entrar más en materia con una serie de nuevos temas sobre punteros! Si no estás al día con los punteros, echa un vistazo a la primera sección de la guía sobre el tema.

## 11.1 Aritmética de punteros

Resulta que se pueden hacer operaciones matemáticas con punteros, sobre todo sumas y restas.

Pero, ¿qué significa hacer eso?

En resumen, si tienes un puntero a un tipo, sumando uno al puntero, te mueves al siguiente elemento de ese tipo, el cual se encuentra, directamente después de él, en memoria.

Es **importante** recordar, que cuando movemos punteros y buscamos en diferentes lugares de la memoria, necesitamos asegurarnos de que **siempre** estamos apuntando a un lugar válido de la memoria, antes de hacer la desreferencia. Si nos vamos por las ramas e intentamos ver qué hay ahí, el comportamiento es indefinido y el resultado habitual es un fallo.

Esto es un poco complicado con La equivalencia de Array/Puntero pero vamos a intentarlo de todas formas.

### 11.1.1 Incrementando punteros

En primer lugar, tomemos una matriz de números.

```
int a[5] = {11, 22, 33, 44, 55};
```

A continuación, vamos a obtener un puntero al primer elemento de esa matriz:

```
int a[5] = {11, 22, 33, 44, 55};

int *p = &a[0];  // O "int *p = a;" funciona igual de bien
```

A continuación, vamos a imprimir el valor allí por desreferenciación del puntero:

```
printf("%d\n", *p);  // Imprime 11
```

Ahora vamos a utilizar la aritmética de punteros para imprimir el siguiente elemento de la matriz, el que está en el índice 1:

```
printf("%d\n", *(p + 1));  // Imprime 22!!
```

¿Qué ha pasado ahí? C sabe que `p` es un puntero a un `int`. Así que sabe el tamaño de un `int`[1] y sabe que debe saltarse esa cantidad de bytes para llegar al siguiente `int` después del primero.

De hecho, el ejemplo anterior podría escribirse de estas dos formas equivalentes:

```
printf("%d\n", *p);        // Imprime 11
printf("%d\n", *(p + 0));  // Imprime 11
```

porque añadiendo `0` a un puntero se obtiene el mismo puntero.

Pensemos en el resultado. Podemos iterar sobre elementos de un array de esta forma en lugar de usar un array:

```
int a[5] = {11, 22, 33, 44, 55};

int *p = &a[0];  // O "int *p = a;" funciona igual de bien

for (int i = 0; i < 5; i++) {
    printf("%d\n", *(p + i));  // ¡Igual que p[i]!
}
```

¡Y eso funciona igual que si utilizáramos la notación array! ¡Oooo! Cada vez más cerca de la equivalencia entre array y puntero. Más sobre esto en este capítulo.

Pero, ¿qué está pasando realmente aquí? ¿Cómo funciona?

¿Recuerdas que la memoria es como un gran array, donde un byte se almacena en cada índice del array?

Y el índice del array en la memoria tiene algunos nombres:

- Índice en memoria
- Localización
- Dirección
- *Puntero!*

Así que un puntero es un índice en la memoria, en algún lugar.

Por poner un ejemplo al azar, digamos que un número 3490 se almacenó en la dirección («índice») 23,237,489,202. Si tenemos un puntero `int` a ese 3490, el valor de ese puntero es 23,237,489,202… porque el puntero es la dirección de memoria. Diferentes palabras para la misma cosa.

Y ahora digamos que tenemos otro número, 4096, almacenado justo después del 3490 en la dirección 23,237,489,210 (8 más alto que el 3490 porque cada `int` en este ejemplo tiene 8 bytes de longitud).

Si añadimos `1` a ese puntero, en realidad salta `sizeof(int)` bytes hasta el siguiente `int`. Sabe que debe saltar tan lejos porque es un puntero `int`. Si fuera un puntero `float`, saltaría `sizeof(float)` bytes adelante para llegar al siguiente float.

Así que puedes ver el siguiente `int`, añadiendo `1` al puntero, el siguiente añadiendo `2` al puntero, y así sucesivamente.

### 11.1.2 Cambio de punteros

En la sección anterior vimos cómo podíamos añadir un entero a un puntero. Esta vez, vamos a *modificar el puntero en sí.*

[1] Recuerda que el operador `sizeof` te dice el tamaño en bytes de un objeto en memoria

Puede añadir (o restar) valores enteros directamente a (o desde) cualquier puntero.

Repitamos el ejemplo, pero con un par de cambios. En primer lugar, voy a añadir un `999` al final de nuestros números para utilizar como un valor centinela. Esto nos permitirá saber dónde está el final de los datos.

```
int a[] = {11, 22, 33, 44, 55, 999};  // Añade 999 aquí como centinela

int *p = &a[0];  // p señala el 11
```

Y también tenemos `p` apuntando al elemento en el índice `0` de `a`, es decir `11`, igual que antes.

Ahora empecemos a *incrementar* `p` para que apunte a los siguientes elementos del array. Haremos esto hasta que `p` apunte al `999`; es decir, lo haremos hasta que `*p == 999`:

```
while (*p != 999) {       // Mientras que la cosa a la que p señala no es 999
    printf("%d\n", *p);   // Imprimir
    p++;                  // Mueve(Incrementa) p para apuntar al siguiente int
}
```

Bastante loco, ¿verdad?

Cuando le damos una vuelta, primero `p` apunta a `11`. Luego incrementamos `p`, y apunta a `22`, y luego otra vez, apunta a `33`. Y así sucesivamente, hasta que apunta a `999` y salimos.

### 11.1.3 Restar punteros

También puedes restar un valor de un puntero para llegar a una dirección anterior, igual que antes.

Pero también podemos restar dos punteros para encontrar la diferencia entre ellos, por ejemplo, podemos calcular cuántos `int`s hay entre dos `int*`s. El problema es que esto sólo funciona dentro de un array[2]. Si los punteros apuntan a cualquier otra cosa, se obtiene un comportamiento indefinido.

¿Recuerdas que las cadenas son `char*`s en C? Veamos si podemos usar esto para escribir otra variante de `strlen()` para calcular la longitud de una cadena que utilice la resta de punteros.

La idea es que si tenemos un puntero al principio de la cadena, podemos encontrar un puntero al final de la cadena buscando el carácter `NUL`.

Y si tenemos un puntero al principio de la cadena, y hemos calculado el puntero al final de la cadena, podemos restar los dos punteros para obtener la longitud de la cadena.

```
#include <stdio.h>

int my_strlen(char *s)
{
    // Empezar a escanear desde el principio de la cadena
    char *p = s;

    // Escanear hasta encontrar el carácter NUL
    while (*p != '\0')
        p++;

    // Devuelve la diferencia de punteros
    return p - s;
```

[2] O cadena, que en realidad es un array de `char`s. Curiosamente, también puedes tener un puntero que haga referencia a *uno pasado* el final del array sin problema y seguir haciendo cálculos con él

```
}

int main(void)
{
    printf("%d\n", my_strlen("Hello, world!"));  // Imprime "13"
}
```

Recuerda que sólo puedes utilizar la resta de punteros entre dos punteros que apunten a la misma matriz.

## 11.2 Equivalencia entre matrices e identificadores

¡Por fin estamos listos para hablar de esto! Hemos visto un montón de ejemplos de lugares donde hemos entremezclado la notación array, pero vamos a dar la *fórmula fundamental de equivalencia array/puntero*:

```
a[b] == *(a + b)
```

¡Estudia eso! Son equivalentes y pueden utilizarse indistintamente.

He simplificado un poco, porque en mi ejemplo anterior `a` y `b` pueden ser ambas expresiones, y podríamos querer algunos paréntesis más para forzar el orden de las operaciones en caso de que las expresiones sean complejas.

La especificación es específica, como siempre, declarando (en C11 §6.5.2.1¶2):

> `E1[E2]` es idéntico a `(*((E1)+(E2)))`

pero eso es un poco más difícil de entender. Sólo asegúrate de incluir paréntesis si las expresiones son complicadas para que todas tus matemáticas ocurran en el orden correcto.

Esto significa que podemos *decidir* si vamos a usar la notación array o puntero para cualquier array o puntero (asumiendo que apunta a un elemento de un array).

Usemos un array y un puntero con ambas notaciones, array y puntero:

```
#include <stdio.h>

int main(void)
{
    int a[] = {11, 22, 33, 44, 55};

    int *p = a;  // p apunta al primer elemento de a, 11

    // Imprime todos los elementos del array de varias maneras:

    for (int i = 0; i < 5; i++)
        printf("%d\n", a[i]);      // Notación de matriz con a

    for (int i = 0; i < 5; i++)
        printf("%d\n", p[i]);      // Notación de matriz con p

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(a + i));  // Notación de puntero con a

    for (int i = 0; i < 5; i++)
```

```
        printf("%d\n", *(p + i));  // Notación de puntero con p

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(p++));    // Puntero móvil p
       //printf("%d\n", *(a++));    // Moviendo la variable de array a--¡ERROR!
}
```

Así que puedes ver que en general, si tienes una variable array, puedes usar puntero o noción de array para acceder a los elementos. Lo mismo con una variable puntero.

La única gran diferencia es que puedes *modificar* un puntero para que apunte a una dirección diferente, pero no puedes hacer eso con una variable array.

### 11.2.1 Equivalencia entre arrays e identificadores en las llamadas a funciones

Aquí es donde más te encontrarás con este concepto.

Si usted tiene una función que toma un argumento puntero, por ejemplo:

```
int my_strlen(char *s)
```

esto significa que puedes pasar un array o un puntero a esta función y que funcione.

```
char s[] = "Antelopes";
char *t = "Wombats";

printf("%d\n", my_strlen(s));  // Funciona!
printf("%d\n", my_strlen(t));  // Tambien funciona!
```

Y también es la razón por la que estas dos firmas de función son equivalentes:

```
int my_strlen(char *s)    // Funciona!
int my_strlen(char s[])   // Tambien funciona!
```

## 11.3 Punteros `void`

Ya has visto que la palabra clave `void` se usa con funciones, pero esto es un animal completamente separado y no relacionado.

A veces es útil tener un puntero a una cosa *de la que no sabes el tipo*.

Lo sé. Ten paciencia conmigo un segundo.

Hay básicamente dos casos de uso para esto.

1. Una función va a operar sobre algo byte a byte. Por ejemplo, `memcpy()` copia bytes de memoria de un puntero a otro, pero esos punteros pueden apuntar a cualquier tipo. `memcpy()` se aprovecha del hecho de que si iteras a través de `char*`s, estás iterando a través de los bytes de un objeto sin importar el tipo del objeto. Más sobre esto en la subsección Valores Multibyte.

2. Otra función está llamando a una función que tú le pasaste (un callback), y te está pasando datos. Tú conoces el tipo de los datos, pero la función que te llama no. Así que te pasa `void*`s—porque no

conoce el tipo—y tú los conviertes al tipo que necesitas. Las funciones incorporadas `qsort()`[3] y `bsearch()`[4] utilizan esta técnica.

Veamos un ejemplo, la función incorporada `memcpy()`:

```
void *memcpy(void *s1, void *s2, size_t n);
```

Esta función copia `n` bytes a partir de la dirección `s2` en la memoria a partir de la dirección `s1`.

Pero, ¡mira! ¡`s1` y `s2` son `void*`s! ¿Por qué? ¿Qué significa esto? Veamos más ejemplos.

Por ejemplo, podríamos copiar una cadena con `memcpy()` (aunque `strcpy()` es más apropiado para cadenas):

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

int main(void)
{
    char s[] = "Goats!";
    char t[100];

    memcpy(t, s, 7);  // Copia 7 bytes - ¡incluyendo el terminador NUL!

    printf("%s\n", t);  // "Goats!"
}
```

O podemos copiar algunos `int`s:

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

int main(void)
{
    int a[] = {11, 22, 33};
    int b[3];

    memcpy(b, a, 3 * sizeof(int));  // Copiar 3 ints de datos

    printf("%d\n", b[1]);  // 22
}
```

Esto es un poco salvaje… ¿has visto lo que hemos hecho con `memcpy()`? Copiamos los datos de `a` a `b`, pero tuvimos que especificar cuántos *bytes* copiar, y un `int` es más de un byte.

Bien, entonces… ¿cuántos bytes ocupa un `int`? Respuesta: depende del sistema. Pero podemos saber cuántos bytes ocupa cualquier tipo con el operador `sizeof`.

Así que.. ahí está la respuesta: un `int` ocupa `sizeof(int)` bytes de memoria para almacenarse.

Y si tenemos 3 de ellos en nuestro array, como en el ejemplo, todo el espacio usado para los 3 `int`s debe ser `3 * sizeof(int)`.

(En el ejemplo de la cadena, habría sido técnicamente más exacto copiar `7 * sizeof(char)` bytes. Pero los `char`s son siempre de un byte, por definición, así que se convierte en `7 * 1`).

[3] https://beej.us/guide/bgclr/html/split/stdlib.html#man-qsort
[4] https://beej.us/guide/bgclr/html/split/stdlib.html#man-bsearch

Incluso podríamos copiar un `float` o un `struct` con `memcpy()`. (Aunque esto es abusivo—deberíamos usar `=` para eso):

```
struct antelope my_antelope;
struct antelope my_clone_antelope;

// ...

memcpy(&my_clone_antelope, &my_antelope, sizeof my_antelope);
```

¡Mira qué versátil es `memcpy()`! Si tienes un puntero a un origen y un puntero a un destino, y tienes el número de bytes que quieres copiar, puedes copiar *cualquier tipo de datos*.

Imagina que no tuviéramos `void*`. Tendríamos que escribir funciones `memcpy()` especializadas para cada tipo:

```
memcpy_int(int *a, int *b, int count);
memcpy_float(float *a, float *b, int count);
memcpy_double(double *a, double *b, int count);
memcpy_char(char *a, char *b, int count);
memcpy_unsigned_char(unsigned char *a, unsigned char *b, int count);

// etc... blech!
```

Es mucho mejor usar `void*` y tener una función que lo haga todo.

Ese es el poder de `void*`. Puedes escribir funciones que no se preocupan por el tipo y aún así son capaces de hacer cosas con él.

Pero un gran poder conlleva una gran responsabilidad. Tal vez no tan grande en este caso, pero hay algunos límites.

1. No se puede hacer aritmética de punteros en un `void*`.
2. No se puede desreferenciar un `void*`.
3. No puedes usar el operador flecha en un `void*`, ya que también es una dereferencia.
4. No puedes usar la notación array en un `void*`, ya que también es una dereferencia [5].

Y si lo piensas, estas reglas tienen sentido. Todas esas operaciones se basan en conocer el tamaño del tipo de dato apuntado, y con `void*` no sabemos el tamaño del dato apuntado, ¡puede ser cualquier cosa!

Pero espera… si no puedes desreferenciar un `void*` ¿de qué te puede servir?

Como con `memcpy()`, te ayuda a escribir funciones genéricas que pueden manejar múltiples tipos de datos. ¡Pero el secreto es que, en el fondo, *conviertes el `void*` a otro tipo antes de usarlo*!

Y la conversión es fácil: sólo tienes que asignar a una variable del tipo deseado [6].

```
char a = 'X';  // Un solo carácter

void *p = &a;  // p señala a "X"
char *q = p;   // q también señala a "X"
```

[5] Porque recuerda que la notación de array es sólo una una desreferencia y algo de matemática de punteros, y no puedes desreferenciar un `void*`

[6] También puedes *castear* el `void*` a otro tipo, pero aún no hemos llegado a los castts

```
printf("%c\n", *p);  // ERROR--¡no se puede hacer referencia a void*!
printf("%c\n", *q);  // Imprime "X"
```

Escribamos nuestro propio `memcpy()` para probarlo. Podemos copiar bytes (`char`s), y sabemos el número de bytes porque se pasa.

```
void *my_memcpy(void *dest, void *src, int byte_count)
{
    // Convertir void*s en char*s
    char *s = src, *d = dest;

    // Ahora que tenemos char*s, podemos desreferenciarlos y copiarlos
    while (byte_count--) {
        *d++ = *s++;
    }

    // La mayoría de estas funciones devuelven el destino, por si acaso
    // que sea útil para el que llama.
    return dest;
}
```

Justo al principio, copiamos los `void*`s en `char*`s para poder usarlos como `char*`s. Así de fácil.

Luego un poco de diversión en un bucle while, donde decrementamos `byte_count` hasta que se convierte en false (`0`). Recuerda que con el post-decremento, se calcula el valor de la expresión (para que `while` lo use) y *entonces* se decrementa la variable.

Y algo de diversión en la copia, donde asignamos `*d = *s` para copiar el byte, pero lo hacemos con post-incremento para que tanto `d` como `s` se muevan al siguiente byte después de hacer la asignación.

Por último, la mayoría de las funciones de memoria y cadena devuelven una copia de un puntero a la cadena de destino por si el que llama quiere utilizarla.

Ahora que hemos hecho esto, sólo quiero señalar rápidamente que podemos utilizar esta técnica para iterar sobre los bytes de *cualquier* objeto en C, `float`s, `struct`s, ¡o cualquier cosa!

Vamos a ejecutar un ejemplo más del mundo real con la rutina incorporada `qsort()` que puede ordenar *cualquier cosa* gracias a la magia de los `void*`s.

(En el siguiente ejemplo, puede ignorar la palabra `const`, que aún no hemos tratado).

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

// El tipo de estructura que vamos a ordenar
struct animal {
    char *name;
    int leg_count;
};

// Esta es una función de comparación llamada por qsort() para ayudarle a determinar
// qué ordenar exactamente. La usaremos para ordenar un array de struct
// animales por leg_count.
int compar(const void *elem1, const void *elem2)
{
```

```
    // Sabemos que estamos ordenando struct animals, así que hagamos ambos
    // argumentos punteros a struct animals
    const struct animal *animal1 = elem1;
    const struct animal *animal2 = elem2;

    // Devolver <0 =0 o >0 dependiendo de lo que queramos ordenar.

    // Vamos a ordenar ascendentemente por leg_count, por lo que
    //devolveremos la diferencia en los leg_counts
    if (animal1->leg_count > animal2->leg_count)
        return 1;

    if (animal1->leg_count < animal2->leg_count)
        return -1;

    return 0;
}

int main(void)
{
    // Construyamos un array de 4 struct animals con diferentes
    // características. Este array está desordenado por leg_count, pero
    // lo ordenaremos en un segundo.
    struct animal a[4] = {
        {.name="Dog", .leg_count=4},
        {.name="Monkey", .leg_count=2},
        {.name="Antelope", .leg_count=4},
        {.name="Snake", .leg_count=0}
    };

    // Llama a qsort() para ordenar el array. qsort() necesita saber exactamente
    // qué ordenar estos datos, y lo haremos dentro de la función compar()
    //
    // Esta llamada dice: qsort array a, que tiene 4 elementos, y
    // cada elemento es sizeof(struct animal) bytes grande, y esta es la función
    // que comparará dos elementos cualesquiera.
    qsort(a, 4, sizeof(struct animal), compar);

    // Imprímelos todos
    for (int i = 0; i < 4; i++) {
        printf("%d: %s\n", a[i].leg_count, a[i].name);
    }
}
```

Mientras le des a `qsort()` una función que pueda comparar dos elementos que tengas en tu array a ordenar, puede ordenar cualquier cosa. Y lo hace sin necesidad de tener los tipos de los elementos codificados en cualquier lugar. `qsort()` simplemente reordena bloques de bytes basándose en los resultados de la función `compar()` que le pasaste.

# Chapter 12

# Asignación manual de memoria

Esta es una de las grandes áreas en las que C probablemente diverge de los lenguajes que ya conoces: *gestión manual de memoria*.

Otros lenguajes usan el conteo de referencias, la recolección de basura u otros medios para determinar cuándo asignar nueva memoria para algunos datos–y desasignarla cuando ninguna variable hace referencia a ella.

Y eso está bien. Está bien poder despreocuparse de ello, simplemente, eliminar todas las referencias a un elemento y confiar en que en algún momento se liberará la memoria asociada a él.

Pero C no es así, del todo.

Por supuesto, en C, algunas variables se asignan y se liberan automáticamente, cuando entran y salen del ámbito. Llamamos a estas variables automáticas. Son las típicas variables «locales» de ámbito de bloque. No hay problema.

Pero, ¿y si quieres que algo persista más tiempo que un bloque concreto? Aquí es donde entra en juego la gestión manual de la memoria.

Puedes decirle explícitamente a C que te asigne un número determinado de bytes que podrás utilizar a tu antojo. Y estos bytes permanecerán asignados hasta que liberes *explícitamente* esa memoria[1].

Es importante que liberes la memoria que hayas utilizado. Si no lo haces, lo llamamos una *fuga de memoria* y tu proceso continuará reservando esa memoria hasta que termine.

Si la asignaste manualmente, tienes que liberarla manualmente cuando termines de usarla.

¿Cómo lo hacemos? Vamos a aprender un par de nuevas funciones, y hacer uso del operador `sizeof` para ayudarnos a saber cuántos bytes asignar.

En el lenguaje común de C, los desarrolladores dicen que las variables locales automáticas se asignan «en la pila» y que la memoria asignada manualmente está «en el montón (heap)». La especificación no habla de ninguna de estas cosas, pero todos los desarrolladores de C, sabrán de qué estás hablando si las mencionas.

Todas las funciones que vamos a aprender en este capítulo se encuentran en `<stdlib.h>`.

## 12.1 Asignación y desasignación, `malloc()` y `free()`.

La función `malloc()` acepta un número de bytes para asignar, y devuelve un puntero void a ese bloque de memoria recién asignado.

[1] O hasta que el programa salga, en cuyo caso se liberará toda la memoria asignada por él. Asterisco: algunos sistemas te permiten asignar memoria que persiste después de que un programa salga, pero esto depende del sistema, está fuera del alcance de esta guía, y seguramente nunca lo harás por accidente

Como es un `void*`, puedes asignarlo al tipo de puntero que quieras… normalmente se corresponderá de alguna manera con el número de bytes que estás asignando.

Entonces… ¿cuántos bytes debo asignar? Podemos usar `sizeof` para ayudarnos con eso. Si queremos asignar espacio suficiente para un único `int`, podemos usar `sizeof(int)` y pasárselo a `malloc()`.

Después de que hayamos terminado con alguna memoria asignada, podemos llamar a `free()` para indicar que hemos terminado con esa memoria y que puede ser utilizada para otra cosa. Como argumento, pasas el mismo puntero que obtuviste de `malloc()` (o una copia del mismo). Es un comportamiento indefinido usar una región de memoria después de haberla liberado (`free()`).

Intentémoslo. Asignaremos suficiente memoria para un `int`, luego almacenaremos algo allí, y lo imprimiremos.

```
// Asignar espacio para un único int (sizeof(int) bytes-worth):

int *p = malloc(sizeof(int));

*p = 12;  // Almacenar algo allí

printf("%d\n", *p);  // Imprímelo: 12

free(p);  // Todo hecho con esa memoria

//*p = 3490;  // ERROR: ¡comportamiento indefinido! ¡Usar después de free()!
```

En ese ejemplo artificioso, realmente no hay ningún beneficio. Nosotros podríamos haber usado un `int` automático y habría funcionado. Pero veremos cómo la capacidad de asignar memoria de esta manera tiene sus ventajas, especialmente con estructuras de datos más complejas.

Otra cosa que verás comúnmente aprovecha el hecho de que `sizeof` puede darte el tamaño del tipo de resultado de cualquier expresión constante. Así que podrías poner un nombre de variable ahí también, y usar eso. Aquí hay un ejemplo de eso, igual que el anterior:

```
int *p = malloc(sizeof *p);  // *p es un int, igual que sizeof(int)
```

## 12.2 Comprobación de errores

Todas las funciones de asignación devuelven un puntero al nuevo tramo de memoria asignado, o `NULL` si la memoria no puede ser asignada por alguna razón.

Algunos sistemas operativos como Linux pueden configurarse de forma que `malloc()` nunca devuelva `NULL`, incluso si se ha quedado sin memoria. Pero a pesar de esto, siempre debes codificarlo con protecciones en mente.

```
int *x;

x = malloc(sizeof(int) * 10);

if (x == NULL) {
    printf("Error al asignar 10 ints\n");
    // Haga algo aquí para manejarlo
}
```

Este es un patrón común que verás, donde hacemos la asignación y la condición en la misma línea:

```
int *x;

if ((x = malloc(sizeof(int) * 10)) == NULL)
    printf("Error al asignar 10 ints\n");
    // haga algo aquí para manejarlo
}
```

## 12.3 Asignación de espacio para una matriz

Ya hemos visto cómo asignar espacio a una sola cosa; ¿qué pasa ahora con un montón de ellas en una matriz?

En C, un array es un montón de la misma cosa, en un tramo contiguo de memoria(espalda con espalda).

Podemos asignar un tramo contiguo de memoria, ya hemos visto cómo hacerlo. Si quisiéramos 3490 bytes de memoria, podríamos simplemente pedirlos:

```
char *p = malloc(3490);  // Voila
```

Y, de hecho, es una matriz de 3490 `char`s (también conocida como cadena), ya que cada `char` es 1 byte. En otras palabras, `sizeof(char)` es `1`.

Nota: no se ha hecho ninguna inicialización en la memoria recién asignada—está llena de basura. Límpiela con `memset()` si quiere, o vea `calloc()`, más abajo.

Pero podemos simplemente multiplicar el tamaño de la cosa que queremos por el número de elementos que queremos, y luego acceder a ellos usando la notación de puntero o de array.

Ejemplo

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

int main(void)
{
    // Asignar espacio para 10 ints
    int *p = malloc(sizeof(int) * 10);

    // Asígneles los valores 0-45:
    for (int i = 0; i < 10; i++)
        p[i] = i * 5;

    // Imprimir todos los valores 0, 5, 10, 15, ..., 40, 45
    for (int i = 0; i < 10; i++)
        printf("%d\n", p[i]);

    // Liberar el espacio
    free(p);
}
```

La clave está en la línea `malloc()`. Si sabemos que cada `int` necesita `sizeof(int)` bytes para contenerlo, y sabemos que queremos 10 de ellos, podemos simplemente asignar exactamente esa cantidad de bytes con:

```
sizeof(int) * 10
```

Y este truco funciona para todos los tipos. Basta con pasarlo a `sizeof` y multiplicarlo por el tamaño del array.

## 12.4 Una alternativa: `calloc()`.

Esta es otra función de asignación que funciona de forma similar a `malloc()`, con dos diferencias clave:

- En lugar de un único argumento, pasas el tamaño de un elemento, y el número de elementos que deseas asignar. Es como si estuviera hecho para asignar arrays.
- Borra la memoria a cero.

Todavía se usa `free()` para liberar la memoria obtenida mediante `calloc()`.

Aquí tienes una comparación entre `calloc()` y `malloc()`.

```
// Asigna espacio para 10 ints con calloc(), inicializado a 0:
int *p = calloc(10, sizeof(int));

// Asigna espacio para 10 ints con malloc(), inicializado a 0:
int *q = malloc(10 * sizeof(int));
memset(q, 0, 10 * sizeof(int));   // Pone en 0
```

De nuevo, el resultado es el mismo para ambos excepto que `malloc()` no pone a cero la memoria por defecto.

## 12.5 Cambio del tamaño asignado con `realloc()`.

Si ya has asignado 10 `int`s, pero más tarde decides que necesitas 20, ¿qué puedes hacer?

Una opción es asignar nuevo espacio y luego `memcpy()` en la memoria… pero resulta que a veces no necesitas mover nada. Y hay una función que es lo suficientemente inteligente como para hacer lo correcto en todas las circunstancias: `realloc()`.

Toma un puntero a memoria previamente ocupada (por `malloc()` o `calloc()`) y un nuevo tamaño para la región de memoria.

Entonces crece o decrece esa memoria, y devuelve un puntero a ella. A veces puede devolver el mismo puntero (si los datos no han tenido que ser copiados en otro lugar), o puede devolver uno diferente (si los datos han tenido que ser copiados).

Asegúrese de que cuando llama a `realloc()`, especifica el número de *bytes* a asignar, ¡y no sólo el número de elementos del array! Esto es:

```
num_floats *= 2;

np = realloc(p, num_floats);  // INCORRECTO: ¡se necesitan bytes, no número de elementos!

np = realloc(p, num_floats * sizeof(float));  // ¡Mejor!
```

Vamos a asignar un array de 20 `float`s, y luego cambiamos de idea y lo convertimos en un array de 40.

Vamos a asignar el valor de retorno de `realloc()` a otro puntero para asegurarnos de que no es `NULL`. Si no lo es, podemos reasignarlo a nuestro puntero original. (Si simplemente asignáramos el valor de retorno directamente al puntero original, perderíamos ese puntero si la función devolviera `NULL` y no tendríamos forma de recuperarlo).

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

int main(void)
{
    // Asignar espacio para 20 floats
    float *p = malloc(sizeof *p * 20);  // sizeof *p igual que sizeof(float)

    // Asígneles valores fraccionarios 0-1:
    for (int i = 0; i < 20; i++)
        p[i] = i / 20.0;

    // Pero, ¡espera! Hagamos de esto un array de 40 elementos
    float *new_p = realloc(p, sizeof *p * 40);

    // Comprueba si hemos reasignado correctamente
    if (new_p == NULL) {
        printf("Error reallocing\n");
        return 1;
    }

    // Si lo hiciéramos, podemos simplemente reasignar p
    p = new_p;

    // Y asigna a los nuevos elementos valores en el rango 1.0-2.0
    for (int i = 20; i < 40; i++)
        p[i] = 1.0 + (i - 20) / 20.0;

    // Imprime todos los valores 0-2 en los 40 elementos:
    for (int i = 0; i < 40; i++)
        printf("%f\n", p[i]);

    // Liberar el espacio
    free(p);
}
```

Fíjate en cómo tomamos el valor de retorno de `realloc()` y lo reasignamos a la misma variable puntero `p` que pasamos. Esto es bastante común.

Además, si la línea 7 te parece rara, con ese `sizeof *p` ahí, recuerda que `sizeof` funciona con el tamaño del tipo de la expresión. Y el tipo de `*p` es `float`, así que esa línea es equivalente a `sizeof(float)`.

### 12.5.1 Lectura de líneas de longitud arbitraria

Quiero demostrar dos cosas con este ejemplo completo.

1. El uso de `realloc()` para hacer crecer un buffer a medida que leemos más datos.
2. El Uso de `realloc()` para reducir el buffer al tamaño perfecto después de completar la lectura.

Lo que vemos aquí es un bucle que llama a `fgetc()` una y otra vez para añadir a un buffer, hasta que vemos que el último carácter es una nueva línea.

Una vez que encuentra la nueva línea, encoge el buffer al tamaño adecuado y lo devuelve.

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

// Leer una línea de tamaño arbitrario de un fichero
//
// Devuelve un puntero a la línea.
// Devuelve NULL en EOF o error.
//
// Es responsabilidad del que llama liberar() este puntero cuando termine de usarlo.
//
// Tenga en cuenta que esto elimina la nueva línea del resultado. Si necesita
// de él, probablemente sea mejor cambiar esto, a un do-while.

char *readline(FILE *fp)
{
    int offset = 0; // Indice del siguiente char en el buffer
   int bufsize = 4; // Preferiblemente con un tamaño inicial que sea potencia de 2
    char *buf; // El buffer
    int c; // El carácter que hemos leído

    buf = malloc(bufsize);  // Asignar búfer inicial

    if (buf == NULL)   // Comprobación de errores
        return NULL;

    // Bucle principal--leer hasta nueva línea o EOF
    while (c = fgetc(fp), c != '\n' && c != EOF) {

        // Comprueba si nos hemos quedado sin espacio en el buffer contabilidad
        // por el byte extra para el terminador NUL
        if (offset == bufsize - 1) {  // -1 para el terminador NUL
            bufsize *= 2;  // 2x el espacio

            char *new_buf = realloc(buf, bufsize);

            if (new_buf == NULL) {
                free(buf);   // En caso de error, libera y paga su fianza.
                return NULL;
            }

            buf = new_buf;  // Reasignación correcta
        }

        buf[offset++] = c;  // Añade el byte al buffer
    }

    // Llegamos a la nueva línea o a EOF...

    // Si es EOF y no leemos bytes, liberamos el buffer y
    // devuelve NULL para indicar que estamos en EOF:
    if (c == EOF && offset == 0) {
        free(buf);
        return NULL;
```

```
    }

    // Ajustar
    if (offset < bufsize - 1) {  // Si nos falta para el final
        char *new_buf = realloc(buf, offset + 1); // +1 por terminación NUL

        // Si tiene éxito, apunta buf a new_buf;
        // de lo contrario dejaremos buf donde está
        if (new_buf != NULL)
            buf = new_buf;
    }

    // Añadir el terminador NUL
    buf[offset] = '\0';

    return buf;
}

int main(void)
{
    FILE *fp = fopen("foo.txt", "r");

    char *line;

    while ((line = readline(fp)) != NULL) {
        printf("%s\n", line);
        free(line);
    }

    fclose(fp);
}
```

Cuando la memoria crece de esta manera, es común (aunque no es una ley) doblar el espacio necesario en cada paso para minimizar el número de `realloc()`s que ocurren.

Por último, tenga en cuenta que `readline()` devuelve un puntero a un buffer `malloc()`. Como tal, es responsabilidad de quien lo llama liberar explícitamente esa memoria cuando termine de usarla.

### 12.5.2 `realloc()` con `NULL`.

¡Hora del Trivial! Estas dos líneas son equivalentes:

```
char *p = malloc(3490);
char *p = realloc(NULL, 3490);
```

Esto podría ser conveniente si se tiene algún tipo de bucle de asignación y no se quiere poner en un caso especial el primer `malloc()`.

```
int *p = NULL;
int length = 0;

while (!done) {
    // Asigna 10 ints más:
```

```
    length += 10;
    p = realloc(p, sizeof *p * length);

    // Hacer cosas increíbles
    // ...
}
```

En ese ejemplo, no necesitábamos un `malloc()` inicial ya que `p` era `NULL` para empezar.

## 12.6 Asignaciones alineadas

Probablemente no vas a necesitar usar esto.

Y no quiero meterme demasiado en la maleza hablando de ello ahora mismo, pero hay una cosa llamada *alineación de memoria*, que tiene que ver con que la dirección de memoria (valor del puntero) sea múltiplo de un cierto número.

Por ejemplo, un sistema puede requerir que los valores de 16 bits comiencen en direcciones de memoria que sean múltiplos de 2. O que los valores de 64 bits comiencen en direcciones de memoria que sean múltiplos de 2, 4 u 8, por ejemplo. Depende de la CPU.

Algunos sistemas requieren este tipo de alineación para un acceso rápido a la memoria, o algunos incluso para el acceso a la memoria en absoluto.

Ahora, si usas `malloc()`, `calloc()`, o `realloc()`, C te dará un trozo de memoria bien alineado para cualquier valor, incluso `structs`. Funciona en todos los casos.

Pero puede haber ocasiones en las que sepas que algunos datos pueden ser alineados en un límite más pequeño, o deben ser alineados en uno más grande por alguna razón. Imagino que esto es más común en la programación de sistemas embebidos.

En esos casos, puede especificar una alineación con `aligned_alloc()`.

La alineación es una potencia entera de dos mayor que cero, así que `2`, `4`, `8`, `16`, etc. y se la das a `aligned_alloc()` antes del número de bytes que te interesan.

La otra restricción es que el número de bytes que asignes tiene que ser múltiplo de la alineación. Pero esto puede estar cambiando. Véase C Informe de defectos 460[2]

Hagamos un ejemplo, asignando en un límite de 64 bytes:

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

int main(void)
{
    // Asignar 256 bytes alineados en un límite de 64 bytes
    char *p = aligned_alloc(64, 256);  // 256 == 64 * 4

    // Copia una cadena e imprímela
    strcpy(p, "Hello, world!");
    printf("%s\n", p);

    // Liberar el espacio
```

[2] http://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm#dr_460

```
    free(p);
}
```

Quiero hacer un comentario sobre `realloc()` y `aligned_alloc()`. `realloc()` no tiene ninguna garantía de alineación, así que si necesitas obtener espacio reasignado alineado, tendrás que hacerlo por las malas con `memcpy()`.

Aquí tienes una función no estándar `aligned_realloc()`, por si la necesitas:

```
void *aligned_realloc(void *ptr, size_t old_size, size_t alignment, size_t size)
{
    char *new_ptr = aligned_alloc(alignment, size);

    if (new_ptr == NULL)
        return NULL;

    size_t copy_size = old_size < size? old_size: size;  // obtener min

    if (ptr != NULL)
        memcpy(new_ptr, ptr, copy_size);

    free(ptr);

    return new_ptr;
}
```

Tenga en cuenta que *siempre* copia datos, lo que lleva tiempo, mientras que `realloc()` real lo evitará si puede. Así que esto es poco eficiente. Evita tener que reasignar datos alineados a medida.

# Chapter 13

# Alcance

El alcance se refiere a, en qué contextos son visibles las variables.

## 13.1 Alcance del bloque

Este es el ámbito de casi todas las variables que definen los desarrolladores. Incluye lo que en otros lenguajes se denomina «ámbito de función», es decir, las variables que se declaran dentro de funciones.

La regla básica es que si has declarado una variable en un bloque delimitado por llaves, el ámbito de esa variable es ese bloque.

Si hay un bloque dentro de otro bloque, las variables declaradas en el bloque *interior* son locales a ese bloque y no pueden verse en el ámbito exterior.

Una vez que el ámbito de una variable termina, ya no se puede hacer referencia a esa variable, y se puede considerar que su valor se ha ido al gran cubo de bits[1] en el cielo.

Un ejemplo con ámbito anidado:

```
#include <stdio.h>

int main(void)
{
   int a = 12;      // Local al bloque exterior, pero visible en el bloque interior

    if  (a == 12) {
      int b = 99;    // Local al bloque interior, no visible en el bloque exterior

        printf("%d %d\n", a, b);  // OK: "12 99"
    }

    printf("%d\n", a);  // OK, todavía estamos en el ámbito de a

    printf("%d\n", b);  // ILEGAL, fuera del ámbito de b
}
```

[1] https://en.wikipedia.org/wiki/Bit_ bucket

### 13.1.1 Dónde definir las variables

Otro dato curioso es que se pueden definir variables en cualquier parte del bloque, dentro de lo razonable: tienen el ámbito de ese bloque, pero no se pueden utilizar antes de definirlas.

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    printf("%d\n", i);     // OK: "0"

   //printf("%d\n", j);  // ILEGAL--no se puede usar j antes de que esté definido

    int j = 5;

    printf("%d %d\n", i, j);   // OK: "0 5"
}
```

Históricamente, C exigía que todas las variables estuvieran definidas antes de cualquier código del bloque, pero esto ya no es así en el estándar C99.

### 13.1.2 Ocultación de variables

Si tienes una variable con el mismo nombre en un ámbito interno y en un ámbito externo, la del ámbito interno tiene preferencia mientras estés ejecutando en el ámbito interno. Es decir, *oculta* a la del ámbito externo durante todo su tiempo de vida.

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    {
        int i = 20;

        printf("%d\n", i);  // Ámbito interno i, 20 (el externo i está oculto)
    }

    printf("%d\n", i);  // Ámbito exterior i, 10
}
```

Te habrás dado cuenta de que en ese ejemplo acabo de lanzar un bloque en la línea 7, ¡ni siquiera una sentencia `for` o `if` para iniciarlo! Esto es perfectamente legal. A veces un desarrollador querrá agrupar un montón de variables locales para un cálculo rápido y hará esto, pero es raro de ver.

## 13.2 Alcance de fichero / Archivo

Si define una variable fuera de un bloque, esa variable tiene *ámbito de fichero*. Es visible en todas las funciones del archivo que vienen después de ella, y compartida entre ellas. (Una excepción es si un bloque define una variable del mismo nombre, ocultaría la que tiene ámbito de archivo).

Es lo más parecido a lo que se consideraría ámbito «global» en otro idioma.

Por ejemplo:

```
#include <stdio.h>

int shared = 10;    // ¡Alcance del fichero!
                    // ¡Visible a todo el archivo después de esto!

void func1(void)
{
    shared += 100;  // Ahora shared tiene 110
}

void func2(void)
{
    printf("%d\n", shared);  // Imprime "110"
}

int main(void)
{
    func1();
    func2();
}
```

Ten en cuenta que si `shared` se declarara al final del fichero, no compilaría. Tiene que ser declarado *antes* de que cualquier función lo use.

Hay otras formas de modificar elementos en el ámbito del fichero, concretamente con static y extern, pero hablaremos de ellas más adelante.

## 13.3 Ambito del bucle `for`

Realmente no sé cómo llamar a esto, ya que C11 §6.8.5.3¶1 no le da un nombre apropiado. También lo hemos hecho ya varias veces en esta guía. Es cuando declaras una variable dentro de la primera cláusula de un bucle `for`:

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

printf("%d\n", i);  // ILEGAL--i sólo está en el ámbito del bucle for
```

En ese ejemplo, el tiempo de vida de `i` comienza en el momento en que se define, y continúa durante la duración del bucle.

Si el cuerpo del bucle está encerrado en un bloque, las variables definidas en el bucle `for` son visibles desde ese ámbito interno.

A menos, por supuesto, que ese ámbito interno las oculte. Este ejemplo loco imprime `999` cinco veces:

```
#include <stdio.h>

int main(void)
{
```

```
    for (int i = 0; i < 5; i++) {
        int i = 999;  // Oculta la i en el ámbito del bucle for
        printf("%d\n", i);
    }
}
```

## 13.4 Nota sobre el alcance de las funciones

La especificación C hace referencia a *function scope* (alcance de funciones), pero se utiliza exclusivamente con *labels* (etiquetas), algo que aún no hemos discutido. Otro día hablaremos de ello.

# Chapter 14

# Tipos II: ¡Muchos más tipos!

Estamos acostumbrados a los tipos `char`, `int` y `float`, pero ha llegado el momento de pasar al siguiente nivel y ver qué más tenemos en el departamento de tipos.

## 14.1 Enteros con y sin signo

Hasta ahora hemos utilizado `int` como un tipo *signed (signado)*, es decir, un valor que puede ser negativo o positivo. Pero C también tiene tipos enteros *unsigned (sin signo)* que sólo pueden contener números positivos.

Estos tipos van precedidos de la palabra clave `unsigned`.

```
int a; // con signo
signed int a; // con signo
signed a; // con signo, «abreviatura» de «int» o «signed int», poco frecuente
unsigned int b; // sin signo
unsigned c; // unsigned, abreviatura de «unsigned int».
```

¿Por qué? ¿Por qué decidiste que sólo querías contener números positivos?

Respuesta: puedes obtener números más grandes en una variable sin signo que en una con signo.

Pero, ¿por qué?

Puedes pensar que los números enteros están representados por un cierto número de *bits*[1]. En mi ordenador, un `int` se representa con 64 bits.

Y cada permutación de bits que son `1` o `0` representa un número. Podemos decidir cómo repartir estos números.

Con los números con signo, utilizamos (aproximadamente) la mitad de las permutaciones para representar números negativos, y la otra mitad para representar números positivos.

Con números sin signo, usamos *todas* las permutaciones para representar números positivos.

En mi ordenador con `int`s de 64 bits que utiliza el complemento a dos[2] para representar números sin signo, tengo los siguientes límites en el rango de enteros:

[1] «Bit» es la abreviatura de *dígito binario.* El binario es otra forma de representar números. En lugar de los dígitos 0-9 a los que estamos acostumbrados, son los dígitos 0-1

[2] https://en.wikipedia.org/wiki/Two%27s_complement

| Tipo | Mínimo | Máximo |
|---|---|---|
| `int` | `-9,223,372,036,854,775,808` | `9,223,372,036,854,775,807` |
| `unsigned int` | `0` | `18,446,744,073,709,551,615` |

Fíjate en que el mayor `unsigned int` positivo es aproximadamente el doble de grande que el mayor `int` positivo. Así que puedes tener cierta flexibilidad.

## 14.2 Tipos de caracteres

¿Recuerdas `char`? ¿El tipo que podemos utilizar para contener un único carácter?

```
char c = 'B';

printf("%c\n", c);  // "B"
```

Tengo una sorpresa para ti: en realidad es un número entero.

```
char c = 'B';

// Cambia esto de %c a %d:
printf("%d\n", c);  // 66 (!!)
```

En el fondo, `char` no es más que un pequeño `int`, es decir, un entero que utiliza un único byte de espacio, limitando su rango a…

Aquí la especificación C se pone un poco rara. Nos asegura que un `char` es un único byte, es decir, `sizeof(char) == 1`. Pero entonces en C11 §3.6¶3 se sale de su camino para decir:

> Un byte está compuesto por una secuencia contigua de bits, cuyo número está definido por la implementación.

Espera… ¿qué? Algunos de ustedes pueden estar acostumbrados a la noción de que un byte es de 8 bits, ¿verdad? Quiero decir, eso es lo que es, ¿verdad? Y la respuesta es: «Casi seguro»[3]. Pero C es un lenguaje antiguo, y las máquinas de antes tenían, digamos, una opinión más *relajada* sobre cuántos bits había en un byte. Y a lo largo de los años, C ha conservado esta flexibilidad.

Pero asumiendo que tus bytes en C son de 8 bits, como lo son en prácticamente todas las máquinas del mundo que verás, el rango de un `char` es…

—Así que antes de que pueda decírtelo, resulta que `char`s puede ser con o sin signo dependiendo de tu compilador. A menos que lo especifiques explícitamente.

En muchos casos, sólo tener `char`s está bien porque no te importa el signo de los datos. Pero si necesitas `char`s con signo o sin signo, *debes* ser específico:

```
char a; // Puede ser con o sin signo
signed char b; // Definitivamente con signo
unsigned char c; // Definitivamente sin signo
```

OK, ahora, finalmente, podemos averiguar el rango de números si asumimos que un `char` es de 8 bits y su sistema utiliza la virtualmente universal representación de complemento a dos para con signo y sin signo[4].

[3] El término industrial para una secuencia de exactamente, indiscutiblemente 8 bits es un *octeto*

[4] En general, f usted tiene un $n$ bit número de complemento a dos, el rango con signo es $-2^{n-1}$ a $2^{n-1}-1$. Y el rango sin signo es de $0$ a $2^n-1$

Así que, asumiendo esas limitaciones, por fin podemos calcular nuestros rangos:

| Tipo `char` | Mínimo | Máximo |
|---|---|---|
| `signed char` | `-128` | `127` |
| `unsigned char` | `0` | `255` |

Y los rangos para `char` están definidos por la implementación.

A ver si lo entiendo. `char` es en realidad un número, así que ¿podemos hacer matemáticas con él?

Sí. Sólo recuerda mantener las cosas dentro del rango de un `char`. //Nota del traductor, esto es **extremadamente importante**, no lo olvides

```
#include <stdio.h>

int main(void)
{
    char a = 10, b = 20;

    printf("%d\n", a + b);  // 30!
}
```

¿Qué ocurre con los caracteres constantes entre comillas simples, como `'B'`? ¿Cómo puede eso tener un valor numérico?

La especificación también es imprecisa en este caso, ya que C no está diseñado para ejecutarse en un único tipo de sistema subyacente.

Pero asumamos por el momento que su juego de caracteres está basado en ASCII[5]para al menos los primeros 128 caracteres. En ese caso, la constante de carácter se convertirá en un `char` cuyo valor es el mismo que el valor ASCII del carácter.

Eso ha sido un trabalenguas. Pongamos un ejemplo:

```
#include <stdio.h>

int main(void)
{
    char a = 10;
    char b = 'B';  // Valor en ASCII 66

    printf("%d\n", a + b);  // 76!
}
```

Esto depende de su entorno de ejecución y del juego de caracteres utilizado[6]. Uno de los conjuntos de caracteres más populares hoy en día es Unicode[7] (que es un superconjunto de ASCII), por lo que para tus 0-9, A-Z, a-z y signos de puntuación básicos, casi seguro que obtendrás los valores ASCII.

## 14.3 Más tipos de enteros: `short`, `long`, `long long`

Hasta ahora hemos estado utilizando generalmente dos tipos enteros:

[5]https://en.wikipedia.org/wiki/ASCII
[6]https://en.wikipedia.org/wiki/ List_of information_system_character_sets
[7]https://en.wikipedia.org/wiki/Unicode

- `char`
- `int`

y recientemente hemos aprendido sobre las variantes sin signo de los tipos enteros. Y aprendimos que `char` era secretamente un pequeño `int` disfrazado. Así que sabemos que los `int`s pueden venir en múltiples tamaños de bit.

Pero hay un par de tipos enteros más que deberíamos ver, y los valores *mínimo minímo* y *máximo* que pueden contener.

Sí, he dicho «mínimo» dos veces. La especificación dice que estos tipos contendrán números de *al menos* estos tamaños, así que tu implementación puede ser diferente. El fichero de cabecera `<limits.h>` define macros que contienen los valores enteros mínimo y máximo; confíe en ello para estar seguro, y *nunca codifique o asuma estos valores*. Estos tipos adicionales son `short int`, `long int` y `long long int`. Normalmente, cuando se utilizan estos tipos, los desarrolladores de C omiten la parte `int` (por ejemplo, `long long`), y el compilador no tiene ningún problema.

```
// Estas dos líneas son equivalentes:
long long int x;
long long x;

// Y estos también:
short int x;
short x;
```

Veamos los tipos y tamaños de datos enteros en orden ascendente, agrupados por signatura.

| Tipo | Bytes mínimos | Valor mínimo | Valor máximo |
|---|---:|---:|---:|
| `char` | 1 | -127 or 0 | 127 or 255[8]. |
| `signed char` | 1 | -127 | 127 |
| `short` | 2 | -32767 | 32767 |
| `int` | 2 | -32767 | 32767 |
| `long` | 4 | -2147483647 | 2147483647 |
| `long long` | 8 | -9223372036854775807 | 9223372036854775807 |
| `unsigned char` | 1 | 0 | 255 |
| `unsigned short` | 2 | 0 | 65535 |
| `unsigned int` | 2 | 0 | 65535 |
| `unsigned long` | 4 | 0 | 4294967295 |
| `unsigned long long` | 8 | 0 | 18446744073709551615 |

No existe el tipo `long long long`. No puedes seguir añadiendo `long`s así. No seas tonto.

> Los aficionados a los complementos a dos habrán notado algo raro en esos números. ¿Por qué, por ejemplo, el `signed char` se detiene en -127 en lugar de -128? Recuerde: estos son sólo los mínimos requeridos por la especificación. Algunas representaciones numéricas (como signo y magnitud[a]) tienen un máximo de ±127.
>
> [a] https://en.wikipedia.org/wiki/Signed_number_representations#Signed_magnitude_representation

Ejecutemos la misma tabla en mi sistema de 64 bits y complemento a dos y veamos qué sale:

[8] Depende de si el valor por defecto de `char` es `signed char` o `unsigned char`

| Tipo | Mis Bytes | Valor mínimo | Valor máximo |
|---|---|---|---|
| `char` | 1 | -128 | 127[9] |
| `signed char` | 1 | -128 | 127 |
| `short` | 2 | -32768 | 32767 |
| `int` | 4 | -2147483648 | 2147483647 |
| `long` | 8 | -9223372036854775808 | 9223372036854775807 |
| `long long` | 8 | -9223372036854775808 | 9223372036854775807 |
| `unsigned char` | 1 | 0 | 255 |
| `unsigned short` | 2 | 0 | 65535 |
| `unsigned int` | 4 | 0 | 4294967295 |
| `unsigned long` | 8 | 0 | 18446744073709551615 |
| `unsigned long long` | 8 | 0 | 18446744073709551615 |

Eso es un poco más sensato, pero podemos ver cómo mi sistema tiene límites mayores que los mínimos de la especificación.

Entonces, ¿qué son las macros en `<limits.h>`?

| Tipo | Macro mínima | Macro máxima |
|---|---|---|
| `char` | `CHAR_MIN` | `CHAR_MAX` |
| `signed char` | `SCHAR_MIN` | `SCHAR_MAX` |
| `short` | `SHRT_MIN` | `SHRT_MAX` |
| `int` | `INT_MIN` | `INT_MAX` |
| `long` | `LONG_MIN` | `LONG_MAX` |
| `long long` | `LLONG_MIN` | `LLONG_MAX` |
| `unsigned char` | `0` | `UCHAR_MAX` |
| `unsigned short` | `0` | `USHRT_MAX` |
| `unsigned int` | `0` | `UINT_MAX` |
| `unsigned long` | `0` | `ULONG_MAX` |
| `unsigned long long` | `0` | `ULLONG_MAX` |

Fíjate que hay una forma oculta de determinar si un sistema utiliza `char`s con signo o sin signo. Si `CHAR_MAX == UCHAR_MAX`, debe ser sin signo.

Fíjate también en que no hay macro mínima para las variantes «sin signo»: son simplemente «0».

## 14.4 Más Float: `double` y `long double`.

Veamos qué dice la especificación sobre los números de coma flotante en §5.2.4.2.2¶1-2:

> Los siguientes parámetros se utilizan para definir el modelo para cada tipo de punto flotante:
>
> | Parametro | Definición |
> |---|---|
> | $s$ | signo ($\pm 1$) |
> | $b$ | base o radix de la representación del exponente (un entero $> 1$) |
> | $e$ | exponente (un número entero entre un mínimo $e_{min}$ y un máximo $e_{max}$) |

[9] Mi `char` está con signo.

| | |
|---|---|
| $p$ | precisión (el número de dígitos de base-$b$ en el significando) |
| $f_k$ | enteros no negativos menores que $b$ (los dígitos significantes) |

Un *número de punto flotante* ($x$) se define mediante el siguiente modelo:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{min} \leq e \leq e_{max}$$

Espero que eso te lo haya aclarado.

De acuerdo. Retrocedamos un poco y veamos qué es práctico.

Nota: nos referimos a un montón de macros en esta sección. Se pueden encontrar en la cabecera `<float.h>`.

Los números en coma flotante se codifican en una secuencia específica de bits (Formato IEEE-754[10] es tremendamente popular) en bytes.

Profundizando un poco más, el número se representa básicamente como el *significando* (que es la parte numérica–los dígitos significativos propiamente dichos, también llamados a veces la *mantisa*) y el *exponente*, que es a qué potencia elevar los dígitos. Recordemos que un exponente negativo puede hacer que un número sea más pequeño.

Imaginemos que usamos $10$ como número a elevar por un exponente. Podríamos representar los siguientes números utilizando un significando de $12345$, y exponentes de $-3$, $4$, y $0$ para codificar los siguientes valores en coma flotante:

$12345 \times 10^{-3} = 12.345$

$12345 \times 10^{4} = 123450000$

$12345 \times 10^{0} = 12345$

Para todos esos números, el significante sigue siendo el mismo. La única diferencia es el exponente.

En tu máquina, la base para el exponente es probablemente $2$, no $10$, ya que a los ordenadores les gusta el binario. Puedes comprobarlo imprimiendo la macro `FLT_RADIX`.

Así que tenemos un número que está representado por un número de bytes, codificados de alguna manera. Como hay un número limitado de patrones de bits, se puede representar un número limitado de números en coma flotante.

Pero más concretamente, sólo se puede representar con precisión un cierto número de dígitos decimales significativos.

¿Cómo conseguir más? Utilizando tipos de datos más grandes.

Y tenemos un par de ellos. Ya conocemos `float`, pero para más precisión tenemos `double`. Y para aún más precisión, tenemos `long double` (no relacionado con `long int` excepto por el nombre).

La especificación no especifica cuántos bytes de almacenamiento debe ocupar cada tipo, pero en mi sistema podemos ver los incrementos de tamaño relativos:

| Tipo | `sizeof` |
|---|---|
| `float` | 4 |
| `double` | 8 |
| `long double` | 16 |

[10] https://en.wikipedia.org/wiki/IEEE_754

Así que cada uno de los tipos (en mi sistema) utiliza esos bits adicionales para obtener más precisión.

¿Pero de cuánta precisión estamos hablando? ¿Cuántos números decimales pueden ser representados por estos valores?

Bueno, C nos proporciona un montón de macros en `<float.h>` para ayudarnos a averiguarlo.

La cosa se complica un poco si estás usando un sistema de base-2 (binario) para almacenar los números (que es prácticamente todo el mundo en el planeta, probablemente incluyéndote a ti), pero ten paciencia conmigo mientras lo resolvemos.

### 14.4.1 ¿Cuántas cifras decimales?

La pregunta del millón es: «¿Cuántos dígitos decimales significativos puedo almacenar en un determinado tipo de coma flotante para que me salga el mismo número decimal al imprimirlo?».

El número de dígitos decimales que puedes almacenar en un tipo de coma flotante y obtener con seguridad el mismo número al imprimirlo viene dado por estas macros:

| Tipo | Dígitos decimales que puede almacenar | Mínimo |
|---|---|---|
| `float` | `FLT_DIG` | 6 |
| `double` | `DBL_DIG` | 10 |
| `long double` | `LDBL_DIG` | 10 |

En mi sistema, `FLT_DIG` es 6, así que puedo estar seguro de que si imprimo un `float` de 6 dígitos, obtendré lo mismo de vuelta. (Podrían ser más dígitos—algunos números volverán correctamente con más dígitos. Pero sin duda me devolverá 6).

Por ejemplo, imprimiendo `float`s siguiendo este patrón de dígitos crecientes, aparentemente llegamos a 8 dígitos antes de que algo vaya mal, pero después de eso volvemos a 7 dígitos correctos.

```
0.12345
0.123456
0.1234567
0.12345678
0.123456791  <-- Las cosas empiezan a ir mal
0.1234567910
```

Hagamos otra demostración. En este código tendremos dos `float`s que contendrán números que tienen `FLT_DIG` dígitos decimales significativos[11]. Luego los sumamos, para lo que deberían ser 12 dígitos decimales significativos. Pero eso es más de lo que podemos almacenar en un `float` y recuperar correctamente como una cadena—así que vemos que cuando lo imprimimos, las cosas empiezan a ir mal después del 7º dígito significativo.

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

int main(void)
{
    // Ambos números tienen 6 dígitos significativos, por lo que pueden ser
    // almacenados con precisión en un float:
```

[11] Este programa se ejecuta como indican sus comentarios en un sistema con `FLT_DIG` de `6` que utiliza números en coma flotante IEEE-754 base-2. De lo contrario podrías obtener una salida diferente

```
    float f = 3.14159f;
    float g = 0.00000265358f;

    printf("%.5f\n", f);   // 3.14159       -- Correcto!
    printf("%.11f\n", g);  // 0.00000265358 -- Correcto!

    // Ahora súmalos
    f += g;                // 3.14159265358 es lo que f _debería_ ser

    printf("%.11f\n", f);  // 3.14159274101 -- ¡Mal!
}
```

(El código anterior tiene una `f` después de las constantes numéricas—esto indica que la constante es de tipo `float`, en oposición al valor por defecto de `double`. Más sobre esto más adelante).

Recuerde que `FLT_DIG` es el número seguro de dígitos que puede almacenar en un `float` y recuperar correctamente.

A veces puedes sacar uno o dos más. Pero otras veces sólo recuperarás dígitos `FLT_DIG`. Lo más seguro: si almacenas cualquier número de dígitos hasta e incluyendo `FLT_DIG` en un `float`, seguro que los recuperas correctamente.

Así que ésa es la historia de `FLT_DIG`. Fin.

…¿O no?

### 14.4.2 Conversión a decimal y viceversa

Pero almacenar un número de base 10 en un número de coma flotante y recuperarlo es sólo la mitad de la historia.

Resulta que los números de coma flotante pueden codificar números que requieren más decimales para imprimirse completamente. Lo que ocurre es que tu número decimal grande puede no corresponder a uno de esos números.

Es decir, cuando miras los números de coma flotante de uno a otro, hay un hueco. Si intentas codificar un número decimal en ese hueco, usará el número de coma flotante más cercano. Por eso sólo puedes codificar `FLT_DIG` para un `float`.

¿Pero qué pasa con los números de coma flotante que *no* están en el hueco? ¿Cuántos lugares necesita para imprimirlos con precisión?

Otra forma de formular esta pregunta es, para cualquier número en coma flotante, ¿cuántos dígitos decimales tengo que conservar si quiero volver a convertir el número decimal en un número idéntico en coma flotante? Es decir, ¿cuántos dígitos tengo que imprimir en base 10 para recuperar **todos** los dígitos en base 2 del número original?

A veces pueden ser sólo unos pocos. Pero para estar seguro, querrás convertir a decimal con un cierto número seguro de decimales. Ese número está codificado en las siguientes macros:

| Macro | Descripción |
|---|---|
| `FLT_DECIMAL_DIG` | Número de dígitos decimales codificados en un `float`. |
| `DBL_DECIMAL_DIG` | Número de dígitos decimales codificados en un `double`. |
| `LDBL_DECIMAL_DIG` | Número de dígitos decimales codificados en un `long double`. |
| `DECIMAL_DIG` | Igual que la codificación más amplia, `LDBL_DECIMAL_DIG`. |

Veamos un ejemplo en el que `DBL_DIG` es 15 (por lo que es todo lo que podemos tener en una constante), pero `DBL_DECIMAL_DIG` es 17 (por lo que tenemos que convertir a 17 números decimales para conservar todos los bits del `double` original).

Asignemos el número de 15 dígitos significativos `0.123456789012345` a `x`, y asignemos el número de 1 dígito significativo `0.0000000000000006` a `y`.

```
x es exactamente: 0.12345678901234500    Impreso con 17 decimales
y es exactamente: 0.00000000000000060
```

Pero sumémoslos. Esto debería dar `0.1234567890123456`, pero es más que `DBL_DIG`, así que podrían pasar cosas raras… veamos:

```
x + y no es del todo correcto: 0.12345678901234559    ¡Debería terminar en 4560!
```

Eso nos pasa por imprimir más que `DBL_DIG`, ¿no? Pero fíjate… ¡ese número de arriba es exactamente representable tal cual!

Si asignamos `0.12345678901234559` (17 dígitos) a `z` y lo imprimimos, obtenemos:

```
z es exactamente: 0.12345678901234559   ¡17 dígitos correctos! ¡Más que DBL_DIG!
```

Si hubiéramos truncado `z` a 15 dígitos, no habría sido el mismo número. Por eso, para conservar todos los bits de un `doble`, necesitamos `DBL_DECIMAL_DIG` y no sólo el menor `DBL_DIG`.

Dicho esto, está claro que cuando estamos jugando con números decimales en general, no es seguro imprimir más de `FLT_DIG`, `DBL_DIG`, o `LDBL_DIG` dígitos para ser sensato en relación con los números originales de base 10 y cualquier matemática posterior.

Pero cuando convierta de `float` a una representación decimal y *de vuelta* a `float`, use definitivamente `FLT_DECIMAL_DIG` para hacerlo, de forma que todos los bits se conserven exactamente.

## 14.5 Tipos numéricos constantes

Cuando escribes un número constante, como `1234`, tiene un tipo. Pero, ¿de qué tipo es? Veamos cómo decide C qué tipo es la constante, y cómo forzarle a elegir un tipo específico.

### 14.5.1 Hexadecimal y octal

Además de los viejos decimales como los que cocía la abuela, C también admite constantes de diferentes bases.

Si encabeza un número con `0x`, se lee como un número hexadecimal:

```
int a = 0x1A2B;   // Hexadecimal
int b = 0x1a2b;   // Las mayúsculas y minúsculas
                  // no importan para los dígitos hexadecimales

printf("%x", a);  // Imprime un número hexadecimal, «1a2b»
```

Si precede un número con un `0`, se lee como un número octal:

```
int a = 012;
```

```
printf("%o\n", a);  // Imprime un número octal, «12»
```

Esto es especialmente problemático para los programadores principiantes que tratan de rellenar los números decimales a la izquierda con «0» para alinear las cosas bien y bonito, cambiando inadvertidamente la base del número:

```
int x = 11111;  // Decimal 11111
int y = 00111;  // Decimal 73 (Octal 111)
int z = 01111;  // Decimal 585 (Octal 1111)
```

#### 14.5.1.1 Nota sobre el binario

Una extensión no oficial[12] en muchos compiladores de C permite representar un número binario con un prefijo `0b`:

```
int x = 0b101010;    // Número binario 101010

printf("%d\n", x);   // Imprime 42 en decimal
```

No existe un especificador de formato `printf()` para imprimir un número binario. Hay que hacerlo carácter a carácter con operadores bit a bit.

### 14.5.2 Constantes enteras

Puedes forzar que un entero constante sea de un tipo determinado añadiéndole un sufijo que indique el tipo.

Haremos algunas asignaciones para demostrarlo, pero la mayoría de los desarrolladores omiten los sufijos a menos que sea necesario para ser precisos. El compilador es bastante bueno asegurándose de que los tipos son compatibles.

```
int           x = 1234;
long int      x = 1234L;
long long int x = 1234LL

unsigned int           x = 1234U;
unsigned long int      x = 1234UL;
unsigned long long int x = 1234ULL;
```

El sufijo puede ser mayúscula o minúscula. Y la `U` y la `L` o `LL` pueden aparecer indistintamente en primer lugar.

| Tipo | Sufijo |
|---|---|
| `int` | Sin sufijo |
| `long int` | `L` |
| `long long int` | `LL` |
| `unsigned int` | `U` |
| `unsigned long int` | `UL` |
| `unsigned long long int` | `ULL` |

[12] Realmente me sorprende que C no tenga esto en la especificación todavía. En el documento C99 Rationale, escriben: «Una propuesta para añadir constantes binarias fue rechazada por falta de precedentes y utilidad insuficiente». Lo que parece una tontería a la luz de algunas de las otras características que han incluido. Apuesto a que una de las próximas versiones lo tiene.

En la tabla mencioné que «sin sufijo» significa `int`… pero en realidad es más complejo que eso.

Entonces, ¿qué sucede cuando usted tiene un número sin sufijo como:

```
int x = 1234;
```

¿Qué tipo es?

Lo que C hace generalmente es elegir el tipo más pequeño a partir de `int` que pueda contener el valor.

Pero específicamente, eso depende de la base del número (decimal, hexadecimal, o octal).

La especificación tiene una gran tabla que indica qué tipo se utiliza para cada valor no fijo. De hecho, voy a copiarla íntegramente aquí.

C11 §6.4.4.1¶5 dice: «El tipo de una constante entera es el primero de la lista correspondiente en la que se puede representar su valor».

Y luego pasa a mostrar esta tabla:

| Sufijo | Constante decimal | Constante/ Octal o Hexadecimal |
|---|---|---|
| Sin sufijo | `int`<br>`long int` | `int`<br>`unsigned int`<br>`long int`<br>`unsigned long int`<br>`long long int`<br>`unsigned long long int` |
| `u` o `U` | `unsigned int`<br>`unsigned long int`<br>`unsigned long long int` | `unsigned int`<br>`unsigned long int`<br>`unsigned long long int` |
| `l` o `L` | `long int`<br>`long long int` | `long int`<br>`unsigned long int`<br>`long long int`<br>`unsigned long long int` |
| Ambos `u` o `U` y `l` o `L` | `unsigned long int`<br>`unsigned long long int` | `unsigned long int`<br>`unsigned long long int` |
| `ll` o `LL` | `long long int` | `long long int`<br>`unsigned long long int` |
| Ambos `u` o `U` y `ll` o `LL` | `unsigned long long int` | `unsigned long long int` |

Lo que esto quiere decir es que, por ejemplo, si especificas un número como `123456789U`, primero C verá si puede ser `unsigned int`. Si no cabe ahí, probará con `unsigned long int`. Y luego `unsigned long long int`. Usará el tipo más pequeño que pueda contener el número.

### 14.5.3 Constantes en coma flotante

Uno pensaría que una constante en coma flotante como `1.23` tendría un tipo por defecto `float`, ¿verdad?

¡Sorpresa! ¡Resulta que los números de coma flotante sin sufijo son del tipo `double`! ¡Feliz cumpleaños atrasado!

Puede forzar que sea de tipo `float` añadiendo una `f` (o `F`—no distingue mayúsculas de minúsculas). Puedes forzar que sea del tipo `long double` añadiendo `l` (o `L`).

| Tipo | Sufijo |
|---|---|
| `float` | `F` |
| `double` | None |
| `long double` | `L` |

Por ejemplo:

```
float x       = 3.14f;
double x      = 3.14;
long double x = 3.14L;
```

Todo este tiempo, sin embargo, hemos estado haciendo esto, ¿verdad?

```
float x = 3.14;
```

¿No es la izquierda un `float` y la derecha un `double`? Sí. Pero C es bastante bueno con las conversiones numéricas automáticas, así que es más común tener una constante de coma flotante fijada que no fijada. Más adelante hablaremos de ello.

#### 14.5.3.1 Notación científica

¿Recuerdas que antes hablamos de cómo se puede representar un número en coma flotante mediante un significando, una base y un exponente?

Bueno, hay una forma común de escribir un número de este tipo, que se muestra aquí seguido de su equivalente más reconocible, que es lo que obtienes cuando ejecutas las matemáticas:

$1.2345 \times 10^3 = 1234.5$

Escribir números de la forma $s \times b^e$ se denomina *notación_científica*[13]. En C, se escriben utilizando la «notación E», por lo que son equivalentes:

| Notación científica | Notación E |
|---|---|
| $1.2345 \times 10^{-3} = 0.0012345$ | `1.2345e-3` |
| $1.2345 \times 10^8 = 123450000$ | `1.2345e+8` |

Puede imprimir un número en esta notación con `%e`:

```
printf("%e\n", 123456.0);  // Impresiones 1,234560e+05
```

Un par de datos curiosos sobre la notación científica:

- No tienes que escribirlos con un solo dígito antes del punto decimal. Cualquier número de cifras puede ir delante.

  ```
  double x = 123.456e+3;  // 123456
  ```

Sin embargo, al imprimirlo, cambiará el exponente para que haya sólo un dígito delante del punto decimal.

[13] https://en.wikipedia.org/wiki/ Scientific_notation

- El más se puede dejar fuera del exponente, ya que es por defecto, pero esto es poco común en la práctica por lo que he visto.

```
1.2345e10 == 1.2345e+10
```

- Puede aplicar los sufijos `F` o `L` a las constantes de anotación E:

```
1.2345e10F
1.2345e10L
```

#### 14.5.3.2 Constantes hexadecimales en coma flotante

Pero espera, ¡todavía hay que flotar más!

Resulta que también existen constantes hexadecimales de coma flotante.

Funcionan de forma similar a los números decimales en coma flotante, pero empiezan por «0x», igual que los números enteros.

El truco es que *debes* especificar un exponente, y este exponente produce una potencia de 2. Es decir: $2^x$.

Y entonces se usa una `p` en lugar de una `e` al escribir el número:

Así que `0xa.1p3` es $10.0625 \times 2^3 == 80.5$.

Cuando usamos constantes hexadecimales en coma flotante, Podemos imprimir notación científica hexadecimal con `%a`:

```
double x = 0xa.1p3;

printf("%a\n", x);  // 0x1.42p+6
printf("%f\n", x);  // 80.500000
```

# Chapter 15

# Tipos III: Conversiones

En este capítulo, queremos hablar de la conversión de un tipo a otro. C tiene una variedad de formas de hacer esto, y algunas pueden ser un poco diferentes a las que estás acostumbrado en otros lenguajes.

Antes de hablar de cómo hacer que las conversiones ocurran, hablemos de cómo funcionan cuando *ocurren*.

## 15.1 Conversiones de cadenas

A diferencia de muchos lenguajes, C no realiza las conversiones de cadena a número (y viceversa) de una forma tan ágil como lo hace con las conversiones numéricas.

Para ello, tendremos que llamar a funciones que hagan el trabajo sucio.

### 15.1.1 Valor numérico a cadena

Cuando queremos convertir un número en una cadena, podemos utilizar `sprintf()` (se pronuncia *SPRINT-f*) o `snprintf()` (*s-n-print-f*)[1].

Básicamente funcionan como `printf()`, excepto que dan salida a una cadena en su lugar, y puedes imprimir esa cadena más tarde, o lo que sea.

Por ejemplo, convirtiendo parte del valor π en una cadena:

```
#include <stdio.h>

int main(void)
{
    char s[10];
    float f = 3.14159;

   // Convertir «f» en cadena, almacenando en «s», escribiendo como máximo 10 caracteres
    // incluido el terminador NUL

    snprintf(s, 10, "%f", f);

    printf("String value: %s\n", s);  // Valor de la cadena: 3.141590
}
```

[1] Son lo mismo, salvo que `snprintf()` permite especificar un número máximo de bytes de salida, evitando que se sobrepase el final de la cadena. Así que es más seguro

Así que puedes usar `%d` o `%u` como estás acostumbrado para los enteros.

### 15.1.2 Cadena a valor numérico

Hay un par de familias de funciones para hacer esto en C. Las llamaremos la familia `atoi` (pronunciado *a-to-i*) y la familia `strtol` (*string-to-long*).

Para la conversión básica de una cadena a un número, pruebe las funciones `atoi` de `<stdlib.h>`. Éstas tienen malas características de gestión de errores (incluyendo un comportamiento indefinido si pasas una cadena incorrecta), así que úsalas con cuidado.

| Función | Descripción |
|---|---|
| `atoi` | Cadena a `int` |
| `atof` | Cadena a `float` |
| `atol` | Cadena a `long int` |
| `atoll` | Cadena a `long long int` |

Aunque la especificación no lo menciona, la `a` al principio de la función significa ASCII[2], así que en realidad `atoi()` es «ASCII a entero» (Ascii To Integer, pero decirlo hoy en día es un poco ASCII-céntrico.

He aquí un ejemplo de conversión de una cadena a un `float`:

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

int main(void)
{
    char *pi = "3.14159";
    float f;

    f = atof(pi);

    printf("%f\n", f);
}
```

Pero, como he dicho, obtenemos un comportamiento indefinido de cosas raras como esta:

```
int x = atoi("what");  // «Qué» no es un número que haya oído nunca
```

(Cuando ejecuto eso, obtengo `0` de vuelta, pero realmente no deberías contar con eso de ninguna manera. Podrías obtener algo completamente diferente).

Para obtener mejores características de manejo de errores, echemos un vistazo a todas esas funciones `strtol`, también en `<stdlib.h>`. No sólo eso, ¡también convierten a más tipos y más bases!

| Función | Descripción |
|---|---|
| `strtol` | Cadena a `long int` |
| `strtoll` | Cadena a `long long int` |
| `strtoul` | Cadena a `unsigned long int` |
| `strtoull` | Cadena a `unsigned long long int` |
| `strtof` | Cadena a `float` |

[2] https://en.wikipedia.org/wiki/ASCII

| Función | Descripción |
|---|---|
| `strtod` | Cadena a `double` |
| `strtold` | Cadena a `long double` |

Todas estas funciones siguen un patrón de uso similar y constituyen la primera experiencia de mucha gente con punteros a punteros. Pero no te preocupes, es más fácil de lo que parece.

Hagamos un ejemplo en el que convertimos una cadena a un `unsigned long`, descartando la información de error (es decir, la información sobre caracteres erróneos en la cadena de entrada):

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

int main(void)
{
    char *s = "3490";

    // Convierte la cadena s, a un número en base 10, a un unsigned long int.
    // NULL significa que no nos interesa conocer ninguna información de error.

    unsigned long int x = strtoul(s, NULL, 10);

    printf("%lu\n", x);  // 3490
}
```

Fíjate en un par de cosas. Aunque no nos dignamos a capturar ninguna información sobre caracteres de error en la cadena, `strtoul()` no nos dará un comportamiento indefinido; simplemente devolverá `0`.

Además, especificamos que se trataba de un número decimal (base 10).

¿Significa esto que podemos convertir números de bases diferentes? Por supuesto. ¡Hagámoslo en binario!

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

int main(void)
{
    char *s = "101010";  // ¿Qué significa este número?

    // Convierte la cadena s, un número en base 2, a un unsigned long int.

    unsigned long int x = strtoul(s, NULL, 2);

    printf("%lu\n", x);  // 42
}
```

Vale, eso es muy divertido, pero ¿qué es eso de «NULL»? ¿Para qué sirve?

Nos ayuda a averiguar si se ha producido un error al procesar la cadena. Es un puntero a un puntero a un `char`, que suena espeluznante, pero no lo es una vez que te haces a la idea.

Hagamos un ejemplo en el que introducimos un número deliberadamente malo, y veremos cómo `strtol()` nos permite saber dónde está el primer dígito inválido.

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

int main(void)
{
    char *s = "34x90";  // ¡«x» no es un dígito válido en base 10!
    char *badchar;

    // Convierte la cadena s, un número en base 10, a un unsigned long int.

    unsigned long int x = strtoul(s, &badchar, 10);

    // Intenta convertir tanto como sea posible, así que llega hasta aquí:

    printf("%lu\n", x);  // 34

    // Pero podemos ver el carácter malo porque badchar
    // lo señala.

    printf("Carácter no válido: %c\n", *badchar);  // "x"
}
```

Así que tenemos a `strtoul()` modificando lo que `badchar` señala para mostrarnos dónde han ido mal las cosas[3].

Pero, ¿y si no pasa nada? En ese caso, `badchar` apuntará al terminador `NUL` al final de la cadena. Así que podemos comprobarlo:

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

int main(void)
{
    char *s = "3490";  // ¡«x» no es un dígito válido en base 10!
    char *badchar;

    // Convierte la cadena s, un número en base 10, a un unsigned long int.

    unsigned long int x = strtoul(s, &badchar, 10);

    // Comprueba si todo ha ido bien

    if (*badchar == '\0') {
        printf("Éxito! %lu\n", x);
    } else  {
        printf("Conversión parcial: %lu\n", x);
        printf("Carácter no válido: %c\n", *badchar);
    }
}
```

[3]Tenemos que pasar un puntero a `badchar` a `strtoul()` o no será capaz de modificarlo de ninguna manera que podamos ver, de forma análoga a por qué tienes que pasar un puntero a un `int` a una función si quieres que esa función sea capaz de cambiar el valor de ese `int`

Ahí lo tienes. Las funciones estilo `atoi()` son buenas en un apuro controlado, pero las funciones estilo `strtol()` le dan mucho más control sobre el manejo de errores y la base de la entrada.

## 15.2 Conversiones `char`

¿Qué pasa si tienes un solo carácter con un dígito, como `'5'`? ¿Es lo mismo que el valor «5»?

Probemos a ver.

```
printf("%d %d\n", 5, '5');
```

En mi sistema UTF-8, esto se imprime:

```
5 53
```

Así que… no. ¿Y 53? ¿Qué es eso? Es el punto de código UTF-8 (y ASCII) para el símbolo de carácter `'5'`[4].

Entonces, ¿cómo convertimos el carácter `'5'` (que aparentemente tiene valor 53) en el valor `5`?

Con un ingenioso truco, ¡así es cómo!

El estándar C garantiza que estos caracteres tendrán puntos de código que están en secuencia y en este orden:

```
0  1  2  3  4  5  6  7  8  9
```

Reflexiona un segundo… ¿cómo podemos utilizar eso? Spoilers por delante…

Echemos un vistazo a los caracteres y sus puntos de código en UTF-8:

```
0  1  2  3  4  5  6  7  8  9
48 49 50 51 52 53 54 55 56 57
```

Ahí ves que `'5'` es `53`, tal como nos salía. Y «0» es «48».

Así que podemos restar «0» de cualquier dígito para obtener su valor numérico:

```
char c = '6';

int x = c;  // x tiene el valor 54, el punto de código para '6'

int y = c - '0'; // y tiene valor 6, tal como queremos
```

Y también podemos convertir en el otro sentido, simplemente añadiendo el valor.

```
int x = 6;

char c = x + '0';  // c tiene valor 54

printf("%d\n", c);  // Imprime 54
printf("%c\n", c);  // imprime 6 con %c
```

[4]Cada carácter tiene un valor asociado para cualquier esquema de codificación de caracteres

Puede que pienses que es una forma rara de hacer esta conversión, y para los estándares de hoy en día, ciertamente lo es. Pero en los viejos tiempos, cuando los ordenadores se hacían literalmente de madera, éste era el método para hacer esta conversión. Y no estaba roto, así que C nunca lo arregló.

## 15.3 Conversiones numéricas

### 15.3.1 Booleano

Si convierte un cero en `bool`, el resultado es `0`. En caso contrario es `1`.

### 15.3.2 Conversión de números enteros en números enteros

Si un tipo entero se convierte a sin signo y no cabe en él, el resultado sin signo se envuelve al estilo cuentakilómetros hasta que quepa en el sin signo[5].

Si un tipo entero se convierte a un número con signo y no cabe, ¡el resultado está definido por la implementación! Ocurrirá algo documentado, pero tendrás que buscarlo[6]

### 15.3.3 Conversiones de enteros y coma flotante

Si un tipo de coma flotante se convierte a un tipo entero, la parte fraccionaria se descarta con prejuicio[7].

Pero–y aquí está el truco—si el número es demasiado grande para caber en el entero, se obtiene un comportamiento indefinido. Así que no lo hagas.

Pasando de entero o punto flotante a punto flotante, C hace el mejor esfuerzo para encontrar el número de punto flotante más cercano al entero que pueda.

De nuevo, sin embargo, si el valor original no puede ser representado, es un comportamiento indefinido.

## 15.4 Conversiones implícitas

Se trata de conversiones que el compilador realiza automáticamente cuando se mezclan y combinan tipos.

### 15.4.1 Promociones de enteros

En varios sitios, si un `int` puede usarse para representar un valor de `chart` o `short` (con o sin signo), ese valor es *promovido* a `int`. Si no cabe en un `int`, se promociona a `unsigned int`.

Así es como podemos hacer algo como esto:

```
char x = 10, y = 20;
int i = x + y;
```

En ese caso, `x` e `y` son promovidos a `int` por C antes de que se realice la operación matemática.

Las promociones a enteros tienen lugar durante las conversiones aritméticas habituales, con funciones variádicas[8], operadores unarios `+` y `-`, o al pasar valores a funciones sin prototipos[9].

[5] En la práctica, lo que probablemente está ocurriendo en tu implementación es que los bits de orden alto simplemente se eliminan del resultado, de modo que un número de 16 bits `0x1234` que se convierte a un número de 8 bits termina como `0x0034`, o simplemente `0x34`

[6] De nuevo, en la práctica, lo que probablemente ocurrirá en tu sistema es que el patrón de bits para el original se truncará y luego sólo se usará para representar el número con signo, complemento a dos. Por ejemplo, mi sistema toma un `unsigned char` de `192` y lo convierte a `signed char` `-64`. En complemento a dos, el patrón de bits para ambos números es binario `11000000`.

[7] En realidad no—simplemente se descarta con regularidad

[8] Funciones con un número variable de argumentos.

[9] Esto se hace raramente porque el compilador se quejará y tener un prototipo es lo *Correcto* de hacer. Creo que esto todavía funciona

### 15.4.2 Las conversiones aritméticas habituales

Se trata de conversiones automáticas que C realiza en torno a las operaciones numéricas que se le solicitan. (por cierto, así es como se llaman, en C11 §6.3.1.8.) Tenga en cuenta que en esta sección sólo hablaremos de tipos numéricos; las cadenas vendrán más adelante.

Estas conversiones responden a preguntas sobre lo que ocurre cuando se mezclan tipos, como en este caso:

```
int x = 3 + 1.2; // Mezcla int y double
                 // 4.2 se convierte en int
                 // 4 se almacena en x

float y = 12 * 2; // Mezcla de float e int
                  // 24 se convierte a float
                  // 24.0 se almacena en y
```

¿Se convierten en `int`s? ¿Se convierten en `float`s? ¿Cómo funciona?

He aquí los pasos, parafraseados para facilitar su comprensión.

1. Si una cosa en la expresión es de tipo flotante, convierte las otras cosas a ese tipo flotante.
2. De lo contrario, si ambos tipos son enteros, realice las promociones de enteros en cada uno, luego haga los tipos de operandos tan grandes como sea necesario para mantener el valor más grande común. A veces esto implica cambiar con signo a sin signo.

Si quiere conocer los detalles, consulte C11 §6.3.1.8. Pero probablemente no lo necesites.

En general, recuerde que los tipos int se convierten en tipos float si hay un tipo de coma flotante en cualquier lugar, y el compilador hace un esfuerzo para asegurarse de que los tipos enteros mixtos no se desborden.

Finalmente, si conviertes de un tipo de coma flotante a otro, el compilador intentará hacer una conversión exacta. Si no puede, hará la mejor aproximación posible. Si el número es demasiado grande para caber en el tipo al que se está convirtiendo, *boom*: ¡comportamiento indefinido!

### 15.4.3 `void*`

El tipo `void*` es interesante porque puede convertirse desde o hacia cualquier tipo de puntero.

```
int x = 10;

void *p = &x; // &x es de tipo int*, pero lo almacenamos en un void*

int *q = p; // p es void*, pero lo almacenamos en un int*
```

## 15.5 Conversiones explícitas

Se trata de conversiones de tipo a tipo que debes solicitar; el compilador no lo hará por ti.

Puedes convertir de un tipo a otro asignando un tipo a otro con un `=`.

También puedes convertir explícitamente con un *cast*.

---

por razones históricas, antes de que los prototipos fueran una cosa

### 15.5.1 Casting

Puedes cambiar explícitamente el tipo de una expresión poniendo un nuevo tipo entre paréntesis delante de ella. Algunos desarrolladores de C fruncen el ceño ante esta práctica a menos que sea absolutamente necesario, pero es probable que te encuentres con algún código C que contenga estos paréntesis.

Hagamos un ejemplo en el que queremos convertir un `int` en un `long` para poder almacenarlo en un `long`.

Nota: este ejemplo es artificial y la conversión en este caso es completamente innecesaria porque la expresión `x + 12` se cambiaría automáticamente a `long int` para coincidir con el tipo más amplio de `y`.

```
int x = 10;
long int y = (long int)x + 12;
```

En ese ejemplo, aunque `x` era antes de tipo `int`, la expresión `(long int)x` es de tipo `long int`. Decimos: «Castamos `x` a `long int`».

Más comúnmente, se puede ver una conversión para convertir un `void*` a un tipo de puntero específico para que pueda ser dereferenciado.

Una llamada de retorno de la función incorporada `qsort()` puede mostrar este comportamiento ya que tiene `void*`s pasados a ella:

```
int compar(const void *elem1, const void *elem2)
{
    if (*((const int*)elem2) > *((const int*)elem1)) return 1;
    if (*((const int*)elem2) < *((const int*)elem1)) return -1;
    return 0;
}
```

Pero también podría escribirlo claramente con un encargo:

```
int compar(const void *elem1, const void *elem2)
{
    const int *e1 = elem1;
    const int *e2 = elem2;

    return *e2 - *e1;
}
```

Uno de los lugares en los que verás más comúnmente las conversiones es para evitar una advertencia al imprimir valores de puntero con el raramente usado `%p` que se pone quisquilloso con cualquier cosa que no sea un `void*`:

```
int x = 3490;
int *p = &x;

printf("%p\n", p);
```

genera esta advertencia:

```
warning: format ‘%p’ expects argument of type ‘void *’, but argument
         2 has type ‘int *’
```

Puedes arreglarlo con una escayola:

```
printf("%p\n", (void *)p);
```

Otro lugar es con cambios explícitos de puntero, si no quieres usar un `void*` intermedio, pero estos también son bastante infrecuentes:

```
long x = 3490;
long *p = &x;
unsigned char *c = (unsigned char *)p;
```

Un tercer lugar donde suele ser necesario es con las funciones de conversión de caracteres en `<ctype.h>`[10] donde debe convertir los valores con signo dudoso a `unsigned char` para evitar comportamientos indefinidos.

Una vez más, en la práctica rara vez se *necesita* el reparto. Si te encuentras casteando, puede que haya otra forma de hacer lo mismo, o puede que estés casteando innecesariamente.

O puede que sea necesario. Personalmente, intento evitarlo, pero no tengo miedo de utilizarlo si es necesario.

[10] https://beej.us/guide/bgclr/html/split/ctype.html

# Chapter 16

# Tipos IV: Calificadores y especificadores

Ahora que tenemos algunos tipos más en nuestro haber, resulta que podemos dar a estos tipos algunos atributos adicionales que controlan su comportamiento. Estos son los *calificadores de tipo* y los *especificadores de clase de almacenamiento*.

## 16.1 Calificadores de tipo

Esto le permitirá declarar valores constantes, y también dar al compilador pistas de optimización que puede utilizar.

### 16.1.1 `const`

Es el calificador de tipo más común. Significa que la variable es constante y que cualquier intento de modificarla, provocará el enfado del compilador.

```
const int x = 2;

x = 4;  // COMPILADOR EMITE SONIDOS DE BOCINAZOS
        // no se puede asignar a una constante
```

No se puede modificar un valor `const`.

A menudo se ve `const` en las listas de parámetros de las funciones:

```
void foo(const int x)
{
    printf("%d\n", x + 30);  // OK, no modifica «x»
}
```

#### 16.1.1.1 `const` y punteros

Esto se pone un poco raro, porque hay dos usos que tienen dos significados cuando se trata de punteros.

Por un lado, podemos hacer que no se pueda cambiar la cosa a la que apunta el puntero. Esto se hace poniendo `const` delante del nombre del tipo (antes del asterisco) en la declaración del tipo.

```
int x[] = {10, 20};
const int *p = x;

p++;  // Podemos modificar p, no hay problema

*p = 30; // ¡Error del compilador! No se puede cambiar a qué apunta
```

De forma un tanto confusa, estas dos cosas son equivalentes:

```
const int *p; // No se puede modificar a qué apunta p
int const *p; // No se puede modificar a qué apunta p,
              // igual que en la línea anterior
```

Genial, así que no podemos cambiar la cosa a la que apunta el puntero, pero podemos cambiar el propio puntero. ¿Qué pasa si queremos lo contrario? ¿Queremos poder cambiar aquello a lo que apunta el puntero, pero *no* el puntero en sí?

Basta con mover el `const` después del asterisco en la declaración:

```
int *const p; // No podemos modificar «p» con aritmética de punteros

p++; // ¡Error del compilador!
```

Pero podemos modificar lo que señalan:

```
int x = 10;
int *const p = &x;

*p = 20;   // Pon «x» a 20, no hay problema
```

También puedes hacer que ambas cosas sean `const`:

```
const int *const p;  // ¡No se puede modificar p o *p!
```

Por último, si tienes varios niveles de indirección, debes `const` los niveles apropiados. Sólo porque un puntero sea `const`, no significa que el puntero al que apunta también deba serlo. Puedes establecerlos explícitamente como en los siguientes ejemplos:

```
char **p;
p++;     // OK!
(*p)++;  // OK!

char **const p;
p++;     // Error!
(*p)++;  // OK!

char *const *p;
p++;     // OK!
(*p)++;  // Error!

char *const *const p;
p++;     // Error!
```

```
(*p)++;  // Error!
```

#### 16.1.1.2 `const` Corrección

Una cosa más que tengo que mencionar es que el compilador advertirá en algo como esto:

```
const int x = 20;
int *p = &x;
```

diciendo algo así como:

```
initialization discards 'const' qualifier from pointer type target
// la inicialización descarta el calificador 'const' del objetivo de tipo puntero
```

¿Qué ocurre ahí?

Bueno, tenemos que mirar los tipos a cada lado de la asignación:

```
    const int x = 20;
    int *p = &x;
//    ^       ^
//    |       |
//  int*    const int*
```

El compilador nos está avisando de que el valor de la derecha de la asignación es `const`, pero el de la izquierda no. Y el compilador nos está avisando de que está descartando la «const-idad» de la expresión de la derecha.

Es decir, *podemos* seguir intentando hacer lo siguiente, pero es incorrecto. El compilador avisará, y es un comportamiento indefinido:

```
const int x = 20;
int *p = &x;

*p = 40;  // Comportamiento indefinido--¡quizás modifica «x», quizás no!

printf("%d\n", x);  // 40, si tienes suerte
```

### 16.1.2 `restrict`

TLDR: nunca tienes que usar esto y puedes ignorarlo cada vez que lo veas. Si lo usas correctamente, es probable que obtengas alguna ganancia de rendimiento. Si lo usas incorrectamente, obtendrás un comportamiento indefinido.

`restrict` es una sugerencia al compilador de que una determinada parte de la memoria sólo será accedida por un puntero y nunca por otro. (Es decir, no habrá aliasing del objeto concreto al que apunta el puntero `restrict`). Si un desarrollador declara que un puntero es `restrict` y luego accede al objeto al que apunta de otra manera (por ejemplo, a través de otro puntero), el comportamiento es indefinido.

Básicamente le estás diciendo a C, «Hey—te garantizo que este único puntero es la única forma en la que accedo a esta memoria, y si miento, puedes sacarme un comportamiento indefinido».

Y C usa esa información para realizar ciertas optimizaciones. Por ejemplo, si estás desreferenciando el puntero `restrict` repetidamente en un bucle, C podría decidir almacenar en caché el resultado en un registro

y sólo almacenar el resultado final una vez que el bucle haya terminado. Si cualquier otro puntero hiciera referencia a esa misma memoria y accediera a ella en el bucle, los resultados no serían exactos.

(Nótese que `restrict` no tiene efecto si nunca se escribe en el objeto apuntado. Se trata de optimizaciones en torno a las escrituras en memoria).

Escribamos una función para intercambiar dos variables, y usaremos la palabra clave `restrict` para asegurar a C que nunca pasaremos punteros a la misma cosa. Y luego intentemos pasar punteros a la misma cosa.

```
void swap(int *restrict a, int *restrict b)
{
    int t;

    t = *a;
    *a = *b;
    *b = t;
}

int main(void)
{
    int x = 10, y = 20;

    swap(&x, &y); // ¡Bien! «a» y “b”, arriba, apuntan a cosas diferentes

    swap(&x, &x); // ¡Comportamiento indefinido! «a» y “b” apuntan a lo mismo
}
```

Si eliminámaramos las palabras clave `restrict`, ambas llamadas funcionarían de forma segura. Pero entonces el compilador podría no ser capaz de optimizar.

`restrict` tiene ámbito de bloque, es decir, la restricción sólo dura el ámbito en el que se usa. Si está en una lista de parámetros de una función, está en el ámbito de bloque de esa función.

Si el puntero restringido apunta a un array, sólo se aplica a los objetos individuales del array. Otros punteros pueden leer y escribir desde el array siempre que no lean o escriban ninguno de los mismos elementos que el restringido.

Si está fuera de cualquier función en el ámbito del fichero, la restricción cubre todo el programa.

Es probable que veas esto en funciones de biblioteca como `printf()`:

```
int printf(const char * restrict format, ...);
```

De nuevo, esto sólo indica al compilador que dentro de la función `printf()` sólo habrá un puntero que haga referencia a cualquier parte de la cadena `format`.

Una última nota: si por alguna razón estás usando la notación de array en el parámetro de tu función en lugar de la notación de puntero, puedes usar `restrict` así:

```
void foo(int p[restrict]) // Sin tamaño

void foo(int p[restrict 10]) // O con un tamaño
```

Pero la notación de puntero sería más común.

### 16.1.3 `volatile`

Es poco probable que veas o necesites esto a menos que estés tratando con hardware directamente.

`volatile` indica al compilador que un valor puede cambiar a sus espaldas y que debe buscarlo cada vez.

Un ejemplo podría ser cuando el compilador está buscando en la memoria una dirección que se actualiza continuamente entre bastidores, por ejemplo, algún tipo de temporizador de hardware.

Si el compilador decide optimizar eso y almacenar el valor en un registro durante un tiempo prolongado, el valor en memoria se actualizará y no se reflejará en el registro.

Al declarar algo `volátil`, le estás diciendo al compilador: «Oye, la cosa a la que esto apunta puede cambiar en cualquier momento por razones ajenas a este código de programa».

```
volatile int *p;
```

### 16.1.4 `_Atomic`

Esta es una característica opcional de C de la que hablaremos en el capítulo Atómica.

## 16.2 Especificadores de clase de almacenamiento

Los especificadores de clase de almacenamiento son similares a los cuantificadores de tipo. Proporcionan al compilador más información sobre el tipo de una variable.

### 16.2.1 `auto`

Apenas se ve esta palabra clave, ya que `auto` es el valor por defecto para las variables de ámbito de bloque. Está implícita.

Son los mismos:

```
{
    int a; // auto es el valor por defecto...
    auto int a; // Esto es redundante
}
```

La palabra clave `auto` indica que este objeto tiene *duración de almacenamiento automática*. Es decir, existe en el ámbito en el que se define, y se desasigna automáticamente cuando se sale del ámbito.

Un inconveniente de las variables automáticas es que su valor es indeterminado hasta que se inicializan explícitamente. Decimos que están llenas de datos «aleatorios» o «basura», aunque ninguna de las dos cosas me hace feliz. En cualquier caso, no sabrás lo que contiene a menos que la inicialices.

Inicialice siempre todas las variables automáticas antes de utilizarlas.

### 16.2.2 `static`

Esta palabra clave tiene dos significados, dependiendo de si la variable es de ámbito de fichero o de ámbito de bloque.

Empecemos con el ámbito de bloque.

#### 16.2.2.1 `static` en Alcance del bloque

En este caso, básicamente estamos diciendo: «Sólo quiero que exista una única instancia de esta variable, compartida entre llamadas».

Es decir, su valor persistirá entre llamadas.

Una variable `static` en ámbito de bloque con un inicializador sólo se inicializará una vez al iniciar el programa, no cada vez que se llame a la función.

Hagamos un ejemplo:

```
#include <stdio.h>

void counter(void)
{
    static int count = 1;  // Se inicializa una vez

    printf("Se ha llamado %d vez(es)\n", count);

    count++;
}

int main(void)
{
    counter(); // «Se ha llamado 1 vez(es)»
    counter(); // «Se ha llamado 2 vez(es)»
    counter(); // «Se ha llamado 3 vez(es)»
    counter(); // «Se ha llamado 4 vez(es)»
}
```

¿Ves cómo el valor de `count` persiste entre llamadas?

Una cosa a tener en cuenta es que las variables de ámbito de bloque `static` se inicializan a `0` por defecto.

```
static int foo; // El valor inicial por defecto es `0`...
static int foo = 0; // Así que la asignación `0` es redundante
```

Por último, ten en cuenta que si escribes programas multihilo, tienes que asegurarte de no dejar que varios hilos pisoteen la misma variable.

#### 16.2.2.2 `static` en Alcance del archivo

Cuando se sale del ámbito del fichero, fuera de cualquier bloque, el significado cambia bastante.

Las variables en el ámbito del fichero ya persisten entre llamadas a funciones, así que ese comportamiento ya existe.

En cambio, lo que `static` significa en este contexto es que esta variable no es visible fuera de este archivo fuente en particular. Algo así como «global», pero sólo en este archivo.

Más sobre esto en la sección sobre construir con múltiples ficheros fuente.

### 16.2.3 `extern`

El especificador de clase de almacenamiento `extern` nos da una forma de referirnos a objetos en otros ficheros fuente.

Digamos, por ejemplo, que el fichero `bar.c` tiene lo siguiente en su totalidad:

```
// bar.c

int a = 37;
```

Sólo eso. Declarando un nuevo `int a` en el ámbito del fichero.

¿Pero qué pasaría si tuviéramos otro fichero fuente, `foo.c`, y quisiéramos referirnos a `a` que está en `bar.c`?

Es fácil con la palabra clave `extern`:

```
// foo.c

extern int a;

int main(void)
{
    printf("%d\n", a);  // 37, ¡desde bar.c!

    a = 99;

    printf("%d\n", a);  // La misma «a» de bar.c, pero ahora es 99
}
```

También podríamos haber hecho el `extern int a` en el ámbito del bloque, y aún así se habría referido al `a` en `bar.c`:

```
// foo.c

int main(void)
{
    extern int a;

    printf("%d\n", a);  // 37, ¡desde bar.c!

    a = 99;

    printf("%d\n", a);  // La misma «a» de bar.c, pero ahora es 99
}
```

Ahora bien, si `a` en `bar.c` se hubiera marcado como `static`, esto no habría funcionado. Las variables `static` en el ámbito de un fichero no son visibles fuera de ese fichero.

Una nota final sobre `extern` en funciones. Para las funciones, `extern` es el valor por defecto, por lo que es redundante. Puedes declarar una función `static` si sólo quieres que sea visible en un único fichero fuente.

### 16.2.4 `register`

Se trata de una palabra clave que indica al compilador que esta variable se utiliza con frecuencia, y que debe ser lo más rápida posible. El compilador no está obligado a aceptarla.

Ahora bien, los modernos optimizadores del compilador de C son bastante eficaces a la hora de averiguar esto por sí mismos, por lo que es raro verlo hoy en día.

Pero si es necesario:

```
#include <stdio.h>

int main(void)
{
    register int a;   // Haz que «a» sea tan rápido de usar como sea posible.

    for (a = 0; a < 10; a++)
        printf("%d\n", a);
}
```

Sin embargo, tiene un precio. No se puede tomar la dirección de un registro:

```
register int a;
int *p = &a;    // ¡ERROR DEL COMPILADOR!
                // No se puede tomar la dirección de un registro!
```

Lo mismo se aplica a cualquier parte de una matriz:

```
register int a[] = {11, 22, 33, 44, 55};
int *p = a;  // ¡ERROR DEL COMPILADOR! No se puede tomar la dirección de a[0]
```

O desreferenciar parte de un array:

```
register int a[] = {11, 22, 33, 44, 55};

int a = *(a + 2);  // ¡ERROR DEL COMPILADOR! Dirección de a[0] tomada
```

Curiosamente, para el equivalente con notación array, gcc sólo avisa:

```
register int a[] = {11, 22, 33, 44, 55};

int a = a[2];  // ¡ADVERTENCIA DEL COMPILADOR!
```

con:

```
warning: ISO C forbids subscripting ‘register’ array
//Advertencia: ISO C prohíbe los subíndices en las matrices 'register'
```

El hecho de que no se pueda tomar la dirección de una variable de registro libera al compilador para realizar optimizaciones en torno a esa suposición, si es que aún no las ha deducido. Además, añadir `register` a una variable `const` evita que accidentalmente se pase su puntero a otra función que ignore su constancia [1].

Un poco de historia: en el interior de la CPU hay pequeñas «variables» dedicadas llamadas *registers / registros*[2]. Su acceso es superrápido comparado con la RAM, por lo que usarlas aumenta la velocidad. Pero no están en la RAM, así que no tienen una dirección de memoria asociada (por eso no puedes tomar la dirección-de u obtener un puntero a ellas).

Pero, como ya he dicho, los compiladores modernos son realmente buenos a la hora de producir código óptimo, utilizando registros siempre que sea posible independientemente de si se ha especificado o no la palabra clave `register`. No sólo eso, sino que la especificación les permite tratarlo como si hubieras escrito «auto», si quieren. Así que no hay garantías.

[1] https://gustedt.wordpress.com/2010/08/17/a-common-misconception-the-register-keyword/
[2] https://en.wikipedia.org/wiki/Processor_register

### 16.2.5 `_Thread_local`

Cuando utilizas varios subprocesos y tienes algunas variables en el ámbito global o `static` del bloque, esta es una forma de asegurarte de que cada subproceso obtiene su propia copia de la variable. Esto te ayudará a evitar condiciones de carrera y que los hilos se pisen unos a otros.

Si estás en ámbito de bloque, tienes que usar esto junto con `extern` o `static`.

Además, si incluyes `<threads.h>`, puedes usar el más agradable `thread_local` como alias del más feo `_Thread_local`.

Puedes encontrar más información en la sección Threads.

# Chapter 17

# Proyectos multiarchivos

Hasta ahora hemos visto programas de juguete que, en su mayor parte, caben en un único archivo. Pero los programas C complejos se componen de muchos archivos que se compilan y enlazan en un único ejecutable.

En este capítulo veremos algunos de los patrones y prácticas más comunes para crear proyectos más grandes.

## 17.1 Incluye prototipos y funciones

Una situación realmente común es que algunas de tus funciones estén definidas en un fichero, y quieras llamarlas desde otro.

Esto en realidad funciona con una advertencia… probemos primero y luego veamos la forma correcta de arreglar la advertencia.

Para estos ejemplos, pondremos el nombre del archivo como el primer comentario en el código fuente.

Para compilarlos, necesitarás especificar todas las fuentes en la línea de comandos:

```
# archivo de salida      archivos de origen
#     v                       |
#   |----| |---------|        |
gcc -o foo foo.c bar.c  <-----|
```

En esos ejemplos, `foo.c` y `bar.c` se construyen en el ejecutable llamado `foo`.

Así que echemos un vistazo al archivo fuente `bar.c`:

```
// Archivo bar.c

int add(int x, int y)
{
    return x + y;
}
```

Y el archivo `foo.c` con main en él:

```
// Archivo foo.c

#include <stdio.h>
```

```
int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Mira cómo desde `main()` llamamos a `add()`–¡pero `add()` está en un fichero fuente completamente diferente! Está en `bar.c`, ¡mientras que la llamada a él está en `foo.c`!

Si construimos esto con:

```
gcc -o foo foo.c bar.c
```

obtenemos este error:

```
error: implicit declaration of function 'add' is invalid in C99
// error: la declaración implícita de la función 'add' no es válida en C99
```

(O puede recibir una advertencia. Que no debes ignorar. Nunca ignores las advertencias en C; atiéndelas todas).

Si recuerdas la sección sobre prototipos, las declaraciones implícitas están prohibidas en el C moderno y no hay ninguna razón legítima para introducirlas en código nuevo. Deberíamos arreglarlo.

Lo que significa `declaración implícita` es que estamos usando una función, en este caso `add()`, sin que C sepa nada de ella de antemano. C quiere saber qué devuelve, qué tipos toma como argumentos, y cosas por el estilo.

Ya vimos cómo arreglar esto antes con un *prototipo de función*. De hecho, si añadimos uno de esos a `foo.c` antes de hacer la llamada, todo funciona bien:

```
// File foo.c

#include <stdio.h>

int add(int, int);  // Añadir el prototipo

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Se acabaron los errores.

Pero eso es un coñazo—tener que teclear el prototipo cada vez que quieres usar una función. Quiero decir, usamos `printf()` justo ahí y no necesitamos teclear un prototipo; ¿qué pasa?

Si recuerdas lo que pasó con «hola.c» al principio del libro, ¡en realidad incluimos el prototipo de `printf()` *! ¡Está en el fichero* `stdio.h` *!* ¡Y lo incluimos con `#include`!

¿Podemos hacer lo mismo con nuestra función `add()`? ¿Hacer un prototipo para ella y ponerlo en un fichero de cabecera?

Por supuesto.

Los ficheros de cabecera en C tienen una extensión `.h` por defecto. Y a menudo, aunque no siempre, tienen el mismo nombre que su correspondiente archivo `.c`. Así que vamos a crear un fichero `bar.h` para nuestro fichero `bar.c`, y vamos a meter el prototipo en él:

```
// Archivo bar.h

int add(int, int);
```

Y ahora vamos a modificar `foo.c` para incluir ese fichero. Suponiendo que está en el mismo directorio, lo incluimos entre comillas dobles (en lugar de corchetes angulares):

```
// Archivo foo.c

#include <stdio.h>

#include "bar.h"  // Incluir desde el directorio actual

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Fíjate en que ya no tenemos el prototipo en `foo.c`, lo hemos incluido en `bar.h`. Ahora *cualquier* fichero que quiera la funcionalidad `add()` puede simplemente `#include «bar.h»` para obtenerla, y no necesitas preocuparte de escribir el prototipo de la función.

Como habrás adivinado, `#include` incluye literalmente el fichero nombrado *ahí mismo* en tu código fuente, como si lo hubieras tecleado.

Y al construir y ejecutar:

```
./foo
5
```

Efectivamente, ¡obtenemos el resultado de $2 + 3$! ¡Bien!

Pero no abras todavía tu bebida preferida. Ya casi hemos llegado. Sólo tenemos que añadir una pieza más de la repetición.

## 17.2 Tratamiento de la repetición `#include`

No es infrecuente que un fichero de cabecera incluya a su vez otras cabeceras necesarias para la funcionalidad de sus correspondientes ficheros C. ¿Por qué no?

Y puede ser que tengas una cabecera `#include` varias veces desde diferentes sitios. Puede que eso no sea un problema, pero puede que cause errores de compilación. ¡Y no podemos controlar en cuántos sitios se hace `#include`!

Peor aún, ¡podríamos llegar a una situación loca en la que la cabecera `a.h` incluya la cabecera `b.h`, y `b.h` incluya `a.h`! ¡Es un ciclo infinito `#include`!

Intentar construir algo así da error:

```
error: #include nested depth 200 exceeds maximum of 200
//error : #include profundidad anidada 200 supera el máximo de 200
```

Lo que tenemos que hacer es que si un archivo se incluye una vez, los subsiguientes `#include`s para ese archivo sean ignorados.

**Lo que vamos a hacer es tan común que deberías hacerlo automáticamente cada vez que crees un archivo de cabecera.**

Y la forma común de hacerlo es con una variable de preprocesador que establecemos la primera vez que `#incluimos` el archivo. Y entonces para los subsiguientes `#includes`, primero comprobamos que la variable no está definida.

Para ese nombre de variable, es supercomún tomar el nombre del fichero de cabecera, como `bar.h`, ponerlo en mayúsculas, y sustituir el punto por un guión bajo: `BAR_H`.

Por lo tanto, compruebe en la parte superior del archivo si ya se ha incluido y, en caso afirmativo, coméntelo todo.

(No pongas un guión bajo inicial (porque un guión bajo inicial seguido de una letra mayúscula está reservado) o un doble guión bajo inicial (porque también está reservado.))

```
#ifndef BAR_H // Si BAR_H no está definido...
#define BAR_H // Definirlo (sin valor particular)

// Archivo bar.h

int add(int, int);

#endif          // Fin del #ifndef BAR_H
```

Esto hará que el fichero de cabecera se incluya sólo una vez, sin importar en cuántos sitios se intente hacer `#include`.

## 17.3 `static` y `extern`

Cuando se trata de proyectos multifichero, puedes asegurarte de que las variables y funciones de un fichero no son visibles desde otros ficheros fuente con la palabra clave `static`.

Y puedes hacer referencia a objetos de otros ficheros con `extern`.

Para más información, echa un vistazo a las secciones en el libro sobre el `static` y `extern` especificadores de clase de almacenamiento.

## 17.4 Compilación con archivos de objetos

Esto no es parte de la especificación, pero es 99,999% común en el mundo C.

Puedes compilar archivos C en una representación intermedia llamada *archivos objeto*. Estos son código máquina compilado que aún no ha sido puesto en un ejecutable.

Los archivos objeto en Windows tienen una extensión `.OBJ`; en Unix, son `.o`.

En gcc, podemos construir algo como esto, con la bandera `-c` (¡sólo compilar!):

```
gcc -c foo.c # produce foo.o
gcc -c bar.c # produce bar.o
```

Y luego podemos *enlazarlos* juntos en un único ejecutable:

```
gcc -o foo foo.o bar.o
```

*Voila*, hemos producido un ejecutable `foo` a partir de los dos ficheros objeto.

Pero usted está pensando, ¿por qué molestarse? ¿No podemos simplemente:

```
gcc -o foo foo.c bar.c
```

¿y matar dos boids[1] de un tiro?

Para programas pequeños, está bien. Yo lo hago todo el tiempo.

Pero para programas más grandes, podemos aprovechar el hecho de que compilar desde el código fuente a los ficheros objeto es relativamente lento, y enlazar un montón de ficheros objeto es relativamente rápido.

Esto realmente se muestra con la utilidad `make` que sólo reconstruye fuentes que son más recientes que sus salidas.

Digamos que tienes mil archivos C. Podrías compilarlos todos a ficheros objeto para empezar (lentamente) y luego combinar todos esos ficheros objeto en un ejecutable (rápido).

Ahora digamos que modificas sólo uno de esos archivos fuente en C -aquí está la magia: ¡sólo tienes que reconstruir ese archivo objeto para ese archivo fuente! Y luego reconstruir el ejecutable (rápido). Todos los demás archivos C no tienen que ser tocados.

En otras palabras, al reconstruir sólo los archivos objeto que necesitamos, reducimos radicalmente los tiempos de compilación. (A menos, claro, que estés haciendo una compilación «limpia», en cuyo caso hay que crear todos los ficheros objeto).

[1] https://en.wikipedia.org/wiki/Boids

# Chapter 18

# Entorno exterior

Cuando ejecutas un programa, en realidad eres tú quien le habla al shell, diciéndole: «Oye, por favor, ejecuta esto». Y el intérprete de comandos dice: «Claro», y luego le dice al sistema operativo: «Oye, ¿podrías crear un nuevo proceso y ejecutar esta cosa?». Y si todo va bien, el sistema operativo cumple y tu programa se ejecuta.

Pero hay todo un mundo fuera de tu programa en el shell con el que se puede interactuar desde C. Veremos algunos de ellos en este capítulo.

## 18.1 Argumentos de la línea de comandos

Muchas utilidades de línea de comandos aceptan *argumentos de línea de comandos*. Por ejemplo, si queremos ver todos los archivos que terminan en `.txt`, podemos escribir algo como esto en un sistema tipo Unix:

```
ls *.txt
```

(o `dir` en lugar de `ls` en un sistema Windows).

En este caso, el comando es `ls`, pero sus argumentos son todos los ficheros que terminan en `.txt`[1].

Entonces, ¿cómo podemos ver lo que se pasa al programa desde la línea de comandos?

Digamos que tenemos un programa llamado `add` que suma todos los números pasados en la línea de comandos e imprime el resultado:

```
./add 10 30 5
45
```

¡Seguro que eso pagará las facturas!

Pero en serio, esta es una gran herramienta para ver cómo obtener esos argumentos de la línea de comandos y desglosarlos.

Primero, veamos cómo obtenerlos. ¡Para ello, vamos a necesitar un nuevo `main()`!

Aquí hay un programa que imprime todos los argumentos de la línea de comandos. Por ejemplo, si nombramos al ejecutable `foo`, podemos ejecutarlo así:

[1] Históricamente, los programas de MS-DOS y Windows hacían esto de forma diferente a Unix. En Unix, el intérprete de comandos *expandía* el comodín en todos los archivos coincidentes antes de que el programa lo viera, mientras que las variantes de Microsoft pasaban la expresión del comodín al programa para que éste se ocupara de ella. En cualquier caso, hay argumentos que se pasan al programa

```
./foo i like turtles
```

y veremos esta salida:

```
arg 0: ./foo
arg 1: i
arg 2: like
arg 3: turtles
```

Es un poco raro, porque el argumento zero es el nombre del ejecutable, en sí mismo. Pero es algo a lo que hay que acostumbrarse. Los argumentos en sí siguen directamente.

Fuente:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("arg %d: %s\n", i, argv[i]);
    }
}
```

¡Vaya! ¿Qué pasa con la firma de la función `main()`? ¿Qué son `argc` y `argv` [2]. (pronunciados *arg-cee* y *arg-vee*)?

Empecemos por lo más fácil: `argc`. Es el *número de argumentos*, incluido el propio nombre del programa. Si piensas en todos los argumentos como un array de cadenas, que es exactamente lo que son, entonces puedes pensar en `argc` como la longitud de ese array, que es exactamente lo que es.

Y así lo que estamos haciendo en ese bucle es ir a través de todos los `argv`s e imprimirlos uno a la vez, por lo que para una entrada dada:

```
./foo i like turtles
```

obtenemos la salida correspondiente:

```
arg 0: ./foo
arg 1: i
arg 2: like
arg 3: turtles
```

Con esto en mente, deberíamos estar listos para empezar con nuestro programa de sumadores.

Nuestro plan:

- Mirar todos los argumentos de la línea de comandos (pasado `argv[0]`, el nombre del programa)
- Convertirlos en enteros
- Sumarlos a un total
- Imprimir el resultado

Manos a la obra.

[2] Como son nombres de parámetros normales, no tienes que llamarlos `argc` y `argv`. Pero es tan idiomático usar esos nombres, que si te pones creativo, otros programadores de C te mirarán con ojos sospechosos

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

int main(int argc, char **argv)
{
    int total = 0;

    for (int i = 1; i < argc; i++) { // Empieza en 1, el primer argumento
       int value = atoi(argv[i]); // Usa strtol() para un mejor manejo de errores

        total += value;
    }

    printf("%d\n", total);
}
```

Recorridos de muestra:

```
$ ./add
0
$ ./add 1
1
$ ./add 1 2
3
$ ./add 1 2 3
6
$ ./add 1 2 3 4
10
```

Por supuesto, podría vomitar si le pasas un no entero, pero endurecer contra eso se deja como un ejercicio para el lector.

### 18.1.1 El último `argv` es `NULL`.

Una curiosidad sobre `argv` es que después de la última cadena hay un puntero a `NULL`.

Eso es:

```
argv[argc] == NULL
```

¡siempre es cierto!

Esto puede parecer inútil, pero resulta ser útil en un par de lugares; vamos a echar un vistazo a uno de ellos ahora mismo.

### 18.1.2 El suplente: `char **argv`

Recuerda que cuando llamas a una función, C no diferencia entre notación de array y notación de puntero en la firma de la función.

Es decir, son lo mismo:

```
void foo(char a[])
void foo(char *a)
```

Ahora, ha sido conveniente pensar en `argv` como una matriz de cadenas, es decir, una matriz de `char*`s, así que esto tenía sentido:

```
int main(int argc, char *argv[])
```

pero debido a la equivalencia, también se podría escribir:

```
int main(int argc, char **argv)
```

¡Sí, es un puntero a un puntero, de acuerdo! Si te resulta más fácil, piensa que es un puntero a una cadena. Pero en realidad, es un puntero a un valor que apunta a un `char`.

Recuerda también que son equivalentes:

```
argv[i]
*(argv + i)
```

Lo que significa que puedes hacer aritmética de punteros en `argv`.

Así que una forma alternativa de consumir los argumentos de la línea de comandos podría ser simplemente caminar a lo largo de la matriz `argv` subiendo un puntero hasta que lleguemos a ese `NULL` al final.

Vamos a modificar nuestro sumador para hacer eso:

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

int main(int argc, char **argv)
{
    int total = 0;

    // Bonito truco para que el compilador deje de advertir sobre la
    // variable no utilizada argc:
    (void)argc;

    for (char **p = argv + 1; *p != NULL; p++) {
        int value = atoi(*p);  // Use strtol() para un mejor manejo de errores

        total += value;
    }

    printf("%d\n", total);
}
```

Personalmente, utilizo la notación array para acceder a `argv`, pero he visto este estilo flotando por ahí, también.

### 18.1.3 Datos curiosos

Algunas cosas más sobre `argc` y `argv`.

- Algunos entornos pueden no establecer `argv[0]` al nombre del programa. Si no está disponible, `argv[0]` será una cadena vacía. Nunca he visto que esto ocurra.

- La especificación es bastante liberal con lo que una implementación puede hacer con `argv` y de dónde vienen esos valores. Pero en todos los sistemas (en los que he estado) funciona de la misma manera, como hemos discutido en esta sección.
- Puedes modificar `argc`, `argv`, o cualquiera de las cadenas a las que apunta `argv`. (¡Sólo no hagas esas cadenas más largas de lo que ya son!)
- En algunos sistemas tipo Unix, modificar la cadena `argv[0]` hace que la salida de `ps` cambie[3].

Normalmente, si tienes un programa llamado `foo` que has ejecutado con `./foo`,podrías ver esto en la salida de `ps`:

```
 4078 tty1     S      0:00 ./foo
```

Pero si modificas `argv[0]` así, teniendo cuidado de que la nueva cadena «`Hi!`» tenga la misma longitud que la anterior «`. foo`»:

```
strcpy(argv[0], "Hi!  ");
```

y luego ejecutamos `ps` mientras el programa `./foo` aún se está ejecutando, veremos esto en su lugar:

```
 4079 tty1     S      0:00 Hi!
```

Este comportamiento no está en la especificación y depende en gran medida del sistema.

## 18.2 Estado de salida

¿Te has dado cuenta de que la firma de la función `main()` devuelve el tipo `int`? ¿A qué se debe? Tiene que ver con una cosa llamada *exit status*, que es un entero que puede ser devuelto al programa que lanzó el suyo para hacerle saber cómo fueron las cosas.

Ahora, hay un número de maneras en que un programa puede salir en C, incluyendo `return` desde `main()`, o llamando a una de las variantes de `exit()`.

Todos estos métodos aceptan un `int` como argumento.

Nota al margen: ¿has visto que básicamente en todos mis ejemplos, aunque se supone que `main()` debe devolver un `int`, en realidad no `devuelvo` nada? En cualquier otra función, esto sería ilegal, pero hay un caso especial en C: si la ejecución llega al final de `main()` sin encontrar un `return`, automáticamente hace un `return 0`.

Pero, ¿qué significa el «0»? ¿Qué otros números podemos poner ahí? ¿Y cómo se utilizan?

La especificación es a la vez clara e imprecisa al respecto, como suele ser habitual. Clara porque detalla lo que se puede hacer, pero vaga porque tampoco lo limita especialmente.

No queda más remedio que *seguir adelante* e ingeniárselas.

Pongámonos Inicio[4] por un segundo: resulta que cuando ejecutas tu programa, *lo estás ejecutando desde otro programa*.

Normalmente este otro programa es algún tipo de shell[5] que no hace mucho por sí mismo, excepto lanzar otros programas.

Pero se trata de un proceso de varias fases, especialmente visible en los shells de línea de comandos:

[3] `ps`, Process Status, es un comando de Unix para ver qué procesos se están ejecutando en ese momento
[4] https://en.wikipedia.org/wiki/Inception
[5] https://en.wikipedia.org/wiki/Shell_(computing)

1. El intérprete de comandos inicia el programa
2. El shell normalmente entra en reposo (para los shells de línea de comandos)
3. Su programa se ejecuta
4. Tu programa termina
5. El shell se despierta y espera otro comando

Ahora, hay una pequeña pieza de comunicación que tiene lugar entre los pasos 4 y 5: el programa puede devolver un *valor de estado* que el shell puede interrogar. Típicamente, este valor se usa para indicar el éxito o fracaso de su programa, y, si es un fracaso, qué tipo de fracaso.

Este valor es el que hemos estado `devolviendo` desde `main()`. Ese es el estado.

Ahora, la especificación C permite dos valores de estado diferentes, que tienen nombres de macros definidos en `<stdlib.h>`:

| Estado | Descripción |
|---|---|
| `EXIT_SUCCESS` o `0` | El programa ha finalizado correctamente. |
| `EXIT_FAILURE` | El programa ha finalizado con un error. |

Vamos a escribir un programa corto que multiplique dos números desde la línea de comandos. Requeriremos que especifiques exactamente dos valores. Si no lo hace, vamos a imprimir un mensaje de error, y salir con un estado de error.

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

int main(int argc, char **argv)
{
    if (argc != 3) {
        printf("usage: mult x y\n");
        return EXIT_FAILURE;   // Indicar al shell que no funcionó
    }

    printf("%d\n", atoi(argv[1]) * atoi(argv[2]));

    return 0;  // igual que EXIT_SUCCESS, todo iba bien.
}
```

Ahora, si intentamos ejecutarlo, obtendremos el efecto esperado hasta que especifiquemos exactamente el número correcto de argumentos de la línea de comandos:

```
$ ./mult
usage: mult x y

$ ./mult 3 4 5
usage: mult x y

$ ./mult 3 4
12
```

Pero eso no muestra realmente el estado de salida que hemos devuelto, ¿verdad? Sin embargo, podemos hacer que el shell lo imprima. Asumiendo que estás ejecutando Bash u otro shell POSIX, puedes usar `echo $?` para verlo[6].

[6] En Windows `cmd.exe`, escribe `echo %errorlevel%`. En PowerShell, escribe `$LastExitCode`

Intentémoslo:

```
$ ./mult
usage: mult x y
$ echo $?
1

$ ./mult 3 4 5
usage: mult x y
$ echo $?
1

$ ./mult 3 4
12
$ echo $?
0
```

¡Interesante! Vemos que en mi sistema `EXIT_FAILURE` es `1`. La especificación no lo especifica, así que podría ser cualquier número. Pero pruébalo; probablemente sea `1` en tu sistema también.

### 18.2.1 Otros valores de estado de salida

El estado `0` definitivamente significa éxito, pero ¿qué pasa con el resto de enteros, incluso los negativos?

Aquí nos salimos de la especificación C y nos adentramos en la tierra de Unix. En general, mientras que `0` significa éxito, un número positivo distinto de cero significa fracaso. Así que sólo puedes tener un tipo de éxito, y múltiples tipos de fallo. Bash dice que el código de salida debe estar entre 0 y 255, aunque hay una serie de códigos reservados.

En resumen, si quieres indicar diferentes estados de salida de error en un entorno Unix, puedes empezar con `1` e ir subiendo.

En Linux, si intentas cualquier código fuera del rango 0-255, el código será bitwise AND con `0xff`, sujetándolo efectivamente a ese rango.

Puedes programar el shell para que más tarde use estos códigos de estado para tomar decisiones sobre qué hacer a continuación.

## 18.3 Variables de entorno

Antes de entrar en materia, debo advertirte que C no especifica qué es una variable de entorno. Así que voy a describir el sistema de variables de entorno que funciona en todas las plataformas importantes que conozco.

Básicamente, el entorno es el programa que va a ejecutar tu programa, por ejemplo, el shell bash. Y puede tener definidas algunas variables bash. En caso de que no lo sepas, el shell puede crear sus propias variables. Cada shell es diferente, pero en bash puedes simplemente escribir `set` y te las mostrará todas.

Aquí hay un extracto de las 61 variables que están definidas en mi shell bash:

```
HISTFILE=/home/beej/.bash_history
HISTFILESIZE=500
HISTSIZE=500
HOME=/home/beej
HOSTNAME=FBILAPTOP
HOSTTYPE=x86_64
```

```
IFS=$' \t\n'
```

Fíjese en que están en forma de pares clave/valor. Por ejemplo, una clave es `HOSTTYPE` y su valor es `x86_64`. Desde una perspectiva C, todos los valores son cadenas, incluso si son números[7].

Así que, ¡como quieras! Resumiendo, es posible obtener estos valores desde dentro de tu programa C.

Escribamos un programa que utilice la función estándar `getenv()` para buscar un valor que hayas establecido en el shell.

La función `getenv()` devolverá un puntero a la cadena de valores, o bien `NULL` si la variable de entorno no existe.

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

int main(void)
{
    char *val = getenv("FROTZ");  // Intenta obtener el valor

    // Comprueba que existe
    if (val == NULL) {
        printf("No se encuentra la variable de entorno FROTZ\n");
        return EXIT_FAILURE;
    }

    printf("Value: %s\n", val);
}
```

Si ejecuto esto directamente, obtengo lo siguiente:

```
$ ./foo
No se encuentra la variable de entorno FROTZ
```

que tiene sentido, ya que no lo he establecido todavía.

En bash, puedo ponerlo a algo con[8]:

```
$ export FROTZ="C is awesome!"
```

Entonces si lo ejecuto, obtengo:

```
$ ./foo
Value: C is awesome!
```

De este modo, puede establecer datos en variables de entorno, y puede obtenerlos en su código C y modificar su comportamiento en consecuencia.

### 18.3.1 Configuración de variables de entorno

Esto no es estándar, pero muchos sistemas proporcionan formas de establecer variables de entorno.

[7] Si necesitas un valor numérico, convierte la cadena con algo como `atoi()` o `strtol()`

[8] En Windows CMD.EXE, usa `set FROTZ=value`. En PowerShell, utilice `$Env:FROTZ=value`

Si está en un sistema tipo Unix, busque la documentación de `putenv()`, `setenv()`, y `unsetenv()`. En Windows, consulte `_putenv()`.

### 18.3.2 Variables de entorno alternativas a Unix

Si estás en un sistema tipo Unix, lo más probable es que tengas otro par de formas de acceder a las variables de entorno. Tenga en cuenta que aunque la especificación señala esto como una extensión común, no es realmente parte del estándar de C. Es, sin embargo, parte del estándar POSIX.

Una de ellas es una variable llamada `environ` que debe declararse así:

```
extern char **environ;
```

Es un array de cadenas terminado con un puntero `NULL`.

Deberías declararlo tú mismo antes de usarlo, o podrías encontrarlo en el fichero de cabecera no estándar `<unistd.h>`.

Cada cadena tiene la forma `«clave=valor»`, por lo que tendrás que dividirla y analizarla tú mismo si quieres obtener las claves y los valores.

Aquí hay un ejemplo de un bucle e impresión de las variables de entorno de un par de maneras diferentes:

```
#include <stdio.h>

extern char **environ;  // DEBE ser externo Y llamarse «environ».

int main(void)
{
    for (char **p = environ; *p != NULL; p++) {
        printf("%s\n", *p);
    }

    // O podrías hacer esto:
    for (int i = 0; environ[i] != NULL; i++) {
        printf("%s\n", environ[i]);
    }
}
```

Para un montón de salida que se parece a esto:

```
SHELL=/bin/bash
COLORTERM=truecolor
TERM_PROGRAM_VERSION=1.53.2
LOGNAME=beej
HOME=/home/beej
... etc ...
```

Utilice `getenv()` si es posible porque es más portable. Pero si tienes que iterar sobre variables de entorno, usar `environ` puede ser la mejor opción.

Otra forma no estándar de obtener las variables de entorno es como parámetro de `main()`. Funciona de forma muy parecida, pero se evita tener que añadir la variable `environ` `extern`. Ni siquiera la especificación POSIX soporta esto.[9] que yo sepa, pero es común en la tierra de Unix.

[9] https://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html

```
#include <stdio.h>

int main(int argc, char **argv, char **env)  // <-- env!
{
    (void)argc; (void)argv;  // Suprimir las advertencias no utilizadas

    for (char **p = env; *p != NULL; p++) {
        printf("%s\n", *p);
    }

    // O podrías hacer esto:
    for (int i = 0; env[i] != NULL; i++) {
        printf("%s\n", env[i]);
    }
}
```

Es como usar `environ` pero *incluso menos portable*. Es bueno tener objetivos.

# Chapter 19

# El preprocesador C

Antes de que el programa se compile, pasa por una fase llamada *preprocesamiento*. Es casi como si hubiera un lenguaje *sobre* el lenguaje C que se ejecuta primero. Y genera el código C, que luego se compila.

¡Ya hemos visto esto hasta cierto punto con `#include`! Ese es el preprocesador C. Cuando ve esa directiva, incluye el fichero nombrado allí mismo, como si lo hubieras escrito allí. Y *entonces* el compilador lo construye todo.

Pero resulta que es mucho más potente que simplemente poder incluir cosas. Puedes definir *macros* que son sustituidas… ¡e incluso macros que toman argumentos!

## 19.1 `#include`

Empecemos por la que ya hemos visto muchas veces. Se trata, por supuesto, de una forma de incluir otras fuentes en tu fuente. Muy comúnmente usado con archivos de cabecera.

Mientras que la especificación permite todo tipo de comportamientos con `#include`, vamos a tomar un enfoque más pragmático y hablar de la forma en que funciona en todos los sistemas que he visto.

Podemos dividir los ficheros de cabecera en dos categorías: sistema y local. Las cosas que están integradas, como `stdio.h`, `stdlib.h`, `math.h`, etc., se pueden incluir con corchetes angulares:

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

Los corchetes angulares le dicen a C: «Oye, no busques este archivo de cabecera en el directorio actual, sino en el directorio de inclusión de todo el sistema».

Lo que, por supuesto, implica que debe haber una forma de incluir archivos locales del directorio actual. Y la hay: con comillas dobles:

```
#include "myheader.h"
```

O muy probablemente puede buscar en directorios relativos usando barras inclinadas y puntos, así:

```
#include "mydir/myheader.h"
#include "../someheader.py"
```

¡No use una barra invertida (`\`) para sus separadores de ruta en su `#include`! Es un comportamiento indefinido. Utilice sólo la barra oblicua (`/`), incluso en Windows.

En resumen, usa corchetes angulares (`<` y `>`) para los includes del sistema, y usa comillas dobles (`"`) para tus includes personales.

## 19.2 Macros sencillas

Un *macro* es un identificador que se *expande* a otro trozo de código antes de que el compilador lo vea. Piense en ello como un marcador de posición - cuando el preprocesador ve uno de esos identificadores, lo sustituye por otro valor que ha definido.

Lo hacemos con `#define` (a menudo se lee «pound define»). He aquí un ejemplo:

```
#include <stdio.h>

#define HELLO "Hello, world"
#define PI 3.14159

int main(void)
{
    printf("%s, %f\n", HELLO, PI);
}
```

En las líneas 3 y 4 definimos un par de macros. Dondequiera que aparezcan en el código (línea 8), serán sustituidas por los valores definidos.

Desde la perspectiva del compilador de C, es exactamente como si hubiéramos escrito esto, en su lugar:

```
#include <stdio.h>

int main(void)
{
    printf("%s, %f\n", "Hello, world", 3.14159);
}
```

¿Ve cómo `HELLO` ha sido sustituido por `«Hola, mundo»` y `PI` por `3,14159`? Desde la perspectiva del compilador, es como si esos valores hubieran "aparecido" en el código.

Tenga en cuenta que las macros no tienen un tipo específico, *per se*. Realmente todo lo que ocurre es que son reemplazadas al por mayor por lo que sea que estén `#definidas`. Si el código C resultante no es válido, el compilador vomitará.

También puedes definir una macro sin valor:

```
#define EXTRA_HAPPY
```

en ese caso, la macro existe y está definida, pero está definida para no ser nada. Así que en cualquier lugar que aparezca en el texto será reemplazada por nada. Veremos un uso para esto más adelante.

Es convencional escribir los nombres de las macros en `ALL_CAPS` aunque técnicamente no sea necesario.

En general, esto le da una manera de definir valores constantes que son efectivamente globales y se pueden utilizar en cualquier lugar. Incluso en aquellos lugares donde una variable `const` no funcionaría, por ejemplo en `switch` `case`s y longitudes de array fijas.

Dicho esto, se debate en la red si una variable `const` tipada es mejor que la macro `#define` en el caso general.

También puede usarse para reemplazar o modificar palabras clave, un concepto completamente ajeno a `const`, aunque esta práctica debería usarse con moderación.

## 19.3 Compilación condicional

Es posible hacer que el preprocesador decida si presentar o no ciertos bloques de código al compilador, o simplemente eliminarlos por completo antes de la compilación.

Para ello, básicamente envolvemos el código en bloques condicionales, similares a las sentencias `if`-`else`.

### 19.3.1 Si está definido, `#ifdef` y `#endif`.

En primer lugar, vamos a intentar compilar código específico dependiendo de si una macro está o no definida.

```
#include <stdio.h>

#define EXTRA_HAPPY

int main(void)
{

#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#endif

    printf("OK!\n");
}
```

En ese ejemplo, definimos `EXTRA_HAPPY` (para que no sea nada, pero *está* definido), luego en la línea 8 comprobamos si está definido con una directiva `#ifdef`. Si está definida, el código subsiguiente se incluirá hasta el `#endif`.

Por lo tanto, al estar definido, el código se incluirá para la compilación y la salida será:

```
I'm extra happy!
OK!
```

Si comentáramos el `#define`, así:

```
//#define EXTRA_HAPPY
```

entonces no se definiría, y el código no se incluiría en la compilación. Y la salida sería simplemente:

```
OK!
```

Es importante recordar que estas decisiones se toman en tiempo de compilación. El código se compila o elimina dependiendo de la condición. Esto contrasta con una sentencia `if` estándar que se evalúa mientras el programa se está ejecutando.

### 19.3.2 Si no está definido, `#ifndef`.

También existe el sentido negativo de «si se define»: «si no está definido», o `#ifndef`. Podríamos cambiar el ejemplo anterior para que salieran cosas diferentes en función de si algo estaba definido o no:

```
#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#endif

#ifndef EXTRA_HAPPY
    printf("I'm just regular\n");
#endif
```

Veremos una forma más limpia de hacerlo en la siguiente sección.

Volviendo a los archivos de cabecera, hemos visto cómo podemos hacer que los archivos de cabecera sólo se incluyan una vez envolviéndolos en directivas de preprocesador como esta:

```
#ifndef MYHEADER_H  // Primera línea de myheader.h
#define MYHEADER_H

int x = 12;

#endif  // Última línea de myheader.h
```

Esto demuestra cómo una macro persiste a través de archivos y múltiples `#include`s. Si aún no está definida, definámosla y compilemos todo el fichero de cabecera.

Pero la próxima vez que se incluya, vemos que `MYHEADER_H` *está* definida, así que no enviamos el fichero de cabecera al compilador— se elimina efectivamente.

### 19.3.3 `#else`

Pero eso no es todo lo que podemos hacer. También podemos añadir un ‘#else.

Modifiquemos el ejemplo anterior:

```
#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#else
    printf("I'm just regular\n");
#endif
```

Ahora, si `EXTRA_HAPPY` no está definido, entrará en la cláusula `#else` e imprimirá:

```
I'm just regular
```

### 19.3.4 Else-If: `#elifdef`, `#elifndef`

Esta función es nueva en C23.

¿Y si quiere algo más complejo? ¿Quizás necesitas una estructura en cascada if-else para que tu código se construya correctamente?

Por suerte tenemos estas directivas a nuestra disposición. Podemos usar `#elifdef` para «definir else if»:

```
#ifdef MODE_1
    printf("This is mode 1\n");
#elifdef MODE_2
```

```
    printf("This is mode 2\n");
#elifdef MODE_3
    printf("This is mode 3\n");
#else
    printf("This is some other mode\n");
#endif
```

Por otro lado, puede utilizar `#elifndef` para «else if not defined».

### 19.3.5 Condicional general: `#if`, `#elif`

Funciona de forma muy parecida a las directivas `#ifdef` y `#ifndef` en el sentido de que también puede tener un `#else` y todo termina con `#endif`.

La única diferencia es que la expresión constante después de `#if` debe evaluarse a verdadero (distinto de cero) para que el código en `#if` sea compilado. Así que en lugar de si algo está definido o no, queremos una expresión que se evalúe como verdadera.

```
#include <stdio.h>

#define HAPPY_FACTOR 1

int main(void)
{

#if HAPPY_FACTOR == 0
    printf("I'm not happy!\n");
#elif HAPPY_FACTOR == 1
    printf("I'm just regular\n");
#else
    printf("I'm extra happy!\n");
#endif

    printf("OK!\n");
}
```

De nuevo, para las cláusulas `#if` no emparejadas, el compilador ni siquiera verá esas líneas. Para el código anterior, después de que el preprocesador haya terminado con él, todo lo que el compilador ve es:

```
#include <stdio.h>

int main(void)
{

    printf("I'm just regular\n");

    printf("OK!\n");
}
```

Un truco que se utiliza es comentar un gran número de líneas rápidamente [1].

Si pones un `#if 0` («si false») al principio del bloque a comentar y un `#endif` al final, puedes conseguir este efecto:

[1] No siempre se puede envolver el código con comentarios `/* */` porque no se anidan

```
#if 0
    printf(«Todo este código»); /* está efectivamente */
    printf(«comentado»); // por el #if 0
#endif
```

¿Qué pasa si estás en un compilador pre-C23 y no tienes soporte para las directivas `#elifdef` o `#elifndef`? ¿Cómo podemos conseguir el mismo efecto con `#if`? Es decir, qué pasaría si quisiera esto

```
#ifdef FOO
    x = 2;
#elifdef BAR  // ERROR POTENCIAL: No soportado antes de C23
    x = 3;
#endif
```

¿Cómo podría hacerlo?

Resulta que hay un operador de preprocesador llamado `defined` que podemos usar con una sentencia `#if`.

Son equivalentes:

```
#ifdef FOO
#if defined FOO
#if defined(FOO)   // Paréntesis opcional
```

Como estos:

```
#ifndef FOO
#if !defined FOO
#if !defined(FOO)   // Paréntesis opcional
```

Observe que podemos utilizar el operador lógico estándar NOT (`!`) para «no definido».

¡Así que ahora estamos de vuelta en la tierra de `#if` y podemos usar `#elif` impunemente!

Este código roto:

```
#ifdef FOO
    x = 2;
#elifdef BAR  // ERROR POTENCIAL: No soportado antes de C23
    x = 3;
#endif
```

puede sustituirse por:

```
#if defined FOO
    x = 2;
#elif defined BAR
    x = 3;
#endif
```

### 19.3.6 Perder una macro: `#undef`

Si has definido algo pero ya no lo necesitas, puedes redefinirlo con `#undef`.

```
#include <stdio.h>

int main(void)
{
#define GOATS

#ifdef GOATS
    printf("Goats detected!\n");  // Imprime
#endif

#undef GOATS  // Hacer que GOATS ya no esté definido

#ifdef GOATS
    printf("Goats detected, again!\n"); // no imprime
#endif
}
```

## 19.4 Macros integradas

El estándar define un montón de macros incorporadas que puedes probar y utilizar para la compilación condicional. Veámoslas aquí.

### 19.4.1 Macros obligatorias

Todos ellos están definidos:

| Macro | Descripción |
|---|---|
| `__DATE__` | La fecha de compilación –como cuando está compilando este archivo– en formato `Mmm dd yyyy` |
| `__TIME__` | La hora de compilación en formato `hh:mm:ss`. |
| `__FILE__` | Una cadena que contiene el nombre de este archivo |
| `__LINE__` | El número de línea del archivo en el que aparece esta macro |
| `__func__` | El nombre de la función en la que aparece, como una cadena[2]. |
| `__STDC__` | Definido con `1` si se trata de un compilador C estándar |
| `__STDC_HOSTED__` | Será `1` si el compilador es una *implementación hospedada*[3], en caso contrario `0`. |
| `__STDC_VERSION__` | Esta versión de C, una constante `long int` de la forma `yyyymmL`, por ejemplo `201710L`. |

Pongámoslos juntos.

```
#include <stdio.h>

int main(void)
{
    printf("Esta función: %s\n", __func__);
    printf("Este archivo %s\n", __FILE__);
```

[2]Esto no es realmente una macro—es técnicamente un identificador. Pero es el único identificador predefinido y se parece mucho a una macro, así que lo incluyo aquí. Como un rebelde

[3]Una implementación hospedada significa básicamente que estás ejecutando el estándar C completo, probablemente en un sistema operativo de algún tipo. Lo cual es probable. Si se está ejecutando en un sistema embebido, probablemente se trate de una implementación *standalone*

```
    printf("Esta linea: %d\n", __LINE__);
    printf("Compilado en: %s %s\n", __DATE__, __TIME__);
    printf("Versión de C: %ld\n", __STDC_VERSION__);
}
```

La salida en mi sistema es:

```
Esta función: main
Este archivo: foo.c
Esta linea: 7
Compilado en: Nov 23 2020 17:16:27
Versión de C : 201710
```

`__FILE__`, `__func__` y `__LINE__` son particularmente útiles para informar de condiciones de error en mensajes a los desarrolladores. La macro `assert()` de `<assert.h>` las utiliza para indicar en qué parte del código ha fallado la aserción.

#### 19.4.1.1 `__STDC_VERSION__`s

Por si te lo estás preguntando, aquí tienes los números de versión de las distintas versiones principales de la especificación del lenguaje C:

| Release | ISO/IEC version | `__STDC_VERSION__` |
|---|---|---|
| C89 | ISO/IEC 9899:1990 | undefined |
| **C89** | ISO/IEC 9899:1990/Amd.1:1995 | `199409L` |
| **C99** | ISO/IEC 9899:1999 | `199901L` |
| **C11** | ISO/IEC 9899:2011/Amd.1:2012 | `201112L` |

Tenga en cuenta que la macro no existía originalmente en C89.

También ten en cuenta que la idea es que los números de versión aumenten de manera estricta, así que siempre podrías verificar, por ejemplo, 'al menos C99' con:

```
#if __STDC_VERSION__ >= 1999901L
```

### 19.4.2 Macros opcionales

Es posible que su aplicación también los defina. O puede que no.

| Macro | Descripción |
|---|---|
| `__STDC_ISO_10646__` | Si está definido, `wchar_t` contiene valores Unicode, si no, otra cosa |
| `__STDC_MB_MIGHT_NEQ_WC__` | Un "1" indica que los valores en caracteres multibyte pueden no corresponderse con los valores en caracteres anchos. |
| `__STDC_UTF_16__` | Un `1` indica que el sistema utiliza la codificación UTF-16 en el tipo `char16_t`. |
| `__STDC_UTF_32__` | A `1` indicates that the system uses UTF-32 encoding in type `char32_t` |
| `__STDC_ANALYZABLE__` | Un `1` indica que el código es analizable[4]. |
| `__STDC_IEC_559__` | `1` if IEEE-754 (aka IEC 60559) floating point is supported |
| `__STDC_IEC_559_COMPLEX__` | `1` si se admite la coma flotante compleja IEC 60559 |

[4] OK, sé que era una respuesta evasiva. Básicamente hay una extensión opcional que los compiladores pueden implementar en la que se comprometen a limitar ciertos tipos de comportamiento indefinido para que el código C sea más susceptible de análisis estático. Es poco probable que necesites usar esto

| Macro | Descripción |
|---|---|
| `__STDC_LIB_EXT1__` | `1` si esta implementación admite una serie de funciones de biblioteca estándar alternativas "seguras" (tienen sufijos `_s` en el nombre) |
| `__STDC_NO_ATOMICS__` | `1` si esta implementación **no** soporta `_Atomic` o `<stdatomic.h>`. |
| `__STDC_NO_COMPLEX__` | `1` si esta implementación **no** soporta tipos complejos o `<complex.h>`. |
| `__STDC_NO_THREADS__` | `1` si esta implementación **no** es compatible con `<threads.h>`. |
| `__STDC_NO_VLA__` | `1` si esta implementación **no** admite matrices de longitud variable |

## 19.5 Macros con argumentos

Sin embargo, las macros son más potentes que una simple sustitución. También puede configurarlas para que acepten argumentos que sean sustituidos.

A menudo surge la pregunta de cuándo utilizar macros parametrizadas frente a funciones. Respuesta corta: usa funciones. Pero verás muchas macros en la naturaleza y en la biblioteca estándar. La gente tiende a usarlas para cosas cortas y matemáticas, y también para características que pueden cambiar de plataforma a plataforma. Puedes definir diferentes palabras clave para una plataforma u otra.

### 19.5.1 Macros con un argumento

Empecemos con uno sencillo que eleva un número al cuadrado:

```
#include <stdio.h>

#define SQR(x) x * x  // No es del todo correcto, pero ten paciencia conmigo

int main(void)
{
    printf("%d\n", SQR(12));  // 144
}
```

Lo que está diciendo es "dondequiera que veas `SQR` con algún valor, reemplázalo con ese valor multiplicado por sí mismo".

Así que la línea 7 se cambiará a:

```
    printf("%d\n", 12 * 12);  // 144
```

que C convierte cómodamente en 144.

Pero en esa macro hemos cometido un error elemental que debemos evitar.

Vamos a comprobarlo. ¿Y si quisiéramos calcular `SQR(3 + 4)`? Bueno, $3 + 4 = 7$, así que debemos querer calcular $7^2 = 49$. Eso es; `49`—respuesta final.

Pongámoslo en nuestro código y veremos que obtenemos… 19?

```
    printf("%d\n", SQR(3 + 4));  // 19!!??
```

¿Qué ha pasado?

Si seguimos la macro expansión, obtenemos

```
    printf("%d\n", 3 + 4 * 3 + 4);  // 19!
```

¡Uy! Como la multiplicación tiene prioridad, hacemos primero $4 + 3 = 12$ y obtenemos $3 + 12 + 4 = 19$. No es lo que buscábamos.

Así que tenemos que arreglar esto para hacerlo bien.

**Esto es tan común que deberías hacerlo automáticamente cada vez que hagas una macro matemática parametrizada.**

La solución es fácil: ¡sólo tienes que añadir algunos paréntesis!

```
#define SQR(x) (x) * (x)   // Mejor... ¡pero aún no lo suficiente!
```

Y ahora nuestra macro se expande a:

```
    printf("%d\n", (3 + 4) * (3 + 4));  // 49! Woo hoo!
```

Pero en realidad seguimos teniendo el mismo problema que podría manifestarse si tenemos cerca un operador de mayor precedencia que multiplicar (`*`).

Así que la forma segura y adecuada de armar la macro es envolver todo entre paréntesis adicionales, así:

```
#define SQR(x) ((x) * (x))   // Perfecto!
```

Acostúmbrate a hacerlo cuando hagas una macro matemática y no te equivocarás.

### 19.5.2 Macros con más de un argumento

Puedes apilar estas cosas tanto como quieras:

```
#define TRIANGLE_AREA(w, h) (0.5 * (w) * (h))
```

Vamos a hacer unas macros que resuelven para $x$ usando la fórmula cuadrática. Por si acaso no la tienes en la cabeza, dice que para ecuaciones de la forma:

$$ax^2 + bx + c = 0$$

puedes resolver $x$ con la fórmula cuadrática:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Lo cual es una locura. También observe el más-o-menos ($\pm$) allí, lo que indica que en realidad hay dos soluciones.

Así que vamos a hacer macros para ambos:

```
#define QUADP(a, b, c) ((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c) ((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
```

Así que eso nos da algunas matemáticas. Pero vamos a definir una más que podemos utilizar como argumentos a `printf()` para imprimir ambas respuestas.

```
//          macro           se reemplaza por
//      |-----------| |---------------------------|
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)
```

Eso es sólo un par de valores separados por una coma - y podemos usar eso como un argumento "combinado" de clases a `printf()` como esto:

```
printf("x = %f or x = %f\n", QUAD(2, 10, 5));
```

Pongámoslo junto en algún código:

```
#include <stdio.h>
#include <math.h>  // Para sqrt()

#define QUADP(a, b, c) ((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c) ((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)

int main(void)
{
    printf("2*x^2 + 10*x + 5 = 0\n");
    printf("x = %f or x = %f\n", QUAD(2, 10, 5));
}
```

Y esto nos da la salida:

```
2*x^2 + 10*x + 5 = 0
x = -0.563508 or x = -4.436492
```

Si introducimos cualquiera de estos valores, obtendremos aproximadamente cero (un poco desviado porque los números no son exactos):

$2 \times -0.563508^2 + 10 \times -0.563508 + 5 \approx 0.000003$

### 19.5.3 Macros con argumentos variables

También hay una forma de pasar un número variable de argumentos a una macro, utilizando elipses (`...`) después de los argumentos conocidos con nombre. Cuando se expande la macro, todos los argumentos extra estarán en una lista separada por comas en la macro `__VA_ARGS__`, y pueden ser reemplazados desde allí:

```
#include <stdio.h>

// Combinar los dos primeros argumentos a un solo número,
// luego tener un commalist del resto de ellos:
#define X(a, b, ...) (10*(a) + 20*(b)), __VA_ARGS__

int main(void)
{
    printf("%d %f %s %d\n", X(5, 4, 3.14, "Hi!", 12));
}
```

La sustitución que tiene lugar en la línea 10 sería:

```
    printf("%d %f %s %d\n", (10*(5) + 20*(4)), 3.14, "Hi!", 12);
```

para la salida:

```
130 3.140000 Hi! 12
```

También se puede "stringificar" `__VA_ARGS__` anteponiéndole un `#`:

```
#define X(...) #__VA_ARGS__

printf("%s\n", X(1,2,3));  // Imprime "1, 2, 3"
```

### 19.5.4 Stringificación

Ya se ha mencionado, justo arriba, que puede convertir cualquier argumento en una cadena precediéndolo de un `#` en el texto de sustitución.

Por ejemplo, podríamos imprimir cualquier cosa como una cadena con esta macro y `printf()`:

```
#define STR(x) #x

printf("%s\n", STR(3.14159));
```

En ese caso, la sustitución conduce a:

```
printf("%s\n", "3.14159");
```

Veamos si podemos usar esto con mayor efecto para que podamos pasar cualquier nombre de variable `int` a una macro, y hacer que imprima su nombre y valor.

```
#include <stdio.h>

#define PRINT_INT_VAL(x) printf("%s = %d\n", #x, x)

int main(void)
{
    int a = 5;

    PRINT_INT_VAL(a);  // Imprime "a = 5"
}
```

En la línea 9, obtenemos la siguiente macro de sustitución:

```
    printf("%s = %d\n", "a", 5);
```

### 19.5.5 Concatenación

También podemos concatenar dos argumentos con `##`. ¡Qué divertido!

```
#define CAT(a, b) a ## b
```

```
printf("%f\n", CAT(3.14, 1592));   // 3.141592
```

## 19.6 Macros multilínea

Es posible continuar una macro en varias líneas si se escapa la nueva línea con una barra invertida (`\`).

Escribamos una macro multilínea que imprima números desde `0` hasta el producto de los dos argumentos pasados.

```
#include <stdio.h>

#define PRINT_NUMS_TO_PRODUCT(a, b) do { \
    int product = (a) * (b); \
    for (int i = 0; i < product; i++) { \
        printf("%d\n", i); \
    } \
} while(0)

int main(void)
{
    PRINT_NUMS_TO_PRODUCT(2, 4);  // Salida de números del 0 al 7
}
```

Un par de cosas a tener en cuenta:

- Escapes al final de cada línea excepto la última para indicar que la macro continúa.
- Todo está envuelto en un bucle `do-while(0)` con llaves de ardilla.

El último punto puede ser un poco raro, pero se trata de absorber el `;` final que el programador deja caer después de la macro.

Al principio pensé que bastaría con usar llaves de ardilla, pero hay un caso en el que falla si el programador pone un punto y coma después de la macro. Este es el caso:

```
#include <stdio.h>

#define FOO(x) { (x)++; }

int main(void)
{
    int i = 0;

    if (i == 0)
        FOO(i);
    else
        printf(":-(\n");

    printf("%d\n", i);
}
```

Parece bastante simple, pero no se construye sin un error de sintaxis:

```
foo.c:11:5: error: ‘else’ without a previous ‘if’
```

¿Lo ve?

Veamos la expansión:

```
    if (i == 0) {
        (i)++;
    };              // <-- ¡Problema con MAYÚSCULAS!

    else
        printf(":-(\n");
```

El `;` pone fin a la sentencia `if`, así que el `else` queda flotando por ahí ilegalmente[5].

Así que envuelve esa macro multilínea con un `do-while(0)`.

## 19.7 Ejemplo: Una macro Assert

Añadir asserts a tu código es una buena forma de detectar condiciones que crees que no deberían ocurrir. C proporciona la funcionalidad `assert()`. Comprueba una condición, y si es falsa, el programa explota diciéndote el fichero y el número de línea en el que falló la aserción.

Pero esto es insuficiente.

1. En primer lugar, no puedes especificar un mensaje adicional con la aserción.
2. En segundo lugar, no hay un interruptor fácil de encendido y apagado para todas las aserciones.

Podemos abordar el primero con macros.

Básicamente, cuando tengo este código:

```
ASSERT(x < 20, "x debe tener menos de 20 años");
```

Quiero que ocurra algo como esto (asumiendo que `ASSERT()` está en la línea 220 de `foo.c`):

```
if (!(x < 20)) {
    fprintf(stderr, "foo.c:220: assertion x < 20 failed: ");
    fprintf(stderr, "x debe tener menos de 20 años\n");
    exit(1);
}
```

Podemos obtener el nombre del fichero de la macro `__FILE__`, y el número de línea de `__LINE__`. El mensaje ya es una cadena, pero `x < 20` no lo es, así que tendremos que encadenarla con `#`. Podemos hacer una macro multilínea utilizando barras invertidas al final de la línea.

```
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
```

[5] *Quebrantando la ley… quebrantando la ley…*

(Parece un poco raro con `__FILE__` así delante, pero recuerda que es una cadena literal, y las cadenas literales una al lado de la otra se concatenan automáticamente. En cambio, `__LINE__` es sólo un `int`).

¡Y funciona! Si ejecuto esto

```
int x = 30;

ASSERT(x < 20, "x debe tener menos de 20 años");
```

Obtengo este resultado:

```
foo.c:23: assertion x < 20 failed: x must be under 20
```

¡Muy bonito!

Lo único que falta es una forma de activarlo y desactivarlo, y podríamos hacerlo con compilación condicional.

Aquí está el ejemplo completo:

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

#define ASSERT_ENABLED 1

#if ASSERT_ENABLED
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
#else
#define ASSERT(c, m)  // Macro vacía si no está activada
#endif

int main(void)
{
    int x = 30;

    ASSERT(x < 20, "x debe tener menos de 20 años");
}
```

Esto tiene la salida:

```
foo.c:23: assertion x < 20 failed: x must be under 20
```

## 19.8 Directiva `#error`

Esta directiva hace que el compilador se equivoque en cuanto la vea.

Normalmente, se utiliza dentro de una condicional para evitar la compilación a menos que se cumplan algunos requisitos previos:

```
#ifndef __STDC_IEC_559__
    #error I really need IEEE-754 floating point to compile. Sorry!
#endif
```

Algunos compiladores tienen una directiva complementaria no estándar `#warning` que mostrará una advertencia pero no detendrá la compilación, pero esto no está en la especificación C11.

## 19.9 Directiva `#embed`

¡Nuevo en C23!

Y actualmente todavía no funciona con ninguno de mis compiladores, ¡así que tómate esta sección con un grano de sal!

La esencia de esto es que puedes incluir bytes de un fichero como constantes enteras como si los hubieras tecleado.

Por ejemplo, si tienes un archivo binario llamado `foo.bin` que contiene cuatro bytes con valores decimales 11, 22, 33, y 44, y haces esto:

```
int a[] = {
#embed "foo.bin"
};
```

Será como si lo hubieras escrito tú:

```
int a[] = {11,22,33,44};
```

Se trata de una forma muy eficaz de inicializar una matriz con datos binarios sin necesidad de convertirlos primero en código: ¡el preprocesador lo hace por ti!

Un caso de uso más típico podría ser un archivo que contenga una pequeña imagen para mostrar y que no quieras cargar en tiempo de ejecución.

He aquí otro ejemplo:

```
int a[] = {
#embed <foo.bin>
};
```

Si utiliza corchetes angulares, el preprocesador busca en una serie de lugares definidos por la implementación para localizar el archivo, igual que haría `#include`. Si utiliza comillas dobles y el recurso no se encuentra, el compilador lo intentará como si hubiera utilizado paréntesis angulares en un último intento desesperado por encontrar el archivo.

`#embed` funciona como `#include` en el sentido de que pega los valores antes de que el compilador los vea. Esto significa que puedes usarlo en todo tipo de lugares:

```
return
#embed "somevalue.dat"
;
```

o

```
int x =
#embed "xvalue.dat"
;
```

¿Son siempre bytes? ¿Significa que tendrán valores de 0 a 255, ambos inclusive? La respuesta es definitivamente por defecto “sí”, excepto cuando es “no”.

Técnicamente, los elementos serán `CHAR_BIT` bits de ancho. Y es muy probable que sean 8 en tu sistema, por lo que obtendrías ese rango de 0 a 255 en tus valores. (Siempre serán no negativos).

Además, es posible que una implementación permita que esto se anule de alguna manera, por ejemplo, en la línea de comandos o con parámetros.

El tamaño del fichero en bits debe ser múltiplo del tamaño del elemento. Es decir, si cada elemento tiene 8 bits, el tamaño del fichero (en bits) debe ser múltiplo de 8. En el uso cotidiano, esta es una forma confusa de decir que cada fichero debe tener un número entero de bytes… que por supuesto lo es. Honestamente, ni siquiera estoy seguro de por qué me molesté con este párrafo. Lee la especificación si realmente tienes curiosidad.

### 19.9.1 Parámetro `#embed`

Hay todo tipo de parámetros que puedes especificar a la directiva `#embed`. He aquí un ejemplo con el parámetro aún no introducido `limit()`:

```
int a[] = {
#embed "/dev/random" limit(5)
};
```

Pero, ¿y si ya tienes definido `limit` en otro lugar? Afortunadamente puedes poner `__` alrededor de la palabra clave y funcionará de la misma manera:

```
int a[] = {
#embed "/dev/random" __limit__(5)
};
```

Ahora… ¿qué es eso de “límite”?

### 19.9.2 Parámetro `limit()`

Puede especificar un límite en el número de elementos a incrustar con este parámetro.

Se trata de un valor máximo, no de un valor absoluto. Si el fichero que se incrusta es más corto que el límite especificado, sólo se importarán esa cantidad de bytes.

El ejemplo `/dev/random` de arriba es un ejemplo de la motivación para esto—en Unix, eso es un *archivo de dispositivo de caracteres* que devolverá un flujo infinito de números bastante aleatorios.

Incrustar un número infinito de bytes es duro para tu RAM, así que el parámetro `limit` te da una forma de parar después de un cierto número.

Finalmente, puedes usar macros `#define` en tu `limit`, por si tienes curiosidad.

### 19.9.3 Parámetro `if_empty`

Este parámetro define cuál debe ser el resultado de la incrustación si el fichero existe pero no contiene datos. Supongamos que el fichero `foo.dat` contiene un único byte con el valor 123. Si hacemos esto

```
int x =
#embed "foo.dat" if_empty(999)
;
```

lo conseguiremos:

```
int x = 123;   // Cuando foo.dat contiene un byte 123
```

Pero, ¿y si el archivo `foo.dat` tiene cero bytes (es decir, no contiene datos y está vacío)? En ese caso, se expandiría a:

```
int x = 999;   // Cuando foo.dat está vacío
```

En particular, si el `limit` se establece en `0`, entonces el `if_empty` siempre será sustituido. Es decir, un límite cero significa que el fichero está vacío.

Esto siempre emitirá `x = 999` sin importar lo que haya en `foo.dat`:

```
int x =
#embed "foo.dat" limit(0) if_empty(999)
;
```

### 19.9.4 Parámetros `prefix()` y `suffix()`.

Esta es una manera de anteponer algunos datos en el embed.

Tenga en cuenta que esto sólo afecta a los datos que no están vacíos. Si el fichero está vacío, ni `prefix` ni `suffix` tienen efecto.

Aquí hay un ejemplo en el que incrustamos tres números aleatorios, pero les ponemos como prefijo `11,` y como sufijo `,99`:

```
int x[] = {
#embed "/dev/urandom" limit(3) prefix(11,) suffix(,99)
};
```

Ejemplo de resultado:

```
int x[] = {11,135,116,220,99};
```

No es obligatorio utilizar tanto `prefix` como `suffix`. Puedes usar ambos, uno, el otro, o ninguno.

Podemos hacer uso de la característica de que estos sólo se aplican a los archivos no vacíos para un efecto limpio, como se muestra en el siguiente ejemplo descaradamente robado de la especificación.

Supongamos que tenemos un archivo `foo.dat` que contiene algunos datos. Y queremos usar esto para inicializar un array, y entonces queremos un sufijo en el array que sea un elemento cero.

No hay problema, ¿verdad?

```
int x[] = {
#embed "foo.dat" suffix(,0)
};
```

Si `foo.dat` tiene 11, 22 y 33, obtendríamos:

```
int x[] = {11,22,33,0};
```

Pero, ¡espera! ¿Y si `foo.dat` está vacío? Entonces obtenemos:

```
int x[] = {};
```

y eso no es bueno.

Pero podemos arreglarlo así:

```
int x[] = {
#embed "foo.dat" suffix(,)
    0
};
```

Dado que el parámetro `suffix` se omite si el archivo está vacío, esto se convertiría simplemente en:

```
int x[] = {0};
```

lo cual está bien.

### 19.9.5 El identificador `__has_embed()`.

Esta es una gran manera de comprobar si un archivo en particular está disponible para ser incrustado, y también si está vacío o no.

Se usa con la directiva `#if`.

Aquí hay un trozo de código que obtendrá 5 números aleatorios del dispositivo de caracteres generador de números aleatorios. Si no existen, intenta obtenerlos de un fichero `myrandoms.dat`. Si no existe, utiliza algunos valores codificados:

```
    int random_nums[] = {
#if __has_embed("/dev/urandom")
    #embed "/dev/urandom" limit(5)
#elif __has_embed("myrandoms.dat")
    #embed "myrandoms.dat" limit(5)
#else
    140,178,92,167,120
#endif
    };
```

Técnicamente, el identificador `__has_embed()` resuelve a uno de tres valores:

| `__has_embed()` Result | Descripción |
|---|---|
| `__STDC_EMBED_NOT_FOUND__` | Si no se encuentra el archivo. |
| `__STDC_EMBED_FOUND__` | Si se encuentra el archivo y no está vacío. |
| `__STDC_EMBED_EMPTY` | Si se encuentra el archivo y está vacío. |

Tengo buenas razones para creer que `__STDC_EMBED_NOT_FOUND__` es `0` y los otros no son cero (porque está implícito en la propuesta y tiene sentido lógico), pero tengo problemas para encontrarlo en esta versión del borrador de la especificación.

TODO

### 19.9.6 Otros parámetros

La implementación de un compilador puede definir otros parámetros incrustados todo lo que quiera—busque estos parámetros no estándar en la documentación de su compilador.

Por ejemplo:

```
#embed "foo.bin" limit(12) frotz(lamp)
```

Normalmente llevan un prefijo para facilitar el espaciado entre nombres:

```
#embed "foo.bin" limit(12) fmc::frotz(lamp)
```

Puede ser sensato intentar detectar si están disponibles antes de usarlos, y por suerte podemos usar `__has_embed` para ayudarnos aquí.

Normalmente, `__has_embed()` nos dirá si el fichero está ahí o no. Pero, y aquí viene lo divertido, ¡también devolverá false si algún parámetro adicional no está soportado!

Así que si le damos un fichero que *sabemos* que existe y un parámetro cuya existencia queremos comprobar, nos dirá efectivamente si ese parámetro está soportado.

Pero, ¿qué fichero existe *siempre*? Resulta que podemos usar la macro `__FILE__`, que se expande al nombre del fichero fuente que lo referencia. Ese fichero *debe* existir, o algo va muy mal en el departamento del huevo y la gallina.

Probemos el parámetro `frotz` para ver si podemos usarlo:

```
#if __has_embed(__FILE__ fmc::frotz(lamp))
    puts("fmc::frotz(lamp) is supported!");
#else
    puts("fmc::frotz(lamp) is NOT supported!");
#endif
```

### 19.9.7 Incrustación de valores multibyte

¿Qué tal si en lugar de bytes individuales se introducen `int`s? ¿Qué pasa con los valores multibyte en el archivo incrustado?

El estándar C23 no lo admite, pero en el futuro podrían definirse extensiones de implementación para ello.

## 19.10 La directiva `#pragma`

Se trata de una directiva peculiar, abreviatura de “pragmática”. Puedes usarla para hacer… bueno, cualquier cosa que tu compilador te permita hacer con ella.

Básicamente la única vez que vas a añadir esto a tu código es si alguna documentación te dice que lo hagas.

### 19.10.1 Pragmas no estándar

He aquí un ejemplo no estándar de uso de `#pragma` para hacer que el compilador ejecute un bucle `for` en paralelo con múltiples hilos (si el compilador soporta la extensión OpenMP[6]):

[6] https://www.openmp.org/

```
#pragma omp parallel for
for (int i = 0; i < 10; i++) { ... }
```

Hay todo tipo de directivas `#pragma` documentadas en las cuatro esquinas del globo.

Todos los `#pragma`s no reconocidos son ignorados por el compilador.

### 19.10.2 Pragmas estándar

También hay algunas estándar, que empiezan por `STDC` y siguen la misma forma:

```
#pragma STDC pragma_name on-off
```

La parte `on-off` puede ser `ON`, `OFF`, o `DEFAULT`.

Y el `pragma_name` puede ser uno de estos:

| Nombre del pragma | Descripción |
|---|---|
| `FP_CONTRACT` | Permitir que las expresiones en coma flotante se contraigan en una sola operación para evitar los errores de redondeo que podrían producirse por múltiples operaciones. |
| `FENV_ACCESS` | Póngalo a `ON` si planea acceder a las banderas de estado de coma flotante. Si está `OFF`, el compilador puede realizar optimizaciones que causen que los valores de las banderas sean inconsistentes o inválidos. |
| `CX_LIMITED_RANGE` | Establezca a `ON` para permitir que el compilador omita las comprobaciones de desbordamiento al realizar aritmética compleja. Por defecto es `OFF`. |

Por ejemplo:

```
#pragma STDC FP_CONTRACT OFF
#pragma STDC CX_LIMITED_RANGE ON
```

En cuanto a `CX_LIMITED_RANGE`, la especificación señala:

> El propósito del pragma es permitir a la implementación utilizar las fórmulas:
> $(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$
> $(x + iy)/(u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$
> $|x + iy| = \sqrt{x^2 + y^2}$
> donde el programador puede determinar que son seguros.

### 19.10.3 Operador `_Pragma`

Esta es otra forma de declarar un pragma que podría utilizar en una macro.

Son equivalentes:

```
#pragma "Unnecessary" quotes
_Pragma("\"Unnecessary\" quotes")
```

Esto se puede utilizar en una macro, si es necesario:

```
#define PRAGMA(x) _Pragma(#x)
```

## 19.11 La directiva `#line`

Esto le permite anular los valores de `__LINE__` y `__FILE__`. Si lo desea.

Nunca he querido hacer esto, pero en K&R2, escriben:

> Para el beneficio de otros preprocesadores que generan programas C […]

Así que tal vez haya eso.

Para anular el número de línea a, digamos 300:

```
#line 300
```

y `__LINE__` seguirá contando a partir de ahí.

Para anular el número de línea y el nombre de fichero:

```
#line 300 "newfilename"
```

## 19.12 La Directiva Nula (`#`)

Un `#` en una línea por sí mismo es ignorado por el preprocesador. Ahora, para ser totalmente honesto, no sé cuál es el caso de uso para esto.

He visto ejemplos como este:

```
#ifdef FOO
    #
#else
    printf("Something");
#endif
```

que es sólo cosmético; la línea con el solitario `#` puede ser eliminado sin ningún efecto nocivo.

O tal vez por coherencia cosmética, así:

```
#
#ifdef FOO
    x = 2;
#endif
#
#if BAR == 17
    x = 12;
#endif
#
```

Pero, con respecto a la cosmética, eso es simplemente feo.

Otro post menciona la eliminación de comentarios—que en GCC, un comentario después de un `#` no será visto por el compilador. No lo dudo, pero la especificación no parece decir que este sea el comportamiento estándar.

Mis búsquedas de fundamentos no están dando muchos frutos. Así que voy a decir que esto es algo de esoterismo de C.

# Chapter 20

# `structs` II: Más diversión con `structs`

Resulta que hay mucho más que puedes hacer con `structs` de lo que hemos hablado, pero es sólo un gran montón de cosas varias. Así que las incluiremos en este capítulo.

Si eres bueno con lo básico de `structs`, puedes completar tus conocimientos aquí.

## 20.1 Inicializadores de `structs` anidadas y matrices

¿Recuerdas cómo podías inicializar los miembros de la estructura siguiendo estas líneas?

```
struct foo x = {.a=12, .b=3.14};
```

Resulta que tenemos más potencia en estos inicializadores de la que habíamos compartido en un principio. ¡Interesante!

Por un lado, si tienes una subestructura anidada como la siguiente, puedes inicializar miembros de esa subestructura siguiendo los nombres de las variables línea abajo:

```
struct foo x = {.a.b.c=12};
```

Veamos un ejemplo:

```
#include <stdio.h>

struct cabin_information {
    int window_count;
    int o2level;
};

struct spaceship {
    char *manufacturer;
    struct cabin_information ci;
};

int main(void)
{
    struct spaceship s = {
```

```
        .manufacturer="General Products",
        .ci.window_count = 8,   // <-- ¡INICIALIZADOR ANIDADO!
        .ci.o2level = 21
    };

    printf("%s: %d seats, %d%% oxygen\n",
        s.manufacturer, s.ci.window_count, s.ci.o2level);
}
```

Fíjate en las líneas 16-17. Ahí es donde estamos inicializando los miembros de la `struct cabin_information` en la definición de `s`, nuestra `struct spaceship`.

Y aquí hay otra opción para ese mismo inicializador - esta vez vamos a hacer algo más estándar, pero cualquiera de los enfoques funciona:

```
    struct spaceship s = {
        .manufacturer="General Products",
        .ci={
            .window_count = 8,
            .o2level = 21
        }
    };
```

Como si la información anterior no fuera lo suficientemente espectacular, también podemos mezclar inicializadores de matrices.

Vamos a cambiar esto para obtener una matriz de información de pasajeros allí, y podemos comprobar cómo los inicializadores trabajan allí, también.

```
#include <stdio.h>

struct passenger {
    char *name;
    int covid_vaccinated; // Booleano
};

#define MAX_PASSENGERS 8

struct spaceship {
    char *manufacturer;
    struct passenger passenger[MAX_PASSENGERS];
};

int main(void)
{
    struct spaceship s = {
        .manufacturer="General Products",
        .passenger = {
            // Inicializar un campo cada vez
            [0].name = "Gridley, Lewis",
            [0].covid_vaccinated = 0,

            // O todos a la vez
```

```
            [7] = {.name="Brown, Teela", .covid_vaccinated=1},
        }
    };

    printf("Passengers for %s ship:\n", s.manufacturer);

    for (int i = 0; i < MAX_PASSENGERS; i++)
        if (s.passenger[i].name != NULL)
            printf("    %s (%svaccinated)\n",
                s.passenger[i].name,
                s.passenger[i].covid_vaccinated? "": "not ");
}
```

## 20.2 `structs` anonimas

Son las “estructuras sin nombre”. También las mencionamos en la sección `typedef`, pero las refrescaremos aquí.

Aquí tenemos una `struct` normal:

```
struct animal {
    char *name;
    int leg_count, speed;
};
```

Y aquí está el equivalente anónimo:

```
struct {             // <-- Sin nombre
    char *name;
    int leg_count, speed;
};
```

Okaaaaay. ¿Así que tenemos una “estructura”, pero no tiene nombre, por lo que no tenemos manera de utilizarla más tarde? Parece bastante inútil.

Es cierto que en ese ejemplo lo es. Pero todavía podemos hacer uso de ella de un par de maneras.

Una es rara, pero como la `struct` anónima representa un tipo, podemos simplemente poner algunos nombres de variables después de ella y usarlos.

```
struct {                  // <-- ¡Sin nombre!
    char *name;
    int leg_count, speed;
} a, b, c;                // 3 variables de este tipo struct

a.name = "antelope";
c.leg_count = 4;          // Por ejemplo
```

Pero sigue sin ser muy útil.

Mucho más común es el uso de `structs` anónimas con un `typedef` para que podamos usarlo más tarde (por ejemplo, para pasar variables a funciones).

```
typedef struct {                    // <-- ¡Sin nombre!
    char *name;
    int leg_count, speed;
} animal;                           // Nuevo tipo: animal

animal a, b, c;

a.name = "antelope";
c.leg_count = 4;           // Por ejemplo
```

Personalmente, no utilizo muchas `structs` anónimas. Creo que es más agradable ver el `struct animal` completo antes del nombre de la variable en una declaración.

Pero eso es sólo mi opinión.

## 20.3 `structs` (Estructuras autorreferenciales)

Para cualquier estructura de datos tipo grafo, es útil poder tener punteros a los nodos/vértices conectados. Pero esto significa que en la definición de un nodo, es necesario tener un puntero a un nodo. Es un rollo.

Pero resulta que se puede hacer esto en C sin ningún problema.

Por ejemplo, aquí hay un nodo de lista enlazada:

```
struct node {
    int data;
    struct node *next;
};
```

Es importante tener en cuenta que `next` es un puntero. Esto es lo que permite todo el asunto incluso construir. A pesar de que el compilador no sabe cómo es el nodo `struct` completo, todos los punteros tienen el mismo tamaño.

Aquí hay un programa de lista enlazada para probarlo:

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

struct node {
    int data;
    struct node *next;
};

int main(void)
{
    struct node *head;

    // Hackishly configurar una lista enlazada (11)->(22)->(33)
    head = malloc(sizeof(struct node));
    head->data = 11;
    head->next = malloc(sizeof(struct node));
    head->next->data = 22;
    head->next->next = malloc(sizeof(struct node));
    head->next->next->data = 33;
```

```
    head->next->next->next = NULL;

    // Atraviésalo
    for (struct node *cur = head; cur != NULL; cur = cur->next) {
        printf("%d\n", cur->data);
    }
}
```

Corriendo que imprime:

```
11
22
33
```

## 20.4 Miembros flexibles de la matriz

En los viejos tiempos, cuando la gente tallaba el código C en madera, algunos pensaban que estaría bien poder asignar `struct`s que tuvieran arrays de longitud variable al final.

Quiero dejar claro que la primera parte de la sección es la forma antigua de hacer las cosas, y que después vamos a hacer las cosas de la forma nueva.

Por ejemplo, podrías definir una `struct` para contener cadenas y la longitud de esa cadena. Tendría una longitud y una matriz para contener los datos. Tal vez algo como esto:

```
struct len_string {
    int length;
    char data[8];
};
```

Pero eso tiene "8" codificados como la longitud máxima de una cadena, y eso no es mucho. ¿Qué pasa si hacemos algo *limpio* y simplemente `malloc()` algún espacio extra al final después de la estructura, y luego dejar que los datos se desborden en ese espacio?

Hagamos eso, y luego asignemos otros 40 bytes encima:

```
struct len_string *s = malloc(sizeof *s + 40);
```

Como `data` es el último campo de la `struct`, si desbordamos ese campo, ¡se acaba el espacio que ya habíamos asignado! Por esta razón, este truco sólo funciona si el array corto es el *último* campo de la `struct`.

```
// Copie más de 8 bytes

strcpy(s->data, "Hello, world!");  // No se estrellará. Probablemente.
```

De hecho, existía una solución común en el compilador para hacer esto, en la que se asignaba un array de longitud cero al final:

```
struct len_string {
    int length;
    char data[0];
};
```

Y entonces cada byte extra que asignaste estaba listo para ser usado en esa cadena.

Como `data` es el último campo de la `struct`, si desbordamos ese campo, ¡se acaba el espacio que ya habíamos asignado!

```
// Copie más de 8 bytes

strcpy(s->data, "Hello, world!");  // No se estrellará. Probablemente.
```

Pero, por supuesto, acceder a los datos más allá del final de la matriz es un comportamiento indefinido. En estos tiempos modernos, ya no nos dignamos a recurrir a semejante salvajada.

Por suerte para nosotros, todavía podemos conseguir el mismo efecto con C99 y posteriores, pero ahora es legal.

Cambiemos nuestra definición anterior para que el array no tenga tamaño[1]:

```
struct len_string {
    int length;
    char data[];
};
```

De nuevo, esto sólo funciona si el miembro del array flexible es el *último* campo de la `struct`.

Y entonces podemos asignar todo el espacio que queramos para esas cadenas haciendo `malloc()`mayor que la `struct len_string` de `construcción`, como hacemos en este ejemplo que hace una nueva `struct len_string` a partir de una cadena C:

```
struct len_string *len_string_from_c_string(char *s)
{
    int len = strlen(s);

    // Asignar "len" más bytes de los que normalmente necesitaríamos
    struct len_string *ls = malloc(sizeof *ls + len);

    ls->length = len;

    // Copia la cadena en esos bytes extra
    memcpy(ls->data, s, len);

    return ls;
}
```

## 20.5 Bytes de relleno

Tenga en cuenta que C puede añadir bytes de relleno dentro o después de una `struct` según le convenga. No puedes confiar en que estarán directamente adyacentes en memoria[2].

Echemos un vistazo a este programa. Obtenemos dos números. Uno es la suma del `tamaño de` los tipos de campo individuales. El otro es el tamaño de toda la estructura.

Es de esperar que sean iguales. El tamaño del total es el tamaño de la suma de sus partes, ¿verdad?

[1] Técnicamente decimos que tiene un *tipo incompleto*

[2] Aunque algunos compiladores tienen opciones para forzar que esto ocurra—busca `__attribute__((packed))` para ver cómo hacer esto con GCC

```
#include <stdio.h>

struct foo {
    int a;
    char b;
    int c;
    char d;
};

int main(void)
{
    printf("%zu\n", sizeof(int) + sizeof(char) + sizeof(int) + sizeof(char));
    printf("%zu\n", sizeof(struct foo));
}
```

Pero en mi sistema, esto sale:

```
10
16
```

No son iguales. El compilador ha añadido 6 bytes de relleno para mejorar el rendimiento. Puede que tu compilador te dé un resultado diferente, pero a menos que lo fuerces, no puedes estar seguro de que no haya relleno.

## 20.6 `offsetof`

En la sección anterior, vimos que el compilador podía inyectar bytes de relleno a voluntad dentro de una estructura.

¿Y si necesitáramos saber dónde están? Podemos medirlo con `offsetof`, definido en `<stddef.h>`.

Modifiquemos el código anterior para imprimir los desplazamientos de los campos individuales en la `struct`:

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

struct foo {
    int a;
    char b;
    int c;
    char d;
};

int main(void)
{
    printf("%zu\n", offsetof(struct foo, a));
    printf("%zu\n", offsetof(struct foo, b));
    printf("%zu\n", offsetof(struct foo, c));
    printf("%zu\n", offsetof(struct foo, d));
}
```

Para mí, estas salidas:

```
0
4
8
12
```

indicando que estamos utilizando 4 bytes para cada uno de los campos. Es un poco raro, porque `char` es sólo 1 byte, ¿verdad? El compilador está poniendo 3 bytes de relleno después de cada `char` para que todos los campos tengan 4 bytes. Presumiblemente esto se ejecutará más rápido en mi CPU.

## 20.7 Falsa OOP

Hay una cosa un poco abusiva que es una especie de OOP-como que se puede hacer con `structs`.

Dado que el puntero a la `struct` es el mismo que un puntero al primer elemento de la `struct`, puedes lanzar libremente un puntero a la `struct` a un puntero al primer elemento.

Esto significa que podemos crear una situación como la siguiente:

```
struct parent {
    int a, b;
};

struct child {
    struct parent super;  // DEBE ser el primero
    int c, d;
};
```

Entonces podemos pasar un puntero a una `struct hija` a una función que espera o bien eso *o* ¡un puntero a una `struct padre`!

Como `struct padre super` es el primer elemento de `struct hijo`, un puntero a cualquier `struct hijo` es lo mismo que un puntero a ese campo `super`[3].

Pongamos un ejemplo. Haremos `structs` como arriba, pero luego pasaremos un puntero a una `struct hija` a una función que necesita un puntero a una `struct padre`… y seguirá funcionando.

```
#include <stdio.h>

struct parent {
    int a, b;
};

struct child {
    struct parent super;  // DEBE ser el primero
    int c, d;
};

// Haciendo el argumento `void*` para que podamos pasarle cualquier tipo
// (es decir, un struct padre o struct hijo)
void print_parent(void *p)
{
    // Espera una estructura padre--pero una estructura hijo también funcionará
```

[3] `super` no es una palabra clave, por cierto. Sólo estoy robando terminología de programación orientada a objetos

```
    // porque el puntero apunta al struct padre en el primer
    // campo:
    struct parent *self = p;

    printf("Parent: %d, %d\n", self->a, self->b);
}

void print_child(struct child *self)
{
    printf("Child: %d, %d\n", self->c, self->d);
}

int main(void)
{
    struct child c = {.super.a=1, .super.b=2, .c=3, .d=4};

    print_child(&c);
    print_parent(&c);  // ¡También funciona aunque sea un struct hijo!
}
```

¿Ves lo que hemos hecho en la última línea de `main()`? Llamamos a `print_parent()` pero pasamos una `struct child*` como argumento. Aunque `print_parent()` necesita que el argumento apunte a una `struct padre`, nos estamos *saliendo con la nuestra* porque el primer campo de la `struct hija` es una `struct padre`.

De nuevo, esto funciona porque un puntero a una `struct` tiene el mismo valor que un puntero al primer campo de esa `struct`.

Todo depende de esta parte de la especificación:

> **§6.7.2.1¶15** […] Un puntero a un objeto estructura, convenientemente convertido, apunta a su miembro inicial […], y viceversa.

y

§§6.5¶7** Sólo se puede acceder al valor almacenado de un objeto mediante una expresión > expresión lvalue que tenga uno de los siguientes tipos: > > * un tipo compatible con el tipo efectivo del objeto > * […]

y mi suposición de que “convenientemente convertido” significa “moldeado al tipo efectivo del miembro inicial”.

## 20.8 Campos de bits

En mi experiencia, rara vez se utilizan, pero puede que los veas por ahí de vez en cuando, especialmente en aplicaciones de bajo nivel que empaquetan bits en espacios más grandes.

Echemos un vistazo a algo de código para demostrar un caso de uso:

```
#include <stdio.h>

struct foo {
    unsigned int a;
    unsigned int b;
    unsigned int c;
    unsigned int d;
```

```
};

int main(void)
{
    printf("%zu\n", sizeof(struct foo));
}
```

Para mí, esto imprime `16`. Lo cual tiene sentido, ya que `unsigned`s son 4 bytes en mi sistema.

Pero, ¿y si supiéramos que todos los valores que se van a almacenar en `a` y `b` se pueden almacenar en 5 bits, y los valores en `c`, y `d` se pueden almacenar en 3 bits? Eso es sólo un total de 16 bits. ¿Por qué tener 128 bits reservados para ellos si sólo vamos a usar 16?

Bueno, podemos decirle a C que por favor intente empaquetar estos valores. Podemos especificar el número máximo de bits que pueden tener los valores (desde 1 hasta el tamaño del tipo que los contiene).

Esto se hace poniendo dos puntos después del nombre del campo, seguido del ancho del campo en bits.

```
struct foo {
    unsigned int a:5;
    unsigned int b:5;
    unsigned int c:3;
    unsigned int d:3;
};
```

Ahora, cuando le pregunto a C cuánto mide mi `estructura foo`, ¡me dice 4! Eran 16 bytes, pero ahora son sólo 4. Ha "empaquetado" esos 4 valores en 4 bytes, lo que supone un ahorro de memoria cuatro veces mayor.

La contrapartida es, por supuesto, que los campos de 5 bits sólo pueden contener valores del 0 al 31 y los de 3 bits sólo pueden contener valores del 0 al 7. Pero la vida es así. Pero, al fin y al cabo, la vida es un compromiso.

### 20.8.1 Campos de bits no adyacentes

Un inconveniente: C sólo combinará campos de bits **adyacentes**. Si están interrumpidos por campos que no son de bits, no se ahorra nada:

```
struct foo { // sizeof(struct foo) == 16 (para mí)
    unsigned int a:1; // ya que a no es adyacente a c.
    unsigned int b;
    unsigned int c:1;
    unsigned int d;
};
```

En ese ejemplo, como "a" no es adyacente a "c", ambas están "empaquetadas" en sus propios "int".

Así que tenemos un `int` para `a`, `b`, `c` y `d`. Como mis `int`s son de 4 bytes, hay un total de 16 bytes.

Una rápida reorganización nos permite ahorrar espacio, de 16 a 12 bytes (en mi sistema):

```
struct foo {            // sizeof(struct foo) == 12 (para mí)
    unsigned int a:1;
    unsigned int c:1;
    unsigned int b;
```

```
    unsigned int d;
};
```

Y ahora, como `a` está junto a `c`, el compilador los junta en un único `int`.

Así que tenemos un `int` para `a` y `c` combinados, y un `int` para `b` y `d`. Para un total de 3 `int`s, o 12 bytes.

Pon todos tus campos de bits juntos para que el compilador los combine.

### 20.8.2 `int`s con signo o sin signo

Si simplemente declaras un campo de bits como `int`, los diferentes compiladores lo tratarán como `signed` o `unsigned`. Igual que ocurre con `char`.

Sea específico sobre el signo cuando utilice campos de bits.

### 20.8.3 Campos de bits sin nombre

En algunas circunstancias concretas, puede que necesites reservar algunos bits por razones de hardware, pero no necesites utilizarlos en código.

Por ejemplo, supongamos que tenemos un byte en el que los 2 bits superiores tienen un significado, el bit inferior tiene un significado, pero los 5 bits centrales no los usamos[4].

Podríamos hacer algo así:

```
struct foo {
    unsigned char a:2;
    unsigned char dummy:5;
    unsigned char b:1;
};
```

Y eso funciona–en nuestro código usamos `a` y `b`, pero nunca `dummy`. Sólo está ahí para consumir 5 bits y asegurarse de que “a” y “b” están en las posiciones “requeridas” (por este ejemplo artificial) dentro del byte.

C nos permite una forma de limpiar esto: campos de bits sin nombre. Puedes omitir el nombre (`dummy`) en este caso, y C está perfectamente satisfecho con el mismo efecto:

```
struct foo {
    unsigned char a:2;
    unsigned char :5;   // <--  campo de bits sin nombre
    unsigned char b:1;
};
```

### 20.8.4 Campos de bits sin nombre de ancho cero

Algo más de esoterismo por aquí… Digamos que estás empaquetando bits en un `unsigned int`, y necesitas algunos campos de bits adyacentes para empaquetarlos en el *siguiente* `unsigned int`.

Es decir, si haces esto:

```
struct foo {
    unsigned int a:1;
    unsigned int b:2;
```

[4]Suponiendo `charts` de 8 bits, es decir `CHAR_BIT == 8`.

```
    unsigned int c:3;
    unsigned int d:4;
};
```

el compilador los empaqueta todos en un único `unsigned int`. ¿Pero qué pasa si necesitas `a` y `b` en un `int`, y `c` y `d` en otro diferente?

Hay una solución para eso: poner un campo de bits sin nombre de ancho `0` donde quieras que el compilador empiece de nuevo a empaquetar bits en un `int` diferente:

```
struct foo {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int :0;   // <--Campo de bits sin nombre de ancho cero
    unsigned int c:3;
    unsigned int d:4;
};
```

Es análogo a un salto de página explícito en un procesador de textos. Le estás diciendo al compilador: "Deja de empaquetar bits en este `unsigned`, y empieza a empaquetarlos en el siguiente".

Añadiendo el campo de bits sin nombre de ancho cero en ese lugar, el compilador pone `a` y `b` en un `unsigned int`, y `c` y `d` en otro `unsigned int`. Dos en total, para un tamaño de 8 bytes en mi sistema (`unsigned int`s son 4 bytes cada uno).

## 20.9 Uniones (Unions)

Son básicamente como `structs`, excepto que los campos se solapan en memoria. La `union` sólo será lo suficientemente grande para el campo más grande, y sólo se puede utilizar un campo a la vez.

Es una forma de reutilizar el mismo espacio de memoria para distintos tipos de datos.

Los declaras como `structs`, excepto que es `union`. Echa un vistazo a esto:

```
union foo {
    int a, b, c, d, e, f;
    float g, h;
    char i, j, k, l;
};
```

Eso son muchos campos. Si esto fuera una `estructura`, mi sistema me diría que se necesitan 36 bytes para contenerlo todo.

Pero es una `union`, así que todos esos campos se solapan en el mismo espacio de memoria. El más grande es `int` (o `float`), que ocupa 4 bytes en mi sistema. Y, de hecho, si pregunto por el `sizeof` de la `unión foo`, ¡me dice 4!

El inconveniente es que sólo se puede utilizar uno de esos campos a la vez. Pero…

### 20.9.1 Unions y Tipo Punning

Se puede escribir de forma no portátil en un campo `union` y leer de otro.

Esto se llama type punning[5], y lo usarías si realmente supieras lo que estás haciendo, normalmente con algún tipo de programación de bajo nivel.

Dado que los miembros de una unión comparten la misma memoria, escribir en un miembro afecta necesariamente a los demás. Y si se lee de uno lo que se ha escrito en otro, se obtienen efectos extraños.

```
#include <stdio.h>

union foo {
    float b;
    short a;
};

int main(void)
{
    union foo x;

    x.b = 3.14159;

    printf("%f\n", x.b);  // 3.14159, bastante justo

    printf("%d\n", x.a);  // Pero, ¿y esto?
}
```

En mi sistema, esto se imprime:

```
3.141590
4048
```

porque bajo el capó, la representación del objeto para el float `3.14159` era la misma que la representación del objeto para el short `4048`. En mi sistema. Tus resultados pueden variar.

### 20.9.2 Punteros a `union`s

Si tienes un puntero a una `union`, puedes convertir ese puntero a cualquiera de los tipos de los campos de esa `union` y obtener los valores de esa forma.

En este ejemplo, vemos que la `union` tiene `int`s y `float`s en ella. Y obtenemos punteros a la `union`, pero los convertimos a los tipos `int*` y `float*` (la conversión silencia las advertencias del compilador). Y si los desreferenciamos, vemos que tienen los valores que almacenamos directamente en la "unión".

```
#include <stdio.h>

union foo {
    int a, b, c, d, e, f;
    float g, h;
    char i, j, k, l;
};

int main(void)
{
    union foo x;
```

[5] https://en.wikipedia.org/wiki/Type_punning

```
    int *foo_int_p = (int *)&x;
    float *foo_float_p = (float *)&x;

    x.a = 12;
    printf("%d\n", x.a);           // 12
    printf("%d\n", *foo_int_p);    // 12, nuevamente

    x.g = 3.141592;
    printf("%f\n", x.g);           // 3.141592
    printf("%f\n", *foo_float_p);  // 3.141592, nuevamente
}
```

Lo contrario también es cierto. Si tenemos un puntero a un tipo dentro de `union`, podemos convertirlo en un puntero a `union` y acceder a sus miembros.

```
union foo x;
int *foo_int_p = (int *)&x;             // Puntero a campo int
union foo *p = (union foo *)foo_int_p;  // Volver al puntero de la unión

p->a = 12;  // Esta línea es la misma que...
x.a = 12;   // este.
```

Todo esto sólo te permite saber que, bajo el capó, todos estos valores en un `union` comienzan en el mismo lugar en la memoria, y eso es lo mismo que donde todo el ‘union es.

### 20.9.3 Secuencias iniciales comunes en las uniones

Si tienes una `union` de `struct`s, y todas esas `struct`s empiezan con una *secuencia inicial común*, es válido acceder a miembros de esa secuencia desde cualquiera de los miembros de la `union`.

¿Cómo?

Aquí hay dos `struct`s con una secuencia inicial común:

```
struct a {
    int x;     //
    float y;   // Secuencia inicial común

    char *p;
};

struct b {
    int x;     //
    float y;   // Secuencia inicial común

    double *p;
    short z;
};
```

¿Lo ves? Es que empiezan con `int` seguido de `float`—esa es la secuencia inicial común. Los miembros en la secuencia de las `struct`s tienen que ser tipos compatibles. Y lo vemos con `x` y `y`, que son `int` y `float` respectivamente.

Ahora vamos a construir una unión de estos:

```
union foo {
    struct a sa;
    struct b sb;
};
```

Lo que nos dice esta regla es que tenemos garantizado que los miembros de las secuencias iniciales comunes son intercambiables en código. Es decir:

- `f.sa.x` es lo mismo que `f.sb.x`.

y

- `f.sa.y` es lo mismo que `f.sb.y`.

Porque los campos `x` e `y` están ambos en la secuencia inicial común.

Además, los nombres de los miembros de la secuencia inicial común no importan. todo lo que importa es que los tipos son los mismos.

En conjunto, esto nos permite añadir de forma segura alguna información compartida entre `struct`s en la `union`. El mejor ejemplo de esto es probablemente el uso de un campo para determinar el tipo de `struct` de todas las `struct`s en la `union` que está actualmente "en uso".

Es decir, si no se nos permitiera esto y pasáramos la `union` a alguna función, ¿cómo sabría esa función qué miembro de la `union` es el que debería mirar?

Echa un vistazo a estas `struct`s. Observa la secuencia inicial común:

```
#include <stdio.h>

struct common {
    int type;   // secuencia inicial común
};

struct antelope {
    int type;   // secuencia inicial común

    int loudness;
};

struct octopus {
    int type;   // secuencia inicial común

    int sea_creature;
    float intelligence;
};
```

Ahora vamos a meterlos en un `union`:

```
union animal {
    struct common common;
    struct antelope antelope;
    struct octopus octopus;
};
```

También, permítanme estos dos `#define`s para la demo:

```
#define ANTELOPE 1
#define OCTOPUS  2
```

Hasta ahora, aquí no ha pasado nada especial. Parece que el campo `type` es completamente inútil.

Pero ahora hagamos una función genérica que imprima un `union animal`. De alguna manera tiene que ser capaz de decir si está mirando un `struct antílope` o un `struct pulpo`.

Gracias a la magia de las secuencias iniciales comunes, puede buscar el tipo de animal en cualquiera de estos lugares para un `animal de unión x` en particular:

```
int type = x.common.type;    \\ or...
int type = x.antelope.type;  \\ or...
int type = x.octopus.type;
```

Todos ellos se refieren al mismo valor en memoria.

Y, como habrás adivinado, el `struct common` está ahí para que el código pueda mirar agnósticamente el tipo sin mencionar un animal en particular.

Veamos el código para imprimir un `union animal`:

```
void print_animal(union animal *x)
{
    switch (x->common.type) {
        case ANTELOPE:
            printf("Antelope: loudness=%d\n", x->antelope.loudness);
            break;

        case OCTOPUS:
            printf("Octopus : sea_creature=%d\n", x->octopus.sea_creature);
            printf("          intelligence=%f\n", x->octopus.intelligence);
            break;

        default:
            printf("Unknown animal type\n");
    }

}

int main(void)
{
    union animal a = {.antelope.type=ANTELOPE, .antelope.loudness=12};
    union animal b = {.octopus.type=OCTOPUS, .octopus.sea_creature=1,
                                        .octopus.intelligence=12.8};

    print_animal(&a);
    print_animal(&b);
}
```

Mira cómo en la línea 29 sólo estamos pasando la `union` –no tenemos ni idea de qué tipo de animal `struct` está en uso dentro de ella.

Pero no pasa nada. Porque en la línea 31 comprobamos el tipo para ver si es un antílope o un pulpo. Y entonces podemos mirar en la `struct` apropiada para obtener los miembros.

Definitivamente es posible conseguir este mismo efecto usando sólo `structs`, pero puedes hacerlo de esta manera si quieres los efectos de ahorro de memoria de una `union`.

## 20.10 Uniones y estructuras sin nombre

Usted sabe cómo puede tener un `struct` sin nombre, así:

```
struct {
    int x, y;
} s;
```

Eso define una variable `s` que es de tipo `struct` anónimo (porque la `struct` no tiene etiqueta de nombre), con los miembros `x` e `y`.

Así que cosas como esta son válidas:

```
s.x = 34;
s.y = 90;

printf("%d %d\n", s.x, s.y);
```

Resulta que puedes soltar esas `structs` sin nombre en `unions` tal y como cabría esperar:

```
union foo {
    struct {       // sin nombre!
        int x, y;
    } a;

    struct {       // sin nombre!
        int z, w;
    } b;
};
```

Y luego acceder a ellos con normalidad:

```
union foo f;

f.a.x = 1;
f.a.y = 2;
f.b.z = 3;
f.b.w = 4;
```

No hay problema.

## 20.11 Pasar y devolver `structs` y `unions`

Puede pasar una `struct` o `union` a una función por valor (en lugar de un puntero a la misma)—se hará una copia de ese objeto al parámetro como si fuera por asignación como de costumbre.

También puedes devolver una `struct` o `union` de una función y también se devuelve por valor.

```
#include <stdio.h>

struct foo {
    int x, y;
};

struct foo f(void)
{
    return (struct foo){.x=34, .y=90};
}

int main(void)
{
    struct foo a = f();  // Se realiza la copia

    printf("%d %d\n", a.x, a.y);
}
```

Dato curioso: si haces esto, puedes utilizar el operador `.` justo después de la llamada a la función:

```
    printf("%d %d\n", f().x, f().y);
```

(Por supuesto, ese ejemplo llama a la función dos veces, de forma ineficiente).

Y lo mismo vale para devolver punteros a `structs` y `unions`—sólo asegúrate de usar el operador de flecha `->` en ese caso.

# Chapter 21

# Caracteres y Strings II

Ya hemos hablado de cómo los tipos `char` son en realidad tipos de enteros pequeños… pero es lo mismo para un carácter entre comillas simples.

Pero una cadena entre comillas dobles es del tipo `const char *`.

Resulta que hay algunos tipos más de cadenas y caracteres, y esto nos lleva a uno de los agujeros de conejo más infames del lenguaje: todo el asunto multibyte/ancho/Unicode/localización.

Vamos a asomarnos a esa madriguera de conejo, pero sin entrar. …¡Todavía!

## 21.1 Secuencias de escape

Estamos acostumbrados a cadenas y caracteres con letras, signos de puntuación y números normales:

```
char *s = "Hello!";
char t = 'c';
```

Pero, ¿y si queremos introducir algún carácter especial que no podemos escribir con el teclado porque no existe (por ejemplo, “€”), o incluso si queremos un carácter que sea una comilla simple? Está claro que no podemos hacerlo:

```
char t = ''';
```

Para hacer estas cosas, utilizamos algo llamado *secuencias de escape*. Se trata del carácter barra invertida (`\`) seguido de otro carácter. Los dos (o más) caracteres juntos tienen un significado especial.

Para nuestro ejemplo de carácter de comilla simple, podemos poner un escape (es decir, `\`) delante de la comilla simple central para resolverlo:

```
char t = '\'';
```

Ahora C sabe que `\'` significa sólo una comilla normal que queremos imprimir, no el final de la secuencia de caracteres.

Puedes decir “barra invertida” o “escape” en este contexto (“escape esa comilla”) y los desarrolladores de C sabrán de qué estás hablando. Además, “escape” en este contexto es diferente de la tecla `Esc` o del código ASCII `ESC`.

### 21.1.1 Escapes de uso frecuente

En mi humilde opinión, estos caracteres de escape constituyen el 99,2%[1] de todos los escapes.

| Codigo | Descripción |
|---|---|
| `\n` | Carácter de nueva línea—cuando se imprime, continúa la salida subsiguiente en la línea siguiente |
| `\'` | Comilla simple: se utiliza para una constante de carácter de comilla simple. |
| `\"` | Comilla doble: se utiliza para una comilla doble en una cadena literal. |
| `\\` | Barra diagonal inversa—utilizada para un literal `\` en una cadena o carácter |

Estos son algunos ejemplos de los escapes y lo que muestran cuando se imprimen.

```
printf("Use \\n for newline\n");  // Usar \n para nueva línea
printf("Say \"hello\"!\n");       // Diga "hello"!
printf("%c\n", '\'');             // '
```

### 21.1.2 Escapes poco utilizados

Pero hay más escapes. Sólo que no se ven tan a menudo.

| Código | Description |
|---|---|
| `\a` | Alerta. Esto hace que el terminal emita un sonido o un destello, ¡o ambos! |
| `\b` | Retroceso. Desplaza el cursor un carácter hacia atrás. No borra el carácter. |
| `\f` | Alimentar formulario. Esto se mueve a la siguiente “página”, pero eso no tiene mucho significado moderno. En mi sistema, esto se comporta como `\v`. |
| `\r` | Volver. Desplazarse al principio de la misma línea. |
| `\t` | Tabulador horizontal. Se mueve al siguiente tabulador horizontal. En mi máquina, esto se alinea en columnas que son múltiplos de 8, pero YMMV. |
| `\v` | Tabulación vertical. Se mueve al siguiente tabulador vertical. En mi máquina, esto se mueve a la misma columna en la línea siguiente. |
| `\?` | Signo de interrogación literal. A veces es necesario para evitar los trígrafos, como se muestra a continuación. |

#### 21.1.2.1 Actualizaciones de estado de línea única

Un caso de uso para `\b` o `\r` es mostrar actualizaciones de estado que aparecen en la misma línea en la pantalla y no causan que la pantalla se desplace. Aquí hay un ejemplo que hace una cuenta atrás desde 10. (Si tu compilador no soporta threading, puedes usar la función POSIX no estándar `sleep()` de `<unistd.h>`—si no estás en un Unix-like, busca tu plataforma y `sleep` para el equivalente).

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

int main(void)
{
    for (int i = 10; i >= 0; i--) {
        printf("\rT minutos %d segundo%s... \b", i, i != 1? "s": "");
```

[1] me acabo de inventar esa cifra, pero probablemente no esté muy lejos

```
        fflush(stdout);  // Forzar la actualización de la salida

        // Sleep for 1 second
        thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
    }

    printf("\rLiftoff!             \n");
}
```

En la línea 7 ocurren varias cosas. En primer lugar, empezamos con un `\r` para llegar al principio de la línea actual, luego sobrescribimos lo que haya allí con la cuenta atrás actual. (Hay un operador ternario para asegurarnos de que imprimimos `1 segundo` en lugar de `1 segundos`).

Además, hay un espacio después de `...` Eso es para que sobrescribamos correctamente el último `.` cuando `i` baje de `10` a `9` y tengamos una columna más estrecha. Pruébalo sin el espacio para ver a qué me refiero.

Y lo envolvemos con un `\b` para retroceder sobre ese espacio para que el cursor se sitúe en el final exacto de la línea de una manera estéticamente agradable.

Observe que la línea 14 también tiene un montón de espacios al final para sobrescribir los caracteres que ya estaban allí desde la cuenta atrás.

Finalmente, tenemos un extraño `fflush(stdout)` ahí, sea lo que sea lo que signifique. La respuesta corta es que la mayoría de los terminales están *line buffered* por defecto, lo que significa que no muestran nada hasta que se encuentra un carácter de nueva línea. Dado que no tenemos una nueva línea (sólo tenemos `\r`), sin esta línea, el programa se quedaría ahí hasta `¡Liftoff!` y entonces imprimiría todo en un instante. `fflush()` anula este comportamiento y fuerza la salida a suceder *ahora*.

#### 21.1.2.2 La fuga de los signos de interrogación

¿Por qué molestarse con esto? Al fin y al cabo, esto funciona muy bien:

```
printf("Doesn't it?\n");
```

Y también funciona bien con el escape:

```
printf("Doesn't it\?\n");   // Note \?
```

Entonces, ¿qué sentido tiene?

Seamos más enfáticos con otro signo de interrogación y exclamación:

```
printf("Doesn't it??!\n");
```

Cuando compilo esto, recibo esta advertencia:

```
foo.c: In function ‘main’:
foo.c:5:23: warning: trigraph ??! converted to | [-Wtrigraphs]
    5 |     printf("Doesn't it??!\n");
      |
```

Y ejecutarlo da este resultado improbable:

```
Doesn't it|
```

¿Así que *trigrafías*? ¿Qué diablos es esto?

Estoy seguro de que volveremos sobre este rincón polvoriento del lenguaje más adelante, pero el resumen es que el compilador busca ciertas tripletas de caracteres que empiezan por `??` y las sustituye por otros caracteres. Así que si estás en un terminal antiguo sin el símbolo de la tubería (`|`) en el teclado, puedes escribir `??!` en su lugar.

Puedes arreglar esto escapando el segundo signo de interrogación, así:

```
printf("Doesn't it?\?!\n");
```

Y entonces se compila y funciona como se esperaba.

Hoy en día, por supuesto, nadie utiliza los trígrafos. Pero ese `??!` completo aparece a veces si decides usarlo en una cadena para darle énfasis.

### 21.1.3 Escapes numéricos

Además, hay formas de especificar constantes numéricas u otros valores de caracteres dentro de cadenas o constantes de caracteres.

Si conoce la representación octal o hexadecimal de un byte, puede incluirla en una cadena o constante de caracteres.

La siguiente tabla contiene números de ejemplo, pero se puede utilizar cualquier número hexadecimal u octal. Rellene con ceros a la izquierda si es necesario para leer el recuento de dígitos correcto.

| Código | Description |
|---|---|
| `\123` | Incrusta el byte con valor octal `123`, 3 dígitos exactamente. |
| `\x4D` | Incrusta el byte con valor hexadecimal `4D`, 2 dígitos. |
| `\u2620` | Incrusta el carácter Unicode en el punto de código con valor hexadecimal `2620`, 4 dígitos. |
| `\U0001243F` | Incrusta el carácter Unicode en el punto de código con valor hexadecimal `1243F`, 8 dígitos. |

He aquí un ejemplo de la notación octal, menos utilizada, para representar la letra “B” entre “A” y “C”. Normalmente esto se usaría para algún tipo de carácter especial no imprimible, pero tenemos otras formas de hacerlo, más abajo, y esto es sólo una demostración octal:

```
printf("A\102C\n");  // 102 es `B` en ASCII/UTF-8
```

Tenga en cuenta que no hay cero a la izquierda en el número octal cuando se incluye de esta manera. Pero tiene que tener tres caracteres, así que rellénalo con ceros a la izquierda si es necesario.

Pero mucho más común es utilizar constantes hexadecimales en estos días. Aquí tienes una demostración que no deberías usar, pero que muestra cómo incrustar los bytes UTF-8 0xE2, 0x80 y 0xA2 en una cadena, lo que corresponde al carácter Unicode “bullet” (-).

```
printf("\xE2\x80\xA2 Bullet 1\n");
printf("\xE2\x80\xA2 Bullet 2\n");
printf("\xE2\x80\xA2 Bullet 3\n");
```

Produce la siguiente salida si estás en una consola UTF-8 (o probablemente basura si no lo estás):

```
• Bullet 1
• Bullet 2
• Bullet 3
```

Pero esa es una forma deficiente de hacer Unicode. Puedes usar los escapes `\u` (16 bits) o `\U` (32 bits) para referirte a Unicode por el número de punto de código. La viñeta es `2022` (hexadecimal) en Unicode, así que puedes hacer esto y obtener resultados más portables:

```
printf("\u2022 Bullet 1\n");
printf("\u2022 Bullet 2\n");
printf("\u2022 Bullet 3\n");
```

Asegúrese de rellenar “u” con suficientes ceros a la izquierda para llegar a cuatro caracteres, y “U” con suficientes ceros a la izquierda para llegar a ocho.

Por ejemplo, esa viñeta podría hacerse con “U” y cuatro ceros a la izquierda:

```
printf("\U00002022 Bullet 1\n");
```

Pero, ¿quién tiene tiempo para ser tan verborreico?

# Chapter 22

# Tipos Enumerados: `enum`

C nos ofrece otra forma de tener valores enteros constantes por nombre: `enum`.

Por ejemplo:

```
enum {
  ONE=1,
  TWO=2
};

printf("%d %d", ONE, TWO);  // 1 2
```

En algunos aspectos, puede ser mejor –o diferente– que usar un `#define`. Diferencias clave:

- Los `enum`s sólo pueden ser tipos enteros.
- `#define` puede definir cualquier cosa.
- Los `enum`s se muestran a menudo por su nombre de identificador simbólico en un depurador.
- Los números definidos con `#define` se muestran como números brutos que son más difíciles de conocer mientras se depura.

Ya que son tipos enteros, pueden ser usados en cualquier lugar donde se puedan usar enteros, incluyendo en dimensiones de arreglos y sentencias `case`.

Vamos a profundizar en esto.

## 22.1 Comportamiento de `enum`

### 22.1.1 Numeración

Los `enum`s se numeran automáticamente a menos que los anules.

Empiezan en `0`, y se autoincrementan desde ahí, por defecto:

```
enum {
    SHEEP,  // El Valor es 0
    WHEAT,  // El Valor es 1
    WOOD,   // El Valor es 2
    BRICK,  // El Valor es 3
    ORE     // El Valor es 4
```

```
};

printf("%d %d\n", SHEEP, BRICK);  // 0 3
```

Puede forzar determinados valores enteros, como vimos anteriormente:

```
enum {
  X=2,
  Y=18,
  Z=-2
};
```

Los duplicados no son un problema:

```
enum {
  X=2,
  Y=2,
  Z=2
};
```

si se omiten los valores, la numeración continúa contando en sentido positivo a partir del último valor especificado. Por ejemplo:

```
enum {
  A,    // 0, valor inicial por defecto
  B,    // 1
  C=4,  // 4, ajustar manualmente
  D,    // 5
  E,    // 6
  F=3   // 3, ajustar manualmente
  G,    // 4
  H     // 5
}
```

### 22.1.2 Comas finales

Esto está perfectamente bien, si ese es tu estilo:

```
enum {
  X=2,
  Y=18,
  Z=-2,   // <-- Coma final
};
```

Se ha hecho más popular en los idiomas de las últimas décadas, así que puede que te alegre verlo.

### 22.1.3 Alcance

`enum`s scope como era de esperar. Si está en el ámbito del fichero, todo el fichero puede verlo. Si está en un bloque, es local a ese bloque.

Es muy común que los `enum`s se definan en ficheros de cabecera para que puedan ser `#include` en el ámbito del fichero.

### 22.1.4 Estilo

Como habrás notado, es común declarar los símbolos `enum` en mayúsculas (con guiones bajos).

Esto no es un requisito, pero es un modismo muy, muy común.

## 22.2 Su `enum` es un Tipo

Esto es algo importante que hay que saber sobre los `enum`: son un tipo, de forma análoga a como una `struct` es un tipo.

Puedes darles un nombre de etiqueta para poder referirte al tipo más tarde y declarar variables de ese tipo.

Ahora bien, dado que los `enum`s son tipos enteros, ¿por qué no usar simplemente `int`?

En C, la mejor razón para esto es la claridad del código–es una forma agradable y tipada de describir tu pensamiento en el código. C (a diferencia de C++) no obliga a que ningún valor esté dentro del rango de un `enum` en particular.

Hagamos un ejemplo donde declaramos una variable `r` de tipo `enum resource` que puede contener esos valores:

```
// Nombrado enum, el tipo es "enum resource"

enum resource {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
};

// Declarar una variable "r" de tipo "enum resource"

enum resource r = BRICK;

if (r == BRICK) {
    printf("I'll trade you a brick for two sheep.\n");
}
```

También puede `typedef` estos, por supuesto, aunque yo personalmente no me gusta hacerlo.

```
typedef enum {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} RESOURCE;

RESOURCE r = BRICK;
```

Otro atajo que es legal pero raro es declarar variables cuando declaras el `enum`:

```
// Declara un enum y algunas variables inicializadas de ese tipo:

enum {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} r = BRICK, s = WOOD;
```

También puedes dar un nombre al `enum` para poder utilizarlo más tarde, que es probablemente lo que quieres hacer en la mayoría de los casos:

```
// Declara un enum y algunas variables inicializadas de ese tipo:

enum resource {   // <-- es ahora "enum resource"
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} r = BRICK, s = WOOD;
```

En resumen, los `enum`s son una gran manera de escribir código limpio, tipado, de alcance y agradable.

# Chapter 23

# Punteros III: Punteros a punteros y más

Aquí es donde cubrimos algunos usos intermedios y avanzados de los punteros. Si no conoces bien los punteros, repasa los capítulos anteriores sobre punteros y aritmética de punteros antes de empezar con esto.

## 23.1 Punteros a punteros

Si puedes tener un puntero a una variable, y una variable puede ser un puntero, ¿puedes tener un puntero a una variable que a su vez sea un puntero?

Sí. Esto es un puntero a un puntero, y se mantiene en una variable de tipo puntero-puntero.

Antes de entrar en materia, quiero que te hagas una idea de cómo funcionan los punteros a punteros.

Recuerda que un puntero es sólo un número. Es un número que representa un índice en la memoria del ordenador, que normalmente contiene un valor que nos interesa.

Ese puntero, que es un número, tiene que ser almacenado en algún lugar. Y ese lugar es la memoria, como todo lo demás[1].

Pero como está almacenado en memoria, debe tener un índice en el que está almacenado, ¿no? El puntero debe tener un índice en la memoria donde se almacena. Y ese índice es un número. Es la dirección del puntero. Es un puntero al puntero.

Empecemos con un puntero regular a un `int`, de los capítulos anteriores:

```
#include <stdio.h>

int main(void)
{
    int x = 3490; // Tipo: int
    int *p = &x; // Tipo: puntero a un int

    printf("%d\n", *p);  // 3490
}
```

Bastante sencillo, ¿verdad? Tenemos dos tipos representados: `int` e `int*`, y configuramos `p` para que apunte a `x`. Entonces podemos desreferenciar `p` en la línea 8 e imprimir el valor `3490`.

[1] Hay un poco de diablo en los detalles con los valores que se almacenan sólo en los registros, pero podemos ignorar con seguridad que para nuestros propósitos aquí. Además, la especificación C no se pronuncia sobre estos "registros" más allá de la palabra clave `register`, cuya descripción no menciona los registros

Pero, como hemos dicho, podemos tener un puntero a cualquier variable… ¿eso significa que podemos tener un puntero a `p`?

En otras palabras, ¿de qué tipo es esta expresión?

```
int x = 3490; // Tipo: int
int *p = &x; // Tipo: puntero a un int

&p // <-- ¿De qué tipo es la dirección de p? ¿Un puntero a p?
```

Si `x` es un `int`, entonces `&x` es un puntero a un `int` que hemos almacenado en `p` que es de tipo `int*`. ¿Entiendes? (¡Repite este párrafo hasta que lo hagas!)

Y por tanto `&p` es un puntero a un `int*`, alias "puntero a un puntero a un `int`". También conocido como "`int`-pointer-pointer".

¿Lo ha entendido? (¡Repite el párrafo anterior hasta que lo entiendas!)

Escribimos este tipo con dos asteriscos: `int **`. Veámoslo en acción.

```
#include <stdio.h>

int main(void)
{
    int x = 3490; // Tipo: int
    int *p = &x; // Tipo: puntero a un int
    int **q = &p; // Tipo: puntero a puntero a int

    printf("%d %d\n", *p, **q);  // 3490 3490
}
```

Vamos a inventar algunas direcciones ficticias para los valores anteriores como ejemplos y ver lo que estas tres variables podrían parecer en la memoria. Los valores de dirección, a continuación son sólo inventados por mí para fines de ejemplo:

| Variable | Almacenada en la dirección | Valor almacenado allí |
|---|---|---|
| `x` | `28350` | `3490`—el valor del código |
| `p` | `29122` | `28350`—¡la dirección de `x`! |
| `q` | `30840` | `29122`—¡la dirección de `p`! |

De hecho, vamos a probarlo de verdad en mi ordenador[2] e imprimir los valores de los punteros con `%p` y volveré a hacer la misma tabla con las referencias reales (impresas en hexadecimal).

| Variable | Almacenada en la dirección | Valor almacenado allí |
|---|---|---|
| `x` | `0x7ffd96a07b94` | `3490`—el valor del código |
| `p` | `0x7ffd96a07b98` | `0x7ffd96a07b94`—la dirección de `x`! |
| `q` | `0x7ffd96a07ba0` | `0x7ffd96a07b98`—la dirección de `p`! |

Puedes ver que esas direcciones son las mismas excepto el último byte, así que céntrate en ellas.

[2] Es muy probable que obtengas números diferentes en el tuyo

En mi sistema, los `int`s son de 4 bytes, por lo que vemos que la dirección aumenta en 4 de `x` a `p`[3] y luego aumenta en 8 de `p` a `q`. En mi sistema, todos los punteros son de 8 bytes.

¿Importa si es un `int*` o un `int**`? ¿Es uno más bytes que el otro? No. Recuerda que todos los punteros son direcciones, es decir, índices de memoria. Y en mi máquina puedes representar un índice con 8 bytes… no importa lo que esté almacenado en ese índice.

Fíjate en lo que hicimos en la línea 9 del ejemplo anterior: hicimos *doble dereferencia* `q` para volver a nuestro `3490`.

Esta es la parte importante sobre punteros y punteros a punteros:

- Puedes obtener un puntero a cualquier cosa con `&` (¡incluyendo a un puntero!)
- Puedes obtener la cosa a la que apunta un puntero con `*` (¡incluyendo un puntero!)

Así que puedes pensar que `&` se usa para hacer punteros, y que `*` es la inversa–va en la dirección opuesta a `&`–para llegar a la cosa apuntada.

En términos de tipo, cada vez que `&`, se añade otro nivel de puntero al tipo.

| Si tiene | Entonces ejecuta | El tipo de resultado es |
|---|---|---|
| `int x` | `&x` | `int *` |
| `int *x` | `&x` | `int **` |
| `int **x` | `&x` | `int ***` |
| `int ***x` | `&x` | `int ****` |

Y cada vez que se utiliza la desreferencia (`*`), hace lo contrario:

| Si tiene | Entonces ejecuta | El tipo de resultado es |
|---|---|---|
| `int ****x` | `*x` | `int ***` |
| `int ***x` | `*x` | `int **` |
| `int **x` | `*x` | `int *` |
| `int *x` | `*x` | `int` |

Tenga en cuenta que puede utilizar varias `*`s seguidas para realizar una desreferencia rápida, como vimos en el código de ejemplo con `**q`, más arriba. Cada uno elimina un nivel de indirección.

| Si tiene | Entonces ejecuta | El tipo de resultado es |
|---|---|---|
| `int ****x` | `***x` | `int *` |
| `int ***x` | `**x` | `int *` |
| `int **x` | `**x` | `int` |

En general, `&*E == E`[4]. La desreferencia "deshace" la dirección-de.

Pero `&` no funciona de la misma manera—sólo puedes hacerlos de uno en uno, y tienes que almacenar el resultado en una variable intermedia:

```
int x = 3490; // Tipo: int
int *p = &x; // Tipo: int *, puntero a un int
```

[3] No hay absolutamente nada en la especificación que diga que esto funcionará siempre así, pero resulta que funciona así en mi sistema

[4] Incluso si `E` es `NULL`, resulta, extrañamente

```
int **q = &p; // Tipo: int **, puntero a puntero a int
int ***r = &q; // Tipo: int ***, puntero a puntero a puntero a int
int ****s = &r; // Tipo: int ****, se entiende la idea
int *****t = &s; // Tipo: int *****
```

### 23.1.1 Puntero Punteros y `const`.

Si recuerdas, declarando un puntero como este:

```
int *const p;
```

significa que no puedes modificar `p`. Intentar `p++` daría un error de compilación.

¿Pero cómo funciona eso con `int **` o `int ***`? ¿Dónde va `const` y qué significa?

Empecemos por lo más sencillo. El símbolo `const` que aparece junto al nombre de la variable se refiere a esa variable. Así que si quieres un `int***` que no puedas cambiar, puedes hacer esto:

```
int ***const p;

p++;  // No autorizado
```

Pero aquí es donde las cosas se ponen un poco raras.

¿Y si tuviéramos esta situación:

```
int main(void)
{
    int x = 3490;
    int *const p = &x;
    int **q = &p;
}
```

Cuando construyo eso, recibo una advertencia:

```
warning: initialization discards ‘const’ qualifier from pointer target type
    7 |     int **q = &p;
      |               ^
```

¿Qué es lo que ocurre? El compilador nos está diciendo aquí que teníamos una variable que era `const`, y estamos asignando su valor a otra variable que no es `const`. La “constancia” se descarta, que probablemente no es lo que queríamos hacer.

El tipo de `p` es `int *const p`, y `&p` es del tipo `int *const *`. E intentamos asignarlo a `q`.

¡Pero `q` es `int **`! ¡Un tipo con diferente `const`ness en el primer `*`! Así que recibimos un aviso de que el `const` en `int *const *` de `p` está siendo ignorado y desechado.

Podemos arreglarlo asegurándonos de que el tipo de `q` es al menos tan `const` como `p`.

```
int x = 3490;
int *const p = &x;
int *const *q = &p;
```

Y ahora estamos contentos.

Podríamos hacer `q` aún más `constante`. Tal como está, arriba, estamos diciendo, "`q` no es en sí `const`, pero la cosa a la que apunta es `const`". Pero podríamos hacer que ambos sean `const`:

```
int x = 3490;
int *const p = &x;
int *const *const q = &p;  // ¡Más const!
```

Y eso también funciona. Ahora no podemos modificar `q`, o el puntero `q` apunta a.

## 23.2 Valores multibyte

Ya lo hemos insinuado en varias ocasiones, pero está claro que no todos los valores pueden almacenarse en un solo byte de memoria. Las cosas ocupan varios bytes de memoria (suponiendo que no sean `chars`). Puedes saber cuántos bytes usando `sizeof`. Y puedes saber qué dirección de memoria es el *primer* byte del objeto usando el operador estándar `&`, que siempre devuelve la dirección del primer byte.

Y aquí tienes otro dato curioso. Si iteras sobre los bytes de cualquier objeto, obtienes su *representación de objeto*. Dos cosas con la misma representación de objeto en memoria son iguales.

Si quieres iterar sobre la representación del objeto, debes hacerlo con punteros a `unsigned char`.

Hagamos nuestra propia versión de `memcpy()`[5] que hace exactamente esto:

```
void *my_memcpy(void *dest, const void *src, size_t n)
{
    // Crear variables locales para src y dest, pero de tipo unsigned char

    const unsigned char *s = src;
    unsigned char *d = dest;

    while (n-- > 0) // Para el número de bytes dado
        *d++ = *s++; // Copia el byte origen al byte dest

    // La mayoría de las funciones de copia devuelven un puntero
    // al dest como conveniencia al que llama

    return dest;
}
```

(También hay algunos buenos ejemplos de post-incremento y post-decremento para que los estudies).

Es importante tener en cuenta que la versión anterior es probablemente menos eficiente que la que viene con su sistema.

Pero puedes pasarle punteros a cualquier cosa, y copiará esos objetos. Puede ser `int*`, `struct animal*`, o cualquier cosa.

Hagamos otro ejemplo que imprima los bytes de representación del objeto de una `struct` para que podamos ver si hay algún relleno ahí y qué valores tiene[6].

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

[5] https://beej.us guide/bgclr/html/split/stringref.html#man-memcpy

[6] Tu compilador de C no está obligado a añadir bytes de relleno, y los valores de cualquier byte de relleno que se añada son indeterminados

```
struct foo {
    char a;
    int b;
};

int main(void)
{
    struct foo x = {0x12, 0x12345678};
    unsigned char *p = (unsigned char *)&x;

    for (size_t i = 0; i < sizeof x; i++) {
        printf("%02X\n", p[i]);
    }
}
```

Lo que tenemos ahí es una `estructura foo` que está construida de tal manera que debería animar al compilador a inyectar bytes de relleno (aunque no tiene por qué). Y entonces obtenemos un `unsigned char *` en el primer byte de la variable `x` de la `struct foo`.

A partir de ahí, todo lo que necesitamos saber es el `sizeof x` y podemos hacer un bucle a través de esa cantidad de bytes, imprimiendo los valores (en hexadecimal para mayor facilidad).

Ejecutar esto da un montón de números como salida. He anotado a continuación para identificar donde se almacenan los valores:

```
12  | x.a == 0x12

AB  |
BF  | padding bytes with "random" value
26  |

78  |
56  | x.b == 0x12345678
34  |
12  |
```

En todos los sistemas, `sizeof(char)` es 1, y vemos que el primer byte en la parte superior de la salida contiene el valor `0x12` que almacenamos allí.

Luego tenemos algunos bytes de relleno–para mí, estos variaron de una ejecución a otra.

Finalmente, en mi sistema, `sizeof(int)` es 4, y podemos ver esos 4 bytes al final. Observa que son los mismos bytes que hay en el valor hexadecimal `0x12345678`, pero extrañamente en orden inverso[7].

Esto es un pequeño vistazo a los bytes de una entidad más compleja en memoria.

## 23.3 El puntero `NULL` y el cero

Estas cosas pueden usarse indistintamente:

- `NULL`
- `0`

[7] Esto variará dependiendo de la arquitectura, pero mi sistema es *little endian*, lo que significa que el byte menos significativo del número se almacena primero. Los sistemas *Big endian* tendrán el `12` primero y el `78` al final. Pero la especificación no dicta nada sobre esta representación

- `'\0'`
- `(void *)0`

Personalmente, siempre utilizo `NULL` cuando me refiero a `NULL`, pero puede que veas otras variantes de vez en cuando. Aunque `'\0'` (un byte con todos los bits a cero) también se compara igual, es *raro* compararlo con un puntero; deberías comparar `NULL` contra el puntero. (Por supuesto, muchas veces en el procesamiento de cadenas, estás comparando *la cosa a la que apunta el puntero* con `'\0`, y eso es correcto).

A `0` se le llama la *constante de puntero nulo*, y, cuando se compara o se asigna a otro puntero, se convierte en un puntero nulo del mismo tipo.

## 23.4 Punteros como enteros

Los punteros se pueden convertir en enteros y viceversa (ya que un puntero no es más que un índice de memoria), pero probablemente sólo sea necesario hacerlo si se realizan operaciones de hardware de bajo nivel. Los resultados de tales maquinaciones están definidos por la implementación, por lo que no son portables. Y pueden ocurrir *cosas raras*.

Sin embargo, C ofrece una garantía: puedes convertir un puntero a un tipo `uintptr_t` y podrás volver a convertirlo en puntero sin perder ningún dato.

`uintptr_t` está definido en `<stdint.h>`[8].

Además, si te apetece que te firmen, puedes usar `intptr_t` con el mismo efecto.

## 23.5 Asignación de punteros a otros punteros

Sólo hay una conversión de puntero segura:

1. Convertir a `intptr_t` o `uintptr_t`.
2. Convertir a y desde `void*`.

¡DOS! Dos conversiones de punteros seguras.

3. Conversión a y desde `char*` (o `signed char*`/`unsigned char*`).

¡TRES! ¡Tres conversiones seguras!

4. Convertir de y a un puntero a una `struct` y a un puntero a su primer miembro, y viceversa.

¡CUATRO! ¡Cuatro conversiones seguras!

Si convierte a un puntero de otro tipo y luego accede al objeto al que apunta, el comportamiento es indefinido debido a algo llamado *aliasing* estricto.

El viejo *aliasing* se refiere a la capacidad de tener más de una forma de acceder al mismo objeto. Los puntos de acceso son alias entre sí.

El *aliasing estricto* dice que sólo se permite acceder a un objeto a través de punteros a *tipos compatibles* con ese objeto.

Por ejemplo, esto está definitivamente permitido:

```
int a = 1;
int *p = &a;
```

`p` es un puntero a un `int`, y apunta a un tipo compatible–a saber `int`–así que estamos bien.

Pero lo siguiente no es bueno porque `int` y `float` no son tipos compatibles:

[8] Es una característica opcional, así que podría no estar ahí—pero probablemente sí

```
int a = 1;
float *p = (float *)&a;
```

Aquí hay un programa de demostración que hace algo de aliasing. Toma una variable `v` de tipo `int32_t` y la aliasea a un puntero a una `struct words`. Esa `struct` tiene dos `int16_t`s dentro. Estos tipos son incompatibles, por lo que estamos violando las reglas estrictas de aliasing. El compilador asumirá que estos dos punteros nunca apuntan al mismo objeto… pero nosotros estamos haciendo que lo hagan. Lo cual es malo por nuestra parte.

Veamos si podemos romper algo.

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

struct words {
    int16_t v[2];
};

void fun(int32_t *pv, struct words *pw)
{
    for (int i = 0; i < 5; i++) {
        (*pv)++;

        // Imprime el valor de 32 bits y los valores de 16 bits:

        printf("%x, %x-%x\n", *pv, pw->v[1], pw->v[0]);
    }
}

int main(void)
{
    int32_t v = 0x12345678;

    struct words *pw = (struct words *)&v;  // Viola el aliasing estricto

    fun(&v, pw);
}
```

¿Ves cómo paso los dos punteros incompatibles a `fun()`? Uno de los tipos es `int32_t*` y el otro es `struct words*`.

Pero ambos apuntan al mismo objeto: el valor de 32 bits inicializado a `0x12345678`.

Así que si miramos los campos de `struct words`, deberíamos ver las dos mitades de 16 bits de ese número. ¿Verdad?

Y en el bucle `fun()`, incrementamos el puntero al `int32_t`. Y ya está. Pero como la `struct` apunta a esa misma memoria, también debería actualizarse al mismo valor.

Así que ejecutémoslo y obtendremos esto, con el valor de 32 bits a la izquierda y las dos porciones de 16 bits a la derecha. Debería coincidir[9]:

[9] Estoy imprimiendo los valores de 16 bits invertidos porque estoy en una máquina little-endian y así es más fácil de leer

```
12345679, 1234-5679
1234567a, 1234-567a
1234567b, 1234-567b
1234567c, 1234-567c
1234567d, 1234-567d
```

y lo hace… HASTA MAÑANA

Probémoslo compilando GCC con `-O3` y `-fstrict-aliasing`:

```
12345679, 1234-5678
1234567a, 1234-5679
1234567b, 1234-567a
1234567c, 1234-567b
1234567d, 1234-567c
```

¡Están separados por uno! ¡Pero apuntan al mismo recuerdo! ¿Cómo es posible? Respuesta: es un comportamiento indefinido poner alias a la memoria de esa manera. Todo es posible, excepto que no en el buen sentido.

Si tu código viola las estrictas reglas de aliasing, que funcione o no depende de cómo alguien decida compilarlo. Y eso es un fastidio, ya que está fuera de tu control. A menos que seas una especie de deidad omnipotente.

Poco probable, lo siento.

GCC puede ser forzado a no usar las reglas de aliasing estricto con `-fno-strict-aliasing`. Compilar el programa de demostración, arriba, con `-O3` y esta bandera hace que la salida sea la esperada.

Por último, *type punning* es usar punteros de diferentes tipos para ver los mismos datos. Antes del aliasing estricto, este tipo de cosas era bastante común:

```
int a = 0x12345678;
short b = *((short *)&a);   // Viola el aliasing estricto
```

Si desea realizar puntuaciones (relativamente) seguras, consulte la sección sobre Uniones y puntuaciones.

## 23.6 Diferencias entre punteros

Como sabes de la sección sobre aritmética de punteros, puedes restar un puntero de otro [10] para obtener la diferencia entre ellos en número de elementos del array.

Ahora el *tipo de esa diferencia* es algo que depende de la implementación, por lo que podría variar de un sistema a otro.

Para ser más portable, puedes almacenar el resultado en una variable de tipo `ptrdiff_t` definida en `<stddef.h>`.

```
int cats[100];

int *f = cats + 20;
int *g = cats + 60;

ptrdiff_t d = g - f;  // la diferencia es de 40
```

[10] Suponiendo que apunten al mismo objeto array.

Y puede imprimirlo anteponiendo al especificador de formato entero `t`:

```
printf("%td\n", d);  // Imprimir decimal: 40
printf("%tX\n", d);  // Imprimir hex:     28
```

## 23.7 Punteros a funciones

Las funciones son sólo colecciones de instrucciones de la máquina en la memoria, así que no hay ninguna razón por la que no podamos obtener un puntero a la primera instrucción de la función.

Y luego llamarla.

Esto puede ser útil para pasar, un puntero a una función a otra función como argumento. Entonces la segunda podría llamar a lo que se haya pasado.

La parte complicada de esto, sin embargo, es que C necesita saber el tipo de la variable que es el puntero a la función.

Y realmente le gustaría conocer todos los detalles.

Como “esto es un puntero a una función que toma dos argumentos `int` y devuelve `void`”.

¿Cómo se escribe todo eso para poder declarar una variable?

Bueno, resulta que se parece mucho a un prototipo de función, excepto que con algunos paréntesis extra:

```
// Declara que p es un puntero a una función.
// Esta función devuelve un float, y toma dos ints como argumentos.

float (*p)(int, int);
```

Fíjate también en que no tienes que dar nombres a los parámetros. Pero puedes hacerlo si quieres; simplemente se ignoran.

```
// Declara que p es un puntero a una función.
// Esta función devuelve un float, y toma dos ints como argumentos.

float (*p)(int a, int b);
```

Ahora que sabemos cómo declarar una variable, ¿cómo sabemos qué asignarle? ¿Cómo obtenemos la dirección de una función?

Resulta que hay un atajo, igual que para obtener un puntero a un array: puedes referirte al nombre de la función sin paréntesis. (Puedes poner un `&` delante si quieres, pero es innecesario y no es idiomático).

Una vez que tienes un puntero a una función, puedes llamarla simplemente añadiendo paréntesis y una lista de argumentos.

Hagamos un ejemplo simple donde efectivamente hago un alias para una función estableciendo un puntero a ella. Luego la llamaremos.

Este código imprime `3490`:

```
#include <stdio.h>

void print_int(int n)
{
```

```
    printf("%d\n", n);
}

int main(void)
{
    // Asigna p a print_int:

    void (*p)(int) = print_int;

    p(3490);          // Llamar a print_int a través del puntero
}
```

Observa cómo el tipo de `p` representa el valor de retorno y los tipos de parámetros de `print_int`. Tiene que ser así, o C se quejará de incompatibilidad de tipos de punteros.

Otro ejemplo muestra cómo podemos pasar un puntero a una función como argumento de otra función.

Escribiremos una función que toma un par de argumentos enteros, más un puntero a una función que opera sobre esos dos argumentos. Luego imprime el resultado.

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int mult(int a, int b)
{
    return a * b;
}

void print_math(int (*op)(int, int), int x, int y)
{
    int result = op(x, y);

    printf("%d\n", result);
}

int main(void)
{
    print_math(add, 5, 7);   // 12
    print_math(mult, 5, 7);  // 35
}
```

Tómate un momento para asimilarlo. La idea aquí es que vamos a pasar un puntero a una función a `print_math()`, y va a llamar a esa función para hacer algo de matemáticas.

De esta forma podemos cambiar el comportamiento de `print_math()` pasándole otra función. Puedes ver que lo hacemos en las líneas 22-23 cuando pasamos punteros a las funciones `add` y `mult`, respectivamente.

Ahora, en la línea 13, creo que todos estamos de acuerdo en que la firma de la función `print_math()` es un espectáculo para la vista. Y, si puedes creerlo, ésta es en realidad bastante sencilla comparada con algunas cosas que puedes construir[11].

[11] El Lenguaje de Programación Go inspiró su sintaxis de declaración de tipos en lo contrario de lo que hace C.

Pero vamos a digerirlo. Resulta que sólo hay tres parámetros, pero son un poco difíciles de ver:

```
//                      op             x      y
//              |-----------------|  |---|  |---|
void print_math(int (*op)(int, int), int x, int y)
```

El primero, `op`, es un puntero a una función que toma dos `int` como argumentos y devuelve un `int`. Esto coincide con las firmas de `add()` y `mult()`.

El segundo y el tercero, `x` e `y`, son parámetros `int` estándar.

Deja que tus ojos recorran la firma lenta y deliberadamente mientras identificas las partes que funcionan. Una cosa que siempre me llama la atención es la secuencia `(*op)(`, los paréntesis y el asterisco. Eso te delata que es un puntero a una función.

Por último, vuelve al capítulo *Pointers II* para ver un puntero a función ejemplo usando la función incorporada `qsort()`.

# Chapter 24

# Operaciones bit a bit

Estas operaciones numéricas permiten manipular bits individuales de las variables, lo que encaja con el hecho de que C sea un lenguaje de bajo nivel[1].

Si no estás familiarizado con las operaciones bit a bit, Wikipedia tiene un buen artículo sobre operaciones bit a bit[2].

## 24.1 AND, OR, XOR y NOT por bits

Para cada uno de ellos, las conversiones aritméticas habituales tienen lugar en los operandos (que en este caso deben ser de tipo entero), y luego se realiza la operación bitwise apropiada.

| Operación | Operador | Ejemplo |
|---|---|---|
| AND | `&` | `a = b & c` |
| OR | `\|` | `a = b \| c` |
| XOR | `^` | `a = b ^ c` |
| NOT | `~` | `a = ~c` |

Observe que son similares a los operadores booleanos `&&` y `||`.

Éstos tienen variantes abreviadas de asignación similares a `+=` y `-=`:

| Operador | Ejemplo | Equivalente taquigráfico |
|---|---|---|
| `&=` | `a &= c` | `a = a & c` |
| `\|=` | `a \|= c` | `a = a \| c` |
| `^=` | `a ^= c` | `a = a ^ c` |

## 24.2 Desplazamiento (Bitwise)

Para ellos, las promociones de enteros se realizan en cada operando (que debe ser de tipo entero) y luego se ejecuta un desplazamiento a nivel de bits. El tipo del resultado es el tipo del operando izquierdo promocionado.

[1] No es que otros lenguajes no lo hagan… lo hacen. Es interesante ver cómo muchos lenguajes modernos utilizan los mismos operadores para bitwise que C

[2] https://en.wikipedia.org/wiki/Bitwise_operation

Los nuevos bits se rellenan con ceros, con una posible excepción indicada en el comportamiento definido por la implementación, a continuación.

| Operación | Operador | Ejemplo |
|---|---|---|
| Desplazamiento a la izquierda | `<<` | `a = b << c` |
| Desplazamiento a la derecha | `>>` | `a = b >> c` |

También existe la misma taquigrafía similar para el desplazamiento:

| Operador | Ejemplo | Equivalente a mano larga |
|---|---|---|
| `>>=` | `a >>= c` | `a = a >> c` |
| `<<=` | `a <<= c` | `a = a << c` |

Tenga cuidado con el comportamiento indefinido: no se permiten desplazamientos negativos ni desplazamientos mayores que el tamaño del operando izquierdo promocionado.

También hay que tener cuidado con el comportamiento definido por la implementación: si se desplaza a la derecha un número negativo, los resultados están definidos por la implementación. (Es perfectamente correcto desplazar a la derecha un `int` con signo, sólo asegúrese de que es positivo).

# Chapter 25

# Funciones variádicas

Variadic_ es una palabra elegante para referirse a las funciones que toman un número arbitrario de argumentos.

Por ejemplo, una función normal toma un número determinado de argumentos:

```
int add(int x, int y)
{
    return x + y;
}
```

Sólo se puede llamar con exactamente dos argumentos que corresponden a los parámetros `x` e `y`.

```
add(2, 3);
add(5, 12);
```

Pero si lo intentas con más, el compilador no te dejará:

```
add(2, 3, 4);  // ERROR
add(5);        // ERROR
```

Las funciones variádicas sortean esta limitación hasta cierto punto.

Ya hemos visto un ejemplo famoso en `printf()`. Puedes pasarle todo tipo de cosas.

```
printf("Hello, world!\n");
printf("The number is %d\n", 2);
printf("The number is %d and pi is %f\n", 2, 3.14159);
```

Parece no importarle cuántos argumentos le des.

Bueno, eso no es del todo cierto. Cero argumentos le dará un error:

```
printf();  // ERROR
```

Esto nos lleva a una de las limitaciones de las funciones variádicas en C: deben tener al menos un argumento.

Pero aparte de eso, son bastante flexibles, incluso permiten que los argumentos tengan diferentes tipos como hace `printf()`.

¡Veamos cómo funcionan!

## 25.1 Elipses en firmas de funciones

¿Cómo funciona, sintácticamente?

Lo que haces es poner todos los argumentos que *deben* pasarse primero (y recuerda que tiene que haber al menos uno) y después de eso, pones `...` .Así:

```
void func(int a, ...)   // Literalmente 3 puntos aquí
```

Aquí hay algo de código para demostrarlo:

```
#include <stdio.h>

void func(int a, ...)
{
    printf("a is %d\n", a);  // Imprime "a es 2"
}

int main(void)
{
    func(2, 3, 4, 5, 6);
}
```

Así que, genial, podemos obtener ese primer argumento que está en la variable `a`, pero ¿qué pasa con el resto de argumentos? ¿Cómo se llega a ellos?

Aquí empieza la diversión.

## 25.2 Obtener los argumentos adicionales

Tendrás que incluir `<stdarg.h>` para que todo esto funcione.

Lo primero es lo primero, vamos a utilizar una variable especial de tipo `va_list` (lista de argumentos de variables) para llevar la cuenta de a qué variable estamos accediendo en cada momento.

La idea es que primero comencemos a procesar los argumentos con una llamada a `va_start()`, procesemos cada argumento a su vez con `va_arg()`, y luego, cuando hayamos terminado, lo cerremos con `va_end()`.

Cuando llame a `va_start()`, necesita pasar el *último parámetro con nombre* (el que está justo antes de `...`) para que sepa dónde empezar a buscar los argumentos adicionales.

Y cuando llame a `va_arg()` para obtener el siguiente argumento, tiene que decirle el tipo de argumento que debe obtener a continuación.

Aquí tienes una demo que suma un número arbitrario de enteros. El primer argumento es el número de enteros a sumar. Lo usaremos para calcular cuántas veces tenemos que llamar a `va_arg()`.

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

int add(int count, ...)
{
    int total = 0;
```

```
    va_list va;

    va_start(va, count);   // Empezar con argumentos después de "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Obtener el siguiente int

        total += n;
    }

    va_end(va);  // Todo hecho

    return total;
}

int main(void)
{
    printf("%d\n", add(4, 6, 2, -4, 17));  // 6 + 2 - 4 + 17 = 21
    printf("%d\n", add(2, 22, 44));        // 22 + 44 = 66
}
```

(Tenga en cuenta que cuando se llama a `printf()`, utiliza el número de `%d`s (o lo que sea) en la cadena de formato para saber cuántos argumentos más hay).

Si la sintaxis de `va_arg()` te parece extraña (debido a ese nombre de tipo suelto flotando por ahí), no eres el único. Esto se implementa con macros de preprocesador para conseguir toda la magia apropiada.

## 25.3 Funcionalidad de `va_list`

¿Qué es esa variable `va_list` que estamos usando ahí arriba? Es una variable opaca [1] que contiene información sobre qué argumento vamos a obtener a continuación con `va_arg()`. ¿Ves cómo llamamos a `va_arg()` una y otra vez? La variable `va_list` es un marcador de posición que mantiene un registro del progreso hasta el momento.

Pero tenemos que inicializar esa variable con algún valor razonable. Ahí es donde `va_start()` entra en juego.

Cuando llamamos a `va_start(va, count)`, arriba, estábamos diciendo: "Inicializa la variable `va` para que apunte al argumento variable *inmediatamente después* de `count`".

Y esa es *la razón* por la que necesitamos tener al menos una variable con nombre en nuestra lista de argumentos[2].

Una vez que tengas ese puntero al parámetro inicial, puedes obtener fácilmente los valores de los argumentos posteriores llamando repetidamente a `va_arg()`. Cuando lo hagas, tienes que pasarle tu variable `va_list` (para que pueda seguirte la pista), así como el tipo de argumento que vas a copiar.

Depende de ti como programador averiguar qué tipo vas a pasar a `va_arg()`. En el ejemplo anterior, acabamos de hacer `int`s. Pero en el caso de `printf()`, utiliza el especificador de formato para determinar qué tipo sacar a continuación.

[1] Es decir, se supone que nosotros, los desarrolladores de pacotilla, no sabemos lo que hay ahí ni lo que significa. La especificación no dicta qué es en detalle

[2] Sinceramente, sería posible eliminar esa limitación del lenguaje, pero la idea es que las macros `va_start()`, `va_arg()`, y `va_end()` se puedan escribir en C. Y para que eso ocurra, necesitamos alguna forma de inicializar un puntero a la ubicación del primer parámetro. Y para ello, necesitamos el *nombre* del primer parámetro. Se necesitaría una extensión del lenguaje para hacer esto posible, y hasta ahora el comité no ha encontrado una razón para hacerlo

Y cuando hayas terminado, llama a `va_end()` para terminar. **Debes** (según la especificación) llamar a esto en una variable `va_list` en particular antes de decidir llamar a `va_start()` o `va_copy()` de nuevo. Sé que aún no hemos hablado de `va_copy()`.

- `va_start()` para inicializar tu variable `va_list`
- Repetidamente `va_arg()` para obtener los valores
- `va_end()` para desinicializar la variable `va_list`

También mencioné `va_copy()` ahí arriba; hace una copia de tu variable `va_list` exactamente en el mismo estado. Es decir, si no has empezado con `va_arg()` con la variable fuente, la nueva tampoco se iniciará. Si has consumido 5 variables con `va_arg()` hasta ahora, la copia también lo reflejará.

va_copy()‘ puede ser útil si necesita recorrer los argumentos pero también necesita recordar su posición actual.

## 25.4 Funciones de biblioteca que utilizan `va_lists`

Uno de los otros usos de estos es bastante bueno: escribir tu propia variante personalizada de `printf()`. Sería un fastidio tener que manejar todos esos especificadores de formato, ¿verdad? ¿Los millones de ellos?

Por suerte, hay variantes de `printf()` que aceptan una `va_list` como argumento. Puedes usarlas para crear tus propios `printf()` personalizados.

Estas funciones empiezan por la letra `v`, como `vprintf()`, `vfprintf()`, `vsprintf()` y `vsnprintf()`. Básicamente, todas las funciones `printf()` de toda la vida, pero con una `v` delante.

Hagamos una función `my_printf()` que funcione igual que `printf()` excepto que toma un argumento extra delante.

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

int my_printf(int serial, const char *format, ...)
{
    va_list va;

    // Haz mi trabajo a medida
    printf("The serial number is: %d\n", serial);

    // Luego pasa el resto a vprintf()
    va_start(va, format);
    int rv = vprintf(format, va);
    va_end(va);

    return rv;
}

int main(void)
{
    int x = 10;
    float y = 3.2;

    my_printf(3490, "x is %d, y is %f\n", x, y);
}
```

¿Ves lo que hemos hecho? En las líneas 12-14 iniciamos una nueva variable `va_list`, y luego la pasamos directamente a `vprintf()`. Y sabe lo que tiene que hacer con ella, porque tiene toda la inteligencia de `printf()` incorporada.

Sin embargo, aún tenemos que llamar a `va_end()` cuando hayamos terminado, ¡así que no lo olvides!

# Chapter 26

# Configuración regional e internacionalización

*La localización* es el proceso de preparar tu aplicación para que funcione bien en distintas localizaciones (o países).

Como sabrás, no todo el mundo utiliza el mismo carácter para los decimales o para los separadores de miles… o para la moneda.

Estas localizaciones tienen nombres, y puedes seleccionar una para usarla. Por ejemplo, una configuración regional de EE.UU. podría escribir un número como:

100,000.00

Mientras que en Brasil, lo mismo podría escribirse con las comas y los puntos decimales intercambiados:

100.000,00

Así es más fácil escribir el código para que se adapte fácilmente a otras nacionalidades.

Bueno, más o menos. Resulta que C sólo tiene una configuración regional incorporada, y es limitada. La especificación realmente deja mucha ambigüedad aquí; es difícil ser completamente portable.

Pero haremos lo que podamos.

## 26.1 Configuración rápida de la localización

Para estas llamadas, incluya `<locale.h>`.

Hay básicamente una cosa que puedes hacer de forma portable aquí en términos de declarar una localización específica. Esto es probablemente lo que quieres hacer si vas a hacer algo de localización:

```
setlocale(LC_ALL, "");  // Utiliza la configuración regional
                        // de este entorno para todo
```

Usted querrá llamar a eso para que el programa se inicialice con su configuración regional actual.

Entrando en más detalles, hay una cosa más que puedes hacer y seguir siendo portable:

```
setlocale(LC_ALL, "C");  // Utilizar la configuración regional C por defecto.
```

pero se ejecuta por defecto cada vez que se inicia el programa, por lo que no es necesario que lo hagas tú mismo.

En la segunda cadena, puedes especificar cualquier configuración regional soportada por tu sistema. Esto depende completamente del sistema, así que variará. En mi sistema, puedo especificar esto:

```
setlocale(LC_ALL, "en_US.UTF-8");  // ¡No portátil!
```

Y funcionará. Pero sólo es portable a sistemas que tengan exactamente el mismo nombre para la misma localización, y no puedes garantizarlo.

Al pasar una cadena vacía (`""`) como segundo argumento, le estás diciendo a C: "Oye, averigua cuál es la configuración regional actual en este sistema para que yo no tenga que decírtelo".

## 26.2 Obtener la configuración regional monetaria

Porque mover papelitos verdes promete ser la clave de la felicidad[1], hablemos de la localización monetaria. Cuando escribes código portable, tienes que saber qué escribir por dinero, ¿verdad? Ya sea "$", "€", "¥", o "£".

¿Cómo puedes escribir ese código sin volverte loco? Por suerte, una vez que llames a `setlocale(LC_ALL, "")`, puedes buscarlas con una llamada a `localeconv()`:

```
struct lconv *x = localeconv();
```

Esta función devuelve un puntero a una `struct lconv` estáticamente asignada que contiene toda la información que estás buscando.

Estos son los campos de `struct lconv` y sus significados.

Primero, algunas convenciones. Un `_p_` significa "positivo", y `_n_` significa "negativo", y `int_` significa "internacional". Aunque muchos de ellos son del tipo `char` o `char*`, la mayoría (o las cadenas a las que apuntan) se tratan en realidad como enteros[2].

Antes de continuar, debes saber que `CHAR_MAX` (de `<limits.h>`) es el valor máximo que puede contener un `char`. Y que muchos de los siguientes valores `char` lo usan para indicar que el valor no está disponible en la localización dada.

| Campo | Descripción |
|---|---|
| `char *mon_decimal_point` | Carácter puntero decimal para dinero, por ejemplo `"."`. |
| `char *mon_thousands_sep` | Carácter separador de miles para dinero, por ejemplo `","`. |
| `char *mon_grouping` | Descripción de la agrupación por dinero (véase más abajo). |
| `char *positive_sign` | Signo positivo para el dinero, por ejemplo `"+"` o `""`. |
| `char *negative_sign` | Signo negativo para el dinero, por ejemplo `"-"`. |
| `char *currency_symbol` | Símbolo de moneda, por ejemplo `"$"`. |
| `char frac_digits` | Al imprimir importes monetarios, cuántos dígitos imprimir después del punto decimal, por ejemplo `2`. |
| `char p_cs_precedes` | `1` si el `símbolo_moneda` viene antes del valor de una cantidad monetaria no negativa, `0` si viene después. |
| `char n_cs_precedes` | `1` si el `símbolo_moneda` viene antes del valor para una cantidad monetaria negativa, `0` si viene después. |

[1] "Este planeta tiene -o más bien tenía- un problema: la mayoría de las personas que viven en él son infelices durante casi todo el tiempo. Se sugirieron muchas soluciones para este problema, pero la mayoría de ellas tenían que ver con el movimiento de pequeños trozos de papel verde, lo cual era extraño porque, en general, no eran los pequeños trozos de papel verde los que eran infelices." —La Guía del Autoestopista Galáctico, Douglas Adams

[2] Recuerda que `char` es sólo un entero del tamaño de un byte

| Campo | Descripción |
|---|---|
| `char p_sep_by_space` | Determina la separación del `símbolo de moneda` del valor para importes no negativos (véase más abajo). |
| `char n_sep_by_space` | Determina la separación del `símbolo de moneda` del valor para los importes negativos (véase más abajo). |
| `char p_sign_posn` | Determina la posición de `positive_sign` para valores no negativos. |
| `char p_sign_posn` | Determina la posición de `positive_sign` para valores negativos. |
| `char *int_curr_symbol` | Símbolo de moneda internacional, por ejemplo `"USD"`. |
| `char int_frac_digits` | Valor internacional para `frac_digits`. |
| `char int_p_cs_precedes` | Valor internacional para `p_cs_precedes`. |
| `char int_n_cs_precedes` | Valor internacional para `n_cs_precedes`. |
| `char int_p_sep_by_space` | Valor internacional para `p_sep_by_space`. |
| `char int_n_sep_by_space` | Valor internacional para `n_sep_by_space`. |
| `char int_p_sign_posn` | Valor internacional para `p_sign_posn`. |
| `char int_n_sign_posn` | Valor internacional para `n_sign_posn`. |

### 26.2.1 Agrupación de dígitos monetarios

Vale, esto es un poco raro. `mon_grouping` es un `char*`, así que podrías pensar que es una cadena. Pero en este caso, no, en realidad es un array de `char`s. Siempre debe terminar en `0` o `CHAR_MAX`.

Estos valores describen cómo agrupar conjuntos de números en moneda a la *izquierda* del decimal (la parte del número entero).

Por ejemplo, podríamos tener:

```
  2   1   0
 --- --- ---
$100,000,000.00
```

Se trata de grupos de tres. El grupo 0 (justo a la izquierda del decimal) tiene 3 dígitos. El grupo 1 (el siguiente a la izquierda) tiene 3 dígitos, y el último también tiene 3.

Así que podríamos describir estos grupos, de la derecha (el decimal) a la izquierda con un montón de valores enteros que representan los tamaños de los grupos:

```
3 3 3
```

Y eso funcionaría para valores de hasta 100.000.000 de dólares.

Pero ¿y si tuviéramos más? Podríamos seguir añadiendo `3`s …

```
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

pero eso es una locura. Por suerte, podemos especificar `0` para indicar que se repite el tamaño de grupo anterior:

```
3 0
```

Lo que significa repetir cada 3. Eso manejaría $100, $1,000, $10,000, $10,000,000, $100,000,000,000, y así sucesivamente.

Usted puede ir legítimamente loco con estos para indicar algunas agrupaciones extrañas.

Por ejemplo:

```
4 3 2 1 0
```

indicaría:

```
$1,0,0,0,0,00,000,0000.00
```

Otro valor que puede aparecer es `CHAR_MAX`. Indica que no se debe agrupar más, y puede aparecer en cualquier parte de la matriz, incluido el primer valor.

```
3 2 CHAR_MAX
```

indicaría:

```
100000000,00,000.00
```

por ejemplo.

Y el simple hecho de tener `CHAR_MAX` en la primera posición del array te indicaría que no iba a haber ningún tipo de agrupación.

### 26.2.2 Separadores y posición del cartel

Todas las variantes de `sep_by_space` se ocupan del espaciado alrededor del signo monetario. Los valores válidos son:

| Valor | Descripción |
|---|---|
| `0` | No hay espacio entre el símbolo de la moneda y el valor. |
| `1` | Separe el símbolo de moneda (y el signo, si existe) del valor con un espacio. |
| `2` | Separe el símbolo de signo del símbolo de moneda (si es adyacente) con un espacio; de lo contrario, separe el símbolo de signo del valor con un espacio. |

Las variantes de `sign_posn` vienen determinadas por los siguientes valores:

| Valor | Descripción |
|---|---|
| `0` | Pon paréntesis alrededor del valor y del símbolo monetario. |
| `1` | Coloque el signo delante del símbolo monetario y del valor. |
| `2` | Poner el signo después del símbolo monetario y del valor. |
| `3` | Poner el signo directamente delante del símbolo de moneda |
| `4` | Coloque el signo directamente detrás del símbolo de moneda. |

### 26.2.3 Ejemplos de valores

Cuando obtengo los valores en mi sistema, esto es lo que veo (cadena de agrupación mostrada como valores de bytes individuales):

```
mon_decimal_point  = "."
mon_thousands_sep  = ","
mon_grouping       = 3 3 0
positive_sign      = ""
negative_sign      = "-"
```

```
currency_symbol    = "$"
frac_digits        = 2
p_cs_precedes      = 1
n_cs_precedes      = 1
p_sep_by_space     = 0
n_sep_by_space     = 0
p_sign_posn        = 1
n_sign_posn        = 1
int_curr_symbol    = "USD "
int_frac_digits    = 2
int_p_cs_precedes  = 1
int_n_cs_precedes  = 1
int_p_sep_by_space = 1
int_n_sep_by_space = 1
int_p_sign_posn    = 1
int_n_sign_posn    = 1
```

## 26.3 Especificidades de localización

Observe cómo pasamos la macro `LC_ALL` a `setlocale()` anteriormente… esto indica que podría haber alguna variante que le permita ser más preciso sobre qué *partes* de la configuración regional está configurando.

Echemos un vistazo a los valores que puede ver para estos:

| Macro | Descripción |
|---|---|
| `LC_ALL` | Establece todo lo siguiente a la configuración regional dada. |
| `LC_COLLATE` | Controla el comportamiento de las funciones `strcoll()` y `strxfrm()`. |
| `LC_CTYPE` | Controla el comportamiento de las funciones de tratamiento de caracteres[3].. |
| `LC_MONETARY` | Controla los valores devueltos por `localeconv()`. |
| `LC_NUMERIC` | Controla el punto decimal para la familia de funciones `printf()`. |
| `LC_TIME` | Controla el formato de hora de las funciones de impresión de fecha y hora `strftime()` y `wcsftime()`. |

Es bastante común ver `LC_ALL`, pero, oye, al menos tienes opciones.

También debo señalar que `LC_CTYPE` es una de las más importantes porque se relaciona con los caracteres anchos, una importante caja de Pandora de la que hablaremos más adelante.

[3]Excepto `isdigit()` e `isxdigit()`.

# Chapter 27

# Unicode, caracteres anchos y todo eso

Antes de empezar, ten en cuenta que esta es un área activa del desarrollo del lenguaje C, ya que trabaja para superar algunos, erm, *dolores de crecimiento*. Cuando salga C2x, es probable que haya actualizaciones.

La mayoría de la gente está básicamente interesada en la engañosamente simple pregunta: "¿Cómo uso tal y tal juego de caracteres en C?". Ya llegaremos a eso. Pero como veremos, puede que ya funcione en tu sistema. O puede que tengas que recurrir a una biblioteca de terceros.

Vamos a hablar de muchas cosas en este capítulo—algunas son agnósticas a la plataforma, y otras son específicas de C.

Hagamos primero un resumen de lo que vamos a ver:

- Antecedentes de Unicode
- Antecedentes de codificación de caracteres
- Conjuntos de caracteres de origen y ejecución
- Uso de Unicode y UTF-8
- Usando otros tipos de caracteres como `wchar_t`, `char16_t`, y `char32_t`

¡Vamos a sumergirnos!

## 27.1 ¿Qué es Unicode?

Antiguamente, en EE.UU. y en gran parte del mundo se solía utilizar una codificación de 7 u 8 bits para los caracteres de la memoria. Esto significaba que podíamos tener 128 o 256 caracteres (incluidos los caracteres no imprimibles) en total. Eso estaba bien para un mundo centrado en EE.UU, pero resulta que en realidad hay otros alfabetos ahí fuera… ¿quién lo iba a decir? El chino tiene más de 50.000 caracteres, y eso no cabe en un byte.

Así que la gente inventó todo tipo de formas alternativas para representar sus propios conjuntos de caracteres personalizados. Y eso estaba bien, pero se convirtió en una pesadilla de compatibilidad.

Para evitarlo, se inventó Unicode. Un conjunto de caracteres para gobernarlos a todos. Se extiende hasta el infinito (efectivamente) para que nunca nos quedemos sin espacio para nuevos caracteres. Incluye caracteres chinos, latinos, griegos, cuneiformes, símbolos de ajedrez, emojis… ¡casi todo! Y cada vez se añaden más.

## 27.2 Puntos de código

Quiero hablar de dos conceptos. Es confuso porque ambos son números… números diferentes para la misma cosa. Pero tengan paciencia.

Definamos vagamente *punto de código* como un valor numérico que representa un carácter. (Los puntos de código también pueden representar caracteres de control no imprimibles, pero suponga que me refiero a algo como la letra "B" o el carácter "π").

Cada punto de código representa un carácter único. Y cada carácter tiene asociado un punto de código numérico único.

Por ejemplo, en Unicode, el valor numérico 66 representa "B", y 960 representa "π". Otros mapeados de caracteres que no son Unicode utilizan valores diferentes, pero olvidémonos de ellos y concentrémonos en Unicode, ¡el futuro!

Así que eso es una cosa: hay un número que representa a cada carácter. En Unicode, estos números van de 0 a más de 1 millón.

¿Entendido?

Porque estamos a punto de voltear la mesa un poco.

## 27.3 Codificación

Si recuerdas, un byte de 8 bits puede contener valores de 0 a 255, ambos inclusive. Eso está muy bien para "B" que es 66—que cabe en un byte. Pero "π" es 960, ¡y eso no cabe en un byte! Necesitamos otro byte. ¿Cómo almacenamos todo eso en la memoria? ¿O qué pasa con los números más grandes, como 195.024? Necesitaremos varios bytes.

La gran pregunta: ¿cómo se representan estos números en la memoria? Esto es lo que llamamos la *codificación* de los caracteres.

Así que tenemos dos cosas: una es el punto de código, que nos indica efectivamente el número de serie de un carácter concreto. Y tenemos la codificación, que nos dice cómo vamos a representar ese número en la memoria.

Hay muchas codificaciones. [1]. Pero vamos a ver algunas codificaciones realmente comunes que se usan con Unicode.

| Codificación | Descripción |
|---|---|
| UTF-8 | Codificación orientada a bytes que utiliza un número variable de bytes por carácter. Esta es la que se debe utilizar. |
| UTF-16 | Una codificación de 16 bits por carácter[2]. |
| UTF-32 | Una codificación de 32 bits por carácter. |

Con UTF-16 y UTF-32, el orden de bytes importa, por lo que puede ver UTF-16BE para big-endian y UTF-16LE para little-endian. Lo mismo ocurre con UTF-32. Técnicamente, si no se especifica, se debe asumir big-endian. Pero como Windows usa UTF-16 extensivamente y es little-endian, a veces se asume[3].

Veamos algunos ejemplos. Voy a escribir los valores en hexadecimal porque son exactamente dos dígitos por byte de 8 bits, y así es más fácil ver cómo se ordenan las cosas en la memoria.

| Caracter | Punto de Código | UTF-16BE | UTF-32BE | UTF-16LE | UTF-32LE | UTF-8 |
|---|---|---|---|---|---|---|
| A | 41 | 0041 | 00000041 | 4100 | 41000000 | 41 |

[1] Por ejemplo, podríamos almacenar el punto de código en un entero big-endian de 32 bits. ¡Sencillo! Acabamos de inventar una codificación. En realidad no; eso es lo que es la codificación UTF-32BE. Oh, bueno… ¡volvamos a la rutina!

[2] Ish. Técnicamente, es de anchura variable—hay una manera de representar puntos de código superiores a $2^{16}$ juntando dos caracteres UTF 16.

[3] Hay un carácter especial llamado *Byte Order Mark* (BOM), punto de código 0xFEFF, que puede preceder opcionalmente al flujo de datos e indicar el endianess. Sin embargo, no es obligatorio

| Caracter | Punto de Código | UTF-16BE | UTF-32BE | UTF-16LE | UTF-32LE | UTF-8 |
|---|---|---|---|---|---|---|
| B | 42 | 0042 | 00000042 | 4200 | 42000000 | 42 |
| ~ | 7E | 007E | 0000007E | 7E00 | 7E000000 | 7E |
| π | 3C0 | 03C0 | 000003C0 | C003 | C0030000 | CF80 |
| € | 20AC | 20AC | 000020AC | AC20 | AC200000 | E282AC |

Busca ahí los patrones. Tenga en cuenta que UTF-16BE y UTF-32BE son simplemente el punto de código representado directamente como valores de 16 y 32 bits[4].

Little-endian es lo mismo, excepto que los bytes están en orden little-endian.

Luego tenemos UTF-8 al final. En primer lugar, te darás cuenta de que los puntos de código de un solo byte se representan como un solo byte. Eso está bien. También puedes observar que los distintos puntos de código ocupan un número diferente de bytes. Se trata de una codificación de ancho variable.

Así que tan pronto como superamos un cierto valor, UTF-8 empieza a utilizar bytes adicionales para almacenar los valores. Y tampoco parecen estar correlacionados con el valor del punto de código.

Los detalles de la codificación UTF-8[5] quedan fuera del alcance de esta guía, pero basta con saber que tiene un número variable de bytes por punto de código, y que esos valores de bytes no coinciden con el punto de código *excepto los 128 primeros puntos de código*. Si realmente quieres aprender más, Computerphile tiene un gran video de UTF-8 con Tom Scott[6].

Esto último es lo bueno de Unicode y UTF-8 desde una perspectiva norteamericana: ¡es compatible con la codificación ASCII de 7 bits! Así que si estás acostumbrado a ASCII, UTF-8 es lo mismo. Todos los documentos codificados en ASCII también están codificados en UTF-8. (Pero no al revés, obviamente).

Probablemente sea este último punto más que ningún otro el que está impulsando a UTF-8 a conquistar el mundo.

## 27.4 Juegos de caracteres de origen y ejecución

Al programar en C, hay (al menos) tres conjuntos de caracteres en juego:

- El que su código existe en el disco como.
- El que el compilador traduce justo cuando comienza la compilación (el *source character set*). Este puede ser el mismo que el del disco, o puede que no.
- Aquel al que el compilador traduce el juego de caracteres fuente para su ejecución (el *juego de caracteres de ejecución*). Este puede ser el mismo que el juego de caracteres fuente, o puede que no.

Su compilador probablemente disponga de opciones para seleccionar estos conjuntos de caracteres en el momento de la compilación.

El juego de caracteres básico tanto para el origen como para la ejecución contendrá los siguientes caracteres:

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
```

[4] De nuevo, esto sólo es cierto en UTF-16 para caracteres que caben en dos bytes
[5] https://en.wikipedia.org/wiki/UTF-8
[6] https://www.youtube.com/watchv=MijmeoH9LT4

```
space tab vertical-tab
form-feed end-of-line
```

Esos son los caracteres que puedes utilizar en tu código fuente y seguir siendo 100% portable.

El conjunto de caracteres de ejecución tendrá además caracteres para alerta (campana/flash), retroceso, retorno de carro y nueva línea.

Pero la mayoría de la gente no llega a ese extremo y utiliza libremente sus conjuntos de caracteres extendidos en el código fuente y el ejecutable, especialmente ahora que Unicode y UTF-8 son cada vez más comunes. Quiero decir, ¡el juego de caracteres básico ni siquiera permite `@`, `$`, o !

En particular, es un engorro (aunque posible con secuencias de escape) introducir caracteres Unicode utilizando sólo el juego de caracteres básico.

## 27.5 Unicode en C

Antes de entrar en la codificación en C, hablemos de Unicode desde el punto de vista de los puntos de código. Hay una manera en C para especificar caracteres Unicode y estos serán traducidos por el compilador en el conjunto de caracteres de ejecución[7].

Entonces, ¿cómo lo hacemos?

¿Qué tal el símbolo del euro, punto de código 0x20AC? (Lo he escrito en hexadecimal porque ambas formas de representarlo en C requieren hexadecimal). ¿Cómo podemos ponerlo en nuestro código C?

Utiliza el escape `\u` para ponerlo en una cadena, por ejemplo `"\u20AC"` (las mayúsculas y minúsculas del hexadecimal no importan). Debe poner **exactamente cuatro** dígitos hexadecimales después de la "u", rellenando con ceros a la izquierda si es necesario.

He aquí un ejemplo:

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

Así pues, `\u` funciona para los puntos de código Unicode de 16 bits, pero ¿qué pasa con los que tienen más de 16 bits? Para eso, necesitamos mayúsculas: `\U`.

Por ejemplo:

```
char *s = "\U0001D4D1";

printf("%s\n", s);  // Imprime una matemática letter "B"
```

Es lo mismo que `\u`, sólo que con 32 bits en lugar de 16. Son equivalentes:

```
\u03C0
\U000003C0
```

De nuevo, se traducen al juego de caracteres de ejecución durante la compilación. Representan puntos de código Unicode, no una codificación específica. Además, si un punto de código Unicode no es representable en el conjunto de caracteres de ejecución, el compilador puede hacer lo que quiera con él.

Ahora bien, puede que se pregunte por qué no puede hacer esto sin más:

[7] Presumiblemente el compilador hace el mejor esfuerzo para traducir el punto de código a cualquiera que sea la codificación de salida, pero no puedo encontrar ninguna garantía en la especificación

```
char *s = "€1.23";

printf("%s\n", s);  // €1.23
```

Y probablemente pueda, dado un compilador moderno. El juego de caracteres fuente será traducido por el compilador al juego de caracteres de ejecución. Pero los compiladores son libres de vomitar si encuentran cualquier carácter que no esté incluido en su juego de caracteres extendido, y el símbolo € ciertamente no está en el juego de caracteres básico.

Advertencia de la especificación: no se puede utilizar `\u` o `\U` para codificar ningún punto de código por debajo de 0xA0 excepto 0x24 (`$`), 0x40 (`@`), y 0x60 (“ “‘)—sí, esos son precisamente el trío de signos de puntuación comunes que faltan en el juego de caracteres básico. Al parecer, esta restricción se relajará en la próxima versión de la especificación.

Por último, también puede utilizar estos en identificadores en su código, con algunas restricciones. Pero no quiero entrar en eso aquí. En este capítulo nos centramos en el manejo de cadenas.

Y eso es todo sobre Unicode en C (excepto la codificación).

## 27.6 Una nota rápida sobre UTF-8 antes de adentrarnos en la maleza

Podría ser que tu archivo fuente en disco, los caracteres fuente extendidos y los caracteres de ejecución extendidos estén todos en formato UTF-8. Y las bibliotecas que utilizas esperan UTF-8. Este es el glorioso futuro de UTF-8 en todas partes.

Si ese es el caso, y no te importa ser no-portable a sistemas que no son así, entonces simplemente corre con ello. Mete caracteres Unicode en tus fuentes y datos a voluntad. Usa cadenas C normales y sé feliz.

Muchas cosas funcionarán (aunque de forma no portable) porque las cadenas UTF-8 pueden terminar en NUL de forma segura como cualquier otra cadena C. Pero tal vez perder portabilidad a cambio de un manejo más sencillo de los caracteres sea una compensación que merezca la pena.

Sin embargo, hay algunas advertencias:

- Cosas como `strlen()` informan del número de bytes de una cadena, no necesariamente del número de caracteres. (La función `mbstowcs()` devuelve el número de caracteres de una cadena cuando la convierte a caracteres anchos. POSIX extiende esto para que pueda pasar `NULL` para el primer argumento si sólo quiere el recuento de caracteres).
- Lo siguiente no funcionará correctamente con caracteres de más de un byte: `strtok()`, `strchr()` (use `str()` en su lugar), funciones del tipo `strspn()`, `toupper()`, `tolower()`, `isalpha()`-type functions, y probablemente más. Cuidado con todo lo que opere sobre bytes.
- `printf()` variants allow for a way to only print so many bytes of a string[8]. Quieres asegurarte de que imprimes el número correcto de bytes para terminar en un límite de carácter.
- Si quieres `malloc()` espacio para una cadena, o declarar un array de `char`s para una, ten en cuenta que el tamaño máximo podría ser más de lo que esperabas. Cada caracter puede ocupar hasta `MB_LEN_MAX` bytes (de `<limits.h>`)—excepto los caracteres del juego de caracteres básico que se garantiza que son de un byte

Y probablemente otros que no he descubierto. Háganme saber qué trampas hay por ahí…

## 27.7 Diferentes tipos de personajes

Quiero introducir más tipos de caracteres. Estamos acostumbrados a `char`, ¿verdad?

[8] Con un especificador de formato como `"%.12s"`, por ejemplo.

Pero eso es demasiado fácil. ¡Hagamos las cosas mucho más difíciles! ¡Sí!

### 27.7.1 Caracteres multibyte

En primer lugar, quiero cambiar potencialmente tu idea de lo que es una cadena (array de `char`s). Son *cadenas multibyte* formadas por *caracteres multibyte*.

Así es, una cadena de caracteres común y corriente es multibyte. Cuando alguien dice "cadena C", quiere decir "cadena multibyte C".

Incluso si un carácter en particular en la cadena es sólo un byte, o si una cadena se compone sólo de caracteres simples, se conoce como una cadena multibyte.

Por ejemplo:

```
char c[128] = "Hello, world!";  // Multibyte string
```

Lo que queremos decir con esto es que un carácter concreto que no esté en el juego de caracteres básico podría estar compuesto por varios bytes. Hasta `MB_LEN_MAX` de ellos (de `<limits.h>`). Claro, sólo parece un carácter en la pantalla, pero podrían ser múltiples bytes.

También puedes meter valores Unicode, como vimos antes:

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

Pero aquí entramos en algo raro, porque mira esto:

```
char *s = "\u20AC1.23";  // €1.23

printf("%zu\n", strlen(s));  // 7!
```

¡¿La longitud de la cadena de `"€1.23"` es `7`?! ¡Sí! Bueno, en mi sistema, ¡sí! Recuerde que `strlen()` devuelve el número de bytes de la cadena, no el número de caracteres. (Cuando lleguemos a "caracteres anchos", más adelante, veremos una forma de obtener el número de caracteres de la cadena).

Tenga en cuenta que aunque C permite constantes individuales multibyte `char` (en oposición a `char*`), el comportamiento de éstas varía según la implementación y su compilador podría advertirle de ello.

GCC, por ejemplo, advierte de constantes de caracteres multibyte para las dos líneas siguientes (y, en mi sistema, imprime la codificación UTF-8):

```
printf("%x\n", '€');
printf("%x\n", '\u20ac');
```

### 27.7.2 Caracteres anchos

Si no es un carácter multibyte, entonces es un *carácter ancho*.

Un carácter ancho es un valor único que puede representar cualquier carácter en la configuración regional actual. Es análogo a los puntos de código Unicode. Pero podría no serlo. O podría serlo.

Básicamente, mientras que las cadenas de caracteres multibyte son matrices de bytes, las cadenas de caracteres anchos son matrices de *caracteres*. Así que puedes empezar a pensar carácter por carácter en lugar de byte por byte (esto último se complica cuando los caracteres empiezan a ocupar un número variable de bytes).

Los caracteres anchos pueden representarse mediante varios tipos, pero el más destacado es `wchar_t`. Es el principal. Es como `char`, pero ancho.

Te estarás preguntando si no puedes saber si es Unicode o no, ¿cómo te permite eso mucha flexibilidad a la hora de escribir código? `wchar_t` abre algunas de esas puertas, ya que hay un rico conjunto de funciones que puedes usar para tratar con cadenas `wchar_t` (como obtener la longitud, etc.) sin preocuparte de la codificación.

## 27.8 Uso de caracteres anchos y `wchar_t`

Es hora de un nuevo tipo: `wchar_t`. Este es el principal tipo de carácter ancho. ¿Recuerdas que un `char` es sólo un byte? ¿Y un byte no es suficiente para representar todos los caracteres, potencialmente? Pues este es suficiente.

Para usar `wchar_t`, incluye `<wchar.h>`.

¿De cuántos bytes es? Bueno, no está del todo claro. Podrían ser 16 bits. Podrían ser 32 bits.

Pero espera, estás diciendo—si son sólo 16 bits, no es lo suficientemente grande como para contener todos los puntos de código Unicode, ¿verdad? Tienes razón, no lo es. La especificación no requiere que lo sea. Sólo tiene que ser capaz de representar todos los caracteres de la configuración regional actual.

Esto puede causar problemas con Unicode en plataformas con `wchar_t`s de 16 bits (ejem—Windows). Pero eso está fuera del alcance de esta guía.

Puede declarar una cadena o carácter de este tipo con el prefijo `L`, y puede imprimirlos con el especificador de formato `%ls` (“ell ess”). O imprimir un `wchar_t` individual con `%lc`.

```
wchar_t *s = L"Hello, world!";
wchar_t c = L'B';

printf("%ls %lc\n", s, c);
```

Ahora bien, ¿estos caracteres se almacenan como puntos de código Unicode o no? Depende de la implementación. Pero puedes comprobar si lo están con la macro . `__STDC_ISO_10646__`. Si está definida, la respuesta es: “¡Es Unicode!”.

Más detalladamente, el valor de esa macro es un número entero de la forma `yyyymm` que le permite saber en qué estándar Unicode puede confiar—el que estuviera en vigor en esa fecha.

Pero, ¿cómo se utilizan?

### 27.8.1 Conversiones de Multibyte a `wchar_t`

Entonces, ¿cómo pasamos de las cadenas estándar orientadas a bytes a las cadenas anchas orientadas a caracteres y viceversa?

Podemos utilizar un par de funciones de conversión de cadenas para hacerlo.

Primero, algunas convenciones de nomenclatura que verás en estas funciones:

- `mb`: multibyte
- `wc`: carácter ancho
- `mbs`: cadena multibyte
- `wcs`: cadena de caracteres anchos

Así que si queremos convertir una cadena multibyte en una cadena de caracteres anchos, podemos llamar a `mbstowcs()`. Y al revés: `wcstombs()`.

| Función de conversión | Descripción |
|---|---|
| `mbtowc()` | Convierte un carácter multibyte en un carácter ancho. |
| `wctomb()` | Convierte un carácter ancho en un carácter multibyte. |
| `mbstowcs()` | Convierte una cadena multibyte en una cadena ancha. |
| `wcstombs()` | Convierte una cadena ancha en una cadena multibyte. |

Hagamos una demostración rápida en la que convertiremos una cadena multibyte en una cadena de caracteres anchos, y compararemos las longitudes de cadena de ambas utilizando sus respectivas funciones.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int main(void)
{
   // Salir de la configuración regional C a una que probablemente tenga el símbolo del euro
    setlocale(LC_ALL, "");

    // Cadena multibyte original con el símbolo del euro (punto 20ac de Unicode)
    char *mb_string = "The cost is \u20ac1.23";  // €1.23
    size_t mb_len = strlen(mb_string);

    // Matriz de caracteres anchos que contendrá la cadena convertida
    wchar_t wc_string[128];  // Contiene hasta 128 caracteres de ancho

    // Convierte la cadena MB a WC; esto devuelve el número de caracteres anchos
    size_t wc_len = mbstowcs(wc_string, mb_string, 128);

    // Imprime el resultado - nota las %ls para cadenas de caracteres anchos
    printf("multibyte: \"%s\" (%zu bytes)\n", mb_string, mb_len);
    printf("wide char: \"%ls\" (%zu characters)\n", wc_string, wc_len);
}
```

En mi sistema, esta salida:

```
multibyte: "The cost is €1.23" (19 bytes)
wide char: "The cost is €1.23" (17 characters)
```

(Su sistema puede variar en el número de bytes dependiendo de su localización).

Una cosa interesante a tener en cuenta es que `mbstowcs()`, además de convertir la cadena multibyte a ancha, devuelve la longitud (en caracteres) de la cadena de caracteres anchos. En sistemas compatibles con POSIX, puede aprovechar un modo especial en el que *sólo* devuelve la longitud en caracteres de una cadena multibyte dada: sólo tiene que pasar `NULL` al destino, y `0` al número máximo de caracteres a convertir (este valor se ignora).

(En el código de abajo, estoy usando mi juego de caracteres fuente extendido—puede que tengas que reemplazarlos con escapes `\u`).

```
setlocale(LC_ALL, "");

// La siguiente cadena tiene 7 caracteres
size_t len_in_chars = mbstowcs(NULL, "§¶°±π€•", 0);

printf("%zu", len_in_chars);  // 7
```

De nuevo, es una extensión POSIX no portable.

Y, por supuesto, si quieres convertir de la otra manera, es `wcstombs()`.

## 27.9 Funcionalidad de los caracteres anchos

Una vez en la tierra de los caracteres anchos, tenemos todo tipo de funciones a nuestra disposición. Sólo voy a resumir un montón de funciones, pero básicamente lo que tenemos aquí son las versiones de caracteres anchos de las funciones de cadena multibyte a las que estamos acostumbrados. (Por ejemplo, conocemos `strlen()` para cadenas multibyte; hay una <]

Muchas de estas funciones utilizan un `wint_t` para contener caracteres individuales, ya sean pasados o devueltos.

Está relacionado con `wchar_t` por naturaleza. Un `wint_t` es un entero que puede representar todos los valores del juego de caracteres extendido, y también un carácter especial de fin de fichero, `WEOF`.

Lo utilizan varias funciones de caracteres anchos orientadas a un solo carácter.

### 27.9.1 Orientación del flujo de E/S

Lo importante es no mezclar funciones orientadas a bytes (como `fprintf()`) con funciones orientadas a ancho (como `fwprintf()`). Decide si un flujo será orientado a bytes o a ancho y quédate con esos tipos de funciones de E/S.

En más detalle: los flujos pueden estar orientados a bytes u orientados a ancho. Cuando un flujo se crea por primera vez, no tiene orientación, pero la primera lectura o escritura establecerá la orientación.

Si utiliza por primera vez una operación amplia (como `fwprintf()`) orientará el flujo de forma amplia.

Si utiliza por primera vez una operación byte (como `fprintf()`) orientará el flujo por bytes.

Puede orientar manualmente un flujo desorientado de una forma u otra con una llamada a . `fwide()`. Puede utilizar esa misma función para obtener la orientación de un flujo.

Si necesitas cambiar la orientación en mitad del vuelo, puedes hacerlo con `freopen()`.

### 27.9.2 Funciones de E/S

Típicamente incluye `<stdio.h>` y `<wchar.h>` para estas.

| I/O Función | Descripción |
|---|---|
| `wprintf()` | Salida de consola formateada. |
| `wscanf()` | Entrada de consola formateada. |
| `getwchar()` | Entrada de consola basada en caracteres. |
| `putwchar()` | Salida de consola basada en caracteres. |
| `fwprintf()` | Salida de archivos formateados. |
| `fwscanf()` | Entrada de archivos formateados. |
| `fgetwc()` | Entrada de archivos basada en caracteres. |

| I/O Función | Descripción |
|---|---|
| `fputwc()` | Salida de archivos basada en caracteres. |
| `fgetws()` | Entrada de archivos basada en cadenas. |
| `fputws()` | Salida de archivos basada en cadenas. |
| `swprintf()` | Cadena formateada output. |
| `swscanf()` | Cadena formateada input. |
| `vfwprintf()` | Salida de archivo con formato variable. |
| `vfwscanf()` | Archivo con formato variadic entrada. |
| `vswprintf()` | Salida de cadena con formato variable. |
| `vswscanf()` | Entrada de cadena con formato variable. |
| `vwprintf()` | Salida de consola con formato variable. |
| `vwscanf()` | Entrada de consola con formato variable. |
| `ungetwc()` | Empuja un carácter ancho hacia atrás en un flujo de salida. |
| `fwide()` | Obtener o establecer la orientación multibyte/ancha del flujo. |

### 27.9.3 Funciones de conversión de tipos

Típicamente incluye `<wchar.h>` para esto.

| Función de conversión | Descripción |
|---|---|
| `wcstod()` | Convierte cadena(String) a `doble`. |
| `wcstof()` | Convierte cadena a `float`. |
| `wcstold()` | Convierte cadena a `long double`. |
| `wcstol()` | Convierte cadena a `long`. |
| `wcstoll()` | Convierte cadena a `long long`. |
| `wcstoul()` | Convierte cadena a `unsigned long`. |
| `wcstoull()` | Convierte cadena a `unsigned long long`. |

### 27.9.4 Funciones de copia de cadenas y memoria

Típicamente incluye `<wchar.h>` para estas.

| Función de copia | Descripción |
|---|---|
| `wcscpy()` | Copiar cadena. |
| `wcsncpy()` | Cadena de copia, de longitud limitada. |
| `wmemcpy()` | Copiar memoria. |
| `wmemmove()` | Copia la memoria potencialmente solapada. |
| `wcscat()` | Concatenar cadenas. |
| `wcsncat()` | Concatenar cadenas, longitud limitada. |

### 27.9.5 Funciones de comparación de cadenas y memoria

Típicamente incluye `<wchar.h>` para estas.

| Función de comparación | Descripción |
|---|---|
| `wcscmp()` | Compara cadenas lexicográficamente. |
| `wcsncmp()` | Compara cadenas lexicográficamente, con límite de longitud. |
| `wcscoll()` | Compara cadenas en orden de diccionario por configuración regional. |

| Función de comparación | Descripción |
|---|---|
| `wmemcmp()` | Compara la memoria lexicográficamente. |
| `wcsxfrm()` | Transforma cadenas en versiones tales que `wcscmp()` se comporta como `wcscoll()`[9]. |

### 27.9.6 Funciones de búsqueda de cadenas

Típicamente incluye `<wchar.h>` para estas.

| Función de búsqueda | Descripción |
|---|---|
| `wcschr()` | Find a character in a string. |
| `wcsrchr()` | Find a character in a string from the back. |
| `wmemchr()` | Find a character in memory. |
| `wcsstr()` | Find a substring in a string. |
| `wcspbrk()` | Find any of a set of characters in a string. |
| `wcsspn()` | Find length of substring including any of a set of characters. |
| `wcscspn()` | Find length of substring before any of a set of characters. |
| `wcstok()` | Find tokens in a string. |

### 27.9.7 Longitud/Funciones varias

Typically include `<wchar.h>` for these.

| Length/Misc Function | Description |
|---|---|
| `wcslen()` | Return the length of the string. |
| `wmemset()` | Set characters in memory. |
| `wcsftime()` | Formatted date and time output. |

### 27.9.8 Funciones de clasificación de caracteres

Include `<wctype.h>` for these.

| Length/Misc Function | Description |
|---|---|
| `iswalnum()` | True if the character is alphanumeric. |
| `iswalpha()` | True if the character is alphabetic. |
| `iswblank()` | True if the character is blank (space-ish, but not a newline). |
| `iswcntrl()` | True if the character is a control character. |
| `iswdigit()` | True if the character is a digit. |
| `iswgraph()` | True if the character is printable (except space). |
| `iswlower()` | True if the character is lowercase. |
| `iswprint()` | True if the character is printable (including space). |

[9] `wcscoll()` es lo mismo que `wcsxfrm()` seguido de `wcscmp()`.

| Length/Misc Function | Description |
|---|---|
| `iswpunct()` | True if the character is punctuation. |
| `iswspace()` | True if the character is whitespace. |
| `iswupper()` | True if the character is uppercase. |
| `iswxdigit()` | True if the character is a hex digit. |
| `towlower()` | Convert character to lowercase. |
| `towupper()` | Convert character to uppercase. |

## 27.10 Estado de análisis, funciones reiniciables

Vamos a entrar un poco en las tripas de la conversión multibyte, pero esto es algo bueno de entender, conceptualmente.

Imagina cómo tu programa toma una secuencia de caracteres multibyte y los convierte en caracteres anchos, o viceversa. Puede que, en algún momento, esté a medio camino de analizar un carácter, o puede que tenga que esperar más bytes antes de determinar el valor final.

Este estado de análisis se almacena en una variable opaca de tipo `mbstate_t` y se utiliza cada vez que se realiza la conversión. Así es como las funciones de conversión llevan la cuenta de dónde se encuentran a mitad de trabajo. Y si cambias a una secuencia de caracteres diferente a mitad del proceso, o intentas buscar un lugar diferente en tu secuencia de entrada, podría confundirse.

Puede que quieras llamarme la atención sobre esto: acabamos de hacer algunas conversiones, arriba, y nunca mencioné ningún `mbstate_t` en ningún sitio.

Eso es porque las funciones de conversión como `mbstowcs()`, `wctomb()`, etc. tienen cada una su propia variable `mbstate_t` que usan. Sólo hay una por función, así que si estás escribiendo código multihilo, no es seguro usarlas.

Afortunadamente, C define versiones *restartable* de estas funciones donde puedes pasar tu propio `mbstate_t` por hilo si lo necesitas. Si estás haciendo cosas multihilo, ¡úsalas!

Nota rápida sobre la inicialización de una variable `mbstate_t`: simplemente `memset()` a cero. No hay ninguna función integrada para forzar su inicialización.

```
mbstate_t mbs;

// Establecer el estado inicial
memset(&mbs, 0, sizeof mbs);
```

Esta es una lista de las funciones de conversión reiniciables: tenga en cuenta la convención de nomenclatura de poner una "`r`" después del tipo "from":

- mbrtowc()‘—carácter multibyte a carácter ancho
- `wcrtomb()`–carácter ancho a multibyte
- `mbsrtowcs()`—cadena multibyte a cadena de caracteres anchos
- `wcsrtombs()`—cadena de caracteres anchos a cadena multibyte

Son muy similares a sus equivalentes no reiniciables, salvo que requieren que pases un puntero a tu propia variable `mbstate_t`. Y también modifican el puntero de la cadena fuente (para ayudarte si se encuentran bytes inválidos), por lo que puede ser útil guardar una copia del original.

Aquí está el ejemplo de antes en el capítulo reelaborado para pasar nuestro propio `mbstate_t`.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int main(void)
{
   // Salir de la configuración regional C a una que probablemente tenga el símbolo del euro
    setlocale(LC_ALL, "");

    // Cadena multibyte original con el símbolo del euro (punto 20ac de Unicode)
    char *mb_string = "The cost is \u20ac1.23";  // €1.23
    size_t mb_len = strlen(mb_string);

    // Matriz de caracteres anchos que contendrá la cadena convertida
    wchar_t wc_string[128];  // Contiene hasta 128 caracteres de ancho

    // Configurar el estado de conversión
    mbstate_t mbs;
    memset(&mbs, 0, sizeof mbs);  // Estado inicial

    // mbsrtowcs() modifica el puntero de entrada para que apunte al primer
    // carácter inválido, o NULL si tiene éxito. Hagamos una copia de
   // del puntero para que mbsrtowcs() se meta con él, así nuestro original queda
    // sin cambios.
    //
    // Este ejemplo probablemente sea exitoso, pero chequeamos mas adelante
    // abajo para ver.
    const char *invalid = mb_string;

    // Convierte la cadena MB a WC; esto devuelve el número de caracteres anchos
    size_t wc_len = mbsrtowcs(wc_string, &invalid, 128, &mbs);

    if (invalid == NULL) {
        printf("No invalid characters found\n");

       // Imprime el resultado - nota las %ls para cadenas de caracteres anchos
        printf("multibyte: \"%s\" (%zu bytes)\n", mb_string, mb_len);
        printf("wide char: \"%ls\" (%zu characters)\n", wc_string, wc_len);
    } else {
        ptrdiff_t offset = invalid - mb_string;
        printf("Invalid character at offset %td\n", offset);
    }
}
```

Para las funciones de conversión que gestionan su propio estado, puedes restablecer su estado interno al inicial pasando `NULL` para sus argumentos `char*`, por ejemplo:

```
mbstowcs(NULL, NULL, 0); // Restablecer el estado de análisis para mbstowcs()
mbstowcs(dest, src, 100); // Analiza algunas cosas
```

Para la E/S, cada flujo ancho gestiona su propio `mbstate_t` y lo utiliza para las conversiones de entrada y salida sobre la marcha.

Y algunas de las funciones de E/S orientadas a bytes como `printf()` y `scanf()` mantienen su propio estado interno mientras hacen su trabajo.

Finalmente, estas funciones de conversión reiniciables tienen su propio estado interno si pasas `NULL` por el parámetro `mbstate_t`. Esto hace que se comporten más como sus homólogas no reiniciables.

## 27.11 Codificaciones Unicode y C

In this section, we'll see what C can (and can't) do when it comes to three specific Unicode encodings: UTF-8, UTF-16, and UTF-32.

### 27.11.1 UTF-8

To refresh before this section, read the UTF-8 quick note, above.

Aside from that, what are C's UTF-8 capabilities?

Well, not much, unfortunately.

You can tell C that you specifically want a string literal to be UTF-8 encoded, and it'll do it for you. You can prefix a string with `u8`:

```
char *s = u8"Hello, world!";

printf("%s\n", s);   // Hello, world!--if you can output UTF-8
```

Now, can you put Unicode characters in there?

```
char *s = u8"€123";
```

Sure! If the extended source character set supports it. (gcc does.)

What if it doesn't? You can specify a Unicode code point with your friendly neighborhood `\u` and `\U`, as noted above.

But that's about it. There's no portable way in the standard library to take arbirary input and turn it into UTF-8 unless your locale is UTF-8. Or to parse UTF-8 unless your locale is UTF-8.

So if you want to do it, either be in a UTF-8 locale and:

```
setlocale(LC_ALL, "");
```

or figure out a UTF-8 locale name on your local machine and set it explicitly like so:

```
setlocale(LC_ALL, "en_US.UTF-8");  // Non-portable name
```

Or use a third-party library.

### 27.11.2 UTF-16, UTF-32, `char16_t`, y `char32_t`

`char16_t` and `char32_t` are a couple other potentially wide character types with sizes of 16 bits and 32 bits, respectively. Not necessarily wide, because if they can't represent every character in the current locale,

they lose their wide character nature. But the spec refers them as “wide character” types all over the place, so there we are.

These are here to make things a little more Unicode-friendly, potentially.

To use, include `<uchar.h>`. (That’s “u”, not “w”.)

This header file doesn’t exist on OS X—bummer. If you just want the types, you can:

```
#include <stdint.h>

typedef int_least16_t char16_t;
typedef int_least32_t char32_t;
```

But if you also want the functions, that’s all on you.

Assuming you’re still good to go, you can declare a string or character of these types with the `u` and `U` prefixes:

```
char16_t *s = u"Hello, world!";
char16_t c = u'B';

char32_t *t = U"Hello, world!";
char32_t d = U'B';
```

Now—are values in these stored in UTF-16 or UTF-32? Depends on the implementation.

But you can test to see if they are. If the macros `__STDC_UTF_16__` or `__STDC_UTF_32__` are defined (to `1`) it means the types hold UTF-16 or UTF-32, respectively.

If you’re curious, and I know you are, the values, if UTF-16 or UTF-32, are stored in the native endianess. That is, you should be able to compare them straight up to Unicode code point values:

```
char16_t pi = u"\u03C0";  // pi symbol

#if __STDC_UTF_16__
pi == 0x3C0;  // Always true
#else
pi == 0x3C0;  // Probably not true
#endif
```

### 27.11.3 Conversiones multibyte

You can convert from your multibyte encoding to `char16_t` or `char32_t` with a number of helper functions.

(Like I said, though, the result might not be UTF-16 or UTF-32 unless the corresponding macro is set to `1`.)

All of these functions are restartable (i.e. you pass in your own `mbstate_t`), and all of them operate character by character[10].

| Conversion Function | Description |
|---|---|
| `mbrtoc16()` | Convert a multibyte character to a `char16_t` character. |
| `mbrtoc32()` | Convert a multibyte character to a `char32_t` character. |
| `c16rtomb()` | Convert a `char16_t` character to a multibyte character. |
| `c32rtomb()` | Convert a `char32_t` character to a multibyte character. |

[10] Ish—things get funky with multi-`char16_t` UTF-16 encodings.

### 27.11.4 Bibliotecas de terceros

For heavy-duty conversion between different specific encodings, there are a couple mature libraries worth checking out. Note that I haven't used either of these.

- iconv[11]—Internationalization Conversion, a common POSIX-standard API available on the major platforms.
- ICU[12]—International Components for Unicode. At least one blogger found this easy to use.

If you have more noteworthy libraries, let me know.

[11] https://en.wikipedia.org/wiki/Iconv
[12] http://site.icu-project.org/

# Chapter 28

# Salir de un programa

Resulta que hay un montón de maneras de hacer esto, e incluso maneras de configurar "ganchos" para que una función se ejecute cuando un programa salga.

En este capítulo nos sumergiremos en ellas y las comprobaremos.

Ya hemos cubierto el significado del código de estado de salida en la sección Exit Status, así que vuelve allí y repásalo si es necesario.

Todas las funciones de esta sección están en `<stdlib.h>`.

## 28.1 Salidas normales

Empezaremos con las formas normales de salir de un programa, y luego saltaremos a algunas de las más raras y esotéricas.

Cuando se sale de un programa normalmente, todos los flujos de E/S abiertos se vacían y los archivos temporales se eliminan. Básicamente es una salida agradable donde todo se limpia y se maneja. Es lo que quieres hacer casi todo el tiempo a menos que tengas razones para hacer lo contrario.

### 28.1.1 Retorno de `main()`

Si te has dado cuenta, `main()` tiene un tipo de retorno `int`… y sin embargo rara vez, o nunca, he estado `devolviendo` nada de `main()` en absoluto.

Esto se debe a que `main()` sólo (y no puedo enfatizar lo suficiente este caso especial *sólo* se aplica a `main()` y a ninguna otra función en ninguna parte) tiene un `return 0` *implícito* si te caes del final.

Puedes `return` explícitamente desde `main()` cuando quieras, y algunos programadores creen que es más *Correcto* tener siempre un `return` al final de `main()`. Pero si lo dejas, C pondrá uno por ti.

Así que… aquí están las reglas de `return` para `main()`:

- Puede devolver un estado de salida desde `main()` con una sentencia `return`. `main()` es la única función con este comportamiento especial. Usar `return` en cualquier otra función sólo devuelve desde esa función a quien la llamó.
- Si no se usa `return` explícitamente y se sale al final de `main()`, es como si se hubiera devuelto `0` o `EXIT_SUCCESS`.

### 28.1.2 `exit()`

Éste también ha aparecido unas cuantas veces. Si llama a `exit()` desde cualquier parte de su programa, éste saldrá en ese punto.

El argumento que pasas a `exit()` es el estado de salida.

### 28.1.3 Configuración de los controladores de salida con `atexit()`

Puede registrar funciones para ser llamadas cuando un programa sale, ya sea volviendo de `main()` o llamando a la función `exit()`.

Una llamada a `atexit()` con el nombre de la función manejadora lo hará. Puede registrar múltiples manejadores de salida, y serán llamados en el orden inverso al registro.

He aquí un ejemplo:

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

void on_exit_1(void)
{
    printf("¡Controlador de salida 1 llamado!\n");
}

void on_exit_2(void)
{
    printf("¡Controlador de salida 2 llamado!\n");
}

int main(void)
{
    atexit(on_exit_1);
    atexit(on_exit_2);

    printf("A punto de salir...\n");
}
```

Y la salida es:

```
A punto de salir...
¡Controlador de salida 2 llamado!
¡Controlador de salida 1 llamado!
```

## 28.2 Salidas más rápidas con `quick_exit()`.

Esto es similar a una salida normal, excepto:

- Los archivos abiertos pueden no ser vaciados.
- Los ficheros temporales pueden no ser eliminados.
- Los manejadores `atexit()` no serán llamados.

Pero hay una forma de registrar manejadores de salida: llame a `at_quick_exit()` de forma análoga a como llamaría a `atexit()`.

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

void on_quick_exit_1(void)
{
    printf("Llamada al gestor de salida rápida 1\n");
}

void on_quick_exit_2(void)
{
    printf("Llamada al gestor de salida rápida 2\n");
}

void on_exit(void)
{
    printf("Salida normal... ¡No me llamarán!\n");
}

int main(void)
{
    at_quick_exit(on_quick_exit_1);
    at_quick_exit(on_quick_exit_2);

    atexit(on_exit);  // Esto no se llamará

    printf("A punto de salir rápidamente...\n");

    quick_exit(0);
}
```

Lo que da esta salida:

```
A punto de salir rápidamente...
Llamada al gestor de salida rápida 2
Llamada al gestor de salida rápida 1
```

Funciona igual que `exit()`/`atexit()`, excepto por el hecho de que el vaciado y limpieza de ficheros puede no realizarse.

## 28.3 Destrúyelo desde la órbita: `_Exit()`

Llamando a `_Exit()` se sale inmediatamente, punto. No se ejecutan funciones de callback al salir. Los ficheros no se vaciarán. Los ficheros temporales no se eliminan.

Use esto si tiene que salir *ahora mismo*.

## 28.4 Saliendo a veces: `assert()`

La sentencia `assert()` se utiliza para insistir en que algo sea cierto, o de lo contrario el programa se cerrará.

Los desarrolladores a menudo utilizan un assert para detectar errores del tipo “no debería ocurrir nunca”.

```
#define PI 3.14159

assert(PI > 3);   // Claro que sí, así que continúa.
```

versus:

```
goats -= 100;

assert(goats >= 0);  // No puede tener cabras negativas
```

En ese caso, si intento ejecutarlo y `goats` cae bajo `0`, ocurre esto:

```
goat_counter: goat_counter.c:8: main: Assertion `goats >= 0' failed.
Aborted
```

y vuelvo a la línea de comandos.

Esto no es muy fácil de usar, así que sólo se usa para cosas que el usuario nunca verá. Y a menudo la gente escribe sus propias macros assert que pueden ser desactivadas más fácilmente.

## 28.5 Salida anormal: `abort()`

Puedes utilizarlo si algo ha ido terriblemente mal y quieres indicarlo al entorno exterior. Esto tampoco limpiará necesariamente cualquier archivo abierto, etc.

Raramente he visto usar esto.

Puedes utilizarlo si algo ha ido terriblemente mal y quieres indicarlo al entorno exterior. Esto tampoco limpiará necesariamente cualquier archivo abierto, etc.

Rara vez he visto que se utilice.

Un poco de anticipación sobre las *señales*: esto en realidad funciona lanzando una `SIGABRT` que terminará el proceso.

Lo que suceda después depende del sistema, pero en los Unix, era común dump core[1] cuando el programa terminaba.

[1] https://en.wikipedia.org/wiki/Core_dump

# Chapter 29

# Manejo de señales

Antes de empezar, voy a aconsejarte que ignores todo este capítulo y utilices las (muy probablemente) superiores funciones de manejo de señales de tu sistema operativo. Los Unix tienen la función .

Una vez aclarado esto, ¿qué son las señales?

## 29.1 ¿Qué son las señales?

Una *señal* es *levantada* en una variedad de eventos externos. Su programa puede ser configurado para ser interrumpido para *manejar* la señal, y, opcionalmente, continuar donde lo dejó una vez que la señal ha sido manejada.

Piense en ello como una función que se llama automáticamente cuando se produce uno de estos eventos externos.

¿Cuáles son estos eventos? En tu sistema, probablemente haya muchos, pero en la especificación C sólo hay unos pocos:

| Signal | Descipción |
|---|---|
| `SIGABRT` | Terminación anormal—lo que ocurre cuando se llama a `abort()`. |
| `SIGFPE` | Excepción de coma flotante. |
| `SIGILL` | Instrucción ilegal. |
| `SIGINT` | Interrupción: normalmente el resultado de pulsar “CTRL-C”. |
| `SIGSEGV` | “Violación de segmentación”: acceso inválido a memoria. |
| `SIGTERM` | Terminación solicitada. |

Puede configurar su programa para ignorar, manejar o permitir la acción por defecto para cada uno de ellos utilizando la función `signal()`.

## 29.2 Manejo de señales con `signal()`.

La llamada a `signal()` toma dos parámetros: la señal en cuestión, y una acción a tomar cuando esa señal es lanzada.

La acción puede ser una de estas tres cosas:

- Un puntero a una función manejadora.
- `SIG_IGN` para ignorar la señal.

- `SIG_DFL` para restaurar el manejador por defecto de la señal.

Escribamos un programa del que no puedas salir con `CTRL-C`. (No te preocupes–en el siguiente programa, también puedes pulsar `RETURN` y saldrá).

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

int main(void)
{
    char s[1024];

    signal(SIGINT, SIG_IGN);    // Ignorar SIGINT, causado por ^C

    printf("Prueba a pulsar ^C... (pulsa RETURN para salir)\n");

    // Esperar una línea de entrada para que el programa no salga sin más
    fgets(s, sizeof s, stdin);
}
```

Mira la línea 8: le decimos al programa que ignore “SIGINT”, la señal de interrupción que se activa cuando se pulsa “CTRL-C”. No importa cuánto la pulses, la señal permanece ignorada. Si comentas la línea 8, verás que puedes pulsar `CTRL-C` impunemente y salir del programa en el acto.

## 29.3 Escribiendo Manejadores de Señales

He mencionado que también se puede escribir una función manejadora que se llama cuando la señal se eleva.

Estos son bastante sencillos, también son muy limitados en cuanto a la capacidad de la especificación.

Antes de empezar, veamos el prototipo de función para la llamada `signal()`:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Bastante fácil de leer, ¿verdad?

ERROR.

Vamos a desmenuzarlo un poco para practicar.

`signal()` toma dos argumentos: un entero `sig` que representa la señal, y un puntero `func` al manejador (el manejador devuelve `void` y toma un `int` como argumento), resaltado abajo:

```
               sig          func
             |-----|  |---------------|
void (*signal(int sig, void (*func)(int)))(int);
```

Básicamente, vamos a pasar en el número de señal que estamos interesados en la captura, y vamos a pasar un puntero a una función de la forma:

```
void f(int x);
```

que hará la captura real.

Ahora… ¿qué pasa con el resto del prototipo? Básicamente es todo el tipo de retorno. Verás, `signal()` devolverá lo que hayas pasado como `func` en caso de éxito… así que eso significa que devuelve un puntero a una función que devuelve `void` y toma un `int` como argumento.

```
returned
function    indicates we're             and
returns     returning a                 that function
void        pointer to function         takes an int
|--|        |                                 |---|
void       (*signal(int sig, void (*func)(int)))(int);
```

Además, puede devolver `SIG_ERR` en caso de error.

Hagamos un ejemplo donde tengamos que pulsar `CTRL-C` dos veces para salir.

Quiero dejar claro que este programa tiene un comportamiento indefinido en un par de formas. Pero probablemente te funcione, y es difícil hacer demos portables no triviales.

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

int count = 0;

void sigint_handler(int signum)
{
    // El compilador puede funcionar:
    //
    // signal(signum, SIG_DFL)
    //
    // cuando el manejador es llamado. Así que aquí reiniciamos el manejador:
    signal(SIGINT, sigint_handler);

    (void)signum;   // Deshacerse del aviso de variable no utilizada
    count++;                        // Comportamiento indefinido
    printf("Count: %d\n", count);   // Comportamiento indefinido
    if (count == 2) {
        printf("Exiting!\n");       // Comportamiento indefinido
        exit(0);
    }
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Try hitting ^C...\n");

    for(;;);  // Espera aquí para siempre
}
```

Una de las cosas que notarás es que en la línea 14 reiniciamos el manejador de señales. Esto es porque C tiene la opción de resetear el manejador de señales a su `SIG_DFL` antes de ejecutar tu manejador personalizado. En otras palabras. Así que lo reseteamos a la primera para volver a manejarlo en la siguiente.

Estamos ignorando el valor de retorno de `signal()` en este caso. Si lo hubiéramos puesto antes en un manejador diferente, devolvería un puntero a ese manejador, que podríamos obtener así:

```
// old_handler es del tipo "puntero a función que toma un único parámetro
// parámetro int y devuelve void":

void (*old_handler)(int);

old_handler = signal(SIGINT, sigint_handler);
```

Dicho esto, no estoy seguro de que haya un caso de uso común para esto. Pero si necesitas el antiguo manejador por alguna razón, puedes conseguirlo de esa manera.

Nota rápida sobre la línea 16—es sólo para decirle al compilador que no advierta que no estamos usando esta variable. Es como decir, "Sé que no la estoy usando; no tienes que advertirme".

Y por último verás que he marcado comportamiento indefinido en un par de sitios. Más sobre esto en la siguiente sección.

## 29.4 ¿Qué podemos hacer realmente?

Resulta que estamos bastante limitados en lo que podemos y no podemos hacer en nuestros manejadores de señales. Esta es una de las razones por las que digo que ni siquiera deberías molestarte con esto y en su lugar utilizar el manejo de señales de tu sistema operativo (por ejemplo, . `sigaction()` para sistemas tipo Unix).

Wikipedia llega a decir que lo único realmente portable que puedes hacer es llamar a `signal()` con `SIG_IGN` o `SIG_DFL` y ya está.

Esto es lo que **no** podemos hacer de forma portable:

- Llama a cualquier función de la biblioteca estándar.
  - Como `printf()`, por ejemplo.
  - Creo que es probablemente seguro llamar a funciones reiniciables/reentrantes, pero la especificación no permite esa libertad.
- Obtener o establecer valores desde una variable local `static`, file scope, o thread-local.
  - A menos que sea un objeto atómico libre de bloqueos o…
  - Estás asignando a una variable de tipo `volatile sig_atomic_t`.

Ese último bit–`sig_atomic_t`–es tu boleto para obtener datos de un manejador de señales. (A menos que quieras usar objetos atómicos sin bloqueo, lo cual está fuera del alcance de esta sección[1]). Es un tipo entero que puede o no estar firmado. Y está limitado por lo que puedes poner ahí Puede consultar los valores mínimo y máximo permitidos en las macros `SIG_ATOMIC_MIN` y `SIG_ATOMIC_MAX`[2].

Confusamente, la especificación también dice que no puedes referirte "a ningún objeto con duración de almacenamiento estático o de hilo que no sea un objeto atómico libre de bloqueos que no sea asignando un valor a un objeto declarado como `volatile sig_atomic_t` […]".

Mi lectura de esto es que no puedes leer o escribir nada que no sea un objeto atómico libre de bloqueo. También puedes asignar a un objeto que es `volatile sig_atomic_t`.

¿Pero puedes leer de él? Honestamente, no veo por qué no, excepto que la especificación es muy específica sobre la mención de asignar a. Pero si tienes que leerlo y tomar cualquier tipo de decisión basándote en ello, podrías estar abriendo espacio para algún tipo de race conditions.

Con esto en mente, podemos reescribir nuestro código "pulsa `CTRL-C` dos veces para salir" para que sea un poco más portable, aunque menos verboso en la salida.

[1] Confusamente, `sig_atomic_t` es anterior a los atómicos sin bloqueo y no es lo mismo

[2] Si `sig_action_t` es con signo, el rango será como mínimo de `-127` a `127`. Si es sin signo, al menos de `0` a `255`

Cambiemos nuestro manejador `SIGINT` para que no haga nada excepto incrementar un valor de tipo `volatile sig_atomic_t`. Así contará el número de `CTRL-C`s que han sido pulsados.

Luego, en nuestro bucle principal, comprobaremos si el contador es mayor de `2`, y si es así, lo abandonaremos.

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

volatile sig_atomic_t count = 0;

void sigint_handler(int signum)
{
    (void)signum;                   // Aviso de variable no utilizada

    signal(SIGINT, sigint_handler);  // Restablecer manejador de señal

    count++;                        // Comportamiento indefinido
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Hit ^C twice to exit.\n");

    while(count < 2);
}
```

¿Otra vez comportamiento indefinido? Yo creo que sí, porque tenemos que leer el valor para incrementarlo y almacenarlo.

Si sólo queremos posponer la salida una pulsación de `CTRL-C`, podemos hacerlo sin demasiados problemas. Pero cualquier otro aplazamiento requeriría un encadenamiento de funciones ridículo.

Lo que haremos es manejarlo una vez, y el manejador restablecerá la señal a su comportamiento por defecto (es decir, a la salida):

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

void sigint_handler(int signum)
{
    (void)signum;                     // Aviso de variable no utilizada
    signal(SIGINT, SIG_DFL);          // Restablecer manejador de señal
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Hit ^C twice to exit.\n");

    while(1);
}
```

Más adelante, cuando veamos las variables atómicas sin bloqueo, veremos una forma de arreglar la versión `count` (suponiendo que las variables atómicas sin bloqueo estén disponibles en tu sistema en particular).

Esta es la razón por la que al principio, sugería comprobar el sistema de señales integrado en tu sistema operativo como una alternativa probablemente superior.

## 29.5 Los amigos no dejan a los amigos `señal()`

De nuevo, usa el manejo de señales integrado en tu sistema operativo o su equivalente. No está en la especificación, no es tan portable, pero probablemente es mucho más capaz. Además, tu sistema operativo probablemente tenga definidas un número de señales que no están en la especificación de C. Y es difícil escribir señales portables. Más adelante, cuando veamos las variables atómicas sin bloqueo, veremos una forma de arreglar la versión `count` (suponiendo que las variables atómicas sin bloqueo estén disponibles en tu sistema en particular).

Esta es la razón por la que al principio, sugería comprobar el sistema de señales integrado en tu sistema operativo como una alternativa probablemente superior.

# Chapter 30

# Matrices de longitud variable (VLA)

C permite declarar un array cuyo tamaño se determina en tiempo de ejecución. Esto te da los beneficios del dimensionamiento dinámico en tiempo de ejecución que obtienes con `malloc()`, pero sin tener que preocuparte de `free()` la memoria después.

A mucha gente no le gustan los VLAs. Por ejemplo, han sido prohibidos en el kernel de Linux. Profundizaremos más en ese razonamiento más tarde.

Se trata de una característica opcional del lenguaje. La macro `__STDC_NO_VLA__` se pone a `1` si los VLAs *no* están presentes. (Eran obligatorios en C99, y luego pasaron a ser opcionales en C11).

```
#if __STDC_NO_VLA__ == 1
   #error Sorry, need VLAs for this program!
#endif
```

Pero como ni GCC ni Clang se molestan en definir esta macro, puede que le saques poco provecho.

Vamos a sumergirnos primero con un ejemplo, y luego buscaremos el diablo en los detalles.

## 30.1 Lo Básico

Un array normal se declara con un tamaño constante, así:

```
int v[10];
```

Pero con VLAs, podemos utilizar un tamaño determinado en tiempo de ejecución para establecer la matriz, así:

```
int n = 10;
int v[n];
```

Ahora, eso parece lo mismo, y en muchos sentidos lo es, pero esto le da la flexibilidad para calcular el tamaño que necesita, y luego obtener una matriz de exactamente ese tamaño.

Vamos a pedir al usuario que introduzca el tamaño de la matriz, y luego almacenar el índice 10 veces, en cada uno de los elementos de la matriz:

```
#include <stdio.h>

int main(void)
{
    int n;
    char buf[32];

    printf("Enter a number: "); fflush(stdout);
    fgets(buf, sizeof buf, stdin);
    n = strtoul(buf, NULL, 10);

    int v[n];

    for (int i = 0; i < n; i++)
        v[i] = i * 10;

    for (int i = 0; i < n; i++)
        printf("v[%d] = %d\n", i, v[i]);
}
```

(En la línea 7, tengo un `fflush()` que debería forzar la salida de la línea aunque no tenga una nueva línea al final).

La línea 10 es donde declaramos el VLA—una vez que la ejecución pasa esa línea, el tamaño del array se establece a lo que sea `n` en ese momento. La longitud del array no se puede cambiar más tarde.

También puedes poner una expresión entre paréntesis:

```
int v[x * 100];
```

Algunas restricciones:

- No puedes declarar una VLA en el ámbito de un fichero, y no puedes hacer una `static` en el ámbito de un bloque[1].
- No puedes usar una lista inicializadora para inicializar el array.

Además, introducir un valor negativo para el tamaño del array invoca un comportamiento indefinido— al menos en este universo.

## 30.2 `sizeof` y VLAs

Estamos acostumbrados a que `sizeof` nos indique el tamaño en bytes de cualquier objeto, incluidas las matrices. Y los VLAs no son una excepción.

La principal diferencia es que `sizeof` en una VLA se ejecuta en *runtime*, mientras que en una variable de tamaño no variable se calcula en *tiempo de compilación*.

Pero el uso es el mismo.

Incluso se puede calcular el número de elementos de un VLA con el truco habitual de los arrays:

```
size_t num_elems = sizeof v / sizeof v[0];
```

[1] Esto se debe a que las VLAs se asignan típicamente en la pila, mientras que las variables `static` están en el montón. Y la idea con las VLAs es que serán automáticamente desasignadas cuando el marco de la pila sea vaciado al final de la función

Hay una implicación sutil y correcta en la línea anterior: la aritmética de punteros funciona como cabría esperar para una matriz normal. Así que adelante, úsala a tu antojo:

```
#include <stdio.h>

int main(void)
{
    int n = 5;
    int v[n];

    int *p = v;

    *(p+2) = 12;
    printf("%d\n", v[2]);  // 12

    p[3] = 34;
    printf("%d\n", v[3]);  // 34
}
```

Al igual que con las matrices normales, puede utilizar paréntesis con `sizeof()` para obtener el tamaño de un posible VLA sin tener que declararlo:

```
int x = 12;

printf("%zu\n", sizeof(int [x]));  // Imprime 48 en mi sistema
```

## 30.3 VLA multidimensionales

puede seguir adelante y hacer todo tipo de VLA con una o más dimensiones establecidas en una variable

```
int w = 10;
int h = 20;

int x[h][w];
int y[5][w];
int z[10][w][20];
```

De nuevo, puedes navegar por ellas como lo harías por un array normal.

## 30.4 Pasar VLAs unidimensionales a funciones

Pasar VLAs unidimensionales a una función no puede ser diferente de pasar un array normal. Basta con hacerlo.

```
#include <stdio.h>

int sum(int count, int *v)
{
    int total = 0;

    for (int i = 0; i < count; i++)
```

```
        total += v[i];

    return total;
}

int main(void)
{
    int x[5];   // Standard array

    int a = 5;
    int y[a];   // VLA

    for (int i = 0; i < a; i++)
        x[i] = y[i] = i + 1;

    printf("%d\n", sum(5, x));
    printf("%d\n", sum(a, y));
}
```

Pero hay algo más. También puedes hacer saber a C que el array tiene un tamaño VLA específico pasándolo primero y luego dando esa dimensión en la lista de parámetros:

```
int sum(int count, int v[count])
{
    // ...
}
```

Por cierto, hay un par de formas de listar un prototipo para la función anterior; una de ellas implica un `*` si no se quiere nombrar específicamente el valor en el VLA. Sólo indica que el tipo es un VLA en lugar de un puntero normal.

Prototipos VLA:

```
void do_something(int count, int v[count]);  // Con nombres
void do_something(int, int v[*]);            // Sin nombres
```

De nuevo, eso de `*` sólo funciona con el prototipo–en la función en sí, tendrás que poner el tamaño explícito.

Ahora… ¡vamos a lo multidimensional! Aquí empieza la diversión.

## 30.5 Pasar VLAs multidimensionales a funciones

Lo mismo que hicimos con la segunda forma de VLAs unidimensionales, arriba, pero esta vez pasamos dos dimensiones y las usamos.

En el siguiente ejemplo, construimos una matriz de tabla de multiplicación de anchura y altura variables, y luego la pasamos a una función para que la imprima. Aquí empieza la diversión.

```
#include <stdio.h>

void print_matrix(int h, int w, int m[h][w])
{
    for (int row = 0; row < h; row++) {
```

```
        for (int col = 0; col < w; col++)
            printf("%2d ", m[row][col]);
        printf("\n");
    }
}

int main(void)
{
    int rows = 4;
    int cols = 7;

    int matrix[rows][cols];

    for (int row = 0; row < rows; row++)
        for (int col = 0; col < cols; col++)
            matrix[row][col] = row * col;

    print_matrix(rows, cols, matrix);
}
```

### 30.5.1 VLA multidimensionales parciales

Puede tener algunas de las dimensiones fijas y otras variables. Digamos que tenemos una longitud de registro fija en 5 elementos, pero no sabemos cuántos registros hay.

```
#include <stdio.h>

void print_records(int count, int record[count][5])
{
    for (int i = 0; i < count; i++) {
        for (int j = 0; j < 5; j++)
            printf("%2d ", record[i][j]);
        printf("\n");
    }
}

int main(void)
{
    int rec_count = 3;
    int records[rec_count][5];

    // Fill with some dummy data
    for (int i = 0; i < rec_count; i++)
        for (int j = 0; j < 5; j++)
            records[i][j] = (i+1)*(j+2);

    print_records(rec_count, records);
}
```

## 30.6 Compatibilidad con matrices regulares

Dado que los VLA son como matrices normales en memoria, es perfectamente permisible pasarlos indistintamente… siempre que las dimensiones coincidan.

Por ejemplo, si tenemos una función que específicamente quiere un array de $3 \times 5$, podemos pasarle un VLA.

```
int foo(int m[5][3]) {...}

\\ ...

int w = 3, h = 5;
int matrix[h][w];

foo(matrix);   // OK!
```

Del mismo modo, si tiene una función VLA, puede pasarle una matriz normal:

```
int foo(int h, int w, int m[h][w]) {...}

\\ ...

int matrix[3][5];

foo(3, 5, matrix);   // OK!
```

Pero cuidado: si las dimensiones no coinciden, es probable que se produzcan comportamientos indefinidos.

## 30.7 `typedef` Y VLAs

Puedes `typedef` un VLA, pero el comportamiento puede no ser el que esperas.

Básicamente, `typedef` hace un nuevo tipo con los valores tal y como existían en el momento en que se ejecutó `typedef`.

Así que no es un `typedef` de un VLA tanto como un nuevo tipo de array de tamaño fijo de las dimensiones en ese momento.

```
#include <stdio.h>

int main(void)
{
    int w = 10;

    typedef int goat[w];

    // goat es un array de 10 ints
    goat x;

    // Init con cuadrados de números
    for (int i = 0; i < w; i++)
        x[i] = i*i;
```

```
    // Imprimirlos
    for (int i = 0; i < w; i++)
        printf("%d\n", x[i]);

    // Ahora vamos a cambiar w...

    w = 20;

    // Pero cabra es TODAVÍA un array de 10 ints, porque ese era el
    // valor de w cuando se ejecutó el typedef.
}
```

Así que actúa como un array de tamaño fijo.

Pero todavía no se puede utilizar una lista de inicializadores en él.

## 30.8 Salto de Trampas

Hay que tener cuidado cuando se usa `goto` cerca de VLAs porque muchas cosas no son legales.

Y cuando usas `longjmp()` hay un caso en el que podrías tener fugas de memoria con VLAs.

Pero ambas cosas las trataremos en sus respectivos capítulos.

## 30.9 Cuestiones generales

Los VLAs han sido prohibidos en el kernel de Linux por varias razones:

- Muchos de los lugares en los que se usaban deberían haber sido de tamaño fijo.
- El código detrás de los VLAs es más lento (en un grado que la mayoría de la gente no notaría, pero que marca la diferencia en un sistema operativo).
- No todos los compiladores de C soportan VLA en el mismo grado.
- El tamaño de la pila es limitado, y los VLAs van en la pila. Si algún código accidentalmente (o maliciosamente) pasa un valor grande a una función del núcleo que asigna un VLA, *Bad Things*™ podría suceder.

Otras personas en línea señalan que no hay manera de detectar un fallo en la asignación de un VLA, y los programas que sufrieran tales problemas probablemente simplemente se bloquearían. Aunque las matrices de tamaño fijo también tienen el mismo problema, es mucho más probable que alguien haga accidentalmente una *VLA de tamaño inusual* que declarar accidentalmente una matriz de tamaño fijo, digamos, de 30 megabytes.

# Chapter 31

# `goto`

La afirmación `goto` es universalmente venerada y puede presentarse aquí sin impugnación.

Es broma. A lo largo de los años, ha habido muchas idas y venidas sobre si `goto` es o no (a menudo no) considerado dañino[1].

En opinión de este programador, deberías usar cualquier construcción que conduzca al *mejor* código, teniendo en cuenta la mantenibilidad y la velocidad. ¡Y a veces esto puede ser `goto`!

En este capítulo, veremos cómo funciona `goto` en C, y luego comprobaremos algunos de los casos comunes en los que se usa[2].

## 31.1 Un ejemplo sencillo

En este ejemplo, vamos a utilizar `goto` para saltar una línea de código y saltar a una *etiqueta*. La etiqueta es el identificador que puede ser un objetivo de `goto`; termina con dos puntos (`:`).

```
#include <stdio.h>

int main(void)
{
    printf("One\n");
    printf("Two\n");

    goto skip_3;

    printf("Three\n");

skip_3:

    printf("Five!\n");
}
```

La salida es :

[1] https://en.wikipedia.org/wiki/Goto#Criticism

[2] Me gustaría señalar que usar `goto` en todos estos casos es evitable. Puedes usar variables y bucles en su lugar. Es sólo que algunas personas piensan que `goto` produce el *mejor* código en esas circunstancias

```
One
Two
Five!
```

`goto` envía la ejecución saltando a la etiqueta especificada, saltándose todo lo que hay entre medias.

Puedes saltar hacia delante o hacia atrás con `goto`.

```
infinite_loop:
    print("Hello, world!\n");
    goto infinite_loop;
```

Las etiquetas se omiten durante la ejecución. Lo siguiente imprimirá los tres números en orden como si las etiquetas no estuvieran allí:

```
    printf("Zero\n");
label_1:
label_2:
    printf("One\n");
label_3:
    printf("Two\n");
label_4:
    printf("Three\n");
```

Como habrá notado, es una convención común justificar las etiquetas hasta el final a la izquierda. Esto aumenta la legibilidad porque un lector puede escanear rápidamente para encontrar el destino.

Las etiquetas tienen *alcance de función*. Es decir, no importa a cuántos niveles de profundidad en los bloques aparezcan, puedes "ir a ellas" desde cualquier parte de la función.

También significa que sólo se puede "ir a" las etiquetas que están en la misma función que la propia "ir a". Las etiquetas de otras funciones están fuera del alcance de `goto`. Y significa que puedes usar el mismo nombre de etiqueta en dos funciones, pero no en la misma función.

## 31.2 Etiqueta `continue`

En algunos lenguajes, puedes especificar una etiqueta para una sentencia `continue`. C no lo permite, pero puedes usar `goto` en su lugar.

Para mostrar el problema, mira `continue` en este bucle anidado:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        continue;   // Siempre pasa al siguiente j
    }
}
```

Como vemos, ese `continue`, como todos los `continues`, va a la siguiente iteración del bucle más cercano. ¿Y si queremos `continuar` en el siguiente bucle exterior, el bucle con `i`?

Bueno, podemos `break` para volver al bucle exterior, ¿no?

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        break;     // Nos lleva a la siguiente iteración de i
    }
}
```

Eso nos da dos niveles de bucle anidado. Pero si anidamos otro bucle, nos quedamos sin opciones. ¿Qué pasa con esto, donde no tenemos ninguna declaración que nos llevará a la siguiente iteración de `i`?

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 3; k++) {
            printf("%d, %d, %d\n", i, j, k);

            continue;  // Nos lleva a la siguiente iteración de k
            break;     // Nos lleva a la siguiente iteración de j
            ????;      // Nos lleva a la siguiente iteración de i???

        }
    }
}
```

¡La sentencia `goto` nos ofrece un camino!

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 3; k++) {
                printf("%d, %d, %d\n", i, j, k);

                goto continue_i;   // ¡¡¡Ahora continuando el bucle i!!!
            }
        }
continue_i: ;
    }
```

Tenemos un `;` al final—eso es porque no se puede tener una etiqueta apuntando al final plano de una sentencia compuesta (o antes de una declaración de variable).

## 31.3 Libertad bajo fianza

Cuando estás super anidado en medio de algún código, puedes usar `goto` para salir de él de una manera que a menudo es más limpia que anidar más `if`s y usar variables flag.

```
    // Pseudocode

    for(...) {
        for (...) {
            while (...) {
                do {
                    if (some_error_condition)
```

```
                        goto bail;

                } while(...);
            }
        }
    }

bail:
    // Limpieza aquí
```

Sin `goto`, tendrías que comprobar una bandera de condición de error en todos los bucles para llegar hasta el final.

## 31.4 Etiqueta `break`

Esta situación es muy similar a la de `continue`, que sólo continúa el bucle más interno. También `break` sólo sale del bucle más interno.

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d, %d\n", i, j);
            break;   // Sólo sale del bucle j
        }
    }

    printf("Done!\n");
```

Pero podemos usar `goto` para ir más lejos:

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d, %d\n", i, j);
            goto break_i;   // ¡Ahora saliendo del bucle i!
        }
    }

break_i:

    printf("Done!\n");
```

## 31.5 Limpieza multinivel

Si estás llamando a varias funciones para inicializar varios sistemas y una de ellas falla, sólo debes desinicializar los que hayas conseguido hasta el momento.

Hagamos un ejemplo falso en el que empezamos a inicializar sistemas y comprobamos si alguno devuelve un error (usaremos `-1` para indicar un error). Si alguno lo hace, tenemos que apagar sólo los sistemas que hemos inicializado hasta ahora.

```
    if (init_system_1() == -1)
        goto shutdown;

    if (init_system_2() == -1)
        goto shutdown_1;

    if (init_system_3() == -1)
        goto shutdown_2;

    if (init_system_4() == -1)
        goto shutdown_3;

    do_main_thing();   // Ejecutar nuestro programa

    shutdown_system4();

shutdown_3:
    shutdown_system3();

shutdown_2:
    shutdown_system2();

shutdown_1:
    shutdown_system1();

shutdown:
    print("All subsystems shut down.\n");
```

Ten en cuenta que estamos apagando en el orden inverso al que inicializamos los subsistemas. Así que si el subsistema 4 no arranca, apagará el 3, el 2 y el 1 en ese orden.

## 31.6 Optimización de las llamadas de cola

Algo así. Sólo para funciones recursivas.

Si no está familiarizado, Optimización de las llamadas de cola (TCO)[3] es una forma de no desperdiciar espacio en la pila cuando se llama a otras funciones bajo circunstancias muy específicas. Por desgracia, los detalles están fuera del alcance de esta guía.

Pero si tienes una función recursiva que sabes que puede ser optimizada de esta manera, puedes hacer uso de esta técnica. (Ten en cuenta que no puedes llamar a la cola a otras funciones debido al ámbito de función de las etiquetas).

Hagamos un ejemplo sencillo, factorial.

Aquí hay una versión recursiva que no es TCO, ¡pero puede serlo!

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

int factorial(int n, int a)
{
```

[3] https://en.wikipedia.org/wiki/Tail_call

```
    if (n == 0)
        return a;

    return factorial(n - 1, a * n);
}

int main(void)
{
    for (int i = 0; i < 8; i++)
        printf("%d! == %ld\n", i, factorial(i, 1));
}
```

Para conseguirlo, puedes sustituir la llamada por dos pasos:

1. Establecer los valores de los parámetros a lo que serían en la siguiente llamada.
2. `goto` una etiqueta en la primera línea de la función.

Vamos a probarlo:

```
#include <stdio.h>

int factorial(int n, int a)
{
tco:  // añade esto

    if (n == 0)
        return a;

    // sustituir return por la fijación de nuevos valores de los parámetros y
    // goto-ando el principio de la función
    //return factorial(n - 1, a * n);

    int next_n = n - 1;  // ¿Ves cómo coinciden con
    int next_n = a * n; // ¿los argumentos recursivos, arriba?

    n = next_n;   // Establecer los parámetros a los nuevos valores
    a = next_a;

    goto tco;   // y repite!
}

int main(void)
{
    for (int i = 0; i < 8; i++)
        printf("%d! == %d\n", i, factorial(i, 1));
}
```

Utilicé variables temporales ahí arriba para establecer los siguientes valores de los parámetros antes de saltar al inicio de la función. ¿Ves cómo corresponden a los argumentos recursivos que estaban en la llamada recursiva?

Ahora bien, ¿por qué usar variables temporales? Podría haber hecho esto en su lugar:

```
    a *= n;
    n -= 1;

    goto tco;
```

y eso funciona muy bien. Pero si descuidadamente invierto esas dos líneas de código:

```
    n -= 1;  // MALAS NOTICIAS
    a *= n;
```

—ahora estamos en problemas. Modificamos `n` antes de usarlo para modificar `a`. Eso es malo porque no es así como funciona cuando llamas recursivamente. Usar las variables temporales evita este problema incluso si no estás atento a ello. Y el compilador probablemente las optimiza, de todos modos.

## 31.7 Reinicio de llamadas al sistema interrumpidas

Esto está fuera de la especificación, pero se ve comúnmente en sistemas tipo Unix.

Ciertas llamadas al sistema de larga duración pueden devolver un error si son interrumpidas por una señal, y `errno` será puesto a `EINTR` para indicar que la llamada al sistema estaba funcionando bien; sólo fue interrumpida.

En esos casos, es muy común que el programador quiera reiniciar la llamada e intentarlo de nuevo.

```
retry:
   byte_count = read(0, buf, sizeof(buf) - 1);  // Llamada al sistema Unix read()

    if (byte_count == -1) {            // Se ha producido un error...
        if (errno == EINTR) {          // Pero sólo fue interrumpido
            printf("Restarting...\n");
            goto retry;
        }
```

Muchos Unix-likes tienen una bandera `SA_RESTART` que puede pasar a `sigaction()` para solicitar al SO que reinicie automáticamente cualquier syscall lenta en lugar de fallar con `EINTR`.

De nuevo, esto es específico de Unix y está fuera del estándar C.

Dicho esto, es posible usar una técnica similar cada vez que cualquier función deba ser reiniciada.

## 31.8 `goto` y el Hilo conductor preferente (Thread Preemption)

Este ejemplo está extraído directamente de *Sistemas operativos: Tres piezas fáciles*, otro excelente libro de autores con ideas afines que también consideran que los libros de calidad deben poder descargarse gratuitamente. No es que sea un testarudo, ni nada por el estilo.

```
retry:

    pthread_mutex_lock(L1);

    if (pthread_mutex_trylock(L2) != 0) {
        pthread_mutex_unlock(L1);
        goto retry;
```

```
    }

    save_the_day();

    pthread_mutex_unlock(L2);
    pthread_mutex_unlock(L1);
```

Allí el hilo adquiere felizmente el mutex `L1`, pero entonces falla potencialmente en conseguir el segundo recurso custodiado por el mutex `L2` (si algún otro hilo no cooperativo lo tiene, digamos). Si nuestro hilo no puede conseguir el bloqueo `L2`, desbloquea `L1` y usa `goto` para reintentarlo limpiamente.

Esperamos que nuestra heroica hebra consiga finalmente adquirir ambos mutexes y salvar el día, todo ello evitando el malvado punto muerto.

## 31.9 `goto` y el ámbito de las variables

Ya hemos visto que las etiquetas tienen ámbito de función, pero pueden ocurrir cosas raras si saltamos más allá de la inicialización de alguna variable.

Mira este ejemplo en el que saltamos de un lugar en el que la variable `x` está fuera de ámbito a la mitad de su ámbito (en el bloque).

```
    goto label;

    {
        int x = 12345;

label:
        printf("%d\n", x);
    }
```

Esto compilará y ejecutará, pero me da una advertencia:

```
warning: ‘x’ is used uninitialized in this function
```

Y luego imprime `0` cuando lo ejecuto (su kilometraje puede variar).

Básicamente lo que ha pasado es que hemos saltado al ámbito de `x` (así que estaba bien referenciarlo en `printf()`) pero hemos saltado la línea que realmente lo inicializaba a `12345`. Así que el valor era indeterminado.

La solución es, por supuesto, obtener la inicialización *después* de la etiqueta de una forma u otra.

```
    goto label;

    {
        int x;

label:
        x = 12345;
        printf("%d\n", x);
    }
```

Veamos un ejemplo más.

```
    {
        int x = 10;

label:

        printf("%d\n", x);
    }

    goto label;
```

¿Qué pasa aquí?

La primera vez a través del bloque, estamos bien. `x` es `10` y eso es lo que se imprime.

Pero después del `goto`, saltamos al ámbito de `x`, pero después de su inicialización. Lo que significa que aún podemos imprimirlo, pero el valor es indeterminado (ya que no ha sido reinicializado).

En mi máquina, imprime `10` de nuevo (hasta el infinito), pero eso es sólo suerte. Podría imprimir cualquier valor después del `goto` ya que `x` no está inicializado.

## 31.10 `goto` y matrices de longitud variable (Variable-Length Arrays)

Cuando se trata de VLAs y `goto`, hay una regla: no se puede saltar desde fuera del ámbito de un VLA al ámbito de ese VLA.

Si intento hacer esto:

```
    int x = 10;

    goto label;

    {
        int v[x];

label:

        printf("Hi!\n");
    }
```

Me aparece un error:

```
error: jump into scope of identifier with variably modified type
```

Puede adelantarse así a la declaración del VLA:

```
    int x = 10;

    goto label;

    {
label:  ;
        int v[x];
```

```
        printf("Hi!\n");
    }
```

Porque de ese modo el VLA se asigna correctamente antes de su inevitable desasignación una vez que queda fuera del ámbito de aplicación.

# Chapter 32

# Tipos Parte V: Literales compuestos y selecciones genéricas

Este es el capítulo final de los tipos. Hablaremos de dos cosas:

- Cómo tener objetos "anónimos" sin nombre y cómo eso es útil.
- Cómo generar código dependiente del tipo.

No están particularmente relacionados, pero realmente no merecen cada uno su propio capítulo. Así que los he metido aquí como un rebelde.

## 32.1 Literales compuestos

Esta es una característica del lenguaje que te permite crear un objeto de algún tipo sobre la marcha sin tener que asignarlo a una variable. Puedes crear tipos simples, arrays, `struct`s, lo que quieras.

Uno de los principales usos de esto es pasar argumentos complejos a funciones cuando no quieres crear una variable temporal para mantener el valor.

La forma de crear un literal compuesto es poner el nombre del tipo entre paréntesis, y después poner una lista inicializadora. Por ejemplo, un array sin nombre de `int`s, podría tener este aspecto:

```
(int []){1,2,3,4}
```

Ahora, esa línea de código no hace nada por sí misma. Crea un array sin nombre de 4 `int`s, y luego los tira sin usarlos.

Podríamos usar un puntero para almacenar una referencia al array…

```
int *p = (int []){1 ,2 ,3 ,4};

printf("%d\n", p[1]);  // 2
```

Pero eso parece una forma un poco prolija de tener una matriz. Quiero decir, podríamos haber hecho esto[1]:

```
int p[] = {1, 2, 3, 4};
```

[1] Que no es exactamente lo mismo, ya que es una matriz, no un puntero a un `int`

```
printf("%d\n", p[1]);  // 2
```

Así que veamos un ejemplo más útil.

### 32.1.1 Pasando Objetos sin Nombre a Funciones

Digamos que tenemos una función para sumar un array de `int`s:

```
int sum(int p[], int count)
{
    int total = 0;

    for (int i = 0; i < count; i++)
        total += p[i];

    return total;
}
```

Si quisiéramos llamarla, normalmente tendríamos que hacer algo como esto, declarando un array y almacenando valores en él para pasárselos a la función:

```
int a[] = {1, 2, 3, 4};

int s = sum(a, 4);
```

Pero los objetos sin nombre nos dan una forma de saltarnos la variable pasándola directamente (nombres de parámetros listados arriba). Compruébalo: vamos a sustituir la variable "a" por una matriz sin nombre que pasaremos como primer argumento:

```
//                   p[]         count
//           |-----------------|  |
int s = sum((int []){1, 2, 3, 4}, 4);
```

¡Muy hábil!

### 32.1.2 `struct`s sin nombre

Podemos hacer algo parecido con `struct`s.

Primero, hagamos las cosas sin objetos sin nombre. Definiremos una `struct` para contener algunas coordenadas `x`/`y`. Luego definiremos una, pasando valores a su inicializador. Finalmente, lo pasaremos a una función para imprimir los valores:

```
#include <stdio.h>

struct coord {
    int x, y;
};

void print_coord(struct coord c)
{
    printf("%d, %d\n", c.x, c.y);
```

```
}

int main(void)
{
    struct coord t = {.x=10, .y=20};

    print_coord(t);   // prints "10, 20"
}
```

¿Suficientemente sencillo?

Vamos a modificarlo para utilizar un objeto sin nombre en lugar de la variable `t` que estamos pasando a `print_coord()`.

Quitaremos `t` y la sustituiremos por una `struct` sin nombre:

```
    /estructurar coord t = {.x=10, .y=20};

    print_coord((struct coord){.x=10, .y=20});   // Imprime "10, 20"
```

¡Todavía funciona!

### 32.1.3 Punteros a objetos sin nombre

Puede que hayas notado en el último ejemplo que, aunque estábamos usando una `struct`, estábamos pasando una copia de la `struct` a `print_coord()` en lugar de pasar un puntero a la `struct`.

Resulta que podemos tomar la dirección de un objeto sin nombre con `&` como siempre.

Esto es porque, en general, si un operador hubiera funcionado en una variable de ese tipo, puedes usar ese operador en un objeto sin nombre de ese tipo.

Modifiquemos el código anterior para que pasemos un puntero a un objeto sin nombre

```
#include <stdio.h>

struct coord {
    int x, y;
};

void print_coord(struct coord *c)
{
    printf("%d, %d\n", c->x, c->y);
}

int main(void)
{
    // Nota el &
    //          |
    print_coord(&(struct coord){.x=10, .y=20});   // Imprime "10, 20"
}
```

Además, esto puede ser una buena manera de pasar incluso punteros a objetos simples:

```
// Pasa un puntero a un int con valor 3490
foo(&(int){3490});
```

Así de fácil.

### 32.1.4 Objetos sin nombre y alcance

El tiempo de vida de un objeto sin nombre termina al final de su ámbito. La forma más grave de que esto ocurra es si creas un nuevo objeto sin nombre, obtienes un puntero a él y luego abandonas el ámbito del objeto. En ese caso, el puntero se referirá a un objeto muerto.

Esto es un comportamiento indefinido:

```
int *p;

{
    p = &(int){10};
}

printf("%d\n", *p);  // INVÁLIDO: El (int){10} se ha salido de ámbito
```

Del mismo modo, no se puede devolver un puntero a un objeto sin nombre desde una función. El objeto se desasigna cuando sale del ámbito:

```
#include <stdio.h>

int *get3490(void)
{
    // No hagas esto
    return &(int){3490};
}

int main(void)
{
    printf("%d\n", *get3490());  // INVALID: (int){3490} cayó fuera de ámbito
}
```

Piense en su alcance como en el de una variable local normal. Tampoco puedes devolver un puntero a una variable local.

### 32.1.5 Ejemplo tonto de objeto sin nombre

Puedes poner cualquier tipo y hacer un objeto sin nombre.

Por ejemplo, estos son efectivamente equivalentes:

```
int x = 3490;

printf("%d\n", x);              // 3490 (variable)
printf("%d\n", 3490);           // 3490 (constant)
printf("%d\n", (int){3490});    // 3490 (unnamed object)
```

Esto último no tiene nombre, pero es una tontería. También podría hacer el simple en la línea anterior.

Pero espero que proporciona un poco más de claridad en la sintaxis.

## 32.2 Selecciones genéricas

Se trata de una expresión que permite seleccionar diferentes fragmentos de código en función del *tipo* del primer argumento de la expresión.

Veremos un ejemplo en un segundo, pero es importante saber que esto se procesa en tiempo de compilación, *no en tiempo de ejecución*. No hay ningún análisis en tiempo de ejecución.

La expresión empieza por `_Generic`, funciona como un `switch`, y toma al menos dos argumentos

El primer argumento es una expresión (o variable[2]) que tiene un *tipo*. Todas las expresiones tienen un tipo. El resto de argumentos de `_Generic` son los casos, de qué sustituir en el resultado de la expresión, si el primer argumento es de ese tipo.

¿Qué?

Probemos a ver.

```
#include <stdio.h>

int main(void)
{
    int i;
    float f;
    char c;

    char *s = _Generic(i,
                    int: "that variable is an int",
                    float: "that variable is a float",
                    default: "that variable is some type"
                );

    printf("%s\n", s);
}
```

Fíjate en la expresión `_Generic` que empieza en la línea 9.

Cuando el compilador la ve, mira el tipo del primer argumento. (En este ejemplo, el tipo de la variable `i`.) Luego busca en los casos algo de ese tipo. Y entonces sustituye el argumento en lugar de toda la expresión `_Generic`.

En este caso, `i` es un `int`, por lo que coincide con ese caso. Entonces la cadena es sustituida por la expresión. Así que la línea se convierte en esto cuando el compilador lo ve:

```
    char *s = "that variable is an int";
```

Si el compilador no puede encontrar una coincidencia de tipo en `_Generic`, busca el caso opcional `default` y lo utiliza.

Si no puede encontrar una coincidencia de tipo y no hay “default”, obtendrá un error de compilación. error de compilación. La primera expresión **debe** coincidir con uno de los tipos o con `default`.

Como es inconveniente escribir `_Generic` una y otra vez, se usa a menudo para hacer el cuerpo de una macro que pueda ser fácilmente reutilizada repetidamente.

[2]Una variable utilizada aquí *es* una expresión.

Hagamos una macro `TYPESTR(x)` que toma un argumento y devuelve una cadena con el tipo del argumento.

Así, `TYPESTR(1)` devolverá la cadena `"int"`, por ejemplo.

Allá vamos:

```
#include <stdio.h>

#define TYPESTR(x) _Generic((x), \
                        int: "int", \
                        long: "long", \
                        float: "float", \
                        double: "double", \
                        default: "something else")

int main(void)
{
    int i;
    long l;
    float f;
    double d;
    char c;

    printf("i is type %s\n", TYPESTR(i));
    printf("l is type %s\n", TYPESTR(l));
    printf("f is type %s\n", TYPESTR(f));
    printf("d is type %s\n", TYPESTR(d));
    printf("c is type %s\n", TYPESTR(c));
}
```

Estas salidas:

```
i is type int
l is type long
f is type float
d is type double
c is type something else
```

Lo cual no debería sorprender, porque, como dijimos, ese código en `main()` es reemplazado por lo siguiente cuando se compila:

```
    printf("i is type %s\n", "int");
    printf("l is type %s\n", "long");
    printf("f is type %s\n", "float");
    printf("d is type %s\n", "double");
    printf("c is type %s\n", "something else");
```

Y esa es exactamente la salida que vemos.

Vamos a hacer una más. He incluido algunas macros aquí para que cuando se ejecuta:

```
int i = 10;
char *s = "Foo!";
```

```
PRINT_VAL(i);
PRINT_VAL(s);
```

se obtiene la salida:

```
i = 10
s = Foo!
```

Para ello tendremos que recurrir a la magia de las macros.

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

// Macro que devuelve un especificador de formato para un tipo
#define FMTSPEC(x) _Generic((x), \
                        int: "%d", \
                        long: "%ld", \
                        float: "%f", \
                        double: "%f", \
                        char *: "%s")
                        // TODO: add more types

// Macro que imprime una variable de la forma "nombre = valor"
#define PRINT_VAL(x) do { \
    char fmt[512]; \
    snprintf(fmt, sizeof fmt, #x " = %s\n", FMTSPEC(x)); \
    printf(fmt, (x)); \
} while(0)

int main(void)
{
    int i = 10;
    float f = 3.14159;
    char *s = "Hello, world!";

    PRINT_VAL(i);
    PRINT_VAL(f);
    PRINT_VAL(s);
}
```

para la salida:

```
i = 10
f = 3.141590
s = Hello, world!
```

Podríamos haberlo metido todo en una gran macro, pero lo dividí en dos para evitar el sangrado de los ojos.

# Chapter 33

# Matrices Parte II

En este capítulo vamos a repasar algunas cosas extra relacionadas con los arrays.

- Calificadores de tipo con parámetros de arrays
- La palabra clave `static` con parámetros de arrays
- Inicializadores parciales de arrays multidimensionales

No son muy comunes, pero los veremos ya que son parte de la nueva especificación.

## 33.1 Calificadores de tipo para matrices en listas de parámetros

Si lo recuerdas de antes, estas dos cosas son equivalentes en las listas de parámetros de funciones:

```
int func(int *p) {...}
int func(int p[]) {...}
```

Y puede que también recuerdes que puedes añadir calificadores de tipo a una variable puntero de esta forma:

```
int *const p;
int *volatile p;
int *const volatile p;
// etc.
```

Pero, ¿cómo podemos hacer eso cuando estamos utilizando la notación de matriz en su lista de parámetros?

Resulta que va entre paréntesis. Y puedes poner el recuento opcional después. Las dos líneas siguientes son equivalentes:

```
int func(int *const volatile p) {...}
int func(int p[const volatile]) {...}
int func(int p[const volatile 10]) {...}
```

Si tiene una matriz multidimensional, debe colocar los calificadores de tipo en el primer conjunto de corchetes.

## 33.2 `static` para matrices en listas de parámetros

Del mismo modo, puede utilizar la palabra clave static en la matriz en una lista de parámetros.

Esto es algo que nunca he visto en la naturaleza. Es **siempre** seguido de una dimensión:

```
int func(int p[static 4]) {...}
```

Lo que esto significa, en el ejemplo anterior, es que el compilador va a asumir que cualquier array que pases a la función tendrá *al menos* 4 elementos.

Cualquier otra cosa es un comportamiento indefinido.

```
int func(int p[static 4]) {...}

int main(void)
{
    int a[] = {11, 22, 33, 44};
    int b[] = {11, 22, 33, 44, 55};
    int c[] = {11, 22};

    func(a); // ¡OK! a tiene 4 elementos, el mínimo
    func(b); // ¡Bien! b tiene al menos 4 elementos
    func(c); // ¡Comportamiento indefinido! c tiene menos de 4 elementos!
}
```

Esto básicamente establece el tamaño mínimo de array que puedes tener.

Nota importante: no hay nada en el compilador que te prohíba pasar un array más pequeño. El compilador probablemente no te advertirá, y no lo detectará en tiempo de ejecución.

Poniendo `static` ahí, estás diciendo, “Prometo en doble secreto que nunca pasaré un array más pequeño que este”. Y el compilador dice, “Sí, bien”, y confía en que no lo harás.

Y entonces el compilador puede hacer ciertas optimizaciones de código, con la seguridad de que tú, el programador, siempre harás lo correcto.

## 33.3 Inicializadores equivalentes

C es un poco, digamos, *flexible* cuando se trata de inicializadores de arrays.

Ya hemos visto algo de esto, donde cualquier valor que falte es reemplazado por cero.

Por ejemplo, podemos inicializar un array de 5 elementos a `1,2,0,0,0` con esto:

```
int a[5] = {1, 2};
```

O poner un array completamente a cero con:

```
int a[5] = {0};
```

Pero las cosas se ponen interesantes cuando se inicializan matrices multidimensionales.

Hagamos un array de 3 filas y 2 columnas:

```
int a[3][2];
```

Escribamos algo de código para inicializarlo e imprimir el resultado:

```
#include <stdio.h>

int main(void)
{
    int a[3][2] = {
        {1, 2},
        {3, 4},
        {5, 6}
    };

    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 2; col++)
            printf("%d ", a[row][col]);
        printf("\n");
    }
}
```

Y cuando lo ejecutamos, obtenemos lo esperado:

```
1 2
3 4
5 6
```

Dejemos fuera algunos de los elementos inicializadores y veamos cómo se ponen a cero:

```
    int a[3][2] = {
        {1, 2},
        {3},    // ¡Deja el 4!
        {5, 6}
    };
```

que produce:

```
1 2
3 0
5 6
```

Ahora dejemos todo el último elemento del medio:

```
    int a[3][2] = {
        {1, 2},
        // {3, 4},   // Solo corta todo esto
        {5, 6}
    };
```

Y ahora tenemos esto, que puede que no sea lo que esperas:

```
1 2
5 6
0 0
```

Pero si te paras a pensarlo, sólo proporcionamos inicializadores suficientes para dos filas, por lo que se utilizaron para las dos primeras filas. Y los elementos restantes se inicializaron a cero.

Hasta aquí todo bien. Generalmente, si omitimos partes del inicializador, el compilador pone los elementos correspondientes a “0”.

Pero pongámonos *locos*.

```
    int a[3][2] = { 1, 2, 3, 4, 5, 6 };
```

¿Qué…? Es un array 2D, ¡pero sólo tiene un inicializador 1D!

Resulta que es legal (aunque GCC avisará de ello con las advertencias adecuadas activadas).

Básicamente, lo que hace es empezar a rellenar los elementos de la fila 0, luego la fila 1, luego la fila 2 de izquierda a derecha.

Así que cuando imprimimos, imprime en orden:

```
1 2
3 4
5 6
```

Si dejamos algunos fuera:

```
    int a[3][2] = { 1, 2, 3 };
```

se llenan con “0”:

```
1 2
3 0
0 0
```

Así que si quieres llenar todo el array con `0`, entonces adelante:

```
    int a[3][2] = {0};
```

Pero mi recomendación es que si tienes un array 2D, uses un inicializador 2D. Hace el código más legible. (Excepto para inicializar todo el array con `0`, en cuyo caso es idiomático usar `{0}` sin importar la dimensión del array).

# Chapter 34

# Saltos largos con `setjmp`, `longjmp`

Ya hemos visto `goto`, que salta en el ámbito de la función. Pero `longjmp()` te permite saltar a un punto anterior en la ejecución, a una función que llamó a ésta.

Hay muchas limitaciones y advertencias, pero puede ser una función útil para saltar desde lo profundo de la pila de llamadas a un estado anterior.

En mi experiencia, esta funcionalidad se utiliza muy raramente.

## 34.1 Usando `setjmp` y `longjmp`

El baile que vamos a hacer aquí es básicamente poner un marcador en ejecución con `setjmp()`. Más tarde, llamaremos a `longjmp()` y volverá al punto anterior de la ejecución donde pusimos el marcador con `setjmp()`.

Y puede hacer esto incluso si has llamado a subfunciones.

Aquí hay una demostración rápida donde llamamos a funciones de un par de niveles de profundidad y luego salimos de ellas.

Vamos a usar una variable de ámbito de fichero `env` para mantener el *estado* de las cosas cuando llamemos a `setjmp()` de forma que podamos restaurarlas cuando llamemos a `longjmp()` más tarde. Esta es la variable en la que recordamos nuestro "lugar".

La variable `env` es de tipo `jmp_buf`, un tipo opaco declarado en `<setjmp.h>`.

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

jmp_buf env;

void depth2(void)
{
    printf("Entering depth 2\n");
    longjmp(env, 3490);           // Libertad bajo fianza
    printf("Leaving depth 2\n");  // Esto no sucederá
}

void depth1(void)
{
```

```
    printf("Entering depth 1\n");
    depth2();
    printf("Leaving depth 1\n");  // Esto no sucederá
}

int main(void)
{
    switch (setjmp(env)) {
      case 0:
          printf("Calling into functions, setjmp() returned 0\n");
          depth1();
          printf("Returned from functions\n"); // Esto no sucederá
          break;

      case 3490:
          printf("Bailed back to main, setjmp() returned 3490\n");
          break;
    }
}
```

Cuando se ejecuta, esta salida:

```
Calling into functions, setjmp() returned 0
Entering depth 1
Entering depth 2
Bailed back to main, setjmp() returned 3490
```

Si intentas cotejar esa salida con el código, está claro que están pasando cosas realmente *funky*.

Una de las cosas más notables es que `setjmp()` devuelve *twice*. ¿Qué demonios? ¡¿Qué es esta brujería?!

Así que esto es lo que pasa: si `setjmp()` devuelve `0`, significa que has establecido con éxito el "marcador" en ese punto.

Si devuelve un valor distinto de cero, significa que has vuelto al "marcador" establecido anteriormente. (Y el valor devuelto es el que pasas a `longjmp()`.)

De esta forma puedes diferenciar entre establecer el marcador y volver a él más tarde.

Así que cuando el código de arriba llama a `setjmp()` la primera vez, `setjmp()` *almacena* el estado en la variable `env` y devuelve `0`. Más tarde, cuando llamamos a `longjmp()` con ese mismo `env`, se restaura el estado y `setjmp()` devuelve el valor que se le pasó a `longjmp()`.

## 34.2 Errores (Pitfalls)

Bajo el capó, esto es bastante sencillo. Normalmente, el *puntero de pila* mantiene un registro de las ubicaciones en memoria en las que se almacenan las variables locales, y el *contador de programa* mantiene un registro de la dirección de la instrucción que se está ejecutando en ese momento[1].

Así que si queremos saltar de nuevo a una función anterior, es básicamente sólo una cuestión de restaurar el puntero de la pila y el contador de programa a los valores guardados en la variable . `jmp_buf`, y asegurarse de que el valor de retorno se establece correctamente. Y entonces la ejecución se reanudará allí.

[1]Tanto el "puntero de pila" como el "contador de programa" están relacionados con la arquitectura subyacente y la implementación de C, y no forman parte de la especificación

Pero una variedad de factores confunden esto, haciendo un número significativo de trampas de comportamiento indefinido.

### 34.2.1 Los valores de las variables locales

Si quieres que los valores de las variables locales automáticas (no `static` y no `extern`) persistan en la función que llamó a `setjmp()` después de que ocurra un `longjmp()`, debe declarar que esas variables son `volatile`.

Técnicamente, sólo tienen que ser `volátiles` si cambian entre el momento en que se llama a `setjmp()` y se llama a `longjmp()`[2].

Por ejemplo, si ejecutamos este código

```
int x = 20;

if (setjmp(env) == 0) {
    x = 30;
}
```

y luego `longjmp()` de vuelta, el valor de `x` será indeterminado.

Si queremos solucionar esto, `x` debe ser `volátil`:

```
volatile int x = 20;

if (setjmp(env) == 0) {
    x = 30;
}
```

Ahora el valor será el correcto `30` después de que un `longjmp()` nos devuelve a este punto.

### 34.2.2 ¿Cuánto Estado se ahorra?

Cuando usted `longjmp()`, la ejecución se reanuda en el punto del `setjmp()` correspondiente. Y eso es todo.

La especificación señala que es como si hubieras vuelto a la función en ese punto con las variables locales establecidas a los valores que tenían cuando se hizo la llamada a `longjmp()`.

Las cosas que no se restauran incluyen, parafraseando la especificación:

- Banderas de estado de punto flotante
- Archivos abiertos
- Cualquier otro componente de la máquina abstracta

### 34.2.3 No Puedes Nombrar Nada `setjmp`

No puedes tener ningún identificador `extern` con el nombre `setjmp`. O, si `setjmp` es una macro, no puedes redefinirla.

Ambos son comportamientos indefinidos.

[2]La razón aquí es que el programa puede almacenar un valor temporalmente en un *registro de la CPU* mientras está trabajando en él. En ese intervalo de tiempo, el registro contiene el valor correcto, y el valor en la pila podría estar desfasado. Entonces más tarde los valores del registro se sobrescribirían y los cambios en la variable se perderían

### 34.2.4 No Puede `setjmp()` en una Expresión Mayor

Es decir, no puedes hacer algo así:

```
if (x == 12 && setjmp(env) == 0) { ... }
```

Eso es demasiado complejo para que lo permita la especificación debido a las maquinaciones que deben ocurrir al desenrollar la pila y todo eso. No podemos `longjmp()` volver a una expresión compleja que sólo se ha ejecutado parcialmente.

Así que hay límites en la complejidad de esa expresión.

- Puede ser toda la expresión controladora de la condicional.

  ```
  if (setjmp(env)) {...}
  ```

  ```
  switch (setjmp(env)) {...}
  ```

- Puede formar parte de una expresión relacional o de igualdad, siempre que el otro operando sea una constante entera. Y el entero es la expresión controladora del condicional.

  ```
  if (setjmp(env) == 0) {...}
  ```

- El operando de una operación lógica NOT (`!`), siendo toda la expresión controladora.

  ```
  if (!setjmp(env)) {...}
  ```

- Una expresión independiente, posiblemente convertida en `void`.

  ```
  setjmp(env);
  ```

  ```
  (void)setjmp(env);
  ```

### 34.2.5 ¿Cuándo no se puede `longjmp()`?

Es un comportamiento indefinido si:

- No has llamado antes a `setjmp()`.
- Llamó a `setjmp()` desde otro thread
- Llamó a `setjmp()` en el ámbito de un array de longitud variable (VLA), y la ejecución dejó el ámbito de ese VLA antes de que `longjmp()` fuera llamado.
- La función que contiene `setjmp()` salió antes de que `longjmp()` fuera llamada.

En este último caso, “salió” incluye los retornos normales de la función, así como el caso de que otro `longjmp()` saltara “antes” en la pila de llamadas que la función en cuestión.

### 34.2.6 No se puede pasar `0` a `longjmp()`.

Si intenta pasar el valor `0` a `longjmp()`, cambiará silenciosamente ese valor a `1`.

Dado que `setjmp()` devuelve este valor, y que `setjmp()` devuelve `0` tiene un significado especial, devolver `0` está prohibido.

### 34.2.7 `longjmp()` y los Arrays de Longitud Variable

Si estás en el ámbito de un VLA y haces `longjmp()` fuera de él, la memoria asignada al VLA podría tener fugas[3].

Lo mismo ocurre si haces `longjmp()` hacia atrás sobre cualquier función anterior que tuviera VLAs todavía en scope.

Esto es algo que realmente me molestaba de los VLAs—que podías escribir código C perfectamente legítimo que derrochaba memoria. Pero, oye… yo no estoy a cargo de la especificación.

[3]Es decir, permanecer asignada hasta que el programa termine sin que haya forma de liberarla

# Chapter 35

# Tipos incompletos

Puede que le sorprenda saber que esto se construye sin errores:

```
extern int a[];

int main(void)
{
    struct foo *x;
    union bar *y;
    enum baz *z;
}
```

Nunca hemos dado un tamaño para "a". Y tenemos punteros a `struct`s `foo`, `bar`, y `baz` que nunca parecen estar declarados en ninguna parte.

Y las únicas advertencias que recibo son que `x`, `y`, y `z` no se usan.

Estos son ejemplos de *tipos incompletos*.

Un tipo incompleto es un tipo cuyo tamaño (es decir, el tamaño que obtendrías de `sizeof`) no se conoce. Otra forma de verlo es un tipo que no has terminado de declarar.

Puedes tener un puntero a un tipo incompleto, pero no puedes desreferenciarlo o usar aritmética de punteros en él. Y no se puede `sizeof`.

¿Qué puedes hacer con él?

## 35.1 Caso práctico: estructuras autorreferenciales

Sólo conozco un caso de uso real: referencias hacia adelante a `struct`s o `union`s con estructuras autorreferenciales o codependientes. (Voy a utilizar `struct` para el resto de estos ejemplos, pero todos se aplican igualmente a `union`s, también).

Hagamos primero el ejemplo clásico.

Pero antes, ¡ten esto en cuenta! Cuando declaras una `struct`, ¡la `struct` está incompleta hasta que se alcanza la llave de cierre!

```
struct antelope {             // struct antelope está incompleto aquí
    int leg_count;            // Aún incompleto
```

```
    float stomach_fullness;    // Aún incompleto
    float top_speed;           // Aún incompleto
    char *nickname;            // Aún incompleto
};                             // AHORA está completo.
```

¿Y qué? Parece bastante sensato.

¿Pero qué pasa si estamos haciendo una lista enlazada? Cada nodo de la lista enlazada necesita tener una referencia a otro nodo. ¿Pero cómo podemos crear una referencia a otro nodo si ni siquiera hemos terminado de declarar el nodo?

C permite tipos incompletos. No podemos declarar un nodo, pero *podemos* declarar un puntero a uno, ¡incluso si está incompleto!

```
struct node {
    int val;
    struct node *next;  // El nodo struct está incompleto, ¡pero no pasa nada!
};
```

Aunque el nodo `struct` está incompleto en la línea 3, aún podemos declarar un puntero a uno[1].

Podemos hacer lo mismo si tenemos dos `struct`s diferentes que se refieren la una a la otra:

```
struct a {
    struct b *x;  // Se refiere a una `estructura b`
};

struct b {
    struct a *x;  // Se refiere a una `estructura a`.
};
```

Nunca seríamos capaces de hacer ese par de estructuras sin las reglas relajadas para tipos incompletos.

## 35.2 Mensajes de error de tipo incompleto

¿Recibe errores como éstos?

```
invalid application of ‘sizeof’ to incomplete type

invalid use of undefined type

dereferencing pointer to incomplete type
```

Culpable más probable: probablemente olvidó `#incluir` el fichero de cabecera que declara el tipo.

## 35.3 Otros tipos incompletos

Declarar una `struct` o `union` sin cuerpo hace un tipo incompleto, por ejemplo `struct foo;`.

Los `enums` son incompletos hasta la llave de cierre.

[1] Esto funciona porque en C, los punteros tienen el mismo tamaño independientemente del tipo de datos al que apunten. Así que el compilador no necesita saber el tamaño del nodo `struct` en este punto; sólo necesita saber el tamaño de un puntero

Los `void` son tipos incompletos.

Los arrays declarados `extern` sin tamaño son incompletos, p.e.:

```
extern int a[];
```

Si es un array no `externo` sin tamaño seguido de un inicializador, está incompleto hasta la llave de cierre del inicializador.

## 35.4 Caso de Uso: Arrays en ficheros de cabecera

Puede ser útil declarar tipos de array incompletos en ficheros de cabecera. En esos casos, el almacenamiento real (donde se declara el array completo) debería estar en un único fichero `.c`. Si lo pones en el fichero `.h`, se duplicará cada vez que se incluya el fichero de cabecera.

Así que lo que puedes hacer es crear un fichero de cabecera con un tipo incompleto que haga referencia al array, así:

```
// File: bar.h

#ifndef BAR_H
#define BAR_H

extern int my_array[];  // Tipo incompleto

#endif
```

Y el en el archivo `.c`, en realidad definir la matriz:

```
// File: bar.c

int my_array[1024];     // ¡Tipo completo!
```

A continuación, puede incluir el encabezado de tantos lugares como desee, y cada uno de esos lugares se refieren a la misma subyacente `my_array`.

```
// File: foo.c

#include <stdio.h>
#include "bar.h"    // incluye el tipo incompleto para mi_array

int main(void)
{
    my_array[0] = 10;

    printf("%d\n", my_array[0]);
}
```

Cuando compile varios archivos, recuerde especificar todos los archivos `.c` al compilador, pero no los archivos `.h`, p. ej:

```
gcc -o foo foo.c bar.c
```

## 35.5 Completar tipos incompletos

Si tienes un tipo incompleto, puedes completarlo definiendo el `struct`, `union`, `enum`, o array completo en el mismo ámbito.

```
struct foo;        // tipo incompleto

struct foo *p;     // puntero, no hay problema

// struct foo f;   // Error: ¡tipo incompleto!

struct foo {
    int x, y, z;
}; // ¡Ahora la estructura foo está completa!

struct foo f;      // ¡Éxito!
```

Ten en cuenta que aunque `void` es un tipo incompleto, no hay forma de completarlo. No es que a nadie se le ocurra hacer esa cosa rara. Pero explica por qué se puede hacer esto:

```
void *p;             // OK: puntero a tipo incompleto
```

y no ninguno de estos:

```
void v;              // Error: declarar variable de tipo incompleto

printf("%d\n", *p);  // Error: referencia a un tipo incompleto
```

Cuanto más sepas…

# Chapter 36

# Números complejos

Un pequeño manual sobre Números complejos[1] robado directamente de Wikipedia:

> Un **número complejo** es un número que puede expresarse de la forma $a+bi$, donde $a$ y $b$ son números reales [es decir, tipos de coma flotante en C], y $i$ representa la unidad imaginaria, satisfaciendo la ecuación $i^2 = -1$. Dado que ningún número real satisface esta ecuación, $i$ se denomina número imaginario. Para el número complejo $a + bi$, $a$ se denomina se llama **parte real**, y $b$ se llama **parte imaginaria**.

Pero hasta aquí voy a llegar. Asumiremos que si estás leyendo este capítulo, sabes lo que es un número complejo y lo que quieres hacer con ellos.

Y todo lo que necesitamos cubrir son las facultades de C para hacerlo.

Resulta, sin embargo, que el soporte de números complejos en un compilador es una característica *opcional*. No todos los compiladores compatibles pueden hacerlo. Y los que lo hacen, puede que lo hagan con distintos grados de completitud.

Puedes comprobar si tu sistema soporta números complejos con:

```
#ifdef __STDC_NO_COMPLEX__
#error Complex numbers not supported!
#endif
```

Además, hay una macro que indica la adhesión a la norma ISO 60559 (IEEE 754) para matemáticas en coma flotante con números complejos, así como la presencia del tipo `_Imaginary`.

```
#if __STDC_IEC_559_COMPLEX__ != 1
#error Need IEC 60559 complex support!
#endif
```

Encontrará más información al respecto en el Anexo G de la especificación C11.

## 36.1 Tipos complejos

Para usar números complejos, `#include <complex.h>`.

Con eso, se obtienen al menos dos tipos:

[1] https://en.wikipedia.org/wiki/Complex_number

```
_Complex
complex
```

Ambos significan lo mismo, por lo que es mejor utilizar el más bonito `complex`.

También dispone de algunos tipos para números imaginarios si su aplicación cumple la norma IEC 60559:

```
_Imaginary
imaginary
```

Ambos significan lo mismo, así que puedes usar el más bonito `imaginary`.

También se obtienen valores para el propio número imaginario $i$:

```
I
_Complex_I
_Imaginary_I
```

La macro `I` se establece en `_Imaginario_I` (si está disponible), o `_I_Complejo`. Así que sólo tiene que utilizar `I` para el número imaginario.

Un inciso: he dicho que si un compilador tiene `__STDC_IEC_559_COMPLEX__` a `1`, debe soportar tipos `_Imaginary` para ser compatible. Esa es mi lectura de la especificación. Sin embargo, no conozco ningún compilador que soporte `_Imaginary` aunque tenga `__STDC_IEC_559_COMPLEX__`. Así que voy a escribir algo de código con ese tipo que no tengo forma de probar. Lo siento.

Bien, ahora que sabemos que existe un tipo `complejo`, ¿cómo podemos usarlo?

## 36.2 Asignando Números Complejos

Dado que el número complejo tiene una parte real y otra imaginaria, pero ambas se basan en números de coma flotante para almacenar valores, también tenemos que decirle a C qué precisión utilizar para esas partes del número complejo.

Para ello, basta con fijar un `float`, un `double` o un `long double` al `complex`, antes o después de él.

Definamos un número complejo que utilice `float` para sus componentes:

```
float complex c; // Spec prefiere esta forma
float complex c; // Lo mismo--el orden no importa
```

Eso está muy bien para las declaraciones, pero ¿cómo las inicializamos o asignamos?

Resulta que podemos utilizar una notación bastante natural. Ejemplo

```
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;
```

Para 5+2i$ y 10+3i$, respectivamente.

## 36.3 Construir, deconstruir e imprimir

Ya estamos llegando…

Ya hemos visto una forma de escribir un número complejo:

```
double complex x = 5 + 2*I;
```

Tampoco hay problema en utilizar otros números de coma flotante para construirlo:

```
double a = 5;
double b = 2;
double complex x = a + b*I;
```

También hay un conjunto de macros para ayudar a construir estos. El código anterior podría escribirse utilizando la macro `CMPLX()`, así:

```
double complex x = CMPLX(5, 2);
```

Según mis investigaciones, son casi equivalentes:

```
double complex x = 5 + 2*I;
double complex x = CMPLX(5, 2);
```

Pero la macro `CMPLX()` tratará siempre correctamente los ceros negativos en la parte imaginaria, mientras que la otra forma podría convertirlos en ceros positivos. Yo *pienso*[2]. Esto parece implicar que si existe la posibilidad de que la parte imaginaria sea cero, debería usar la macro… ¡pero que alguien me corrija si me equivoco!

La macro `CMPLX()` funciona con tipos `doble`. Hay otras dos macros para `float` y `long double`: `CMPLXF()` y `CMPLXL()`. (Estos sufijos "f" y "l" aparecen en prácticamente todas las funciones relacionadas con los números complejos).

Ahora intentemos lo contrario: si tenemos un número complejo, ¿cómo lo descomponemos en sus partes real e imaginaria?

Aquí tenemos un par de funciones que extraerán las partes real e imaginaria del número: `creal()` y `cimag()`:

```
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;

printf("x = %f + %fi\n", creal(x), cimag(x));
printf("y = %f + %fi\n", creal(y), cimag(y));
```

para la salida:

```
x = 5.000000 + 2.000000i
y = 10.000000 + 3.000000i
```

Tenga en cuenta que la `i` que tengo en la cadena de formato `printf()` es una `i` literal que se imprime—no es parte del especificador de formato. Ambos valores devueltos por `creal()` y `cimag()` son `double`.

Y como siempre, hay variantes `float` y `long double` de estas funciones: `crealf()`, `cimagf()`, `creall()`, and `cimagl()`.

[2]Esto ha sido más difícil de investigar, y aceptaré cualquier información adicional que alguien pueda darme. que `I` podría definirse como `_Complex_I` o `_Imaginary_I`, si este último existe. `_Imaginary_I` manejará ceros con signo, pero `_Complex_I` *puede* que no. Esto tiene implicaciones con los cortes de rama y otras cosas de números complejos. Tal vez. ¿Te das cuenta de que me estoy saliendo de mi elemento? En cualquier caso, las macros `CMPLX()` se comportan como si `I` estuviera definido como `_Imaginary_I`, con ceros con signo, aunque `_Imaginary_I` no exista en el sistema

## 36.4 Aritmética compleja y comparaciones

Es posible realizar operaciones aritméticas con números complejos, aunque su funcionamiento matemático queda fuera del alcance de esta guía.

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

int main(void)
{
    double complex x = 1 + 2*I;
    double complex y = 3 + 4*I;
    double complex z;

    z = x + y;
    printf("x + y = %f + %fi\n", creal(z), cimag(z));

    z = x - y;
    printf("x - y = %f + %fi\n", creal(z), cimag(z));

    z = x * y;
    printf("x * y = %f + %fi\n", creal(z), cimag(z));

    z = x / y;
    printf("x / y = %f + %fi\n", creal(z), cimag(z));
}
```

por un resultado de:

```
x + y = 4.000000 + 6.000000i
x - y = -2.000000 + -2.000000i
x * y = -5.000000 + 10.000000i
x / y = 0.440000 + 0.080000i
```

También puede comparar dos números complejos para la igualdad (o desigualdad):

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

int main(void)
{
    double complex x = 1 + 2*I;
    double complex y = 3 + 4*I;

    printf("x == y = %d\n", x == y);  // 0
    printf("x != y = %d\n", x != y);  // 1
}
```

con la salida:

```
x == y = 0
x != y = 1
```

Son iguales si ambos componentes prueban igual. Tenga en cuenta que, al igual que ocurre con todas las operaciones en coma flotante, podrían ser iguales si se aproximan lo suficiente debido a un error de redondeo[3].

## 36.5 Matemáticas complejas

Pero, ¡espera! Hay mucho más que simple aritmética compleja.

Aquí tienes una tabla resumen de todas las funciones matemáticas disponibles con números complejos.

Sólo voy a listar la versión `doble` de cada función, pero para todas ellas hay una versión `float` que puedes obtener añadiendo `f` al nombre de la función, y una versión `long double` que puedes obtener añadiendo `l`.

Por ejemplo, la función `cabs()` para calcular el valor absoluto de un número complejo también tiene variantes `cabsf()` y `cabsl()`. Las omito por brevedad.

### 36.5.1 Funciones de trigonometría

| Function | Description |
|---|---|
| `ccos()` | Coseno |
| `csin()` | Seno |
| `ctan()` | Tangente |
| `cacos()` | Arco coseno |
| `casin()` | Arco seno |
| `catan()` | Jugar a *Settlers of Catan* |
| `ccosh()` | Coseno hiperbólico |
| `csinh()` | Hyperbolic sine |
| `ctanh()` | Tangente hiperbólica |
| `cacosh()` | Arco coseno hiperbólico |
| `casinh()` | Arco seno hiperbólico |
| `catanh()` | Arco hiperbólico tangente |

### 36.5.2 Funciones exponenciales y logarítmicas

| Función | Descripción |
|---|---|
| `cexp()` | Base-$e$ exponente |
| `clog()` | Logaritmo natural (base-$e$) |

### 36.5.3 Funciones de potencia y valor absoluto

| Función | Descripción |
|---|---|
| `cabs()` | Valor absoluto |
| `cpow()` | Potencia |
| `csqrt()` | Raíz cuadrada |

[3] La simplicidad de esta afirmación no hace justicia a la increíble cantidad de trabajo que supone simplemente entender cómo funciona realmente la coma flotante. https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

### 36.5.4 Funciones de manipulación

| Función | Descripción |
|---|---|
| `creal()` | Devolver parte real |
| `cimag()` | Devolver parte imaginaria |
| `CMPLX()` | Construir un número complejo |
| `carg()` | Argumento/ángulo de fase |
| `conj()` | Conjugar[4] |
| `cproj()` | Proyección sobre la esfera de Riemann |

[4] Este es el único que no comienza con una “c” extra, extrañamente.

# Chapter 37

# Tipos enteros de anchura fija

C tiene todos esos tipos de enteros pequeños, grandes y más grandes como `int` y `long` y todo eso. Y puedes mirar en la sección sobre límites para ver cuál es el int más grande con `INT_MAX` y así sucesivamente.

¿Qué tamaño tienen esos tipos? Es decir, ¿cuántos bytes ocupan? Podríamos usar `sizeof` para obtener esa respuesta.

Pero, ¿y si quisiera ir por otro camino? ¿Y si necesitara un tipo que tuviera exactamente 32 bits (4 bytes) o al menos 16 bits o algo así?

¿Cómo podemos declarar un tipo que tenga un tamaño determinado? La cabecera `<stdint.h>` nos da una manera.

## 37.1 Tipos con tamaño de bit

Tanto para los enteros con signo como para los enteros sin signo, podemos especificar un tipo que tenga un cierto número de bits, con algunas salvedades, por supuesto.

Y hay tres clases principales de estos tipos (en estos ejemplos, el `N` sería reemplazado por un cierto número de bits):

- Enteros de exactamente un cierto tamaño (`intN_t`)
- Enteros que tienen al menos un cierto tamaño (`int_leastN_t`)
- Enteros que tienen al menos un cierto tamaño y son lo más rápidos posible (`int_fastN_t`)[1]

¿Cuánto más rápido es `fast`? Definitivamente, quizás algo más rápido. Probablemente. La especificación no dice cuánto más rápido, sólo que será el más rápido en esta arquitectura. Sin embargo, la mayoría de los compiladores de C son bastante buenos, por lo que probablemente sólo lo veas usado en lugares donde se necesita garantizar la mayor velocidad posible (en lugar de simplemente esperar que el compilador produzca un código bastante rápido, como es el caso).

Finalmente, estos tipos de números sin signo tienen una “u” inicial para diferenciarlos.

Por ejemplo, estos tipos tienen el significado listado correspondiente:

```
int32_t w; // x es exactamente 32 bits, con signo
uint16_t x; // y es exactamente 16 bits, sin signo
```

[1] Algunas arquitecturas tienen datos de distinto tamaño con los que la CPU y la RAM pueden operar a mayor velocidad que con otros. En esos casos, si necesitas el número de 8 bits más rápido, puede que te de un tipo de 16 o 32 bits en su lugar porque simplemente es más rápido. Así que con esto, no sabrás lo grande que es el tipo, pero será al menos tan grande como tú digas.

```
int_least8_t y; // y es de al menos 8 bits, con signo

uint_fast64_t z; //z es la representación más rápida
                 // sin signo de al menos 64 bits.
```

Se garantiza la definición de los siguientes tipos:

```
int_least8_t      uint_least8_t
int_least16_t     uint_least16_t
int_least32_t     uint_least32_t
int_least64_t     uint_least64_t

int_fast8_t       uint_fast8_t
int_fast16_t      uint_fast16_t
int_fast32_t      uint_fast32_t
int_fast64_t      uint_fast64_t
```

También puede haber otros de diferentes anchos, pero son opcionales.

¿Dónde están los tipos fijos como `int16_t`? Resulta que son completamente opcionales… a menos que se cumplan ciertas condiciones[2]. Y si tienes un sistema informático moderno normal y corriente, probablemente se cumplan esas condiciones. Y si lo son, tendrás estos tipos:

```
int8_t      uint8_t
int16_t     uint16_t
int32_t     uint32_t
int64_t     uint64_t
```

Pueden definirse otras variantes con anchuras diferentes, pero son opcionales.

## 37.2 Tipo de tamaño entero máximo

Hay un tipo que puedes usar que contiene los enteros representables más grandes disponibles en el sistema, tanto con signo como sin signo:

```
intmax_t
uintmax_t
```

Utilice estos tipos cuando quiera ir lo más grande posible.

Obviamente los valores de cualquier otro tipo entero del mismo signo cabrán en este tipo, necesariamente.

## 37.3 Uso de Constantes de Tamaño Fijo

Si tienes una constante que quieres que quepa en un cierto número de bits, puedes usar estas macros para añadir automáticamente el sufijo apropiado al número (por ejemplo `22L` o `3490ULL`).

```
INT8_C(x)     UINT8_C(x)
INT16_C(x)    UINT16_C(x)
```

[2] Es decir, que el sistema tenga enteros de 8, 16, 32 o 64 bits sin relleno que utilicen la representación del complemento a dos, en cuyo caso la variante `intN_t` para ese número concreto de bits *debe* estar definida

```
INT32_C(x)    UINT32_C(x)
INT64_C(x)    UINT64_C(x)
INTMAX_C(x)   UINTMAX_C(x)
```

De nuevo, sólo funcionan con valores enteros constantes.

Por ejemplo, podemos utilizar uno de estos para asignar valores constantes así:

```
uint16_t x = UINT16_C(12);
intmax_t y = INTMAX_C(3490);
```

## 37.4 Límites de enteros de tamaño fijo

También tenemos definidos algunos límites para que puedas obtener los valores máximos y mínimos de estos tipos:

```
INT8_MAX           INT8_MIN           UINT8_MAX
INT16_MAX          INT16_MIN          UINT16_MAX
INT32_MAX          INT32_MIN          UINT32_MAX
INT64_MAX          INT64_MIN          UINT64_MAX

INT_LEAST8_MAX     INT_LEAST8_MIN     UINT_LEAST8_MAX
INT_LEAST16_MAX    INT_LEAST16_MIN    UINT_LEAST16_MAX
INT_LEAST32_MAX    INT_LEAST32_MIN    UINT_LEAST32_MAX
INT_LEAST64_MAX    INT_LEAST64_MIN    UINT_LEAST64_MAX

INT_FAST8_MAX      INT_FAST8_MIN      UINT_FAST8_MAX
INT_FAST16_MAX     INT_FAST16_MIN     UINT_FAST16_MAX
INT_FAST32_MAX     INT_FAST32_MIN     UINT_FAST32_MAX
INT_FAST64_MAX     INT_FAST64_MIN     UINT_FAST64_MAX

INTMAX_MAX         INTMAX_MIN         UINTMAX_MAX
```

Tenga en cuenta que `MIN` para todos los tipos sin signo es `0`, por lo que, como tal, no hay macro para ello.

## 37.5 Especificadores de formato

Para imprimir estos tipos, necesita enviar el especificador de formato correcto a `printf()`. (Y lo mismo para obtener la entrada con la función `scanf()`). `scanf()`.)

Pero, ¿cómo vas a saber qué tamaño tienen los tipos bajo el capó? Por suerte, una vez más, C proporciona algunas macros para ayudar con esto.

Todo esto se puede encontrar en `<inttypes.h>`.

Ahora, tenemos un montón de macros. Como una explosión de complejidad de macros. Así que voy a dejar de enumerar cada una y sólo voy a poner la letra minúscula `n` en el lugar donde deberías poner `8`, `16`, `32`, o `64` dependiendo de tus necesidades.

Veamos las macros para imprimir enteros con signo:

```
PRIdn    PRIdLEASTn    PRIdFASTn    PRIdMAX
PRIin    PRIiLEASTn    PRIiFASTn    PRIiMAX
```

Busque allí los patrones. Puedes ver que hay variantes para los tipos fijo, mínimo, rápido y máximo.

Y también tienes una "d" minúscula y una "i" minúscula. Corresponden a los especificadores de formato `printf()` `%d` y `%i`.

Así que si tengo algo de tipo

```
int_least16_t x = 3490;
```

Puedo imprimirlo con el especificador de formato equivalente para `%d` utilizando `PRIdLEAST16`.

¿Pero cómo? ¿Cómo usamos esa macro?

En primer lugar, esa macro especifica una cadena que contiene la letra o letras que `printf()` necesita usar para imprimir ese tipo. Como, por ejemplo, podría ser `"d"` o `"ld"`.

Así que todo lo que tenemos que hacer es incrustar eso en nuestra cadena de formato para la llamada a `printf()`.

Para ello, podemos aprovechar un hecho sobre C que puede que hayas olvidado: las cadenas literales adyacentes se concatenan automáticamente en una sola cadena. Por ejemplo

```
printf("Hello, " "world!\n");   // Imprime "Hello, world!"
```

Y como estas macros son literales de cadena, podemos usarlas así:

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

int main(void)
{
    int_least16_t x = 3490;

    printf("The value is %" PRIdLEAST16 "!\n", x);
}
```

También tenemos un montón de macros para imprimir tipos sin signo:

```
PRIon    PRIoLEASTn    PRIoFASTn    PRIoMAX
PRIun    PRIuLEASTn    PRIuFASTn    PRIuMAX
PRIxn    PRIxLEASTn    PRIxFASTn    PRIxMAX
PRIXn    PRIXLEASTn    PRIXFASTn    PRIXMAX
```

En este caso, `o`, `u`, `x`, y `X` corresponden a los especificadores de formato documentados en `printf()`.

Y, como antes, la minúscula `n` debe sustituirse por `8`, `16`, `32`, o `64`.

Pero justo cuando crees que ya has tenido suficiente con las macros, resulta que tenemos un completo conjunto complementario de ellas para ¡`scanf()`!

```
SCNdn    SCNdLEASTn    SCNdFASTn    SCNdMAX
SCNin    SCNiLEASTn    SCNiFASTn    SCNiMAX
SCNon    SCNoLEASTn    SCNoFASTn    SCNoMAX
SCNun    SCNuLEASTn    SCNuFASTn    SCNuMAX
SCNxn    SCNxLEASTn    SCNxFASTn    SCNxMAX
```

Recuerde: cuando quiera imprimir un tipo entero de tamaño fijo con `printf()` o `scanf()`, tome la especificación de formato correspondiente de `<inttypes.h>`.

# Chapter 38

# Funciones de fecha y hora

> "El tiempo es una ilusión. La hora de comer doblemente".
> —Ford Prefect, La guía del autoestopista galáctico

Esto no es demasiado complejo, pero puede ser un poco intimidante al principio, tanto con los diferentes tipos disponibles y la forma en que podemos convertir entre ellos.

Mezcla GMT (UTC) y la hora local y tenemos toda la *Usual Fun*™ que uno consigue con horas y fechas.

Y, por supuesto, nunca olvides la regla de oro de las fechas y horas: *Nunca intentes escribir tu propia funcionalidad de fecha y hora. Sólo usa lo que te da la librería.*

El tiempo es demasiado complejo para que los simples programadores mortales lo manejen correctamente. En serio, todos debemos un punto a todos los que trabajaron en cualquier librería de fecha y hora, así que ponlo en tu presupuesto.

## 38.1 Terminología rápida e información

Sólo un par de términos rápidos en caso de que no los tengas apuntados. * **UTC**: El Tiempo Universal Coordinado (Universal Time Coordinated) es una hora absoluta acordada universalmente[1]. Todo el mundo en el planeta piensa que es la misma hora ahora mismo en UTC… aunque tengan diferentes horas locales.

- **GMT**: Hora del meridiano de Greenwich, efectivamente la misma que la UTC[2]. Probablemente quieras decir UTC, o "hora universal". Si te refieres específicamente a la zona horaria GMT, di GMT. Confusamente, muchas de las funciones UTC de C son anteriores a UTC y todavía se refieren a la hora media de Greenwich. Cuando vea eso, sepa que C quiere decir UTC.
- **Hora local**: qué hora es en el lugar donde se encuentra el ordenador que ejecuta el programa. Se describe como un desfase con respecto a UTC. Aunque hay muchas zonas horarias en el mundo, la mayoría de los ordenadores trabajan en hora local o UTC.

Como regla general, si estás describiendo un evento que ocurre una sola vez, como una entrada de registro, o el lanzamiento de un cohete, o cuando los punteros finalmente hicieron clic para ti, utiliza UTC.

En cambio, si se trata de algo que ocurre a la misma hora *en todas las zonas horarias*, como Nochevieja o la hora de cenar, utiliza la hora local.

Dado que muchos lenguajes sólo son buenos en la conversión entre UTC y hora local, puedes causarte mucho dolor si eliges almacenar tus fechas en la forma incorrecta. (Pregúntame cómo lo sé).

[1] En la Tierra, al menos. A saber qué locos sistemas usan *allá*…

[2] ¡Vale, no me mates! GMT es técnicamente una zona horaria, mientras que UTC es un sistema horario mundial. Además, algunos países pueden ajustar GMT para el horario de verano, mientras que UTC nunca se ajusta para el horario de verano

## 38.2 Tipos de fecha

Hay dos[3] tipos principales en C cuando se trata de fechas: `time_t` y `struct tm`.

La especificación no dice mucho sobre ellos:

- `time_t`: un tipo real capaz de contener un tiempo. Según la especificación, puede ser un tipo flotante o un tipo entero. En POSIX (Unix-likes), es un entero. Esto contiene *tiempo de calendario*. Que se puede considerar como la hora UTC.
- `struct tm`: contiene los componentes de una hora de calendario. Se trata de una *hora desglosada*, es decir, los componentes de la hora, como hora, minuto, segundo, día, mes, año, etc.

En muchos sistemas, `time_t` representa el número de segundos transcurridos desde *Época(Epoch)*[4].. Epoch es en cierto modo el comienzo del tiempo desde la perspectiva del ordenador, que es comúnmente el 1 de enero de 1970 UTC. `time_t` puede ser negativo para representar tiempos anteriores a Epoch. Windows se comporta de la misma manera que Unix por lo que puedo decir.

¿Y qué hay en una `struct tm`? Los siguientes campos:

```
struct tm {
    int tm_sec; // segundos después del minuto -- [0, 60]
    int tm_min; // minutos después de la hora -- [0, 59]
    int tm_hour; // horas desde medianoche -- [0, 23]
    int tm_mday; // día del mes -- [1, 31]
    int tm_mon; // meses desde enero -- [0, 11]
    int tm_year; // años desde 1900
    int tm_wday; // días desde el domingo -- [0, 6]
    int tm_yday; // días desde el 1 de enero -- [0, 365]
    int tm_isdst; // indicador del horario de verano
};
```

Tenga en cuenta que todo tiene base cero excepto el día del mes.

Es importante saber que puedes poner los valores que quieras en estos tipos. Hay funciones que ayudan a obtener la hora *ahora*, pero los tipos contienen *una* hora, no *la* hora.

Así que la pregunta es: "¿Cómo se inicializan los datos de estos tipos, y cómo se convierten entre ellos?"

## 38.3 Inicialización y conversión entre tipos

En primer lugar, puedes obtener la hora actual y almacenarla en un `time_t` con la función `time()`.

```
time_t now;  // Variable para mantener la hora actual

now = time(NULL);  // Puedes conseguirlo así...

time(&now);        // ...o esto. Igual que la línea anterior.
```

Estupendo. Ahora tienes una variable que te da la hora.

Curiosamente, sólo hay una forma portable de imprimir lo que hay en un `time_t`, y es la función `ctime()`, raramente utilizada, que imprime el valor en tiempo local:

[3] Hay que admitir que hay más de dos.
[4] https://en.wikipedia.org/wiki/Unix_time

```
now = time(NULL);
printf("%s", ctime(&now));
```

Devuelve una cadena con una forma muy específica que incluye una nueva línea al final:

```
Sun Feb 28 18:47:25 2021
```

Así que eso es un poco inflexible. Si quieres más control, deberías convertir ese `time_t` en un `struct tm`.

### 38.3.1 Convertir `time_t` en `struct tm`

Hay dos formas increíbles de hacer esta conversión:

- `localtime()`: esta función convierte un `time_t` en un `struct tm` en tiempo local.
- `gmtime()`: esta función convierte un `time_t` en un `struct tm` en UTC. (¿Ves el antiguo GMT en el nombre de la función?)

Veamos qué hora es ahora imprimiendo una `struct tm` con la función `asctime()`:

```
printf("Local: %s", asctime(localtime(&now)));
printf("  UTC: %s", asctime(gmtime(&now)));
```

Salida (estoy en la zona horaria estándar del Pacífico):

```
Local: Sun Feb 28 20:15:27 2021
  UTC: Mon Mar  1 04:15:27 2021
```

Una vez que tienes tu `time_t` en una `struct tm`, se abren todo tipo de puertas. Puede imprimir la hora de varias maneras, averiguar qué día de la semana es una fecha, y así sucesivamente. O convertirlo de nuevo en un `time_t`.

Pronto hablaremos de ello.

### 38.3.2 Convertir `struct tm` a `time_t`

Si quieres ir por otro camino, puedes usar `mktime()` para obtener esa información.

`mktime()` establece los valores de `tm_wday` y `tm_yday` por ti, así que no te molestes en rellenarlos porque serán sobrescritos.

Además, puedes poner `tm_isdst` a `-1` para que lo determine por ti. O puede establecerlo manualmente en verdadero o falso.

```
// No tenga la tentación de poner ceros a la izquierda en estos números
// (a menos que usted quiera que estén en octal)

struct tm algún_tiempo = {
    .tm_year=82, // años desde 1900
    .tm_mon=3, // meses desde enero -- [0, 11]
    .tm_mday=12, // día del mes -- [1, 31]
    .tm_hour=12, // horas desde medianoche -- [0, 23]
    .tm_min=0, // minutos después de la hora -- [0, 59]
    .tm_sec=4, // segundos después del minuto -- [0, 60]
    .tm_isdst=-1, // indicador del horario de verano
```

```
};

time_t some_time_epoch;

some_time_epoch = mktime(&some_time);

printf("%s", ctime(&some_time_epoch));
printf("Is DST: %d\n", some_time.tm_isdst);
```

Salida:

```
Mon Apr 12 12:00:04 1982
Is DST: 0
```

Cuando cargas manualmente una `struct tm` como esa, debería estar en hora local. `mktime()` convertirá esa hora local en una hora de calendario `time_t`.

Extrañamente, sin embargo, el estándar no nos da una manera de cargar una `struct tm` con una hora UTC y convertirla en un `time_t`. Si quieres hacer eso con Unix-likes, prueba la no-estándar . `timegm()`. En Windows, `_mkgmtime()`.

## 38.4 Salida de fecha formateada

Ya hemos visto un par de formas de imprimir fechas formateadas en la pantalla. Con `time_t` podemos usar `ctime()`, y con `struct tm` podemos usar `asctime()`.

```
time_t now = time(NULL);
struct tm *local = localtime(&now);
struct tm *utc = gmtime(&now);

printf("Hora local: %s", ctime(&now)); // Hora local con time_t
printf("Hora local: %s", asctime(local)); // Hora local con struct tm
printf("UTC : %s", asctime(utc)); // UTC con struct tm
```

Pero, ¿y si te dijera, querido lector, que hay una forma de tener mucho más control sobre la impresión de la fecha?

Claro, podríamos pescar campos individuales de la `struct tm`, pero hay una gran función llamada `strftime()` que hará mucho del trabajo duro por ti. Es como `printf()`, ¡pero para fechas!

Veamos algunos ejemplos. En cada uno de ellos, pasamos un buffer de destino, un número máximo de caracteres a escribir, y luego una cadena de formato (al estilo de—pero no igual que—`printf()`) que le dice a `strftime()` qué componentes de una `struct tm` imprimir y cómo.

Puede añadir otros caracteres constantes para incluir en la salida de la cadena de formato, así, al igual que con `printf()`.

Obtenemos un `struct tm` en este caso de `localtime()`, pero cualquier fuente funciona bien.

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

int main(void)
{
```

```
    char s[128];
    time_t now = time(NULL);

    // %c: print date as per current locale
    strftime(s, sizeof s, "%c", localtime(&now));
    puts(s);   // Sun Feb 28 22:29:00 2021

    // %A: full weekday name
    // %B: full month name
    // %d: day of the month
    strftime(s, sizeof s, "%A, %B %d", localtime(&now));
    puts(s);   // Sunday, February 28

    // %I: hour (12 hour clock)
    // %M: minute
    // %S: second
    // %p: AM or PM
    strftime(s, sizeof s, "It's %I:%M:%S %p", localtime(&now));
    puts(s);   // Es 10:29:00 PM

    // %F: ISO 8601 yyyy-mm-dd
    // %T: ISO 8601 hh:mm:ss
    // %z: ISO 8601 time zone offset
    strftime(s, sizeof s, "ISO 8601: %FT%T%z", localtime(&now));
    puts(s);   // ISO 8601: 2021-02-28T22:29:00-0800
}
```

Hay *toneladas* de especificadores de formato de impresión de fecha para `strftime()`, así que asegúrese de comprobarlos en la fl[ página de referencia `strftime()` |https://beej.us/guide/bgclr/html/split/time.html#man-strftime]].

## 38.5 Más Resolución con `timespec_get()`

Puede obtener el número de segundos y nanosegundos desde Epoch con `timespec_get()`.

Quizás.

Es posible que las implementaciones no tengan una resolución de nanosegundos (que es una milmillonésima parte de un segundo), así que quién sabe cuántos lugares significativos obtendrás, pero inténtalo y verás.

La función `timespec_get()` recibe dos argumentos. Uno es un puntero a una `struct timespec` que contiene la información horaria. Y el otro es la `base`, que la especificación te permite establecer a `TIME_UTC` indicando que estás interesado en segundos desde Epoch. (Otras implementaciones pueden ofrecer más opciones para la "base").

Y la propia estructura tiene dos campos:

```
struct timespec {
    time_t tv_sec;   // Segundos
    long   tv_nsec;  // Nanosegundos (milmillonésimas de segundo)
};
```

He aquí un ejemplo en el que obtenemos la hora y la imprimimos como valor entero y también como valor flotante:

```
struct timespec ts;

timespec_get(&ts, TIME_UTC);

printf("%ld s, %ld ns\n", ts.tv_sec, ts.tv_nsec);

double float_time = ts.tv_sec + ts.tv_nsec/1000000000.0;
printf("%f seconds since epoch\n", float_time);
```

Ejemplo de salida:

```
1614581530 s, 806325800 ns
1614581530.806326 seconds since epoch
```

`struct timespec` también hace su aparición en un número de funciones de threading que necesitan ser capaces de especificar el tiempo con esa resolución.

## 38.6 Diferencias entre tiempos

Una nota rápida sobre cómo obtener la diferencia entre dos `time_t`s: ya que la especificación no dicta cómo ese tipo representa un tiempo, puede que no seas capaz de simplemente restar dos `time_t`s y obtener algo sensato[5].

Por suerte, puede utilizar `difftime()` para calcular la diferencia en segundos entre dos fechas.

En el siguiente ejemplo, tenemos dos eventos que ocurren con cierta diferencia de tiempo, y usamos `difftime()` para calcular la diferencia.

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

int main(void)
{
    struct tm time_a = {
        .tm_year=82, // años desde 1900
        .tm_mon=3, // meses desde enero -- [0, 11]
        .tm_mday=12, // día del mes -- [1, 31]
        .tm_hour=4, // horas desde medianoche -- [0, 23]
        .tm_min=00, // minutos después de la hora -- [0, 59]
        .tm_sec=04, // segundos después del minuto -- [0, 60]
        .tm_isdst=-1, // indicador del horario de verano
    };

    struct tm time_b = {
        .tm_year=120, // años desde 1900
        .tm_mon=10, // meses desde enero -- [0, 11]
        .tm_mday=15, // día del mes -- [1, 31]
        .tm_hour=16, // horas desde medianoche -- [0, 23]
        .tm_min=27, // minutos después de la hora -- [0, 59]
        .tm_sec=00, // segundos después del minuto -- [0, 60]
```

[5] Lo harás en POSIX, donde `time_t` es definitivamente un entero. Desafortunadamente el mundo entero no es POSIX, así que ahí estamos

```
        .tm_isdst=-1, // indicador del horario de verano
    };

    time_t cal_a = mktime(&time_a);
    time_t cal_b = mktime(&time_b);

    double diff = difftime(cal_b, cal_a);

    double years = diff / 60 / 60 / 24 / 365.2425;  // bastante cerca

    printf("%f seconds (%f years) between events\n", diff, years);
}
```

Output:

```
1217996816.000000 seconds (38.596783 years) between events
```

Y ya está. Recuerda usar `difftime()` para tomar la diferencia de tiempo. Aunque puedes simplemente restar en un sistema POSIX, es mejor ser portable.

# Chapter 39

# Multihilo (Multithreading)

C11 introdujo, formalmente, el multithreading en el lenguaje C. Es muy similar a POSIX threads[1], si alguna vez los has usado.

Y si no, no te preocupes. Hablaremos de ello.

Sin embargo, ten en cuenta, que no pretendo que esto sea un tutorial completo de multihilo clásico[2]; tendrás que coger un libro diferente y muy grueso específicamente para eso. Lo siento.

El roscado es una característica opcional. Si un compilador C11+ define `__STDC_NO_THREADS__`, los hilos **no** estarán presentes en la librería. Por qué decidieron usar un sentido negativo en esa macro es algo que se me escapa, pero ahí estamos.

Puedes comprobarlo así:

```
#ifdef __STDC_NO_THREADS__
#error I need threads to build this program!
#endif
```

Además, es posible que tenga que especificar ciertas opciones del enlazador durante la compilación. En el caso de los Unix, prueba añadir `-lpthreads` al final de la línea de órdenes para enlazar la librería `pthreads`[3]:

```
gcc -std=c11 -o foo foo.c -lpthreads
```

Si obtiene errores del enlazador en su sistema, podría deberse a que no se incluyó la biblioteca apropiada.

## 39.1 Background

Los hilos son una forma de hacer que todos esos brillantes núcleos de CPU por los que has pagado trabajen para ti en el mismo programa.

Normalmente, un programa en C se ejecuta en un único núcleo de la CPU. Pero si sabes cómo dividir el trabajo, puedes dar partes de él a un número de hilos y hacer que lo hagan simultáneamente.

Aunque la especificación no lo dice, en tu sistema es muy probable que C (o el SO a sus órdenes) intente equilibrar los hilos entre todos los núcleos de la CPU.

[1] https://en.wikipedia.org/wiki/POSIX_Threads

[2] Yo soy más un fan de compartir-nada, y mis habilidades con las construcciones de multihilo clásico están oxidadas, por decir algo

[3] Sí, `pthreads` con "`p`". Es la abreviatura de POSIX threads, una librería de la que C11 tomó prestado libremente para su implementación de hilos

Y si tienes más hilos que núcleos, no pasa nada. Simplemente no te darás cuenta de todas esas ganancias si todos ellos están tratando de competir por el tiempo de CPU.

## 39.2 Cosas que puedes hacer

Puedes crear un hilo. Comenzará a ejecutar la función que especifiques. El hilo padre que lo generó también continuará ejecutándose.

Y puedes esperar a que el hilo se complete. Esto se llama *joining* (*unirse*).

O si no te importa cuando se complete el hilo y no quieres esperar, puedes *detach it* (*separarlo*).

Un hilo puede *exit* (*salir*) explícitamente, o puede llamarlo implícitamente *quits* (*se retira*) al volver de su función principal.

Un hilo también puede *Sleep* (*dormir*) durante un periodo de tiempo, sin hacer nada mientras otros hilos se ejecutan.

El programa `main()` también es un hilo.

Además, tenemos almacenamiento local de hilos, mutexes y variables condicionales. Pero hablaremos de ello más adelante. Por ahora, veamos sólo lo básico.

## 39.3 Razas de datos y la biblioteca estándar

Algunas de las funciones de la biblioteca estándar (por ejemplo, `asctime()` y `strtok()`) devuelven o utilizan elementos de datos `static` que no son threadsafe. Pero en general, a menos que se diga lo contrario, la biblioteca estándar hace un esfuerzo para serlo[4].

Pero tenga cuidado. Si una función de la biblioteca estándar mantiene el estado entre llamadas en una variable que no te pertenece, o si una función devuelve un puntero a algo que no has pasado, no es threadsafe.

## 39.4 Creando y Esperando Hilos

¡Vamos a hackear algo!

Crearemos algunos hilos (create) y esperaremos a que se completen (join).

Aunque primero tenemos que entender un poco.

Cada hilo se identifica por una variable opaca de tipo `thrd_t`. Es un identificador único por hilo en tu programa. Cuando creas un hilo, se le da un nuevo ID.

También, cuando creas el hilo, tienes que darle un puntero a una función para ejecutar, y un puntero a un argumento para pasarle (o `NULL` si no tienes nada que pasar).

El hilo comenzará la ejecución de la función que especifiques.

Cuando quieras esperar a que un hilo se complete, tienes que especificar su ID de hilo para que C sepa a cuál esperar.

Así que la idea básica es

1. Escribe una función que actúe como "`main`" del hilo. No es `main()` propiamente dicha, pero es análoga a ella. El hilo comenzará a ejecutarse allí.
2. Desde el hilo principal, lanzar un nuevo hilo con `thrd_create()`, y pasarle un puntero a la función a ejecutar.
3. En esa función, haz que el hilo haga lo que tenga que hacer.

[4] Según §7.1.4¶5.

4. Mientras tanto, el hilo principal puede seguir haciendo lo que tenga que hacer.
5. Cuando el hilo principal lo decida, puede esperar a que el hilo hijo termine llamando a la función . `thrd_join()`. Generalmente **debe** `thrd_join()` el hilo para limpiar después de él o de lo contrario perderá memoria[5].

`thrd_create()` toma un puntero a la función a ejecutar, y es de tipo `thrd_start_t`, que es `int (*)(void *)`. Esto significa en griego "un puntero a una función que toma un `void*` como argumento, y devuelve un `int`".

¡Vamos a crear un hilo! Lo lanzaremos desde el hilo principal con `thrd_create()` para ejecutar una función, hacer algunas otras cosas, y luego esperar a que se complete con `thrd_join()`. He llamado a la función principal del hilo `run()`, pero puedes llamarla como quieras siempre que los tipos coincidan con `thrd_start_t`.

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

// Esta es la función que el hilo ejecutará. Puede llamarse cualquier cosa.
//
// arg es el puntero del argumento pasado a `thrd_create()`.
//
// El hilo padre obtendrá el valor de retorno de `thrd_join()`'
// más tarde.

int run(void *arg)
{
    int *a = arg; // Pasaremos un int* desde thrd_create()

    printf("THREAD: Running thread with arg %d\n", *a);

    return 12; // Valor que recogerá thrd_join() (eligió 12 al azar)
}

int main(void)
{
    thrd_t t;  // t contendrá el ID del hilo
    int arg = 3490;

    printf("Launching a thread\n");

    // Lanza un hilo a la función run(), pasando un puntero a 3490
    // como argumento. También almacena el ID del hilo en t:

    thrd_create(&t, run, &arg);

    printf("Hacer otras cosas mientras se ejecuta el hilo\n");

    printf("Esperando a que se complete el hilo...\n");

    int res;  // Mantiene el valor de retorno de la salida del hilo

    // Espera aquí a que el hilo termine; almacena el valor de retorno
    // en res:
```

[5] A menos que `thrd_detach()`. Más sobre esto más adelante

```
    thrd_join(t, &res);

    printf("Thread finalizado con valor de retorno %d\n", res);
}
```

¿Ves cómo hicimos el `thrd_create()` allí para llamar a la función `run()`? Luego hicimos otras cosas en `main()` y luego paramos y esperamos a que el hilo se complete con `thrd_join()`.

Ejemplos de resultados (los suyos pueden variar):

```
Launching a thread
Doing other things while the thread runs
Waiting for thread to complete...
THREAD: Running thread with arg 3490
Thread exited with return value 12
```

La `arg` que pases a la función tiene que tener un tiempo de vida lo suficientemente largo como para que el hilo pueda recogerla antes de que desaparezca. Además, no debe ser sobrescrita por el hilo principal antes de que el nuevo hilo pueda utilizarla.

Veamos un ejemplo que lanza 5 hilos. Una cosa a tener en cuenta aquí es cómo usamos un array de `thrd_t`s para mantener un registro de todos los IDs de los hilos.

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

int run(void *arg)
{
    int i = *(int*)arg;

    printf("THREAD %d: running!\n", i);

    return i;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    int i;

    printf("Launching threads...\n");
    for (i = 0; i < THREAD_COUNT; i++)

        // ¡NOTA! En la siguiente línea, pasamos un puntero a i,
        // pero cada hilo ve el mismo puntero. Así que
        // imprimirán cosas raras cuando i cambie de valor aquí en
        // el hilo principal! (Más en el texto, abajo.)

        thrd_create(t + i, run, &i);
```

```
    printf("Doing other things while the thread runs...\n");
    printf("Waiting for thread to complete...\n");

    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;
        thrd_join(t[i], &res);

        printf("Thread %d complete!\n", res);
    }

    printf("All threads complete!\n");
}
```

Cuando ejecuto los hilos, cuento `i` de 0 a 4. Y le paso un puntero a `thrd_create()`. Este puntero termina en la rutina `run()` donde hacemos una copia del mismo.

¿Es sencillo? Aquí está el resultado:

```
Launching threads...
THREAD 2: running!
THREAD 3: running!
THREAD 4: running!
THREAD 2: running!
Doing other things while the thread runs...
Waiting for thread to complete...
Thread 2 complete!
Thread 2 complete!
THREAD 5: running!
Thread 3 complete!
Thread 4 complete!
Thread 5 complete!
All threads complete!
```

¿Qué…? ¿Dónde está el "HILO 0"? ¿Y por qué tenemos un `THREAD 5` cuando claramente `i` nunca es más de `4` cuando llamamos a `thrd_create()`? ¿Y dos "THREAD 2"? ¡Qué locura!

Esto es entrar en la divertida tierra de *condiciones de carrera*. El hilo principal está modificando `i` antes de que el hilo tenga la oportunidad de copiarlo. De hecho, `i` llega hasta `5` y termina el bucle antes de que el último hilo tenga la oportunidad de copiarlo.

Tenemos que tener una variable por hilo a la que podamos referirnos para poder pasarla como `arg`.

Podríamos tener un gran array de ellas. O podríamos `malloc()` espacio (y liberarlo en algún lugar - tal vez en el propio hilo.)

Vamos a intentarlo:

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

int run(void *arg)
{
    int i = *(int*)arg;  // Copia el arg
```

```
    free(arg);  // Terminado con esto

    printf("THREAD %d: running!\n", i);

    return i;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    int i;

    printf("Launching threads...\n");
    for (i = 0; i < THREAD_COUNT; i++) {

        // Obtener espacio para un argumento por hilo:

        int *arg = malloc(sizeof *arg);
        *arg = i;

        thrd_create(t + i, run, arg);
    }

    // ...
```

Observa que en las líneas 27-30 hacemos `malloc()` para un `int` y copiamos el valor de `i` en él. Cada nuevo hilo obtiene su propia variable recién `malloc()` y le pasamos un puntero a la función `run()`.

Una vez que `run()` hace su propia copia de la `arg` en la línea 7, `free()`s la `malloc()` `int`. Y ahora que tiene su propia copia, puede hacer con ella lo que le plazca.

Y una ejecución muestra el resultado:

```
Launching threads...
THREAD 0: running!
THREAD 1: running!
THREAD 2: running!
THREAD 3: running!
Doing other things while the thread runs...
Waiting for thread to complete...
Thread 0 complete!
Thread 1 complete!
Thread 2 complete!
Thread 3 complete!
THREAD 4: running!
Thread 4 complete!
All threads complete!
```

¡Allá vamos! ¡Hilos 0-4 todos en efecto!

Tu ejecución puede variar—cómo los hilos se programan para ejecutarse, está más allá de la especificación C. Vemos en el ejemplo anterior que el subproceso 4 ni siquiera comenzó hasta que los subprocesos 0-1 se

completaron. De hecho, si ejecuto esto de nuevo, es probable que obtenga resultados diferentes. No podemos garantizar un orden de ejecución de hilos.

## 39.5 Separación de hilos

Si desea despedir y olvidar un hilo (es decir, para no tener que `thrd_join()` más tarde), puede hacerlo con `thrd_detach()`.

Esto elimina la capacidad de la hebra padre de obtener el valor de retorno de la hebra hija, pero si no te importa eso y sólo quieres que las hebras se limpien bien por sí solas, este es el camino a seguir.

Básicamente vamos a hacer esto:

```
thrd_create(&t, run, NULL);
thrd_detach(t);
```

donde la llamada a `thrd_detach()` es la hebra padre diciendo, “Hey, no voy a esperar a que esta hebra hija termine con `thrd_join()`. Así que sigue adelante y límpialo por tu cuenta cuando se complete”.

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

int run(void *arg)
{
    (void)arg;

    //printf("¡Hilo en ejecución! %lu\n", thrd_current()); // ¡no portable!
    printf("Thread running!\n");

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t;

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(&t, run, NULL);
        thrd_detach(t);               // <-- DETACH!
    }

    // Duerme un segundo para que todos los hilos terminen
    thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
}
```

Tenga en cuenta que en este código, ponemos el hilo principal a dormir durante 1 segundo con `thrd_sleep()`—más sobre esto más adelante.

También en la función `run()`, tengo una línea comentada que imprime el ID del hilo como un `unsigned long`. Esto es no-portable, porque la especificación no dice qué tipo es un `thrd_t` bajo el capó—podría ser una `struct` por lo que sabemos. Pero esa línea funciona en mi sistema.

Algo interesante que vi cuando ejecuté el código anterior e imprimí los IDs de los hilos fue que ¡algunos hilos tenían IDs duplicados! Esto parece que debería ser imposible, pero a C se le permite *reutilizar* los IDs de los hilos después de que el hilo correspondiente haya salido. Así que lo que estaba viendo era que algunos hilos completaban su ejecución antes de que otros hilos fueran lanzados.

## 39.6 Datos Locales del Hilo

Los hilos son interesantes porque no tienen memoria propia más allá de las variables locales. Si quieres una variable `static` o una variable de ámbito de fichero, todos los hilos verán esa misma variable.

Esto puede conducir a condiciones de carrera, donde se obtienen *Weird Things*™ (*Cosas raras*) sucediendo.

Mira este ejemplo. Tenemos una variable `static` `foo` en el ámbito de bloque en `run()`. Esta variable será visible para todos los hilos que pasen por la función `run()`. Y los distintos hilos pueden pisarse unos a otros.

Cada hilo copia `foo` en una variable local `x` (que no es compartida entre hilos— todos los hilos tienen sus propias pilas de llamadas). Así que *deberían* ser iguales, ¿no?

Y la primera vez que los imprimimos, lo son[6]. Pero justo después de eso comprobamos que siguen siendo los mismos.

Y *normalmente* lo son. Pero no siempre.

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

int run(void *arg)
{
    int n = *(int*)arg;  // Número de hilo para que los humanos lo diferencien

    free(arg);

    static int foo = 10;  // Valor estático compartido entre hilos

    int x = foo;  // Variable local automática--cada hilo tiene la suya propia

    // Acabamos de asignar x desde foo, así que más vale que sean iguales aquí.
    // (En todas mis pruebas, lo eran, ¡pero ni siquiera esto está garantizado!)

    printf("Thread %d: x = %d, foo = %d\n", n, x, foo);

    // Y aquí deberían ser iguales, ¡pero no siempre lo son!
    // (¡A veces sí, a veces no!)

    // Lo que pasa es que otro hilo entra e incrementa foo
    // en este momento, ¡pero la x de este thread sigue siendo la que era antes!

    if (x != foo) {
        printf("Thread %d: Craziness! x != foo! %d != %d\n", n, x, foo);
    }

    foo++;  // Incrementar el valor compartido
```

[6]Aunque no creo que tengan que serlo. Es sólo que los hilos no parecen ser reprogramados hasta que alguna llamada al sistema como podría ocurrir con un `printf()`… que es por lo que tengo el `printf()` ahí

```
    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Contiene un número de serie del hilo
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }
}
```

He aquí un ejemplo (aunque varía de una ejecución a otra):

```
Thread 0: x = 10, foo = 10
Thread 1: x = 10, foo = 10
Thread 1: Craziness! x != foo! 10 != 11
Thread 2: x = 12, foo = 12
Thread 4: x = 13, foo = 13
Thread 3: x = 14, foo = 14
```

En el hilo 1, entre los dos `printf()`s, el valor de `foo` de alguna manera cambió de `10` a `11`, ¡aunque claramente no hay incremento entre los `printf()`s!

¡Fue otro hilo el que entró ahí (probablemente el hilo 0, por lo que parece) e incrementó el valor de `foo` a espaldas del hilo 1!

Resolvamos este problema de dos maneras diferentes. (Si quieres que todos los hilos compartan la variable *y* no se pisen unos a otros, tendrás que seguir leyendo la sección mutex).

### 39.6.1 Clase de almacenamiento `_Thread_local`

Lo primero es lo primero, veamos la forma más sencilla de evitarlo: la clase de almacenamiento `_Thread_local`.

Básicamente, vamos a ponerla delante de nuestra variable `static` de ámbito de bloque, ¡y todo funcionará! Le dice a C que cada hilo debe tener su propia versión de esta variable, para que ninguno de ellos pise a los demás.

El `<threads.h>` define `thread_local` como un alias de `_Thread_local` para que tu código no tenga que verse tan feo.

Tomemos el ejemplo anterior y convirtamos `foo` en una variable `thread_local` para no compartir esos datos.

```
int run(void *arg)
{
    int n = *(int*)arg;  // Número de hilo para que los humanos lo diferencien

    free(arg);

    thread_local static int foo = 10;  // <-- ¡¡¡Ya no se comparte!!!
```

Y corriendo llegamos:

```
Thread 0: x = 10, foo = 10
Thread 1: x = 10, foo = 10
Thread 2: x = 10, foo = 10
Thread 4: x = 10, foo = 10
Thread 3: x = 10, foo = 10
```

Se acabaron los problemas raros.

Una cosa: si una variable `thread_local` tiene ámbito de bloque, **debe** ser `static`. Esas son las reglas. (Pero esto está bien porque las variables no `static` ya son per-thread ya que cada thread tiene sus propias variables no `static`).

Un poco de mentira: las variables `thread_local` de ámbito de bloque también pueden ser `extern`.

### 39.6.2 Otra opción: Almacenamiento específico de subprocesos

El almacenamiento específico de subprocesos (TSS) es otra forma de obtener datos por subproceso.

Una característica adicional es que estas funciones permiten especificar un destructor que será llamado sobre los datos cuando la variable TSS sea borrada. Comúnmente este destructor es `free()` para limpiar automáticamente los datos por hilo `malloc()`d. O `NULL` si no necesitas destruir nada.

El destructor es de tipo `tss_dtor_t` que es un puntero a una función que devuelve `void` y toma un `void*` como argumento (el `void*` apunta a los datos almacenados en la variable). En otras palabras, es un `void (*)(void*)`, si eso lo aclara. Que admito que probablemente no. Mira el ejemplo de abajo.

Generalmente, `thread_local` es probablemente tu mejor opción, pero si te gusta la idea del destructor, entonces puedes hacer uso de eso.

El uso es un poco raro en el sentido de que necesitamos una variable de tipo `tss_t` para representar el valor de cada hilo. Luego la inicializamos con `tss_create()`. Finalmente nos deshacemos de él con `tss_delete()`. Nótese que llamar a `tss_delete()` no ejecuta todos los destructores–es `thrd_exit()` (o volver de la función run) la que lo hace. `tss_delete()` sólo libera la memoria asignada por `tss_create()`.

En el medio, los hilos pueden llamar `tss_set()` y `tss_get()` para establecer y obtener el valor.

En el siguiente código, establecemos la variable TSS antes de crear los hilos, y luego limpiamos después de los hilos.

En la función `run()`, los hilos `malloc()` un poco de espacio para una cadena y almacenan ese puntero en la variable TSS.

Cuando el hilo sale, la función destructora (`free()` en este caso) es llamada para *todos* los hilos.

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

```
tss_t str;

void some_function(void)
{
    // Recuperar el valor por hilo de esta cadena
    char *tss_string = tss_get(str);

    // E imprimirlo
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Obtener el número de serie de este hilo
    free(arg);

    // malloc() espacio para guardar los datos de este hilo
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Cadena feliz

    // Establece esta variable TSS para que apunte a la cadena
    tss_set(str, s);

    // Llama a una función que obtendrá la variable
    some_function();

    return 0;   // Equivalente a thrd_exit(0)
}

#define THREAD_COUNT 15

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Crea una nueva variable TSS, la función free() es el destructor
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Contiene un número de serie del hilo
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // Todos los hilos están hechos, así que hemos terminado con esto
    tss_delete(str);
}
```

Una vez más, esta es una forma un poco dolorosa de hacer las cosas en comparación con `thread_local`, así

que a menos que realmente necesites esa funcionalidad destructor, yo usaría eso en su lugar.

## 39.7 Mutexes

Si sólo quieres permitir que un único hilo entre en una sección crítica de código a la vez, puedes proteger esa sección con un mutex[7].

Por ejemplo, si tuviéramos una variable `static` y quisiéramos poder obtenerla y establecerla en dos operaciones sin que otro hilo saltara en medio y la corrompiera, podríamos usar un mutex para eso.

Puedes adquirir un mutex o liberarlo. Si intentas adquirir el mutex y tienes éxito, puedes continuar la ejecución. Si lo intentas y fallas (porque alguien más lo tiene), te *bloquearás*[8] hasta que el mutex sea liberado.

Si varios hilos están bloqueados esperando a que se libere un mutex, uno de ellos será elegido para ejecutarse (al azar, desde nuestra perspectiva), y los demás seguirán durmiendo.

El plan de juego es que primero inicializaremos una variable mutex para que esté lista para usar con `mtx_init()`.

Entonces los hilos subsiguientes pueden llamar a `mtx_lock()` y `mtx_unlock()` para obtener y liberar el mutex.

Cuando hayamos terminado completamente con el mutex, podemos destruirlo con la función `mtx_destroy()`, el opuesto lógico de `mtx_init()`.

Primero, veamos algo de código que *no* usa un mutex, e intenta imprimir un número de serie compartido (`static`) y luego incrementarlo. Debido a que no estamos usando un mutex sobre la obtención del valor (para imprimirlo) y el ajuste (para incrementarlo), los hilos podrían interponerse en el camino de los demás en esa sección crítica.

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

int run(void *arg)
{
    (void)arg;

    static int serial = 0;   // ¡Variable estática compartida!

    printf("Thread running! %d\n", serial);

    serial++;

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(t + i, run, NULL);
```

[7] Abreviatura de “exclusión mutua”, alias un “bloqueo” en una sección de código que sólo un hilo puede ejecutar
[8] Es decir, tu proceso entrará en reposo

```
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }
}
```

Cuando ejecuto esto, obtengo algo parecido a esto:

```
Thread running! 0
Thread running! 0
Thread running! 0
Thread running! 3
Thread running! 4
Thread running! 5
Thread running! 6
Thread running! 7
Thread running! 8
Thread running! 9
```

Claramente múltiples hilos están entrando y ejecutando el `printf()` antes de que nadie pueda actualizar la variable `serial`.

Lo que queremos hacer es envolver la obtención de la variable y su establecimiento en un único tramo de código protegido por mutex.

Añadiremos una nueva variable para representar el mutex de tipo `mtx_t` en el ámbito del fichero, la inicializaremos, y entonces los hilos podrán bloquearla y desbloquearla en la función `run()`.

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

mtx_t serial_mtx;     // <-- MUTEX VARIABLE

int run(void *arg)
{
    (void)arg;

    static int serial = 0;   // ¡Variable estática compartida!

   // Adquirir el mutex--todos los hilos se bloquearán en esta llamada hasta que
    // obtengan el bloqueo:

    mtx_lock(&serial_mtx);           // <-- ACQUIRE MUTEX

    printf("Thread running! %d\n", serial);

    serial++;

    // Terminado de obtener y fijar los datos, libera el bloqueo. Esto
    // desbloqueará los hilos en la llamada a mtx_lock():

    mtx_unlock(&serial_mtx);         // <-- RELEASE MUTEX
```

```
    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Inicializar la variable mutex, indicando que esto es un normal
    // sin florituras, mutex:

    mtx_init(&serial_mtx, mtx_plain);        // <-- CREATE MUTEX

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(t + i, run, NULL);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // Done with the mutex, destroy it:

    mtx_destroy(&serial_mtx);                // <-- DESTROY MUTEX
}
```

Mira cómo en las líneas 38 y 50 de `main()` inicializamos y destruimos el mutex.

Pero cada hilo individual adquiere el mutex en la línea 15 y lo libera en la línea 24.

Entre `mtx_lock()` y `mtx_unlock()` está la *sección crítica*, el área de código en la que no queremos que varios hilos se metan al mismo tiempo.

Y ahora obtenemos una salida adecuada.

```
Thread running! 0
Thread running! 1
Thread running! 2
Thread running! 3
Thread running! 4
Thread running! 5
Thread running! 6
Thread running! 7
Thread running! 8
Thread running! 9
```

Si necesitas múltiples mutexes, no hay problema: simplemente ten múltiples variables mutex.

Y recuerda siempre la Regla Número Uno de los Mutexes Múltiples: Desbloquea los mutex en el orden opuesto al que los bloqueaste.

### 39.7.1 Diferentes Tipos de Mutex

Como se ha insinuado antes, tenemos algunos tipos de mutex que puedes crear con `mtx_init()`. (Algunos de estos tipos son el resultado de una operación bitwise-OR, como se indica en la tabla).

| Tipo | Descripción |
|---|---|
| `mtx_plain` | Mutex normal y corriente |
| `mtx_timed` | Mutex que admite tiempos de espera |
| `mtx_plain\|mtx_recursive` | Mutex recursivo |
| `mtx_timed\|mtx_recursive` | Mutex recursivo que admite tiempos de espera |

"Recursivo" significa que el poseedor de un bloqueo puede llamar a `mtx_lock()` varias veces sobre el mismo bloqueo. (Tienen que desbloquearlo un número igual de veces antes de que alguien más pueda tomar el mutex). Esto puede facilitar la codificación de vez en cuando, especialmente si llamas a una función que necesita bloquear el mutex cuando ya tienes el mutex.

Y el tiempo de espera da a un hilo la oportunidad de *intentar* obtener el bloqueo durante un tiempo, pero luego abandonarlo si no puede conseguirlo en ese plazo.

Para un mutex con tiempo de espera, asegúrate de crearlo con `mtx_timed`:

```
mtx_init(&serial_mtx, mtx_timed);
```

Y luego, cuando lo esperas, tienes que especificar una hora en UTC en la que se desbloqueará[9]

La función `timespec_get()` de `<time.h>` puede ser de ayuda aquí. Te dará la hora actual en UTC en una `struct timespec` que es justo lo que necesitamos. De hecho, parece existir sólo para este propósito.

Tiene dos campos: `tv_sec` tiene el tiempo actual en segundos desde la época, y `tv_nsec` tiene los nanosegundos (milmillonésimas de segundo) como parte "fraccionaria".

Así que puedes cargarlo con el tiempo actual, y luego añadirlo para obtener un tiempo de espera específico.

Entonces llame a `mtx_timedlock()` en lugar de a `mtx_lock()`. Si devuelve el valor `thrd_timedout`, se ha agotado el tiempo de espera.

```
struct timespec timeout;

timespec_get(&timeout, TIME_UTC);  // Obtener la hora actual
timeout.tv_sec += 1;             // Tiempo de espera 1 segundo después de ahora

int result = mtx_timedlock(&serial_mtx, &timeout));

if (result == thrd_timedout) {
    printf("Mutex lock timed out!\n");
}
```

Aparte de eso, los bloqueos temporizados son iguales que los bloqueos normales.

## 39.8 Variables de condición

Las variables de condición son la última pieza del rompecabezas que necesitamos para crear aplicaciones multihilo eficaces y componer estructuras multihilo más complejas.

[9] Puede que esperaras que fuera "hora a partir de ahora", pero te gustaría pensar eso, ¿verdad?

Una variable de condición proporciona una manera para que los hilos vayan a dormir hasta que ocurra algún evento en otro hilo.

En otras palabras, podemos tener un número de hilos que están listos para empezar, pero tienen que esperar hasta que algún evento se cumpla antes de continuar. Básicamente se les está diciendo "¡esperad!" hasta que se les notifique.

Y esto trabaja mano a mano con mutexes ya que lo que vamos a esperar generalmente depende del valor de algunos datos, y esos datos generalmente necesitan ser protegidos por un mutex.

Es importante tener en cuenta que la variable de condición en sí no es el titular de ningún dato en particular desde nuestra perspectiva. Es simplemente la variable mediante la cual C realiza un seguimiento del estado de espera/no espera de un hilo o grupo de hilos en particular.

Escribamos un programa artificial que lea grupos de 5 números del hilo principal de uno en uno. Entonces, cuando se hayan introducido 5 números, el subproceso hijo se despierta, suma esos 5 números e imprime el resultado.

Los números se almacenarán en una matriz global compartida, al igual que el índice de la matriz del número que se va a introducir.

Como se trata de valores compartidos, al menos tenemos que esconderlos detrás de un mutex tanto para el hilo principal como para el secundario. (El principal escribirá datos en ellos y el hijo los leerá).

Pero eso no es suficiente. El subproceso hijo necesita bloquearse ("dormir") hasta que se hayan leído 5 números en el array. Y entonces la hebra padre tiene que despertar a la hebra hija para que pueda hacer su trabajo.

Y cuando se despierte, necesita mantener ese mutex. Y lo hará. Cuando un hilo espera en una variable de condición, también adquiere un mutex cuando se despierta.

Todo esto tiene lugar alrededor de una variable adicional de tipo `cnd_t` que es la *variable de condición*. Creamos esta variable con la función `cnd_init()` y la destruimos cuando acabemos con ella con la `cnd_destroy()`.

Pero, ¿cómo funciona todo esto? Veamos el esquema de lo que hará el hilo hijo:

1. Bloquea el mutex con `mtx_lock()`.
2. Si no hemos introducido todos los números, esperar en la variable condición con `cnd_wait()`.
3. Hacer el trabajo que haya que hacer
4. Desbloquear el mutex con `mtx_unlock()`.

Mientras tanto el hilo principal estará haciendo lo siguiente

1. Bloquear el mutex con `mtx_lock()`.
2. Almacenar el número leído recientemente en el array
3. Si el array está lleno, indica al hijo que se despierte con `cnd_signal()`.
4. Desbloquea el mutex con `mtx_unlock()`.

Si no lo has ojeado demasiado (no pasa nada, no me ofendo), puede que notes algo raro: ¿cómo puede el hilo principal mantener el bloqueo mutex y enviar una señal al hijo, si el hijo tiene que mantener el bloqueo mutex para esperar la señal? No pueden mantener ambos el bloqueo.

Y de hecho no lo hacen. Hay algo de magia entre bastidores con las variables de condición: cuando `cnd_wait()`, libera el mutex que especifiques y el hilo se va a dormir. Y cuando alguien indica a ese hilo que se despierte, vuelve a adquirir el bloqueo como si nada hubiera pasado.

Es un poco diferente en el lado `cnd_signal()` de las cosas. Esto no hace nada con el mutex. La hebra señalizadora aún debe liberar manualmente el mutex antes de que las hebras en espera puedan despertarse.

Una cosa más sobre `cnd_wait()`. Probablemente llame a `cnd_wait()` si alguna condición[10] aún no se

[10] ¡Y por eso se llaman *variables de condición*!

cumple (por ejemplo, en este caso, si aún no se han introducido todos los números). Este es el problema: esta condición debería estar en un bucle `while`, no en una sentencia `if`. ¿Por qué?

Por un misterioso fenómeno llamado *spurious wakeup*. A veces, en algunas implementaciones, un hilo puede ser despertado de una suspensión `cnd_wait()` aparentemente *sin razón [X-Files music]*. No digo que sean aliens… pero son aliens. Vale, en realidad es más probable que otro hilo se haya despertado y haya llegado al trabajo primero]. Y así tenemos que comprobar que la condición que necesitamos todavía se cumple cuando nos despertamos. Y si no es así, ¡a dormir!

Así que ¡manos a la obra! Empezando por el hilo principal:

- El hilo principal creará el mutex y la variable condición, y lanzará el hilo hijo.
- Luego, en un bucle infinito, obtendrá números de la consola.
- También adquirirá el mutex para almacenar los números introducidos en un array global.
- Cuando el array tenga 5 números, la hebra principal indicará a la hebra hija que es hora de despertar y hacer su trabajo.
- Entonces el hilo principal desbloqueará el mutex y volverá a leer el siguiente número de la consola.

Mientras tanto, el subproceso hijo ha estado haciendo sus propias travesuras:

- El hilo hijo toma el mutex
- Mientras no se cumpla la condición (es decir, mientras el array compartido no tenga todavía 5 números), la hebra hija duerme esperando en la variable de condición. Cuando espera, implícitamente desbloquea el mutex.
- Una vez que el hilo principal indica al hilo hijo que se despierte, éste se despierta para hacer el trabajo y recupera el bloqueo mutex.
- El subproceso hijo suma los números y restablece la variable que es el índice en la matriz.
- Entonces libera el mutex y se ejecuta de nuevo en un bucle infinito.

Y aquí está el código. Estudialo un poco para que puedas ver donde se manejan todas las piezas anteriores:

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

#define VALUE_COUNT_MAX 5

int value[VALUE_COUNT_MAX];  // Global compartido
int value_count = 0;   // Global compartido, también

mtx_t value_mtx;   // Mutex alrededor del valor
cnd_t value_cnd;   // Condicionar la variable al valor

int run(void *arg)
{
    (void)arg;

    for (;;) {
        mtx_lock(&value_mtx);      // <-- GRAB THE MUTEX

        while (value_count < VALUE_COUNT_MAX) {
            printf("Thread: is waiting\n");
            cnd_wait(&value_cnd, &value_mtx);  // <-- CONDITION WAIT
```

```
        }

        printf("Thread: is awake!\n");

        int t = 0;

        // Add everything up
        for (int i = 0; i < VALUE_COUNT_MAX; i++)
            t += value[i];

        printf("Thread: total is %d\n", t);

        // Reset input index for main thread
        value_count = 0;

        mtx_unlock(&value_mtx);   // <-- MUTEX UNLOCK
    }

    return 0;
}

int main(void)
{
    thrd_t t;

    // Crear un nuevo tema

    thrd_create(&t, run, NULL);
    thrd_detach(t);

    // Configurar el mutex y la variable de condición

    mtx_init(&value_mtx, mtx_plain);
    cnd_init(&value_cnd);

    for (;;) {
        int n;

        scanf("%d", &n);

        mtx_lock(&value_mtx);    // <-- LOCK MUTEX

        value[value_count++] = n;

        if (value_count == VALUE_COUNT_MAX) {
            printf("Main: signaling thread\n");
            cnd_signal(&value_cnd);  // <-- SIGNAL CONDITION
        }

        mtx_unlock(&value_mtx);  // <-- UNLOCK MUTEX
    }

    // Limpiar (Sé que es un bucle infinito aquí arriba, pero yo
```

```
    // quiero al menos pretender ser correcto):

    mtx_destroy(&value_mtx);
    cnd_destroy(&value_cnd);
}
```

Y aquí hay algunos ejemplos de salida (los números individuales en las líneas son mis entradas):

```
Thread: is waiting
1
1
1
1
1
Main: signaling thread
Thread: is awake!
Thread: total is 5
Thread: is waiting
2
8
5
9
0
Main: signaling thread
Thread: is awake!
Thread: total is 24
Thread: is waiting
```

Es un uso común de las variables de condición en situaciones productor-consumidor como ésta. Si no tuviéramos una forma de poner el hilo hijo a dormir mientras espera a que se cumpla alguna condición, se vería forzado a sondear, lo cual es un gran desperdicio de CPU.

### 39.8.1 Timed Condition Wait

Hay una variante de `cnd_wait()` que permite especificar un tiempo de espera para que pueda dejar de esperar.

Dado que el subproceso hijo debe volver a bloquear el mutex, esto no significa necesariamente que vaya a volver a la vida en el instante en que se produce el tiempo de espera; todavía debe esperar a que cualquier otro subproceso libere el mutex.

Pero sí significa que no estarás esperando hasta que ocurra la `cnd_signal()`.

Para que esto funcione, llame a la función `cnd_timedwait()` en lugar de `cnd_wait()`. Si devuelve el valor `thrd_timedout`, se ha agotado el tiempo de espera.

La marca de tiempo es un tiempo absoluto en UTC, no un tiempo-desde-ahora. Gracias a la función `timespec_get()` en `<time.h>` parece hecha a medida exactamente para este caso.

```
struct timespec timeout;

timespec_get(&timeout, TIME_UTC);  // Obtener la hora actual
timeout.tv_sec += 1;             // Tiempo de espera 1 segundo después de ahora
```

```
int result = cnd_timedwait(&condition, &mutex, &timeout));

if (result == thrd_timedout) {
    printf("Condition variable timed out!\n");
}
```

### 39.8.2 Broadcast: Despertar todos los hilos en espera

`cnd_signal()` function]] `cnd_signal()` only wakes up one thread to continue working. Depending on how you have your logic done, it might make sense to wake up more than one thread to continue once the condition is met.

Of course only one of them can grab the mutex, but if you have a situation where:

- The newly-awoken thread is responsible for waking up the next one, and—
- There's a chance the spurious-wakeup loop condition will prevent it from doing so, then—

you'll want to broadcast the wake up so that you're sure to get at least one of the threads out of that loop to launch the next one.

How, you ask?

Simply use `cnd_broadcast()` instead of `cnd_signal()`. Exact same usage, except `cnd_broadcast()` wakes up **all** the sleeping threads that were waiting on that condition variable.

## 39.9 Running a Function One Time

Let's say you have a function that *could* be run by many threads, but you don't know when, and it's not work trying to write all that logic.

There's a way around it: use `call_once()`. Tons of threads could try to run the function, but only the first one counts[11]

To work with this, you need a special flag variable you declare to keep track of whether or not the thing's been run. And you need a function to run, which takes no parameters and returns no value.

```
once_flag of = ONCE_FLAG_INIT;  // Initialize it like this

void run_once_function(void)
{
    printf("I'll only run once!\n");
}

int run(void *arg)
{
    (void)arg;

    call_once(&of, run_once_function);

    // ...
```

In this example, no matter how many threads get to the `run()` function, the `run_once_function()` will only be called a single time.

[11] Survival of the fittest! Right? I admit it's actually nothing like that.

# Chapter 40

# Atomics

> "¿Lo intentaron y fracasaron, todos ellos?"> "Oh, no." Sacudió la cabeza. "Lo intentaron y murieron."
>
> —Paul Atreides y la Reverenda Madre Gaius Helen Mohiam, *Dune*

Este es uno de los aspectos más desafiantes del multithreading con C. Pero intentaremos tomárnoslo con calma.

Básicamente, hablaré de los usos más sencillos de las variables atómicas, qué son, cómo funcionan, etc. Y mencionaré algunos de los caminos más increíblemente complejos que están a tu disposición.

Pero no voy a ir por esos caminos. No sólo apenas estoy cualificado para escribir sobre ellos, sino que me imagino que si sabes que los necesitas, ya sabes más que yo.

Pero hay algunas cosas raras incluso en lo básico. Así que abróchense los cinturones, porque Kansas se va.

## 40.1 Pruebas de compatibilidad atómica

Los atómicos son opcionales. Hay una macro `__STDC_NO_ATOMICS__` que es `1` si *no* tienes atómicos.

Esa macro podría no existir antes de C11, así que deberíamos comprobar la versión del lenguaje con `__STDC_VERSION__`[1].

```
#if __STDC_VERSION__ < 201112L || __STDC_NO_ATOMICS__ == 1
#define HAS_ATOMICS 0
#else
#define HAS_ATOMICS 1
#endif
```

Si esas pruebas pasan, entonces puedes incluir con seguridad `<stdatomic.h>`, la cabecera en la que se basa el resto de este capítulo. Pero si no hay soporte atómico, puede que esa cabecera ni siquiera exista.

En algunos sistemas, puede que necesites añadir `-latomic` al final de tu línea de comandos de compilación para usar cualquier función del fichero de cabecera.

## 40.2 Variables atómicas

Esto es *parte* de cómo funcionan las variables atómicas:

[1] La macro `__STDC_VERSION__` no existía a principios de C89, así que si estás preocupado por eso, compruébalo con `#ifdef`

Si tienes una variable atómica compartida y escribes en ella desde una hebra, esa escritura será *todo o nada* en otra hebra.

Es decir, el otro proceso verá la escritura completa de, digamos, un valor de 32 bits. No la mitad. No hay forma de que un subproceso interrumpa a otro que está en medio de una escritura atómica multibyte.

Es casi como si hubiera un pequeño bloqueo en torno a la obtención y el establecimiento de esa variable. (¡Y *podría* haberlo! Ver Variables atómicas libres de bloqueo, más abajo).

Y en esa nota, usted puede conseguir lejos con nunca usando atomics si usted utiliza mutexes para trabar sus secciones críticas. Es sólo que hay una clase de *estructuras de datos libres de bloqueo* que siempre permiten a otros hilos progresar en lugar de ser bloqueados por un mutex… pero son difíciles de crear correctamente desde cero, y son una de las cosas que están más allá del alcance de la guía, lamentablemente.

Eso es sólo una parte de la historia. Pero es la parte con la que empezaremos.

Antes de continuar, ¿cómo se declara que una variable es atómica?

Primero, incluye `<stdatomic.h>`.

Esto nos da tipos como `atomic_int`.

Y entonces podemos simplemente declarar variables para que sean de ese tipo.

Pero hagamos una demostración donde tenemos dos hilos. El primero se ejecuta durante un tiempo y luego establece una variable a un valor específico, luego sale. El otro se ejecuta hasta que ve que el valor se establece, y luego se sale.

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

atomic_int x;   // ¡EL PODER DE LA ATOMIC! ¡BWHAHAHA!

int thread1(void *arg)
{
    (void)arg;

    printf("Thread 1: Sleeping for 1.5 seconds\n");
    thrd_sleep(&(struct timespec){.tv_sec=1, .tv_nsec=500000000}, NULL);

    printf("Thread 1: Setting x to 3490\n");
    x = 3490;

    printf("Thread 1: Exiting\n");
    return 0;
}

int thread2(void *arg)
{
    (void)arg;

    printf("Thread 2: Waiting for 3490\n");
    while (x != 3490) {}  // spin here

    printf("Thread 2: Got 3490--exiting!\n");
    return 0;
```

```
}

int main(void)
{
    x = 0;

    thrd_t t1, t2;

    thrd_create(&t1, thread1, NULL);
    thrd_create(&t2, thread2, NULL);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);

    printf("Main    : Threads are done, so x better be 3490\n");
    printf("Main    : And indeed, x == %d\n", x);
}
```

El segundo hilo gira en su lugar, mirando la bandera y esperando a que se establezca en el valor `3490`. Y el primero lo hace.

Y obtengo esta salida:

```
Thread 1: Sleeping for 1.5 seconds
Thread 2: Waiting for 3490
Thread 1: Setting x to 3490
Thread 1: Exiting
Thread 2: Got 3490--exiting!
Main    : Threads are done, so x better be 3490
Main    : And indeed, x == 3490
```

¡Mira, ma! ¡Estamos accediendo a una variable desde diferentes hilos y sin usar un mutex! Y eso funcionará siempre gracias a la naturaleza atómica de las variables atómicas.

Te estarás preguntando qué pasa si en vez de eso es un `int` normal no atómico. Bueno, en mi sistema sigue funcionando… a menos que haga una compilación optimizada, en cuyo caso se cuelga en el hilo 2 esperando a que se establezca el 3490[2].

Pero esto es sólo el principio de la historia. La siguiente parte va a requerir más poder mental y tiene que ver con algo llamado *sincronización*.

## 40.3 Sincronización

La siguiente parte de nuestra historia trata sobre cuándo ciertas escrituras de memoria en un hilo se hacen visibles para las de otro hilo.

Podrías pensar que es inmediatamente, ¿verdad? Pero no es así. Varias cosas pueden ir mal. Raramente mal.

El compilador puede haber reordenado los accesos a memoria de modo que cuando crees que estableces un valor relativo a otro puede no ser cierto. E incluso si el compilador no lo hizo, tu CPU podría haberlo hecho

[2] La razón de esto es que cuando está optimizado, mi compilador ha puesto el valor de `x` en un registro para hacer que el bucle `while` sea rápido. Pero el registro no tiene forma de saber que la variable fue actualizada en otro hilo, así que nunca ve el `3490`. Esto no está realmente relacionado con la parte *todo o nada* de la atomicidad, sino que está más relacionado con los aspectos de sincronización de la siguiente sección

sobre la marcha. O puede que haya algo más en esta arquitectura que haga que las escrituras en una CPU se retrasen antes de ser visibles en otra.

La buena noticia es que podemos condensar todos estos problemas potenciales en uno: los accesos no sincronizados a la memoria pueden aparecer fuera de orden dependiendo del hilo que esté haciendo la observación, como si las propias líneas de código hubieran sido reordenadas.

A modo de ejemplo, ¿qué ocurre primero en el siguiente código, la escritura en `x` o la escritura en `y`?

```
int x, y;  // global

// ...

x = 2;
y = 3;

printf("%d %d\n", x, y);
```

Respuesta: no lo sabemos. El compilador o la CPU podrían invertir silenciosamente las líneas 5 y 6 y no nos daríamos cuenta. El código se ejecutaría con un único hilo *como si* se ejecutara en el orden del código.

En un escenario multihilo, podríamos tener algo como este pseudocódigo:

```
int x = 0, y = 0;

thread1() {
    x = 2;
    y = 3;
}

thread2() {
    while (y != 3) {}  // spin
    printf("x is now %d\n", x);  // 2? ...or 0?
}
```

¿Cuál es la salida del hilo 2?

Bueno, si a `x` se le asigna `2` *antes* de que a `y` se le asigne `3`, entonces esperaría que la salida fuera la muy sensata:

```
x is now 2
```

Pero algo astuto podría reordenar las líneas 4 y 5 haciendo que veamos el valor de `0` para `x` cuando lo imprimamos.

En otras palabras, todo está perdido a menos que podamos decir de alguna manera: “A partir de este punto, espero que todas las escrituras anteriores en otro hilo sean visibles en este hilo”.

Dos hilos *sincronizan* cuando coinciden en el estado de la memoria compartida. Como hemos visto, no siempre están de acuerdo con el código. Entonces, ¿cómo se ponen de acuerdo?

El uso de variables atómicas puede forzar el acuerdo[3]. Si un hilo escribe en una variable atómica, está diciendo “cualquiera que lea esta variable atómica en el futuro también verá todos los cambios que hice en la memoria (atómica o no) hasta la variable atómica inclusive”.

[3] Hasta que diga lo contrario, estoy hablando en general de operaciones *secuencialmente consistentes*. Más sobre lo que eso significa pronto

O, en términos más humanos, sentémonos a la mesa de conferencias y asegurémonos de que estamos de acuerdo en qué partes de la memoria compartida contienen qué valores. Estás de acuerdo en que los cambios de memoria que has hecho hasta e incluyendo el almacenamiento atómico serán visibles para mí después de que haga una carga de la misma variable atómica.

Así que podemos arreglar fácilmente nuestro ejemplo:

```
int x = 0;
atomic int y = 0;  // Make y atomic

thread1() {
    x = 2;
    y = 3;             // Sincronizar al escribir
}

thread2() {
    while (y != 3) {}  // Sincronizar en lectura
    printf("x is now %d\n", x);  // 2, period.
}
```

Como los hilos se sincronizan a través de `y`, todas las escrituras en el hilo 1 que ocurrieron *antes* de la escritura en `y` son visibles en el hilo 2 *después* de la lectura de `y` (en el bucle `while`).

Es importante tener en cuenta un par de cosas aquí:

1. Nada duerme. La sincronización no es una operación de bloqueo. Ambos hilos están funcionando a toda máquina hasta que salen. Incluso el que está atascado en el bucle no está bloqueando la ejecución de ningún otro.
2. La sincronización ocurre cuando un hilo lee una variable atómica que otro hilo escribió. Así que cuando el hilo 2 lee `y`, todas las escrituras de memoria anteriores en el hilo 1 (es decir, la configuración de `x`) serán visibles en el hilo 2.
3. Observa que `x` no es atómica. Eso está bien porque no estamos sincronizando sobre `x`, y la sincronización sobre `y` cuando la escribimos en el hilo 1 significa que todas las escrituras previas - incluyendo `x` - en el hilo 1 serán visibles para otros hilos… si esos otros hilos leen `y` para sincronizarse.

Forzar esta sincronización es ineficiente y puede ser mucho más lento que usar una variable normal. Esta es la razón por la que no usamos atomics a menos que sea necesario para una aplicación en particular.

Esto es lo básico. Profundicemos un poco más.

## 40.4 Adquirir y Liberar

Más terminología. Vale la pena aprender esto ahora.

Cuando un hilo lee una variable atómica, se dice que es una operación de *adquisición*.

Cuando un hilo escribe una variable atómica, se dice que es una operación de *liberación*.

¿Qué es esto? Vamos a alinearlas con los términos que ya conoces cuando se trata de variables atómicas:

**Leer = Cargar = Adquirir**. Como cuando comparas una variable atómica o la lees para copiarla a otro valor.

**Escribir = Almacenar = Liberar**. Como cuando asignas un valor a una variable atómica.

Cuando se usan variables atómicas con esta semántica de adquisición/liberación, C especifica qué puede ocurrir cuándo.

La adquisición/liberación es la base de la sincronización de la que acabamos de hablar.

Cuando un hilo adquiere una variable atómica, puede ver los valores establecidos en otro hilo que liberó esa misma variable.

En otras palabras:

Cuando un hilo lee una variable atómica, puede ver los valores establecidos en otro hilo que escribió en esa misma variable.

La sincronización se produce a través del par acquire/release.

Más detalles:

Con lectura/carga/adquisición de una variable atómica particular:

- Todas las escrituras (atómicas o no atómicas) en otro hilo que ocurrieron antes de que ese otro hilo escribiera/almacenara/liberara esta variable atómica son ahora visibles en este hilo.
- El nuevo valor de la variable atómica establecida por el otro hilo también es visible en este hilo.
- Ninguna lectura o escritura de cualquier variable/memoria en el hilo actual puede ser reordenada para ocurrir antes de esta adquisición.
- La adquisición actúa como una barrera unidireccional cuando se trata de reordenar código; las lecturas y escrituras en el hilo actual pueden moverse de *antes* de la adquisición a *después* de ella. Pero, más importante para la sincronización, nada puede moverse hacia arriba desde *después* de la adquisición a *antes* de ella.

Con escritura/almacenamiento/liberación de una variable atómica particular:

- Todas las escrituras (atómicas o no atómicas) en el subproceso actual que se produjeron antes de esta liberación se vuelven visibles para otros subprocesos que han leído/cargado/adquirido la misma variable atómica.
- El valor escrito en esta variable atómica por este hilo también es visible para otros hilos.
- Ninguna lectura o escritura de cualquier variable/memoria en el hilo actual puede ser reordenada para que ocurra después de esta liberación.
- La liberación actúa como una barrera unidireccional cuando se trata de reordenar código: las lecturas y escrituras en el hilo actual pueden moverse de *después* de la liberación a *antes* de ella. Pero, lo que es más importante para la sincronización, nada puede moverse hacia abajo desde *antes* de la liberación a *después* de ella.

De nuevo, el resultado es la sincronización de la memoria de un subproceso a otro. El segundo hilo puede estar seguro de que las variables y la memoria se escriben en el orden previsto por el programador.

```
int x, y, z = 0;
atomic_int a = 0;

thread1() {
    x = 10;
    y = 20;
    a = 999;  // Liberación
    z = 30;
}

thread2()
{
    while (a != 999) { } // Adquirir
```

```
    assert(x == 10);  // nunca se afirma, x es siempre 10
    assert(y == 20);  // nunca se afirma, y es siempre 20

    assert(z == 0);  // ¡¡podría afirmarlo!!
}
```

En el ejemplo anterior, `thread2` puede estar seguro de los valores de `x` y `y` después de adquirir `a` porque fueron establecidos antes de que `thread1` liberara el atómico `a`.

Pero `thread2` no puede estar seguro del valor de `z` porque ocurrió después de la liberación. Quizás la asignación a `z` se movió antes que la asignación a `a`.

Una nota importante: liberar una variable atómica no tiene efecto sobre las adquisiciones de diferentes variables atómicas. Cada variable está aislada de las demás.

## 40.5 Consistencia secuencial

¿Estás aguantando? Estamos a través de la carne de la utilización más simple de atómica. Y como ni siquiera vamos a hablar aquí de los usos más complejos, puedes relajarte un poco.

La consistencia secuencial es lo que se llama un ordenamiento de memoria. Hay muchos ordenamientos de memoria, pero la consistencia secuencial es la más sana[4] que C tiene para ofrecer. También es la predeterminada. Tienes que salir de tu camino para usar otros ordenamientos de memoria.

Todo lo que hemos estado hablando hasta ahora ha sucedido dentro del ámbito de la consistencia secuencial.

Hemos hablado de cómo el compilador o la CPU pueden reordenar las lecturas y escrituras de memoria en un único hilo siempre que siga la regla *as-if*.

Y hemos visto cómo podemos frenar este comportamiento sincronizando sobre variables atómicas.

Formalicemos un poco más.

Si las operaciones son *secuencialmente consistentes*, significa que al final del día, cuando todo está dicho y hecho, todos los hilos pueden levantar sus pies, abrir su bebida de elección, y todos están de acuerdo en el orden en que los cambios de memoria se produjeron durante la ejecución. Y ese orden es el especificado por el código.

Uno no dirá: "¿Pero *B* no sucedió antes que *A*?" si el resto dice: "*A* definitivamente sucedió antes que *B*". Aquí todos son amigos.

En particular, dentro de un hilo, ninguna de las adquisiciones y liberaciones puede reordenarse entre sí. Esto se suma a las reglas sobre qué otros accesos a memoria pueden reordenarse a su alrededor.

Esta regla da un nivel adicional de cordura a la progresión de cargas/adquisiciones y almacenamientos/liberación atómicos.

Cualquier otro orden de memoria en C implica una relajación de las reglas de reordenación, ya sea para adquisiciones/liberaciones u otros accesos a memoria, atómicos o no. Lo harías si *realmente* supieras lo que estás haciendo y necesitaras el aumento de velocidad. Aquí hay ejércitos de dragones…

Hablaremos de ello más adelante, pero por ahora vamos a ceñirnos a lo seguro y práctico.

## 40.6 Asignaciones y operadores atómicos

Algunos operadores sobre variables atómicas son atómicos. Y otros no lo son.

[4] Más sana desde la perspectiva del programador.

Empecemos con un contraejemplo:

```
atomic_int x = 0;

thread1() {
    x = x + 3;  // NOT atomic!
}
```

Dado que hay una lectura de `x` a la derecha de la asignación y una escritura efectiva a la izquierda, se trata de dos operaciones. Otro hilo podría colarse en medio y hacerte infeliz.

Pero *puedes* usar la abreviatura `+=` para obtener una operación atómica:

```
atomic_int x = 0;

thread1() {
    x += 3;   // ATOMIC!
}
```

En ese caso, `x` se incrementará atómicamente en `3`–ningún otro hilo puede saltar en medio.

En particular, los siguientes operadores son operaciones atómicas de lectura-modificación-escritura con consistencia secuencial, así que úsalos con alegre abandono. (En el ejemplo, `a` es atómico).

```
a++       a--       --a       ++a
a += b    a -= b    a *= b    a /= b    a %= b
a &= b    a |= b    a ^= b    a >>= b   a <<= b
```

## 40.7 Funciones de biblioteca que se sincronizan automáticamente

Hasta ahora hemos hablado de cómo se puede sincronizar con variables atómicas, pero resulta que hay algunas funciones de biblioteca que hacen algunas limitadas detrás de las escenas de sincronización, ellos mismos.

```
call_once()      thrd_create()       thrd_join()
mtx_lock()       mtx_timedlock()     mtx_trylock()
malloc()         calloc()            realloc()
aligned_alloc()
```

**`call_once()`**—Sincroniza con todas las llamadas posteriores a `call_once()` para una bandera en particular. De esta forma, las llamadas posteriores pueden estar seguras de que si otro hilo establece la bandera, la verán.

**`thrd_create()`**—Sincroniza con el inicio del nuevo hilo, que puede estar seguro de que verá todas las escrituras en memoria compartida del hilo padre desde antes de la llamada a `thrd_create()`.

**`thrd_join()`**Cuando un hilo muere, se sincroniza con esta función. La hebra que ha llamado a `thrd_join()` puede estar segura de que puede ver todas las escrituras compartidas de la hebra fallecida.

**`mtx_lock()`**—Las llamadas anteriores a `mtx_unlock()` en el mismo mutex se sincronizan en esta llamada. Este es el caso que más refleja el proceso de adquisición/liberación del que ya hemos hablado. `mtx_unlock()` realiza una liberación en la variable mutex, asegurando que cualquier hilo posterior que haga una adquisición con `mtx_lock()` pueda ver todos los cambios de memoria compartida en la sección crítica.

**`mtx_timedlock()`** y **`mtx_trylock()`**—Similar a la situación con `mtx_lock()`, si esta llamada tiene éxito, las llamadas anteriores a `mtx_unlock()` se sincronizan con ésta.

**Funciones de memoria dinámica**: si asignas memoria, se sincroniza con la anterior liberación de esa misma memoria. Y las asignaciones y desasignaciones de esa región de memoria en particular ocurren en un único orden total que todos los hilos pueden acordar. Creo que la idea aquí es que la desasignación puede limpiar la región si lo desea, y queremos estar seguros de que una asignación posterior no vea los datos no limpiados. Que alguien me diga si hay algo más.

## 40.8 Especificador de Tipo Atómico, Calificador

Bajemos un poco el nivel y veamos qué tipos tenemos disponibles, y cómo podemos incluso crear nuevos tipos atómicos.

Lo primero es lo primero, echemos un vistazo a los tipos atómicos incorporados y a lo que son `typedef`. (Spoiler: `_Atomic` es un calificador de tipo)

| Atomic type | Longhand equivalent |
|---|---|
| `atomic_bool` | `_Atomic _Bool` |
| `atomic_char` | `_Atomic char` |
| `atomic_schar` | `_Atomic signed char` |
| `atomic_uchar` | `_Atomic unsigned char` |
| `atomic_short` | `_Atomic short` |
| `atomic_ushort` | `_Atomic unsigned short` |
| `atomic_int` | `_Atomic int` |
| `atomic_uint` | `_Atomic unsigned int` |
| `atomic_long` | `_Atomic long` |
| `atomic_ulong` | `_Atomic unsigned long` |
| `atomic_llong` | `_Atomic long long` |
| `atomic_ullong` | `_Atomic unsigned long long` |
| `atomic_char16_t` | `_Atomic char16_t` |
| `atomic_char32_t` | `_Atomic char32_t` |
| `atomic_wchar_t` | `_Atomic wchar_t` |
| `atomic_int_least8_t` | `_Atomic int_least8_t` |
| `atomic_uint_least8_t` | `_Atomic uint_least8_t` |
| `atomic_int_least16_t` | `_Atomic int_least16_t` |
| `atomic_uint_least16_t` | `_Atomic uint_least16_t` |
| `atomic_int_least32_t` | `_Atomic int_least32_t` |
| `atomic_uint_least32_t` | `_Atomic uint_least32_t` |
| `atomic_int_least64_t` | `_Atomic int_least64_t` |
| `atomic_uint_least64_t` | `_Atomic uint_least64_t` |
| `atomic_int_fast8_t` | `_Atomic int_fast8_t` |
| `atomic_uint_fast8_t` | `_Atomic uint_fast8_t` |
| `atomic_int_fast16_t` | `_Atomic int_fast16_t` |
| `atomic_uint_fast16_t` | `_Atomic uint_fast16_t` |
| `atomic_int_fast32_t` | `_Atomic int_fast32_t` |
| `atomic_uint_fast32_t` | `_Atomic uint_fast32_t` |
| `atomic_int_fast64_t` | `_Atomic int_fast64_t` |
| `atomic_uint_fast64_t` | `_Atomic uint_fast64_t` |
| `atomic_intptr_t` | `_Atomic intptr_t` |
| `atomic_uintptr_t` | `_Atomic uintptr_t` |
| `atomic_size_t` | `_Atomic size_t` |
| `atomic_ptrdiff_t` | `_Atomic ptrdiff_t` |

| Atomic type | Longhand equivalent |
|---|---|
| `atomic_intmax_t` | `_Atomic intmax_t` |
| `atomic_uintmax_t` | `_Atomic uintmax_t` |

Utilízalos cuando quieras. Son consistentes con los alias atómicos que se encuentran en C++, si eso ayuda.

Pero, ¿y si quieres más?

Puedes hacerlo con un calificador de tipo o un especificador de tipo.

En primer lugar, el especificador. Es la palabra clave `_Atomic` con un tipo en paréntesis después[5]—apto para usarse con `typedef`:

```
typedef _Atomic(double) atomic_double;

atomic_double f;
```

Restricciones en el especificador: el tipo que está haciendo atómico no puede ser de tipo array o función, ni puede ser atómico o calificado de otra manera.

Siguiente, ¡calificador! Es la palabra clave `_Atomic` *sin* un tipo entre paréntesis.

Así que hacen cosas similares[6]:

```
_Atomic(int) i;   // especificador de tipo
_Atomic int  j;   // especificador de tipo
```

Lo que ocurre es que puedes incluir otros calificadores de tipo con este último:

```
_Atomic volatile int k;   // variable atómica cualificada
```

Restricciones en el calificador: el tipo que estás haciendo atómico no puede ser de tipo array o función.

## 40.9 Variables atómicas sin bloqueo

Las arquitecturas de hardware están limitadas en la cantidad de datos que pueden leer y escribir atómicamente. Depende de cómo esté cableado. Y varía.

Si usas un tipo atómico, puedes estar seguro de que los accesos a ese tipo serán atómicos… pero hay una trampa: si el hardware no puede hacerlo, se hace con un bloqueo, en su lugar.

Así que el acceso atómico se convierte en bloqueo-acceso-desbloqueo, lo que es bastante más lento y tiene algunas implicaciones para los manejadores de señales.

Banderas atómicas, más abajo, es el único tipo atómico que está garantizado que esté libre de bloqueos en todas las implementaciones conformes. En el mundo típico de los ordenadores de sobremesa/portátiles, es probable que otros tipos más grandes no tengan bloqueos.

Afortunadamente, tenemos un par de maneras de determinar si un tipo en particular es atómico libre de bloqueo o no.

En primer lugar, algunas macros—puedes usarlas en tiempo de compilación con `#if`. Se aplican tanto a tipos con signo como sin signo.

[5] Aparentemente C++23 está añadiendo esto como una macro.

[6] La especificación señala que pueden diferir en tamaño, representación y alineación

| Atomic Type | Lock Free Macro |
|---|---|
| `atomic_bool` | `ATOMIC_BOOL_LOCK_FREE` |
| `atomic_char` | `ATOMIC_CHAR_LOCK_FREE` |
| `atomic_char16_t` | `ATOMIC_CHAR16_T_LOCK_FREE` |
| `atomic_char32_t` | `ATOMIC_CHAR32_T_LOCK_FREE` |
| `atomic_wchar_t` | `ATOMIC_WCHAR_T_LOCK_FREE` |
| `atomic_short` | `ATOMIC_SHORT_LOCK_FREE` |
| `atomic_int` | `ATOMIC_INT_LOCK_FREE` |
| `atomic_long` | `ATOMIC_LONG_LOCK_FREE` |
| `atomic_llong` | `ATOMIC_LLONG_LOCK_FREE` |
| `atomic_intptr_t` | `ATOMIC_POINTER_LOCK_FREE` |

Estas macros pueden tener *tres* valores diferentes:

| Value | Meaning |
|---|---|
| `0` | Never lock-free. |
| `1` | *Sometimes* lock-free. |
| `2` | Always lock-free. |

Espera… ¿cómo puede algo estar *a veces* libre de bloqueos? Esto sólo significa que la respuesta no se conoce en tiempo de compilación, pero podría conocerse en tiempo de ejecución. Tal vez la respuesta varía dependiendo de si se está ejecutando este código en Intel o AMD Genuine, o algo así[7]

Pero siempre se puede probar en tiempo de ejecución con la `atomic_is_lock_free()`. Esta función devuelve verdadero o falso si el tipo en particular es atómico en este momento.

¿Por qué nos importa?

Lock-free es más rápido, así que tal vez hay un problema de velocidad que usted codificaría de otra manera. O quizás necesites usar una variable atómica en un manejador de señales.

### 40.9.1 Manejadores de señales y atómicos sin bloqueo

Si lees o escribes una variable compartida (duración de almacenamiento estático o `_Thread_Local`) en un manejador de señales, es un comportamiento indefinido [¡jajaja!]… A menos que hagas una de las siguientes cosas

1. Escribir en una variable de tipo `volatile sig_atomic_t`.
2. Leer o escribir en una variable atómica sin bloqueo.

Hasta donde yo sé, las variables atómicas sin bloqueo son una de las pocas formas de obtener información de un manejador de señales de forma portable.

La especificación es un poco vaga, en mi lectura, sobre el orden de memoria cuando se trata de adquirir o liberar variables atómicas en el manejador de señales. C++ dice, y tiene sentido, que tales accesos no tienen secuencia con respecto al resto del programa[8]. La señal puede ser levantada, después de todo, en cualquier momento. Así que asumo que el comportamiento de C es similar.

[7] Acabo de sacar ese ejemplo de la nada. Quizás no importe en Intel/AMD, pero podría importar en algún sitio, ¡maldita sea!

[8] C++ elabora que si la señal es el resultado de una llamada a `raise()`, se secuencia *después* de la función `raise()`

## 40.10 Banderas atómicas

Sólo hay un tipo que el estándar garantiza que será un atómico sin bloqueo: `atomic_flag`. Es un tipo opaco para operaciones test-and-set[9].

Puede ser *set* o *clear*. Puedes inicializarlo a clear con:

```
atomic_flag f = ATOMIC_FLAG_INIT;
```

Puede establecer la bandera atómicamente con `atomic_flag_test_and_set()`, que establecerá la bandera y devolverá su estado anterior como `_Bool` (true para set).

Puede borrar la bandera atómicamente con `atomic_flag_clear()`.

Este es un ejemplo en el que initamos la bandera a limpiar, la establecemos dos veces y la volvemos a limpiar.

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

atomic_flag f = ATOMIC_FLAG_INIT;

int main(void)
{
    bool r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 0

    r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 1

    atomic_flag_clear(&f);
    r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 0
}
```

## 40.11 Estructuras y uniones atómicas(Atomic `structs` and `unions`)

Usando el cualificador o especificador `_Atomic`, ¡puedes hacer `structs` o `unions` atómicas! Bastante asombroso.

Si no hay muchos datos (por ejemplo, un puñado de bytes), el tipo atómico resultante puede estar libre de bloqueos. Compruébalo con `atomic_is_lock_free()`.

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

int main(void)
{
    struct point {
        float x, y;
    };
```

[9] https://en.wikipedia.org/wiki/Test-and-set

```
    _Atomic(struct point) p;

    printf("Is lock free: %d\n", atomic_is_lock_free(&p));
}
```

Aquí está el truco: no puedes acceder a los campos de una `struct` o `union` atómica… ¿entonces qué sentido tiene? Bueno, puedes *copiar* atómicamente toda la `struct` en una variable no atómica y luego usarla. También puedes copiar atómicamente a la inversa.

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

int main(void)
{
    struct point {
        float x, y;
    };

    _Atomic(struct point) p;
    struct point t;

    p = (struct point){1, 2};  // Atomic copy

    //printf("%f\n", p.x);  // Error

    t = p;   // Atomic copy

    printf("%f\n", t.x);  // OK!
}
```

También puede declarar una `estructura` en la que los campos individuales sean atómicos. La implementación define si los tipos atómicos están permitidos en los campos de bits.

## 40.12 Punteros Atómicos

Sólo una nota sobre la colocación de `_Atomic` cuando se trata de punteros.

En primer lugar, los punteros a atómicos (es decir, el valor del puntero no es atómico, pero la cosa a la que apunta sí lo es):

```
_Atomic int x;
_Atomic int *p;  // p es un puntero a un int atómico

p = &x;  // OK!
```

En segundo lugar, los punteros atómicos a valores no atómicos (es decir, el valor del puntero en sí es atómico, pero la cosa a la que apunta no lo es):

```
int x;
int * _Atomic p;  // p es un puntero atómico a un int

p = &x;  // OK!
```

Por último, punteros atómicos a valores atómicos (es decir, el puntero y la cosa a la que apunta son ambos atómicos):

```
_Atomic int x;
_Atomic int * _Atomic p;  // p es un puntero atómico a un int atómico

p = &x;  // OK!
```

## 40.13 Orden de Memoria

Ya hemos hablado de la coherencia secuencial, que es la más sensata de todas. Pero hay muchas otras:

| `memory_order` | Description |
|---|---|
| `memory_order_seq_cst` | Sequential Consistency |
| `memory_order_acq_rel` | Acquire/Release |
| `memory_order_release` | Release |
| `memory_order_acquire` | Acquire |
| `memory_order_consume` | Consume |
| `memory_order_relaxed` | Relaxed |

Puede especificar otras con determinadas funciones de biblioteca. Por ejemplo, puedes añadir un valor a una variable atómica así:

```
atomic_int x = 0;

x += 5;  // Consistencia secuencial, por defecto
```

O puede hacer lo mismo con esta función de biblioteca:

```
atomic_int x = 0;

atomic_fetch_add(&x, 5);  // Consistencia secuencial, por defecto
```

O puedes hacer lo mismo con una ordenación explícita de la memoria:

```
atomic_int x = 0;

atomic_fetch_add_explicit(&x, 5, memory_order_seq_cst);
```

Pero, ¿y si no quisiéramos coherencia secuencial? Y quisieras adquirir / liberar en su lugar por cualquier razón? Sólo dilo:

```
atomic_int x = 0;

atomic_fetch_add_explicit(&x, 5, memory_order_acq_rel);
```

A continuación haremos un desglose de los diferentes órdenes de memoria. No te metas con nada que no sea consistencia secuencial a menos que sepas lo que estás haciendo. Es realmente fácil cometer errores que causarán fallos raros y difíciles de reproducir.

### 40.13.1 Consistencia Secuencial

- Adquisición de operaciones de carga (véase más adelante).
- Las operaciones de almacenamiento se liberan (véase más abajo).
- Las operaciones de lectura-modificación-escritura adquieren y luego liberan.

Además, para mantener el orden total de las adquisiciones y liberaciones, ninguna adquisición o liberación se reordenará entre sí. (Las reglas de adquisición/liberación no prohíben reordenar una liberación seguida de una adquisición. Pero las reglas de coherencia secuencial sí lo hacen).

### 40.13.2 Acquire

Esto es lo que ocurre en una operación de carga/lectura de una variable atómica.

- Si otro hilo liberó esta variable atómica, todas las escrituras que ese hilo hizo son ahora visibles en este hilo.
- Los accesos a memoria en este thread que ocurran después de esta carga no pueden ser reordenados antes.

### 40.13.3 Release

Esto es lo que ocurre al almacenar/escribir una variable atómica.

- Si otro hilo adquiere más tarde esta variable atómica, toda la memoria en este hilo antes de su escritura atómica se vuelven visibles para ese otra hebra.
- Los accesos a memoria en este hilo que ocurran antes de la liberación no pueden reordenarse después.

### 40.13.4 Consume

Esta es una extraña, similar a una versión menos estricta de adquirir. Afecta a los accesos a memoria que son *data dependent* de la variable atómica.

Ser "dependiente de datos" significa vagamente que la variable atómica se utiliza en un cálculo.

Es decir, si un hilo consume una variable atómica entonces todas las operaciones en ese hilo que utilicen esa variable atómica podrán ver las escrituras de memoria en el hilo que la libera.

Compárese con adquirir donde las escrituras en memoria en el subproceso que libera serán visibles para *todas* las operaciones en el subproceso actual, no sólo las que dependen de los datos. las dependientes de datos.

También como en acquire, hay una restricción sobre qué operaciones pueden ser reordenadas *antes* de consumir. Con acquire, no se podía reordenar nada antes. Con consume, no puedes reordenar nada que dependa del valor atómico cargado antes de él.

### 40.13.5 Acquire/Release

Esto sólo se aplica a las operaciones de lectura-modificación-escritura. Es una adquisición y liberación en uno.

- Una adquisición ocurre para la lectura.
- Una liberación ocurre para la escritura.

### 40.13.6 Relaxed

Sin reglas; ¡es la anarquía! Perros y gatos viviendo juntos… ¡histeria colectiva!

En realidad, hay una regla. Las lecturas y escrituras atómicas siguen siendo "todo o nada". Pero las operaciones pueden reordenarse caprichosamente y hay cero sincronización entre hilos.

Hay algunos casos de uso para este orden de memoria, que puedes encontrar con un poco de búsqueda, por ejemplo, contadores simples.

Y puedes usar una valla para forzar la sincronización después de un montón de escrituras relajadas.

## 40.14 Fences

¿Sabes que las liberaciones y adquisiciones de variables atómicas se producen al leerlas y escribirlas?

Bueno, es posible hacer una liberación o adquisición *sin* una variable atómica, también.

Esto se llama un *fence*. Así que si quieres que todas las escrituras en un hilo sean visibles en otro lugar, puedes poner una valla de liberación en un hilo y una valla de adquisición en otro, igual que con el funcionamiento de las variables atómicas.

Como una operación consume no tiene sentido en un vallado[10], `memory_order_consume` se trata como una adquisición.

Usted puede poner una cerca con cualquier orden especificado:

```
atomic_thread_fence(memory_order_release);
```

También hay una versión ligera de una valla para usar con manejadores de señales, llamada `atomic_signal_fence()`.

Funciona de la misma manera que `atomic_thread_fence()`, excepto que:

- Sólo se ocupa de la visibilidad de valores dentro del mismo hilo; no hay sincronización con otros hilos.
- No se emiten instrucciones hardware fence.

Si quieres estar seguro de que los efectos secundarios de las operaciones no atómicas (y de las operaciones atómicas relajadas) son visibles en el manejador de señales, puedes usar esta valla.

La idea es que el manejador de señales se está ejecutando en *este* hilo, no en otro, por lo que esta es una forma más ligera de asegurarse de que los cambios fuera del manejador de señales son visibles dentro de él (es decir, que no han sido reordenados).

## 40.15 References

Si quieres aprender más sobre este tema, aquí tienes algunas de las cosas que me ayudaron a superarlo:

- Herb Sutter's `atomic<>` *Weapons* talk:
  - Part 1[11]
  - part 2[12]
- Jeff Preshing's materials[13], in particular:
  - An Introduction to Lock-Free Programming[14]
  - Acquire and Release Semantics[15]
  - The *Happens-Before* Relation[16]

[10] Porque consume se refiere a las operaciones que dependen del valor de la variable atómica adquirida, y no hay variable atómica en un vallado

[11] https://www.youtube.com/watch?v=A8eCGOqgvH4

[12] https://www.youtube.com/watch?v=KeLBd2EJLOU

[13] https://preshing.com/archives/

[14] https://preshing.com/20120612/an-introduction-to-lock-free-programming/

[15] https://preshing.com/20120913/acquire-and-release-semantics/

[16] https://preshing.com/20130702/the-happens-before-relation/

    - The *Synchronizes-With* Relation[17]
    - The Purpose of `memory_order_consume` in C++11[18]
    - You Can Do Any Kind of Atomic Read-Modify-Write Operation[19]
- CPPReference:
    - Memory Order[20]
    - Atomic Types[21]
- Bruce Dawson's Lockless Programming Considerations[22]
- The helpful and knowledgeable folks on r/C_Programming[23]

[17] https://preshing.com/20130823/the-synchronizes-with-relation/
[18] https://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/
[19] https://preshing.com/20150402/you-can-do-any-kind-of-atomic-read-modify-write-operation/
[20] https://en.cppreference.com/w/c/atomic/memory_order
[21] https://en.cppreference.com/w/c/language/atomic
[22] https://docs.microsoft.com/en-us/windows/win32/dxtecharts/lockless-programming
[23] https://www.reddit.com/r/C_Programming/

# Chapter 41

# Especificadores de Función, Especificadores/Operadores de Alineación

En mi experiencia, estos no se utilizan mucho, pero los cubriremos aquí en aras de la exhaustividad.

## 41.1 Especificadores de función

Cuando declaras una función, puedes dar al compilador un par de consejos sobre cómo podrían o serán utilizadas las funciones. Esto permite o anima al compilador a realizar ciertas optimizaciones.

### 41.1.1 `inline` para la Velocidad—tal vez

Puede declarar una función para que sea inline de la siguiente manera:

```
static inline int add(int x, int y) {
    return x + y;
}
```

Esto pretende animar al compilador a hacer esta llamada a la función lo más rápido posible. Y, históricamente, una forma de hacerlo era *inlining*, lo que significa que el cuerpo de la función se incrustaba en su totalidad donde se realizaba la llamada. Esto evitaría toda la sobrecarga de establecer la llamada a la función y desmontarla a expensas de un mayor tamaño del código, ya que la función se copiaba por todas partes en lugar de reutilizarse.

Las cosas rápidas y sucias que hay que recordar son:

1. Probablemente no necesites usar `inline` por velocidad. Los compiladores modernos saben qué es lo mejor.
2. Si lo usas por velocidad, úsalo con ámbito de archivo, es decir, `static inline`. Esto evita las desordenadas reglas de vinculación externa y funciones inline.

Deja de leer esta sección ahora.

Glotón para el castigo, ¿eh?

Vamos a tratar de dejar el `static` off.

```
#include <stdio.h>

inline int add(int x, int y)
{
    return x + y;
}

int main(void)
{
    printf("%d\n", add(1, 2));
}
```

`gcc` da un error de enlazador en `add()`[1]. La especificación requiere que si tienes una función en línea no `externa` también debes proporcionar una versión con enlace externo.

Así que tendrías que tener una versión `externa` en algún otro lugar para que esto funcione. Si el compilador tiene tanto una función `inline` en el fichero actual como una versión externa de la misma función en otro lugar, puede elegir a cuál llamar. Así que recomiendo encarecidamente que sean la misma.

Otra cosa que puedes hacer es declarar la función como `extern inline`. Esto intentará inline en el mismo archivo (por velocidad), pero también creará una versión con enlace externo.

### 41.1.2 `noreturn` y `_Noreturn`

Esto indica al compilador que una función concreta no volverá nunca a su invocador, es decir, que el programa saldrá por algún mecanismo antes de que la función retorne.

Esto permite al compilador realizar algunas optimizaciones en torno a la llamada a la función.

También le permite indicar a otros desarrolladores que cierta lógica del programa depende de que una función *no* regrese.

Es probable que nunca necesite usar esto, pero lo verá en algunas llamadas a bibliotecas como `exit()`[2] y `abort()`[3].

La palabra clave incorporada es `_Noreturn`, pero si no rompe su código existente, todo el mundo recomendaría incluir `<stdnoreturn.h>` y usar la más fácil de leer `noreturn` en su lugar.

Es un comportamiento indefinido si una función especificada como `noreturn` realmente retorna. Es computacionalmente deshonesto.

Aquí hay un ejemplo de uso correcto de `noreturn`:

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

noreturn void foo(void) // ¡Esta función nunca debe retornar!
{
    printf("Happy days\n");

    exit(1);            // Y no vuelve... ¡Sale por aquí!
```

[1] ¡A menos que compiles con las optimizaciones activadas (probablemente)! Pero creo que cuando hace esto, no se está comportando según la especificación

[2] https://beej.us/guide/bgclr/html/split/stdlib.html#man-exit

[3] https://beej.us/guide/bgclr/html/split/stdlib.html#man-abort

```
}

int main(void)
{
    foo();
}
```

Si el compilador detecta que una función `noreturn` podría retornar, podría advertirte, de forma útil.

Sustituyendo la función `foo()` por esto:

```
noreturn void foo(void)
{
    printf("Breakin' the law\n");
}
```

me da una advertencia:

```
foo.c:7:1: warning: function declared 'noreturn' should not return
```

## 41.2 Especificadores y operadores de alineación

*Alignment*[4] se refiere a los múltiplos de direcciones en los que se pueden almacenar objetos. ¿Se puede almacenar en cualquier dirección? ¿O debe ser una dirección inicial divisible por 2? ¿O por 8? ¿O 16?

Si estás programando algo de bajo nivel, como un asignador de memoria que interactúa con tu sistema operativo, puede que tengas que tener esto en cuenta. La mayoría de los desarrolladores pasan sus carreras sin utilizar esta funcionalidad en C.

### 41.2.1 `alignas` y `_Alignas`

No es una función. Más bien, es un *especificador de alineación* que puedes usar con una declaración de variable.

El especificador incorporado es `_Alignas`, pero la cabecera `<stdalign.h>` lo define como `alignas` para que se vea mejor.

Si necesitas que tu `char` esté alineado como un `int`, puedes forzarlo así cuando lo declares:

```
char alignas(int) c;
```

También puede pasar un valor constante o una expresión para la alineación. Esto tiene que ser algo soportado por el sistema, pero la especificación no llega a dictar qué valores se pueden poner ahí. Las potencias pequeñas de 2 (1, 2, 4, 8 y 16) suelen ser apuestas seguras.

```
char alignas(8) c;   // alinear en límites de 8 bytes
```

Si quiere alinear al máximo alineamiento usado por su sistema, incluya `<stddef.h>` y use el tipo `max_align_t`, así:

```
char alignas(max_align_t) c;
```

[4] https://en.wikipedia.org/wiki/Data_structure_alignment

Usted podría potencialmente *sobre-alinear* especificando una alineación mayor que la de `max_align_t`, pero si tales cosas están o no permitidas depende del sistema.

### 41.2.2 `alignof` y `_Alignof`

Este operador devolverá el múltiplo de dirección que un tipo particular utiliza para la alineación en este sistema. Por ejemplo, puede que `char`s se alinee cada 1 dirección, y `int`s se alinee cada 4 direcciones.

El operador incorporado es `_Alignof`, pero la cabecera `<stdalign.h>` lo define como `alignof` si quieres parecer más guay.

Aquí hay un programa que imprimirá las alineaciones de una variedad de tipos diferentes. De nuevo, estos variarán de un sistema a otro. Tenga en cuenta que el tipo `max_align_t` le dará la alineación máxima utilizada por el sistema.

```
#include <stdalign.h>
#include <stdio.h>     // for printf()
#include <stddef.h>    // for max_align_t

struct t {
    int a;
    char b;
    float c;
};

int main(void)
{
    printf("char       : %zu\n", alignof(char));
    printf("short      : %zu\n", alignof(short));
    printf("int        : %zu\n", alignof(int));
    printf("long       : %zu\n", alignof(long));
    printf("long long  : %zu\n", alignof(long long));
    printf("double     : %zu\n", alignof(double));
    printf("long double: %zu\n", alignof(long double));
    printf("struct t   : %zu\n", alignof(struct t));
    printf("max_align_t: %zu\n", alignof(max_align_t));
}
```

Salida en mi sistema:

```
char       : 1
short      : 2
int        : 4
long       : 8
long long  : 8
double     : 8
long double: 16
struct t   : 16
max_align_t: 16
```

## 41.3 Función `memalignment()`

¡Nuevo en C23!

(Advertencia: ninguno de mis compiladores soporta esta función todavía, así que el código está en gran parte sin probar).

`alignof` es genial si conoces el tipo de tus datos. ¿Pero qué pasa si *desconoce* el tipo y sólo tiene un puntero a los datos?

¿Cómo podría ocurrir eso?

Bueno, con nuestro buen amigo el `void*`, por supuesto. No podemos pasarlo a `alignof`, pero ¿y si necesitamos saber la alineación de lo que apunta?

Podríamos querer saber esto si estamos a punto de usar la memoria para algo que tiene necesidades significativas de alineación. Por ejemplo, los tipos atómicos y flotantes a menudo se comportan mal si están mal alineados.

Así que con esta función podemos comprobar la alineación de algunos datos siempre que tengamos un puntero a esos datos, incluso si es un `void*`.

Hagamos una prueba rápida para ver si un puntero void está bien alineado para usarlo como tipo atómico, y, si es así, hagamos que una variable lo use como ese tipo:

```
void foo(void *p)
{
    if (memalignment(p) >= alignof(atomic int)) {
        atomic int *i = p;
        do_things(i);
    } else
        puts("This pointer is no good as an atomic int\n");

...
```

Sospecho que rara vez (hasta el punto de nunca, probablemente) necesitará utilizar esta función a menos que esté haciendo algunas cosas de bajo nivel.

Y ahí lo tienen. ¡Alineación!

# Index