

LUCAS EGA ACADEMY



LOW-LEVEL POWER



MEM - 64MB - 1.2V



MEM - 64MB - 1.2V



MEM - 64MB - 1.2V



MEM - 64MB - 1.2V

Curso Completo de C

© 2026 Esteban Gutiérrez

Colibrí Editorial EGA

Hecho en México

ÍNDICE

PRÓLOGO	6
A quien va dirigido este libro	6
Cómo está organizado el libro	7
Cómo estudiar este libro	7
Lo que necesitas	8
Una última cosa antes de empezar	8
UNIDAD 1: Tus primeros pasos con C	9
Introducción	9
Explicación detallada	11
Ejemplos paso a paso	28
Práctica	33
Resumen de la unidad	39
UNIDAD 2: Variables, tipos y printf	41
Introducción	41
Explicación detallada	43
Ejemplos paso a paso	59
Práctica	67
Resumen de la unidad	73
UNIDAD 3: Entrada de datos y operadores	76
Introducción	76
Explicación detallada	77
Ejemplos paso a paso	91
Práctica	99
Resumen de la unidad	106
UNIDAD 4: Tomar decisiones	108
Introducción	108
Explicación detallada	109
Ejemplos paso a paso	126
Práctica	136
Resumen de la unidad	146

UNIDAD 5: Bucles: repetir sin cansarte	148
Introducción	148
Explicación detallada	150
Ejemplos paso a paso	166
Práctica	176
Resumen de la unidad	184
UNIDAD 6: Funciones	187
Introducción	187
Explicación detallada	188
Ejemplos paso a paso	206
Práctica	215
Resumen de la unidad	223
UNIDAD 7: Arreglos y cadenas de texto	226
Introducción	226
Explicación detallada	227
Ejemplos paso a paso	242
Práctica	253
Resumen de la unidad	265
UNIDAD 8: Apuntadores	267
Introducción	267
Explicación detallada	269
Ejemplos paso a paso	285
Práctica	294
Resumen de la unidad	302
UNIDAD 9: Structs y memoria dinámica	304
Introducción	304
Explicación detallada	305
Ejemplos paso a paso	323
Práctica	334
Resumen de la unidad	341
UNIDAD 10: Archivos, buenas prácticas y proyecto final	343
Introducción	343

Explicación detallada	344
Ejemplos paso a paso	356
Práctica	372
Resumen de la unidad	386
EXAMEN FINAL	389
Preguntas	389
Respuestas y soluciones	397

PRÓLOGO

Bienvenido a "Curso Completo de C", el libro de Lucas EGA Academy para aprender desde cero el lenguaje que mueve el mundo: C. Este libro nace del curso en video de 30 clases de la academia, pero aquí tienes todo explicado con muchísimo más detalle: conceptos a fondo, analogías de la vida real, errores comunes de principiantes, buenas prácticas de la industria, ejemplos paso a paso para probar en tu computadora, prácticas con soluciones explicadas y un examen final para medir lo que aprendiste.

Si este es tu primer lenguaje de programación, elegiste bien: C te enseña a pensar como piensa la computadora. Si ya programas en Python, Java o JavaScript, C te va a revelar lo que pasa debajo del cofre: la memoria, los apuntadores, la forma en que tus variables viven de verdad en la máquina.

C nació en los laboratorios Bell en 1972, creado por Dennis Ritchie. Más de cincuenta años después sigue vivo en todas partes: sistemas operativos como Windows, Linux y macOS están escritos en C, los microcontroladores de tu lavadora y tu carro usan C, y lenguajes modernos como Python o PHP están contruidos sobre C. Aprender C es aprender el idioma madre de la computación moderna.

A quien va dirigido este libro

Este libro es para ti si empiezas desde cero y quieres un lenguaje que te dé bases solidas de verdad. Es para ti si estudias ingenieria, sistemas o electrónica y necesitas dominar C para tus materias. Es para ti si ya programas en otro lenguaje y quieres entender la memoria, los apuntadores y el rendimiento a bajo nivel.

También es para curiosos que quieren saber cómo funciona una computadora por dentro, para quienes quieren programar microcontroladores y Arduino, y para quienes sueñan con crear sistemas operativos, videojuegos o software embebido. No necesitas matemáticas avanzadas: solo una computadora, curiosidad y ganas de practicar todos los días.

Si vienes de otros cursos de Lucas EGA Academy, este libro encaja perfecto: ya sabes la lógica de programar, y ahora la llevaras al nivel más profundo.

Cómo está organizado el libro

El libro tiene 10 unidades que agrupan las 30 clases del curso en video. Cada unidad cubre tres clases y sigue siempre la misma estructura para que sepas que esperar:

Primero, una introducción extensa que te pone en contexto: que vas a aprender, por qué importa y donde se usa en la vida real. Después, la explicación detallada con analogías, errores comunes que cometen los principiantes y buenas prácticas que usan los programadores profesionales. Luego, ejemplos paso a paso con código que puedes copiar, compilar y correr en tu computadora. Después, una sección de práctica con al menos cinco ejercicios y sus soluciones explicadas. Y al final, un resumen de la unidad con los puntos clave.

Al final del libro hay un examen de 50 preguntas: opción multiple, verdadero o falso, y preguntas abiertas con código. Hazlo sin ver las respuestas primero. Si sacas 90 o más, dominas C a nivel intermedio.

Cómo estudiar este libro

No leas este libro como una novela. C se aprende escribiendo código, no solo leyendolo. Mi recomendación:

1. Instala un compilador desde el día uno. En la Unidad 1 te explico cómo instalar gcc en Windows, Linux y macOS. Todo el libro asume que compilas y corres cada ejemplo.
2. Escribe cada ejemplo con tus manos. No copies y pegues: escribe el código, compilalo, correlo y luego rompelo a propósito. Cambia valores, quita líneas y observa que pasa. Así se aprende de verdad.
3. Haz todos los ejercicios antes de ver las soluciones. Si te atorras más de treinta minutos, mira la solución, entiendela y luego escríbela tu solo sin verla.
4. No avances con dudas. C es un lenguaje donde cada tema se construye sobre el anterior: si no entiendes variables, los apuntadores te van a doler. Repasa la unidad las veces que necesites.
5. Dedicar sesiones cortas pero constantes. Una hora diaria de práctica real vale más que siete horas un solo día.

Lo que necesitas

Una computadora con Windows, Linux o macOS, un compilador de C (gcc, que es gratis) y un editor de texto. Puedes usar Visual Studio Code, que también es gratis, o hasta el Bloc de notas si quieres empezar simple. No necesitas nada más: ni pagar licencias, ni una computadora potente. C compila rapidísimo hasta en máquinas viejas.

Una última cosa antes de empezar

C tiene fama de difícil, y te voy a ser honesto: los apuntadores y la memoria dinámica si exigen pensar. Pero esa dificultad es su regalo: cuando dominas C, cualquier otro lenguaje se te hace fácil, porque ya entiendes lo que pasa debajo. Los mejores programadores del mundo pasaron por C, y tú puedes hacerlo también.

Vamos paso a paso, sin prisa pero sin pausa. Tu primera unidad te espera: la historia de C, tu compilador y tu primer programa. A programar.

UNIDAD 1: Tus primeros pasos con C

Introducción

Bienvenido a tu viaje con el lenguaje de programación C. Si acabas de abrir este libro, probablemente tienes una mezcla de emoción y nervios: emoción porque vas a aprender uno de los lenguajes más influyentes de la historia de la computación, y nervios porque tal vez escuchaste por ahí que C es "difícil". Dejame decirte algo desde el principio: C no es difícil, solo es honesto. No te esconde nada, no hace magia por debajo de la mesa, y eso, aunque al inicio parezca una desventaja, es exactamente lo que lo convierte en el mejor maestro que puedes tener.

C es un lenguaje de programación creado en 1972, lo que significa que tiene más de cincuenta años de vida. Para que te des una idea, en el mundo de la tecnología cincuenta años es una eternidad. Los lenguajes de moda nacen y mueren cada pocos años; frameworks enteros que hoy parecen imprescindibles dentro de una década serán una nota al pie en un libro de historia. Y sin embargo, C sigue aquí, vivo, fuerte y moviendo al mundo. Eso no es casualidad: es la prueba de que quien diseñó este lenguaje entendió algo fundamental sobre cómo funcionan las computadoras.

Piensa en C como los cimientos de un edificio. Cuando entras a un rascacielos moderno con vidrios polarizados, elevadores inteligentes y aire acondicionado central, no ves los cimientos, pero están ahí, sosteniendo todo. Así es C: no lo ves, pero está en todas partes. El sistema operativo de tu computadora, sea Windows, macOS o Linux, tiene su corazón escrito en C. El teléfono que traes en el bolsillo corre un sistema (Android o iOS) cuyo núcleo también está escrito en C. Cuando juegas un videojuego con gráficos impresionantes, el motor que mueve esos gráficos probablemente tiene componentes críticos escritos en C o en su hijo directo, C++. Los microcontroladores dentro de tu lavadora, de tu carro, del control remoto de tu televisión, de los semáforos de tu ciudad: casi todos se programan en C. Los intérpretes de Python, de PHP y de Ruby están escritos en C. Incluso muchos de los lenguajes modernos que hoy están de moda le deben su existencia, directa o indirectamente, a las ideas que C introdujo hace más de medio siglo.

Entonces, ¿por qué aprender C hoy, en un mundo lleno de lenguajes "más fáciles" como Python o JavaScript? La respuesta es sencilla: porque C te enseña como piensa realmente una computadora. Los lenguajes modernos son como manejar un carro automático: pisas el acelerador y avanzas, sin preocuparte por los cambios de velocidad. C es como manejar un carro estándar: al principio te cuesta trabajo coordinar el clutch, pero cuando lo dominas,

entiendes el carro de verdad y puedes manejar cualquier cosa. Un programador que aprendió C entiende la memoria, entiende los apuntadores, entiende por qué un programa es rápido o lento, y cuando después aprende Python, Java, JavaScript o cualquier otro lenguaje, lo aprende al doble de velocidad porque ya conoce lo que pasa "debajo del cofre".

Además, C es el lenguaje de los sistemas embebidos y del hardware. Si alguna vez sonaste con programar un robot, un dron, un dispositivo del Internet de las Cosas, o trabajar en videojuegos, sistemas operativos o ciberseguridad, C no es una opción: es el requisito. Las empresas que contratan para estos puestos buscan gente que domine C porque saben que quien domina C puede dominar cualquier otra cosa.

Otra razón poderosa: C es pequeño. Mientras que otros lenguajes tienen miles de funciones integradas y bibliotecas gigantescas que tienes que memorizar, C cabe en tu cabeza. Sus palabras reservadas son apenas unas treinta y dos. Puedes aprender todo el lenguaje base en cuestión de semanas. Lo que toma tiempo dominar no es la sintaxis, sino la forma de pensar: cómo dividir un problema en partes pequeñas, cómo manejar la memoria con cuidado, cómo escribir código limpio y eficiente. Y esa forma de pensar te va a servir para toda tu carrera, sin importar a qué lenguaje te dediques después.

En esta primera unidad vamos a dar los pasos más importantes de todo el curso: los primeros. Vas a conocer la historia de C, desde los laboratorios Bell en los años setenta hasta el estándar moderno que usamos hoy. Vas a instalar tu compilador (el programa que traduce tu código C a un programa que la computadora puede ejecutar) en Windows, en Linux o en macOS, paso a paso, sin dar nada por sentado. Vas a escribir, compilar y ejecutar tu primer programa: el clásico "Hola, mundo", ese ritual de iniciación que todo programador ha hecho al menos una vez en su vida. Y vas a desarmar ese programa pieza por pieza para entender su anatomía: qué hace la línea que empieza con `#include`, por qué existe la función `main`, para qué sirve el `return 0` del final.

No te preocupes si al principio sientes que hay muchas piezas nuevas. Es normal. Aprender a programar es cómo aprender a cocinar: al principio hasta hervir agua parece complicado porque no sabes dónde están las ollas, cómo se prende la estufa ni cuánto tiempo dejarla. Pero una vez que preparas tu primer platillo, el segundo ya es más fácil, y al decimo ya ni lo piensas. En esta unidad vamos a "hervir agua" juntos, con calma y con ejemplos que puedes probar en tu propia computadora.

Al terminar esta unidad serás capaz de instalar un compilador de C en tu sistema operativo, escribir un programa sencillo en C usando un editor de texto, compilarlo desde la terminal, ejecutarlo y explicar qué hace cada parte del código. Parece poco, pero es la base de absolutamente todo lo que viene después: en la Unidad 2 trabajaremos con variables y con la

entrada y salida de datos, en las unidades siguientes con decisiones, ciclos, funciones, arreglos y apuntadores, hasta llegar al proyecto final de la Unidad 10. Todo ese edificio se construye sobre lo que aprendas aquí.

Un último consejo antes de empezar: programa junto con el libro. No te limites a leer los ejemplos; escríbelos tú mismo, compílalos, ejecútalos, rompelos a propósito y arreglalos. La programación no se aprende leyendo, se aprende haciendo, equivocándose y corrigiendo. Cada error que cometes ahora es una lección que ya no tendrás que aprender después. Y si en algún momento te atorras, recuerda que en lucasegaacademy.com tienes más recursos, cursos y una comunidad que está recorriendo el mismo camino que tu.

Vamos a empezar. Tu aventura con C comienza ahora mismo.

Explicación detallada

La historia de C: como nació el lenguaje que mueve al mundo

Para entender por que C es como es, tienes que conocer su historia, porque cada característica del lenguaje es una respuesta a un problema real que alguien tuvo hace más de cincuenta años. Y como toda buena historia, esta empieza en un laboratorio.

Corre el año 1969. En los Laboratorios Bell, en Murray Hill, Nueva Jersey, un grupo de investigadores brillantes está trabajando en un sistema operativo experimental llamado Unix. Entre ellos hay dos nombres que debes grabarte: Ken Thompson y Dennis Ritchie. Thompson había escrito las primeras versiones de Unix en lenguaje ensamblador, que es el lenguaje más cercano al hardware: potentísimo, pero lentísimo de escribir y un dolor de cabeza para mantener, porque cada procesador tiene su propio ensamblador y el código no se puede llevar de una máquina a otra.

Thompson quería un lenguaje mejor, así que tomó un lenguaje llamado BCPL (Basic Combined Programming Language), creado por Martin Richards, y diseñó su propia versión simplificada a la que llamó B. Con B escribió parte de Unix, pero B tenía un problema serio: era un lenguaje sin tipos de datos. En B, todo era una "palabra" de memoria, un número, y punto. Eso funcionaba en las computadoras para las que se había diseñado, pero cuando los Laboratorios Bell compraron una computadora nueva, la PDP-11 de Digital Equipment Corporation, B empezó a quedarse corto: la PDP-11 manejaba bytes y direcciones de memoria de formas que B no podía expresar con elegancia.

Aquí entra Dennis Ritchie. Entre 1971 y 1973, Ritchie tomó el lenguaje B y lo extendió, agregándole lo que le faltaba: tipos de datos (caracteres, enteros de distintos tamaños,

números decimales), estructuras para agrupar datos relacionados, y un sistema de tipos que permitía al compilador detectar errores antes de ejecutar el programa. A ese nuevo lenguaje lo llamó C, simplemente porque venía después de B. Así de sencillo y así de legendario: el nombre es solo la siguiente letra del abecedario.

En 1973, Ritchie y Thompson reescribieron el núcleo de Unix en C. Este fue un momento histórico: era la primera vez que un sistema operativo completo se escribía en un lenguaje de alto nivel en lugar de ensamblador. Las consecuencias fueron enormes. Primero, Unix se volvió portable: el mismo código podía compilarse en distintas computadoras con cambios mínimos. Segundo, el sistema se volvió mucho más fácil de mantener y mejorar. Tercero, y esto es lo más importante para ti, C demostró que un lenguaje de alto nivel podía ser tan eficiente como el ensamblador si estaba bien diseñado. Esa es la filosofía que C conserva hasta hoy: darte abstracciones útiles sin quitarte el control ni cobrarte rendimiento.

En 1978, Brian Kernighan y Dennis Ritchie publicaron el libro "The C Programming Language", conocido por todos los programadores simplemente como "K&R" (por las iniciales de sus autores). Este libro no solo enseñaba C: durante años FUE la definición del lenguaje. Si querías saber cómo funcionaba C, leías K&R. El libro incluía el famosísimo primer ejemplo, el programa "hola, mundo", que desde entonces se convirtió en el ritual de iniciación de casi todos los lenguajes de programación. La versión de C descrita en ese libro se conoce como "K&R C".

Pero había un problema: como C se hizo popular, cada fabricante de compiladores le agregaba sus propias extensiones y detalles. Un programa que compilaba perfecto en una máquina fallaba en otra. La comunidad necesitaba un estándar, una regla oficial que todos siguieran. Así que en 1983 el instituto ANSI (American National Standards Institute) formó un comité para estandarizar C, y en 1989 se publicó el estándar ANSI C, también conocido como C89. Un año después, la organización internacional ISO lo adoptó como C90. Por primera vez había una definición oficial y universal del lenguaje: si tu compilador seguía el estándar, tu código funcionaba en todas partes.

El lenguaje siguió evolucionando. En 1999 llegó C99, que agregó cosas muy útiles: comentarios de una línea con //, variables declaradas en cualquier parte del código (no solo al inicio de los bloques), el tipo booleano `_Bool`, y los arreglos de longitud variable, entre otras mejoras. En 2011 llegó C11, que agregó soporte para programación multihilo, operaciones atómicas y mejoras en Unicode. Más recientemente se publicaron C17 (una revisión menor de correcciones) y C23 (con mejoras modernas como el tipo `bool` de verdad, `constexpr` y atributos estandarizados).

Lo impresionante es que, a pesar de todas estas evoluciones, un programa escrito en C89 hace más de treinta años sigue compilando y funcionando hoy. Esa estabilidad es otra razón por la que la industria ama a C: el código que escribes hoy seguirá sirviendo dentro de décadas. Muy pocos lenguajes pueden presumir eso.

Una analogía para que te quede claro: imagina que los lenguajes de programación son herramientas de construcción. Python sería un taladro eléctrico moderno con batería: fácil de usar, hace el trabajo rápido, pero si se acaba la batería o necesitas algo muy específico, estás limitado. C sería un juego de desarmadores y llaves de precisión: requiere más habilidad, pero con él puedes desarmar y armar cualquier cosa, hasta el taladro mismo. Por eso los que fabrican taladros (los creadores de Python, de Java, de JavaScript) usan C para construir sus herramientas.

Dennis Ritchie falleció en 2011, pero su legado es inmenso: sin él no habría Unix, sin Unix no habría Linux ni macOS ni Android ni Internet como lo conocemos, y sin C la computación moderna sería irreconocible. Cuando escribas tu primer programa en C, estarás usando la herramienta que el forjo hace más de cincuenta años. No está nada mal para empezar.

Por qué aprender C hoy: ventajas reales en el mundo actual

Ya vimos en la introducción algunas razones para aprender C, pero vamos a profundizar porque esta es probablemente la pregunta que más te van a hacer tus amigos cuando les digas que estás aprendiendo C en pleno 2026: "y eso para que sirve, no es un lenguaje viejito?".

Primera ventaja: C te da control total sobre la máquina. En C no hay recolector de basura que limpie la memoria por ti, no hay verificaciones automáticas que te impidan cometer errores tontos. Tú decides cuando pedir memoria, cuando liberarla, como organizar tus datos. Esto suena aterrador al principio, pero es liberador cuando lo dominas: significa que tu programa hace exactamente lo que tu le dices, ni más ni menos, y por eso los programas en C son rapidísimos y usan poquísima memoria. En aplicaciones donde cada microsegundo y cada byte cuentan (un marcapasos, el sistema de frenos de un carro, el motor de un videojuego), ese control no es un lujo: es una necesidad.

Segunda ventaja: C es el lenguaje de los sistemas embebidos. Un sistema embebido es una computadora pequeña metida dentro de otro aparato: el microcontrolador de una lavadora, el sensor de un termostato inteligente, el control de un dron, la computadora de a bordo de un automovil. Estos aparatos no tienen gigabytes de memoria ni procesadores potentes; tienen recursos muy limitados y necesitan código eficiente. C fue diseñado exactamente para eso: genera programas pequeños, rápidos y predecibles. Si tu sueño es trabajar con hardware, robótica o Internet de las Cosas, C es tu boleto de entrada. No hay vuelta de hoja.

Tercera ventaja: C es la base de los sistemas operativos. El núcleo de Linux (que mueve servidores, supercomputadoras, teléfonos Android y millones de dispositivos) está escrito en C. El núcleo de Windows tiene componentes enormes en C. El núcleo de macOS e iOS también. Si quieres entender cómo funciona realmente un sistema operativo, o peor aun, si quieres trabajar creando uno, necesitas C. Es el idioma oficial del software de sistemas.

Cuarta ventaja: C es rapidísimo. Cuando se compila, tu código C se traduce directamente a código máquina optimizado, sin capas intermedias. Python, Java y JavaScript necesitan un intérprete o una máquina virtual que traduzca el código mientras se ejecuta, lo que los hace más lentos. Para la mayoría de las aplicaciones esa diferencia no importa, pero en videojuegos, en procesamiento de video, en cálculo científico y en sistemas de trading financiero, la velocidad de C es la diferencia entre ganar y perder.

Quinta ventaja: aprender C te hace mejor programador en cualquier otro lenguaje. Esto no es un cliché motivacional, es un hecho pedagógico. Cuando aprendes Python primero, la memoria es "magia": creas variables y nunca te preguntas donde viven ni cuánto pesan. Cuando aprendes C primero, entiendes que una variable es un espacio reservado en la memoria RAM, que tiene una dirección y un tamaño, y que si no la manejas bien tu programa falla. Ese entendimiento profundo te convierte en un programador que escribe mejor código en cualquier lenguaje, porque sabes lo que pasa debajo.

Sexta ventaja: el mercado laboral. Los programadores de C son menos que los de Python o JavaScript, pero la demanda en sectores como sistemas embebidos, automoción, aeroespacial, videojuegos, sistemas operativos y ciberseguridad es constante y los sueldos suelen ser altos, precisamente porque es una habilidad escasa y difícil de reemplazar. Además, C casi nunca se aprende "de moda": es una inversión a largo plazo.

Ahora, seamos honestos también con las desventajas, porque un buen libro no te vende humo. C no tiene manejo automático de memoria, así que tú eres responsable de no cometer errores con ella (en la Unidad 7 veremos esto a fondo). C no trae bibliotecas gigantes incluidas: si quieres una interfaz gráfica bonita o conectarte a una base de datos, tendrás que buscar bibliotecas externas o escribir más código tú mismo. Y los errores en C pueden ser

crudos: en lugar de un mensaje amable, a veces tu programa simplemente "truena" y tienes que investigar por que. Pero aquí esta el secreto: cada una de esas desventajas es también una lección. C no te trata como bebe; te trata como adulto responsable. Y eso, a la larga, te forma como un programador de verdad.

Piensalo así: aprender C es cómo aprender a manejar estándar pudiendo haber aprendido automático. Al principio sufres más, pero cuando terminas puedes manejar cualquier carro del mundo, entiendes cómo funciona la transmisión y hasta manejas mejor el automático porque sabes lo que está pasando por dentro. Quien solo aprendió automático, en cambio, se queda varado frente a un estándar.

Donde se usa C: del espacio a tu bolsillo

Vamos a hacer un recorrido rápido por donde vive C hoy, para que veas que no es un lenguaje de museo:

Sistemas operativos: Linux, Windows, macOS, Android, iOS. Todos tienen su núcleo escrito principalmente en C. Cada vez que enciendes tu computadora o tu teléfono, estás ejecutando millones de líneas de C.

Sistemas embebidos e Internet de las Cosas: microcontroladores como los de Arduino, ESP32, PIC y ARM se programan típicamente en C. Tu refrigerador inteligente, tu reloj, los sensores de una fábrica: C por todas partes.

Videojuegos: motores como Unreal Engine tienen su núcleo en C++ (el hijo de C), y muchísimos juegos, especialmente en consolas y móviles, usan C para las partes críticas de rendimiento. La industria del videojuego es una de las que mejor paga a programadores de C/C++.

Bases de datos y software de infraestructura: MySQL, PostgreSQL, SQLite, Redis, Nginx (el servidor web que mueve gran parte de Internet), Git: todos escritos en C. Cuando usas Internet, estás usando C sin saberlo.

Ciencia y espacio: la NASA usa C en muchísimos de sus sistemas, desde rovers en Marte hasta software de vuelo. El código que llevo al hombre a la Luna no era C (era ensamblador), pero el que opera la Estación Espacial Internacional y las misiones modernas si tiene muchísimo C.

Ciberseguridad: para entender vulnerabilidades, exploits y análisis de malware necesitas entender C, porque es el lenguaje en el que están escritos los sistemas que vas a proteger (o a poner a prueba). Los hackers éticos más respetados dominan C.

Interpretes y compiladores: Python (CPython), PHP, Ruby y muchos otros lenguajes están implementados en C. Incluso el compilador de C más usado, GCC, está escrito en C. Es un lenguaje que se construye a si mismo, como una serpiente que se muerde la cola de forma productiva.

Instalando tu compilador: GCC paso a paso en Windows, Linux y macOS

Antes de escribir tu primer programa necesitas dos herramientas: un editor de texto para escribir el código y un compilador para traducirlo a un programa ejecutable. Para el editor puedes usar el que quieras: el Bloc de notas de Windows, Gedit en Linux, o mejor aun, un editor pensado para programar como Visual Studio Code (gratis y multiplataforma), que te va a facilitar muchísimo la vida con colores para el código y detección de errores. Pero el compilador es obligatorio, y en este curso usaremos GCC, el GNU Compiler Collection, el compilador de C más usado y respetado del mundo. Es gratis, es de código abierto y está disponible para todos los sistemas operativos.

Qué es exactamente un compilador? Es un programa que lee tu código fuente (el texto que tú escribes en C) y lo traduce a código máquina, que es el lenguaje de unos y ceros que entiende tu procesador. El resultado es un archivo ejecutable: un programa que puedes correr con doble clic o desde la terminal. Analogía de cocina: tu código fuente es la receta escrita en español, el compilador es el cocinero que la interpreta, y el programa ejecutable es el platillo ya servido, listo para comerse. Sin cocinero no hay platillo, por más buena que este la receta.

Vamos a instalarlo según tu sistema operativo. Sigue solo la sección que te corresponda.

Instalación en Windows con MinGW-w64

En Windows, la forma más sencilla de tener GCC es mediante MinGW-w64 (Minimalist GNU for Windows, versión de 64 bits). Sigue estos pasos con calma:

Paso 1: Descarga el instalador. Abre tu navegador y busca "MinGW-w64 download" o ve al sitio oficial. La forma más fácil hoy en día es descargar la versión ya compilada desde la página de releases del proyecto (busca "mingw-w64 installer" o usa la distribución standalone de niXman, muy popular). Descarga la versión para 64 bits (x86_64) con el formato que prefieras; la más común viene como archivo .7z o como instalador .exe.

Paso 2: Instala o descomprime. Si descargaste el instalador .exe, ejecutalo y sigue el asistente: elige la arquitectura x86_64, los hilos (threads) posix o win32 (cualquiera funciona para este curso; elige posix si no sabes cuál), y la excepción seh. Si descargaste el .7z, descomprimelo con 7-Zip en una ruta sencilla sin espacios, por ejemplo C:\mingw64. Evita

rutas con espacios o acentos como "C:\Mis documentos", porque algunos programas de programación se confunden con los espacios.

Paso 3: Agrega el compilador al PATH. Este es el paso que más gente olvida y el que más dolores de cabeza causa, así que pon atención. El PATH es una lista de carpetas donde Windows busca programas cuando escribes un comando en la terminal. Si no agregas la carpeta bin de MinGW al PATH, Windows no encontrará el comando gcc. Hazlo así: abre el menú inicio, escribe "variables de entorno" y abre "Editar las variables de entorno del sistema". En la ventana que aparece, haz clic en "Variables de entorno". En la sección de "Variables del sistema" (o de usuario, cualquiera sirve), selecciona la variable Path y haz clic en "Editar". Haz clic en "Nuevo" y escribe la ruta completa a la carpeta bin, por ejemplo C:\mingw64\bin. Acepta todo con OK.

Paso 4: Verifica la instalación. Abre una terminal nueva (importante: si tenías una terminal abierta antes de modificar el PATH, ciérrala y abre otra, porque no se enteró del cambio). Presiona Win + R, escribe cmd y presiona Enter. Escribe:

```
gcc --version
```

Si ves algo como "gcc (x86_64-posix-seh-rev...) 13.2.0" o similar, felicidades: ya tienes compilador. Si ves "'gcc' no se reconoce como un comando", regresa al paso 3 y revisa que la ruta sea correcta y que hayas abierto una terminal nueva.

Alternativa moderna: si prefieres no pelear con el PATH, puedes instalar WSL (Windows Subsystem for Linux), que te da un Linux completo dentro de Windows, y ahí instalar GCC como se explica en la sección de Linux. También puedes usar Visual Studio Code con la extensión de C/C++ de Microsoft, que te ayuda a compilar con un clic. Pero te recomiendo aprender primero a compilar desde la terminal: entender el proceso te hará un programador más completo.

Instalación en Linux con apt (Ubuntu, Debian y derivados)

Si usas Linux, estás de suerte: instalar GCC es cuestión de un comando. Abre la terminal (Ctrl + Alt + T en la mayoría de las distribuciones) y ejecuta:

```
sudo apt update
sudo apt install gcc
```

El primer comando actualiza la lista de paquetes disponibles y el segundo instala el compilador. Te pedirá tu contraseña (es normal que no veas asteriscos mientras la escribes) y una confirmación. Después verifica con:

```
gcc --version
```

Deberías ver algo como "gcc (Ubuntu 13.2.0...) 13.2.0". Listo. En otras distribuciones el comando cambia un poco: en Fedora o CentOS usa `sudo dnf install gcc`, en Arch Linux usa `sudo pacman -S gcc`. Pero la idea es la misma.

Un consejo: en Linux también te conviene instalar las herramientas de desarrollo completas con `sudo apt install build-essential`, que incluye GCC, el programa make y otras utilidades que usaremos más adelante en el curso.

Instalación en macOS con las Command Line Tools de Xcode

En macOS, Apple no incluye GCC de fábrica, pero te ofrece las Command Line Tools de Xcode, que incluyen el compilador Clang. Clang no es GCC, pero es totalmente compatible con el estándar de C y para este curso funciona igual: todos los comandos que veas con gcc también funcionan si escribes gcc, porque macOS crea un alias. Para instalarlas, abre la Terminal (la encuentras en Aplicaciones > Utilidades) y ejecuta:

```
xcode-select --install
```

Aparecerá una ventana preguntándote si quieres instalar las herramientas; acepta y espera a que termine la descarga. Después verifica con:

```
gcc --version
```

Verás algo como "Apple clang versión 15.0.0". Aunque diga "clang", el comando gcc funciona y compila código C estándar sin problemas. Si algún día quieres el GCC "de verdad" en macOS, puedes instalarlo con Homebrew (`brew install gcc`), pero para este curso no lo necesitas.

Tu primer encuentro con la terminal

Vas a usar mucho la terminal en este curso, así que vamos a perderle el miedo desde ahora. La terminal (también llamada consola o línea de comandos) es un programa donde escribes órdenes con el teclado en lugar de hacer clic. Los programadores la amamos porque es rápida, precisa y funciona igual en todos los sistemas.

Solo necesitas tres comandos para empezar: ``cd`` para cambiar de carpeta (por ejemplo ``cd Documentos`` entra a la carpeta Documentos), ``dir`` en Windows o ``ls`` en Linux/macOS para ver que archivos hay en la carpeta actual, y el comando ``gcc`` para compilar. Con eso tienes para toda esta unidad. No necesitas ser un experto en la terminal; solo necesitas no tenerle miedo. Piensa en ella como en la palanca de velocidades del carro estándar: al principio la evitas, después no puedes vivir sin ella.

Tu primer programa: el ritual del "Hola, mundo"

Llegó el momento que estabas esperando: vas a escribir tu primer programa en C. Abre tu editor de texto favorito (el Bloc de notas sirve, Visual Studio Code es mejor) y escribe exactamente esto:

```
#include <stdio.h>
int main() {
    printf("Hola, mundo\n");
    return 0;
}
```

Guardalo con el nombre ``hola.c`` en una carpeta fácil de encontrar, por ejemplo en Documentos o en una carpeta llamada ``cursos-c`` que crees para este curso. El ``.c`` del final es importante: es la extensión que le dice al compilador "este archivo contiene código fuente en C".

Ahora abre la terminal y navega hasta la carpeta donde guardaste el archivo. Por ejemplo, si lo guardaste en Documentos/curso-c, escribe:

```
cd Documentos/curso-c
```

En Windows la ruta podría ser ``cd C:\Users\TuNombre\Documents\curso-c``. Usa el comando ``dir`` (Windows) o ``ls`` (Linux/macOS) para comprobar que el archivo ``hola.c`` esta ahí. Si no lo ves, estás en la carpeta equivocada: navega hasta encontrarlo.

Ahora compila el programa con este comando mágico:

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

Vamos a desmenuzar lo que acabas de escribir, porque cada parte importa: ``gcc`` es el compilador; ``hola.c`` es tu archivo de código fuente, la "receta"; ``-o hola`` le dice al compilador como quieres que se llame el programa resultante (la ``-o`` viene de "output", salida). Si todo

sale bien, el compilador no dice nada y regresa a la terminal en silencio. En el mundo de la compilación, el silencio es una buena noticia: significa que no hubo errores. Ahora lista los archivos de la carpeta y verás que apareció un archivo nuevo: `hola.exe` en Windows o `hola` en Linux/macOS. Ese es tu programa, ya traducido a código máquina y listo para ejecutarse.

Para correrlo, en Windows escribe:

```
hola
```

En Linux o macOS escribe:

```
./hola
```

(El `./` le dice al sistema "busca el programa aquí mismo, en la carpeta actual".) Y entonces verás aparecer en tu pantalla, como por arte de magia:

```
Hola, mundo
```

Felicidades. Acabas de hacer lo mismo que hizo Brian Kernighan en 1978 y lo mismo que han hecho millones de programadores desde entonces. Puede parecer poca cosa, dos palabras en la pantalla, pero piénsalo: tu escribiste una receta en un lenguaje humano-legible, una herramienta la tradujo al idioma de la máquina, y la máquina la ejecutó. Ese ciclo (escribir, compilar, ejecutar) es el corazón de la programación, y lo vas a repetir miles de veces en tu carrera. Cada programa gigante empezó siendo un "hola, mundo" que alguien se atrevió a compilar.

Que pasa cuando compilas: el viaje de tu código

El comando `gcc hola.c -o hola` parece una sola acción, pero por dentro el compilador hace cuatro etapas. No necesitas memorizarlas ahora, pero entenderlas te va a ahorrar muchos dolores de cabeza cuando algo falle:

Etape 1, preprocesamiento: el preprocesador lee tu archivo y procesa las líneas que empiezan con `#`, como `#include <stdio.h>`. Lo que hace es copiar y pegar el contenido de ese archivo de cabecera dentro de tu programa. Es como cuando en una receta dice "usa la salsa base de la página 5": el preprocesador va a la página 5, copia la receta de la salsa y la pega en tu receta. Puedes ver el resultado con `gcc -E hola.c`.

Etape 2, compilación: el compilador traduce tu código C (ya con las cabeceras incluidas) a lenguaje ensamblador, que es un lenguaje de muy bajo nivel pero todavía legible por humanos

expertos. Aquí es donde se detectan la mayoría de los errores de sintaxis: si olvidaste un punto y coma o escribiste mal una palabra, el compilador se queja aquí.

Etapas 3, ensamblado: el ensamblador traduce el código ensamblador a código objeto, que son unos y ceros organizados, pero todavía no es un programa completo: le faltan piezas, como el código de la función ``printf`` que vive en otro lado.

Etapas 4, enlazado: el enlazador (linker) junta tu código objeto con las bibliotecas necesarias (como la biblioteca estándar donde vive ``printf``) y produce el ejecutable final. Si usas una función que no existe en ningún lado, el error aparece aquí, en el enlazado.

Analogía completa: eres un arquitecto (tu), escribes los planos de una casa (código fuente), el preprocesador consigue los planos de los componentes estándar (puertas, ventanas), el compilador traduce tus planos al lenguaje de los albaniles, el ensamblador corta los ladrillos a la medida, y el enlazador junta tu casa con los servicios de la ciudad (agua, luz) para que sea habitable (ejecutable). Cada etapa depende de la anterior, y si algo falla, el mensaje de error te dice en que etapa fue.

El comando gcc que debes dominar

Vamos a practicar las variantes del comando de compilación que usarás todo el curso. La forma básica ya la conoces:

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

Si omites el ``-o``, el compilador igual genera un ejecutable, pero con un nombre genérico: ``a.exe`` en Windows o ``a.out`` en Linux/macOS (nombres heredados de los años setenta). Funciona, pero es mala práctica porque si compilas varios programas terminas con un montón de archivos ``a.out`` y no sabes cuál es cuál. Acostumbrate a usar ``-o`` siempre y a ponerle un nombre descriptivo a tu programa.

Para ver advertencias (warnings) que te ayudan a escribir mejor código, agrega ``-Wall``:

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

``-Wall`` significa "muestrame todas las advertencias" (Wall = Warn all). No es obligatorio, pero es una excelente costumbre: el compilador se convierte en tu maestro particular y te avisa de prácticas sospechosas antes de que se vuelvan errores. Con esto ya tienes lo esencial para leer los warnings desde hoy.

Si tu programa usa funciones matemáticas como ``sqrt`` o ``sin`` (de la biblioteca `math.h`), en Linux necesitas enlazar la biblioteca matematica con ``-lm`` al final:

```
gcc programa.c -o programa -lm
```

En Windows y macOS normalmente no hace falta, pero no te asustes si algun día lo ves en un tutorial: solo significa "incluye la biblioteca de matemáticas".

Anatomia de un programa en C: desarmando el "Hola, mundo"

Ya lograste que tu programa funcione, pero un programador de verdad no se conforma con que funcione: quiere entender POR QUE funciona. Vamos a desarmar el programa línea por línea, como un mecanico desarma un motor para entender cada pieza.

```
#include <stdio.h>
int main() {
    printf("Hola, mundo\n");
    return 0;
}
```

Línea 1: `#include <stdio.h>`

Esta línea es una orden para el preprocesador (recuerda la etapa 1 de la compilación). Le dice: "antes de compilar, traeme el contenido del archivo `stdio.h`". Las letras `stdio` significan "standard input-output", entrada y salida estándar. Ese archivo es como un catalogo donde están declaradas funciones basicas como ``printf`` (para imprimir en pantalla) y ``scanf`` (para leer del teclado, que veremos en la Unidad 2).

Sin esta línea, el compilador no sabria que existe la función ``printf`` y te marcaria un error o una advertencia. Es como querer usar la licuadora sin haberla sacado de la caja: la herramienta existe, pero tienes que "incluirla" en tu cocina primero.

Los símbolos ``<`` y ``>`` alrededor de ``stdio.h`` le dicen al compilador "busca este archivo en las carpetas del sistema, donde viven las cabeceras estándar". Más adelante en el curso verás que también puedes incluir tus propios archivos usando comillas dobles, como ``#include "misfunciones.h"``, que significa "busca este archivo aquí mismo, en mi carpeta".

El ``#`` al inicio no es un adorno: indica que es una directiva del preprocesador, una instrucción que se procesa antes de la compilación real. Nota algo curioso: las directivas del preprocesador NO llevan punto y coma al final. Ya veremos por que el punto y coma es tan importante en las demás líneas.

Línea 2: `int main() {`

Esta es la línea más importante de todo programa en C. Vamos por partes:

``int`` significa "integer", número entero. Indica que esta función, cuando termine, va a devolver un número entero al sistema operativo. Es como el "tipo de dato" del resultado.

``main`` es el nombre de la función. Y no es un nombre cualquiera: es EL nombre. Cuando ejecutas un programa en C, el sistema operativo busca la función llamada ``main`` y empieza a ejecutar el código desde ahí. Todo programa en C debe tener exactamente una función ``main``; sin ella, el enlazador se queja porque no sabe por donde empezar. Es la puerta de entrada de tu programa, como la puerta principal de una casa: por más cuartos que tenga, se entra por la puerta principal.

Los paréntesis ``()`` vacíos indican que esta función no recibe datos de entrada (en programas más avanzados, ``main`` puede recibir argumentos desde la terminal, pero eso lo veremos mucho más adelante).

La llave de apertura ``{`` marca el inicio del cuerpo de la función, es decir, donde empieza el código que pertenece a ``main``. Las llaves en C son como los signos de agrupación en matemáticas: todo lo que esta entre la llave que abre y la que cierra pertenece al mismo bloque.

Línea 3: `printf("Hola, mundo\n");`

Esta es la línea que hace el trabajo visible: imprime texto en la pantalla. Analicemos sus piezas:

``printf`` es una función de la biblioteca estándar cuyo nombre viene de "print formatted", impresión con formato. Le pasas un texto entre comillas dobles y ella lo muestra en la pantalla. En la Unidad 2 la exprimiremos a fondo con formatos para números, pero por ahora quedate con que imprime lo que le pongas entre comillas.

``"Hola, mundo\n"`` es el texto a imprimir, llamado "cadena de texto" o "string". Fíjate en el ``\n`` del final: no es una "n" normal, es una secuencia de escape que representa un salto de línea. Cuando ``printf`` la encuentra, no imprime una "n": baja el cursor al siguiente renglon. Sin el ``\n``,

el texto se quedaría en el mismo renglón y la siguiente cosa que se imprima (o el prompt de tu terminal) saldría pegada a tu mensaje. Pruebalo: quita el `\n`, compila y ejecuta, y verás la diferencia.

El punto y coma `;` del final es OBLIGATORIO en C. Marca el fin de una instrucción, como el punto marca el fin de una oración en español. Si lo olvidas, el compilador se confunde y te marca un error, a veces en una línea que no es la del problema (porque el compilador sigue leyendo buscando el punto y coma). Acostumbrate: cada instrucción en C termina con punto y coma. Es la regla número uno de la sintaxis.

Línea 4: `return 0;`

Esta línea le dice a la función `main` que termine y devuelva el valor 0 al sistema operativo. Por convención universal, devolver 0 significa "todo salió bien, el programa terminó sin errores". Si tu programa detecta un error grave, puede devolver otro número (como 1) para avisarle al sistema que algo falló. Es como el semáforo de salida: verde (0) significa "todo bien", cualquier otro número es una luz roja con un código que indica que pasó.

Te preguntarás: si `main` dice `int` al principio (promete devolver un entero), tiene que cumplir su promesa con un `return` que devuelva un entero. Así es: el `int` del inicio y el `return 0` del final son pareja, se necesitan mutuamente. En estándares modernos de C, si omites el `return 0`, el compilador asume que devuelves 0, pero escribirlo explícito es buena práctica y deja claro tu propósito.

Línea 5: `}`

La llave de cierre marca el fin del cuerpo de la función `main` y, por lo tanto, el fin de tu programa. Todo lo que este entre las llaves de `main` es lo que se ejecuta; lo que este fuera, no. Las llaves siempre van en pareja: cada llave que abre debe tener su llave que cierra. Si te falta una, el compilador te lo hará saber con un error.

El formato no afecta, pero si importa para los humanos

Fíjate que el código está indentado (con sangría): la línea del `printf` y la del `return` están un poco a la derecha. Esa sangría NO es obligatoria para el compilador: C ignora los espacios en blanco extra, los saltos de línea y la indentación. Podrías escribir todo el programa en una sola línea y compilaría igual. Pero para los humanos que leen tu código (incluyendote a ti mismo dentro de dos semanas), la indentación es oro: muestra visualmente que esas líneas pertenecen al interior de `main`. Acostúmbrate a indentar con 4 espacios (o una tabulación) cada nivel de llaves. Es una de las buenas prácticas más simples y más poderosas que existen.

Los comentarios también son para humanos: el compilador los ignora por completo. En C puedes escribir comentarios de una línea con `//` y comentarios de varias líneas entre `/*` y `*/`. Usalos para explicar el porqué de tu código, no el QUE (el que ya se ve en el código). Ejemplo:

```
#include <stdio.h>
int main() {
    // Imprimo un saludo en la pantalla
    printf("Hola, mundo\n");
    return 0; // 0 significa que todo salio bien
}
```

A partir de la Unidad 2 usaremos comentarios en todos los ejemplos para que aprendas a documentar tu código como un profesional.

Errores comunes de principiantes (y como evitarlos)

Todos los programadores, sin excepción, cometieron estos errores cuando empezaron. Conocerlos de antemano no significa que no los vayas a cometer (los vas a cometer, y está bien), pero si significa que cuando aparezcan los vas a reconocer en segundos en lugar de sufrirlos durante horas. Vamos con la lista de los clásicos:

Error 1: olvidar guardar el archivo antes de compilar. Escribes tu código, corres `gcc hola.c -o hola`, y el compilador te marca errores en líneas que tu ya corregiste... o compila una versión vieja y tu programa no hace lo nuevo que le pusiste. La causa: no guardaste (Ctrl + S) después de editar. Parece obvio, pero le pasa a todo el mundo, especialmente cuando estás concentrado. Haz del Ctrl + S un reflejo: edita, guarda, compila. Siempre en ese orden.

Error 2: compilar sin el `-o` y no encontrar tu programa. Escribes `gcc hola.c`, compila sin errores, pero no ves ningún `hola.exe` por ningún lado. Lo que pasó es que el compilador creó `a.exe` (Windows) o `a.out` (Linux/macOS). Tu programa si existe, solo que se llama distinto. Solución: usa siempre `-o` con un nombre claro.

Error 3: olvidar el punto y coma. El error más famoso de C. Escribes `printf("Hola")` sin el `;` y el compilador te suelta un mensaje como "error: expected ';' before 'return'". Lo tramposo es que a veces el error lo marca en la línea SIGUIENTE, porque el compilador siguió leyendo buscando el punto y coma. Regla de oro: cuando veas un error de sintaxis, revisa también la línea anterior a la que marca el compilador.

Error 4: confundir `#include` con instrucción normal y ponerle punto y coma. Si escribes `#include <stdio.h>;` con punto y coma al final, algunos compiladores se quejan y otros lo ignoran, pero es incorrecto: las directivas del preprocesador no llevan punto y coma. Recuerda: `#` al inicio significa "esto lo maneja el preprocesador, no lleva punto y coma".

Error 5: no cerrar las llaves o los paréntesis. Cada `{` necesita su `}`, cada `(` necesita su `)`. Cuando tu programa crezca, es fácil perder la cuenta. El truco profesional: escribe primero las dos llaves `{}` y después escribe el código en medio. Así nunca te falta una. Los editores como Visual Studio Code también te ayudan resaltando la pareja de cada llave.

Error 6: usar comillas simples en lugar de dobles para el texto. En C, `"Hola"` (comillas dobles) es una cadena de texto, pero `'H'` (comillas simples) es un solo carácter, y son cosas distintas. Si escribes `printf('Hola, mundo');` el compilador se quejará. Para texto siempre usa comillas dobles; las simples las veremos en la Unidad 2 con caracteres individuales.

Error 7: el programa "no hace nada" porque olvidaste el `\n` o porque la ventana se cierra. En Windows, si ejecutas tu programa con doble clic en lugar de desde la terminal, la ventana se abre, imprime el texto y se cierra tan rápido que no alcanzas a verlo. Eso no es un error de tu código: es que el programa terminó y Windows cerró la ventana. Solución: ejecútalo siempre desde la terminal, donde la ventana no se cierra sola.

Error 8: compilar en la carpeta equivocada. Escribes `gcc hola.c -o hola` y el compilador dice "hola.c: No such file or directory". No es que tu archivo no exista: es que no estás parado en la carpeta donde lo guardaste. Usa `dir` o `ls` para ver dónde estás y `cd` para moverte a la carpeta correcta.

Error 9: no leer los mensajes de error. Los principiantes ven un mensaje de error largo en inglés y entran en pánico. Pero esos mensajes son tus amigos: casi siempre te dicen el archivo, el número de línea y que esperaban encontrar. Aprende a leerlos con calma: busca el primer error de la lista (los demás suelen ser consecuencia del primero), ve a esa línea, y corrígelo. El 90 por ciento de las veces es un punto y coma, una llave o una palabra mal escrita.

Error 10: cambiar muchas cosas a la vez cuando algo falla. Tu programa no compila, y en lugar de arreglar un error a la vez, reescribes medio programa "a ver si así funciona".

Resultado: ahora tienes tres errores nuevos y no sabes cuál era el original. La técnica correcta es científica: haz UN cambio, compila, observa. Si el error sigue, revierte el cambio y prueba otra cosa. Así siempre sabes que causo que.

Buenas prácticas desde el día uno

Un buen hábito aprendido al principio vale por cien correcciones después. Estas son las prácticas que te recomiendo adoptar desde hoy, cuando tus programas todavía son pequeños y es fácil hacerlo bien:

Práctica 1: un archivo, un programa, un propósito. Guarda cada programa en su propio archivo con un nombre descriptivo: ``hola.c``, ``saludo.c``, ``mi_edad.c``. Nada de ``programa1.c``, ``prueba_final2.c``, ``este_si_funciona.c``. Tu yo del futuro te lo agradecerá.

Práctica 2: compila con ``-Wall`` desde el principio. Ya vimos que ``-Wall`` activa las advertencias del compilador. Un programa puede compilar sin errores y aún así tener advertencias que indican problemas futuros. Trata las advertencias como errores: no des por terminado un programa hasta que compile limpio, sin warnings.

Práctica 3: indenta tu código siempre. Ya lo mencionamos en la anatomía: 4 espacios por nivel. El código bien indentado se lee en segundos; el código sin indentar es un muro de texto que nadie quiere tocar, ni tú mismo.

Práctica 4: comenta el porqué, no el que. Un comentario como ``// imprime hola`` al lado de un ``printf("Hola")`` no aporta nada: el código ya dice eso. Un buen comentario explica decisiones: ``// uso \n para que el prompt no salga pegado al mensaje``. Cuando tus programas crezcan, estos comentarios serán tu memoria externa.

Práctica 5: prueba tu programa después de cada cambio pequeño. No escribas 50 líneas de un jalón y compiles al final: escribe 5 líneas, compila, ejecuta, verifica. Si algo falla, sabes que el problema está en esas 5 líneas. Este hábito, que parece lento, es en realidad lo más rápido que existe, porque evita sesiones maratónicas de depuración.

Práctica 6: guarda versiones de tu trabajo. Cuando un programa ya funciona y vas a agregarle algo nuevo, guarda una copia (por ejemplo ``saludo_v1.c``) antes de modificarlo. Así, si lo rompes, puedes regresar a la versión que funcionaba. Más adelante aprenderás a usar Git para esto de forma profesional, pero la idea empieza aquí.

Práctica 7: lee código de otros. En lucasegaacademy.com y en miles de sitios hay ejemplos de código C. Leer código ajeno es cómo leer libros para un escritor: te ensena estilos, trucos y formas de resolver problemas que no se te habrían ocurrido. Cuando veas un programa que no entiendas, copialo, compilalo, modificalo y observa que pasa. Esa curiosidad es el motor del aprendizaje.

Práctica 8: no copies y pegues código que no entiendes. Es tentador, sobre todo cuando tienes prisa, pero el código copiado sin entender es una bomba de tiempo: funciona hoy y explota mañana, y cuando explote no sabrás por que. Si vas a usar código de otro lado, tomalo como punto de partida y asegurate de entender cada línea antes de darlo por tuyo.

Ejemplos paso a paso

La teoria sin práctica es como una receta sin cocina: suena bien pero no alimenta. Aquí tienes una serie de ejemplos completos que debes escribir, compilar y ejecutar tú mismo. Cada ejemplo incluye una explicación de lo que vas a ver y una sección de "que observar" para que pongas atención en los detalles importantes. No te saltes ninguno: cada uno construye sobre el anterior.

Ejemplo 1: El clásico "Hola, mundo"

Este es el programa con el que empezamos la unidad. Escríbelo exactamente como está, compilalo con ``gcc hola.c -o hola`` y ejecutalo.

```
#include <stdio.h>
int main() {
    printf("Hola, mundo\n");
    return 0;
}
```

Que observar: primero, confirma que el programa imprime "Hola, mundo" y baja a la siguiente línea gracias al ``\\n``. Segundo, haz la prueba del silencio: compila y nota que el compilador no dice nada cuando todo está bien; el silencio significa éxito. Tercero, experimenta rompiendo cosas a propósito: quita el punto y coma del ``printf`` y compila para ver el mensaje de error; luego quita el ``\\n`` y observa como el prompt de la terminal sale pegado al mensaje; después cambia el texto entre comillas por tu nombre y vuelve a compilar. Cada experimento te ensena algo. Este programa es tu laboratorio: tan simple que cualquier cambio que hagas tiene un efecto visible e inmediato.

Ejemplo 2: Anatomía comentada línea por línea

Ahora vamos a escribir el mismo programa pero con comentarios que explican cada parte. Los comentarios son ignorados por el compilador, así que el programa funciona igual, pero ahora el código se explica solo. Guardalo como `anatomia.c`.

```
#include <stdio.h>    // Biblioteca de entrada y salida estandar

int main() {          // La funcion principal: aqui empieza todo
    // Esta linea imprime el saludo en la pantalla
    printf("Hola, mundo\n"); // \n es un salto de linea
    return 0;         // 0 le dice al sistema que todo salio bien
}                     // Aqui termina la funcion main
```

Que observar: nota como hay dos estilos de comentario en juego (aunque aquí solo usamos `//`): los comentarios pueden ir en su propia línea o al lado del código. Observa que el programa compila y funciona exactamente igual que el Ejemplo 1: los comentarios no afectan al programa, solo a los humanos que lo leen. Haz la prueba de borrar todos los comentarios, compilar de nuevo y comparar: mismo resultado. Esta es la prueba de que los comentarios son para ti, no para la máquina. A partir de ahora, acostúmbrate a comentar tus programas: tu yo del futuro, que tendrá que leer este código dentro de un mes, te lo va a agradecer.

Ejemplo 3: Imprimir varias líneas

Un solo `printf` puede imprimir varias líneas si usas varios `\n`, y también puedes usar varios `printf` seguidos. Este ejemplo muestra ambas técnicas. Guardalo como `varias\_lineas.c`.

```
#include <stdio.h>

int main() {
    // Un solo printf con varios saltos de linea
    printf("Linea uno\nLinea dos\nLinea tres\n");

    // Varios printf, uno por linea
    printf("Esta es otra linea\n");
    printf("Y esta es la ultima\n");

    return 0;
}
```

Que observar: fíjate en que el primer `printf` imprime tres líneas con una sola llamada, porque cada `\n` baja el cursor. Los `printf` no agregan saltos de línea automáticamente: si olvidas el `\n`, el siguiente texto sale pegado. Prueba quitar uno de los `\n` y observa como dos líneas se

fusionan en una. También nota que puedes mezclar ambas técnicas libremente: C no te impone una forma, solo te da las herramientas. Preguntate: cuando conviene un solo `printf` y cuando convienen varios? Si el texto es un bloque fijo, un solo `printf` es más compacto; si cada línea se genera en una parte distinta de tu lógica, varios `printf` son más claros. Esa decisión de estilo es parte de convertirte en programador.

Ejemplo 4: Dibujar figuras con printf

Aquí viene lo divertido: puedes "dibujar" usando solo texto. Este programa imprime un triángulo y una carita usando caracteres. Guardalo como `dibujo.c`.

```
#include <stdio.h>
int main() {
    printf("Dibujando un triangulo:\n");
    printf("    *\n");
    printf("   ***\n");
    printf("  *****\n");
    printf(" *****\n");
    printf("*****\n");

    printf("\nDibujando una carita:\n");
    printf("   ***** \n");
    printf("  *          * \n");
    printf(" *    0 0    * \n");
    printf(" *      ^    * \n");
    printf(" *  \\____/  * \n");
    printf(" *          * \n");
    printf("   ***** \n");

    return 0;
}
```

Que observar: lo primero es que el ``printf("\n");`` del medio imprime una línea vacía para separar las dos figuras: un ``\n`` solito equivale a un renglon en blanco. Lo segundo, y más importante, es el detalle de la línea ``printf("* \_\_\_ / *\n");``: fíjate en el doble backslash ``\\``. La diagonal invertida ``\`` es el carácter que inicia las secuencias de escape (como ``\n``), así que si quieres imprimir una diagonal invertida de verdad, tienes que "escaparla" escribiéndola dos veces: ``\\`` le dice a ``printf`` "imprime una sola diagonal invertida". Si escribieras un solo ``\``, el compilador intentaría interpretarlo como una secuencia de escape y te daría un error o un resultado raro. Este es un detalle que atormenta a los principiantes, así que grabatelo: para imprimir ``\``, escribe ``\\``. Prueba modificar el dibujo: haz un cuadrado, una flecha o tus iniciales. Jugar con estos dibujos es una excelente forma de practicar sin darte cuenta.

Ejemplo 5: Secuencias de escape en acción

Las secuencias de escape son combinaciones de caracteres que representan algo especial. Ya conoces ``\n`` (salto de línea) y ``\\`` (diagonal invertida). Este ejemplo te muestra las más útiles. Guardalo como ``escapes.c``.

```
#include <stdio.h>
int main() {
    printf("Salto de linea:\nprimera\nsegunda\n");
    printf("Tabulador:\ncolumna1\tcolumna2\tcolumna3\n");
    printf("Comillas dentro del texto: \"Hola\", dijo el programador\n");
    printf("Diagonal invertida: C:\\cursos\\c\\c\\c\n");
    printf("Porcentaje: aprobe con 100%% de calificacion\n");
    return 0;
}
```

Que observar: el ``\t`` es un tabulador: alinea el texto en columnas, útil para tablas sencillas. Las comillas dobles dentro de un texto se escriben como ``\"``: si pusieras comillas normales, el compilador pensaría que ahí termina la cadena. El símbolo de porcentaje ``%%`` se escribe como ``%%`` porque, como verás en la Unidad 2, el ``%`` tiene un significado especial en ``printf`` (marca donde van los valores). Cada secuencia de escape existe por la misma razón: permitirte escribir caracteres que de otro modo serían imposibles o ambiguos. Haz la tabla mental: ``\n`` nueva línea, ``\t`` tabulador, ``\\`` diagonal, ``\"`` comilla, ``%%`` porcentaje. Con esas cinco tienes el 99 por ciento de los casos cubiertos.

Ejemplo 6: Tu tarjeta de presentación

Vamos a juntar todo lo aprendido en un programa útil: una tarjeta de presentación en texto. Personalízala con tus datos reales. Guardala como ``presentacion.c``.


```
#include <stdio.h>
int main() {
    printf("=====\n");
    printf("  TARJETA DE PRESENTACION\n");
    printf("=====\n");
    printf("Nombre: Tu Nombre Aqui\n");
    printf("Curso: Programacion en C\n");
    printf("Escuela: lucasegaacademy.com\n");
    printf("Meta: Dominar C en 30 clases\n");
    printf("=====\n");
    return 0;
}
```

Que observar: este ejemplo no usa nada nuevo, y esa es la lección: con solo `printf` y `\n` ya puedes construir programas que se ven profesionales y organizados. Fíjate en como las líneas de `====` crean un marco visual que hace que la información se vea ordenada. El diseño de la salida es parte del programa: un programa que imprime información desordenada se siente barato, aunque el cálculo por dentro sea correcto. Cambia los datos por los tuyos, agrega más líneas (tu ciudad, tu lenguaje favorito, tu meta del año), y juega con el marco hasta que te guste cómo se ve. Este será tu primer programa "personal": guardalo, porque dentro de unas unidades, cuando sepas usar variables, lo vas a mejorar y te vas a sorprender de cuánto avanzaste.

Ejemplo 7: Errores a propósito (laboratorio de depuración)

Este ejemplo es diferente: es un programa con errores intencionales. Tu trabajo es encontrarlos y corregirlos. Guardalo como `roto.c` tal como está (con errores) e intenta compilarlo.

```
#include <stdio.h>
int main() {
    printf("Este programa tiene errores\n")
    printf("Encuentralos y corrígelos\n");
    return 0;
}
```

Que observar: al compilar, el compilador te marcara un error. Léelo con calma: te dirá algo como "expected ';' before 'printf'" y te dará un número de línea. Nota que el error está en la primera línea del `printf` (falta el punto y coma), pero el compilador lo reporta en la línea del segundo `printf`: esto confirma lo que aprendiste en la sección de errores comunes. Corrígelo agregando el `;` y compila de nuevo. Ahora haz más experimentos: quita una llave, cambia

``printf`` por ``Printf`` (C distingue mayúsculas de minúsculas: ``Printf`` no existe), ponle punto y coma al ``#include``. Cada error que provoques y corrijas ahora es un error que reconocerás al instante en el futuro. Los mejores depuradores no son los que nunca se equivocan: son los que ya vieron todos los errores antes.

Práctica

Es hora de ensuciarte las manos. Los siguientes ejercicios están ordenados de más fácil a más retador. Hazlos todos: no pases al siguiente sin haber compilado y ejecutado el anterior. Si te atorras más de 20 minutos en uno, mira la solución, entiendela, y después haz el ejercicio de nuevo sin mirar. Recuerda la regla de oro: escribe el código tú mismo, no lo copies y pegues.

Ejercicio 1: Escribe un programa que imprima tu nombre completo en una línea y tu edad en la siguiente. Guardalo como ``ejercicio1.c``, compilalo y ejecutalo. Parece trivial, y lo es: el objetivo es que practiques el ciclo completo (escribir, guardar, compilar, ejecutar) hasta que te salga de forma natural.

Ejercicio 2: Escribe un programa que imprima las siguientes tres líneas exactamente, incluyendo los espacios: en la primera línea tu ciudad, en la segunda tu comida favorita, en la tercera tu pasatiempo favorito. Cada dato debe imprimirse con su propia llamada a ``printf``. Guardalo como ``ejercicio2.c``.

Ejercicio 3: Dibuja con ``printf`` una casa sencilla: un techo triangular, una pared rectangular, una puerta y una ventana. No hay una única respuesta correcta: lo importante es que uses solo caracteres de texto y que la figura se reconozca. Guardalo como ``ejercicio3.c``. Si te sale bien, intenta dibujar después un árbol o un carro.

Ejercicio 4: Escribe un programa que imprima un "recibo" de una tienda ficticia con este formato: una línea de igualdades ``====`` como encabezado, el nombre de la tienda, una línea separadora de guiones ``----``, tres productos inventados cada uno en su línea, otra línea separadora, y al final la palabra "GRACIAS POR SU COMPRA". Usa secuencias de escape donde las necesites. Guardalo como ``ejercicio4.c``.

Ejercicio 5: Provoca y corrige tres errores distintos. Toma el programa del Ejercicio 1 y crea tres copias: en la primera quita un punto y coma, en la segunda escribe ``Main`` con mayúscula en lugar de ``main``, en la tercera quita la línea ``#include <stdio.h>``. Compila cada una, anota el mensaje de error exacto que te da el compilador, corrígela y verifica que compile. Guardalas como ``error1.c``, ``error2.c`` y ``error3.c`` ya corregidas. El objetivo no es el programa, sino entrenar tu ojo para leer mensajes de error.

Ejercicio 6 (reto): Escribe un programa que imprima un calendario semanal sencillo usando tabuladores `\t`: los días de la semana como encabezado (LUN MAR MIE JUE VIE SAB DOM) y debajo dos filas de actividades inventadas, alineadas en columnas. Guardalo como ``ejercicio6.c``. Si las columnas no se alinean bien, ajusta los tabuladores hasta que se vean ordenadas.

Ejercicio 7 (reto): Escribe un programa que imprima tu "contrato de programador": un texto de al menos 8 líneas donde te comprometes contigo mismo a terminar este curso (por ejemplo: "Yo, [tu nombre], me comprometo a programar todos los días..."). Decora el texto con un marco de asteriscos ``*`` alrededor, como en el Ejemplo 6. Guardalo como ``ejercicio7.c`` e imprimelo en papel si puedes: pegalo donde veas tu computadora todos los días.

Solución del ejercicio 1

```
#include <stdio.h>
int main() {
    printf("Esteban\n");
    printf("30 anos\n");
    return 0;
}
```

Por qué funciona: cada ``printf`` imprime una línea gracias al ``\n`` del final. Usamos dos llamadas separadas porque cada dato va en su propia línea, y eso hace el código más legible que meter todo en un solo ``printf``. Cambia el nombre y la edad por los tuyos: el programa debe compilar con ``gcc ejercicio1.c -o ejercicio1`` sin advertencias y al ejecutarlo debe mostrar tus datos en dos líneas.

Solución del ejercicio 2

```
#include <stdio.h>
int main() {
    printf("Ciudad de Mexico\n");
    printf("Tacos al pastor\n");
    printf("Jugar videojuegos\n");
    return 0;
}
```

Por qué funciona: el ejercicio pedía explícitamente una llamada a ``printf`` por línea, y eso es lo que hacemos. Nota que no hay nada "inteligente" aquí: la lección es que en programación muchas tareas son así de directas, y no hay que buscar complicaciones donde no las hay. Un

principiante suele pensar "seguro hay un truco"; no lo hay. Tres líneas, tres `printf`, listo. Personaliza los datos con tu ciudad, tu comida y tu pasatiempo.

Solución del ejercicio 3

```
#include <stdio.h>
int main() {
    // Techo triangular
    printf("  /\n\n");
    printf(" /  \n\n");
    printf("/   \n\n");
    printf("/____\n\n");
    // Paredes de la casa
    printf("|      |\n");
    printf("|  []  |\n");
    printf("|    _ |\n");
    printf("|    | | |\n");
    printf("|____|_| |\n");
    return 0;
}
```

Por qué funciona: cada línea del dibujo es un `printf` con los caracteres exactos que forman esa fila de la figura. Fíjate en los detalles técnicos: para dibujar las diagonales del techo usamos `\\` (doble diagonal invertida) porque una sola `\\` sería interpretada como inicio de secuencia de escape. Los comentarios separan las partes de la casa (techo y paredes), lo que hace el código más fácil de modificar: si quieres una casa más alta, agregas más líneas de pared entre los comentarios. Tu dibujo puede ser distinto al mío y estar perfectamente bien: lo que importa es que compiles, ejecutes y veas una casa reconocible. Si tu casa se ve "chueca", cuenta los espacios con cuidado: en el arte ASCII cada espacio cuenta.

Solución del ejercicio 4

```
#include <stdio.h>
int main() {
    printf("=====\n");
    printf("    TIENDA EL BUEN PRECIO\n");
    printf("=====\n");
    printf("-----\n");
    printf("1 kg de tortillas\n");
    printf("1 litro de leche\n");
    printf("1 barra de pan\n");
    printf("-----\n");
    printf("GRACIAS POR SU COMPRA\n");
    printf("=====\n");
    return 0;
}
```

Por qué funciona: el programa no hace cálculos todavía (eso llegará en la Unidad 2 con variables), pero demuestra algo importante: con puras impresiones de texto ya puedes simular la salida de un sistema real, como el ticket de una tienda. Las líneas de `=` y `-` actúan como separadores visuales que organizan la información, igual que en el Ejemplo 6 de la tarjeta de presentación. Observa el patrón: encabezado, separador, contenido, separador, despedida. Ese patrón de "marco + contenido" lo verás una y otra vez en programas reales. Inventa tu propia tienda con tus propios productos: personalizar los ejercicios hace que el aprendizaje se fije mejor.

Solución del ejercicio 5

No hay un único código solución porque el ejercicio consiste en experimentar, pero aquí están los tres errores y lo que debes observar en cada uno:

Error provocado 1 (falta el punto y coma):

```
#include <stdio.h>
int main() {
    printf("Esteban\n")
    printf("30 años\n");
    return 0;
}
```

El compilador dirá algo como "error: expected ';' before 'printf'" y señalará la línea del segundo `printf`. La lección: el error se reporta donde el compilador se dio cuenta del problema, no necesariamente donde está el problema. Revisa siempre la línea anterior.

Error provocado 2 (`Main` con mayúscula):

```
#include <stdio.h>
int Main() {
    printf("Esteban\n");
    printf("30 anos\n");
    return 0;
}
```

El compilador o el enlazador se quejará de que no encuentra la función `main` (algo como "undefined reference to `main`"). La lección: C distingue entre mayúsculas y minúsculas, y `Main` no es `main`. La puerta de entrada del programa debe llamarse exactamente `main`, en minúsculas.

Error provocado 3 (sin `#include <stdio.h>`):

```
int main() {
    printf("Esteban\n");
    printf("30 anos\n");
    return 0;
}
```

El compilador lanzará una advertencia como "implicit declaration of function 'printf'" (declaración implícita de la función). Con `-Wall` la verás claramente. La lección: sin la cabecera, el compilador no conoce `printf`. En compiladores modernos esto puede ser solo una advertencia y el programa hasta podría funcionar, pero es mala práctica: incluye siempre las cabeceras que necesites.

La verdadera lección del ejercicio: los mensajes de error no son reganos, son pistas. Aprende a leerlos y la mitad de tu frustración como programador desaparecerá.

Solución del ejercicio 6

```
#include <stdio.h>
int main() {
    printf("HORARIO SEMANAL\n");
    printf("LUN\tMAR\tMIE\tJUE\tVIE\tSAB\tDOM\n");
    printf("Gym\tC\tGym\tC\tGym\tLibre\tLibre\n");
    printf("Leer\tCine\tLeer\tCine\tLeer\tSalir\tDormir\n");
    return 0;
}
```

Por qué funciona: el tabulador `\t` avanza el cursor hasta la siguiente parada de tabulación, lo que alinea las columnas siempre que los textos no sean demasiado largos. Si una palabra es muy larga, puede "empujar" la columna y desalinear la tabla: en ese caso agrega un `\t` extra o acorta el texto. Este ejercicio te entrena el ojo para la presentación de datos en texto, una habilidad que usarás cuando imprimas tablas de números en unidades futuras. Personalízalo con tu horario real: así prácticas y de paso organizas tu semana.

Solución del ejercicio 7

```
#include <stdio.h>
int main() {
    printf("*****\n");
    printf("*          MI CONTRATO DE PROGRAMADOR          *\n");
    printf("*****\n");
    printf("Yo, Esteban, me comprometo a:\n");
    printf("1. Programar todos los días, aunque sea 30 minutos.\n");
    printf("2. No copiar código que no entienda.\n");
    printf("3. Leer los mensajes de error con calma.\n");
    printf("4. Terminar las 30 clases de este curso.\n");
    printf("5. Construir mi proyecto final en la Unidad 10.\n");
    printf("6. Celebrar cada programa que compile a la primera.\n");
    printf("7. Pedir ayuda cuando me atore, sin pena.\n");
    printf("8. Recordar que todo experto alguna vez fue principiante.\n");
    printf("*****\n");
    printf("Firma: _____ Fecha: _____\n");
    printf("*****\n");
    return 0;
}
```

Por qué funciona: técnicamente no hay nada nuevo, y esa es la intención: el reto estaba en la constancia de escribir 8 líneas coherentes dentro de un marco decorativo, manteniendo la alineación del marco. Fíjate en que todas las líneas del marco tienen la misma longitud: si una línea de asteriscos es más corta o más larga, el marco se ve chueco. Contar caracteres es parte del oficio. Y el verdadero valor del ejercicio no es técnico: es psicológico. Escribir tu compromiso y verlo impreso cada día es una de las técnicas más efectivas para terminar lo que empiezas. Los cursos no los terminan los más inteligentes, los terminan los más constantes.

Resumen de la unidad

Llegamos al final de la Unidad 1. Si hiciste los ejemplos y los ejercicios, ya no eres alguien que "quiere aprender a programar": eres alguien que ya programa. Repasemos los puntos clave de todo lo que vimos:

- C nació entre 1971 y 1973 de la mano de Dennis Ritchie en los Laboratorios Bell, como evolución del lenguaje B de Ken Thompson, y sirvió para reescribir el sistema operativo Unix. Su historia explica su filosofía: control total, eficiencia máxima y cero magia escondida.
- El libro "The C Programming Language" de Kernighan y Ritchie (1978) definió el lenguaje durante años; después llegaron los estándares oficiales ANSI C (1989), C99, C11, C17 y C23. Un programa bien escrito en C de hace décadas sigue compilando hoy.
- C sigue vivo más de 50 años después porque está en todas partes: núcleos de sistemas operativos (Linux, Windows, macOS, Android, iOS), sistemas embebidos y microcontroladores, motores de videojuegos, bases de datos, servidores web e intérpretes de otros lenguajes. Aprender C es aprender como piensa realmente la computadora.
- Un compilador traduce tu código fuente a un programa ejecutable. En este curso usamos GCC: se instala con MinGW-w64 en Windows (sin olvidar agregar la carpeta bin al PATH), con `sudo apt install gcc` en Linux, y con `xcode-select --install` en macOS. Se verifica con `gcc --version`.
- El ciclo de trabajo es: escribir el código en un archivo `.c`, guardarlo, compilarlo con `gcc programa.c -o programa`, y ejecutarlo (`programa` en Windows, `./programa` en Linux/macOS). El silencio del compilador significa éxito.
- Compilar tiene cuatro etapas internas: preprocesamiento (procesa las líneas `#`), compilación (traduce a ensamblador y detecta errores de sintaxis), ensamblado (genera código objeto) y enlazado (junta tu código con las bibliotecas para crear el ejecutable).
- La anatomía del programa mínimo es: `#include <stdio.h>` para traer la biblioteca de entrada y salida; `int main()` como puerta de entrada obligatoria que devuelve un entero; `printf("texto\n")` para imprimir en pantalla; `return 0` para avisar que todo salió bien; y las llaves `{}` que delimitan el cuerpo de la función. Cada instrucción termina con punto y coma.
- Las secuencias de escape permiten escribir caracteres especiales: `\n` salto de línea, `\t` tabulador, `\\` diagonal invertida, `\"` comillas dobles dentro del texto y `%%` el signo de porcentaje.

- Los errores más comunes del principiante son: olvidar guardar antes de compilar, compilar sin `-o`, olvidar el punto y coma (el error se reporta una línea después), confundir mayúsculas con minúsculas (`Main` no es `main`), y no leer los mensajes del compilador, que en realidad son pistas para corregir.
- Las buenas prácticas desde el día uno son: compilar siempre con `-Wall`, indentar el código con 4 espacios, comentar el porqué y no el que, probar después de cada cambio pequeño, y nunca copiar código que no entiendas.
- En la Unidad 2 daremos el siguiente gran paso: las variables y los tipos de datos, y aprenderemos a usar `printf` con formatos para números y `scanf` para leer datos del teclado. Tu programa dejara de ser un cartel estático y empezara a conversar con el usuario. Nos vemos ahí, y recuerda que en lucasegaacademy.com tienes más cursos y recursos para seguir avanzando.

UNIDAD 2: Variables, tipos y printf

Introducción

Bienvenido a la Unidad 2. En la Unidad 1 lograste algo que mucha gente nunca termina: instalaste un compilador, escribiste tu primer programa en C y lo viste correr. Tu programa decía "Hola, mundo" y la computadora obedeció. Fue emocionante, pero seamos honestos: ese programa era un loro. Repetía siempre lo mismo, no recordaba nada, no calculaba nada, no podía guardar ningún dato. Era un loro muy obediente, pero un loro al fin.

Ahora vas a darle memoria. En esta unidad tu programa va a aprender a guardar información: la edad de un usuario, el precio de un producto, la calificación de un examen, una letra, un nombre completo. Y no solo guardarla: también va a aprender a mostrarla en pantalla de forma ordenada y profesional, como lo hacen los programas de verdad.

Para lograr eso necesitas entender una idea fundamental: en C los datos necesitan tipo. Y para entenderla, imaginemos juntos una analogía que te va a acompañar durante todo el curso.

Piensa en la memoria de tu computadora como un almacén gigantesco. No un almacén cualquiera, sino uno de esos galpones enormes donde hay miles y miles de cajas apiladas en estantes. Cada caja es un pedazo de memoria. Cuando tu programa corre, necesita guardar cosas en ese almacén: números, letras, textos. El problema es que el almacén está oscuro y las cajas son todas iguales por fuera. Si guardas algo en una caja y no la marcas, cuando vuelvas por ella no vas a saber dónde quedó. Peor aun: no vas a saber qué había dentro.

Por eso existen las variables: una variable es una caja con una etiqueta. Tu le pones un nombre a la caja (la etiqueta) y guardas algo dentro. Cuando necesitas el dato, no tienes que recorrer todo el almacén: simplemente pides "la caja que se llama edad" y la computadora va directo a ella. La etiqueta es el nombre de la variable. Lo que guardas adentro es el valor.

Pero aquí viene la parte interesante, y es la razón de ser de esta unidad: no todas las cajas son iguales. No es lo mismo guardar un par de zapatos que guardar agua. Para los zapatos necesitas una caja de cartón de cierto tamaño. Para el agua necesitas un recipiente sellado, porque si la echas en una caja de cartón se escurre y haces un desastre. Los datos son iguales: un número entero no ocupa el mismo espacio ni se guarda de la misma forma que un número con decimales, y una letra se guarda de forma distinta a un texto largo.

El tipo de dato es la descripción de la caja: que tamaño tiene, que cabe dentro y cómo se debe interpretar su contenido. Cuando declaras una variable en C, le estás diciendo a la computadora tres cosas al mismo tiempo: "reservame una caja de este tamaño (el tipo), ponle esta etiqueta (el nombre) y aquí te dejo lo que va adentro (el valor)". Si intentas guardar agua en una caja de carton, es decir, un número con decimales en una variable pensada solo para enteros, vas a perder información. Y si intentas leer una caja de zapatos como si fuera un garrafon de agua, vas a obtener basura. En C, el tipo manda.

Hay algo todavía más profundo que tienes que entender desde hoy: dentro de la computadora todo son unos y ceros. Todo. El número 65, la letra 'A', el color rojo de una imagen, el sonido de una canción: todo se reduce a secuencias de bits. Entonces, cuando la computadora encuentra en memoria la secuencia 01000001, como sabe si eso es el número 65 o la letra 'A'? La respuesta es: lo sabe por el tipo. El tipo es el lente con el que se miran los bits. Con el lente de "número entero", 01000001 se lee como 65. Con el lente de "carácter", esos mismos bits se leen como la letra 'A'. El mismo contenido, dos lecturas distintas. Por eso en C no existe eso de "guardar un dato así nomas": siempre guardas un dato con su tipo, y ese tipo decide cómo se interpreta cada bit.

Esta idea es la que separa a quien programa de verdad de quien solo copia código. Cuando entiendas los tipos, vas a entender por que a veces un cálculo te da un resultado raro, por que un número "se corta", por que una división entre enteros no da decimales, y por que ciertos errores aparecen. La mayoría de los dolores de cabeza de un principiante en C vienen de no respetar los tipos. Esta unidad existe para que tu no sufras eso.

Vamos a repasar lo que vas a ver, clase por clase, para que sepas a dónde vamos.

En la clase 4, "Variables y tipos de datos", vas a aprender que es exactamente una variable: cómo se declara, cómo se le asigna un valor y cómo se inicializa. Conocerás los cuatro tipos básicos de C: int para números enteros, float y double para números con decimales, y char para caracteres individuales. Aprenderás cuánto espacio ocupa cada uno en memoria y que rangos de valores puede guardar. También descubrirás el secreto mejor guardado de los principiantes: que el tipo char en realidad guarda números, y que existe una tabla llamada ASCII que traduce esos números a letras.

En la clase 5, "Constantes, #define y const", vas a aprender a crear valores que no cambian nunca: el valor de PI, el IVA, los días de la semana, el límite de intentos de un juego. C te da dos herramientas para eso: const, que crea una variable de solo lectura, y #define, que es una orden para el preprocesador. Parecen lo mismo, pero no lo son, y vas a aprender exactamente cuando usar cada una. Esta clase te va a enseñar a escribir programas más legibles y más fáciles de mantener, porque un programa lleno de números mágicos regados por todos lados es una pesadilla.

En la clase 6, "printf: salida formateada", vas a dominar la función más famosa de C. Ya la usaste en la Unidad 1 para imprimir texto fijo, pero printf es muchísimo más que eso: es una herramienta de formato profesional. Aprenderás los especificadores de formato (%d, %f, %c, %s, %x, %o y más), que son como moldes que le dicen a printf como mostrar cada dato. Aprenderás a controlar cuántos decimales se muestran, a alinear columnas para hacer tablas bonitas, y a usar secuencias de escape como \n para saltos de línea y \t para tabulaciones. Cuando termines esta clase, tus programas van a verse como programas de verdad, con salidas limpias y ordenadas.

Al final de la unidad encontrarás una sección de ejemplos paso a paso con código completo que puedes compilar y modificar, una sección de práctica con ejercicios y sus soluciones explicadas, y un resumen de todo lo aprendido. Mi recomendación es que no solo leas el código: escríbelo tú mismo, compílalo, corrélo y luego rompeló a propósito. Cambia un %d por un %f y mira qué pasa. Declara una variable sin inicializarla e imprime su valor. Esos experimentos te van a enseñar más que cien páginas de teoría.

Un consejo antes de empezar: esta unidad es la base de TODO lo que sigue. Los apuntadores de la Unidad 5, los arreglos de la Unidad 4, las estructuras de la Unidad 6... todo se construye sobre variables y tipos. Si dominas esta unidad, el resto del curso se te va a hacer mucho más fácil. Si la pasas de largo, cada unidad nueva te va a costar el doble. Así que tomátele con calma, práctica mucho y no avances hasta sentirte cómodo.

Y recuerda que no estás solo en este camino. En lucasegaacademy.com tienes todo el curso completo de C con más ejemplos, ejercicios y material de apoyo para seguir practicando. Esta unidad es tu siguiente paso, y cuando la termines vas a mirar tu programa "Hola, mundo" y vas a sonreír, porque vas a saber exactamente como convertirlo en algo mucho más poderoso.

Vamos a ello. Abre tu editor, prepara tu compilador y empecemos a llenar cajas en el almacén.

Explicación detallada

Que es una variable: declaración, inicialización y asignación

Una variable es un espacio de memoria con nombre donde tu programa guarda un dato que puede cambiar durante la ejecución. Volviendo a la analogía del almacén: la variable es la caja etiquetada. El nombre que le pones es la etiqueta. El tipo que eliges es el tamaño y el material de la caja. Y el valor es lo que guardas dentro.

En C, trabajar con una variable tiene tres momentos distintos que debes aprender a distinguir: la declaración, la inicialización y la asignación. Suenan parecido, pero cada uno hace algo diferente y confundirlos es fuente de errores.

La declaración es el momento en que le dices a la computadora: "reservame una caja". Le das el tipo y el nombre, y la computadora aparta ese espacio en memoria. Todavía no guardas nada dentro. La sintaxis es sencilla: primero el tipo, luego el nombre, y punto y coma al final.

```
int edad;
```

Con esa línea le estás diciendo a la computadora: "reservame espacio para un número entero y ponle la etiqueta edad". A partir de ese momento, el nombre edad existe en tu programa y representa ese pedazo de memoria. Pero ojo con un detalle crucial: la caja recién reservada NO está vacía. Bueno, más bien, no está limpia. Contiene lo que haya quedado ahí de antes: basura. La memoria es reutilizada constantemente por el sistema, así que una variable recién declarada contiene un valor impredecible, restos de algún otro programa o de alguna otra parte del tuyo. Esto es importantísimo y lo vamos a repetir varias veces: una variable declarada pero no inicializada contiene basura.

La inicialización es darle un valor inicial a la variable en el mismo momento en que la declaras. Es como etiquetar la caja y meter el contenido de una vez.

```
int edad = 25;
```

Aquí pasan dos cosas en una sola línea: se reserva el espacio y se guarda el número 25 dentro. Esta es la forma recomendada de crear variables en C: declarar e inicializar al mismo tiempo. Te ahorra la basura, te ahorra errores y hace tu código más claro.

La asignación es cambiar el valor de una variable que ya existe. La variable ya fue declarada antes, y ahora le metes un valor nuevo, reemplazando el que tenía.

```
edad = 26;
```

Fíjate que aquí ya no se escribe el tipo int. El tipo solo se escribe una vez, en la declaración. Si lo escribes de nuevo, la computadora cree que estás declarando una variable nueva y te va a marcar error porque ya existe una con ese nombre. La asignación usa el operador = , que en C

no significa "igual que en matemáticas" sino "guardame este valor aquí". Es una orden, no una comparación. Esta diferencia confunde a muchos principiantes, así que grabatela: = asigna, == compara. De la comparación hablaremos en la Unidad 3.

También puedes declarar varias variables del mismo tipo en una sola línea, separandolas con comas:

```
int día, mes, año;
```

```
int x = 5, y = 10, z = 15;
```

Esto es válido, pero usalo con moderación. Si la línea se vuelve larga o confusa, mejor una variable por línea. La claridad siempre gana.

Los nombres de las variables tienen reglas en C. Pueden contener letras (mayúsculas y minúsculas), números y el guion bajo `_`. No pueden empezar con un número, no pueden contener espacios ni caracteres especiales, y no pueden ser palabras reservadas del lenguaje como `int`, `return` o `while`. Además, C distingue entre mayúsculas y minúsculas: `Edad` y `edad` son dos variables distintas. Técnicamente puedes llamar a tus variables como quieras dentro de esas reglas, pero aquí entra la buena práctica: usa nombres descriptivos. Una variable llamada `e` no dice nada; una variable llamada `edadDelUsuario` dice exactamente que guarda. Tu yo del futuro, cuando relea el código dentro de seis meses, te lo va a agradecer.

Por convención, en C los nombres de variables se escriben en minúsculas, y si el nombre tiene varias palabras se separan con guion bajo: `total_ventas`, `promedio_final`, `numero_de_intentos`. Esta convención se llama `snake_case` (porque las palabras parecen unidas como una serpiente) y es la más usada en el mundo de C. Siguela y tu código se verá profesional.

Otro detalle: el nombre de la variable debe reflejar lo que guarda, no el tipo. No llames a una variable `int_edad` o `miFloat`; eso es redundante y se vuelve un problema si algún día cambias el tipo. Llamala simplemente `edad` o `precio` o `calificación`.

Para cerrar esta sección, una idea que te va a servir todo el curso: piensa en la variable como un trato entre tu y la computadora. Tu le prometes: "esta caja siempre va a guardar este tipo de dato". Y la computadora te promete: "siempre que pidas esta etiqueta, te dare el último valor que guardaste ahí". Mientras ambos cumplan el trato, todo funciona. Cuando alguien lo rompe, por ejemplo guardando un texto donde iba un número, aparecen los errores raros. Respeta los tipos y los tipos te van a respetar a ti.

Tipos basicos: int, float, double y char

C es un lenguaje de tipos estaticos, lo que significa que el tipo de cada variable se decide cuando escribes el programa y ya no cambia mientras corre. No puedes declarar una variable como int y después guardar en ella un texto. Esa rigidez, que al principio puede parecer molesta, es en realidad una de las grandes fortalezas de C: permite que el compilador detecte errores antes de que el programa corra, y hace que los programas sean rápidos porque la computadora sabe exactamente cuánto espacio necesita cada dato y como manejarlo.

C tiene cuatro tipos basicos, y con ellos puedes representar casi cualquier dato. Vamos a conocerlos uno por uno, con sus tamaños y sus rangos. Un aviso importante: los tamaños exactos pueden variar según la computadora y el compilador, porque C define tamaños minimos pero permite que sean mayores. Los valores que te doy aquí son los tipicos en computadoras modernas de 64 bits con gcc, que es lo que estás usando en este curso. Más adelante aprenderás a usar el operador sizeof para preguntarle a tu propia computadora cuánto mide cada tipo.

El tipo int guarda números enteros: positivos, negativos y el cero. Es el tipo más usado en C. Un int típicamente ocupa 4 bytes de memoria, es decir, 32 bits. Con 32 bits puedes representar 2 elevado a 32 valores distintos, que repartidos entre negativos y positivos dan un rango aproximado de -2,147,483,648 a 2,147,483,647. Para que te des una idea: dos mil millones. Con eso te alcanza para contar la edad de una persona, el número de habitantes de un país, los segundos de un día, los intentos de un juego. Casi todo lo que cuentes en la vida diaria cabe en un int.

```
int habitantes = 128000000; int temperatura = -5; int intentos = 0;
```

Si necesitas contar cosas más grandes, por ejemplo los segundos desde 1970 o el número de granos de arena de una playa, existe el tipo long, que típicamente ocupa 8 bytes y llega hasta números de 19 dígitos. Y si quieres solo números positivos y duplicar el rango positivo, existe unsigned int. No te preocupes por memorizar todos los sabores ahora; con int tienes para rato, y cuando necesites más, ya sabrás que existen.

El tipo float guarda números con decimales, lo que en matemáticas se llaman números reales: 3.14, -0.5, 99.99. Un float típicamente ocupa 4 bytes y puede guardar números con alrededor de 6 o 7 dígitos decimales de precisión. Suena bien, pero aquí hay una trampa que debes conocer desde hoy: los números con decimales en la computadora no son exactos. Por la forma en que se guardan en binario, un float a veces guarda 0.1 como 0.10000000149 o algo parecido. Para la mayoría de los usos cotidianos no importa, pero nunca compares dos floats con == esperando exactitud perfecta, y nunca uses float para dinero en un sistema bancario real. Para eso existen técnicas especiales que verás más adelante en tu carrera.

```
float precio = 19.99; float pi = 3.14159; float temperatura_corporal = 36.6;
```

Fíjate en un detalle de sintaxis: en C los decimales se escriben con punto, no con coma. Si escribes 19,99 el compilador se va a quejar. El punto decimal es obligatorio en la notación anglosajona que usa C.

El tipo double es el hermano mayor del float: también guarda números con decimales, pero ocupa típicamente 8 bytes y ofrece alrededor de 15 o 16 dígitos de precisión. Es mucho más preciso que float. De hecho, cuando escribes un número con decimales directamente en el código, como 3.14159, C lo trata como double por defecto. Por eso muchos programadores prefieren usar double para cálculos donde la precisión importa: promedios, cálculos científicos, coordenadas. Ocupa el doble de memoria, pero en computadoras modernas eso rara vez es un problema.

```
double pi_preciso = 3.141592653589793; double distancia_tierra_sol = 149597870.7;
```

Una regla práctica: usa float cuando la memoria sea limitada o trabajes con gráficos y videojuegos donde la velocidad importa más que la precisión extrema; usa double cuando hagas cálculos matemáticos, científicos o financieros donde cada decimal cuenta. Cuando dudes, double es la opción segura.

El tipo char guarda un solo carácter: una letra, un dígito, un símbolo de puntuación. Ocupa típicamente 1 byte. Los caracteres se escriben entre comillas simples, no dobles: 'A', 'z', '5', '?'. Las comillas simples son para un carácter; las comillas dobles son para textos completos, que en C se llaman cadenas y los verás en la clase 21. Confundir unas con otras es un error clasísimo de principiante, así que fíjate bien desde ahora.

```
char inicial = 'J'; char simbolo = '$'; char digito = '7';
```

Ojo con algo curioso: el carácter '7' no es el número 7. Es el símbolo del siete, el dibujito. Para la computadora son cosas distintas, y en la siguiente sección vas a entender exactamente por qué. Por ahora quedate con la idea: char guarda símbolos, no cantidades.

Aquí tienes un resumen de los cuatro tipos básicos con sus tamaños típicos en gcc moderno:

- char: 1 byte. Guarda un carácter.
- int: 4 bytes. Guarda enteros de -2,147,483,648 a 2,147,483,647.
- float: 4 bytes. Guarda decimales con unas 6-7 cifras de precisión.
- double: 8 bytes. Guarda decimales con unas 15-16 cifras de precisión.

Existe además el tipo void, que significa "sin tipo" o "vacío", pero no sirve para guardar datos: se usa para funciones que no devuelven nada, y lo verás en la clase 16. Y existen los

modificadores signed, unsigned, short y long que ajustan el tamaño y el rango de los tipos enteros. Los menciono para que no te asusten cuando los veas, pero no necesitas dominarlos todavía.

La lección más importante de esta sección es: elige el tipo según el dato, no según tu antojo. Si el dato es una cantidad contable (edad, número de alumnos, intentos), usa int. Si tiene decimales (precio, promedio, temperatura), usa float o double. Si es un símbolo individual (inicial, respuesta s/n, calificación con letra), usa char. Esa decisión, que toma dos segundos, evita horas de errores misteriosos.

El tipo char y el código ASCII

Ahora viene una de las revelaciones más divertidas de esta unidad: el tipo char, que según acabamos de decir guarda caracteres, en realidad guarda números. No es magia ni es trampa: es que las letras también son números para la computadora.

Recuerda la idea central: en memoria todo son bits. Un char ocupa 1 byte, o sea 8 bits, y con 8 bits se pueden representar 256 valores distintos: del 0 al 255. Entonces, como guardamos una letra en esos 256 valores? Con un acuerdo. Hace décadas, los ingenieros se pusieron de acuerdo en una tabla que asigna un número a cada símbolo: a la 'A' le toca el 65, a la 'B' el 66, a la 'a' minúscula el 97, al dígito '0' el 48, al espacio el 32, y así con todos. Esa tabla se llama ASCII, por las siglas en inglés de Código Estadounidense Estándar para el Intercambio de Información.

Cuando escribes char letra = 'A'; lo que realmente se guarda en memoria es el número 65. Las comillas simples son una cortesía del lenguaje: le dicen al compilador "busca en la tabla ASCII que número le corresponde a este símbolo y guarda ese número". Y cuando imprimes esa variable como carácter, el proceso va al revés: la computadora toma el 65, busca en la tabla y muestra la 'A'.

Esto tiene consecuencias prácticas muy interesantes. Como un char es en el fondo un número entero pequeño, puedes hacer aritmética con caracteres. Mira este truco clásico:

```
char letra = 'A'; letra = letra + 1;
```

Cuánto vale ahora letra? La 'A' es 65, más 1 es 66, y el 66 en ASCII es la 'B'. Así que letra ahora guarda la 'B'. Este truco se usa muchísimo: para recorrer el abecedario, para convertir minúsculas en mayúsculas (en ASCII la 'a' es 97 y la 'A' es 65, se llevan 32 de diferencia), para cifrados sencillos como el cifrado Cesar. Cuando llegues a los ejercicios vas a practicarlo.

La tabla ASCII tiene 128 símbolos principales, del 0 al 127. Los primeros 32 son caracteres de control, símbolos invisibles que no se imprimen pero dan ordenes: el 10 es el salto de línea

(por eso `\n` funciona), el 9 es la tabulación, el 7 es un beep. Del 32 al 126 están los símbolos imprimibles: primero los signos de puntuación y el espacio, luego los dígitos del '0' (48) al '9' (57), luego las mayúsculas de 'A' (65) a 'Z' (90), luego más símbolos, y luego las minúsculas de 'a' (97) a 'z' (122). El 127 es la tecla de borrar. Es una tabla diseñada con lógica: los dígitos están en orden, las mayúsculas están en orden y las minúsculas están en orden, lo que permite hacer comparaciones y conversiones con aritmética simple.

Aquí tienes los valores que más vas a usar, para que los tengas a la mano:

- '0' a '9': del 48 al 57. Fíjate: el carácter '0' NO es el número 0, es el número 48.
- 'A' a 'Z': del 65 al 90.
- 'a' a 'z': del 97 al 122.
- espacio ' ': 32.
- salto de línea '\n': 10.

Un error típico: querer guardar el número 65 en un char y esperar ver "65" en pantalla. Si haces `char c = 65;` y lo imprimes como carácter, verás la letra 'A', porque 65 es el código de la 'A'. Si lo imprimes como número, verás 65. El mismo dato, dos lecturas: el lente del tipo decide, como te conte en la introducción. En la clase 6, cuando veamos `printf` a fondo, vas a aprender a elegir el lente con los especificadores `%c` y `%d`.

Otro detalle importante: como char es un número de 1 byte, también puedes usarlo para guardar enteros muy pequeños, del -128 al 127 en su versión con signo. Algunos programas lo aprovechan para ahorrar memoria cuando tienen millones de valores pequeños. No es su uso principal, pero es bueno que sepas que se puede.

Y una advertencia sobre el español: la tabla ASCII original no incluye la ñ ni las vocales acentuadas. Es un estándar estadounidense de los años sesenta, y en esa época nadie pensó en nuestros caracteres. Por eso en este libro escribimos todo sin acentos ni ñes: no es solo una regla de estilo, es una lección práctica de cómo trabaja C con el texto. Existen extensiones y otros estándares como UTF-8 que sí incluyen nuestros caracteres, y los verás más adelante en tu camino, pero el C clásico y el ASCII puro son el punto de partida. Cuando veas que un programa en C muestra símbolos raros en lugar de una ñ, ya vas a saber por qué: alguien intentó meter un carácter que no está en la tabla.

Para cerrar: el char es el ejemplo perfecto de la idea central de esta unidad. Un byte de memoria con unos y ceros no significa nada por sí solo. El tipo char le dice a la computadora: "interpreta estos bits con la tabla ASCII". Y de pronto, un 65 se convierte en una 'A'. Los tipos no son burocracia del lenguaje: son el diccionario con el que la computadora traduce los bits a significado.

Constantes: const y #define (y sus diferencias)

Hasta ahora todas las variables que hemos visto pueden cambiar de valor: la edad cumple años, el saldo sube y baja, los intentos se acumulan. Pero muchos datos de un programa no cambian nunca. El valor de PI siempre es 3.14159. El IVA en Mexico es 16 por ciento. Una semana siempre tiene 7 días. El número máximo de intentos en tu juego siempre es 5. Esos valores fijos se llaman constantes, y C te da dos formas de crearlas: const y #define. Hacen un trabajo parecido pero por dentro son mecanismos completamente distintos, y elegir mal entre ellas es un error muy común. Vamos a verlas con calma.

La palabra clave const crea una variable de solo lectura. Se declara como una variable normal, pero con const al frente, y OBLIGATORIAMENTE se inicializa en la declaración, porque después ya no la podrás cambiar.

```
const float PI = 3.14159; const int DIAS_SEMANA = 7; const float IVA = 0.16;
```

Si más adelante en el programa intentas hacer `PI = 3.0;`, el compilador te va a detener con un error. Y eso es exactamente lo que queremos: que la computadora cuide que nadie cambie por accidente un valor que debe ser fijo. Fíjate que por convención las constantes se escriben en MAYÚSCULAS con guion bajo separando palabras: PI, DIAS_SEMANA, IVA, MAX_INTENTOS. No es obligatorio, pero es una convención universal en C: cuando ves un nombre en mayúsculas, sabes al instante que es una constante. Adoptala desde hoy.

La otra forma es #define, que no crea una variable sino una orden para el preprocesador. Recuerda la Unidad 1: antes de compilar, el preprocesador hace una pasada por tu código y ejecuta las líneas que empiezan con #. Cuando encuentra #define PI 3.14159, hace algo muy simple y muy literal: busca en todo tu código cada aparición de la palabra PI y la reemplaza por 3.14159, como si fuera un buscar y reemplazar de un editor de texto. No hay variable, no hay tipo, no hay espacio en memoria reservado con nombre: es una sustitución de texto.

```
#define PI 3.14159 #define DIAS_SEMANA 7 #define IVA 0.16 #define MAX_INTENTOS 5
```

Fíjate en tres detalles de sintaxis: primero, no lleva punto y coma al final (si lo pones, el punto y coma se vuelve parte del reemplazo y te da errores extranísimos); segundo, por convención también se escribe en mayúsculas; tercero, normalmente se coloca arriba del todo, junto a los #include.

Entonces, si las dos sirven para lo mismo, cuál uso? La respuesta corta: prefiere const casi siempre, y usa #define para casos específicos. La respuesta larga necesita que entiendas las diferencias de fondo:

Primera diferencia: el tipo. Una constante `const` tiene tipo: `const float PI` es un flotante, `const int DIAS_SEMANA` es un entero. El compilador sabe que tipo es y puede verificar que la uses correctamente. En cambio, `#define PI 3.14159` no tiene tipo: es solo texto que será reemplazado. El compilador nunca ve la palabra `PI`, solo ve el número suelto. Esto significa que con `#define` pierdes la verificación de tipos, una de las redes de seguridad más valiosas de C.

Segunda diferencia: el alcance. Una constante `const` obedece las reglas de alcance como cualquier variable: si la declaras dentro de una función, solo existe ahí; si la declaras fuera de todas las funciones, es global. Ya hablaremos del alcance en la clase 18. En cambio, un `#define` es global a partir del punto donde aparece hasta el final del archivo, sin importar funciones ni bloques. Es más difícil de controlar.

Tercera diferencia: la depuración. Cuando depuras un programa con `const float PI`, el depurador conoce el nombre `PI` y te muestra su valor. Con `#define`, el nombre `PI` desaparece antes de compilar: el depurador solo ve números sueltos y es más difícil rastrear de donde salieron.

Cuarta diferencia: el comportamiento con expresiones. Aquí hay una trampa famosa. Mira esto:

```
#define DOBLE(x) x * 2
```

Parece una función que duplica, pero es una sustitución de texto. Si escribes `DOBLE(3 + 1)`, el preprocesador lo convierte en `3 + 1 * 2`, que por precedencia de operadores es `3 + 2 = 5`, no 8. Para arreglarlo habría que escribir `#define DOBLE(x) ((x) * 2)`, con paréntesis por todos lados. Es un ejemplo de por qué las macros con parámetros son peligrosas para principiantes. Las constantes `const` no tienen este problema porque son valores de verdad, no texto.

Entonces, cuando SI conviene `#define`? En tres casos principales. Primero, para valores que necesita el preprocesador mismo, como el tamaño de un arreglo declarado de forma estática: `int calificaciones[MAX_ALUMNOS]`; con `#define MAX_ALUMNOS 30` funciona, con `const int` no siempre (depende del estándar de C que uses). Segundo, para compilación condicional, como `#ifdef DEBUG`, que es una técnica avanzada para incluir código solo en modo de pruebas. Tercero, por tradición en código antiguo y en bibliotecas del sistema, donde verás miles de `#define`. Cuando trabajes con código ajeno, los vas a encontrar por todos lados y debes saber leerlos.

Hay un tercer concepto que debes conocer para no confundirlo: los literales. Un literal es un valor escrito directamente en el código: el 25 en `int edad = 25;`, el 3.14 en `float pi = 3.14;`, la 'A' en `char letra = 'A';`. Los literales son constantes anónimas: tienen valor fijo pero no tienen

nombre. El problema de regar literales por todo el programa es que se vuelven "números mágicos": dentro de seis meses vas a encontrar un 0.16 en tu código y no vas a recordar si era el IVA, un descuento o un margen de error. Por eso la buena práctica es: si un valor literal aparece más de una vez o tiene un significado en el negocio, dale nombre con `const` o `#define`. Tu código se vuelve autodocumentado: `precio * IVA` se entiende solo; `precio * 0.16` requiere un comentario.

En resumen: `const` crea una constante de verdad, con tipo, con alcance y visible para el depurador: es tu opción por defecto. `#define` hace sustitución de texto antes de compilar: usalo para tamaños de arreglos, compilación condicional y cuando trabajes con código que ya lo usa. Y a los números mágicos, ni los saludes: ponles nombre.

printf a fondo: especificadores, precisión, ancho de campo y secuencias de escape

Ya usaste `printf` en la Unidad 1 para imprimir "Hola, mundo". Ahora vas a conocerla de verdad, porque `printf` es mucho más que una impresora de texto: es una fábrica de formato. Su nombre viene de "print formatted", imprimir con formato, y su superpoder es tomar tus variables y mostrarlas en pantalla exactamente como tu quieras: con tantos decimales como decidas, alineadas en columnas perfectas, mezcladas con texto explicativo.

La forma general de `printf` es esta: `printf("texto con moldes", valor1, valor2, ...);`. El primer argumento es una cadena de formato: texto normal donde intercalas especificadores, que son códigos que empiezan con `%` y funcionan como moldes vacíos. Cada molde se rellena con el valor correspondiente, en orden. Mira:

```
int edad = 25; printf("Tienes %d años", edad);
```

El `%d` es el molde para números enteros. `printf` lo reemplaza por el valor de `edad` y en pantalla aparece: Tienes 25 años. Si `edad` valiera 40, aparecería 40, sin cambiar nada más. Esa es la gracia: el texto es fijo, los datos son variables.

Los especificadores básicos que debes dominar son estos:

- `%d` o `%i`: números enteros (int). El más usado.
- `%f`: números con decimales (float y double). Por defecto muestra 6 decimales.
- `%c`: un carácter (char). Muestra el símbolo según ASCII.
- `%s`: una cadena de texto (string). La verás a fondo en la clase 21, pero la usarás desde hoy para imprimir textos guardados en variables.
- `%x`: entero en hexadecimal (base 16), con letras minúsculas.

- %o: entero en octal (base 8).
- %u: entero sin signo (unsigned).
- %ld: entero largo (long).
- %e: decimal en notación científica, útil para números muy grandes o muy pequeños.
- %%%: imprime un signo de porcentaje literal, porque un % solo siempre inicia un especificador.

Y aquí viene la primera regla de oro de printf: el especificador debe corresponder al tipo del dato. Si imprimes un int con %d, perfecto. Si imprimes un int con %f, printf va a interpretar esos bits como si fueran un decimal y te va a mostrar basura, un número absurdo. Si imprimes un float con %d, lo mismo al revés. El compilador moderno a veces te avisa con un warning, pero no siempre, y en ejecución no hay red de seguridad: verás basura y no sabrás por qué. Cada vez que escribas un printf, preguntate: qué tipo es cada variable y qué molde le corresponde?

Vamos con ejemplos de cada especificador para que los veas en acción:

```
int edad = 25; float precio = 19.99; char inicial = 'J'; printf("Edad: %d\n", edad); printf("Precio: %f\n", precio); printf("Inicial: %c\n", inicial);
```

En pantalla verías: Edad: 25 Precio: 19.990000 Inicial: J

Fíjate en el precio: %f muestra 6 decimales por defecto, así que 19.99 se ve como 19.990000. Para un ticket de compra eso se ve mal. Aquí entra el control de precisión: entre el % y la letra puedes poner un punto y un número para indicar cuántos decimales quieres.

```
printf("Precio: %.2f\n", precio);
```

Ahora verás: Precio: 19.99. El %.2f significa "decimal con 2 cifras después del punto". printf redondea automáticamente: si el precio fuera 19.999, con %.2f verías 20.00. Puedes usar %.0f para cero decimales, %.3f para tres, %.10f para diez. Este truco lo usarás poquísimo... es broma, lo usarás poquísimas veces? No: lo usarás TODO el tiempo. Precios, promedios, porcentajes, calificaciones: casi todo lo que muestres con decimales necesita precisión controlada.

El ancho de campo es el siguiente nivel. Entre el % y la letra puedes poner un número que indica el ancho mínimo que ocupará el valor en pantalla, rellenando con espacios a la izquierda si es necesario. Esto sirve para alinear columnas y hacer tablas.

```
printf("%5d\n", 42);
```

El %5d dice: "muestra el entero ocupando al menos 5 espacios". El 42 ocupa 2, así que printf pone 3 espacios antes: " 42". Si el número fuera más largo que el ancho, no se corta: el ancho es un mínimo, no un máximo.

Si quieres alinear a la izquierda en lugar de a la derecha, pones un guion: %-5d. Y si quieres rellenar con ceros en lugar de espacios, pones un cero: %05d muestra 00042. Combinando ancho y precisión puedes hacer tablas profesionales:

```
printf("%-15s %5d %8.2f\n", "Manzanas", 10, 25.50);
```

Esto imprime el nombre alineado a la izquierda en 15 espacios, la cantidad alineada a la derecha en 5, y el precio con 2 decimales en 8 espacios. Con varias líneas así construyes un ticket o un reporte que se ve de lujo. En la sección de ejemplos haremos una tabla completa.

Puedes combinar banderas: %-8.2f significa ancho 8, 2 decimales, alineado a la izquierda. El orden es: % [banderas: - o 0] [ancho] [.precisión] letra. No tienes que memorizar todas las combinaciones hoy; con %d, %.2f, %5d y %-15s ya puedes formatear el 95 por ciento de lo que necesites.

Ahora las secuencias de escape: son combinaciones que empiezan con la diagonal invertida \ y representan caracteres especiales dentro del texto. Las más importantes:

- \n: salto de línea. Mueve el cursor al inicio de la siguiente línea. Sin \n, todo lo que imprimas sale pegado en la misma línea.
- \t: tabulación. Avanza hasta la siguiente parada de tabulador, útil para separar columnas rápidamente.
- \\: imprime una diagonal invertida literal, porque una sola \ siempre inicia una secuencia de escape.
- \": imprime comillas dobles dentro de un texto delimitado por comillas dobles.
- \': imprime comillas simples.
- \r: retorno de carro, vuelve al inicio de la línea actual (se usa en barras de progreso).
- \b: retroceso, borra el carácter anterior (efectos sencillos en consola).
- \0: el carácter nulo, marca el fin de las cadenas en C. Lo verás en la clase 21.

Ejemplo:

```
printf("Nombre:\tJuan\nEdad:\t25\n");
```

Muestra: Nombre: Juan Edad: 25

(La \t alinea los valores en columnas según las paradas de tabulador.)

Un error común es olvidar el `\n` al final y luego sorprenderse de que el siguiente `printf` salga pegado, o de que el prompt del sistema aparezca en la misma línea del resultado. Acostumbrate a terminar tus líneas con `\n` salvo que tengas una razón para no hacerlo.

Otro punto fino: `printf` devuelve un valor, la cantidad de caracteres que imprimió. Casi nadie lo usa, pero existe. Y `printf` por sí sola no lee nada del teclado: para eso está `scanf`, que es la protagonista de la clase 7. No las confundas: `printf` saca datos a la pantalla, `scanf` mete datos desde el teclado.

Para cerrar esta sección, un consejo de oficio: cuando tu salida se vea mal, el culpable casi siempre es uno de estos tres: olvidaste el `\n`, el especificador no corresponde al tipo, o la precisión no es la que querías. Revisa esos tres y el 90 por ciento de los problemas de formato se resuelven solos. `printf` es como un traje a la medida: al principio cuesta, pero cuando le agarras la mano, tus programas se ven profesionales.

Errores comunes (y como evitarlos)

En esta sección vamos a ver los errores que cometen casi todos los principiantes con variables, tipos y `printf`. Te los presento aquí, con su síntoma y su cura, para que cuando te topes con uno no pierdas horas: lo reconozcas al instante.

Error 1: usar una variable sin inicializar. Declaras `int total`; y luego haces `printf("%d", total)`; esperando ver un cero. Pero C no inicializa las variables automáticamente: `total` contiene basura, y verás un número absurdo como 32767 o -872348. Peor aun: cada vez que corres el programa puede salir un número distinto, porque la basura cambia. El síntoma es un valor loco e impredecible. La cura es simple: inicializa siempre al declarar. `int total = 0`; cuesta dos segundos y te ahorra un dolor de cabeza. Hay programadores con años de experiencia que siguen cayendo en este error cuando van rápido, así que no te sientas mal: solo adopta el hábito.

Error 2: especificador equivocado en `printf`. Imprimes un float con `%d` o un int con `%f` y obtienes basura. Por ejemplo:

```
float pi = 3.14159; printf("%d\n", pi);
```


En pantalla no verás 3 ni 3.14159: verás un número entero gigantesco y sin sentido, porque printf interpreto los bits del float como si fueran un int. El sintoma es un valor absurdo justo donde esperabas algo razonable. La cura: verifica que cada % corresponda al tipo de su variable. %d para int, %f para float y double, %c para char, %s para cadenas. Y presta atención a los warnings del compilador: gcc muchas veces te avisa cuando el formato no cuadra. Compila siempre con -Wall para activar esas advertencias; es como tener un asistente que revisa tu trabajo.

Error 3: overflow, o desbordamiento. Guardas en un int un número más grande de lo que cabe, como 3,000,000,000, y el valor "da la vuelta" y se convierte en un número negativo. El sintoma es un resultado negativo donde esperabas un positivo grande, o un valor que no tiene nada que ver. La cura es elegir el tipo con el rango adecuado: si vas a manejar cantidades enormes, usa long o unsigned long. Y ojo con las operaciones intermedias: al multiplicar dos int grandes, el resultado puede desbordarse aunque la variable destino sea long, porque la multiplicación se hace en int primero. Es un error sutil que hasta los intermedios cometen.

Error 4: división entera accidental. Escribes float promedio = 7 / 2; esperando 3.5, pero obtienes 3.0. ¿Por qué? Porque 7 y 2 son enteros, la división se hace en enteros (da 3, truncando los decimales) y después ese 3 se convierte a float (3.0). El tipo del resultado lo deciden los operandos, no la variable destino. El sintoma es un promedio o un porcentaje que siempre sale "cerrado", sin decimales. La cura: asegurate de que al menos un operando sea decimal: 7.0 / 2, o (float)7 / 2. Este error aparece en casi todos los programas de principiantes que calculan promedios, así que aprendelo bien desde hoy. Lo veremos a fondo en la clase 8 con los operadores.

Error 5: confundir = con ==. Escribes if (edad = 18) creyendo que comparas, pero en realidad asignas 18 a edad. Como una asignación vale lo asignado (18, distinto de cero, o sea "verdadero"), el if siempre se cumple. El sintoma es una condición que siempre es verdadera o un valor que cambia misteriosamente. La cura: en comparaciones usa ==, y si tu compilador te avisa con -Wall, hazle caso. Algunos programadores escriben las comparaciones al revés, if (18 == edad), para que si se les va un = el compilador marque error; es un truco válido, aunque con la práctica dejarás de necesitarlo.

Error 6: comillas simples vs comillas dobles. Escribes char letra = "A"; con comillas dobles y el compilador se queja, o peor, el programa se comporta raro. Las comillas dobles crean una cadena (un arreglo de caracteres terminado en \0), no un carácter. El sintoma es un warning o error del compilador hablando de apuntadores o tipos incompatibles. La cura: un carácter, comillas simples ('A'); texto, comillas dobles ("Hola"). Grabatelo como mantra.

Error 7: olvidar el & en scanf... perdon, eso es de la clase 7. Pero te lo adelanto para que no te asuste cuando llegues: scanf necesita la dirección de la variable, no su valor. Si lo olvidas, el programa puede crashear. Lo veremos con detalle en la siguiente clase.

Error 8: punto y coma después de #define. Escribes #define PI 3.14159; y luego radio * PI se convierte en radio * 3.14159;, con un punto y coma en medio de la expresión. El compilador te dará un error que parece no tener sentido, apuntando a una línea que se ve bien. El sintoma es un error de sintaxis en una línea inocente. La cura: los #define no llevan punto y coma. Nunca.

Error 9: no cerrar las comillas o los paréntesis en printf. printf("Hola); sin la comilla de cierre genera un error que a veces apunta a líneas posteriores, confundiendo. La cura es revisar la línea marcada y las anteriores: el error real suele estar un poco antes de donde el compilador dice.

Error 10: usar %s con un char o %c con una cadena. Son primos del error 2 pero merecen mención aparte porque el sintoma es espectacular: con %s y un char, printf intentara leer memoria como si fuera texto hasta encontrar un \0, e imprimira basura o crasheara. La cura es la misma: el molde debe corresponder al dato.

La moraleja de esta sección: casi todos los errores de esta unidad vienen de tres descuidos: no inicializar, no respetar los tipos y no cerrar lo que abres. Si desarrollas el habito de revisar esos tres puntos, programaras con mucha menos frustración.

Buenas prácticas

Las buenas prácticas son habitos que separan el código que funciona del código que funciona y además se entiende, se mantiene y no te averguenza dentro de un año. Aquí van las esenciales de esta unidad.

Primera: nombres descriptivos. Una variable debe decir que guarda: edad, precio, calificación, total, nombre. Evita nombres de una letra salvo para contadores de ciclos (i, j, k es una tradición aceptada) o coordenadas (x, y). Evita también nombres genericos como dato, variable1, cosa, temp (a menos que de verdad sea temporal y de vida corta). Cuando leas tu código dentro de seis meses, los nombres son tu documentación gratis.

Segunda: inicializa siempre. Ya lo dijimos en errores comunes, pero vale como práctica positiva: cada variable nace con un valor. int contador = 0; float suma = 0.0; char respuesta = '\n';. Así eliminas la basura de memoria de tu lista de sospechosos cuando algo falle.

Tercera: declara las variables cerca de donde las usas y en el orden lógico del programa. No amontones veinte declaraciones al inicio de la función "por si acaso". Una variable declarada

justo antes de usarse es más fácil de seguir y de depurar. (En C moderno puedes declarar variables en cualquier parte del bloque, no solo al inicio de la función como exigía el C antiguo.)

Cuarta: una variable, un propósito. No reutilices la misma variable para dos cosas distintas en el mismo programa ("aquí guardo la edad y más abajo la reutilizo para el total"). Eso ahorra una línea y crea diez minutos de confusión. Las variables son baratas; la claridad no.

Quinta: usa const para todo valor que no deba cambiar. No solo PI e IVA: también límites, tamaños, configuraciones. Si un valor es fijo por diseño, blindalo con const y deja que el compilador te cuide. Y cuando un número aparezca más de una vez o tenga significado, dale nombre: adios números mágicos.

Sexta: se consistente con el formato de printf. Decide cuántas líneas de encabezado llevan tus reportes, cuántos decimales muestran tus precios, como alineas tus tablas, y mantenlo en todo el programa. Un programa que muestra precios a veces con 2 decimales y a veces con 6 se ve descuidado y genera desconfianza.

Septima: compila con warnings activados. gcc -Wall -o programa programa.c debe ser tu comando de compilación de todos los días. Los warnings son el compilador diciendote "esto compila, pero huele mal". Muchos errores de tipos y de formato los detecta antes de que corras el programa. Ignorar los warnings es como ignorar el ruido raro del motor: tarde o temprano pagas la cuenta.

Octava: comenta lo no obvio. No comentarios `int edad = 25; // guarda la edad porque es redundante`. Comenta el porqué, no el que: `// El sensor entrega la temperatura en decimas de grado, por eso se divide entre 10`. Un buen comentario explica decisiones, no repite código.

Novena: prueba con valores extremos. Cuando tu programa calcule algo, pruebalo con cero, con números negativos, con números enormes y con el valor máximo permitido. Esos casos borde son donde se esconden los overflows y las divisiones entre cero. Cinco minutos de pruebas te ahorran horas de depuración.

Decima: escribe código para humanos. Tu programa lo va a leer la computadora una vez para compilarlo, pero lo van a leer humanos muchas veces: tu yo del futuro, tus compañeros, tu profesor. El código claro con buenos nombres, buena indentación y formato consistente no es vanidad: es profesionalismo. En lucasegaacademy.com insistimos mucho en esto porque es lo que distingue a un programador que entrega de uno que solo "hace que funcione".

Adopta estas prácticas desde esta unidad, cuando tus programas son pequeños y es fácil. Cuando llegues a programas de cientos de líneas con apuntadores y memoria dinámica, estos hábitos serán tu red de seguridad.

Ejemplos paso a paso

En esta sección cada ejemplo es un programa completo: puedes copiarlo, compilarlo con gcc, correrlo y modificarlo. Te recomiendo que después de que funcione, cambies valores, rompas cosas a propósito y observes que pasa. Así se aprende de verdad.

Ejemplo 1: Declarar e imprimir variables de cada tipo

Empecemos con lo básico: declarar una variable de cada tipo, inicializarla y mostrarla con el especificador correcto. Fíjate como cada % corresponde exactamente al tipo de su variable.

```
#include <stdio.h>

int main() {
    int edad = 25;
    float precio = 19.99;
    double pi = 3.141592653589793;
    char inicial = 'J';

    printf("Edad: %d\n", edad);
    printf("Precio: %.2f\n", precio);
    printf("Pi: %.5f\n", pi);
    printf("Inicial: %c\n", inicial);

    return 0;
}
```

Salida: Edad: 25 Precio: 19.99 Pi: 3.14159 Inicial: J

Que está pasando aquí? Declaramos cuatro variables, una de cada tipo básico, todas inicializadas al momento de declararlas (buena práctica número dos). Luego imprimimos cada una con su molde: %d para el int, %.2f para el float (con 2 decimales, ideal para precios), %.5f para el double (con 5 decimales para aprovechar su precisión) e %c para el char. Observa que el mismo número pi podría mostrarse con %.2f, %.10f o %f según lo que necesites: el valor guardado no cambia, solo cambia cómo se muestra. Esa es la magia de printf: separa el dato de su presentación. Experimento sugerido: cambia %.2f por %f en el precio y mira como aparecen los seis decimales por defecto. Luego cambia %d por %f en la edad y observa la basura que sale: así se ve el error 2 de la sección anterior.

Ejemplo 2: El char es un número (jugando con ASCII)

Este ejemplo demuestra que un char guarda un número y que puedes hacer aritmética con el. También muestra como el mismo valor se ve distinto según lo imprimas con %c o con %d.

```
#include <stdio.h>

int main() {
    char letra = 'A';

    printf("Como caracter: %c\n", letra);
    printf("Como numero: %d\n", letra);

    letra = letra + 1;
    printf("Despues de sumar 1: %c\n", letra);

    char minuscula = 'a';
    char mayuscula = minuscula - 32;
    printf("'%' en mayuscula es '%c'\n", minuscula, mayuscula);

    return 0;
}
```

Salida: Como carácter: A Como número: 65 Después de sumar 1: B 'a' en mayúscula es 'A'

La variable letra guarda el número 65, que en ASCII es la 'A'. Con %c la vemos como letra; con %d la vemos como número. Al sumarle 1 pasamos al 66, que es la 'B': así de fácil se recorre el abecedario. El truco de minúscula - 32 funciona porque en ASCII las minúsculas están exactamente 32 posiciones después de sus mayúsculas ('a' es 97, 'A' es 65). Este tipo de manipulación es la base de muchos ejercicios clásicos: cifrados, validaciones y conversiones de texto. Experimento sugerido: imprime los caracteres del 48 al 57 con un ciclo... todavía no vemos ciclos, así que hazlo a mano con varias líneas y descubre que símbolos son. Pista: son los dígitos.

Ejemplo 3: Constantes con const y #define

Un programa que calcula el área de un círculo y el precio con IVA, usando constantes con nombre en lugar de números mágicos. Compara la legibilidad con la versión que usaría 3.14159 y 0.16 regados por el código.

```

#include <stdio.h>

#define PI 3.14159

int main() {
    const float IVA = 0.16;
    float radio = 5.0;
    float precio_sin_iva = 100.0;

    float area = PI * radio * radio;
    float precio_con_iva = precio_sin_iva * (1 + IVA);

    printf("Radio: %.1f\n", radio);
    printf("Area del circulo: %.2f\n", area);
    printf("Precio sin IVA: $%.2f\n", precio_sin_iva);
    printf("Precio con IVA: $%.2f\n", precio_con_iva);

    return 0;
}

```

Salida: Radio: 5.0 Área del círculo: 78.54 Precio sin IVA: \$100.00 Precio con IVA: \$116.00

Usamos `#define` para PI (un valor matematico universal, estilo clásico) y `const` para IVA (un valor de negocio con tipo float). Ambas se escriben en mayúsculas por convención. El programa se lee solo: $\text{área} = \text{PI} * \text{radio} * \text{radio}$ no necesita comentarios. Si mañana el IVA cambia a 0.18, cambias una sola línea y todo el programa se actualiza: esa es la ventaja de no regar números mágicos. Nota el detalle de $(1 + \text{IVA})$: en lugar de escribir 1.16, expresamos la idea "el precio más el impuesto". Experimento sugerido: intenta agregar la línea `IVA = 0.20;` después del cálculo y mira como el compilador te detiene. Así se siente la protección de `const`.

Ejemplo 4: Tabla formateada con printf (ticket de compra)

Aquí `printf` muestra su poder real: columnas alineadas, precios con 2 decimales y un total. Es el mismo formato que usan los tickets y reportes de los programas profesionales.

```
#include <stdio.h>

int main() {
    printf("%-15s %8s %10s\n", "Producto", "Cantidad", "Precio");
    printf("-----\n");
    printf("%-15s %8d %10.2f\n", "Manzanas", 10, 25.50);
    printf("%-15s %8d %10.2f\n", "Pan", 3, 45.00);
    printf("%-15s %8d %10.2f\n", "Leche", 2, 38.75);
    printf("-----\n");

    float total = 10 * 25.50 + 3 * 45.00 + 2 * 38.75;
    printf("%-15s %8s %10.2f\n", "TOTAL", "", total);

    return 0;
}
```

Salida: Producto	Cantidad	Precio	-----	Manzanas	10
25.50 Pan	3	45.00	Leche	2	38.75
TOTAL	467.50				

El truco está en los anchos de campo: %-15s deja el nombre alineado a la izquierda en 15 espacios, %8d alinea la cantidad a la derecha en 8, y %10.2f muestra el precio con 2 decimales en 10 espacios. Como todas las filas usan los mismos anchos, las columnas quedan perfectamente alineadas. La línea de guiones es un separador hecho a mano: cuenta los caracteres para que mida lo mismo que la tabla (15 + 1 + 8 + 1 + 10 = 35). Fíjate que en el encabezado usamos %8s y %10s con texto en lugar de números: los anchos funcionan igual con cadenas. Experimento sugerido: cambia %-15s por %15s y mira como los nombres se alinean a la derecha y la tabla se ve distinta. Luego agrega un cuarto producto con tu precio favorito.

Ejemplo 5: Calculadora de promedio simple

El programa clásico de la escuela: promedia tres calificaciones. Además de practicar variables y printf, te enseña la trampa de la división entera y como evitarla.

```
#include <stdio.h>

int main() {
    float calif1 = 8.5;
    float calif2 = 9.0;
    float calif3 = 7.5;

    float suma = calif1 + calif2 + calif3;
    float promedio = suma / 3;

    printf("Calificaciones: %.1f, %.1f, %.1f\n", calif1, calif2, calif3);
    printf("Suma: %.1f\n", suma);
    printf("Promedio: %.2f\n", promedio);

    return 0;
}
```

Salida: Calificaciones: 8.5, 9.0, 7.5 Suma: 25.0 Promedio: 8.33

Usamos float para las calificaciones porque tienen decimales. La división suma / 3 funciona bien porque suma es float: cuando un operando es decimal, toda la operación es decimal. Pero ojo: si las calificaciones fueran int, la división sería entera y perderías los decimales (el error 4 de la sección anterior). Con %.1f mostramos las calificaciones con un decimal y con %.2f el promedio con dos, que es lo usual en las boletas. Experimento sugerido: cambia los tipos a int con valores 8, 9 y 7, calcula el promedio y observa que sale 8.00 en lugar de 8.00... en realidad sale 8.00? No: sale 8.00 truncado de 8.0? Haz la prueba y entiende por que: $24 / 3 = 8$ exacto es mal ejemplo; usa 8, 9, 8 para que veas el truncamiento de 8.33 a 8.00. Luego arreglalo dividiendo entre 3.0.

Ejemplo 6: Secuencias de escape en acción

Un solo programa que demuestra \n, \t, \\ y \" trabajando juntas. Es corto, pero cada línea te ensena algo.


```
#include <stdio.h>

int main() {
    printf("Línea 1\nLínea 2\nLínea 3\n");
    printf("Nombre:\tJuan\nEdad:\t25\n");
    printf("La ruta es C:\\documentos\\tarea\n");
    printf("Ella dijo: \"Hola, mundo\"\n");
    printf("Progreso: 50%% completado\n");

    return 0;
}
```

Salida: Línea 1 Línea 2 Línea 3 Nombre: Juan Edad: 25 La ruta es C:\documentos\tarea Ella dijo: "Hola, mundo" Progreso: 50% completado

Cada `\n` baja el cursor una línea. La `\t` salta a la siguiente parada de tabulador, alineando "Juan" y "25" en una columna improvisada. Para mostrar una diagonal invertida real necesitas `\\`, porque una sola `\` siempre empieza una secuencia: fíjate que en la ruta se ve una sola diagonal aunque escribimos dos. Las comillas dobles dentro del texto necesitan `\"` para que el compilador no crea que ahí termina la cadena. Y para mostrar un signo de porcentaje literal se usa `%%`, porque un `%` solo siempre inicia un especificador. Experimento sugerido: quita una de las diagonales en la ruta y mira que desastre sale (la `\t` se interpreta como tabulación). Es la mejor forma de entender por qué existe `\\`.

Ejemplo 7: Especificadores `%x`, `%o` y `%u` (otras bases)

Los programadores también cuentan en hexadecimal y octal, sobre todo cuando trabajan con hardware, colores o permisos de archivos. Este ejemplo muestra el mismo número en varias bases.

```
#include <stdio.h>

int main() {
    int numero = 255;

    printf("Decimal: %d\n", numero);
    printf("Hexadecimal: %x\n", numero);
    printf("Hexadecimal (mayusculas): %X\n", numero);
    printf("Octal: %o\n", numero);
    printf("Sin signo: %u\n", numero);

    int negativo = -1;
    printf("-1 como sin signo: %u\n", negativo);

    return 0;
}
```

Salida: Decimal: 255 Hexadecimal: ff Hexadecimal (mayúsculas): FF Octal: 377 Sin signo: 255
-1 como sin signo: 4294967295

El 255 en hexadecimal es ff y en octal es 377: mismo valor, distinta representación. %X lo muestra en mayúsculas (FF). Lo más instructivo es la última línea: -1 interpretado como unsigned da 4294967295, que es el valor máximo de un entero sin signo de 32 bits. Esto pasa por la representación interna de los negativos (complemento a dos) y es otro ejemplo perfecto de la idea central de la unidad: los bits son los mismos, el tipo decide cómo se leen. No necesitas dominar el complemento a dos hoy, pero ya viste con tus ojos como el tipo cambia el significado. Experimento sugerido: prueba con otros valores como 16, 1000 y -255, y verifica los resultados con la calculadora de tu sistema en modo programador.

Ejemplo 8: Ficha personal completa (integrando todo)

El ejemplo final integra variables, constantes, varios tipos, precisión, ancho de campo y secuencias de escape en un programa que genera una ficha ordenada. Es el tipo de programa que ya podrías presumir.

```
#include <stdio.h>

#define ANO_ACTUAL 2026

int main() {
    const char* nombre = "Estephan";
    char inicial = 'E';
    int ano_nacimiento = 1990;
    float estatura = 1.75;
    double peso = 82.5;

    int edad = ANO_ACTUAL - ano_nacimiento;

    printf("=====\n");
    printf("      FICHA PERSONAL\n");
    printf("=====\n");
    printf("Nombre completo: %s\n", nombre);
    printf("Inicial: %c\n", inicial);
    printf("Edad: %d anos\n", edad);
    printf("Estatura: %.2f m\n", estatura);
    printf("Peso: %.1f kg\n", peso);
    printf("=====\n");

    return 0;
}
```

```
Salida:          ===== FICHA          PERSONAL
===== Nombre completo: Esteban Inicial: E Edad: 36 años
Estatura: 1.75 m Peso: 82.5 kg =====
```

Aquí hay de todo: un `#define` para el año actual, variables de cuatro tipos distintos, un cálculo (la edad), precisión controlada (`%.2f` para la estatura, `%.1f` para el peso) y un marco decorativo hecho con signos de igual. Nota `const char*` nombre: es un apuntador a una cadena de texto, algo que entenderás a fondo en las clases 21 y 22; por ahora usalo como receta para guardar nombres. El `%s` imprime la cadena completa. Cambia los datos por los tuyos, recompila y tendrás tu primera ficha personal generada por un programa escrito por ti. Pequeño? Si. Tuyo? Completamente. Y todo programa grande empezó siendo uno pequeño como este.

Práctica

Llegó la hora de ensuciarte las manos. Los ejercicios están ordenados de más fácil a más retador. Intenta resolver cada uno por tu cuenta antes de mirar la solución: la lucha es parte del aprendizaje. Si te atorras más de 20 minutos, mira la solución, entiendela y luego escríbela tú mismo sin copiar.

1. Ficha de estudiante. Declara variables para guardar tu nombre (cadena), tu edad (int), tu promedio actual (float) y la letra de tu grupo (char). Imprime todo con printf en un formato ordenado, con etiquetas claras. El promedio debe mostrarse con 2 decimales.
2. Rectángulo con constantes. Escribe un programa que calcule el área y el perímetro de un rectángulo. Guarda la base y la altura en variables, y usa una constante con nombre para... bueno, aquí no hay un valor físico fijo, así que el reto es: define con #define el número de rectángulos que vas a procesar (por ejemplo 3) e imprime los cálculos para rectángulos de distintos tamaños usando ese dato en un mensaje. El objetivo es practicar const y #define con sentido.
3. Conversor de temperatura. Escribe un programa que convierta grados Celsius a Fahrenheit. La fórmula es $F = C * 9 / 5 + 32$. Guarda los grados Celsius en una variable, calcula los Fahrenheit y muestra ambos con 1 decimal. Prueba con 0, 25 y 100 grados. Cuidado con la división entera.
4. Detective ASCII. Escribe un programa que declare un char con la letra 'm', imprima su código ASCII con %d, luego le sume 3 e imprima el carácter resultante con %c. Además, toma la letra 'P', conviértela a minúscula restando la diferencia que corresponda e imprime el resultado. Explica en un comentario por qué funciona.
5. Ticket de la tiendita. Adapta el Ejemplo 4 para una compra real: 3 productos con nombre, cantidad y precio unitario a tu elección. Calcula el subtotal de cada producto (cantidad * precio), el total general, y aplícale el IVA del 16% usando una constante. Imprime todo en una tabla alineada con 2 decimales, incluyendo la línea del IVA y el total a pagar.
6. Promedio con veredicto. Escribe un programa que guarde 4 calificaciones (float), calcule el promedio con 2 decimales y muestre un mensaje según el resultado: "Aprobado" si el promedio es mayor o igual a 6, "Reprobado" en caso contrario. Pista: necesitarás el if de la Unidad 3, pero puedes intentarlo leyendo la condición como una pregunta al programa; si prefieres, haz primero solo el cálculo y el mensaje fijo, y agrega la decisión después de estudiar la clase 10.

Solución del ejercicio 1

```
#include <stdio.h>

int main() {
    const char* nombre = "Ana Garcia";
    int edad = 20;
    float promedio = 8.756;
    char grupo = 'B';

    printf("----- FICHA DE ESTUDIANTE -----\\n");
    printf("Nombre: %s\\n", nombre);
    printf("Edad: %d anos\\n", edad);
    printf("Promedio: %.2f\\n", promedio);
    printf("Grupo: %c\\n", grupo);

    return 0;
}
```

Explicación: cada dato usa su tipo natural: la cadena con %s, el entero con %d, el decimal con %.2f (que redondea 8.756 a 8.76) y el carácter con %c. El programa es directo, pero fíjate en los detalles profesionales: etiquetas claras antes de cada valor, el promedio con precisión controlada y el nombre guardado en una constante de tipo cadena porque no va a cambiar durante el programa. Si tu solución uso puts o varios printf para lo mismo, también es válida: en programación casi siempre hay más de un camino correcto.

Solución del ejercicio 2

```

#include <stdio.h>

#define CANTIDAD_RECTANGULOS 3

int main() {
    const float PI_EXTRA = 3.14159;

    float base1 = 5.0, altura1 = 3.0;
    float base2 = 10.0, altura2 = 4.5;
    float base3 = 2.5, altura3 = 2.5;

    printf("Se procesaran %d rectangulos\n\n", CANTIDAD_RECTANGULOS);

    printf("Rectangulo 1: base %.1f, altura %.1f\n", base1, altura1);
    printf("  Area: %.2f, Perimetro: %.2f\n\n",
        base1 * altura1, 2 * (base1 + altura1));

    printf("Rectangulo 2: base %.1f, altura %.1f\n", base2, altura2);
    printf("  Area: %.2f, Perimetro: %.2f\n\n",
        base2 * altura2, 2 * (base2 + altura2));

    printf("Rectangulo 3: base %.1f, altura %.1f\n", base3, altura3);
    printf("  Area: %.2f, Perimetro: %.2f\n",
        base3 * altura3, 2 * (base3 + altura3));

    (void)PI_EXTRA;

    return 0;
}

```

Explicación: el área es $\text{base} * \text{altura}$ y el perímetro es $2 * (\text{base} + \text{altura})$; los paréntesis son obligatorios porque sin ellos la multiplicación se haría antes que la suma. El `#define CANTIDAD_RECTANGULOS` se usa en el mensaje inicial: si mañana procesas 10 rectangulos, cambias un solo número. La constante `PI_EXTRA` está ahí como ejemplo de `const` declarada correctamente, y la línea `(void)PI_EXTRA` es un truco para decirle al compilador "sé que no la uso, no me avises": en código real simplemente no declararías lo que no usas. Lo importante del ejercicio era practicar las dos formas de constante y ver que el código con fórmulas directas dentro del `printf` también es válido, aunque para programas grandes es mejor calcular primero en variables con nombre.

Solución del ejercicio 3

```
#include <stdio.h>

int main() {
    float celsius = 25.0;
    float fahrenheit = celsius * 9 / 5 + 32;

    printf("%.1f grados C son %.1f grados F\n", celsius, fahrenheit);

    celsius = 0.0;
    fahrenheit = celsius * 9 / 5 + 32;
    printf("%.1f grados C son %.1f grados F\n", celsius, fahrenheit);

    celsius = 100.0;
    fahrenheit = celsius * 9 / 5 + 32;
    printf("%.1f grados C son %.1f grados F\n", celsius, fahrenheit);

    return 0;
}
```

Salida: 25.0 grados C son 77.0 grados F 0.0 grados C son 32.0 grados F 100.0 grados C son 212.0 grados F

Explicación: la fórmula $F = C * 9 / 5 + 32$ funciona perfecto aquí porque celsius es float: toda la operación se hace en decimales. La trampa mortal sería escribir $9 / 5$ con enteros, que da 1 en lugar de 1.8; pero como $celsius * 9$ se evalúa primero y celsius es float, el 9 se convierte a float automáticamente y la división sale bien. Este es el error 4 de la sección de errores comunes, y este ejercicio existe para que lo vivas: prueba cambiar celsius a int y mira como 25 grados te dan un resultado mal. Con %.1f mostramos un decimal, suficiente para temperaturas. Cuando aprendas scanf en la clase 7, podrás pedirle los grados al usuario en lugar de escribirlos en el código.

Solución del ejercicio 4

```
#include <stdio.h>

int main() {
    char letra = 'm';

    printf("La letra '%c' tiene codigo ASCII %d\n", letra, letra);

    letra = letra + 3;
    printf("Tres posiciones despues: '%c'\n", letra);

    char mayus = 'P';
    char minus = mayus + 32;
    printf("'%c' en minuscula es '%c'\n", mayus, minus);

    return 0;
}
```

Salida: La letra 'm' tiene código ASCII 109 Tres posiciones después: 'p' 'P' en minúscula es 'p'

Explicación: la 'm' es el código 109 en ASCII; al sumarle 3 llegamos al 112, que es la 'p'. Funciona porque las letras están en orden consecutivo en la tabla, así que sumar y restar se traduce en avanzar y retroceder en el abecedario. La conversión de mayúscula a minúscula suma 32 porque en ASCII cada mayúscula está exactamente 32 posiciones antes que su minúscula ('P' es 80, 'p' es 112). Fíjate en el primer printf: usamos la misma variable dos veces con dos moldes distintos, %c y %d, y obtenemos dos lecturas del mismo dato. Ese es el concepto del "lente del tipo" hecho programa. Como reto extra: escribe la operación inversa, de minúscula a mayúscula restando 32, y verifica que 'p' - 32 te devuelva la 'P'.

Solución del ejercicio 5


```
#include <stdio.h>

int main() {
    const float IVA = 0.16;

    float precio1 = 25.50, precio2 = 45.00, precio3 = 38.75;
    int cant1 = 10, cant2 = 3, cant3 = 2;

    float sub1 = cant1 * precio1;
    float sub2 = cant2 * precio2;
    float sub3 = cant3 * precio3;
    float total = sub1 + sub2 + sub3;
    float impuesto = total * IVA;
    float total_pagar = total + impuesto;

    printf("%-12s %8s %10s %10s\n", "Producto", "Cantidad", "P. unit", "Subtotal");
    printf("-----\n");
    printf("%-12s %8d %10.2f %10.2f\n", "Manzanas", cant1, precio1, sub1);
    printf("%-12s %8d %10.2f %10.2f\n", "Pan", cant2, precio2, sub2);
    printf("%-12s %8d %10.2f %10.2f\n", "Leche", cant3, precio3, sub3);
    printf("-----\n");
    printf("%-12s %8s %10s %10.2f\n", "TOTAL", "", "", total);
    printf("%-12s %8s %10s %10.2f\n", "IVA (16%)", "", "", impuesto);
    printf("%-12s %8s %10s %10.2f\n", "A PAGAR", "", "", total_pagar);

    return 0;
}
```

```
Salida: Producto  Cantidad  P. unit  Subtotal ----- Manzanas
      10    25.50   255.00 Pan           3    45.00   135.00 Leche           2    38.75
77.50 ----- TOTAL                               467.50 IVA (16%)
              74.80 A PAGAR                               542.30
```

Explicación: este es el Ejemplo 4 llevado a la vida real. Cada subtotal se calcula en su propia variable con nombre descriptivo, lo que hace el código fácil de revisar: si un subtotal sale mal, sabes exactamente que línea mirar. El IVA va en una constante con nombre, así que si la tasa cambia solo tocas una línea. La tabla usa cuatro columnas con anchos fijos (%-12s, %8s, %10s, %10s) y las filas de totales reutilizan esos mismos anchos con cadenas vacías "" para mantener la alineación. Todos los montos usan %.2f porque es dinero. Verifica la aritmética: $467.50 * 0.16 = 74.80$, y $467.50 + 74.80 = 542.30$. Si tus números coinciden, tu programa calcula como una caja registradora de verdad.

Solución del ejercicio 6

```
#include <stdio.h>

int main() {
    float c1 = 8.0, c2 = 7.5, c3 = 9.0, c4 = 6.5;
    float promedio = (c1 + c2 + c3 + c4) / 4;

    printf("Calificaciones: %.1f, %.1f, %.1f, %.1f\n", c1, c2, c3, c4);
    printf("Promedio: %.2f\n", promedio);

    if (promedio >= 6.0) {
        printf("Resultado: Aprobado\n");
    } else {
        printf("Resultado: Reprobado\n");
    }

    return 0;
}
```

Salida: Calificaciones: 8.0, 7.5, 9.0, 6.5 Promedio: 7.75 Resultado: Aprobado

Explicación: el promedio de cuatro calificaciones se calcula sumando y dividiendo entre 4.0 (aquí el 4 entero funciona porque la suma es float, pero escribir 4.0 es más explícito y evita la división entera si algún día cambias los tipos). Con %.2f el promedio se ve como en la boleta. El if con su else es un adelanto de la clase 10: le pregunta al programa "el promedio es mayor o igual a 6?" y ejecuta un mensaje u otro. Si todavía no has estudiado la Unidad 3, no te preocupes por entender cada símbolo del if: quedate con la idea de que el programa puede tomar decisiones con los datos que calculaste. Prueba cambiar las calificaciones para que el promedio baje de 6 y mira como el veredicto cambia solo. Ese momento, cuando el programa "decide", es cuando muchos se enamoran de programar.

Resumen de la unidad

Llegaste al final de la Unidad 2. Hagamos un repaso de todo lo que ahora sabes, porque es mucho más de lo que parece.

Aprendiste que una variable es un espacio de memoria con nombre: una caja etiquetada en el almacén de la memoria. Conoces sus tres momentos: la declaración reserva el espacio, la inicialización le da su primer valor y la asignación lo cambia después. Y grabaste la advertencia más importante: una variable no inicializada contiene basura, así que inicializa siempre al declarar.

Conociste los cuatro tipos básicos de C y cuando usar cada uno: `int` para enteros (4 bytes, hasta más de dos mil millones), `float` para decimales de precisión moderada (4 bytes), `double` para decimales de alta precisión (8 bytes) y `char` para un solo carácter (1 byte). Entendiste que los tamaños pueden variar según la plataforma y que con `sizeof` puedes preguntarle a tu propia computadora. Aprendiste la regla de oro para elegir tipo: según el dato, no según tu antojo.

Descubriste el secreto del `char`: en realidad guarda un número, y la tabla ASCII traduce esos números a símbolos. Viste que 'A' es 65, que los dígitos van del 48 al 57, que las minúsculas están 32 posiciones después de las mayúsculas, y que puedes hacer aritmética con caracteres para recorrer el abecedario y convertir mayúsculas en minúsculas. Y entendiste la idea más profunda de la unidad: los bits no significan nada por sí solos; el tipo es el lente que les da significado.

Aprendiste a crear constantes con `const` (una variable de solo lectura, con tipo y con alcance, tu opción por defecto) y con `#define` (sustitución de texto del preprocesador, útil para tamaños de arreglos y código heredado). Sabes que ambas se escriben en MAYÚSCULAS por convención, que `#define` no lleva punto y coma, y que a los números mágicos hay que ponerles nombre.

Dominaste `printf` como herramienta de formato profesional: los especificadores `%d`, `%f`, `%c`, `%s`, `%x`, `%o` y sus compañeros, el control de precisión con `%.2f`, el ancho de campo con `%8d` para alinear columnas, las banderas `-` y `0`, y las secuencias de escape `\n`, `\t`, `\\`, `\"` y `%%`. Ya puedes generar tablas, tickets y reportes que se ven como los de un programa de verdad.

Repasaste los diez errores más comunes, de la variable sin inicializar al especificador equivocado, del overflow a la división entera accidental, y adoptaste diez buenas prácticas que te van a acompañar toda tu carrera: nombres descriptivos, inicializar siempre, compilar con `-Wall`, probar con valores extremos y escribir código para humanos.

Y lo más importante: practicaste. Escribiste fichas, tickets, conversores y calculadoras. Cada programa que compiló a la primera (y cada uno que te dio un error raro que luego entendiste) te hizo un mejor programador.

Lo que sigue es la Unidad 3, donde tus programas van a aprender a leer datos del teclado con `scanf` y a tomar decisiones con `if`, `else` y `switch`. Todo lo que aprendiste aquí será la materia prima: las variables que declares, los tipos que elijas y el `printf` con el que muestres resultados. La base está puesta, y está puesta bien.

Sigue practicando, sigue rompiendo código a propósito para entenderlo, y cuando quieras más ejercicios y material de apoyo, ya sabes dónde encontrarlos: en lucasegaacademy.com tienes el curso completo esperandote. Nos vemos en la Unidad 3.

UNIDAD 3: Entrada de datos y operadores

Introducción

Hasta este punto del curso tus programas han sido un poco como un locutor de radio: hablan y hablan, muestran mensajes en la pantalla con `printf`, hacen cálculos internos y se despiden. Eso está muy bien para empezar, porque ya dominas las variables, los tipos de datos y la salida formateada. Pero seamos honestos: un programa que solo habla y nunca escucha es un programa de juguete. Los programas de verdad, los que usas todos los días, son una conversación.

Piensa en el cajero automatico: te pregunta tu NIP, te pregunta cuánto quieres retirar y te responde con billetes o con un mensaje de fondos insuficientes. Piensa en un videojuego: lee cada tecla que presionas y reacciona en fracciones de segundo. Piensa en WhatsApp: lees, escribes, envías. Hasta Google, que parece tan simple, no hace nada útil hasta que tú escribes algo en la caja de búsqueda. La diferencia entre un programa de juguete y un programa útil es la entrada de datos: la capacidad de recibir información del usuario y actuar en consecuencia.

En esta unidad le vas a poner oídos a tus programas. Vas a aprender a usar `scanf`, la función gemela de `printf`, que lee lo que el usuario escribe en el teclado y lo guarda en tus variables. Parece fácil, y lo es en la superficie, pero `scanf` tiene sus trucos y sus trampas: los especificadores de formato, el famoso símbolo `&` que no puedes olvidar, lo que pasa cuando el usuario escribe espacios, y como limpiar el bufer del teclado cuando las cosas se ensucian. Todo eso lo vas a dominar aquí, porque un programador que no controla la entrada de su programa es como un cocinero que no controla sus ingredientes: el platillo sale mal aunque la receta sea perfecta.

Pero escuchar no basta: también hay que saber que hacer con lo que se escucha. Por eso la segunda mitad de esta unidad son los operadores, los verbos del lenguaje C. Si las variables son los sustantivos (las cosas) y `printf` y `scanf` son la forma de hablar y escuchar, los operadores son las acciones: sumar, restar, multiplicar, comparar, decidir. Con ellos vas a construir la lógica de tus programas. Un operador toma uno o dos valores y produce un resultado nuevo, y esa es literalmente la forma en que una computadora piensa.

Esta unidad agrupa tres clases del curso en video de lucasegaacademy.com: la clase 7, donde aprendes `scanf` para leer datos del teclado; la clase 8, donde dominas los operadores aritmeticos y de asignación; y la clase 9, donde entran los operadores relacionales y lógicos, la precedencia y la conversión de tipos. Al terminar, vas a poder escribir programas que le pidan

datos al usuario, hagan cálculos con ellos y tomen decisiones básicas. Es el salto de "mi programa imprime cosas" a "mi programa trabaja conmigo".

Y ojo con algo importante: los operadores que vas a ver aquí no son solo de C. Casi todos existen igualitos en Java, JavaScript, Python, PHP y docenas de lenguajes más. Lo que aprendas en esta unidad te va a servir en todos los lenguajes que toques en tu vida. C te está dando, una vez más, las bases universales de la programación.

Una advertencia amistosa antes de empezar: esta unidad está llena de trampas clásicas donde caen todos los principiantes. La división entera que te roba los decimales sin avisar. El `=` que se disfraza de `==` y arruina tu comparación. El `&` olvidado en `scanf` que congela tu programa. El `++` que se comporta distinto según donde lo pongas. No te frustres cuando caigas en alguna: cada una de esas trampas te va a hacer un mejor programador, y aquí te las voy a mostrar todas de frente, con ejemplos que puedes correr en tu computadora, para que cuando te topes con ellas en tus propios programas las reconozcas al instante.

Vamos a darle oídos y verbos a tus programas. Empezamos.

Explicación detallada

3.1 scanf a fondo: como tu programa escucha al teclado

Ya conoces a `printf`, que saca información de tu programa hacia la pantalla. `scanf` es su gemelo inverso: mete información desde el teclado hacia tu programa. Su nombre viene de "scan formatted", escanear con formato, y vive en la misma biblioteca `stdio.h`, así que no necesitas incluir nada nuevo. La idea es simple: tu le dices a `scanf` que tipo de dato esperas y en que variable lo debe guardar, el programa se pausa, el usuario escribe y presiona Enter, y el valor queda guardado en tu variable listo para usarse.

Mira la forma general:

```
scanf("especificador", &variable);
```

Hay dos piezas que no puedes perder de vista. El especificador de formato es el mismo que usas en `printf`: `%d` para enteros, `%f` para flotantes, `%c` para caracteres, `%s` para cadenas de texto. La segunda pieza es el símbolo `&` antes del nombre de la variable, y este es el detalle que más programas de principiantes rompe en todo el mundo. Ese `&` significa "la dirección de memoria de" y se lo tienes que pasar a `scanf` porque `scanf` necesita saber DONDE guardar el dato, no solo cual es su valor actual. Sin el `&`, `scanf` recibe una copia del valor de la variable y

no tiene forma de modificar la original; con el &, recibe la ubicación exacta en memoria donde debe escribir. En la Unidad 8 vas a entender a fondo por qué funciona así, cuando estudies apuntadores, pero por ahora grabate la regla de oro: en scanf, las variables siempre llevan & delante. Siempre. Sin excepciones (bueno, casi: las cadenas con %s son la excepción, y ahorita te explico por qué).

Vamos con un ejemplo completo que ya puedes compilar:

```
#include <stdio.h>

int main() {
    int edad;
    printf("Cuantos años tienes? ");
    scanf("%d", &edad);
    printf("Tienes %d años. Bienvenido al curso.\n", edad);
    return 0;
}
```

Fíjate en el flujo: primero declaras la variable edad, luego le pides al usuario el dato con printf (esto se llama "prompt", el mensaje que invita a escribir), luego scanf se queda esperando, el usuario escribe 25 y presiona Enter, y finalmente usas el valor guardado. El programa se detiene en la línea de scanf hasta que el usuario presiona Enter: es un punto de pausa natural, y vas a usar ese comportamiento muchísimo.

Los especificadores más usados en scanf son estos: %d lee un entero y lo guarda en un int; %f lee un decimal y lo guarda en un float; %lf lee un decimal y lo guarda en un double (ojo: con la l de long, porque en scanf float y double NO usan el mismo especificador, a diferencia de printf donde %f sirve para los dos); %c lee un solo carácter y lo guarda en un char; %s lee una palabra, una cadena sin espacios, y la guarda en un arreglo de char. También existen %ld para long, %u para unsigned int y %x para leer números en hexadecimal, pero con los cinco primeros tienes para rato.

Ahora, las trampas de scanf, que son legendarias. Primera trampa: los espacios. Si usas %s para leer un nombre y el usuario escribe "Juan Perez", scanf solo guarda "Juan" y deja "Perez" esperando en el bufer. Eso pasa porque %s se detiene en el primer espacio en blanco. Para leer texto con espacios necesitas otras técnicas que verás en la Unidad 7 con cadenas; por ahora, ten presente la limitación: %s lee una sola palabra.

Segunda trampa: el carácter fantasma con %c. Este es el error más confuso para principiantes. Mira este programa:

```
#include <stdio.h>

int main() {
    int edad;
    char inicial;
    printf("Escribe tu edad: ");
    scanf("%d", &edad);
    printf("Escribe la inicial de tu nombre: ");
    scanf("%c", &inicial);
    printf("Edad: %d, inicial: %c\n", edad, inicial);
    return 0;
}
```

Si lo corres, vas a ver algo raro: después de escribir tu edad y presionar Enter, el programa NO te deja escribir la inicial, se la salta directo al resultado. Qué pasó? Cuando presionaste Enter después de la edad, en el teclado quedó un carácter de salto de línea, el famoso '\n', esperando en el bufer. scanf con %d lo ignora al leer el número, pero ahí se quedó. Cuando llegó el scanf con %c, que lee exactamente UN carácter cualquiera, agarro ese '\n' pendiente y siguió de largo. El programa no se trabó: simplemente leyo el Enter que habías dejado.

La solución es fácil y elegante: pon un espacio antes del %c, así: scanf(" %c", &inicial). Ese espacio le dice a scanf "saltate todos los espacios en blanco pendientes antes de leer el carácter de verdad". Grábatelo: %c casi siempre va con un espacio antes en programas reales.

Tercera trampa: el bufer sucio. El bufer del teclado es como una bandeja donde se acumula lo que el usuario escribe. Si el usuario escribe algo que scanf no esperaba, por ejemplo letras cuando pedías un número con %d, scanf falla, no guarda nada y deja esas letras atoradas en la bandeja. El siguiente scanf las vuelve a encontrar y vuelve a fallar, y tu programa entra en un ciclo raro donde parece que ignora al usuario. La forma simple de limpiar la bandeja en programas de principiante es leer y tirar los caracteres pendientes con un ciclo pequeño:

```
int c;
while ((c = getchar()) != '\n' && c != EOF);
```

Esa línea lee carácter por carácter hasta encontrar el Enter o el fin de la entrada, tirando todo a la basura. No te preocupes si aún no entiendes cada pieza: getchar y los ciclos los verás a fondo más adelante. Por ahora guardala como tu escoba: cuando un scanf se porte raro, barre el bufer con esa línea antes del siguiente scanf.

Cuarta trampa, y la más peligrosa: %s sin límite puede desbordar tu arreglo. Si declaras `char nombre[20]` y el usuario escribe una palabra de 50 letras, `scanf` las escribe todas y se pasa del espacio reservado, corrompiendo la memoria vecina. Esto se llama desbordamiento de bufer y es una de las fallas de seguridad más famosas de la historia del software. La buena práctica es limitar la lectura: `scanf("%19s", nombre)` lee máximo 19 caracteres más el terminador y cabe perfecto en un arreglo de 20. Fíjate también que con %s NO se usa &: el nombre del arreglo ya representa su dirección de memoria. En la Unidad 7 lo verás con detalle, pero la regla práctica es: %s sin &, y siempre con límite.

Un detalle más que te va a dar tranquilidad: `scanf` devuelve un valor. Devuelve cuántos datos leyo con éxito. Si pides dos números con `scanf("%d %d", &a, &b)` y el usuario escribe bien los dos, devuelve 2. Si algo falla, devuelve un número menor. Los programadores profesionales revisan ese valor para validar la entrada:

```
if (scanf("%d", &edad) != 1) {  
    printf("Eso no es un numero valido.\n");  
}
```

Aún no hemos visto `if` a fondo (es la clase 10, la siguiente unidad), pero ya vas viendo la idea: los buenos programas no confían ciegamente en lo que escribe el usuario.

Para cerrar esta sección, un resumen de reglas de oro de `scanf`: incluye `stdio.h`; usa el especificador correcto para cada tipo (%d, %f, %lf para `double`, %c, %s); pon & antes de cada variable excepto con %s; usa " %c" con espacio para leer caracteres después de números; limita %s con un número (como %19s); y si algo se comporta raro, limpia el bufer. Con eso tienes el 95 por ciento de los problemas de entrada resueltos.

3.2 Operadores aritmeticos: la calculadora dentro de tu programa

Si las variables son los sustantivos del lenguaje, los operadores aritmeticos son los verbos más basicos: sumar, restar, multiplicar, dividir y obtener el residuo. C tiene cinco: + suma, - resta, * multiplicación, / división y % módulo, que te da el residuo de una división entera. Los primeros cuatro los conoces de la primaria; el quinto, el módulo, es el nuevo amigo que vas a aprender a querer.

La suma, la resta y la multiplicación se comportan como esperas:

```
int a = 10, b = 3;
int suma = a + b;      // 13
int resta = a - b;     // 7
int multi = a * b;     // 30
```

La división es donde empieza la diversión, porque en C hay dos divisiones distintas y el compilador elige según los tipos de los operandos. Si divides dos enteros, obtienes una división ENTERA: el resultado se trunca, se corta sin redondear, y los decimales simplemente desaparecen. Si al menos uno de los dos operandos es flotante, obtienes la división real con decimales. Mira:

```
int a = 10, b = 3;
printf("%d\n", a / b);      // Imprime 3, no 3.33. Los decimales se pierden.

float x = 10.0, y = 3.0;
printf("%f\n", x / y);      // Imprime 3.333333. Division real.

printf("%f\n", a / y);      // Imprime 3.333333. Basta un flotante para que sea
real.
```

Esta es LA trampa clásica de C, la que ha hecho llorar a generaciones de estudiantes. Calculas el promedio de tres calificaciones así: $\text{promedio} = (85 + 90 + 78) / 3$, y como todo es entero, C te da 84 en lugar de 84.333. Tu programa no se queja, no marca error, simplemente te da un resultado mal redondeado hacia abajo y tu ni cuenta te das. Peor aun: si haces $(1 / 2)$ esperando 0.5, C te da 0. Cero. Así de brutal es la división entera.

La regla es simple de enunciar y fácil de olvidar: entre enteros, la división es entera y trunca. Si quieres decimales, asegurate de que al menos un operando sea flotante: $10.0 / 3$, o $(\text{float})a / b$ usando un cast, que veremos en la sección 3.8. Muchos programadores escriben 2.0 en lugar de 2 a propósito, como recordatorio visual de "aquí quiero decimales".

El operador módulo % te da el residuo de la división entera: $10 \% 3$ es 1, porque 10 entre 3 cabe 3 veces (9) y sobra 1. Solo funciona con enteros, no con flotantes. Y aunque suena poco útil, es una navaja suiza: $\text{número} \% 2$ te dice si un número es par (residuo 0) o impar (residuo 1); $\text{número} \% 10$ te da el último dígito de cualquier número; segundos $\% 60$ te da los segundos que sobran al convertir a minutos. Lo vas a usar más de lo que crees, sobre todo en ciclos y en programas que trabajan con tiempo, dígitos o patrones que se repiten.

Un par de detalles finos: dividir entre cero con enteros es un error grave que normalmente truena tu programa en ejecución; con flotantes, en cambio, C te da infinito o un valor especial, según tu sistema. Y el signo del residuo con números negativos sigue el signo del primer operando: $-10 \% 3$ da -1 . No te obsesiones con eso por ahora, solo tenlo presente.

3.3 Operadores de asignación: guardar y actualizar sin repetirte

Ya conoces el operador de asignación simple $=$: toma el valor de la derecha y lo guarda en la variable de la izquierda. `int edad = 25` guarda 25 en `edad`. Pero C te da cinco atajos para el caso super común de "toma la variable, operala con algo y guarda el resultado en ella misma". En lugar de escribir:

```
contador = contador + 1;
total = total - descuento;
precio = precio * 1.16;
```

puedes escribir:

```
contador += 1;
total -= descuento;
precio *= 1.16;
```

Los cinco son: $+=$ suma y asigna, $-=$ resta y asigna, $*=$ multiplica y asigna, $/=$ divide y asigna, $\%=$ saca el módulo y asigna. Funcionan así: `variable OP= valor` es exactamente lo mismo que `variable = variable OP valor`. Son más cortos, se leen mejor cuando te acostumbras y, lo más importante, evitan que escribas dos veces el nombre de una variable larga y te equivoques en una de ellas.

Hay un detalle de precedencia que vale oro: en una expresión como `total += precio * cantidad`, primero se evalúa toda la derecha (`precio * cantidad`) y después se suma a `total`. Es decir, `total += precio * cantidad` equivale a `total = total + (precio * cantidad)`, no a `(total + precio) * cantidad`. La derecha del operador de asignación siempre se resuelve completa primero. Grabatelo porque te va a ahorrar dolores de cabeza.

3.4 Incremento y decremento: los operadores más usados de C

Hay dos operaciones tan comunes en programación que C les dio su propio operador: sumarle 1 a una variable y restarle 1. Son $++$ (incremento) y $--$ (decremento). `contador++` es lo mismo que `contador += 1`, que es lo mismo que `contador = contador + 1`. Los vas a ver en casi cada ciclo `for` que escribas en tu vida.

Pero aquí viene el truco que separa a los principiantes de los que ya entendieron: ++ y -- existen en dos versiones, prefijo y postfijo, y se comportan distinto cuando los usas DENTRO de una expresión más grande. Mira con atención:

```
int a = 5;
int b = a++; // Postfijo: primero asigna, despues incrementa.
printf("a = %d, b = %d\n", a, b); // a = 6, b = 5

int c = 5;
int d = ++c; // Prefijo: primero incrementa, despues asigna.
printf("c = %d, d = %d\n", c, d); // c = 6, d = 6
```

La regla nemotecnica: en postfijo (a++), el ++ va después, así que la variable se usa primero con su valor viejo y se incrementa después. En prefijo (++a), el ++ va antes, así que la variable se incrementa primero y se usa ya con el valor nuevo. En ambos casos la variable termina valiendo 6; lo que cambia es que valor se entrego a la expresión.

Cuando los usas solos, en su propia línea (contador++;), no hay ninguna diferencia: usa el que quieras, aunque por convención casi todo el mundo escribe el postfijo. La diferencia solo importa dentro de expresiones, y mi recomendación honesta es: evita usar ++ dentro de expresiones complicadas. Escribe código claro en dos líneas en lugar de una línea "inteligente" que nadie entiende, ni tú mismo dentro de dos semanas. Los programadores profesionales valoran la claridad sobre la astucia.

Un error común es pensar que ++ suma 1.0 a un flotante o algo raro: no, ++ suma exactamente 1, del tipo que sea la variable. Y otro detalle: no puedes hacer (a + b)++ porque ++ necesita una variable real donde guardar, no un resultado temporal.

3.5 Operadores relacionales: las preguntas que tu programa puede hacer

Los operadores aritmeticos calculan; los operadores relacionales PREGUNTAN. Toman dos valores y responden con un si o un no: en C, eso significa 1 (verdadero) o 0 (falso). C no tenía un tipo booleano en sus orígenes, así que la convención es: cero es falso, cualquier valor distinto de cero es verdadero. (Hoy existe _Bool y bool con stdbool.h, pero la convención clásica sigue viva en todas partes y la verás en código real toda tu vida.)

Los seis operadores relacionales son: == igual que, != distinto de, < menor que, > mayor que, <= menor o igual que, >= mayor o igual que. Fíjate bien en el primero: la comparación de igualdad usa DOS signos de igual ==. Un solo = es asignación, dos == es pregunta. Esta es la confusión más famosa de la programación, y te va a pasar: vas a escribir if (edad = 18) cuando querias decir if (edad == 18), y C no te va a marcar error. Te explico por qué es tan traicionero:

edad = 18 es una asignación válida que además "devuelve" el valor asignado, 18, que es distinto de cero, o sea verdadero. Así que tu if siempre se cumple, la edad siempre termina valiendo 18, y tu programa se comporta de forma misteriosa. Cuando un if se cumpla "siempre" sin razón aparente, lo primero que debes revisar es si escribiste = en lugar de ==.

Mira cómo se ven en acción:

```
int edad = 20;
printf("%d\n", edad == 18);    // 0: falso, 20 no es igual a 18
printf("%d\n", edad != 18);    // 1: verdadero, 20 si es distinto de 18
printf("%d\n", edad >= 18);    // 1: verdadero, 20 es mayor o igual a 18
printf("%d\n", edad < 18);     // 0: falso
```

Puedes comparar enteros con flotantes sin problema ($5 < 5.5$ es verdadero), y también puedes comparar caracteres, porque en C un char es en el fondo un número, su código ASCII: 'a' < 'b' es verdadero porque 97 es menor que 98, y 'A' < 'a' también es verdadero porque las mayúsculas tienen códigos menores que las minúsculas. Eso te sirve para ordenar letras y validar rangos.

Un truco elegante: como las comparaciones devuelven 1 o 0, puedes usarlas directamente en aritmética. puntos = 100 + (nivel == 5) * 50; le suma 50 puntos solo si el nivel es 5. No abuses de este truco, pero es bueno saber que existe.

3.6 Operadores lógicos: combinar preguntas con Y, O y NO

Una sola comparación rara vez basta en la vida real. Para entrar a un antro necesitas tener 18 o más Y traer identificación. Para que un número este entre 1 y 10 necesitas que sea mayor o igual a 1 Y menor o igual a 10. Los operadores lógicos combinan preguntas simples en preguntas compuestas: && es el Y lógico (AND), || es el O lógico (OR), y ! es el NO lógico (NOT).

! niega una condición: si era verdadera la vuelve falsa y viceversa. !(edad >= 18) es lo mismo que edad < 18. Se usa mucho para preguntar "si NO paso tal cosa".

&& pide que AMBAS condiciones sean verdaderas para que el resultado sea verdadero. || pide que AL MENOS UNA sea verdadera. Aquí están sus tablas de verdad, las tablas sagradas de la lógica que todo programador debe saberse de memoria:

Tabla de verdad de && (Y):

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

Solo hay un 1 en toda la tabla: cuando ambas son verdaderas. Piensa en un candado con dos llaves: necesitas las dos.

Tabla de verdad de || (O):

A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

Solo hay un 0: cuando ambas son falsas. Piensa en entrar a una fiesta: pasas si traes boleto O si estás en la lista; solo te quedas fuera si no cumples ninguna.

Tabla de verdad de ! (NO):

A	!A
0	1
1	0

Simple: invierte el valor.

En código se ven así:

```
int edad = 20;
int traeID = 1; // 1 significa que si trae identificacion

printf("%d\n", (edad >= 18) && traeID); // 1: puede entrar al antro
printf("%d\n", (edad >= 21) || traeID); // 1: al menos una se cumple
printf("%d\n", !(edad >= 18));         // 0: no es menor de edad
```

Dos superpoderes de los operadores lógicos en C que debes conocer. Primero, la evaluación de cortocircuito: en `A && B`, si A es falsa, C ni siquiera evalúa B, porque el resultado ya está decidido (mira la tabla: con A en 0, el resultado es 0 pase lo que pase). En `A || B`, si A es verdadera, C se salta B. Esto no es solo optimización: te permite escribir cosas como `(n != 0) && (100 / n > 5)`, donde la división solo se ejecuta si n no es cero, evitando una división entre cero. El orden importa: pon siempre primero la condición que protege a la segunda.

Segundo, el error clásico de comparar contra dos valores: para preguntar si x vale 1 o 2, los principiantes escriben `(x == 1 || 2)`, y eso está mal. Esa expresión se evalúa como `(x == 1) || (2)`, y como 2 es distinto de cero, es verdadero SIEMPRE. Lo correcto es repetir la variable: `(x == 1 || x == 2)`. C no entiende el español implícito; cada lado del `||` debe ser una pregunta completa.

Otro error típico: confundir `&` con `&&` y `|` con `||`. Un solo `&` y un solo `|` son operadores a nivel de bits, un tema avanzado que verás mucho después. Si los usas por accidente en lugar de los lógicos, tu programa compila pero se comporta raro. Doble símbolo para lógica, siempre.

3.7 Precedencia y asociatividad: quien se evalúa primero

Cuando una expresión tiene varios operadores, C no los evalúa de izquierda a derecha a lo loco: sigue un orden estricto llamado precedencia, igual que en matemáticas donde la multiplicación va antes que la suma. Y cuando dos operadores tienen la misma precedencia, la asociatividad decide si se agrupan de izquierda a derecha o de derecha a izquierda.

La tabla de precedencia, de mayor a menor, para lo que has visto hasta ahora:

1. Paréntesis `()`: lo más fuerte, tu los controlas.
2. `!`, `++`, `--` (prefijo), `cast`, `&` unario: operadores unarios, se evalúan de derecha a izquierda.
3. `*`, `/`, `%`: multiplicativos, de izquierda a derecha.
4. `+`, `-`: aditivos, de izquierda a derecha.
5. `<`, `<=`, `>`, `>=`: relacionales, de izquierda a derecha.
6. `==`, `!=`: igualdad, de izquierda a derecha (ojo: más abajo que los relacionales).
7. `&&`: Y lógico, de izquierda a derecha.
8. `||`: O lógico, de izquierda a derecha.
9. `=`, `+=`, `-=`, `*=`, `/=`, `%=`: asignación, de derecha a izquierda.

No te la aprendas de memoria como loro; aprendela con ejemplos tramposos, que es cómo se queda grabada.

Ejemplo trampa 1: $2 + 3 * 4$. La multiplicación va primero: $2 + 12 = 14$, no 20. Igual que en la escuela.

Ejemplo trampa 2: $10 - 4 - 3$. La resta es asociativa a la izquierda: $(10 - 4) - 3 = 3$. Si alguien la agrupara al revés daría $10 - 1 = 9$, que es incorrecto. Con operadores del mismo nivel, avanza de izquierda a derecha.

Ejemplo trampa 3, el clásico de entrevistas: $!edad == 0$. El $!$ tiene más precedencia que $==$, así que se evalúa como $(!edad) == 0$. Si edad vale 5, $!5$ es 0, y $0 == 0$ es 1, verdadero. Mucha gente cree que significa $!(edad == 0)$, que sería lo mismo en este caso pero no siempre. Moraleja: con $!$ y comparaciones, usa paréntesis.

Ejemplo trampa 4: $a = b = 5$. La asignación es asociativa a la derecha: primero $b = 5$, y esa asignación "devuelve" 5, que luego se asigna a a . Las dos quedan en 5. Por eso funcionan las asignaciones en cadena.

Ejemplo trampa 5, el más peligroso: $x \& 1 == 1$, donde $\&$ es el operador a nivel de bits. Como $==$ tiene MÁS precedencia que $\&$ (aunque suene ilógico), se evalúa como $x \& (1 == 1)$, o sea $x \& 1$. De casualidad da lo mismo aquí, pero en otros casos muerde. Los diseñadores de C admitieron después que esa precedencia fue un error histórico. Lección: cuando mezcles operadores de familias distintas, los paréntesis son tus amigos.

La regla de oro de los programadores profesionales: si tienes que pensar más de dos segundos en el orden de evaluación, pon paréntesis. $(a + b) * c$ es obvio para cualquiera; $a + b * c$ exige recordar la tabla. Los paréntesis no cuestan nada en rendimiento y salvan lectores, incluido tu yo del futuro. Escribe código para humanos, no para presumir que te sabes la tabla.

Un último detalle de asociatividad con el ejemplo favorito de los exámenes: $100 / 10 * 2$. Como $/$ y $*$ tienen la misma precedencia y asocian a la izquierda, se evalúa como $(100 / 10) * 2 = 20$, no como $100 / (10 * 2) = 5$. La calculadora de tu teléfono hace lo mismo.

3.8 Conversión de tipos: cuando C mezcla peras con manzanas

En tus programas vas a mezclar tipos todo el tiempo: un `int` con un `float`, un `char` con un `int`. C no se queja: convierte automáticamente al tipo "más grande" para que la operación tenga sentido. Esto se llama conversión implícita o promoción, y sigue una jerarquía: `char` se convierte a `int`, `int` se convierte a `float`, `float` se convierte a `double`. El resultado de una operación siempre es del tipo del operando más grande.


```
int a = 10;
float b = 2.5;
float r = a + b;    // a se convierte a 10.0, r vale 12.5
```

Hasta aquí todo bien. El problema es cuando la conversión va en dirección contraria, de un tipo grande a uno chico, porque ahí se puede perder información. Si guardas un float en un int, C trunca los decimales sin avisarte:

```
float pi = 3.99;
int entero = pi;    // entero vale 3. Los decimales se pierden en silencio.
```

C no te avisa, no te pregunta, no te marca error: simplemente corta. Lo mismo pasa si metes un número grande en un tipo chico: `int grande = 100000`; `char c = grande`; el char solo guarda 8 bits y el valor se deforma. Estas conversiones silenciosas son fuente de bugs difíciles de encontrar, porque el programa compila perfecto y el resultado solo está "un poco mal".

Para tomar el control existe el cast explícito: le dices a C "convierte este valor a este tipo, lo estoy haciendo a propósito". Se escribe con el tipo entre paréntesis antes del valor:

```
int a = 10, b = 3;
float division = (float)a / b;    // 3.333333. El cast convierte a a en flotante.
```

Fíjate que el cast se aplica a `a`, y luego la división es entre un float y un int, así que C promueve `b` también y hace división real. Si hubieras escrito `(float)(a / b)`, el cast llegaría tarde: primero se haría la división entera (`3`) y luego se convertiría a `3.0`. El lugar del cast importa muchísimo: ponlo sobre el operando, antes de la operación.

El cast también sirve para dejar claro que la pérdida de información es intencional: `int parteEntera = (int)3.99`; le dice a quien lea tu código "sé que estoy cortando los decimales y está bien". Esa es la diferencia entre un bug y una decisión: el cast documenta tu intención.

Dos casos especiales que debes conocer. Primero, en `printf` y `scanf` el tipo debe coincidir con el especificador: si pasas un `double` a `%d`, el resultado es basura, no una conversión elegante. El compilador a veces avisa con un warning; hazle caso. Segundo, los caracteres son números: `(int)'A'` vale 65, y `(char)66` es `'B'`. Eso te permite hacer aritmética con letras, como convertir mayúsculas a minúsculas sumando 32: `'A' + 32` es `'a'`. Lo usarás en la unidad de cadenas.

3.9 Errores comunes: la galería de los horrores del principiante

Vamos a juntar en un solo lugar las trampas que ya vimos, más algunas nuevas, porque reconocer un error cuando lo ves es la mitad de la depuración. Cada uno de estos le ha costado horas a miles de programadores; que a ti te cueste cinco minutos de lectura.

Error 1: usar `=` en lugar de `==`. `if (x = 5)` asigna 5 a `x` y siempre es verdadero. El compilador moderno a veces avisa con un warning del tipo "suggest parentheses around assignment used as truth value": hazle caso siempre. Un truco defensivo que usan algunos programadores es escribir la constante primero: `if (5 == x)`. Si te equivocas y pones un solo `=`, queda `if (5 = x)`, que es un error de compilación obvio porque no puedes asignar a una constante. Es feo de leer, pero te salva. Con la práctica, revisar el `==` se vuelve automático.

Error 2: olvidar el `&` en `scanf`. `scanf("%d", edad)` sin `&` compila (a veces con warning) pero en ejecución el programa suele tronar o comportarse de forma loca, porque `scanf` escribe en una dirección de memoria inventada. Si tu programa se congela o se cierra justo al pedir un dato, el `&` es el primer sospechoso. La excepción: con `%s` no lleva `&`.

Error 3: la división entera inesperada. `promedio = (a + b + c) / 3` con todo entero te da un promedio truncado. Solución: haz flotante al menos un operando, con `3.0` o con un cast. Este error es traicionero porque el programa "funciona" y el resultado parece razonable, solo está mal.

Error 4: comparar flotantes con `==`. Los números flotantes se guardan en binario con precisión limitada, así que `0.1 + 0.2` no es exactamente `0.3` en la computadora. Si comparas `(0.1 + 0.2 == 0.3)`, puede darte falso. La práctica correcta es comparar con un margen de tolerancia: en lugar de preguntar si son iguales, pregunta si su diferencia es muy pequeña, por ejemplo `fabs(a - b) < 0.0001`. Este detalle te va a salvar en programas científicos y financieros.

Error 5: el punto y coma después del `if...` pero esa es la próxima unidad. Lo menciono para que no te espantes cuando lo veas.

Error 6: usar `%d` para leer un float o `%f` para leer un double en `scanf`. Recuerda: en `scanf`, float usa `%f` y double usa `%lf`. Si los confundes, `scanf` escribe los bytes en el formato equivocado y tu variable queda con basura. En `printf`, en cambio, `%f` sirve para ambos. Esta asimetría es una de las cosas más confusas de C para principiantes: `scanf` es estricto, `printf` es flexible.

Error 7: creer que `&&` y `||` devuelven "verdadero o falso" como palabras. Devuelven 1 o 0, números normales que puedes imprimir, sumar y guardar. Y al revés: cualquier número distinto de cero cuenta como verdadero en una condición, incluso los negativos.

3.10 Buenas prácticas: como escriben los profesionales

Ya sabes lo suficiente para escribir código que funciona; ahora, cómo escribir código que funciona Y se entiende. Estas prácticas las usan los programadores profesionales todos los días.

Primera: compila siempre con warnings activados. `gcc -Wall -Wextra tu_programa.c` le pide al compilador que te avise de cosas sospechosas: el = dentro de un if, el especificador que no coincide con el tipo, la variable sin usar. Los warnings son el compilador diciendote "esto compila, pero huele mal". Trata cada warning como un error hasta que entiendas exactamente por que aparece. En la clase 29 veremos depuración a fondo, pero este habito empieza hoy.

Segunda: inicializa tus variables al declararlas. `int total = 0;` es mejor que `int total;` porque una variable sin inicializar contiene basura, lo que haya quedado en esa zona de memoria. Si la usas antes de asignarle algo, tu programa hace cálculos con basura y los resultados son impredecibles. Acostumbrate: declarar e inicializar van juntos.

Tercera: usa paréntesis para aclarar, no para presumir. Ya lo vimos en precedencia: si la expresión exige recordar la tabla, pon paréntesis. Tu código lo va a leer un humano cansado a las 2 de la mañana, y ese humano puedes ser tu.

Cuarta: no confies en el usuario. Valida la entrada: revisa lo que devuelve `scanf`, piensa que pasa si el usuario escribe letras donde pedias números o un número negativo donde esperabas una edad. Los programas robustos no truenan con entradas raras; los programas de juguete si.

Quinta: dale nombres que expliquen. `totalCompra` dice más que `t`, y `edadUsuario` dice más que `e`. En esta unidad tus programas son cortos y parece exagerado, pero es un músculo que se entrena desde el día uno.

Sexta: un concepto por línea. En lugar de `printf("%d", (a = b++) + ++c);` escribe las operaciones paso a paso en líneas separadas. El código "compacto e inteligente" es el código que nadie puede depurar, y en C, donde cada detalle de evaluación importa, la claridad te salva de bugs invisibles.

Con esto cierras la teoria. Ahora viene lo más importante: ensuciarte las manos con ejemplos que vas a correr tú mismo.

Ejemplos paso a paso

Todos los ejemplos de esta sección están listos para compilar y correr. Escríbelos tú mismo, no los copies y pegues: la memoria muscular también programa. Compila cada uno con `gcc -Wall -Wextra` para atrapar warnings desde el primer día.

Ejemplo 1: leer nombre y edad

El programa interactivo más básico: pide dos datos de distinto tipo y los usa. Práctica `scanf` con `%s` y `%d`, el prompt amigable y el `&` obligatorio.

```
#include <stdio.h>

int main() {
    char nombre[30];
    int edad;

    printf("Como te llamas? ");
    scanf("%29s", nombre);      // %s sin &, con limite para no desbordar

    printf("Cuantos años tienes, %s? ", nombre);
    scanf("%d", &edad);        // %d con &

    printf("\nMucho gusto, %s. Tienes %d años.\n", nombre, edad);
    printf("En 10 años tendrás %d.\n", edad + 10);

    return 0;
}
```

Que observar al correrlo: el programa se pausa en cada `scanf` esperando tu Enter. Prueba escribir un nombre con apellido, como "Juan Perez", y mira como solo toma "Juan": es la limitación de `%s` con espacios que vimos en la teoría. Prueba escribir letras cuando pide la edad y observa el comportamiento raro: es el bufer sucio en acción. Romperlo a propósito es la mejor forma de entenderlo.

Ejemplo 2: calculadora básica

Lee dos números flotantes y un operador, y muestra los cuatro resultados. Práctica `%f`, `%lf` según el caso, aritmética y la división real contra la entera.

```

#include <stdio.h>

int main() {
    double a, b;

    printf("Escribe el primer numero: ");
    scanf("%lf", &a);          // double usa %lf en scanf
    printf("Escribe el segundo numero: ");
    scanf("%lf", &b);

    printf("\nResultados:\n");
    printf("Suma: %f\n", a + b);
    printf("Resta: %f\n", a - b);
    printf("Multiplicacion: %f\n", a * b);

    if (b != 0) {
        printf("Division: %f\n", a / b);
    } else {
        printf("Division: no se puede dividir entre cero.\n");
    }

    return 0;
}

```

Nota como protegemos la división con una comparación `b != 0`: es tu primer uso real de un operador relacional para cuidar tu programa. El `if` lo verás a fondo en la siguiente unidad, pero ya lo estás usando como un profesional: para validar antes de operar. Prueba dividir 10 entre 3 y mira los decimales; luego cambia `double` por `int` y `%lf` por `%d`, y mira como la división se vuelve entera. Ese contraste te va a grabar la diferencia para siempre.

Ejemplo 3: residuo y par/impar

El operador `%` en su uso más famoso: decidir si un número es par o impar. También muestra como obtener el último dígito de un número.

```

#include <stdio.h>

int main() {
    int numero;

    printf("Escribe un numero entero: ");
    scanf("%d", &numero);

    int residuo = numero % 2;

    printf("El numero %d es ", numero);
    if (residuo == 0) {
        printf("par.\n");
    } else {
        printf("impar.\n");
    }

    printf("Su ultimo digito es %d.\n", numero % 10);
    printf("Sin el ultimo digito queda %d.\n", numero / 10);

    return 0;
}

```

Tres trucos en un solo ejemplo: número % 2 para par/impar, número % 10 para el último dígito y número / 10 (división entera) para quitar el último dígito. Con estos tres trucos puedes desarmar cualquier número dígito por dígito, que es la base de muchísimos ejercicios clásicos: invertir un número, sumar sus dígitos, verificar capicuas. Pruebalo con 2026 y con números negativos para ver qué pasa con el signo del residuo.

Ejemplo 4: prefijo vs postfijo en acción

El ejemplo que aclara de una vez por todas la diferencia entre ++a y a++. Leelo con calma y predice cada resultado antes de correrlo.

```

#include <stdio.h>

int main() {
    int a = 5;
    int b = 5;

    printf("--- Postfijo (a++): primero se usa, despues se incrementa ---\n");
    a = 5;
    int x = a++;    // x recibe 5, luego a sube a 6
    printf("x = %d, a = %d\n", x, a);

    printf("--- Prefijo (++b): primero se incrementa, despues se usa ---\n");
    b = 5;
    int y = ++b;    // b sube a 6, luego y recibe 6
    printf("y = %d, b = %d\n", y, b);

    int c = 10;
    c++;
    printf("Despues de c++, c vale: %d\n", c);
    ++c;
    printf("Despues de ++c, c vale: %d\n", c);

    return 0;
}

```

Fíjate en el estilo: cada incremento va en su propia línea, separado de cualquier otra operación. Así no hay ambigüedad posible y cualquiera que lea tu código entiende exactamente que pasa y en que orden. Cuando veas código que mezcla ++ dentro de expresiones grandes, sospecha: ahí viven los bugs difíciles.

Ejemplo 5: tablas de verdad en código

En lugar de memorizar las tablas de verdad, genera tú mismo la tabla del && y del || con un programa. Ver los unos y ceros salir de tu propia computadora hace que la lógica se quede grabada.

```

#include <stdio.h>

int main() {
    int a, b;

    printf("Tabla de verdad de && (Y logico):\n");
    printf("  A  |  B  | A && B\n");
    printf("-----+-----+-----\n");
    for (a = 0; a <= 1; a++) {
        for (b = 0; b <= 1; b++) {
            printf("  %d  |  %d  |   %d\n", a, b, a && b);
        }
    }

    printf("\nTabla de verdad de || (O logico):\n");
    printf("  A  |  B  | A || B\n");
    printf("-----+-----+-----\n");
    for (a = 0; a <= 1; a++) {
        for (b = 0; b <= 1; b++) {
            printf("  %d  |  %d  |   %d\n", a, b, a || b);
        }
    }

    printf("\nTabla de verdad de ! (NO logico):\n");
    for (a = 0; a <= 1; a++) {
        printf("  !%d = %d\n", a, !a);
    }

    return 0;
}

```

Este ejemplo usa for, que verás a fondo en la clase 14, pero su idea es intuitiva: repite con a y b tomando los valores 0 y 1 en todas las combinaciones. No te preocupes si el for aun se siente nuevo; concentrare en la salida. Comprueba que coincide con las tablas de la sección 3.6: el && solo da 1 cuando ambas son 1, el || solo da 0 cuando ambas son 0. Agrega tu la tabla de una expresión compuesta, por ejemplo !(a && b), y verifica que equivale a (!a || !b): acabas de descubrir una ley de De Morgan con código.

Ejemplo 6: precedencia bajo la lupa

Un programa que imprime una serie de expresiones tramposas junto a su resultado real, para que compares con lo que tu habrias respondido. Haz el ejercicio mental primero: tapa la columna de resultados, escribe tu predicción y luego corre el programa.


```

#include <stdio.h>

int main() {
    int a = 10, b = 4, c = 3;

    printf("2 + 3 * 4 = %d (multiplicacion primero)\n", 2 + 3 * 4);
    printf("(2 + 3) * 4 = %d (parentesis mandan)\n", (2 + 3) * 4);
    printf("a - b - c = %d (izquierda a derecha: (10-4)-3)\n", a - b - c);
    printf("a / b * c = %d (misma precedencia, izquierda a derecha)\n", a / b * c);
    printf("a %% b + c = %d (modulo antes que suma)\n", a % b + c);
    printf("!0 == 1 es %d (! se evalua antes que ==)\n", !0 == 1);

    int x = 5;
    printf("x > 3 && x < 10 es %d (relacionales antes que &&)\n", x > 3 && x < 10);
    printf("x == 5 || x == 6 && x > 100 es %d (&& antes que ||)\n",
        x == 5 || x == 6 && x > 100);

    return 0;
}

```

Detente en la última: `x == 5 || x == 6 && x > 100` se evalúa como `x == 5 || (x == 6 && x > 100)`, porque `&&` tiene más precedencia que `||`. Como `x` vale 5, la primera parte ya es verdadera y el resultado es 1. Si alguien lee esa línea sin saber la precedencia, puede malinterpretarla por completo. Este es el ejemplo perfecto de cuando un par de paréntesis habría hecho el código obvio: `(x == 5) || ((x == 6) && (x > 100))`. Más paréntesis, menos dudas.

Ejemplo 7: el cast que salva promedios

El programa que todo estudiante necesita: calcula el promedio de tres calificaciones. La primera versión lo hace mal (división entera) y la segunda lo corrige con un cast. Ver el error y su arreglo lado a lado es la mejor vacuna contra la trampa de la división entera.

```
#include <stdio.h>

int main() {
    int cal1, cal2, cal3;

    printf("Escribe tres calificaciones (0-100): ");
    scanf("%d %d %d", &cal1, &cal2, &cal3);

    // Version ingenua: todo entero, division entera, decimales perdidos
    int promedioMal = (cal1 + cal2 + cal3) / 3;
    printf("Promedio (version ingenua): %d\n", promedioMal);

    // Version correcta: el cast convierte la suma a float ANTES de dividir
    float promedioBien = (float)(cal1 + cal2 + cal3) / 3;
    printf("Promedio (version correcta): %.2f\n", promedioBien);

    // Otra forma correcta: usar 3.0 en lugar de 3
    float promedioBien2 = (cal1 + cal2 + cal3) / 3.0;
    printf("Promedio (con 3.0): %.2f\n", promedioBien2);

    return 0;
}
```

Prueballo con 85, 90 y 78: la versión ingenua dice 84 y las correctas dicen 84.33. Fíjate en el detalle crucial: el cast va sobre la suma, antes de la división. Si escribieras `(float)((cal1 + cal2 + cal3) / 3)`, el cast llegaría tarde y obtendrías 84.00: un promedio "flotante" pero igual de mal. El `%.2f` del `printf` limita la salida a dos decimales para que se vea limpio. Este patrón, `(float)suma / cantidad`, lo vas a usar en decenas de programas: promedios, porcentajes, proporciones.

Ejemplo 8: mini programa interactivo, la tiendita

Juntamos todo lo de la unidad en un programa útil de verdad: una tiendita que pide el precio y la cantidad de tres productos, calcula el subtotal, agrega el IVA del 16 por ciento y muestra el total. Usa `scanf`, aritmética, asignación compuesta, módulo para redondeo y cast donde hace falta.

```
#include <stdio.h>

int main() {
    float precio1, precio2, precio3;
    int cant1, cant2, cant3;
```

```

printf("=== LA TIENDITA DE DON BETO ===\n\n");

printf("Precio del producto 1: ");
scanf("%f", &precio1);
printf("Cantidad del producto 1: ");
scanf("%d", &cant1);

printf("Precio del producto 2: ");
scanf("%f", &precio2);
printf("Cantidad del producto 2: ");
scanf("%d", &cant2);

printf("Precio del producto 3: ");
scanf("%f", &precio3);
printf("Cantidad del producto 3: ");
scanf("%d", &cant3);

float subtotal = 0.0;
subtotal += precio1 * cant1; // acumulamos con +=
subtotal += precio2 * cant2;
subtotal += precio3 * cant3;

float iva = subtotal * 0.16;
float total = subtotal + iva;

printf("\n--- TICKET ---\n");
printf("Subtotal: $%.2f\n", subtotal);
printf("IVA (16%%): $%.2f\n", iva);
printf("TOTAL: $%.2f\n", total);

// Cuanto pagaria en numeros cerrados? Redondeo hacia arriba con modulo
int totalCerrado = (int)total;
if ((total - totalCerrado) > 0) {
    totalCerrado += 1;
}
printf("Si pagas con billetes cerrados, serian $%.2f\n", totalCerrado);

int totalCentavos = (int)(total * 100 + 0.5); // truco de redondeo
printf("Eso es %d pesos con %02d centavos.\n",
    totalCentavos / 100, totalCentavos % 100);

return 0;
}

```

Observa los detalles profesionales: el %% dentro del printf para imprimir un signo de porcentaje literal; el acumulador subtotal que empieza en 0.0 y crece con +=; la separación de pesos y centavos usando división entera y módulo sobre el total en centavos; y el truco de sumar 0.5 antes del cast a int para redondear en lugar de truncar. Pruebalo con precios reales, cambia el IVA, agrega un cuarto producto. Este programa ya hace algo útil: es la semilla de un punto de venta de verdad.

Práctica

Es hora de programar tu. Los ejercicios están ordenados de menor a mayor dificultad y cubren todo lo de la unidad: scanf, aritmética, módulo, comparaciones, lógica, precedencia y casts. Regla del curso: intenta cada uno POR TU CUENTA antes de mirar la solución. Escribe el código, compilalo con gcc -Wall -Wextra, correlo, equivocate y arreglalo. Si te atorras más de treinta minutos, mira la solución, entiendela a fondo y luego escribela tu solo sin verla. Así se aprende C de verdad.

Ejercicio 1: área y perimetro del rectangulo

Escribe un programa que pida la base y la altura de un rectangulo (números decimales) y muestre su área (base por altura) y su perimetro (dos veces la base más dos veces la altura). Usa especificadores correctos para float en scanf y printf.

Ejercicio 2: de Celsius a Fahrenheit

Escribe un programa que pida una temperatura en grados Celsius y la convierta a Fahrenheit con la fórmula $F = C * 9 / 5 + 32$. Cuidado: esta fórmula esconde la trampa de la división entera si la escribes sin pensar. Asegurate de que el resultado tenga decimales correctos. Pruebalo con 0 (debe dar 32), con 100 (debe dar 212) y con 37 (debe dar 98.6).

Ejercicio 3: segundos a horas, minutos y segundos

Escribe un programa que pida una cantidad total de segundos y la convierta al formato horas:minutos:segundos. Por ejemplo, 3661 segundos son 1:01:01. Pista: la división entera y el módulo son tus mejores amigos aquí. Las horas son segundos / 3600, y lo que sobra se reparte entre minutos y segundos.

Ejercicio 4: par, impar y multiplo

Escribe un programa que pida un número entero e informe tres cosas: si es par o impar, si es multiplo de 5, y si es par Y multiplo de 5 al mismo tiempo (usa &&). Debe funcionar también con números negativos y con el cero. Piensa: que imprime tu programa si el usuario escribe 0? Es par? Es multiplo de 5?

Ejercicio 5: promedio con decimales (y sin trampas)

Escribe un programa que pida las 4 calificaciones parciales de un estudiante (enteros de 0 a 100) y muestre el promedio con dos decimales. El programa debe evitar la trampa de la división entera usando un cast explícito. Además, muestra el promedio truncado a entero al lado, para que compares la diferencia. Pruebalo con 85, 90, 78 y 92.

Ejercicio 6 (reto): el cajero que da cambio

Escribe un programa que pida una cantidad entera en pesos y calcule el número mínimo de billetes y monedas necesarios para formar esa cantidad, usando billetes de 500, 200, 100, 50 y 20, y monedas de 10, 5, 2 y 1. Por ejemplo, 873 pesos son: 1 de 500, 1 de 200, 1 de 100, 1 de 50, 1 de 20 y 3 de 1. Pista: es el mismo truco del ejemplo 3 aplicado en cadena: división entera para saber cuántos billetes de cada tipo, módulo para el resto que sigue con el siguiente billete. Solo muestra las denominaciones que se usen (si son cero, no las muestres).

Solución del ejercicio 1

```
#include <stdio.h>

int main() {
    float base, altura;

    printf("Base del rectangulo: ");
    scanf("%f", &base);
    printf("Altura del rectangulo: ");
    scanf("%f", &altura);

    float area = base * altura;
    float perimetro = 2 * base + 2 * altura;

    printf("Area: %.2f\n", area);
    printf("Perimetro: %.2f\n", perimetro);

    return 0;
}
```

Explicación: usamos float porque las medidas pueden tener decimales, y en scanf el float usa %f con su & correspondiente. El área es multiplicación directa. El perimetro aprovecha la precedencia: $2 * base + 2 * altura$ se evalúa como $(2 * base) + (2 * altura)$ porque * va antes que +. El %.2f muestra dos decimales, suficiente para medidas. Variante para practicar: pide los datos en centímetros y muestra el área en centímetros cuadrados.

Solución del ejercicio 2

```
#include <stdio.h>

int main() {
    float celsius;

    printf("Temperatura en Celsius: ");
    scanf("%f", &celsius);

    // OJO: 9 / 5 con enteros daría 1. Usamos 9.0 / 5.0 para forzar decimales.
    float fahrenheit = celsius * 9.0 / 5.0 + 32;

    printf("%.1f C son %.1f F\n", celsius, fahrenheit);

    return 0;
}
```

Explicación: este ejercicio es una trampa con disfraz de ejercicio fácil. Si escribes `celsius * 9 / 5 + 32` con todo entero, el `9 / 5` se evalúa primero como división entera y da 1, así que la fórmula se convierte en `celsius * 1 + 32`, que está mal. Al escribir `9.0 / 5.0`, uno de los operandos es flotante y la división es real: 1.8. Como `celsius` ya es float, toda la expresión se promueve a flotante. Prueba mental: con `celsius = 37`, tenemos $37 * 1.8 = 66.6$, más 32 = 98.6. Correcto. Este es exactamente el tipo de bug silencioso que la sección 3.9 te enseñó a cazar: el programa compila, corre y da un número "casi bien".

Solución del ejercicio 3

```
#include <stdio.h>

int main() {
    int totalSegundos;

    printf("Cantidad total de segundos: ");
    scanf("%d", &totalSegundos);

    int horas = totalSegundos / 3600;           // division entera: horas completas
    int resto = totalSegundos % 3600;          // lo que sobra despues de las horas
    int minutos = resto / 60;                  // minutos completos del resto
    int segundos = resto % 60;                 // segundos finales

    printf("%d segundos son %d:%02d:%02d\n", totalSegundos, horas, minutos,
segundos);

    return 0;
}
```

Explicación: la técnica es "divide y quedate con el resto", aplicada dos veces. Primero sacas las horas completas con división entera entre 3600; el módulo te deja el sobrante menor a una hora. De ese sobrante sacas los minutos completos entre 60, y el módulo final te deja los segundos. Con 3661: horas = 1, resto = 61, minutos = 1, segundos = 1, o sea 1:01:01. El `%02d` en el `printf` rellena con un cero a la izquierda para que se vea como reloj digital (01 en lugar de 1). Este patrón de división y módulo en cadena aparece en todos lados: convertir minutos a días, centavos a pesos, o bytes a gigabytes.

Solución del ejercicio 4

```
#include <stdio.h>

int main() {
    int n;

    printf("Escribe un numero entero: ");
    scanf("%d", &n);

    int esPar = (n % 2 == 0);
    int esMultiploDe5 = (n % 5 == 0);

    printf("El numero %d:\n", n);

    if (esPar) {
        printf("- Es par.\n");
    } else {
        printf("- Es impar.\n");
    }

    if (esMultiploDe5) {
        printf("- Es multiplo de 5.\n");
    } else {
        printf("- No es multiplo de 5.\n");
    }

    if (esPar && esMultiploDe5) {
        printf("- Es par Y multiplo de 5 al mismo tiempo.\n");
    }

    return 0;
}
```

Explicación: guardamos cada pregunta en una variable (esPar, esMultiploDe5) para no repetir la expresión: código más claro y más fácil de depurar. $n \% 2 == 0$ detecta pares porque el residuo de dividir entre 2 solo puede ser 0 o 1 (o -1 con negativos, pero $-4 \% 2$ sigue siendo 0, así que los pares negativos también pasan). El cero es par y también es multiplo de 5, porque $0 \% 5 == 0$: matemáticamente correcto, aunque a algunos les sorprende. La última condición usa $\&\&$ para pedir las dos cosas a la vez, justo como en la tabla de verdad: solo es verdadera cuando ambas lo son. Nota los paréntesis en $(n \% 2 == 0)$: el $\%$ tiene más precedencia que $==$, así que funcionaria sin ellos, pero con ellos se lee sin tener que pensar.

Solución del ejercicio 5

```
#include <stdio.h>

int main() {
    int c1, c2, c3, c4;

    printf("Escribe las 4 calificaciones separadas por espacios: ");
    scanf("%d %d %d %d", &c1, &c2, &c3, &c4);

    int suma = c1 + c2 + c3 + c4;

    // Sin cast: division entera, decimales perdidos
    int promedioTruncado = suma / 4;

    // Con cast: la suma se convierte a float ANTES de dividir
    float promedioReal = (float)suma / 4;

    printf("Suma total: %d\n", suma);
    printf("Promedio truncado (entero): %d\n", promedioTruncado);
    printf("Promedio real (con decimales): %.2f\n", promedioReal);

    return 0;
}
```

Explicación: con 85, 90, 78 y 92 la suma es 345. La versión sin cast hace $345 / 4 = 86$ (truncado); la versión con cast hace $345.0 / 4 = 86.25$. Medio punto de diferencia que en la vida real decide si pasas o no una materia. Fíjate en dos detalles: primero, scanf puede leer varios valores en una sola llamada si separas los especificadores, y el usuario los escribe separados por espacios o Enters; scanf devuelve 4 si leyó los cuatro bien. Segundo, el cast va sobre suma, antes de la división: $(\text{float})\text{suma} / 4$. Si lo pusieras después, $(\text{float})(\text{suma} / 4)$, obtendrías 86.00, un flotante con el mismo error. El lugar del cast lo es todo, como viste en el ejemplo 7.

Solución del ejercicio 6

```

#include <stdio.h>

int main() {
    int cantidad;

    printf("Cantidad en pesos: ");
    scanf("%d", &cantidad);

    int resto = cantidad;

    int b500 = resto / 500; resto %= 500;
    int b200 = resto / 200; resto %= 200;
    int b100 = resto / 100; resto %= 100;
    int b50  = resto / 50;  resto %= 50;
    int b20  = resto / 20;  resto %= 20;
    int m10  = resto / 10;  resto %= 10;
    int m5   = resto / 5;   resto %= 5;
    int m2   = resto / 2;   resto %= 2;
    int m1   = resto;       // lo que queda son monedas de 1

    printf("\nPara $%d necesitas:\n", cantidad);
    if (b500 > 0) printf("- %d billete(s) de $500\n", b500);
    if (b200 > 0) printf("- %d billete(s) de $200\n", b200);
    if (b100 > 0) printf("- %d billete(s) de $100\n", b100);
    if (b50  > 0) printf("- %d billete(s) de $50\n", b50);
    if (b20  > 0) printf("- %d billete(s) de $20\n", b20);
    if (m10  > 0) printf("- %d moneda(s) de $10\n", m10);
    if (m5   > 0) printf("- %d moneda(s) de $5\n", m5);
    if (m2   > 0) printf("- %d moneda(s) de $2\n", m2);
    if (m1   > 0) printf("- %d moneda(s) de $1\n", m1);

    return 0;
}

```

Explicación: este es el algoritmo "voraz" o greedy: siempre toma tantos billetes como pueda de la denominación más grande, y con el resto sigue a la siguiente. Cada paso usa el mismo patrón de dos líneas: división entera para contar cuántos billetes de ese tipo caben, y %= para quedarte con el resto. Con 873: $873 / 500 = 1$, resto 373; $373 / 200 = 1$, resto 173; $173 / 100 = 1$, resto 73; $73 / 50 = 1$, resto 23; $23 / 20 = 1$, resto 3; $3 / 10 = 0$, resto 3; $3 / 5 = 0$; $3 / 2 = 1$, resto 1; queda 1 moneda de 1. Total: $1+1+1+1+1+1+1 = 7$ piezas, el mínimo posible. Los if imprimen evitan mostrar denominaciones en cero, un detalle de presentación que separa un programa decente de uno profesional. Este patrón voraz funciona con las denominaciones del peso mexicano porque cada billete es múltiplo útil del siguiente; no funciona con cualquier sistema de monedas del mundo, pero ese ya es tema de algoritmos avanzados.

Resumen de la unidad

Cerramos la Unidad 3, y mira todo lo que tus programas ya saben hacer: escuchar al usuario, calcular, comparar y combinar condiciones. Repasemos los puntos clave.

scanf es la puerta de entrada de tu programa: lee datos del teclado y los guarda en tus variables con scanf("formato", &variable). Usa el especificador correcto para cada tipo (%d, %f, %lf para double, %c, %s), no olvides el & (excepto con %s), limita las cadenas (%29s) y recuerda el espacio antes de %c (" %c") para evitar el carácter fantasma del Enter. scanf devuelve cuántos datos leyó: usalo para validar.

Los operadores aritméticos son + - * / %, y la lección más importante de toda la unidad es la división entera: entre enteros, C trunca sin avisar. Si quieres decimales, mete un flotante en la operación con 2.0 o con un cast. El % te da el residuo y sirve para par/impar, últimos dígitos y repartos.

Los operadores de asignación compuesta (+= -= *= /= %=) actualizan una variable sin repetir su nombre, y ++ y -- le suman o restan 1. En su propia línea, prefijo y postfijo son idénticos; dentro de una expresión, ++ usa el valor viejo y ++a usa el valor nuevo. Escribe incrementos en líneas separadas y evita la "astucia" compacta.

Los operadores relacionales (== != < > <= >=) hacen preguntas que responden 1 o 0, y los lógicos (! && ||) las combinan siguiendo sus tablas de verdad. C evalúa && y || con cortocircuito: se detiene en cuanto el resultado está decidido. Y el error más famoso de la programación sigue siendo escribir = donde querías ==: cuando un if se cumpla siempre sin razón, revisa eso primero.

La precedencia decide el orden: paréntesis, unarios, * / %, + -, relacionales, == !=, &&, ||, y al final la asignación. Si dudas, pon paréntesis: el código claro le gana al código astuto.

La conversión implícita promueve al tipo más grande automáticamente, pero al guardar en un tipo más chico se pierde información en silencio. El cast explícito, `(float)x`, toma el control y documenta tu intención; ponlo antes de la operación, no después.

Si dominas esta unidad, ya tienes las herramientas para programas interactivos de verdad: pedir datos, validarlos, calcular con ellos y combinar condiciones. En la siguiente unidad, la 4, les vas a dar cerebro a tus programas con `if`, `else` y `switch`: las decisiones. Y recuerda que en lucasegaacademy.com tienes el curso en video con estas mismas 30 clases, por si quieres ver los temas en acción. Sigue practicando todos los días, compila con `-Wall -Wextra`, y nos vemos en la Unidad 4. A programar.

UNIDAD 4: Tomar decisiones

Introducción

Bienvenido a la unidad 4. Hasta aquí has aprendido a guardar datos en variables, a mostrarlos con `printf` y a leerlos con `scanf`. Tus programas ya pueden hacer cálculos y conversar un poco con el usuario. Pero seamos honestos: hasta ahora tus programas son tontos. Siguen una lista de instrucciones de arriba hacia abajo, sin desviarse ni un milímetro, como un tren sobre rieles que no puede elegir camino. En esta unidad eso cambia por completo: tus programas van a aprender a pensar, a comparar y a decidir por sí mismos.

Imagina que vas en coche y llegas a un semáforo. No pasas de frente siempre, verdad. Miras la luz: si está en verde, avanzas; si está en rojo, te detienes; si está en amarillo, decides rápido entre frenar o acelerar. Tu cerebro toma decisiones cada segundo basándose en las condiciones que ve. Un programa sin decisiones es como un conductor que solo sabe acelerar: tarde o temprano choca. Los programas que de verdad sirven, los que te gustan y usas todos los días, toman miles de decisiones por segundo.

Pienza en tu celular. Cuando escribes tu contraseña y te equivocas, te dice que el acceso fue denegado. Eso es una decisión: el programa comparo lo que escribiste con lo que esperaba, y según el resultado eligio un camino o el otro. Cuando pides un viaje en una aplicación y te cobra más porque llueve, eso es una decisión: si hay lluvia, el precio sube. Cuando en un videojuego tu personaje pierde vida al tocar un enemigo, decisión. Cuando la página web te muestra un mensaje de error porque dejaste un campo vacío en un formulario, decisión. Todo el software interesante esta construido sobre la pregunta más simple del mundo: que hago Si esto es cierto?

En esta unidad vas a dominar las tres grandes herramientas de decisión en C. La primera es la sentencia `if`, la más usada de todas: si la condición se cumple, ejecuta este bloque; si no, ejecuta este otro. La vas a conocer a fondo, con sus formas `else if` para manejar varias opciones y `else` para el caso por defecto. La segunda es la sentencia `switch`, perfecta para cuando tienes muchas opciones claras, como un menu o los días de la semana: más limpia que una escalera interminable de `if`. La tercera es el operador ternario, una forma compacta y elegante de tomar decisiones cortas en una sola línea.

Pero no solo vas a aprender la sintaxis. Vas a entender como piensa C sobre la verdad: aquí no hay valores booleanos complicados, hay una regla de oro que tienes que grabarte a fuego. Vas a ver las trampas más clásicas que atrapan a todos los principiantes, como el famoso

punto y coma después del if que rompe todo en silencio, o la confusión entre el signo igual de asignación y el doble igual de comparación, errores que te van a perseguir si no los aprendes a detectar desde hoy. Vas a ver los if anidados, cómo evitar que se conviertan en un laberinto imposible de leer, y el truco del else colgante que confunde hasta a programadores con experiencia.

Al terminar esta unidad serás capaz de escribir programas que reaccionan: que validan datos, que clasifican resultados, que ofrecen menus, que toman caminos distintos según lo que el usuario haga. Es el salto de programas que calculan a programas que se comportan. Y creeme, se siente increíble la primera vez que tu programa le dice al usuario "acceso denegado" o "aprobado" por si solo, como si de verdad entendiera.

Así que respira, pon atención y vamos a darle. Esta unidad es larga porque el tema lo merece: las decisiones son el corazón de la programación. Todo lo que aprendas aquí lo vas a usar en cada programa que escribas de ahora en adelante, sin excepción. Vamos paso a paso, con calma, como siempre lo hacemos en lucasegaacademy.com.

Explicación detallada

La sentencia if: la decisión fundamental

La sentencia if es, sin exagerar, la herramienta más importante de toda la programación. Su idea es ridículamente simple: evalúa una condición, y si esa condición es verdadera, ejecuta un bloque de código. Si es falsa, lo salta y sigue con lo que venga después. Así de fácil. Así de poderosa.

Su forma básica es esta:

```
if (condicion) {  
    // codigo que se ejecuta solo si la condicion es verdadera  
}
```

Fíjate en las partes. La palabra reservada if, luego la condición entre paréntesis, y luego el bloque entre llaves. Las llaves agrupan las instrucciones que pertenecen al if. Todo lo que este dentro de esas llaves solo se ejecuta cuando la condición es verdadera. Cuando es falsa, el programa brinca todo el bloque y continua después de la llave de cierre, como si ese código no existiera.

Vamos a verlo con un diagrama mental, como si dibujáramos el flujo en un pizarrón. Imagina una línea que viene de arriba con el flujo del programa. Llega a un rombo, que en los

diagramas de flujo representa una decisión. Dentro del rombo hay una pregunta: es verdadera la condición? Del rombo salen dos flechas. La flecha marcada SI baja hacia un rectángulo que contiene el bloque de código del if; después de ese rectángulo, la flecha se une de nuevo con el flujo principal. La flecha marcada NO se salta el rectángulo por completo y se une directamente con el flujo principal más abajo. Ese es todo el misterio: el if es un desvío en el camino. Si la condición se cumple, el programa toma el desvío, ejecuta esas instrucciones y regresa al camino principal. Si no se cumple, sigue derecho sin desviarse.

Un ejemplo concreto: quieres un programa que salude de forma especial solo si el usuario es mayor de edad.

```
int edad;
printf("Dime tu edad: ");
scanf("%d", &edad);

if (edad >= 18) {
    printf("Eres mayor de edad. Bienvenido.\n");
}

printf("Fin del programa.\n");
```

Si el usuario escribe 25, la condición `edad >= 18` es verdadera, se imprime el mensaje de bienvenida y luego "Fin del programa". Si escribe 12, la condición es falsa, el mensaje de bienvenida se salta por completo y solo se imprime "Fin del programa". El programa decidió.

Condiciones y valores de verdad en C: la regla de oro

Aquí viene algo que tienes que entender a fondo porque C es distinto a otros lenguajes en este punto. En C no existe un tipo de dato booleano en el C clásico (el C99 agregó `_Bool` y la cabecera `stdbool.h`, pero la esencia sigue siendo la misma). C trabaja con una regla de oro brutalmente simple: el cero es falso y todo lo demás es verdadero.

Grabate esto: 0 significa falso. Cualquier número distinto de cero, positivo o negativo, significa verdadero. El 1 es verdadero, el 5 es verdadero, el -100 es verdadero, el 0.5 es verdadero. Solo el 0 es falso.

Esto significa que puedes escribir `if (edad)` y funciona: si `edad` vale 0 es falso, si vale cualquier otra cosa es verdadero. También puedes escribir `if (!edad)` usando el operador lógico NOT, que invierte el valor: verdadero cuando `edad` es cero.

Los operadores relacionales que aprendiste en la unidad anterior devuelven 1 cuando la comparación es verdadera y 0 cuando es falsa. Por eso `if (edad >= 18)` funciona: la expresión `edad >= 18` se evalúa a 1 o a 0, y el `if` actúa según ese valor. Todo encaja.

Un detalle importante: C no te obliga a que la condición sea una comparación. Cualquier expresión vale. `if (x = 5)` asigna 5 a `x` y luego evalúa 5, que es distinto de cero, así que siempre es verdadero. Esto es una trampa clásica que veremos en la sección de errores comunes, pero por ahora quedate con la idea: la condición del `if` es simplemente una expresión cuyo valor se interpreta con la regla de oro.

También puedes usar los operadores lógicos para construir condiciones compuestas. Recuerda: `&&` es Y lógico (verdadero solo si ambas partes son verdaderas), `||` es O lógico (verdadero si al menos una parte es verdadera), `!` es negación. Ejemplo: para verificar que una edad es válida para votar en muchos países, necesitarían ser mayor de 18 Y menor de 120, algo así:

```
if (edad >= 18 && edad <= 120) {  
    printf("Edad valida para votar.\n");  
}
```

C evalúa estas expresiones con cortocircuito: en un `&&`, si la primera parte es falsa, ya no evalúa la segunda porque el resultado total será falso de todos modos. En un `||`, si la primera parte es verdadera, se salta la segunda. Esto no solo es eficiente, a veces es esencial para evitar errores, como veremos más adelante.

else: el camino alternativo

El `if` solo ya es útil, pero muchas decisiones tienen dos caras: si pasa esto hago A, si no pasa hago B. Para eso existe `else`. La estructura queda así:

```
if (condicion) {  
    // se ejecuta si la condicion es verdadera  
} else {  
    // se ejecuta si la condicion es falsa  
}
```


El diagrama de flujo ahora tiene el rombo de decisión con dos flechas que llevan a dos rectángulos distintos: el de la izquierda (SI) con el bloque del if, el de la derecha (NO) con el bloque del else. Después de ejecutar cualquiera de los dos bloques, ambas flechas se unen y el flujo continúa. Exactamente uno de los dos bloques se ejecuta, nunca los dos, nunca ninguno.

Ejemplo clásico: par o impar.

```
int numero;
printf("Dame un numero: ");
scanf("%d", &numero);

if (numero % 2 == 0) {
    printf("El numero es par.\n");
} else {
    printf("El numero es impar.\n");
}
```

El operador % te da el residuo de la división. Si el residuo de dividir entre 2 es cero, el número es par. Si no, es impar. No hay tercera opción: todo número entero es par o impar, así que el if-else cubre todos los casos posibles. Esa es la belleza del else: te garantiza que siempre se ejecuta exactamente una de las dos ramas.

else if: escaleras de decisión

Que pasa cuando hay más de dos posibilidades? Por ejemplo, calificar un examen: si sacaste 90 o más es excelente, si sacaste 70 o más es aprobado, si sacaste menos es reprobado. Ahí entra else if, que te permite encadenar varias condiciones en una escalera:

```
if (calificacion >= 90) {
    printf("Excelente.\n");
} else if (calificacion >= 70) {
    printf("Aprobado.\n");
} else {
    printf("Reprobado.\n");
}
```

El programa evalúa las condiciones de arriba hacia abajo, en orden. En cuanto encuentra una verdadera, ejecuta su bloque y se salta todo lo demás, incluyendo el else final. Si ninguna condición es verdadera, se ejecuta el else. El else final es opcional: si lo omites y ninguna condición se cumple, simplemente no se ejecuta ningún bloque.

El orden de las condiciones importa muchísimo. Fíjate que primero preguntamos por ≥ 90 y después por ≥ 70 . Si lo hicieras al revés, preguntando primero ≥ 70 , un 95 entraría en la primera condición y nunca llegaría a la de excelente. La regla general es ordenar de la condición más específica o más restrictiva a la más general. Es como un embudo: los casos especiales primero, los generales después.

El diagrama de flujo de una escalera `else if` es una serie de rombos en cascada. El flujo entra al primer rombo; si la respuesta es SI, ejecuta su bloque y sale de toda la estructura; si es NO, cae al siguiente rombo, y así sucesivamente hasta llegar al `else` final o salir sin hacer nada. Visualízalo como una serie de puertas: el programa toca la primera puerta, si se abre entra y se acaba; si no, sigue a la siguiente.

Puedes encadenar todos los `else if` que necesites, no hay límite en el lenguaje, aunque si tienes demasiados quizá un `switch` sea mejor opción, como veremos.

If anidados y la trampa del `else` colgante

Un `if` anidado es simplemente un `if` dentro de otro `if`. Lo usas cuando una decisión depende de otra previa. Por ejemplo: primero verificas si el usuario ingresó una contraseña correcta, y solo si es correcta verificas si tiene permisos de administrador.

```
if (contrasenaCorrecta) {
    printf("Acceso permitido.\n");
    if (esAdministrador) {
        printf("Tienes permisos de administrador.\n");
    } else {
        printf("Eres usuario normal.\n");
    }
} else {
    printf("Contraseña incorrecta.\n");
}
```

Las llaves son tus mejores amigas aquí: dejan clarísimo a qué `if` pertenece cada `else`. Pero ¿qué pasa si omites las llaves? Llegamos a una de las trampas más famosas de C: el `else` colgante, también conocido como el problema del `dangling else`.

Mira este código sin llaves:

```
if (a > 0)
    if (b > 0)
        printf("Ambos positivos.\n");
    else
        printf("a positivo, b no.\n");
```

A que if pertenece ese else? A simple vista podría parecer que pertenece al primer if, pero no. La regla de C es clara y despiadada: un else siempre se asocia con el if más cercano que no tenga su propio else. En este ejemplo, el else pertenece al if (b > 0), al if interno. El programa dice "a positivo, b no" cuando a es positivo pero b no lo es. Si a no es positivo, no se imprime nada.

El problema es que la indentación puede mentir. Un programador podría indentar el else como si perteneciera al if externo, y el compilador lo ignoraría olímpicamente porque el compilador no lee indentación, lee reglas. Por eso la lección de oro: usa llaves siempre, incluso cuando el bloque tiene una sola línea. Las llaves eliminan toda ambigüedad y hacen tu código a prueba de confusiones. Es una de las buenas prácticas que veremos al final de la explicación.

Los if anidados son útiles, pero tienen un costo: cada nivel de anidamiento hace el código más difícil de leer. Tres o cuatro niveles de if dentro de if se convierten en una pesadilla que los programadores llaman "código espagueti" o "la pirámide de la perdición", porque el código se va recorriendo hacia la derecha formando una pirámide. La solución elegante son las guard clauses, cláusulas de guarda, que veremos en buenas prácticas: en lugar de anidar, verificas las condiciones de error primero y sales temprano.

La sentencia switch: decisiones con muchas opciones claras

Cuando tienes una variable y quieres compararla contra muchos valores posibles, una escalera larga de else if se vuelve tediosa. Para esos casos existe switch, diseñado exactamente para eso: tomar una expresión, normalmente un entero o un carácter, y saltar directamente al caso que coincida.

```
switch (expresion) {  
    case valor1:  
        // codigo si expresion == valor1  
        break;  
    case valor2:  
        // codigo si expresion == valor2  
        break;  
    default:  
        // codigo si no coincidio ningun caso  
        break;  
}
```

Cómo funciona: C evalúa la expresión una sola vez y luego busca un case cuyo valor coincida. Cuando lo encuentra, empieza a ejecutar desde ahí hacia abajo. La palabra break es crucial: rompe el switch y salta hasta después de su llave de cierre. Si olvidas el break, la ejecución continua con el siguiente case aunque su valor no coincida, en un comportamiento llamado fall-through o caída. A veces el fall-through se usa a propósito, pero el 99 por ciento de las veces olvidarlo es un error.

El default es opcional y funciona como el else: se ejecuta cuando ningún case coincidio. Es buena práctica incluirlo siempre, aunque sea para manejar el caso inesperado.

Ejemplo: día de la semana.

```

int dia;
printf("Dame un numero del 1 al 7: ");
scanf("%d", &dia);

switch (dia) {
    case 1:
        printf("Lunes.\n");
        break;
    case 2:
        printf("Martes.\n");
        break;
    case 3:
        printf("Miercoles.\n");
        break;
    case 4:
        printf("Jueves.\n");
        break;
    case 5:
        printf("Viernes.\n");
        break;
    case 6:
        printf("Sabado.\n");
        break;
    case 7:
        printf("Domingo.\n");
        break;
    default:
        printf("Numero invalido.\n");
        break;
}

```

El diagrama de flujo del switch es como una central de trenes: el flujo llega, se evalúa la expresión una vez, y luego hay una vía para cada case que lleva directo a su bloque. Cada bloque termina con un break que devuelve el flujo a la vía principal después del switch. Sin break, el tren sigue de largo a la siguiente vía sin detenerse, ejecutando código que no le correspondía.

Cuando conviene switch y cuando conviene if? Usa switch cuando compares una sola variable contra valores constantes, discretos y conocidos: números enteros, caracteres, opciones de menú. Usa if cuando las condiciones sean rangos ($\text{edad} \geq 18$), combinaciones lógicas complejas ($a > 0 \ \&\& \ b < 10$), o comparaciones con variables. El switch solo acepta valores constantes en los case, no puedes poner case ($x > 5$): ni rangos. Esa es su limitación principal.

Una pregunta frecuente es si el switch es más rápido que una escalera de if. En teoría, el compilador puede optimizar un switch con muchos casos usando una tabla de saltos, que brinca directo al caso correcto en lugar de comparar uno por uno. En la práctica, con los tamaños de programa que vas a manejar ahora, la diferencia es insignificante. Elige switch o if por claridad, no por velocidad: el código que se entiende fácil es el código que se mantiene fácil. La optimización prematura es una de las trampas mentales más comunes en programación; primero que funcione y se entienda, después, si de verdad hace falta, que corra más rápido.

Un truco legítimo del fall-through: agrupar varios case que comparten el mismo código.

```
char vocal;
printf("Dame una letra: ");
scanf(" %c", &vocal);

switch (vocal) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        printf("Es vocal.\n");
        break;
    default:
        printf("No es vocal.\n");
        break;
}
```

Aquí los cinco case caen uno tras otro hasta llegar al printf, porque ninguno tiene break hasta el final del grupo. Es la forma idiomática de decir "si es cualquiera de estos valores". Fíjate también en el espacio antes de %c en el scanf: ese espacio le dice a scanf que ignore espacios en blanco previos, incluyendo el Enter que quedó en el bufer de una lectura anterior. Es un detalle que te ahorra dolores de cabeza.

El operador ternario: decisiones en una línea

El operador ternario es el único operador de C que toma tres operandos, de ahí su nombre. Es una forma compacta de escribir un if-else que asigna un valor o produce un resultado simple. Su sintaxis:

```
condicion ? valor_si_verdadero : valor_si_falso;
```

Se lee así: evalúa la condición; si es verdadera, toda la expresión vale `valor_si_verdadero`; si es falsa, vale `valor_si_falso`. Ejemplo: obtener el mayor de dos números.

```
int a = 10, b = 20;
int mayor = (a > b) ? a : b;
printf("El mayor es %d.\n", mayor);
```

Esto equivale a:

```
int mayor;
if (a > b) {
    mayor = a;
} else {
    mayor = b;
}
```

El ternario brilla en decisiones cortas y simples: asignar un valor según una condición, elegir entre dos cadenas para imprimir, calcular un ajuste. Pero tiene una regla de uso sensata: si el ternario se vuelve difícil de leer, usa un `if` normal. Anidar ternarios uno dentro de otro es técnicamente posible pero es una mala idea casi siempre, porque el código se vuelve un acertijo. La legibilidad manda.

Otro uso común: imprimir mensajes condicionales directamente.

```
int edad = 20;
printf("Eres %s.\n", (edad >= 18) ? "mayor de edad" : "menor de edad");
```

Aquí el ternario elige entre dos cadenas y `printf` imprime la elegida. Limpio, directo, en una línea.

Patrones de decisión que verás en todas partes

Con la experiencia vas a notar que las decisiones en los programas reales se repiten en unos cuantos patrones. Conocerlos te ahorra tiempo porque en lugar de inventar la lógica desde cero, reconoces la situación y aplicas el molde.

El patrón de validación ya lo usaste en varios ejemplos: pedir un dato, verificar que tenga sentido y quejarse si no lo tiene. La forma profesional es validar de inmediato, justo después de leer el dato, antes de hacer cualquier cálculo con él. Así los errores se atrapan en la entrada y no contaminan el resto del programa. Todo programa serio valida sus entradas; los que no lo hacen son los que se caen con el primer usuario distraído.

El patrón de bandera usa una variable como interruptor. Por ejemplo, quieres saber si un número tiene alguna característica después de revisarlo por partes: creas `int encontrado = 0`; y cuando la encuentras pones `encontrado = 1`; Al final preguntas `if (encontrado)` para decidir que mensaje mostrar. La bandera recuerda que algo paso, aunque haya pasado hace muchas líneas. Es una técnica sencilla y poderosísima que verás una y otra vez, sobre todo cuando combines decisiones con ciclos en la próxima unidad.

El patrón de menu es el del ejemplo 8: mostrar opciones numeradas, leer la elección y repartir con `switch`. Casi todo programa interactivo de consola usa este molde. La variante con caracteres en lugar de números también es común: 'c' para continuar, 's' para salir. Cuando el menu crezca mucho, cada case suele llamar a una función, tema de la unidad 5, para que el `switch` no se vuelva gigante.

El patrón de clasificación por rangos es el de las calificaciones y el IMC: una escalera `else if` donde cada peldano atrapa un intervalo. La clave está en que los intervalos no se traslapen ni dejen huecos, y en probar siempre los valores frontera. Este patrón aparece en sistemas de calificaciones, niveles de videojuegos, tarifas por consumo, rangos de impuestos y cientos de aplicaciones más.

El patrón de confirmación protege las acciones peligrosas: antes de borrar o salir, preguntas "Estás seguro? (s/n)" y solo procedes si la respuesta es afirmativa. Es un `if` simple, pero evita desastres. Acostumbrate a pedir confirmación cuando tu programa vaya a hacer algo irreversible.

Reconocer estos patrones es cómo aprender jugadas en un deporte: al principio las ejecutas pensando cada paso, con la práctica salen solas. Cuando leas código ajeno, intenta identificar que patrón usa cada decisión; es la forma más rápida de entender programas que no escribiste tu.

Técnica de escritorio: prueba tu lógica en papel

Antes de compilar, los buenos programadores prueban su lógica en la cabeza o en papel con una técnica llamada prueba de escritorio o desk check. La idea es simple: tomas tu código y lo ejecutas tú mismo, línea por línea, anotando el valor de cada variable en una tabla, con un juego de datos de prueba que tu eliges.

Supon este fragmento y el dato de entrada `edad = 17`:


```
if (edad >= 18) {  
    printf("Mayor.\n");  
} else if (edad >= 13) {  
    printf("Adolescente.\n");  
} else {  
    printf("Nino.\n");  
}
```

Tu tabla mental dice: edad vale 17. Primera condición: $17 \geq 18$ es falso, no se ejecuta. Segunda: $17 \geq 13$ es verdadero, se imprime "Adolescente" y se salta el resto. Listo, el programa hace lo esperado con ese dato. Ahora prueba con 18, con 12, con 13 y con un valor absurdo como -5. Cada prueba tarda segundos y te revela errores de lógica antes de que existan en la máquina.

La prueba de escritorio brilla especialmente con condiciones compuestas y anidamientos, donde es fácil perder el hilo. Elige tus datos con intención: un valor típico de cada rama, los valores frontera donde una rama cambia a otra, y al menos un valor inválido o extremo. Si tu lógica sobrevive a esos casos en papel, casi seguro sobrevive en la computadora.

Combina esta técnica con compilaciones frecuentes: escribe un poco, prueba en papel, compila, ejecuta con tus datos de prueba. Los principiantes suelen escribir cien líneas y compilar al final, y luego no saben dónde está el error entre esas cien líneas. Los profesionales avanzan en pasos cortos y verifican cada paso. Adoptar ese ritmo desde ahora te va a ahorrar frustraciones enormes.

Decisiones con caracteres y cadenas de texto

Hasta ahora decidimos con números, pero los programas reales también deciden con texto: una letra de respuesta, una opción de menú, un nombre de usuario. Aquí hay que distinguir dos casos, porque en C los caracteres y las cadenas se manejan muy distinto.

Los caracteres individuales son fáciles: un char es en el fondo un número entero, el código ASCII del símbolo. Por eso puedes compararlos directamente con `==`, `<` o `>` dentro de un `if` o un `switch`. Las letras mayúsculas y minúsculas tienen códigos distintos: 'A' no es igual a 'a'. Un patrón muy común es preguntar si o no:

```

char respuesta;
printf("Quieres continuar? (s/n): ");
scanf(" %c", &respuesta);

if (respuesta == 's' || respuesta == 'S') {
    printf("Continuamos.\n");
} else {
    printf("Nos vemos.\n");
}

```

Fíjate en dos detalles. Primero, el carácter va entre comillas simples: 's' es un carácter, "s" es una cadena. Confundirlas es un error típico. Segundo, aceptamos tanto 's' como 'S' con el operador ||, porque el usuario puede escribir la letra en mayúsculas. Para no duplicar condiciones, puedes convertir la letra con las funciones tolower o toupper de la cabecera ctype.h, que convierten un carácter a minúsculas o mayúsculas:

```

#include <ctype.h>
/* ... */
respuesta = tolower(respuesta);
if (respuesta == 's') {
    printf("Continuamos.\n");
}

```

Así solo comparas contra una versión. Es un truco simple que hace tus programas más amables con el usuario.

Las cadenas completas son otro animal. En C una cadena es un arreglo de caracteres, y el nombre del arreglo es en realidad la dirección de memoria donde empieza. Por eso `if (nombre == "Juan")` no compara el contenido, compara direcciones, y casi siempre da falso aunque el texto sea idéntico. La forma correcta es la función `strcmp` de `string.h`, que devuelve 0 cuando las dos cadenas son iguales:

```
#include <string.h>
/* ... */
char usuario[50];
printf("Nombre de usuario: ");
scanf("%49s", &usuario[0]);

if (strcmp(usuario, "admin") == 0) {
    printf("Bienvenido, administrador.\n");
} else {
    printf("Hola, %s.\n", usuario);
}
```

La condición `strcmp(usuario, "admin") == 0` se lee: "si la comparación da cero, o sea, si son iguales". Al principio se siente al revés, pero te acostumbras rápido. También existe `strncmp`, que compara solo los primeros N caracteres, útil cuando solo te importa el inicio del texto.

Un último consejo para decisiones con texto: cuando leas cadenas con `scanf` y `%s`, la lectura se detiene en el primer espacio, así que "Juan Perez" quedaría solo como "Juan". Para leer líneas completas con espacios se usa `fgets`, que verás en la unidad de cadenas. Por ahora, con `%s` y el límite de tamaño como `%49s` tienes suficiente para los ejercicios de esta unidad.

El if de una sola línea: por que los libros viejos omiten las llaves

Si hojeas libros o tutoriales viejos de C, verás muchos `if` sin llaves cuando el bloque tiene una sola instrucción:

```
if (edad >= 18)
    printf("Mayor de edad.\n");
```

Esto es válido en C: sin llaves, el `if` controla únicamente la siguiente sentencia. Funciona y es más corto. Entonces, por que en esta unidad insistimos en usar llaves siempre? Por tres razones prácticas que pesan más que ahorrarse dos caracteres.

Primera: el mantenimiento. Hoy el bloque tiene una línea, pero mañana querrás agregar una segunda, algo así como registrar el acceso en un archivo. Si agregas la línea sin poner llaves, solo la primera sentencia queda dentro del `if` y la segunda se ejecuta siempre. Es un error facilísimo de cometer y difícilísimo de detectar a simple vista, porque la indentación sugiere que ambas están dentro.

Segunda: la claridad visual. Las llaves delimitan exactamente donde empieza y termina cada bloque, sin que tengas que contar sentencias ni adivinar. Cuando lees código ajeno, o tu propio código de hace meses, esa delimitación explícita vale oro.

Tercera: la trampa del else colgante, que ya viste. Sin llaves, en los anidamientos el else se pega al if más cercano y la indentación puede mentirte. Con llaves, la asociación es explícita y no hay lugar a dudas.

Hay programadores experimentados que omiten las llaves en casos triviales y les funciona, porque tienen disciplina ferrea. Pero mientras estás aprendiendo, y en cualquier código que otras personas vayan a leer, las llaves siempre son la decisión correcta. Es como el cinturón de seguridad: los expertos también se lo ponen.

Buenas prácticas al tomar decisiones

Primera y más importante: usa llaves siempre, incluso para bloques de una sola línea. Ya viste la trampa del else colgante. Las llaves cuestan dos caracteres y te ahorran horas de depuración. Además, si algún día agregas una segunda línea al bloque y olvidas poner las llaves, el código se rompe de formas misteriosas. Hazlo un hábito desde hoy.

Segunda: evita el anidamiento profundo con guard clauses. En lugar de meter if dentro de if dentro de if, verifica las condiciones de salida primero y maneja los casos especiales al inicio. Compara:

```
/* Anidado y difícil de leer */
if (edad >= 0) {
    if (edad < 130) {
        if (edad >= 18) {
            printf("Puedes votar.\n");
        }
    }
}
```

```

/* Con guard clauses: plano y claro */
if (edad < 0 || edad >= 130) {
    printf("Edad invalida.\n");
    return 1;
}
if (edad < 18) {
    printf("Aun no puedes votar.\n");
    return 0;
}
printf("Puedes votar.\n");

```

La segunda versión se lee de arriba hacia abajo sin tener que rastrear niveles. Cada if maneja un caso y sale. El camino feliz, el caso normal, queda al final sin indentación extra. Este estilo lo usan los profesionales y hace tu código mucho más fácil de mantener.

Tercera: ordena las condiciones de la más específica a la más general en las escaleras else if, como ya vimos con las calificaciones.

Cuarta: en el switch, no olvides el break y considera incluir siempre el default, aunque sea para reportar un valor inesperado. Un default que diga "opción invalida" convierte un error silencioso en un mensaje claro.

Quinta: usa el ternario solo para decisiones simples. Si necesitas más de una línea para entenderlo, es señal de que un if es mejor.

Sexta: escribe condiciones en forma positiva cuando puedas. if (esValido) se lee mejor que if (!esInvalido). Las negaciones dobles confunden a cualquiera.

Errores comunes que debes evitar

El error número uno de los principiantes: el punto y coma después del if.

```

int edad = 15;
if (edad >= 18);    /* <--- este punto y coma es un error */
{
    printf("Eres mayor de edad.\n");
}

```

Ese punto y coma termina la sentencia if ahí mismo: el if queda con un cuerpo vacío. El bloque entre llaves que sigue ya no pertenece al if, es un bloque independiente que siempre se ejecuta. Resultado: el programa imprime "Eres mayor de edad" aunque la edad sea 15. El compilador no se queja porque es código válido. Es un error silencioso y traicionero. Solución:

nunca pongas punto y coma después del paréntesis del if, del else, del for ni del while en la línea de la condición.

El error número dos: confundir = con ==.

```
int x = 5;
if (x = 10) { /* <--- asignacion, no comparacion */
    printf("x vale 10.\n");
}
```

Aquí no estás preguntando si x vale 10, estás asignando 10 a x. La expresión x = 10 vale 10, que es distinto de cero, así que la condición siempre es verdadera. El programa siempre entra al if y además x cambio de valor sin que te dieras cuenta. Algunos compiladores modernos te avisan con un warning, y como viste en la Unidad 1, con -Wall puedes activar y leer esos warnings. Por ahora, cada vez que escribas una condición, pregúntate: quiero comparar (==) o asignar (=)?

El error número tres: olvidar el break en el switch. Sin break, la ejecución cae al siguiente case y ejecuta código que no correspondía. Si tu menú imprime dos opciones en lugar de una, casi seguro olvidaste un break.

El error número cuatro: condiciones que nunca pueden ser verdaderas por contradicción lógica, como if (edad > 18 && edad < 10). Ningún número es mayor que 18 y menor que 10 a la vez. Lee tus condiciones en voz alta: "si la edad es mayor que 18 Y menor que 10". Suena mal porque está mal.

El error número cinco: comparar números de punto flotante con ==. Por la forma en que las computadoras guardan los decimales, 0.1 + 0.2 no es exactamente igual a 0.3 en binario. En lugar de if (precio == 19.99), compara con un margen: if (fabs(precio - 19.99) < 0.001). Es un detalle avanzado pero te ahorra sorpresas raras.

El error número seis: el else colgante que ya vimos. Solución: llaves siempre.

El error número siete: comparar cadenas con == en lugar de strcmp. if (color == "rojo") compara direcciones de memoria, no el texto, y el programa se comporta como si nunca coincidieran. Para caracteres sueltos si vale ==, como respuesta == 's'; para cadenas completas, strcmp. Si tu comparación de texto "nunca funciona", este es el primer sospechoso.

El error número ocho: leer un carácter con scanf("%c") justo después de leer un número, sin el espacio antes de %c. El scanf se traga el Enter que quedó en el bufer y tu pregunta parece saltarse sola. Solución: escribe " %c", con el espacio. Parece magia, pero es solo el bufer haciendo de las suyas.

Con estas herramientas y estas precauciones, ya puedes escribir programas que deciden. En la siguiente sección lo vas a practicar con ejemplos paso a paso, desmenuzando cada uno para que no quede ninguna duda. Recuerda que en lucasegaacademy.com tienes más material y cursos gratuitos para seguir practicando estos temas hasta dominarlos.

Ejemplos paso a paso

En esta sección vamos a desmenuzar ocho ejemplos completos, del más simple al más elaborado. Cada uno incluye el código, una explicación línea por línea de lo importante y un análisis de que camino toma el programa con datos de ejemplo. Escríbelos tú mismo en tu editor, compílalos con gcc y pruébalos con distintos valores. Leer código sirve, pero escribirlo y romperlo es lo que de verdad enseña.

Un consejo antes de empezar: compila siempre con los warnings activados, así:

```
gcc -Wall programa.c -o programa
```

La bandera -Wall le pide al compilador que te avise de construcciones sospechosas, como una posible confusión entre = y == en una condición. Los warnings no impiden que el programa compile, pero son el compilador tratando de ayudarte. Ya viste en la Unidad 1 los warnings a fondo; por ahora, acostúmbrate a compilar con -Wall y a leer lo que te dice. Un programa que compila limpio, sin warnings, es un programa más confiable.

Ejemplo 1: Mayor de edad

El programa más clásico para estrenar el if-else: pedir la edad y decir si la persona es mayor de edad o no.

```
#include <stdio.h>

int main() {
    int edad;

    printf("Cuantos años tienes? ");
    scanf("%d", &edad);

    if (edad >= 18) {
        printf("Eres mayor de edad.\n");
    } else {
        printf("Eres menor de edad.\n");
    }

    return 0;
}
```

Que está pasando aquí? Declaramos la variable edad y la leemos con scanf. Luego viene la decisión: la condición edad >= 18 se evalúa. Si el usuario escribió 21, la expresión vale 1 (verdadero) y se ejecuta el bloque del if: imprime "Eres mayor de edad". Si escribió 15, la expresión vale 0 (falso) y se ejecuta el bloque del else. Observa que exactamente una de las dos ramas se ejecuta siempre. No hay forma de que se impriman ambos mensajes ni ninguno: el else garantiza la cobertura total de los dos casos posibles.

Pruebalo con 18 justo: la condición es 18 >= 18, que es verdadera, así que dice mayor de edad. Los límites siempre merecen una prueba: 17 y 18 son los valores frontera de este programa.

Ejemplo 2: Calificaciones con else if

Ahora una escalera de decisiones: convertir una calificación numérica en una palabra.


```

#include <stdio.h>

int main() {
    float calificacion;

    printf("Dame tu calificacion (0-100): ");
    scanf("%f", &calificacion);

    if (calificacion >= 90) {
        printf("Excelente, sacaste A.\n");
    } else if (calificacion >= 80) {
        printf("Muy bien, sacaste B.\n");
    } else if (calificacion >= 70) {
        printf("Bien, sacaste C.\n");
    } else if (calificacion >= 60) {
        printf("Apenas pasaste, sacaste D.\n");
    } else {
        printf("Reprobado, sacaste F.\n");
    }

    return 0;
}

```

Sigue el flujo con un 85. Primera condición: $85 \geq 90$? Falso. Segunda: $85 \geq 80$? Verdadero. Se imprime "Muy bien, sacaste B" y el programa salta todo lo demás, incluyendo el else final. Ahora con un 95: la primera condición ya es verdadera, imprime "Excelente" y sale. Con un 40: todas las condiciones son falsas y cae al else final: "Reprobado".

Fíjate en el orden: de la más exigente a la menos exigente. Si pusieras primero calificación ≥ 60 , un 95 entraría ahí y diría "Apenas pasaste", lo cual sería un error lógico aunque el programa compile perfecto. El orden de las condiciones es parte de la lógica, no un detalle estético.

Nota que usamos float para la calificación, por si alguien saca 89.5. Con int perderíamos los decimales. Los operadores relacionales funcionan igual con flotantes.

Ejemplo 3: Par o impar con validación

Vamos a combinar decisiones: primero validamos que el número sea positivo y luego decidimos si es par o impar. Aquí aparecen los if anidados bien usados, con llaves claras.

```
#include <stdio.h>

int main() {
    int numero;

    printf("Dame un numero entero positivo: ");
    scanf("%d", &numero);

    if (numero < 0) {
        printf("Error: el numero debe ser positivo.\n");
    } else {
        if (numero % 2 == 0) {
            printf("El %d es par.\n", numero);
        } else {
            printf("El %d es impar.\n", numero);
        }
    }

    return 0;
}
```

El primer if es una guarda: si el número es negativo, muestra el error y no sigue analizando. Solo cuando el número pasa esa validación entramos al else, donde vive el segundo if que decide par o impar. Las llaves dejan clarísimo que el else interno pertenece al if interno ($\text{numero} \% 2 == 0$) y el else externo al if externo ($\text{numero} < 0$). Sin llaves, el else colgante nos jugaría una mala pasada.

Pruebalo con -4: entra al primer if, imprime el error y termina. Con 7: el primer if es falso, entra al else, el if interno evalúa $7 \% 2 == 0$, que es falso, e imprime "El 7 es impar". Con 0: $0 \% 2$ es 0, así que dice que es par. Correcto: el cero es par.

Este patrón de validar primero y procesar después lo vas a usar en casi todos tus programas serios. Acostúmbrate a él.

Ejemplo 4: Calculadora con switch

Una calculadora simple es el ejemplo perfecto para el switch: el usuario elige una operación con un carácter y el programa salta directo al caso correspondiente.

```

#include <stdio.h>

int main() {
    float a, b, resultado;
    char operacion;

    printf("Dame el primer numero: ");
    scanf("%f", &a);
    printf("Dame el segundo numero: ");
    scanf("%f", &b);
    printf("Que operacion? (+, -, *, /): ");
    scanf(" %c", &operacion);

    switch (operacion) {
        case '+':
            resultado = a + b;
            printf("%.2f + %.2f = %.2f\n", a, b, resultado);
            break;
        case '-':
            resultado = a - b;
            printf("%.2f - %.2f = %.2f\n", a, b, resultado);
            break;
        case '*':
            resultado = a * b;
            printf("%.2f * %.2f = %.2f\n", a, b, resultado);
            break;
        case '/':
            if (b != 0) {
                resultado = a / b;
                printf("%.2f / %.2f = %.2f\n", a, b, resultado);
            } else {
                printf("Error: no se puede dividir entre cero.\n");
            }
            break;
        default:
            printf("Operacion no valida.\n");
            break;
    }

    return 0;
}

```

Hay varias cosas que notar. Primero, el switch evalúa operación una vez y salta al case que coincida. Si el usuario escribió '*', ejecuta ese bloque y el break lo saca del switch. Sin ese break, caería al case '/' y haría un desastre.

Segundo, dentro del case '/' hay un if anidado que protege contra la división entre cero. Es un ejemplo hermoso de como las estructuras se combinan: el switch elige la operación y el if valida que sea segura. La división entre cero con flotantes no truená el programa como con enteros, pero el resultado no tiene sentido, así que la validamos de todos modos.

Tercero, el default atrapa cualquier carácter raro que el usuario escriba, como '%' o 'x', y muestra un mensaje claro en lugar de no hacer nada en silencio.

Cuarto, el espacio en " %c" del scanf. Sin ese espacio, scanf leería el carácter de nueva línea que quedó en el bufer cuando el usuario presiono Enter después del segundo número, y el programa parecería saltarse la pregunta. Ese espacio le dice a scanf: "ignora los espacios en blanco y dame el primer carácter de verdad". Grabatelo, lo vas a necesitar cada vez que leas un carácter después de leer números.

Ejemplo 5: Día de la semana con mensajes especiales

Vamos a darle más vida al switch: no solo nombramos el día, también damos un mensaje según si es día laboral o fin de semana, usando el fall-through a propósito.

```

#include <stdio.h>

int main() {
    int dia;

    printf("Dame un numero del 1 al 7: ");
    scanf("%d", &dia);

    switch (dia) {
        case 1:
            printf("Lunes: a darle con todo.\n");
            break;
        case 2:
            printf("Martes: sigue el ritmo.\n");
            break;
        case 3:
            printf("Miercoles: ya vas a la mitad.\n");
            break;
        case 4:
            printf("Jueves: casi es viernes.\n");
            break;
        case 5:
            printf("Viernes: ultimo esfuerzo.\n");
            break;
        case 6:
        case 7:
            printf("Fin de semana: a descansar.\n");
            break;
        default:
            printf("Ese dia no existe.\n");
            break;
    }

    return 0;
}

```

El truco está en case 6: seguido inmediatamente de case 7:, sin break entre ellos. Si día vale 6, el programa entra en case 6, no encuentra código ni break, cae a case 7 y ejecuta el printf del fin de semana. Si día vale 7, entra directo a case 7. Ambos días comparten el mensaje. Este es el uso legítimo del fall-through: agrupar casos. Pero ojo, funciona porque lo hicimos a propósito y esta clarísimo. Un break olvidado por accidente se ve igual y hace dano igual. Cuando uses fall-through intencional, un comentario como */* cae a propósito */* ayuda a quien lea tu código, incluso si ese lector eres tu dentro de seis meses.

Ejemplo 6: Validación de rangos con operadores lógicos

Un programa que pide la edad y la clasifica en rangos: niño, adolescente, adulto, adulto mayor. Aquí las condiciones compuestas con `&&` hacen el trabajo pesado.

```
#include <stdio.h>

int main() {
    int edad;

    printf("Dame tu edad: ");
    scanf("%d", &edad);

    if (edad < 0 || edad > 130) {
        printf("Esa edad no parece valida.\n");
    } else if (edad < 13) {
        printf("Eres un niño.\n");
    } else if (edad >= 13 && edad < 18) {
        printf("Eres adolescente.\n");
    } else if (edad >= 18 && edad < 65) {
        printf("Eres adulto.\n");
    } else {
        printf("Eres adulto mayor.\n");
    }

    return 0;
}
```

La primera condición es la guarda: edades imposibles se rechazan de entrada con el operador `||`, porque basta que una de las dos sea verdadera para que la edad sea invalida. Después, cada `else if` define un rango. Fíjate en cómo se combinan los límites: `edad >= 13 && edad < 18` cubre del 13 al 17 sin huecos ni traslapes con los rangos vecinos. Los rangos quedan así: 0-12 niño, 13-17 adolescente, 18-64 adulto, 65-130 adulto mayor. Cada edad válida cae exactamente en una categoría.

Prueba los valores frontera: 12, 13, 17, 18, 64, 65. Cada uno debe caer en la categoría correcta. Probar fronteras es un hábito de programador profesional: los errores casi siempre se esconden en los límites.

También nota el cortocircuito en acción: en `edad >= 13 && edad < 18`, si la edad es 8, la primera parte es falsa y C ni siquiera evalúa la segunda. No cambia el resultado aquí, pero en condiciones con llamadas a funciones o accesos a datos puede evitar errores reales.

Ejemplo 7: Operador ternario para el máximo y mensajes

El ternario en su habitat natural: decisiones cortas que producen un valor.

```
#include <stdio.h>

int main() {
    int a, b;

    printf("Dame dos numeros enteros: ");
    scanf("%d %d", &a, &b);

    int mayor = (a > b) ? a : b;
    int menor = (a < b) ? a : b;

    printf("El mayor es %d y el menor es %d.\n", mayor, menor);
    printf("Los numeros son %s.\n", (a == b) ? "iguales" : "diferentes");

    return 0;
}
```

La línea `int mayor = (a > b) ? a : b;` hace en una línea lo que un `if-else` haría en cinco. Se lee: "mayor vale a si a es mayor que b; si no, vale b". Lo mismo para el menor. Y en el `printf` usamos el ternario directamente como argumento: elige entre las cadenas "iguales" y "diferentes" según `a == b`.

Qué pasa si a y b son iguales, digamos 7 y 7? `mayor = (7 > 7) ? 7 : 7` da 7, correcto. menor igual. Y el mensaje dice "iguales". Todo consistente. El ternario maneja bien el caso de igualdad porque ambas ramas dan el mismo valor.

Los paréntesis alrededor de la condición son opcionales pero recomendados: hacen clarísimo donde termina la condición y empieza el `?`. Con el tiempo los vas a poner por instinto.

Ejemplo 8: Menu interactivo completo

El ejemplo final combina todo: un menu con `switch`, validación con `if`, y el programa decide que hacer según la opción. Es el tipo de programa que vas a escribir una y otra vez.

```

#include <stdio.h>

int main() {
    int opcion;
    float celsius, fahrenheit;

    printf("=== CONVERSION DE TEMPERATURA ===\n");
    printf("1. Celsius a Fahrenheit\n");
    printf("2. Fahrenheit a Celsius\n");
    printf("Elige una opcion: ");
    scanf("%d", &opcion);

    if (opcion != 1 && opcion != 2) {
        printf("Opcion invalida. Adios.\n");
        return 1;
    }

    printf("Dame la temperatura: ");

    switch (opcion) {
        case 1:
            scanf("%f", &celsius);
            fahrenheit = celsius * 9.0 / 5.0 + 32;
            printf("%.2f C son %.2f F.\n", celsius, fahrenheit);
            break;
        case 2:
            scanf("%f", &fahrenheit);
            celsius = (fahrenheit - 32) * 5.0 / 9.0;
            printf("%.2f F son %.2f C.\n", fahrenheit, celsius);
            break;
    }

    printf("Gracias por usar el conversor.\n");
    return 0;
}

```

El programa primero muestra el menu y lee la opción. Luego viene la guarda: if (opción != 1 && opción != 2) detecta cualquier opción invalida, muestra el mensaje y termina el programa con return 1. Ese 1 distinto de cero le dice al sistema operativo que algo salio mal, es una convención que verás en todos lados.

Gracias a esa guarda, el switch ya no necesita un default: es imposible llegar ahí con una opción invalida. El switch solo maneja los dos casos válidos, cada uno con su fórmula. Al final, el mensaje de despedida se imprime siempre, sin importar que opción se eligio.

Fíjate en un detalle de las fórmulas: escribimos $9.0 / 5.0$ en lugar de $9 / 5$. Con enteros, $9 / 5$ da 1 por la división entera, y la conversión saldría mal. Con 9.0 y 5.0 forzamos división de flotantes. Es un error clásico que ya debería estar en tu radar desde las unidades anteriores, pero aquí aparece en contexto real.

Este programa es un mini sistema completo: presenta opciones, válida la entrada, procesa según la decisión y reporta el resultado. Si entiendes este ejemplo a fondo, entiendes la unidad. Modifícalo: agrega una tercera opción para convertir a Kelvin, o un mensaje distinto según si la temperatura está bajo cero. Jugar con el código es la mejor forma de aprender.

Práctica

Es hora de que tomes el teclado. Los siguientes ejercicios están ordenados de menor a mayor dificultad y cubren todo lo que vimos: if, else if, else, switch, ternario, validaciones y anidamiento. Intenta resolver cada uno por tu cuenta antes de mirar la solución. Si te atorras más de veinte minutos, mira la solución, entiéndela a fondo y luego escríbela tú mismo sin copiar. Esa es la forma en que el conocimiento se queda.

Una estrategia que funciona: antes de escribir código, escribe en español los pasos de tu programa, como una receta. "Pedir la edad. Si es menor que cero o mayor que 130, mostrar error y terminar. Si es menor que 12, el precio es 50..." Ese texto es tu algoritmo en lenguaje natural, y traducirlo a C después es casi mecánico. Los principiantes que se lanzan directo al código suelen enredarse; los que planean primero avanzan más rápido aunque parezca que van más lento. Además, prueba cada ejercicio con al menos tres datos: uno de cada camino posible del programa. Si tu programa tiene tres ramas, necesitas mínimo tres pruebas. Esa disciplina de probar todos los caminos es lo que separa al que "cree que funciona" del que "sabe que funciona".

1. Semaforo: pide al usuario un color ("verde", "amarillo" o "rojo") y muestra la acción correspondiente: "Avanza", "Precaution: frena con cuidado" o "Detente". Si escribe otra cosa, muestra "Color no válido". Resuélvelo una vez con if-else if y otra vez con switch sobre el primer carácter. Compara cuál versión te parece más clara.
2. Descuento por edad en el cine: el boleto cuesta 100 pesos. Los niños menores de 12 pagan la mitad, los adultos mayores de 60 o más pagan 70 pesos, y el resto paga completo. Pide la edad, válida que esté entre 0 y 130, y muestra cuánto debe pagar. Usa una escalera else if.
3. Calculadora de IMC: pide el peso en kilos y la estatura en metros, calcula el índice de masa corporal con la fórmula $\text{peso} / (\text{estatura} * \text{estatura})$ y clasifica el resultado: menos de

18.5 "Bajo peso", 18.5 a 24.9 "Normal", 25 a 29.9 "Sobrepeso", 30 o más "Obesidad". Valida que el peso y la estatura sean positivos antes de calcular.

4. Adivina el número secreto: el programa guarda un número secreto del 1 al 10 (tu eliges cuál en el código). Pide al usuario que lo adivine y responde con una de tres opciones: "Adivinaste", "El secreto es mayor" o "El secreto es menor". Usa if-else if-else.
5. Cajero automatico simplificado: muestra un menu con switch: 1) Consultar saldo (empieza con 5000 pesos), 2) Retirar (pide la cantidad y verifica con un if que haya fondos suficientes), 3) Depositar (pide la cantidad y la suma), 4) Salir. Si la opción no existe, el default muestra "Opción invalida". Usa una variable para el saldo y actualizala según la operación.
6. El reto final: conversor de calificaciones mixto. Pide una calificación del 0 al 100 y haz todo esto: válida el rango; con un switch sobre calificación / 10 asigna la letra (A, B, C, D, F); con un ternario decide el mensaje "Aprobado" o "Reprobado" (se aprueba con 60 o más); y con un if muestra "Felicidades, excelencia" solo si la calificación es 90 o más. Todo en un solo programa que combine las tres herramientas de decisión de la unidad.

Solución del ejercicio 1

```

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

int main() {
    char color[20];

    printf("Color del semaforo (verde, amarillo, rojo): ");
    scanf("%19s", &color[0]);

    /* Version con if-else if */
    if (strcmp(color, "verde") == 0) {
        printf("Avanza.\n");
    } else if (strcmp(color, "amarillo") == 0) {
        printf("Precaution: frena con cuidado.\n");
    } else if (strcmp(color, "rojo") == 0) {
        printf("Detente.\n");
    } else {
        printf("Color no valido.\n");
    }

    /* Version con switch sobre el primer caracter */
    switch (color[0]) {
        case 'v':
            printf("Avanza.\n");
            break;
        case 'a':
            printf("Precaution: frena con cuidado.\n");
            break;
        case 'r':
            printf("Detente.\n");
            break;
        default:
            printf("Color no valido.\n");
            break;
    }

    return 0;
}

```

Explicación: en C no puedes comparar cadenas con `==`, porque eso compararía las direcciones de memoria, no el contenido. La función `strcmp` de `string.h` compara dos cadenas carácter por carácter y devuelve 0 cuando son iguales. Por eso la condición es `strcmp(color, "verde") == 0`. Es un detalle que confunde a todos al principio, así que grabatelo: cadenas se comparan con `strcmp`.

La versión con switch no puede comparar cadenas completas porque switch solo trabaja con enteros y caracteres, así que usamos el primer carácter: 'v' de verde, 'a' de amarillo, 'r' de rojo. Funciona para este caso porque cada color empieza con letra distinta, pero es un truco con límites: si agregaras "violeta", chocaría con "verde". La versión con if y strcmp es más robusta y más clara aquí. Moraleja: el switch no siempre es la mejor opción, aunque el problema parezca de "varias opciones". La herramienta correcta depende de los datos.

El %19s en el scanf limita la lectura a 19 caracteres para no desbordar el arreglo de 20 (se deja espacio para el terminador '\0'). Es una buena práctica de seguridad que vale la pena adoptar desde ya.

Solución del ejercicio 2

```
#include <stdio.h>

int main() {
    int edad;
    float precio;

    printf("Dame tu edad: ");
    scanf("%d", &edad);

    if (edad < 0 || edad > 130) {
        printf("Edad no valida.\n");
        return 1;
    }

    if (edad < 12) {
        precio = 50.0;
    } else if (edad >= 60) {
        precio = 70.0;
    } else {
        precio = 100.0;
    }

    printf("Tu boleto cuesta $%.2f pesos.\n", precio);
    return 0;
}
```

Explicación: primero la guarda que rechaza edades imposibles y termina el programa con return 1. Después la escalera: los niños menores de 12 pagan 50, que es la mitad de 100. Los adultos de 60 o más pagan 70. El else atrapa a todos los demás, de 12 a 59, que pagan completo.

Fíjate que no hizo falta escribir `edad >= 12 && edad < 60` en la última rama: el else ya significa "todos los casos que no entraron arriba". Esa es la elegancia del else final, evita condiciones redundantes. También nota que el orden importa: si la condición de 60 o más estuviera primero no pasaría nada malo aquí porque los rangos no se traslapan, pero en general conviene pensar el orden con cuidado.

Usamos float para el precio aunque los valores sean cerrados, porque en un programa real los precios suelen tener centavos. El `%.2f` muestra dos decimales.

Solución del ejercicio 3

```

#include <stdio.h>

int main() {
    float peso, estatura, imc;

    printf("Dame tu peso en kilos: ");
    scanf("%f", &peso);
    printf("Dame tu estatura en metros: ");
    scanf("%f", &estatura);

    if (peso <= 0 || estatura <= 0) {
        printf("El peso y la estatura deben ser positivos.\n");
        return 1;
    }

    imc = peso / (estatura * estatura);

    printf("Tu IMC es %.2f: ", imc);
    if (imc < 18.5) {
        printf("Bajo peso.\n");
    } else if (imc < 25) {
        printf("Normal.\n");
    } else if (imc < 30) {
        printf("Sobrepeso.\n");
    } else {
        printf("Obesidad.\n");
    }

    return 0;
}

```

Explicación: la validación va primero, como buena guarda: si el peso o la estatura no son positivos, el cálculo no tendría sentido (y con estatura cero habría división entre cero). Solo con datos válidos calculamos el IMC.

En la escalera usamos un truco elegante: en lugar de escribir rangos completos como `imc >= 18.5 && imc < 25`, aprovechamos el orden. Si llegamos al segundo `else if`, ya sabemos que `imc` no es menor que 18.5, así que basta preguntar `imc < 25`. Cada condición solo verifica el límite superior porque el inferior quedó descartado por las condiciones anteriores. Es más corto, más claro y menos propenso a errores de traslape. Este patrón de "solo el límite superior" es muy común en clasificaciones por rangos.

El printf antes de la escalera imprime el valor del IMC sin salto de línea, y luego cada rama completa la frase. Así el número y la clasificación salen en la misma línea.

Solución del ejercicio 4

```
#include <stdio.h>

int main() {
    int secreto = 7;
    int intento;

    printf("Adivina el numero secreto (1-10): ");
    scanf("%d", &intento);

    if (intento == secreto) {
        printf("Adivinaste.\n");
    } else if (intento < secreto) {
        printf("El secreto es mayor.\n");
    } else {
        printf("El secreto es menor.\n");
    }

    return 0;
}
```

Explicación: un if-else if-else perfecto para tres resultados posibles. La primera condición detecta el acierto con ==. Si no acertaste, solo hay dos posibilidades: tu número es menor que el secreto (entonces el secreto es mayor) o es mayor (entonces el secreto es menor). El else final cubre el segundo caso sin necesidad de otra condición.

Este es el corazón de muchísimos juegos: comparar el intento contra el objetivo y dar una pista. Si quisieras hacerlo más interesante, podrías envolverlo en un ciclo while para dar varios intentos, tema de la próxima unidad. Por ahora, práctica cambiar el número secreto y probar los tres caminos: un intento igual, uno menor y uno mayor.

Nota el uso correcto de == para comparar. Si por accidente escribieras intento = secreto, estarías asignando y la condición siempre sería verdadera (a menos que el secreto fuera 0). Cada vez que compares, verifica que haya dos signos igual.

Solución del ejercicio 5

```
#include <stdio.h>

int main() {
    int opcion;
    float saldo = 5000.0;
    float cantidad;

    printf("=== CAJERO AUTOMATICO ===\n");
    printf("1. Consultar saldo\n");
    printf("2. Retirar\n");
    printf("3. Depositar\n");
    printf("4. Salir\n");
    printf("Elige una opcion: ");
    scanf("%d", &opcion);

    switch (opcion) {
        case 1:
            printf("Tu saldo es $%.2f.\n", saldo);
            break;
        case 2:
            printf("Cuanto quieres retirar? ");
            scanf("%f", &cantidad);
            if (cantidad <= 0) {
                printf("Cantidad no valida.\n");
            } else if (cantidad > saldo) {
                printf("Fondos insuficientes.\n");
            } else {
                saldo = saldo - cantidad;
                printf("Retiro exitoso. Nuevo saldo: $%.2f.\n", saldo);
            }
            break;
        case 3:
            printf("Cuanto quieres depositar? ");
            scanf("%f", &cantidad);
            if (cantidad <= 0) {
                printf("Cantidad no valida.\n");
            } else {
                saldo = saldo + cantidad;
                printf("Deposito exitoso. Nuevo saldo: $%.2f.\n", saldo);
            }
            break;
        case 4:
            printf("Gracias por tu visita.\n");
            break;
    }
}
```



```

        default:
            printf("Opcion invalida.\n");
            break;
    }

    return 0;
}

```

Explicación: el switch reparte las cuatro opciones del menu más el default para lo invalido. Dentro de los case de retirar y depositar hay if anidados que validan la cantidad: primero que sea positiva, y en el retiro además que no supere el saldo. Solo si pasa las validaciones se actualiza la variable saldo.

Este programa muestra la combinación natural de switch e if: el switch decide QUE operación hacer y el if decide SI la operación es válida. Cada herramienta en lo que mejor sabe hacer. El saldo vive en una variable que se modifica según la operación, que es exactamente cómo funcionan los sistemas reales por dentro, solo que con base de datos en lugar de una variable.

Como mejora, podrias agregar la validación de que el retiro sea multiplo de 50, como hacen los cajeros reales, usando el operador %. Te lo dejo de tarea extra.

Solución del ejercicio 6

```

#include <stdio.h>

int main() {
    int calificacion;
    char letra;

    printf("Dame tu calificacion (0-100): ");
    scanf("%d", &calificacion);

    if (calificacion < 0 || calificacion > 100) {
        printf("Calificacion no valida.\n");
        return 1;
    }

    /* Switch: la letra segun la decena */
    switch (calificacion / 10) {
        case 10:
        case 9:
            letra = 'A';
            break;
        case 8:

```

```

        letra = 'B';
        break;
    case 7:
        letra = 'C';
        break;
    case 6:
        letra = 'D';
        break;
    default:
        letra = 'F';
        break;
}

/* Ternario: aprobado o reprobado */
printf("Letra: %c. %s.\n", letra,
       (calificacion >= 60) ? "Aprobado" : "Reprobado");

/* If: mensaje especial de excelencia */
if (calificacion >= 90) {
    printf("Felicidades, excelencia.\n");
}

return 0;
}

```

Explicación: este programa usa las tres herramientas de decisión, cada una donde brilla. Primero la guarda con if valida el rango y sale si algo está mal.

El switch trabaja sobre calificación / 10, que es división entera: 95 / 10 da 9, 82 / 10 da 8, 100 / 10 da 10. Así convertimos rangos de decenas en valores discretos que el switch puede manejar. El case 10 y case 9 comparten la letra A gracias al fall-through intencional: un 100 también es A. El default atrapa todo lo de 50 hacia abajo como F. Es un truco muy usado en la práctica para clasificar por decenas.

El ternario elige entre "Aprobado" y "Reprobado" en una sola expresión dentro del printf. Corto y legible, justo el caso de uso ideal.

El if final es independiente: no forma parte de una escalera, es una decisión suelta que agrega el mensaje de excelencia solo cuando aplica. Un programa puede tener tantas decisiones independientes como necesite, no todo tiene que estar encadenado.

Si llegaste hasta aquí y el programa te quedó claro línea por línea, felicidades: dominas las decisiones en C. Lo que sigue en lucasegaacademy.com es poner todo esto en movimiento con los ciclos, donde tus decisiones se van a repetir miles de veces.

Resumen de la unidad

Respira hondo, porque avanzaste muchísimo. En esta unidad tus programas dejaron de ser una línea recta de instrucciones y aprendieron a tomar decisiones. Vamos a repasar lo esencial para que quede grabado.

La sentencia `if` es la decisión fundamental: evalúa una condición y ejecuta un bloque solo si es verdadera. Con `else` agregas el camino alternativo, y con `else if` encadenas varias condiciones en una escalera que se evalúa de arriba hacia abajo hasta encontrar la primera verdadera. El orden de las condiciones importa: de la más específica a la más general.

Aprendiste la regla de oro de la verdad en C: el cero es falso y todo lo demás es verdadero. Los operadores relacionales devuelven 1 o 0, y los operadores lógicos `&&`, `||` y `!` te permiten construir condiciones compuestas, con evaluación de cortocircuito.

Viste los `if` anidados y la trampa del `else` colgante: un `else` siempre se asocia con el `if` más cercano sin `else` propio, así que usa llaves siempre para que no haya ambigüedad. También aprendiste a evitar el anidamiento profundo con `guard clauses`: válida los casos especiales primero y deja el camino normal al final, plano y legible.

La sentencia `switch` compara una expresión contra valores constantes en los `case` y salta directo al que coincida. El `break` es obligatorio para no caer al siguiente caso, el `default` maneja lo inesperado, y el `fall-through` intencional sirve para agrupar casos. Usa `switch` para opciones discretas y claras, e `if` para rangos y condiciones complejas.

El operador ternario, condición `? valor_si : valor_no`, resuelve decisiones cortas en una línea. Perfecto para asignaciones condicionales y mensajes simples; evítalo cuando la lógica se complique o cuando anidar ternarios oscurezca el código.

Las buenas prácticas que te llevas: llaves siempre, condiciones ordenadas de específica a general, `guard clauses` en lugar de pirámides de `if`, `break` y `default` en cada `switch`, ternario solo para lo simple, y condiciones en forma positiva cuando se pueda.

Y los errores que ya no te van a atrapar: el punto y coma después del `if` que vacía la condición, la confusión entre `=` y `==`, el `break` olvidado en el `switch`, las condiciones contradictorias imposibles, la comparación exacta de flotantes con `==`, y el `else` colgante.

Con esto ya puedes escribir programas que validan datos, clasifican resultados, ofrecen menus y reaccionan al usuario. En la próxima unidad viene algo igual de poderoso: los ciclos, que te van a permitir repetir decisiones miles de veces sin escribir miles de líneas. Tus programas están a punto de volverse de verdad útiles. Sigue practicando en lucasegaacademy.com, escribe cada ejemplo con tus manos y no tengas miedo de romper cosas: así se aprende a programar de verdad.

Antes de pasar a la siguiente unidad, hazte estas preguntas de autoevaluación. Si respondes que sí a todas, vas por excelente camino.

Puedo explicar con mis palabras por que en C el cero es falso y cualquier otro número es verdadero? Se escribir un if-else sin mirar apuntes y predecir que rama se ejecuta con un dato dado? Entiendo por que un else se asocia con el if más cercano y como las llaves evitan la confusión? Se cuando conviene un switch y cuando una escalera de else if, y puedo convertir uno en otro? Uso el break en cada case y se explicar que pasa si lo olvido? Escribo el operador ternario para decisiones simples sin que se vuelva ilegible? Válido las entradas del usuario antes de procesarlas? Y compiló con -Wall y leo los warnings?

Si alguna respuesta fue no, regresa a esa sección, reescribe el ejemplo correspondiente con tus manos y prueba los casos frontera. No hay prisa: una unidad bien dominada vale más que tres unidades vistas por encima. Las decisiones que aprendiste aquí aparecen en cada programa que escribas de ahora en adelante, así que cada minuto invertido en dominarlas se multiplica por el resto del curso. Nos vemos en la unidad 5, donde tus decisiones van a empezar a repetirse en ciclos.

UNIDAD 5: Bucles: repetir sin cansarte

Introducción

Imagina que te pido que escribas en tu cuaderno los números del 1 al 10. Fácil, verdad. Ahora imagina que te pido los números del 1 al 1000. Ya no suena tan divertido. Y si te pido los números del 1 al millón, directamente me mandas a volar. Tu mano se cansa, tu mente se aburre, y en algún punto cometes un error y ya ni sabes en qué número ibas.

Pero a una computadora eso no le pasa. La computadora no se cansa. No se aburre. No se distrae con el celular a media tarea. Puedes pedirle que repita algo diez veces, mil veces o un millón de veces, y lo va a hacer con la misma precisión la primera vez que la última. Esa capacidad de repetir sin cansarse es una de las razones principales por las que las computadoras son tan útiles. Y en esta unidad vas a aprender a controlar ese poder: los bucles.

Un bucle, también llamado ciclo o loop, es una estructura que le dice a la computadora: "repite este bloque de instrucciones mientras se cumpla una condición". Es como darle una orden una sola vez y que ella la ejecute cientos o miles de veces sin que tengas que escribirla de nuevo. Si en la unidad anterior aprendiste a tomar decisiones con `if`, `else` y `switch`, ahora vas a aprender a repetir acciones. Decisión y repetición son los dos pilares de todo programa: con ellos puedes construir prácticamente cualquier cosa.

Piensa en la vida real. Lavar los trastes es un bucle: "mientras haya trastes sucios en el fregadero, agarra uno, lavalo, enjuagalo y secalo". La condición es "hay trastes sucios". El bloque que se repite es lavar, enjuagar y secar. Cuando ya no hay trastes sucios, el bucle termina y sigues con tu día. Dar vueltas a una pista para correr es otro bucle: "repite 10 veces: corre una vuelta". Aquí no hay una condición que se revisa, sino un número fijo de repeticiones. Esperar a que hierva el agua también es un bucle: "repite: mira la olla, hasta que el agua hierva". Fíjate que este último tiene algo especial: tienes que mirar la olla al menos una vez para saber si ya hirvió. Esa idea, la de hacer algo al menos una vez antes de revisar la condición, existe en C y se llama `do-while`. Ya la vas a conocer a fondo.

Los bucles están en todas partes en el software que usas todos los días. Cuando abres tu galería de fotos y ves cientos de miniaturas, un bucle recorrió la lista de archivos para mostrarlos. Cuando buscas algo en Google y obtienes millones de resultados en una fracción de segundo, hay bucles comparando tu búsqueda con páginas. Cuando juegas un videojuego, hay un bucle gigante que se repite sesenta veces por segundo: lee tus controles, mueve los

personajes, dibuja la pantalla, y vuelve a empezar. Ese bucle se llama "game loop" y es el corazón de todo videojuego. Cuando Netflix te muestra tu lista de series, cuando tu banco calcula los intereses de tu cuenta, cuando Spotify mezcla tu lista de reproducción: bucles, bucles y más bucles.

En C existen tres bucles principales y cada uno tiene su personalidad. El `while` repite mientras una condición sea verdadera, revisando la condición antes de cada repetición. El `do-while` es su primo: también repite mientras una condición sea verdadera, pero revisa la condición después, así que el bloque se ejecuta al menos una vez si o si. El `for` es el favorito cuando sabes de antemano cuántas veces quieres repetir: tiene un lugar para empezar, un lugar para revisar la condición y un lugar para avanzar al siguiente paso, todo en una sola línea elegante.

Además de los tres bucles, en esta unidad vas a dominar dos herramientas de control que te permiten mandar en el bucle como un director de orquesta: `break`, que rompe el bucle y sale del inmediatamente, y `continue`, que salta a la siguiente repetición sin terminar la actual. Vas a aprender a anidar bucles, es decir, a poner un bucle dentro de otro, que es cómo se dibujan tablas, patrones y se recorren datos en dos dimensiones. También vas a conocer al famoso bucle infinito: ese que nunca termina. A veces es un error que congela tu programa, y a veces es una técnica legítima que se usa con `break` para salir. Vas a aprender a distinguir uno del otro.

Esta unidad cubre las clases 13, 14 y 15 del curso: el bucle `while`, el bucle `for`, y el bucle `do-while` junto con `break` y `continue`. Al terminar, vas a poder escribir programas que cuenten, que sumen listas de números, que dibujen figuras con asteriscos, que calculen factoriales, que generen la serie de Fibonacci y que muestren menús que se repiten hasta que el usuario decida salir. Todo eso que hoy parece magia, al final de esta unidad va a ser tu pan de cada día.

Un consejo antes de empezar: los bucles se aprenden practicando, no solo leyendo. Cada vez que veas un ejemplo, escríbelo tú mismo, compílalo y ejecútalo. Cambia los números, rompe el programa a propósito para ver qué pasa, y luego arreglalo. Esa es la forma en que los conceptos se quedan grabados. Y si algo no te sale a la primera, no te frustres: hasta los programadores con mucha experiencia se equivocan con los bucles. La diferencia es que ellos ya saben dónde buscar el error, y al final de esta unidad tú también vas a saberlo.

En lucasegaacademy.com tienes todo el curso completo de C con 30 clases, y esta unidad es el punto donde muchos estudiantes sienten que "ya programan de verdad", porque los bucles abren la puerta a programas que hacen trabajo pesado de verdad. Así que respira hondo, y vamos a repetir. Y repetir. Y repetir. Hasta que lo domines.

Explicación detallada

El bucle while: repite mientras sea verdad

El bucle while es el más directo de los tres. Su lógica es simple: "mientras la condición sea verdadera, repite este bloque". Antes de cada repetición, C revisa la condición. Si es verdadera, ejecuta el bloque una vez más. Si es falsa, sale del bucle y sigue con el programa. La sintaxis es esta:

```
while (condicion) {  
    // bloque de instrucciones que se repite  
}
```

Fíjate en los detalles. La palabra while va seguida de la condición entre paréntesis, igual que en el if que aprendiste en la unidad anterior. Después van las llaves que encierran el bloque. No lleva punto y coma al final de la primera línea. Ese punto y coma extra es uno de los errores más comunes y más adelante lo vas a ver en la sección de errores.

Vamos con un ejemplo clásico: contar del 1 al 5.

```
#include <stdio.h>  
  
int main() {  
    int contador = 1;  
  
    while (contador <= 5) {  
        printf("%d\n", contador);  
        contador = contador + 1;  
    }  
  
    printf("Termine de contar.\n");  
    return 0;  
}
```

Vamos a seguirle la pista a este programa paso a paso, como si fueras la computadora. Primero se crea la variable contador con el valor 1. Luego llegas al while y revisas la condición: contador <= 5? Como contador vale 1, la respuesta es sí, así que entras al bloque. Se imprime 1. Luego contador pasa a valer 2. Termina el bloque y regresas al while. Revisas de nuevo: 2 <= 5? Sí. Se imprime 2, contador pasa a 3. Así sigues: imprimes 3, 4 y 5. Cuando contador vale 6, revisas la condición: 6 <= 5? No. Entonces sales del bucle y se imprime "Termine de contar."

Hay tres piezas que todo while necesita para funcionar bien, y conviene que las identifiques siempre:

1. La inicialización: la variable que controla el bucle debe empezar con un valor conocido antes del while. En el ejemplo, `int contador = 1`. Si no inicializas, la variable trae basura y el bucle se comporta de forma impredecible.
2. La condición: es la pregunta que se hace antes de cada repetición. Tiene que poder volverse falsa en algún momento, porque si no, el bucle nunca termina. En el ejemplo, `contador <= 5`.
3. La actualización: dentro del bloque, algo tiene que cambiar para que la condición eventualmente se vuelva falsa. En el ejemplo, `contador = contador + 1`. Si olvidas esta línea, `contador` siempre vale 1, la condición siempre es verdadera, y tienes un bucle infinito.

Ese trio (inicialización, condición, actualización) es la columna vertebral de casi todos los bucles. Memorízalo, porque te va a salvar muchas veces cuando un programa se quede "pegado".

La condición del while puede ser cualquier expresión que C pueda evaluar como verdadera o falsa: comparaciones, variables, combinaciones con operadores lógicos. Por ejemplo, puedes pedirle al usuario que escriba números hasta que escriba un cero:

```
#include <stdio.h>

int main() {
    int numero = 1;

    while (numero != 0) {
        printf("Escribe un numero (0 para salir): ");
        scanf("%d", &numero);
        if (numero != 0) {
            printf("Escribiste el %d\n", numero);
        }
    }

    printf("Adios.\n");
    return 0;
}
```

Fíjate en un detalle importante: inicializamos `numero` con 1 para garantizar que entramos al bucle la primera vez. Si lo inicializáramos con 0, la condición `numero != 0` sería falsa desde el

inicio y el bloque nunca se ejecutaría. Este es un patrón muy común: inicializar la variable de control con un valor que asegure la primera entrada al bucle.

También puedes usar el `while` para validar datos. Supon que pides la edad de una persona y quieres asegurarte de que escriba un número razonable, entre 1 y 120:

```
#include <stdio.h>

int main() {
    int edad;

    printf("Escribe tu edad: ");
    scanf("%d", &edad);

    while (edad < 1 || edad > 120) {
        printf("Esa edad no es valida. Intenta de nuevo: ");
        scanf("%d", &edad);
    }

    printf("Tu edad es %d. Dato aceptado.\n", edad);
    return 0;
}
```

Aquí el bucle se repite mientras la edad sea inválida. En cuanto el usuario escribe un valor aceptable, la condición se vuelve falsa y el programa continúa. Este patrón de "repite hasta que el dato sea válido" lo vas a usar muchísimo en programas reales.

El bucle `do-while`: primero actúa, después pregunta

El `do-while` se parece mucho al `while`, pero con una diferencia clave que lo hace único: primero ejecuta el bloque y después revisa la condición. Eso significa que el bloque se ejecuta al menos una vez, pase lo que pase. La sintaxis es esta:

```
do {
    // bloque de instrucciones que se repite
} while (condicion);
```

Nota dos cosas: la palabra `do` abre el bloque sin condición, y al final, después de la llave de cierre, va `while (condición)` con punto y coma. Ese punto y coma al final sí es obligatorio. Es la única de las tres estructuras de bucle que lo lleva.

La diferencia entre while y do-while se entiende mejor con la analogía de la olla con agua. El while es como preguntar "ya hirvió?" antes de mirar la olla: si por alguna razón el agua ya estuviera hirviendo, nunca mirarías. El do-while es como mirar la olla primero y después preguntar "ya hirvió?": la miras al menos una vez, garantizado. En programas, esa garantía es útil cuando la primera ejecución tiene que ocurrir si o si, como mostrar un menú al menos una vez antes de preguntar si el usuario quiere continuar.

Mira el mismo ejemplo de pedir números, ahora con do-while:

```
#include <stdio.h>

int main() {
    int numero;

    do {
        printf("Escribe un numero (0 para salir): ");
        scanf("%d", &numero);
        if (numero != 0) {
            printf("Escribiste el %d\n", numero);
        }
    } while (numero != 0);

    printf("Adios.\n");
    return 0;
}
```

Compara con la versión en while. Ya no necesitamos inicializar número con un valor truco como 1, porque el bloque se ejecuta primero y dentro del bloque se lee el valor real con scanf. Después de ejecutar el bloque, se revisa la condición. El código queda más limpio y más honesto: no hay valores inventados solo para entrar al bucle.

Otro ejemplo donde el do-while brilla: un menú que se muestra al menos una vez.

```

#include <stdio.h>

int main() {
    int opcion;

    do {
        printf("\n--- MENU ---\n");
        printf("1. Saludar\n");
        printf("2. Despedir\n");
        printf("0. Salir\n");
        printf("Elige una opcion: ");
        scanf("%d", &opcion);

        if (opcion == 1) {
            printf("Hola, que tengas buen dia.\n");
        } else if (opcion == 2) {
            printf("Adios, vuelve pronto.\n");
        } else if (opcion != 0) {
            printf("Opcion no valida.\n");
        }
    } while (opcion != 0);

    printf("Programa terminado.\n");
    return 0;
}

```

El menu aparece en pantalla la primera vez sin necesidad de trucos, el usuario elige, y el bucle se repite mientras la opción no sea 0. Cuando elige 0, la condición se vuelve falsa y el programa termina. Este patrón de menu lo vas a ver en la sección de ejemplos paso a paso con más detalle.

Cuando usar while y cuando do-while? La regla práctica es esta: si el bloque debe ejecutarse al menos una vez por la naturaleza del problema (mostrar un menu, pedir un dato por primera vez, lanzar los dados una vez), usa do-while. Si es posible que el bloque no deba ejecutarse ninguna vez (recorrer una lista que podría estar vacía, procesar datos mientras haya datos), usa while. Cuando dudes, piensa: "tiene sentido que esto se haga cero veces?". Si la respuesta es sí, while. Si la respuesta es no, do-while.

El bucle for: el favorito para repetir un número fijo de veces

El for es el bucle más usado en C y probablemente el que más vas a escribir en tu vida como programador. Está diseñado para el caso más común de todos: repetir algo un número conocido de veces. Su sintaxis mete las tres piezas del bucle (inicialización, condición y actualización) en una sola línea:

```
for (inicializacion; condicion; actualizacion) {  
    // bloque de instrucciones que se repite  
}
```

Vamos a desarmar cada parte con el ejemplo de contar del 1 al 5:

```
#include <stdio.h>  
  
int main() {  
    int i;  
  
    for (i = 1; i <= 5; i = i + 1) {  
        printf("%d\n", i);  
    }  
  
    return 0;  
}
```

Primera parte, la inicialización: $i = 1$. Se ejecuta una sola vez, antes de empezar el bucle. Aquí preparas tu contador.

Segunda parte, la condición: $i \leq 5$. Se revisa antes de cada repetición, igual que en el while. Mientras sea verdadera, el bloque se ejecuta.

Tercera parte, la actualización: $i = i + 1$. Se ejecuta al final de cada repetición, después del bloque. Aquí avanzas el contador.

El orden de ejecución es: inicialización (una vez), revisar condición, ejecutar bloque, hacer actualización, revisar condición, ejecutar bloque, hacer actualización... y así hasta que la condición sea falsa.

La variable del for por convención se llama i , y si necesitas una segunda, j , y luego k . Es una tradición que viene de las matemáticas, donde i , j y k se usan como índices. No es obligatoria, pero todo programador de C la reconoce al instante, así que usala: tu código se vuelve más legible para los demás.

En C moderno puedes declarar la variable directamente dentro del for, y es una buena práctica porque limita su alcance al bucle:

```
for (int i = 1; i <= 5; i = i + 1) {  
    printf("%d\n", i);  
}
```

Así la variable `i` solo existe dentro del bucle y no estorba en el resto del programa. En la unidad 8 vas a estudiar el alcance de variables a fondo; por ahora quedate con que declarar el contador dentro del for es limpio y seguro.

El operador de incremento `++` es el mejor amigo del for. En lugar de escribir `i = i + 1`, puedes escribir `i++`. Hacen exactamente lo mismo: sumar 1 a la variable. De la misma forma, `i--` resta 1. Así el for clásico queda así:

```
for (int i = 1; i <= 5; i++) {  
    printf("%d\n", i);  
}
```

Esa línea, `for (int i = 1; i <= 5; i++)`, la vas a escribir miles de veces. Memorízala cómo se memoriza una canción: inicializas en 1, repites mientras `i` sea menor o igual a 5, y sumas 1 cada vez.

El for también puede contar hacia atrás. Solo cambias la inicialización, la condición y la actualización:

```
for (int i = 10; i >= 1; i--) {  
    printf("%d\n", i);  
}  
printf("Despegue.\n");
```

Este imprime 10, 9, 8... hasta 1, y luego "Despegue.". Contar hacia atrás es útil cuando recorres cosas en orden inverso, cómo se verá en la unidad de arreglos.

También puedes avanzar de 2 en 2, de 5 en 5, o como necesites:

```
for (int i = 0; i <= 20; i = i + 2) {  
    printf("%d\n", i);  
}
```

Este imprime los números pares del 0 al 20. La actualización no tiene que ser de 1 en 1; puede ser cualquier cambio que te acerque a terminar el bucle.

Un detalle poderoso: las tres partes del for son opcionales. Puedes dejar vacía la inicialización si ya inicializaste antes, o la actualización si la haces dentro del bloque. Incluso puedes dejar las tres vacías y te queda `for (;;)` , que es un bucle infinito (ya hablaremos de él). Pero ojo: los dos puntos y coma siempre van, aunque las partes esten vacías. Por ahora, mientras aprendes, escribe siempre las tres partes completas. Ya tendrás tiempo de jugar con versiones exóticas.

Bucles anidados: un bucle dentro de otro

Un bucle anidado es simplemente un bucle que vive dentro de otro bucle. El bucle de afuera se llama bucle externo y el de adentro, bucle interno. Cada vez que el bucle externo da una repetición, el bucle interno se ejecuta completo, de principio a fin. Es como las manecillas del reloj: por cada vuelta completa que da la manecilla de las horas, la de los minutos da 12 vueltas completas. O como un calendario: por cada mes (bucle externo), recorres todos sus días (bucle interno).

El ejemplo más claro es dibujar una tabla de multiplicar o un patrón de asteriscos. Vamos a imprimir un cuadrado de 4 por 4:

```
#include <stdio.h>

int main() {
    for (int fila = 1; fila <= 4; fila++) {
        for (int columna = 1; columna <= 4; columna++) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

La salida es:

```
* * * *
* * * *
* * * *
* * * *
```

Sigue la ejecución con calma. El bucle externo empieza con fila = 1. Entra a su bloque y se encuentra con el bucle interno, que se ejecuta completo: columna va de 1 a 4 imprimiendo "*" cuatro veces en la misma línea. Cuando el bucle interno termina, se ejecuta `printf("\n")`, que salta a la siguiente línea. Regresa el control al bucle externo: fila pasa a 2, y el bucle interno se ejecuta completo otra vez. Así hasta que fila llega a 5 y el bucle externo termina. En total, el bloque interno se ejecutó $4 \times 4 = 16$ veces.

Nota que el `printf("\n")` está dentro del bucle externo pero fuera del interno. Esa posición es clave: el salto de línea ocurre una vez por fila, no una vez por asterisco. Si lo pusieras dentro del bucle interno, cada asterisco quedaría en su propia línea y no se formaría el cuadrado. Cuando trabajes con bucles anidados, siempre pregúntate: "esto debe pasar una vez por vuelta externa o una vez por vuelta interna?".

Puedes anidar cualquier combinación: un `for` dentro de un `while`, un `while` dentro de un `for`, un `do-while` dentro de un `for`. También puedes anidar más de dos niveles, aunque con tres niveles el código ya se vuelve difícil de seguir. La regla de oro es usar variables con nombres claros (fila y columna en vez de `i` y `j`) cuando anidas, porque así no te pierdes.

Los bucles anidados son la base para trabajar con tablas, matrices y datos en dos dimensiones, que vas a estudiar en la clase 20 con arreglos multidimensionales. También se usan para comparar cada elemento de una lista con todos los demás, para generar combinaciones y para dibujar figuras. En la sección de ejemplos vas a practicar patrones con asteriscos, que son el ejercicio clásico para dominar el anidamiento.

break: salir del bucle a la fuerza

A veces necesitas abandonar el bucle antes de que la condición se vuelva falsa. Para eso existe `break`: cuando `C` lo encuentra, sale del bucle inmediatamente, sin terminar la repetición actual y sin revisar la condición. El programa continúa con la primera instrucción después del bucle.

El uso más común de `break` es buscar algo y detenerse cuando lo encuentras. Imagina que buscas el primer número mayor a 50 en una secuencia:

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 100; i++) {
        if (i > 50) {
            printf("Encontre el %d, ya no sigo buscando.\n", i);
            break;
        }
    }
    printf("Sali del bucle.\n");
    return 0;
}
```

El bucle estaba planeado para llegar hasta 100, pero cuando *i* vale 51, el *if* se cumple, se imprime el mensaje y *break* saca al programa del *for*. Se imprime "Sali del bucle." y el programa sigue. Sin el *break*, el bucle seguiría hasta 100 haciendo trabajo inútil.

Otro uso clásico: salir de un bucle infinito intencional. Mira este patrón, muy usado en programas reales:

```
while (1) {
    // hacer algo
    if (condicion_de_salida) {
        break;
    }
}
```

while (1) es un bucle infinito porque la condición 1 siempre es verdadera en C (cualquier valor distinto de cero cuenta como verdadero). Pero no es un error: es una técnica deliberada. El bucle se repite para siempre... hasta que algo dentro del ejecuta *break*. Es como decir "repite esto sin parar, y yo te aviso cuando pares". Lo vas a ver en el ejemplo del menú: el menú se repite con *while (1)* y solo sale cuando el usuario elige la opción de salir.

Un detalle importante: *break* solo sale del bucle más interno. Si tienes bucles anidados y haces *break* dentro del interno, el externo sigue corriendo. Mira:


```

for (int fila = 1; fila <= 3; fila++) {
    for (int columna = 1; columna <= 3; columna++) {
        if (columna == 2) {
            break;
        }
        printf("fila %d, columna %d\n", fila, columna);
    }
}

```

El break está dentro del bucle de columna, así que solo interrumpe ese. El bucle de fila sigue con la siguiente fila. La salida muestra la columna 1 de cada fila, pero nunca la columna 2 ni la 3.

continue: saltar a la siguiente repetición

Mientras break abandona el bucle por completo, continue abandona solo la repetición actual y salta directamente a la siguiente. Es como decir "esta vuelta no me interesa, pasemos a la próxima". En el for, continue salta a la actualización (i++); en el while y do-while, salta a revisar la condición.

El uso típico de continue es saltarse los casos que no te interesan. Por ejemplo, imprimir los números del 1 al 10 excepto los múltiplos de 3:

```

#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 3 == 0) {
            continue;
        }
        printf("%d\n", i);
    }
    return 0;
}

```

Cuando i es 3, 6 o 9, el if se cumple y continue salta el printf, pasando a la siguiente vuelta. La salida es 1, 2, 4, 5, 7, 8, 10. Sin continue tendrías que escribir el printf dentro de un else, y con varias condiciones el código se llenaría de anidamientos. continue mantiene el código plano y legible: primero descartas lo que no quieres, y luego trabajas con lo que sí.

Otro ejemplo práctico: sumar solo los números positivos que escriba el usuario, ignorando los negativos:

```
#include <stdio.h>

int main() {
    int suma = 0;
    int numero;

    for (int i = 1; i <= 5; i++) {
        printf("Escribe el numero %d: ", i);
        scanf("%d", &numero);
        if (numero < 0) {
            printf("Negativo ignorado.\n");
            continue;
        }
        suma = suma + numero;
    }

    printf("La suma de los positivos es %d\n", suma);
    return 0;
}
```

Cuidado con continue dentro de while: como salta directo a la condición sin pasar por el resto del bloque, si la actualización del contador esta al final del bloque, se la salta también y puedes crear un bucle infinito sin querer. Mira el peligro:

```
int i = 1;
while (i <= 10) {
    if (i % 3 == 0) {
        continue; // PELIGRO: se salta el i++ de abajo
    }
    printf("%d\n", i);
    i++;
}
```

Cuando i vale 3, continue salta el i++, i sigue valiendo 3 para siempre, y el bucle nunca termina. En el for este problema no existe porque la actualización (i++) está en la cabecera del for y continue si la ejecuta. Moraleja: con while, asegurate de que el contador avance antes del continue, o mejor usa for cuando necesites continue con contador.

Bucles infinitos: el error y la técnica

Un bucle infinito es un bucle cuya condición nunca se vuelve falsa. El programa se queda repitiendo para siempre: no avanza, no termina, y tienes que cerrarlo a la fuerza con Ctrl + C en la terminal. La mayoría de las veces es un error, y casi siempre la causa es una de estas tres:

1. Olvidaste actualizar la variable de control:

```
int i = 1;
while (i <= 5) {
    printf("%d\n", i);
    // falta i++
}
```

i siempre vale 1, la condición siempre es verdadera, y el programa imprime 1 eternamente.

2. La condición está mal planteada y nunca puede volverse falsa:

```
for (int i = 1; i > 0; i++) {
    printf("%d\n", i);
}
```

i empieza en 1 y solo crece, así que *i* > 0 siempre es verdad. Querías escribir *i* <= 10 o algo parecido.

3. El punto y coma fantasma después del while o del for (lo verás en errores comunes):

```
int i = 1;
while (i <= 5); // este punto y coma crea un bucle vacío infinito
{
    printf("%d\n", i);
    i++;
}
```

El punto y coma después de la condición forma un bucle cuyo bloque es "nada". Como nada cambia a *i* dentro de ese bucle vacío, nunca termina. El bloque entre llaves que sigue ya no pertenece al while: es código normal que nunca se alcanza.

Pero no todo bucle infinito es un error. Hay una técnica legítima y muy usada: el bucle infinito intencional con salida por break. Se usa cuando no sabes de antemano cuántas veces se va a repetir, o cuando la condición de salida solo se puede revisar en medio del bloque, no al principio ni al final. Los dos patrones clásicos son:

```
while (1) {  
    // ...  
    if (ya_terminamos) break;  
}
```

y

```
for (;;) {  
    // ...  
    if (ya_terminamos) break;  
}
```

Ambos hacen lo mismo: repetir para siempre hasta que un break los saque. Los vas a ver en menus interactivos, en servidores que atienden peticiones sin parar, y en videojuegos con su game loop. La clave para que sea técnica y no error es que SIEMPRE haya un break alcanzable dentro. Si escribes while (1) sin ningún break, es un error garantizado.

Cuando usar cada bucle: la guía práctica

Con tres bucles disponibles, la pregunta natural es cuál elegir. Aquí va la guía que usan los programadores experimentados:

Usa for cuando sabes cuántas veces se repite, o cuando recorres una secuencia con un contador. Es el caso más común: contar del 1 al 10, recorrer los 30 alumnos de una lista, repetir algo 5 veces. Si tu problema suena a "repite N veces", la respuesta es for. Además, el for junta la inicialización, la condición y la actualización en un solo lugar visible, lo que hace más difícil olvidar alguna de las tres piezas.

Usa while cuando no sabes cuántas veces se va a repetir y la condición se revisa antes de cada vuelta. Casos típicos: leer datos hasta que llegue un valor especial, esperar a que algo cambie, validar la entrada del usuario, recorrer datos mientras haya datos. Si tu problema suena a "repite mientras pase esto", la respuesta es while.

Usa do-while cuando el bloque tiene que ejecutarse al menos una vez por la naturaleza del problema. Casos típicos: mostrar un menu, pedir un dato por primera vez, jugar una ronda y preguntar si quiere otra. Si tu problema suena a "haz esto y luego pregunta si seguimos", la respuesta es do-while.

En la práctica, el for se lleva como el 70 por ciento de los casos, el while como el 25, y el do-while el resto. No es una regla matemática, es experiencia acumulada: la mayoría de las repeticiones en programas reales tienen un número conocido de vueltas o un contador claro.

Y un secreto a voces: cualquier bucle se puede escribir con cualquiera de los tres. El for de contar del 1 al 5 se puede escribir como while, y el while de validar edad se puede escribir como for. La diferencia no es lo que pueden hacer, sino lo que comunican. Elegir el bucle adecuado es como elegir la palabra adecuada: el programa funciona igual, pero el que lo lee entiende tu intención más rápido. Escribe código para que lo entienda un humano, no solo la computadora.

Errores comunes con bucles (y como evitarlos)

Estos son los errores que todo programador comete con los bucles, del principiante al experto. Conocelos de memoria y te vas a ahorrar horas de frustración.

Error 1: olvidar actualizar el contador. Ya lo viste: el bucle nunca termina porque la variable de control no cambia. Sintoma: el programa se queda "pegado" imprimiendo lo mismo. Solución: verifica siempre que dentro del bloque algo modifique la variable de la condición. Con for es más difícil cometerlo porque la actualización está a la vista en la cabecera.

Error 2: el punto y coma después del for o del while. Mira con atención:

```
for (int i = 1; i <= 5; i++); // punto y coma asesino
{
    printf("%d\n", i);
}
```

Ese punto y coma después del paréntesis convierte al for en un bucle con cuerpo vacío: repite "nada" cinco veces. El bloque entre llaves que sigue ya no es parte del bucle: se ejecuta una sola vez, después del bucle, y para colmo la variable *i* ya no existe ahí (si la declaraste dentro del for), así que ni siquiera compila, o usa basura. Sintoma: el programa no imprime nada dentro del supuesto bucle, o marca error de compilación raro. Solución: nunca pongas punto y coma después de la cabecera del for, del while ni del if. El único que lo lleva es el do-while al final.

Error 3: el error de off-by-one, o "errar por uno". Es el error más famoso de la programación: repetir una vez de más o una vez de menos por confundir *<* con *<=*. Mira:

```
// Quieres imprimir del 1 al 10...
for (int i = 1; i < 10; i++) // imprime del 1 al 9: falta el 10
```

y al revés:

```
// Quieres 10 repeticiones empezando en 0...
for (int i = 0; i <= 10; i++) // hace 11 repeticiones: sobra una
```

La forma de no fallar es contar con los dedos las primeras veces: escribe los valores que toma `i` en un papel y verifica cuántos son. Con el tiempo lo harás de memoria. Un truco útil: el patrón `for (int i = 0; i < N; i++)` hace exactamente `N` repeticiones (de 0 a `N-1`). El patrón `for (int i = 1; i <= N; i++)` también hace `N` repeticiones (de 1 a `N`). Elige uno y usalo siempre igual para no confundirte.

Error 4: usar `=` en lugar de `==` en la condición. Ya lo viste en la unidad de decisiones, pero en bucles es peor:

```
int i = 1;
while (i = 5) { // asignacion, no comparacion: i vale 5, que es verdadero
    // bucle infinito
}
```

`i = 5` asigna 5 a `i` y la expresión vale 5, que C interpreta como verdadero. Bucle infinito garantizado. Solución: compila siempre con warnings activados (`gcc -Wall`) y el compilador te avisa de este error. En la clase 29 vas a profundizar en warnings y depuración.

Error 5: modificar la variable del `for` dentro del bloque de forma inesperada. El `for` ya actualiza `i` en su cabecera; si además la modificas dentro del bloque, el conteo se vuelve impredecible:

```
for (int i = 1; i <= 10; i++) {
    printf("%d\n", i);
    i = i + 2; // el contador avanza 3 por vuelta: confuso
}
```

A veces es intencional, pero casi siempre es un accidente. Solución: trata la variable del `for` como de solo lectura dentro del bloque. Si necesitas saltar valores, mejor ajusta la actualización en la cabecera.

Error 6: condición que nunca cambia porque depende de algo que no se actualiza. Por ejemplo, leer un sensor o una variable que otra parte del programa debería cambiar, pero no lo hace. El bucle espera algo que nunca llega. Solución: asegurate de que todo lo que aparece en la condición pueda cambiar dentro del bucle o por efecto del bucle.

Buenas prácticas con bucles

1. Inicializa siempre tus contadores. Una variable sin inicializar trae un valor basura y el bucle empieza en un lugar aleatorio. Con `for (int i = ...)` esto sale gratis.
2. Prefiere `for` para conteos fijos. Junta las tres piezas en un solo lugar y reduce los olvidos.
3. Usa nombres claros en bucles anidados. `i` y `j` están bien para casos simples, pero `fila`, `columna`, `intento` o `alumno` comunican mucho más.
4. Manten el bloque del bucle corto. Si el bloque crece más de 15 o 20 líneas, considera moverlo a una función (lo verás en la unidad 6). Un bucle corto se entiende de un vistazo.
5. Evita bucles infinitos sin `break` alcanzable. Cada `while` (1) o `for` (;) debe tener su `break` claramente visible.
6. No abuses de `break` y `continue`. Son herramientas útiles, pero un bucle lleno de `breaks` y `continues` es difícil de seguir. Uno o dos por bucle está bien; si necesitas más, probablemente la lógica se puede reordenar.
7. Prueba los casos borde: qué pasa si el bucle debe ejecutarse cero veces, una vez, y muchas veces. Esos tres casos revelan casi todos los errores de `off-by-one`.
8. Compila con warnings: `gcc -Wall -o programa programa.c`. El compilador es tu mejor amigo para cazar errores de bucles antes de que te cachén a ti.

Dominar los bucles es dominar la repetición, y dominar la repetición es lo que separa a quien "escribe código" de quien "programa de verdad". En lucasegaacademy.com este es el punto del curso donde los estudiantes empiezan a resolver problemas que antes parecían imposibles. Tu ya estás ahí. Vamos a los ejemplos.

Ejemplos paso a paso

Aquí van ocho programas completos que cubren todo lo de la unidad. Escríbelos, compílalos y ejecútalos tú mismo. Después juega con ellos: cambia los números, combina ideas de un ejemplo con otro, y observa que pasa.

Ejemplo 1: contar del 1 al 10 con los tres bucles

El mismo problema resuelto con `while`, `do-while` y `for`, para que compares los tres estilos lado a lado.

```

#include <stdio.h>

int main() {
    printf("Con while:\n");
    int a = 1;
    while (a <= 10) {
        printf("%d ", a);
        a++;
    }

    printf("\n\nCon do-while:\n");
    int b = 1;
    do {
        printf("%d ", b);
        b++;
    } while (b <= 10);

    printf("\n\nCon for:\n");
    for (int c = 1; c <= 10; c++) {
        printf("%d ", c);
    }
    printf("\n");

    return 0;
}

```

La salida de las tres versiones es idéntica: 1 2 3 4 5 6 7 8 9 10. Fíjate como el `for` resuelve el mismo problema en menos líneas y con las tres piezas del bucle juntas en la cabecera. Por eso es el favorito para conteos fijos. El `while` necesita la inicialización antes y el incremento dentro; el `do-while` es igual al `while` pero garantiza la primera vuelta. Cuando el problema es "repite N veces", el `for` comunica tu intención de forma más clara.

Ejemplo 2: tabla de multiplicar

El programa pide un número y muestra su tabla de multiplicar del 1 al 10. Es el ejemplo perfecto del `for` con un propósito real.


```
#include <stdio.h>

int main() {
    int numero;

    printf("Escribe un numero para ver su tabla de multiplicar: ");
    scanf("%d", &numero);

    printf("\nTabla del %d:\n", numero);
    for (int i = 1; i <= 10; i++) {
        printf("%d x %d = %d\n", numero, i, numero * i);
    }

    return 0;
}
```

Si el usuario escribe 7, la salida es $7 \times 1 = 7$, $7 \times 2 = 14$, y así hasta $7 \times 10 = 70$. Observa el printf con tres %d: el primero es el número fijo, el segundo es el contador i que va cambiando, y el tercero es el resultado de $\text{numero} \times i$. Este patrón de "un valor fijo combinado con un contador que avanza" aparece en muchísimos programas: tablas, calendarios, listas numeradas. Como reto, modifica el programa para que muestre las tablas del 1 al 10 completas usando bucles anidados: el bucle externo recorre el número de la tabla y el interno recorre del 1 al 10.

Ejemplo 3: suma de los primeros N números

El usuario dice cuántos números quiere sumar, y el programa suma del 1 hasta ese número usando un acumulador. Este patrón de acumulador es uno de los más importantes en programación.

```
#include <stdio.h>

int main() {
    int n;
    int suma = 0;

    printf("Hasta que numero quieres sumar? ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        suma = suma + i;
    }

    printf("La suma del 1 al %d es %d\n", n, suma);
    return 0;
}
```

La variable suma es un acumulador: empieza en 0 y en cada vuelta se le agrega el valor actual de i. Sigue la ejecución con n = 5: suma empieza en 0; vuelta 1: suma = 0 + 1 = 1; vuelta 2: suma = 1 + 2 = 3; vuelta 3: suma = 3 + 3 = 6; vuelta 4: suma = 6 + 4 = 10; vuelta 5: suma = 10 + 5 = 15. Resultado: 15. El acumulador siempre se inicializa antes del bucle (casi siempre en 0 para sumas o en 1 para multiplicaciones) y se actualiza dentro. Si olvidas inicializarlo, empieza con basura y el resultado es incorrecto. Dato curioso: existe una fórmula matemática para esto, $n * (n + 1) / 2$, que da el resultado sin bucle. Pero el acumulador con bucle sirve para sumas donde no hay fórmula, como sumar solo los números pares o los que cumplen una condición.

Ejemplo 4: adivina el número (do-while)

El clásico juego de adivinar: la computadora piensa un número y tú intentas adivinarlo. El do-while es perfecto porque el jugador debe intentar al menos una vez.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int secreto;
    int intento;
    int contador = 0;

    srand(time(0));
    secreto = rand() % 100 + 1;

    printf("Pense un numero del 1 al 100. Adivinalo.\n");

    do {
        printf("Tu intento: ");
        scanf("%d", &intento);
        contador++;

        if (intento < secreto) {
            printf("Muy bajo, intenta mas alto.\n");
        } else if (intento > secreto) {
            printf("Muy alto, intenta mas bajo.\n");
        }
    } while (intento != secreto);

    printf("Correcto. Lo adivinaste en %d intentos.\n", contador);
    return 0;
}

```

Hay varias cosas nuevas aquí. `srand(time(0))` y `rand()` sirven para generar números aleatorios: sin `srand`, `rand` daría siempre la misma secuencia. `rand() % 100` da un número entre 0 y 99, y al sumarle 1 queda entre 1 y 100. El `do-while` pide el intento, lo compara con el secreto y da pistas ("muy bajo", "muy alto"), y se repite mientras `intento != secreto`. El contador cuenta cuántos intentos llevo el jugador. Fíjate que el bloque se ejecuta al menos una vez por definición: no tiene sentido preguntar si quiere jugar antes de jugar la primera ronda. Como mejora, podrías limitar los intentos a 7 y usar `break` para salir si se acaban, o agregar niveles de dificultad. Este juego, aunque simple, ya tiene todos los ingredientes de un programa interactivo real.

Ejemplo 5: patrones con asteriscos (bucles anidados)

Tres figuras clásicas que se dibujan con bucles anidados. Son el ejercicio definitivo para entender cómo se coordinan el bucle externo y el interno.

```
#include <stdio.h>

int main() {
    int n = 5;

    printf("Triangulo:\n");
    for (int fila = 1; fila <= n; fila++) {
        for (int col = 1; col <= fila; col++) {
            printf("* ");
        }
        printf("\n");
    }

    printf("\nTriangulo invertido:\n");
    for (int fila = n; fila >= 1; fila--) {
        for (int col = 1; col <= fila; col++) {
            printf("* ");
        }
        printf("\n");
    }

    printf("\nCuadrado hueco:\n");
    for (int fila = 1; fila <= n; fila++) {
        for (int col = 1; col <= n; col++) {
            if (fila == 1 || fila == n || col == 1 || col == n) {
                printf("* ");
            } else {
                printf("  ");
            }
        }
        printf("\n");
    }

    return 0;
}
```

La salida con n = 5 es:

Triangulo:

```
*  
* *  
* * *  
* * * *  
* * * * *
```

Triangulo invertido:

```
* * * * *  
* * * *  
* * *  
* *  
*
```

Cuadrado hueco:

```
* * * * *  
*       *  
*       *  
*       *  
* * * * *
```

Analiza el triángulo: el bucle externo recorre las filas del 1 al 5. El bucle interno imprime asteriscos desde 1 hasta fila. En la fila 1 imprime 1 asterisco, en la fila 2 imprime 2, y así. La clave es que el límite del bucle interno depende de la variable del bucle externo ($col \leq fila$). Esa dependencia entre los dos bucles es lo que crea la forma triangular. En el triángulo invertido, el externo va de 5 hacia abajo. En el cuadrado hueco, el if decide: si estamos en el borde (primera o última fila, primera o última columna), imprime asterisco; si no, imprime espacios. Cambia la n a 8 o a 3 y observa como las figuras se adaptan solas. Luego intenta dibujar una piramide centrada: necesitarás un tercer bucle interno para los espacios de la izquierda.

Ejemplo 6: factorial de un número

El factorial de n (escrito $n!$) es el producto de todos los números del 1 al n . Por ejemplo, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Aquí el acumulador empieza en 1 y multiplica en vez de sumar.

```
#include <stdio.h>

int main() {
    int n;
    long long factorial = 1;

    printf("Escribe un numero para calcular su factorial: ");
    scanf("%d", &n);

    if (n < 0) {
        printf("El factorial no esta definido para negativos.\n");
    } else {
        for (int i = 1; i <= n; i++) {
            factorial = factorial * i;
        }
        printf("El factorial de %d es %lld\n", n, factorial);
    }

    return 0;
}
```

Sigue la ejecución con $n = 5$: factorial empieza en 1; vuelta 1: $1 \times 1 = 1$; vuelta 2: $1 \times 2 = 2$; vuelta 3: $2 \times 3 = 6$; vuelta 4: $6 \times 4 = 24$; vuelta 5: $24 \times 5 = 120$. Dos detalles importantes. Primero, usamos long long en vez de int porque el factorial crece rapidísimo: $12!$ ya es 479 millones y $13!$ no cabe en un int normal. Con long long llegas hasta $20!$. Segundo, el caso borde: $0!$ vale 1 por definición matemática, y el programa lo maneja bien porque si n es 0, el for no se ejecuta ninguna vez y factorial conserva su valor inicial de 1. Los negativos se rechazan con el if. Este ejemplo muestra algo hermoso: el mismo patrón de acumulador del ejemplo 3, pero cambiando la operación de suma a multiplicación y el valor inicial de 0 a 1.

Ejemplo 7: serie de Fibonacci

La serie de Fibonacci empieza con 0 y 1, y cada número siguiente es la suma de los dos anteriores: 0, 1, 1, 2, 3, 5, 8, 13, 21... Aparece en la naturaleza, en el arte y en muchísimos problemas de programación.

```

#include <stdio.h>

int main() {
    int n;
    long long a = 0, b = 1, siguiente;

    printf("Cuantos terminos de Fibonacci quieres ver? ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Escribe un numero positivo.\n");
    } else if (n == 1) {
        printf("0\n");
    } else {
        printf("%lld %lld ", a, b);
        for (int i = 3; i <= n; i++) {
            siguiente = a + b;
            printf("%lld ", siguiente);
            a = b;
            b = siguiente;
        }
        printf("\n");
    }

    return 0;
}

```

La técnica aquí es llevar los dos últimos valores en las variables a y b. En cada vuelta se calcula $\text{siguiente} = a + b$, se imprime, y luego se "recorren" los valores: a toma el valor viejo de b, y b toma el valor de siguiente. Sigue la ejecución: $a = 0$, $b = 1$. Se imprimen 0 y 1. Vuelta $i = 3$: $\text{siguiente} = 0 + 1 = 1$, se imprime 1, a pasa a 1, b pasa a 1. Vuelta $i = 4$: $\text{siguiente} = 1 + 1 = 2$, se imprime 2, $a = 1$, $b = 2$. Vuelta $i = 5$: $\text{siguiente} = 1 + 2 = 3$... y así. El orden de las asignaciones es crítico: primero $a = b$ y después $b = \text{siguiente}$. Si lo hicieras al revés, perderías el valor de b antes de pasarlo a a. Este patrón de "deslizar" variables hacia adelante aparece en muchos algoritmos, así que entiéndelo bien.

Ejemplo 8: menu que se repite hasta salir

El programa muestra un menu de opciones, ejecuta la elegida y vuelve a mostrar el menu hasta que el usuario elige salir. Es el patrón de bucle infinito intencional con break.

```

#include <stdio.h>

int main() {
    int opcion;

    while (1) {
        printf("\n===== MENU PRINCIPAL =====\n");
        printf("1. Contar del 1 al 10\n");
        printf("2. Mostrar tabla del 5\n");
        printf("3. Calcular factorial de 6\n");
        printf("0. Salir\n");
        printf("Elige una opcion: ");
        scanf("%d", &opcion);

        if (opcion == 1) {
            for (int i = 1; i <= 10; i++) {
                printf("%d ", i);
            }
            printf("\n");
        } else if (opcion == 2) {
            for (int i = 1; i <= 10; i++) {
                printf("5 x %d = %d\n", i, 5 * i);
            }
        } else if (opcion == 3) {
            long long f = 1;
            for (int i = 1; i <= 6; i++) {
                f = f * i;
            }
            printf("6! = %lld\n", f);
        } else if (opcion == 0) {
            printf("Gracias por usar el programa. Adios.\n");
            break;
        } else {
            printf("Opcion no valida, intenta de nuevo.\n");
        }
    }

    return 0;
}

```

El while (1) crea el bucle infinito intencional: el menu se muestra una y otra vez. Cada opción ejecuta su bloque, que a su vez usa bucles for internos: aquí ves bucles dentro de un bucle, combinando todo lo de la unidad. La única salida es la opción 0, que ejecuta break y rompe el while. Si el usuario escribe algo invalido, se le avisa y el menu vuelve a aparecer. Este

esqueleto de menu te va a servir para el proyecto final de la clase 30 y para casi cualquier programa interactivo que escribas: muestras opciones, lees la elección, actúas, y repites. Cuando aprendas funciones en la unidad 6, cada opción del menu se convertirá en una llamada a función y el código quedará aun más limpio.

Práctica

Es momento de ensuciarte las manos. Intenta resolver cada ejercicio por tu cuenta antes de mirar la solución. Si te atorras más de 20 minutos, mira la solución, entiendela, ciérrala y escríbela de nuevo sin copiar. Así es cómo se aprende de verdad.

Antes de lanzarte, un método para atacar cualquier ejercicio de bucles. Primero, escribe en una frase que debe hacer el programa, sin código: "imprimir los pares del 2 al 50", "sumar números hasta que llegue un cero". Segundo, decide que bucle usar con la guía de la unidad: sabes cuántas vueltas son? `for`. No lo sabes y la condición se revisa antes? `while`. El bloque debe correr al menos una vez? `do-while`. Tercero, identifica tus variables: necesitas un contador? un acumulador? una bandera? Escríbelas con su valor inicial antes de tocar el bucle. Cuarto, escribe el bucle y dentro del una sola cosa a la vez: primero logra que repita bien, luego agrega la lógica. Quinto, prueba con valores chicos que puedas verificar a mano: si tu programa suma del 1 al 5, tú sabes que debe dar 15. Si da otra cosa, el error está a la vista. Este método de cinco pasos parece lento al inicio, pero es más rápido que adivinar código y rezar: te obliga a pensar antes de escribir, que es exactamente lo que distingue a quien programa de quien solo teclea.

Un recordatorio importante sobre los errores: te vas a equivocar, y está bien. Cada bucle infinito que provoques, cada off-by-one que cometas, cada punto y coma fantasma que se te escape, te está enseñando algo que ningún libro puede enseñarte solo con palabras. La frustración que sientes cuando el programa no hace lo que quieres es la sensación exacta del aprendizaje ocurriendo. Los programadores profesionales no son los que nunca se equivocan: son los que se equivocan rápido, encuentran el error rápido y lo arreglan rápido. Cada ejercicio que resuelvas, incluso mirando la solución después de intentarlo, te acerca a esa velocidad. Así que respira, lee el enunciado otra vez con calma, y empieza por lo más simple que se te ocurra. Lo simple que funciona siempre le gana a lo elegante que no compila.

1. Números pares del 2 al 50: escribe un programa que imprima todos los números pares entre 2 y 50 usando un bucle `for`. Hazlo de dos formas: avanzando de 2 en 2, y usando el operador módulo `%` con `continue` para saltar los impares.

2. Suma hasta que el usuario diga basta: escribe un programa que pida números al usuario una y otra vez, los vaya sumando, y se detenga cuando el usuario escriba 0. Al final debe mostrar la suma total y cuántos números se sumaron (sin contar el cero).
3. Piramide de asteriscos: usando bucles anidados, dibuja una piramide centrada de altura 5, asi:

```
  *
 ***
*****
*****
*****
*****
```

Pista: cada fila tiene espacios a la izquierda y luego un número impar de asteriscos.

4. Número primo o no: escribe un programa que pida un número entero y determine si es primo. Un número primo solo es divisible entre 1 y el mismo. Usa un for para probar divisores y break para salir en cuanto encuentres uno.
5. Contador de digitos y suma de digitos: pide un número entero positivo y calcula cuántos digitos tiene y cuánto suman sus digitos. Por ejemplo, 473 tiene 3 dígitos y sus digitos suman 14. Pista: usa un while que en cada vuelta tome el último dígito con número % 10 y luego quite ese dígito con número / 10.
6. Juego de adivinar con intentos limitados: modifica el juego de adivina el número del ejemplo 4 para que el jugador tenga máximo 7 intentos. Si adivina, felicitalo y muestra en cuántos intentos lo logro. Si se le acaban los intentos, revela el número secreto. Usa break para salir cuando adivine.

Solución del ejercicio 1

```

#include <stdio.h>

int main() {
    printf("Forma 1: avanzando de 2 en 2\n");
    for (int i = 2; i <= 50; i = i + 2) {
        printf("%d ", i);
    }

    printf("\n\nForma 2: con continue y modulo\n");
    for (int i = 2; i <= 50; i++) {
        if (i % 2 != 0) {
            continue;
        }
        printf("%d ", i);
    }
    printf("\n");

    return 0;
}

```

Explicación: la primera forma es la más eficiente y clara: el contador avanza de 2 en 2, así que solo visita números pares. La actualización $i = i + 2$ hace todo el trabajo. La segunda forma recorre todos los números del 2 al 50 y usa `continue` para saltarse los impares: si $i \% 2 \neq 0$, el número es impar y se salta el `printf`. Ambas dan el mismo resultado, pero la primera hace 25 vueltas y la segunda 49. Cuando puedes evitar vueltas innecesarias ajustando el paso del `for`, hazlo: es más rápido y más legible. La segunda forma es útil cuando la condición para saltar es más compleja que "ser par".

Solución del ejercicio 2

```

#include <stdio.h>

int main() {
    int numero;
    int suma = 0;
    int cantidad = 0;

    printf("Escribe numeros para sumarlos (0 para terminar):\n");

    while (1) {
        printf("Numero: ");
        scanf("%d", &numero);

        if (numero == 0) {
            break;
        }

        suma = suma + numero;
        cantidad++;
    }

    printf("Sumaste %d numeros.\n", cantidad);
    printf("La suma total es %d\n", suma);

    return 0;
}

```

Explicación: aquí usamos el patrón de bucle infinito intencional while (1) con salida por break, porque no sabemos cuántos números va a escribir el usuario. En cada vuelta se lee un número; si es 0, break rompe el bucle antes de sumarlo. Si no es 0, se acumula en suma y se incrementa cantidad. Fíjate que el break esta antes de la suma: así el cero que sirve como señal de salida no contamina ni la suma ni el conteo. Este patrón de "valor centinela" (un valor especial que marca el fin de los datos) es muy común: el 0 aquí, o un número negativo en otros programas. Como variante, podrias usar do-while con la condición número != 0, pero tendrías que cuidar no sumar el cero final.

Solución del ejercicio 3

```

#include <stdio.h>

int main() {
    int altura = 5;

    for (int fila = 1; fila <= altura; fila++) {
        for (int e = 1; e <= altura - fila; e++) {
            printf(" ");
        }
        for (int a = 1; a <= 2 * fila - 1; a++) {
            printf("*");
        }
        printf("\n");
    }

    return 0;
}

```

Explicación: cada fila tiene dos partes: espacios a la izquierda y asteriscos. Para la fila 1: 4 espacios y 1 asterisco. Para la fila 2: 3 espacios y 3 asteriscos. Para la fila 3: 2 espacios y 5 asteriscos. El patrón de espacios es $\text{altura} - \text{fila}$ (4, 3, 2, 1, 0) y el de asteriscos es $2 * \text{fila} - 1$ (1, 3, 5, 7, 9), que son los números impares. Por eso necesitamos dos bucles internos: uno para los espacios y otro para los asteriscos, ambos dentro del bucle externo de filas. Este ejercicio combina todo: anidamiento, fórmulas que dependen de la variable del bucle externo, y tres bucles coordinados. Si lograste deducir las fórmulas $\text{altura} - \text{fila}$ y $2 * \text{fila} - 1$ por tu cuenta, ya piensas como programador: convertiste un dibujo en matemáticas y las matemáticas en código.

Solución del ejercicio 4

```

#include <stdio.h>

int main() {
    int n;
    int es_primo = 1;

    printf("Escribe un numero: ");
    scanf("%d", &n);

    if (n <= 1) {
        es_primo = 0;
    } else {
        for (int i = 2; i * i <= n; i++) {
            if (n % i == 0) {
                es_primo = 0;
                break;
            }
        }
    }

    if (es_primo) {
        printf("%d es primo.\n", n);
    } else {
        printf("%d no es primo.\n", n);
    }

    return 0;
}

```

Explicación: la estrategia es probar si n tiene algún divisor entre 2 y $n - 1$. Si encontramos uno, no es primo y `break` nos saca del bucle de inmediato: ya no tiene caso seguir buscando. La variable `es_primo` funciona como bandera: empieza en 1 (suponemos que es primo) y cambia a 0 si encontramos un divisor. Los números menores o iguales a 1 no son primos por definición, así que se manejan aparte. El truco $i * i \leq n$ es una optimización: basta probar divisores hasta la raíz cuadrada de n , porque si n tuviera un divisor mayor a su raíz cuadrada, el divisor compañero sería menor y ya lo habríamos encontrado. Por ejemplo, para $n = 100$ basta probar hasta 10. Sin esta optimización el programa seguiría siendo correcto, solo más lento con números grandes. Este es tu primer algoritmo de verdad: una receta con nombre propio que resuelve un problema matemático clásico.

Solución del ejercicio 5

```
#include <stdio.h>

int main() {
    int numero;
    int digitos = 0;
    int suma = 0;

    printf("Escribe un numero entero positivo: ");
    scanf("%d", &numero);

    if (numero <= 0) {
        printf("Tiene que ser positivo.\n");
    } else {
        while (numero > 0) {
            int ultimo = numero % 10;
            suma = suma + ultimo;
            digitos++;
            numero = numero / 10;
        }
        printf("Tiene %d digitos y suman %d\n", digitos, suma);
    }

    return 0;
}
```

Explicación: la idea es "pelar" el número digito por digito desde la derecha. $\text{numero} \% 10$ te da el último digito ($473 \% 10 = 3$) y $\text{numero} / 10$ le quita ese último digito ($473 / 10 = 47$, porque la división entera trunca). En cada vuelta del while: tomas el último digito, lo sumas al acumulador, cuentas un dígito más, y le quitas el último digito al número. Sigue la ejecución con 473: vuelta 1: último = 3, suma = 3, digitos = 1, número = 47. Vuelta 2: último = 7, suma = 10, digitos = 2, número = 4. Vuelta 3: último = 4, suma = 14, digitos = 3, número = 0. Como número ya no es mayor que 0, el while termina. Resultado: 3 dígitos que suman 14. Este es un while natural: no sabias de antemano cuántas vueltas darian porque depende de cuántos digitos tenga el número. El bucle se repite "mientras queden digitos", que es exactamente lo que dice la condición $\text{número} > 0$.

Solución del ejercicio 6

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

int main() {
    int secreto;
    int intento;
    int max_intentos = 7;
    int adivino = 0;

    srand(time(0));
    secreto = rand() % 100 + 1;

    printf("Pense un numero del 1 al 100. Tienes %d intentos.\n", max_intentos);

    for (int i = 1; i <= max_intentos; i++) {
        printf("Intento %d de %d: ", i, max_intentos);
        scanf("%d", &intento);

        if (intento == secreto) {
            printf("Correcto. Lo lograste en %d intentos.\n", i);
            adivino = 1;
            break;
        } else if (intento < secreto) {
            printf("Muy bajo.\n");
        } else {
            printf("Muy alto.\n");
        }
    }

    if (!adivino) {
        printf("Se acabaron los intentos. El numero era %d.\n", secreto);
    }

    return 0;
}
```

Explicación: como ahora el número de intentos es fijo (7), el for es el bucle natural: for (int i = 1; i <= max_intentos; i++). En cada vuelta se pide el intento y se compara. Si adivina, se marca la bandera adivino = 1 y break sale del for antes de gastar los intentos restantes. Si se acaban las 7 vueltas sin adivinar, el for termina solo y la bandera sigue en 0, así que se revela el secreto. Compara con el ejemplo 4: allá usamos do-while porque los intentos eran ilimitados

("repite mientras no adivine"); aquí usamos for porque el límite es conocido ("repite 7 veces como máximo"). Esa es exactamente la guía práctica de cuando usar cada bucle, aplicada a un caso real. Como reto extra, agrega una opción de jugar otra vez usando un do-while alrededor de todo el juego.

Resumen de la unidad

Llegaste al final de la unidad 5 y ahora controlas una de las herramientas más poderosas de la programación: la repetición. Vamos a repasar lo esencial para que quede bien grabado.

Los bucles le dicen a la computadora que repita un bloque de instrucciones mientras se cumpla una condición. La computadora no se cansa, no se aburre y no se equivoca al repetir, y eso convierte a los bucles en la herramienta ideal para procesar listas, hacer cálculos largos y automatizar tareas tediosas.

C tiene tres bucles. El while revisa la condición antes de cada vuelta: si es falsa desde el inicio, el bloque no se ejecuta ninguna vez. Su forma es while (condición) { ... } y es ideal cuando no sabes cuántas vueltas darán o cuando es válido que no de ninguna. El do-while ejecuta el bloque primero y revisa la condición después: el bloque se ejecuta al menos una vez garantizado. Su forma es do { ... } while (condición); con punto y coma al final, y es ideal para menus y para pedir datos por primera vez. El for junta la inicialización, la condición y la actualización en una sola línea: for (inicialización; condición; actualización) { ... }. Es el favorito cuando sabes cuántas veces quieres repetir, y la variable contadora por convención se llama i.

Todo bucle sano necesita tres piezas: inicialización de la variable de control antes del bucle, una condición que pueda volverse falsa, y una actualización dentro del bucle que acerque a la condición hacia su fin. Si falta alguna, el bucle se vuelve infinito o se comporta de forma impredecible.

Los bucles se pueden anidar: un bucle dentro de otro. Cada vuelta del bucle externo ejecuta completo el bucle interno. Sirven para tablas, matrices, patrones y comparaciones de todos contra todos. Usa nombres claros como fila y columna para no perderte.

break sale del bucle inmediatamente, sin terminar la vuelta actual. Sirve para dejar de buscar cuando ya encontraste, y para salir de bucles infinitos intencionales. Solo afecta al bucle más interno. continue salta el resto de la vuelta actual y pasa a la siguiente: sirve para descartar los casos que no te interesan sin anidar else. Cuidado con continue dentro de while si la actualización del contador está al final del bloque.

El bucle infinito casi siempre es un error causado por olvidar actualizar el contador, por una condición mal planteada o por el punto y coma fantasma después del while o del for. Pero también existe como técnica legítima: while (1) o for (;;) con un break alcanzable dentro, usado en menus, servidores y videojuegos.

La guía para elegir: for cuando sabes cuántas vueltas son ("repite N veces"), while cuando no lo sabes y la condición se revisa antes ("repite mientras pase esto"), do-while cuando el bloque debe ejecutarse al menos una vez ("haz esto y luego pregunta si seguimos").

Los errores que debes vigilar: olvidar actualizar el contador, el punto y coma después de la cabecera del for o while, el off-by-one de confundir < con <=, usar = en vez de == en la condición, y modificar la variable del for dentro del bloque. Las buenas prácticas que te protegen: inicializar siempre, preferir for para conteos, mantener bloques cortos, probar los casos borde (cero, una y muchas vueltas), y compilar con gcc -Wall.

En los ejemplos construiste programas reales: contar con los tres bucles, tabla de multiplicar, suma con acumulador, el juego de adivinar el número, patrones con asteriscos, factorial, Fibonacci y un menu interactivo. Esos patrones (acumulador, bandera, valor centinela, bucle infinito con break, deslizamiento de variables) los vas a reutilizar en todo el resto del curso.

Antes de cerrar, llevate esta lista de verificación mental para cada bucle que escribas de aquí en adelante. Son cinco preguntas que toman diez segundos y que te salvan de la mayoría de los errores: primera, inicialice la variable de control antes del bucle? Segunda, la condición puede volverse falsa en algún momento? Tercera, algo dentro del bloque modifica la variable de la condición? Cuarta, use < o <= según lo que realmente necesito, y conte cuántas vueltas da? Quinta, si es un while (1) o un for (;;), hay un break alcanzable que garantice la salida? Si respondes que sí a las cinco, tu bucle tiene el 95 por ciento de probabilidades de funcionar a la primera. El otro 5 por ciento son los detalles que solo se aprenden rompiendo programas y arreglandolos, así que no le tengas miedo a experimentar.

Y un último consejo de programador a programador: cuando un bucle no haga lo que esperas, no te quedes mirando el código esperando que te hable. Agrega prints temporales dentro del bucle que muestren el valor de tus variables en cada vuelta. Ver los números cambiar vuelta por vuelta revela el error casi siempre: un contador que no avanza, una condición que nunca se cumple, un acumulador que empieza mal. Esa técnica de "imprimir para entender" es la depuración más antigua y más efectiva que existe, y en la clase 29 vas a conocer herramientas más avanzadas. Por ahora, el printf es tu mejor detective.

Lo que sigue: en la unidad 6 vas a aprender funciones, que te permiten empaquetar bloques de código con nombre para reutilizarlos. Combinadas con los bucles que ya dominas, tus programas van a subir de nivel: bucles que llaman funciones, funciones que contienen bucles,

y menus donde cada opción es una función. Si esta unidad te hizo sentir que ya programas de verdad, la que sigue te va a hacer sentir que programas como profesional. Sigue practicando, y nos vemos en la unidad 6 en lucasegaacademy.com.

UNIDAD 6: Funciones

Introducción

Llegaste a la unidad que va a cambiar tu forma de programar para siempre. Hasta ahora has escrito programas donde todo pasa dentro de `main`, de arriba hacia abajo, como una carretera recta. Eso funciona cuando el programa es chico, pero imagina que te piden escribir el código de un cajero automatico: tiene que leer la tarjeta, pedir el PIN, mostrar el menu, consultar el saldo, permitir retiros, hacer depositos, imprimir tickets, manejar errores... Si pones todo eso en un solo `main`, terminas con mil líneas enredadas donde un error chiquito se vuelve imposible de encontrar.

Aquí es donde entran las funciones, y la idea es más vieja que las computadoras. "Divide y vencerás": esa frase la uso Julio Cesar en la guerra, la usan los generales, los ingenieros y los buenos cocineros. Nadie cocina un banquete haciendo todo al mismo tiempo en un solo sartén gigante; divides el trabajo en platillos, cada platillo tiene su receta, y al final los juntas en la mesa. Programar con funciones es exactamente lo mismo: divides tu programa en pedazos pequeños, cada pedazo hace una sola cosa, y el programa grande solo se encarga de llamarlos en el orden correcto.

Piensalo con otra analogía: una caja de herramientas. No andas cargando el taladro, el martillo y el desarmador pegados a la mano todo el día; cada herramienta esta guardada en su lugar, y cuando necesitas clavar un clavo, sacas el martillo, lo usas y lo regresas. Las funciones son las herramientas de tu programa: las defines una vez, las guardas con un nombre claro, y cada vez que las necesitas las llamas. La herramienta no desaparece después de usarla: ahí sigue para la próxima vez.

Y eso nos lleva al superpoder más grande de las funciones: la reutilización. En las unidades anteriores ya usaste funciones sin darte cuenta: `printf`, `scanf`, `strlen`... todas son funciones que alguien más escribió una vez y que tú usas mil veces sin saber cómo funcionan por dentro. No necesitas saber como `printf` pinta letras en la pantalla; solo necesitas saber que le pasas y que te devuelve. Eso es abstracción: usar algo poderoso sin cargar con su complejidad. Cuando tú escribas tus propias funciones, otros programadores (o tú mismo dentro de tres meses) van a poder usarlas igual de fácil.

En esta unidad vas a aprender las tres piezas de una función en C: la declaración (el prototipo, que es como el anuncio de "esta función existe"), la definición (el cuerpo, donde vive el código de verdad) y la llamada (el momento en que la usas). Vas a entender la diferencia entre

parámetro y argumento, una distinción que confunde a casi todos al principio pero que aquí te va a quedar clarísima. Vas a ver como las funciones devuelven valores con return, para que sirven las funciones void que no devuelven nada, y el detalle más tramposo de C: el paso por valor, que significa que cuando le pasas una variable a una función, la función recibe una copia y el original no se toca.

También vas a dominar el alcance de las variables: donde nace una variable, donde puede usarse y donde muere. Vas a conocer las variables static, que son como variables con memoria que recuerdan su valor entre llamadas. Y vas a enfrentar al monstruo favorito de los exámenes y las entrevistas de trabajo: la recursividad, funciones que se llaman a si mismas. La vamos a desarmar paso a paso con trazas completas del factorial y de fibonacci, para que deje de ser magia negra y se vuelva una herramienta más.

Al final de esta unidad vas a escribir programas modulares de verdad: un menu principal chiquito que llama funciones, cada función con su tarea, código limpio que se puede leer, probar y arreglar por partes. Esa es la diferencia entre alguien que "sabe sintaxis" y alguien que "sabe programar". Los programadores profesionales no escriben programas largos: escriben muchas funciones cortas que trabajan en equipo. Bienvenido al siguiente nivel.

Si quieres reforzar estos temas con video, en lucasegaacademy.com tienes el curso completo de C con las clases 16, 17 y 18 dedicadas a funciones, con ejemplos en pizarron paso a paso.

Explicación detallada

Que es una función: declaración, definición y llamada

Una función es un bloque de código con nombre que hace una tarea especifica. Recibe datos de entrada (parámetros), hace su trabajo y puede devolver un resultado (valor de retorno). En C, todo programa tiene al menos una función: main. Hasta ahora main era tu única función; ahora vas a crear las tuyas.

Una función en C tiene tres momentos en su vida, y es importantísimo no confundirlos:

1. La declaración (también llamada prototipo): es el anuncio. Le dice al compilador "existe una función con este nombre, recibe estos tipos de datos y devuelve este tipo". Es como la portada de un libro: no cuenta la historia, solo dice que el libro existe y de que trata. El prototipo termina con punto y coma y normalmente va arriba del main o en un archivo de cabecera.

2. La definición: es el cuerpo real de la función, donde esta el código que hace el trabajo. Es el libro completo, capítulo por capítulo. Solo puede existir una definición por función en todo el programa.
3. La llamada: es el momento en que usas la función desde otra parte del código, normalmente desde main. Es cuando abres el libro y lo lees. Puedes llamar a una función cuántas veces quieras.

Mira la anatomía completa:

```
tipo_de_retorno nombre_de_la_funcion(tipo parametro1, tipo parametro2);
```

Esa línea con punto y coma es el prototipo. Y la definición se ve así:

```
tipo_de_retorno nombre_de_la_funcion(tipo parametro1, tipo parametro2)
{
    // cuerpo: las instrucciones que hacen el trabajo
    return valor; // si devuelve algo
}
```

Fíjate en las diferencias: el prototipo termina con punto y coma y no tiene cuerpo; la definición no lleva punto y coma después del paréntesis y trae las llaves con el código. La llamada es simplemente el nombre con los valores entre paréntesis, por ejemplo `suma(5, 3);`.

Por que el prototipo es necesario? Porque C compila de arriba hacia abajo. Si main llama a una función que está definida más abajo en el archivo, cuando el compilador llega a la llamada todavía no sabe que esa función existe y te marca error. El prototipo arriba le avisa con tiempo. Hay dos soluciones: poner el prototipo arriba del main, o definir la función completa antes del main. Ambas funcionan; los profesionales prefieren prototipos porque dejan el main al principio del archivo, que es lo primero que uno quiere leer.

El nombre de una función sigue las mismas reglas que los nombres de variables: letras, números y guion bajo, sin espacios, sin empezar con número. Y por convención se escriben en minúsculas, con guion bajo para separar palabras: `calcular_promedio`, `leer_entero`, `mostrar_menu`. Un buen nombre dice lo que la función hace: si tienes que leer el cuerpo para saber que hace, el nombre está mal.

Parámetros vs argumentos: la diferencia que casi todos confunden

Parámetro y argumento no son lo mismo, aunque mucha gente los usa como sinonimos. La diferencia es sencilla si la ves con la analogía de una receta:

- Los parámetros son los ingredientes genericos que aparecen en la receta: "harina", "azucar", "huevos". Son los nombres que pones en la definición de la función, como variables de mentira que solo existen para explicar que necesita la función.
- Los argumentos son los ingredientes reales que pones cuando cocinas: "2 tazas de harina marca X", "3 huevos". Son los valores concretos que pasas en la llamada.

Ejemplo:

```
int sumar(int a, int b)    // a y b son PARAMETROS
{
    return a + b;
}

int main()
{
    int resultado = sumar(5, 3); // 5 y 3 son ARGUMENTOS
}
```

Cuando llamas a `sumar(5, 3)`, C toma el valor 5 y lo mete en el parámetro a, toma el 3 y lo mete en el parámetro b, ejecuta el cuerpo y devuelve 8. Los argumentos pueden ser valores literales (5, 3), variables (x, y) o expresiones ($x + 1$, $y * 2$): C primero calcula el valor de la expresión y luego lo pasa.

El número y el tipo de los argumentos tienen que coincidir con los parámetros. Si la función espera dos int y le pasas un int y un float, C intenta convertirlo, pero eso es pedir problemas: mejor pasa exactamente lo que pide. Y el orden importa: `sumar(5, 3)` no es lo mismo que `restar(5, 3)` con los argumentos al revés.

Un detalle fino: los parámetros son variables locales de la función. Nacen cuando la función es llamada y mueren cuando la función termina. Eso conecta con el tema del alcance que veremos más adelante.

Valor de retorno y return

Muchas funciones hacen un cálculo y te devuelven el resultado. Eso se declara con el tipo de retorno antes del nombre: `int sumar(...)` devuelve un entero, `float promedio(...)` devuelve un decimal, `char inicial(...)` devuelve un carácter. Dentro del cuerpo, la instrucción `return` seguida del valor es la que hace la entrega.

`return` hace dos cosas al mismo tiempo: entrega el valor y termina la función inmediatamente. Todo lo que este después de un `return` que se ejecuta ya no se corre. Por eso puedes tener

varios return en una función (por ejemplo, en distintas ramas de un if), pero solo uno se ejecutara por llamada.

```
int mayor(int a, int b)
{
    if (a > b)
        return a;    // si entra aqui, la funcion termina y entrega a
    return b;        // si no, entrega b
}
```

El valor que devuelves tiene que ser compatible con el tipo declarado. Si la función dice int y haces return 3.7;, C trunca el decimal y devuelve 3, sin avisarte mucho. Los warnings del compilador (que veremos en la unidad 9) te ayudan a cachar estos detalles: compila siempre con -Wall.

Puedes usar el valor devuelto directamente: guardarlo en una variable (int r = sumar(5, 3);), imprimirlo (printf("%d", sumar(5, 3));), o usarlo dentro de otra expresión (int doble = sumar(5, 3) * 2;). Si una función devuelve un valor y tu lo ignoras en la llamada, C no se queja, pero normalmente es señal de que algo anda mal en tu lógica.

Funciones void: las que no devuelven nada

No todas las funciones calculan algo. Algunas solo hacen una acción: imprimir un menu, mostrar un mensaje, guardar datos. Esas se declaran con tipo de retorno void, que significa "vacío", "nada".

```
void saludar(void)
{
    printf("Hola, bienvenido al programa\n");
}
```

Fíjate en el (void): significa que la función no recibe parámetros. En C, escribir solo () también funciona, pero (void) es más explícito y es la forma recomendada porque deja claro que la ausencia de parámetros es intencional.

Una función void no lleva return con valor. Puede llevar un return; solito (sin nada después) para salir antes de tiempo, por ejemplo si detecta un error y ya no quiere seguir. Pero lo normal es que simplemente llegue al final de las llaves y termine.

Cuando usar void y cuando devolver valor? Preguntate: "esta función calcula algo que el resto del programa necesita, o solo hace algo?". Si calcula, devuelve el valor. Si solo muestra,

guarda o modifica algo, void es lo correcto. Un error común de principiantes es hacer funciones void que imprimen el resultado en vez de devolverlo; eso amarra tu función a la pantalla y no la puedes reutilizar. Mejor: la función calcula y devuelve, y el main decide si lo imprime, lo guarda o lo usa en otro cálculo.

Paso por valor: C siempre pasa copias

Este es uno de los conceptos más importantes y más tramposos de C, así que pon atención. En C, los argumentos SIEMPRE se pasan por valor. Eso significa que cuando llamas a una función pasándole una variable, la función NO recibe tu variable: recibe una COPIA del valor. Es como si le sacaras una fotocopia a un documento y se la dieras a alguien: esa persona puede rayar la fotocopia todo lo que quiera, pero tu documento original sigue intacto.

Mira este ejemplo clásico que confunde a todos los principiantes:

```
void intentar_cambiar(int x)
{
    x = 100;
    printf("Dentro de la funcion: x = %d\n", x);
}

int main()
{
    int numero = 5;
    intentar_cambiar(numero);
    printf("Despues de la llamada: numero = %d\n", numero);
    return 0;
}
```

Que imprime? "Dentro de la función: x = 100" y después "Después de la llamada: número = 5". El 5 original NO cambio. ¿Por qué? Porque el parámetro x es una variable nueva, una copia que vive solo dentro de intentar_cambiar. Cambiar x no toca a número para nada. Son dos cajitas distintas en la memoria con el mismo valor al inicio, y nada más.

Esto tiene una consecuencia de diseño enorme: si quieres que una función modifique una variable del que la llama, no basta con pasarle la variable. Necesitas pasarle la dirección de la variable usando apuntadores, y eso lo verás en la Unidad 7 (clases 22 a 24). Por ahora quedate con la regla de oro: en C, las funciones reciben copias; lo que pase dentro de la función, se queda dentro de la función.

El paso por valor también es una protección: puedes estar tranquilo de que llamar a una función no va a corromper tus variables por accidente. La función trabaja con sus copias y tus

datos originales están a salvo. Cuando quieras modificar de verdad, lo harás de forma explícita con apuntadores, y esa explicitud es una de las cosas que hacen a C un lenguaje honesto.

Alcance de variables: donde vive cada variable

El alcance (scope) de una variable es la region del programa donde esa variable existe y puede usarse. En C hay tres niveles que debes dominar:

Variables locales: se declaran dentro de una función (o dentro de un bloque) y solo existen ahí. Nacen cuando se entra a la función y mueren cuando la función termina. Cada llamada a la función crea variables locales nuevas, frescas, sin memoria de llamadas anteriores. Los parámetros también son variables locales. Dos funciones distintas pueden tener variables locales con el mismo nombre sin conflicto: son cajitas separadas.

```
void funcion_a(void)
{
    int x = 10; // x solo existe dentro de funcion_a
    printf("%d\n", x);
}

void funcion_b(void)
{
    int x = 99; // otra x distinta, no hay conflicto
    printf("%d\n", x);
}
```

Variables globales: se declaran FUERA de todas las funciones, normalmente arriba del todo del archivo. Existen durante todo el programa y cualquier función puede leerlas y modificarlas. Suenan comodas, pero son peligrosas: cualquier función puede cambiarlas en cualquier momento, y cuando algo sale mal no sabes quién fue el culpable. Es como dejar la cartera abierta sobre la mesa en una fiesta: cualquiera la puede tocar.

La regla profesional es: evita las variables globales tanto como puedas. Si una función necesita un dato, pasaselo como parámetro; si produce un resultado, devuelvelo con return. Hay excepciones legítimas (constantes globales de configuración, por ejemplo), pero como principiante, tu meta debe ser cero variables globales. En la sección de buenas prácticas lo reforzamos.

Variables de bloque: en C puedes abrir llaves { } en cualquier parte (por ejemplo dentro de un if o un for) y declarar variables ahí dentro. Esas variables solo existen dentro de esas llaves. Esto es útil para variables temporales muy específicas:

```

int main()
{
    int x = 5;
    {
        int y = 10; // y solo existe dentro de estas llaves
        printf("%d %d\n", x, y);
    }
    // aqui y ya no existe; usarla seria error de compilacion
    printf("%d\n", x);
    return 0;
}

```

Y hay una regla de sombreado (shadowing) que debes conocer: si declaras una variable local con el mismo nombre que una global, la local "tapa" a la global dentro de esa función. El compilador no te avisa y tu te confundes. Evita reutilizar nombres: ponle nombres distintos y claros a cada cosa.

El tiempo de vida es distinto del alcance: una variable local muere cuando su función termina, así que nunca devuelvas la dirección de una variable local (eso lo entenderás a fondo en la unidad de apuntadores; por ahora, grábatelo como regla). Y nunca uses una variable fuera de su alcance: el compilador te lo va a impedir, y eso es bueno.

Variables static: las que tienen memoria

Las variables locales normales son amnesicas: cada vez que llamas a la función, nacen de cero. Pero si declaras una variable local con la palabra static, se comporta distinto: se inicializa una sola vez y CONSERVA su valor entre llamadas. Es como una variable local con memoria de elefante.

```

void contador(void)
{
    static int llamadas = 0; // se inicializa UNA sola vez
    llamadas++;
    printf("Esta funcion se ha llamado %d veces\n", llamadas);
}

int main()
{
    contador(); // imprime 1
    contador(); // imprime 2
    contador(); // imprime 3
    return 0;
}

```

Sin el static, llamadas siempre imprimiría 1, porque nacería en cero en cada llamada. Con static, la variable vive durante todo el programa (como una global en cuanto a tiempo de vida), pero su alcance sigue siendo local: solo la función contador puede verla. Es lo mejor de los dos mundos para casos como contadores de llamadas, acumuladores o estados internos.

Ojo con dos detalles: primero, una variable static local solo se puede inicializar con un valor constante (static int x = 5; está bien, static int x = y; no compila). Segundo, no abuses de static: si una función depende de un valor guardado de la llamada anterior, se vuelve más difícil de entender y de probar, porque su resultado depende de la historia. Usala cuando de verdad la necesites, no por flojera de pasar parámetros.

También existe static para funciones y variables globales, donde significa otra cosa (visibilidad limitada al archivo), pero eso es tema avanzado que verás cuando trabajes con varios archivos fuente. Por ahora, static local es tu herramienta nueva.

Recursividad: funciones que se llaman a si mismas

La recursividad es cuando una función se llama a si misma para resolver un problema. Suena a truco de magia o a error, pero es una técnica legítima y poderosa, ideal para problemas que se pueden partir en "un pedacito más el mismo problema en chiquito". La definición clásica: para entender la recursividad, primero hay que entender la recursividad.

Toda función recursiva necesita DOS partes obligatorias:

1. El caso base: la condición que DETIENE la recursión. Sin caso base, la función se llama infinitamente hasta que se acaba la memoria y el programa truena (stack overflow). Es como una escalera: necesitas el piso para dejar de bajar.

2. El caso recursivo: la llamada a si misma con un problema MÁS CHIQUITO, acercandose al caso base. Si el problema no se achica, nunca llegas al caso base y también truenas.

El ejemplo clásico es el factorial. El factorial de n (escrito $n!$) es $n * (n-1) * ... * 2 * 1$. Por ejemplo, $5! = 5 * 4 * 3 * 2 * 1 = 120$. Fíjate que $5! = 5 * 4!$, y $4! = 4 * 3!$... cada factorial se define en terminos de un factorial más chiquito. Esa es la forma recursiva natural: $n! = n * (n-1)!$, y el caso base es $0! = 1$ (por definición matematica).

```
int factorial(int n)
{
    if (n == 0)          // CASO BASE: aqui se detiene todo
        return 1;
    return n * factorial(n - 1); // CASO RECURSIVO: problema mas chiquito
}
```

Vamos a trazar factorial(3) paso a paso, como si fueras el compilador. Cada llamada crea sus propias variables locales (recuerda el alcance), así que hay varios n distintos viviendo al mismo tiempo:

- Llamada 1: factorial(3). n vale 3. No es 0, así que calcula $3 * \text{factorial}(2)$. Pero para multiplicar necesita primero saber cuánto vale factorial(2), así que se pausa y hace la llamada 2.
- Llamada 2: factorial(2). n vale 2. No es 0, así que calcula $2 * \text{factorial}(1)$. Se pausa y hace la llamada 3.
- Llamada 3: factorial(1). n vale 1. No es 0, así que calcula $1 * \text{factorial}(0)$. Se pausa y hace la llamada 4.
- Llamada 4: factorial(0). n vale 0. CASO BASE: devuelve 1 directamente, sin más llamadas.

Ahora se deshace la pila en reversa, como una fila de dominos que cae hacia atras:

- La llamada 3 recibe el 1, calcula $1 * 1 = 1$ y devuelve 1.
- La llamada 2 recibe el 1, calcula $2 * 1 = 2$ y devuelve 2.
- La llamada 1 recibe el 2, calcula $3 * 2 = 6$ y devuelve 6.

Resultado: factorial(3) = 6. Correcto: $3 * 2 * 1 = 6$.

Cada llamada en espera ocupa memoria en la pila (stack): guarda sus variables locales y donde debe continuar. Por eso la recursión profunda con números grandes puede agotar la memoria. El factorial crece rapidísimo además: $12!$ ya es 479,001,600 y $13!$ no cabe en un int de 32 bits (se desborda). Para jugar con recursión usa números chicos.

El segundo ejemplo clásico es fibonacci: la serie donde cada número es la suma de los dos anteriores: 0, 1, 1, 2, 3, 5, 8, 13, 21... La definición recursiva es: $\text{fib}(0) = 0$, $\text{fib}(1) = 1$ (dos casos base), y $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$.

```
int fibonacci(int n)
{
    if (n == 0) return 0; // caso base 1
    if (n == 1) return 1; // caso base 2
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Tracemos $\text{fibonacci}(4)$: la llamada se ramifica como un árbol porque hay DOS llamadas recursivas por nivel.

- $\text{fib}(4)$ necesita $\text{fib}(3) + \text{fib}(2)$.
- $\text{fib}(3)$ necesita $\text{fib}(2) + \text{fib}(1)$. $\text{fib}(1)$ es caso base: devuelve 1.
- $\text{fib}(2)$ necesita $\text{fib}(1) + \text{fib}(0)$: $1 + 0 = 1$.
- Entonces $\text{fib}(3) = 1 + 1 = 2$.
- El otro $\text{fib}(2)$ (el de $\text{fib}(4)$) también vale 1.
- Entonces $\text{fib}(4) = 2 + 1 = 3$. Correcto: la serie es 0, 1, 1, 2, 3.

Fíjate en algo importante: $\text{fib}(2)$ se calcula DOS veces, y con números más grandes la repetición explota. $\text{fibonacci}(30)$ hace más de un millón de llamadas. La recursión de fibonacci es elegante pero ineficiente; en la práctica se resuelve con un ciclo o guardando resultados (memoización). La lección: la recursión es hermosa para expresar la idea, pero siempre pregúntate si hay una versión iterativa más eficiente. Muchos problemas recursivos se pueden reescribir con for o while, y a veces debes hacerlo.

Cuando usar recursión entonces? Cuando el problema tiene estructura naturalmente recursiva: recorrer árboles y directorios de archivos, algoritmos como quicksort o búsqueda binaria, rompecabezas como las Torres de Hanoi, y definiciones matemáticas como factorial. Y en entrevistas de trabajo te van a pedir recursión si o si, así que domina la traza paso a paso: es la habilidad que separa a quien memorizo de quien entendió.

Buenas prácticas con funciones

Escribir funciones que compilan es fácil; escribir funciones que otros humanos pueden leer y mantener es el verdadero oficio. Estas reglas te van a servir toda tu carrera:

Una función, una tarea. Si tu función se llama `calcular_promedio` pero también imprime el menú y guarda el archivo, está haciendo tres trabajos y su nombre miente. Partela en tres funciones. Una buena prueba: si no puedes describir lo que hace la función en una frase corta sin usar la palabra "y", es muy grande. Las funciones ideales miden entre 10 y 30 líneas; si pasa de 50, casi seguro se puede partir.

Nombres claros y honestos. El nombre debe decir que hace la función: `leer_entero`, `es_par`, `convertir_a_mayusculas`. Usa verbos para acciones (calcular, mostrar, validar) y preguntas para las que devuelven si/no (`es_valido`, `tiene_saldo`). Evita nombres genericos como `hacer_cosa`, `funcion1`, `proceso` o `temp`: no dicen nada y dentro de un mes ni tu los vas a entender.

Evita las variables globales. Ya lo dijimos en alcance, pero vale repetirlo como regla de diseño: una función debe comunicarse con el mundo solo por sus parámetros y su valor de retorno. Así la puedes probar sola, reusarla en otro programa y entenderla sin leer todo el código. Las globales convierten tu programa en una telarania donde todo puede afectar a todo.

No mezcles cálculo con entrada/salida. Una función que calcula el promedio no debería pedir los datos con `scanf` ni imprimir con `printf`; debería recibir los números como parámetros y devolver el promedio. El `main` (o una función dedicada) se encarga de pedir e imprimir. Así tu función de cálculo sirve igual en un programa de consola, en un videojuego o en un sistema embebido sin pantalla.

Valida tus parámetros. Si tu función divide, revisa que el divisor no sea cero. Si recibe un tamaño de arreglo, revisa que sea positivo. No confíes en que "el que la llama lo va a hacer bien": los errores más caros nacen de suposiciones. Una función robusta se defiende sola.

Manten los prototipos ordenados arriba. Pon todos los prototipos juntos después de los `#include` y antes del `main`, con un comentario corto si hace falta. Así cualquiera que abra tu archivo ve el "menú" de funciones disponibles de un vistazo, como el índice de un libro.

Errores comunes con funciones (y como evitarlos)

Olvidar el prototipo. Llamas a una función definida después del `main` sin haber declarado el prototipo arriba, y el compilador se queja (en compiladores viejos asumía que devolvía `int` y causaba errores raros). Solución: escribe el prototipo siempre, es un hábito de dos segundos que te ahorra dolores de cabeza.

Declarar un tipo de retorno y no devolver nada en todos los caminos. Si tu función dice `int` pero en alguna rama del `if` se te olvida el `return`, la función devuelve basura: lo que haya quedado en el registro del procesador. Compila con `-Wall` y el compilador te avisa. Regla: cada camino posible de la función debe terminar en un `return` con valor.

Crear que la función modifique tu variable. El clásico del paso por valor: pasas número a una función que lo cambia adentro, y afuera sigue igual. No es un bug del compilador: es C funcionando como debe. Si necesitas modificar el original, la respuesta son los apuntadores (Unidad 7).

Recursión sin caso base (o sin acercarse a él). La función se llama infinitamente, la pila se llena y el programa muere con `stack overflow`. Antes de escribir una función recursiva, escribe primero el caso base en papel y verifica que cada llamada recursiva achique el problema.

Confundir `=` con `==` en un `return` o condición. `return a = b`; asigna en vez de comparar, y casi nunca es lo que quieres. Los warnings también cachan esto.

Tipos incompatibles entre parámetros y argumentos. Pasar un `float` donde se espera un `int`, o al revés, causa conversiones silenciosas y resultados raros. Revisa que cada argumento corresponda en tipo y orden con su parámetro.

Recursividad infinita por caso base inalcanzable. Por ejemplo, factorial con `n` negativo: `n` nunca llega a 0 restando de uno en uno... bueno, si llega pasando por negativos hasta desbordar, pero el resultado es basura y tarda una eternidad. Valida la entrada antes de llamar a la función recursiva.

La pila de llamadas: qué pasa por dentro cuando llamas a una función

Hasta ahora has usado funciones como cajas mágicas: las llamas y hacen su trabajo. Pero entender qué pasa por dentro cuando hay una llamada te va a dar una intuición que pocos principiantes tienen, y te va a servir para depurar errores raros y para entender la recursión a fondo.

Cada vez que tu programa llama a una función, C hace lo siguiente, en este orden:

1. Pausa la función actual justo en el punto de la llamada y guarda donde debe continuar (la dirección de retorno).
2. Reserva espacio en una zona de memoria llamada la pila (stack) para las variables locales y los parámetros de la función llamada. A ese paquete se le llama marco de pila (stack frame).

3. Copia los valores de los argumentos en los parámetros (el paso por valor que ya conoces).
4. Salta a ejecutar el cuerpo de la función.
5. Cuando la función termina (por return o por llegar al final), toma el valor devuelto, libera el marco de pila (las variables locales mueren aquí) y regresa al punto guardado para continuar.

La pila funciona como una pila de platos: el último que entra es el primero que sale (LIFO). Si main llama a A, y A llama a B, y B llama a C, los marcos se apilan: main abajo, luego A, luego B, luego C arriba. Cuando C termina, se desapila y B continúa; cuando B termina, A continúa; cuando A termina, main continúa. Cada función en espera está pausada, con sus variables intactas, esperando su turno.

Esto explica varias cosas que ya viste. Primero, por que las variables locales mueren al terminar la función: su marco de pila se libera. Segundo, por que la recursión profunda puede agotar la memoria: cada llamada recursiva apila un marco nuevo sin desapilar los anteriores, y si son miles, la pila se llena (stack overflow). Tercero, por qué nunca debes usar la dirección de una variable local después de que la función terminó: ese espacio de memoria ya se recicla y otra función lo puede estar usando.

Hay un límite práctico: la pila suele medir unos pocos megabytes. Para variables normales es muchísimo, pero para recursión infinita o arreglos locales gigantescos no alcanza. Cuando en la Unidad 7 veas memoria dinámica (malloc), entenderás la alternativa: pedir memoria fuera de la pila, en el heap, que vive hasta que tu la liberes.

Un experimento mental útil: cuando leas código con varias llamadas anidadas, dibuja la pila en papel. Cada llamada es un rectángulo apilado con sus variables dentro. Tacha el rectángulo cuando la función retorna. Si puedes hacer ese dibujo, puedes trazar cualquier programa, por enredado que parezca. Los depuradores profesionales (como gdb, que verás en la unidad de depuración) te muestran exactamente esta pila cuando el programa truena: se llama "stack trace" o "backtrace", y leerlo es una de las habilidades más valiosas que vas a adquirir.

Diseño de funciones: cohesión y acoplamiento

Ya sabes escribir funciones; ahora hablemos de diseñarlas bien, que es lo que separa el código de estudiante del código profesional. Dos palabras técnicas que valen oro: cohesión y acoplamiento.

Cohesión significa "que tan enfocada esta una función en una sola cosa". Alta cohesión es bueno: la función hace una tarea y la hace bien. Baja cohesión es malo: la función hace de

todo un poco. Preguntate siempre: "puedo describir esta función en una frase sin usar la palabra 'y'"? Si la descripción necesita un "y" ("calcula el promedio Y lo imprime Y guarda el archivo"), partela. Cada "y" es una función esperando nacer.

Acoplamiento significa "que tanto depende una función del resto del programa". Bajo acoplamiento es bueno: la función solo necesita sus parámetros y devuelve su resultado, sin leer globales ni depender del estado de otras funciones. Alto acoplamiento es malo: la función solo funciona si antes se llamó a otra, o si una variable global tiene cierto valor. Las funciones con bajo acoplamiento se pueden probar solas, reusar en otros proyectos y entender sin leer todo el programa.

Mira la diferencia con un ejemplo. Versión acoplada (mala):

```
int total; // global: cualquiera la puede tocar

void procesar(void)
{
    // lee total global, lo modifica, imprime...
    // para probarla necesitas preparar el global antes
}
```

Versión desacoplada (buena):

```
int procesar(int valor)
{
    // recibe lo que necesita, devuelve el resultado
    // se prueba sola: procesar(5) siempre da lo mismo
}
```

La segunda función es predecible: con la misma entrada siempre da la misma salida. A esas funciones se les llama puras, y son las más fáciles de probar y las menos propensas a bugs. No siempre puedes hacer todo puro (alguien tiene que hablar con el usuario y con los archivos), pero la estrategia profesional es: empuja toda la lógica hacia funciones puras y deja la entrada/salida en una capa delgada arriba (el main y unas pocas funciones de interfaz).

Otra regla de diseño: el número de parámetros. Una función con 6 o 7 parámetros es sospechosa: probablemente está haciendo demasiado o le falta una estructura que agrupe esos datos (los structs de la Unidad 7 resuelven eso elegantemente). Tres o menos parámetros es lo ideal; más de cuatro, piensalo dos veces.

Y un consejo sobre el tamaño: si tu función no cabe en una pantalla (más de 40 o 50 líneas), casi seguro se puede dividir. Las funciones cortas se leen de un vistazo, se prueban fácil y se

nombran mejor, porque una tarea pequeña tiene un nombre obvio. "Divide y vencerás" no era solo el lema de la introducción: es la técnica de diseño más importante de la programación.

Recursión vs iteración: cuando usar cada una

Ya viste que el factorial se puede escribir recursivo. Pero también se puede escribir con un ciclo for, y de hecho así es como la mayoría de los programadores lo haría en código real:

```
int factorial_iterativo(int n)
{
    int resultado = 1;
    int i;
    for (i = 2; i <= n; i++)
        resultado = resultado * i;
    return resultado;
}
```

Compara las dos versiones. La recursiva expresa la definición matemática casi literal: $n! = n * (n-1)!$. La iterativa es más eficiente: no apila marcos en la pila, usa memoria constante y corre más rápido. Entonces, cuál elegir? Depende del problema, y aquí van las guías prácticas:

Usa recursión cuando el problema tiene estructura naturalmente recursiva y la versión iterativa sería enredada. Los casos clásicos: recorrer árboles (como el sistema de archivos de tu disco: una carpeta contiene carpetas que contienen carpetas), algoritmos de división y conquista como quicksort o búsqueda binaria, y rompecabezas como las Torres de Hanoi. En esos casos la recursión no es un lujo: es la forma clara de pensar el problema, y la versión con ciclos termina siendo más larga y más difícil de entender.

Usa iteración cuando el problema es una repetición simple: sumar una serie, recorrer un arreglo, repetir algo n veces. Ahí un for es más rápido, usa menos memoria y es más fácil de depurar. La recursión de fibonacci es el ejemplo perfecto de cuando NO usar recursión: es elegante en el pizarrón pero calcula lo mismo millones de veces. La versión iterativa de fibonacci es trivial y rapidísima:

```

int fibonacci_iterativo(int n)
{
    if (n == 0) return 0;
    int anterior = 0, actual = 1, i, temp;
    for (i = 2; i <= n; i++)
    {
        temp = anterior + actual;
        anterior = actual;
        actual = temp;
    }
    return actual;
}

```

Esta versión calcula fibonacci(100) en un parpadeo; la recursiva tardaría más tiempo que la edad del universo (bueno, casi). La lección no es "la recursión es mala", sino "elige la herramienta según el problema". Un buen programador conoce ambas y sabe cuando cada una brilla.

Hay un tercer camino intermedio que vale la pena conocer de nombre: la memoización, que es guardar los resultados ya calculados para no repetirlos. Convierte la recursión lenta de fibonacci en una rápida, manteniendo la elegancia. No la implementaremos aquí (necesita arreglos, que ya dominas de la Unidad 5, y un poco más de diseño), pero cuando la veas en el futuro, sabrás exactamente que problema resuelve.

Y una advertencia final sobre recursión: en C no hay optimización garantizada de "recursión de cola" (cuando la llamada recursiva es lo último que hace la función). Algunos compiladores la aplican, otros no, así que no escribas recursión profunda confiando en que el compilador la va a convertir en ciclo. Si la profundidad puede ser de miles de llamadas, mejor escribe el ciclo tu directamente.

Prototipos y archivos de cabecera: el siguiente paso

Hasta ahora todos tus programas viven en un solo archivo .c. Eso está bien para aprender, pero los programas reales se dividen en varios archivos: uno para la lógica de matemáticas, otro para el manejo de archivos, otro para la interfaz, etc. Cómo se conectan? Con archivos de cabecera (.h) que contienen los prototipos.

La idea es simple: si tienes funciones de matemáticas en matemáticas.c, creas matemáticas.h con solo los prototipos:

```
/* matematicas.h */
int sumar(int a, int b);
int restar(int a, int b);
float promedio(float a, float b);
```

Y en tu programa principal haces `#include "matemáticas.h"` (con comillas, no con `<>`, porque es TU archivo, no uno del sistema). El compilador ve los prototipos y confía en que las definiciones existen en otro archivo; el enlazador (linker) las junta al final. Así puedes tener un equipo donde una persona escribe las funciones matemáticas y otra las usa, sin necesidad de ver el código interno: el `.h` es el contrato entre ustedes.

No vamos a compilar multi-archivo en este libro todavía (eso llega con proyectos más grandes), pero entiende el concepto desde hoy porque explica POR QUÉ existen los prototipos: no son burocracia, son el mecanismo que permite que C escale de programas de juguete a sistemas operativos. Cada vez que escribes `#include <stdio.h>`, estás incluyendo los prototipos de `printf`, `scanf` y docenas de funciones más, escritos por alguien hace décadas. Tus propios `.h` serán igual de útiles cuando tus proyectos crezcan.

Mientras tanto, la disciplina de "prototipos arriba, definiciones abajo" en un solo archivo es el entrenamiento perfecto: cuando partas tu código en varios archivos, cada `.h` será simplemente la colección de prototipos que ya sabes escribir.

Cómo depurar funciones: técnicas que si funcionan

Tarde o temprano una de tus funciones va a devolver un valor incorrecto y no vas a saber por qué. En vez de mirar el código fijamente esperando que el bug se confiese solo, usa estas técnicas, en este orden:

1. Imprime las entradas y salidas. La técnica más simple y más efectiva: al inicio de la función imprime los parámetros que recibió, y antes de cada `return` imprime lo que va a devolver. El 80% de los bugs de funciones se revelan aquí: descubres que la función recibió un valor distinto al que creías, o que devuelve algo inesperado. Cuando encuentres el bug, quita los prints (o mejor, dejalos comentados por si los necesitas de nuevo).

```
int dividir_seguro(int a, int b)
{
    printf("DEBUG: dividir_seguro(%d, %d)\n", a, b); // quitar al final
    if (b == 0)
    {
        printf("DEBUG: division entre cero\n");
        return 0;
    }
    return a / b;
}
```

2. Verifica la llamada, no solo la función. Muchas veces la función está bien y el bug está en COMO la llamas: argumentos en el orden incorrecto, tipo equivocado, o una variable que no vale lo que crees en ese punto. Imprime los argumentos justo antes de la llamada y compáralos con lo que la función reporta haber recibido.
3. Aísla la función en un programa de prueba chiquito. Copia la función a un archivo nuevo con un main mínimo que la llame con valores conocidos. Si ahí funciona, el bug está en el resto de tu programa; si ahí falla, ya tienes un caso pequeño y reproducible para arreglarla. Esta técnica se llama "reproducir el bug en mínimo" y es lo primero que hace un profesional.
4. Revisa los caminos del return. Si tu función tiene varios if con varios return, verifica mentalmente (o con prints) cuál camino se está tomando. Un bug clásico: el caso que tú pruebas nunca entra al if que tú crees, y se va por el else que devuelve otra cosa.
5. Para recursión, imprime la profundidad. Agrega un parámetro extra (o una variable static) que indique el nivel de la llamada e imprímelo con indentación. Ver el árbol de llamadas dibujado en la consola hace obvios los casos donde la recursión no se detiene o se ramifica de más.

```
int factorial_debug(int n, int nivel)
{
    printf("%sllamada factorial(%d)\n", nivel * 2, "", n);
    if (n == 0) return 1;
    int r = n * factorial_debug(n - 1, nivel + 1);
    printf("%sretorna %d\n", nivel * 2, "", r);
    return r;
}
```

El truco %s imprime espacios según el nivel, así cada nivel de recursión se ve indentado. Correló con $n = 4$ y verás el árbol completo.

6. Compila siempre con `-Wall -Wextra`. El compilador es tu primer depurador y es gratis: te avisa de funciones usadas sin prototipo, valores de retorno faltantes, comparaciones sospechosas y variables sin usar. Ignorar los warnings es como ignorar el dolor: el problema no se va, solo se esconde hasta que duele más.

La mentalidad correcta para depurar es científica: formula una hipótesis ("creo que `b` llega en cero"), diseña un experimento (imprime `b`), observa el resultado y ajusta. Nunca cambies código al azar esperando que funcione: eso introduce bugs nuevos sobre el viejo. Un bug bien entendido se arregla en minutos; un bug atacado a ciegas te roba la tarde.

Ejemplos paso a paso

Ejemplo 1: Tu primera función (saludar)

Empezamos con lo más simple: una función void que no recibe nada y no devuelve nada, solo hace una acción. Copialo, compílalo con `gcc ejemplo1.c -o ejemplo1 -Wall` y corrélo.

```
#include <stdio.h>

void saludar(void)
{
    printf("Hola, bienvenido a la Unidad 6\n");
    printf("Hoy aprendes funciones en C\n");
}

int main()
{
    saludar();    // primera llamada
    saludar();    // segunda llamada: la función sigue ahí, lista
    return 0;
}
```

Que pasa aquí: definimos `saludar` antes del `main`, así que no necesitamos prototipo (el compilador ya la conoce cuando llega a `main`). La llamamos dos veces y cada llamada ejecuta las dos líneas del cuerpo. Si mañana quieres cambiar el saludo, lo cambias en UN solo lugar y las dos llamadas muestran el nuevo texto. Esa es la magia de no repetir código: un solo punto de verdad.

Ejemplo 2: Función que devuelve un valor (suma)

Ahora una función que recibe parámetros y devuelve un resultado. Fíjate como el main usa el valor devuelto de tres formas distintas.

```
#include <stdio.h>

int sumar(int a, int b)
{
    int resultado = a + b;
    return resultado;
}

int main()
{
    int x = 10, y = 20;

    int r1 = sumar(x, y);           // guardado en variable
    printf("Suma guardada: %d\n", r1);

    printf("Suma directa: %d\n", sumar(7, 8)); // usada al imprimir

    int doble = sumar(x, y) * 2;    // usada en una expresion
    printf("Doble de la suma: %d\n", doble);

    return 0;
}
```

Observa que `sumar(x, y)` pasa los VALORES de `x` e `y` (10 y 20), no las variables. Dentro de `sumar`, `a` y `b` son copias locales. Y el valor devuelto por `return` se puede usar como cualquier otro valor: guardarlo, imprimirlo o meterlo en una cuenta más grande.

Ejemplo 3: Prototipos (declarar arriba, definir abajo)

Este ejemplo muestra el estilo profesional: prototipos arriba del main, definiciones abajo. El main queda al principio del archivo, que es lo primero que lees.


```

#include <stdio.h>

// PROTOTIPOS: el menu de funciones disponibles
int sumar(int a, int b);
int restar(int a, int b);
void mostrar_resultado(int valor);

int main()
{
    int s = sumar(15, 27);
    int r = restar(15, 27);
    mostrar_resultado(s);
    mostrar_resultado(r);
    return 0;
}

// DEFINICIONES: el codigo real, abajo y en orden
int sumar(int a, int b)
{
    return a + b;
}

int restar(int a, int b)
{
    return a - b;
}

void mostrar_resultado(int valor)
{
    printf("El resultado es: %d\n", valor);
}

```

Prueba quitar uno de los prototipos y compilar: verás el error del compilador quejandose de función no declarada. Vuelvelo a poner y todo funciona. Asi de importante es ese renglon con punto y coma. Acostumbrate a escribir primero los prototipos y luego el main: es el esqueleto de todo programa modular.

Ejemplo 4: Paso por valor demostrado (el original no cambia)

Este es el ejemplo que debes correr y releer hasta que te salga natural. Demuestra con hechos que C pasa copias.

```

#include <stdio.h>

void duplicar(int n)
{
    n = n * 2;
    printf("Dentro de duplicar: n = %d\n", n);
}

int main()
{
    int numero = 21;

    printf("Antes de llamar: numero = %d\n", numero);
    duplicar(numero);
    printf("Despues de llamar: numero = %d\n", numero);

    // Prueba extra: pasar una expresion tambien funciona,
    // porque solo se necesita el VALOR
    duplicar(numero + 4); // pasa 25, imprime 50
    printf("El original sigue valiendo: %d\n", numero);

    return 0;
}

```

Salida:

```

Antes de llamar: numero = 21
Dentro de duplicar: n = 42
Despues de llamar: numero = 21
Dentro de duplicar: n = 50
El original sigue valiendo: 21

```

La variable `n` dentro de `duplicar` y la variable `número` en `main` son dos cajitas distintas. Cambiar `n` no toca `número`. Si esto te sorprende, es normal: casi todos los lenguajes modernos se comportan distinto en algunos casos, pero en C la regla es clara y sin excepciones: se pasa el valor, punto. Cuando en la Unidad 7 aprendas apuntadores, verás la técnica para modificar el original a propósito.

Ejemplo 5: Calculadora con funciones (programa modular chico)

Aquí juntamos varias funciones en un programa útil: una calculadora que suma, resta, multiplica y divide. Cada operación es una función; el main solo pide datos y muestra resultados.

```
#include <stdio.h>

float sumar(float a, float b)    { return a + b; }
float restar(float a, float b)   { return a - b; }
float multiplicar(float a, float b){ return a * b; }

float dividir(float a, float b)
{
    if (b == 0)
    {
        printf("Error: no se puede dividir entre cero\n");
        return 0;
    }
    return a / b;
}

int main()
{
    float x, y;

    printf("Dame el primer numero: ");
    scanf("%f", &x);
    printf("Dame el segundo numero: ");
    scanf("%f", &y);

    printf("%.2f + %.2f = %.2f\n", x, y, sumar(x, y));
    printf("%.2f - %.2f = %.2f\n", x, y, restar(x, y));
    printf("%.2f * %.2f = %.2f\n", x, y, multiplicar(x, y));
    printf("%.2f / %.2f = %.2f\n", x, y, dividir(x, y));

    return 0;
}
```

Fíjate en dividir: válida su parámetro antes de operar (buena práctica: la función se defiende sola). Y nota el diseño: las funciones de cálculo no piden ni imprimen datos (salvo el mensaje de error), solo calculan y devuelven. El main es el que habla con el usuario. Si mañana quieres usar sumar en otro programa, la copias tal cual y funciona.

Ejemplo 6: static en acción (función con memoria)

Compara una función con variable local normal contra una con static. Correlo y observa la diferencia.

```
#include <stdio.h>

void sin_memoria(void)
{
    int contador = 0;    // nace en 0 en CADA llamada
    contador++;
    printf("sin_memoria: %d\n", contador);
}

void con_memoria(void)
{
    static int contador = 0; // se inicializa UNA sola vez
    contador++;
    printf("con_memoria: %d\n", contador);
}

int main()
{
    printf("Llamando tres veces a cada una:\n");
    sin_memoria(); // siempre imprime 1
    sin_memoria();
    sin_memoria();

    con_memoria(); // imprime 1, 2, 3
    con_memoria();
    con_memoria();

    return 0;
}
```

La variable static conserva su valor entre llamadas porque vive durante todo el programa, aunque solo con_memoria puede verla. Usos reales: contar cuántas veces se llamó a una función, generar números de folio consecutivos, o guardar un estado interno. Recuerda no abusar: si el resultado de tu función depende de llamadas anteriores, documentalo con un comentario.

Ejemplo 7: Factorial recursivo con traza impresa

Este ejemplo hace visible lo invisible: imprime cada llamada y cada retorno para que veas la pila de recursión en acción.

```
#include <stdio.h>

int factorial(int n)
{
    printf("Llamada: factorial(%d)\n", n);

    if (n == 0)
    {
        printf("Caso base: devuelvo 1\n");
        return 1;
    }

    int sub = factorial(n - 1);    // llamada recursiva
    int resultado = n * sub;
    printf("Retorno: factorial(%d) = %d * %d = %d\n", n, n, sub, resultado);
    return resultado;
}

int main()
{
    int numero = 4;
    printf("Calculando %d!\n\n", numero);
    int f = factorial(numero);
    printf("\nResultado final: %d! = %d\n", numero, f);
    return 0;
}
```

Salida (resumida):

Calculando 4!

```
Llamada: factorial(4)
Llamada: factorial(3)
Llamada: factorial(2)
Llamada: factorial(1)
Llamada: factorial(0)
Caso base: devuelvo 1
Retorno: factorial(1) = 1 * 1 = 1
Retorno: factorial(2) = 2 * 1 = 2
Retorno: factorial(3) = 3 * 2 = 6
Retorno: factorial(4) = 4 * 6 = 24

Resultado final: 4! = 24
```

Mira el patrón: primero una cascada de llamadas hacia abajo (4, 3, 2, 1, 0) y luego una cascada de retornos hacia arriba. Cada llamada espera pausada a que termine la siguiente. Esa es la pila de llamadas funcionando frente a tus ojos. Cuando escribas tus propias funciones recursivas, agrega prints así para depurar: es la mejor forma de ver si tu caso base se alcanza y si el problema se achica.

Ejemplo 8: Programa modular completo (menu con funciones)

El ejemplo final junta todo: prototipos, funciones void y con retorno, paso por valor, validación de parámetros y un main limpio que solo coordina. Es el modelo a seguir para tus programas de ahora en adelante.

```
#include <stdio.h>

// Prototipos
void mostrar_menu(void);
int leer_opcion(void);
float pedir_numero(char letra);
float sumar(float a, float b);
float restar(float a, float b);
float multiplicar(float a, float b);
float dividir(float a, float b);

int main()
{
    int opcion;

    do
```

```

{
    mostrar_menu();
    opcion = leer_opcion();

    if (opcion >= 1 && opcion <= 4)
    {
        float x = pedir_numero('X');
        float y = pedir_numero('Y');
        float r = 0;

        if (opcion == 1) r = sumar(x, y);
        else if (opcion == 2) r = restar(x, y);
        else if (opcion == 3) r = multiplicar(x, y);
        else r = dividir(x, y);

        printf("Resultado: %.2f\n\n", r);
    }
    else if (opcion != 5)
    {
        printf("Opcion invalida, intenta de nuevo.\n\n");
    }

} while (opcion != 5);

printf("Gracias por usar la calculadora. Hasta luego.\n");
return 0;
}

void mostrar_menu(void)
{
    printf("=== CALCULADORA MODULAR ===\n");
    printf("1. Sumar\n");
    printf("2. Restar\n");
    printf("3. Multiplicar\n");
    printf("4. Dividir\n");
    printf("5. Salir\n");
}

int leer_opcion(void)
{
    int op;
    printf("Elige una opcion: ");
    scanf("%d", &op);
    return op;
}

```

```

float pedir_numero(char letra)
{
    float n;
    printf("Dame el numero %c: ", letra);
    scanf("%f", &n);
    return n;
}

float sumar(float a, float b)      { return a + b; }
float restar(float a, float b)     { return a - b; }
float multiplicar(float a, float b) { return a * b; }

float dividir(float a, float b)
{
    if (b == 0)
    {
        printf("Error: division entre cero.\n");
        return 0;
    }
    return a / b;
}

```

Estudia la estructura: el main no hace cuentas ni imprime el menu; solo orquesta. Cada función tiene una sola tarea y un nombre que la describe. Si quieres agregar la operación "potencia", solo escribes la función, su prototipo y una opción más en el menu: el resto del programa ni se entera. Asi se construye software que crece sin romperse. Este es el nivel al que apunta todo lo que aprendas en lucasegaacademy.com: programas chicos por fuera, bien organizados por dentro.

Práctica

Escribe, compila y corre cada ejercicio antes de ver la solución. La regla de oro de esta unidad: si no lo programaste tu, no lo aprendiste. Usa gcc con -Wall para que el compilador te avise de descuidos.

1. Escribe una función `es_par` que reciba un entero y devuelva 1 si es par y 0 si es impar. Pruebala en main con varios números, incluyendo negativos y el cero.
2. Escribe una función `potencia` que reciba una base entera y un exponente entero no negativo, y devuelva la base elevada al exponente usando un ciclo for (sin usar la función `pow` de `math.h`). Pruebala con base 2 y exponente 10 (debe dar 1024).

3. Demuestra el paso por valor: escribe una función `poner_en_cero` que reciba un entero e intente ponerlo en 0. En main, crea una variable con valor 50, llama a la función e imprime la variable antes y después. Explica con tus palabras por que no cambio.
4. Escribe una función recursiva `suma_naturales` que reciba `n` y devuelva la suma de $1 + 2 + \dots + n$. Identifica claramente el caso base y el caso recursivo. Pruebala con `n = 5` (debe dar 15) y con `n = 100` (debe dar 5050).
5. Escribe una función `contar_llamadas` que use una variable static para llevar la cuenta de cuántas veces ha sido llamada, e imprima el número de llamada cada vez. Llamala 5 veces desde main dentro de un ciclo for.
6. Mini-proyecto: escribe un programa modular para convertir temperaturas. Debe tener las funciones `celsius_a_fahrenheit`, `fahrenheit_a_celsius`, `mostrar_menu` y `leer_opcion`, más un main con ciclo do-while que ofrezca las dos conversiones y la opción de salir. Las fórmulas son: $F = C * 9/5 + 32$ y $C = (F - 32) * 5/9$. Cuidado con la división entera: usa `9.0/5.0`.

Solución del ejercicio 1

```

#include <stdio.h>

int es_par(int n);

int main()
{
    int pruebas[] = {4, 7, 0, -3, -8, 15};
    int i;

    for (i = 0; i < 6; i++)
    {
        if (es_par(pruebas[i]))
            printf("%d es par\n", pruebas[i]);
        else
            printf("%d es impar\n", pruebas[i]);
    }

    return 0;
}

int es_par(int n)
{
    return n % 2 == 0;
}

```

Explicación: la expresión `n % 2 == 0` ya vale 1 (verdadero) o 0 (falso) en C, así que la devolvemos directo sin necesidad de un `if`. Funciona con negativos porque en C el residuo de un número par negativo entre 2 es 0 (por ejemplo, `-8 % 2 == 0`). El cero también es par. Nota el estilo: la función no imprime nada, solo responde la pregunta; el `main` decide que hacer con la respuesta. Así debe ser.

Solución del ejercicio 2

```

#include <stdio.h>

int potencia(int base, int exp);

int main()
{
    printf("2^10 = %d\n", potencia(2, 10)); // 1024
    printf("5^3 = %d\n", potencia(5, 3));   // 125
    printf("7^0 = %d\n", potencia(7, 0));   // 1
    return 0;
}

int potencia(int base, int exp)
{
    int resultado = 1;
    int i;

    for (i = 0; i < exp; i++)
    {
        resultado = resultado * base;
    }

    return resultado;
}

```

Explicación: multiplicamos la base por si misma exp veces, empezando el acumulador en 1 (el neutro de la multiplicación). El caso $\text{exp} = 0$ funciona solo: el ciclo no se ejecuta y se devuelve 1, que es lo matemáticamente correcto. Si el exponente fuera negativo este código fallaría, por eso el ejercicio pedía exponente no negativo: una función honesta documenta lo que espera. Podrías agregar una validación que devuelva 0 o un mensaje si $\text{exp} < 0$.

Solución del ejercicio 3

```

#include <stdio.h>

void poner_en_cero(int n)
{
    n = 0;
    printf("Dentro de la funcion: n = %d\n", n);
}

int main()
{
    int valor = 50;

    printf("Antes de llamar: valor = %d\n", valor);
    poner_en_cero(valor);
    printf("Despues de llamar: valor = %d\n", valor);

    return 0;
}

```

Explicación: valor sigue valiendo 50 después de la llamada. ¿Por qué? Porque C pasa argumentos por valor: el parámetro n es una copia independiente que vive solo dentro de poner_en_cero. Asignar n = 0 modifica la copia, no el original. Es como darle a alguien una fotocopia de tu credencial: puede rayarla, pero tu credencial sigue intacta. Si quieres modificar el original necesitas apuntadores, que verás en la Unidad 7.

Solución del ejercicio 4

```

#include <stdio.h>

int suma_naturales(int n);

int main()
{
    printf("Suma del 1 al 5: %d\n", suma_naturales(5));    // 15
    printf("Suma del 1 al 100: %d\n", suma_naturales(100)); // 5050
    return 0;
}

int suma_naturales(int n)
{
    if (n <= 1)                // CASO BASE: la suma hasta 1 es 1
        return 1;
    return n + suma_naturales(n - 1); // CASO RECURSIVO
}

```

Explicación: el caso base es $n \leq 1$ (usamos $\leq$ en vez de $==$ para protegernos si alguien pasa 0 o negativo: así la función termina en vez de llamarse eternamente). El caso recursivo achica el problema: $\text{suma_naturales}(5) = 5 + \text{suma_naturales}(4)$, y así hasta llegar al 1. Traza corta: $\text{suma}(3) = 3 + \text{suma}(2) = 3 + (2 + \text{suma}(1)) = 3 + 2 + 1 = 6$. Cada llamada espera a la siguiente, igual que en el factorial. Para $n = 100$ hay 100 llamadas anidadas, que la pila aguanta sin problema.

Solución del ejercicio 5

```

#include <stdio.h>

void contar_llamadas(void)
{
    static int cuenta = 0; // se inicializa una sola vez
    cuenta++;
    printf("Esta es la llamada numero %d\n", cuenta);
}

int main()
{
    int i;

    for (i = 0; i < 5; i++)
    {
        contar_llamadas();
    }

    return 0;
}

```

Explicación: la variable cuenta es static, así que no se reinicia en cada llamada: conserva su valor entre una llamada y la siguiente. La salida es "llamada número 1" hasta "llamada número 5". Sin el static, siempre imprimiría 1. Recuerda que static se inicializa una sola vez (la primera vez que se ejecuta la función) y que solo contar_llamadas puede ver esa variable: su alcance sigue siendo local aunque su tiempo de vida sea todo el programa.

Solución del ejercicio 6

```

#include <stdio.h>

void mostrar_menu(void);
int leer_opcion(void);
float celsius_a_fahrenheit(float c);
float fahrenheit_a_celsius(float f);

int main()
{
    int opcion;

    do
    {
        mostrar_menu();
    }
}

```

```

    opcion = leer_opcion();

    if (opcion == 1)
    {
        float c;
        printf("Grados Celsius: ");
        scanf("%f", &c);
        printf("%.2f C = %.2f F\n\n", c, celsius_a_fahrenheit(c));
    }
    else if (opcion == 2)
    {
        float f;
        printf("Grados Fahrenheit: ");
        scanf("%f", &f);
        printf("%.2f F = %.2f C\n\n", f, fahrenheit_a_celsius(f));
    }
    else if (opcion != 3)
    {
        printf("Opcion invalida.\n\n");
    }

} while (opcion != 3);

printf("Hasta luego.\n");
return 0;
}

void mostrar_menu(void)
{
    printf("=== CONVERSION DE TEMPERATURA ===\n");
    printf("1. Celsius a Fahrenheit\n");
    printf("2. Fahrenheit a Celsius\n");
    printf("3. Salir\n");
}

int leer_opcion(void)
{
    int op;
    printf("Elige: ");
    scanf("%d", &op);
    return op;
}

float celsius_a_fahrenheit(float c)
{

```

```

    return c * 9.0 / 5.0 + 32;
}

float fahrenheit_a_celsius(float f)
{
    return (f - 32) * 5.0 / 9.0;
}

```

Explicación: este es el programa modular completo en miniatura. El main solo coordina: muestra el menu, lee la opción y delega. Las funciones de conversión solo calculan y devuelven, sin pedir ni imprimir datos: son puras y reutilizables. El detalle tramposo era la división entera: en C, 9/5 con enteros da 1 (trunca el decimal), así que hay que escribir 9.0/5.0 para forzar división de flotantes. Si hubieras puesto $c * 9/5 + 32$, con $c = 100$ obtendrías 132 en vez de 212. Ese es exactamente el tipo de bug silencioso que los warnings no siempre cachan y que solo se evita entendiendo los tipos.

Resumen de la unidad

Llegaste al final de la unidad más transformadora del curso hasta ahora. Ya no programas en una sola carretera recta: ahora divides los problemas en funciones pequeñas que trabajan en equipo. Repasemos lo esencial:

- Una función es un bloque de código con nombre que hace una tarea específica. Tiene tres momentos: la declaración o prototipo (el anuncio, con punto y coma, arriba del main), la definición (el cuerpo real con el código) y la llamada (cuando la usas).
- Los parámetros son los nombres genericos en la definición (los ingredientes de la receta); los argumentos son los valores concretos en la llamada (los ingredientes reales que cocinas). El número, tipo y orden deben coincidir.
- return entrega un valor y termina la función al instante. El tipo devuelto debe coincidir con el declarado. Las funciones void no devuelven nada y sirven para acciones como mostrar menus o mensajes.
- En C los argumentos se pasan POR VALOR: la función recibe una copia. Lo que pase dentro de la función se queda dentro de la función; el original no cambia. Para modificar el original necesitarás apuntadores (Unidad 7).
- El alcance define donde vive cada variable: las locales existen solo dentro de su función, las globales existen en todo el programa (evitalas tanto como puedas) y las de bloque solo dentro de sus llaves. No devuelvas ni uses variables fuera de su alcance.

- Las variables static locales se inicializan una sola vez y conservan su valor entre llamadas. Útiles para contadores y estados internos; usalas con medida.
- La recursividad es una función que se llama a si misma. Necesita un caso base que detenga y un caso recursivo que achique el problema. Traza factorial y fibonacci paso a paso hasta que la pila de llamadas te salga natural, y recuerda que la recursión elegante a veces es ineficiente: considera la versión con ciclo.
- Buenas prácticas: una función, una tarea; nombres claros con verbos; nada de globales; separa cálculo de entrada/salida; válida tus parámetros; prototipos ordenados arriba.
- Errores que ya no vas a cometer: olvidar el prototipo, no devolver valor en todos los caminos, creer que la función cambio tu variable, recursión sin caso base, y la división entera accidental (9/5 en vez de 9.0/5.0).

En la Unidad 7 entramos al territorio más famoso (y temido) de C: los apuntadores. Ahí entenderás por fin como lograr que una función modifique la variable original, y verás la relación íntima entre apuntadores y arreglos. Todo lo que aprendiste aquí (prototipos, parámetros, alcance) es la base sobre la que se construye eso.

Sigue practicando todos los días: reescribe los ejemplos sin verlos, inventa tus propias funciones y arma programas modulares cada vez más grandes. Si quieres ver estos temas explicados en video con pizarrón, los tienes en el curso de C de lucasegaacademy.com, clases 16, 17 y 18. Nos vemos en la Unidad 7.

Autoevaluación: verifica que dominas la unidad

Antes de pasar a la Unidad 7, responde estas preguntas sin ver el texto. Si fallas alguna, regresa a su sección y repasa:

- 1.Cuál es la diferencia entre declaración, definición y llamada de una función? Donde va el punto y coma?
2. En `int sumar(int a, int b)` ejecutado como `sumar(5, 3)`, cuáles son los parámetros y cuáles los argumentos?
3. Qué hacen las dos cosas que logra la instrucción `return`?
4. Para que sirve una función void y que significa el (void) en los paréntesis?
5. Si pasas una variable a una función y la función la modifica adentro, cambia el original? ¿Por qué?
- 6.Cuál es la diferencia entre una variable local, una global y una de bloque?Cuál debes evitar y ¿por qué?

7. Que tiene de especial una variable static local? Donde se inicializa y cuántas veces?
8. Cuáles son las dos partes obligatorias de una función recursiva? Qué pasa si falta el caso base?
9. Traza en papel factorial(4) y fibonacci(5): cuántas llamadas se hacen en cada caso?
10. Por que la versión recursiva de fibonacci es ineficiente y cuál es la alternativa?
11. Qué significan cohesión alta y acoplamiento bajo? Da un ejemplo de cada uno.
12. Que información te da un stack trace cuando un programa truena?
13. Por que $c * 9/5 + 32$ da un resultado incorrecto para convertir temperaturas y cómo se arregla?
- 14.Cuál es la primera técnica de depuración que debes aplicar cuando una función devuelve un valor incorrecto?

Si respondiste las 14 con seguridad, felicidades: piensas como programador modular. Si alguna te costó, no es derrota: es tu mapa de repaso. Las funciones son el tema que más se pregunta en exámenes y entrevistas de programación en C, así que vale la pena dejarlo solido. En la Unidad 7, los apuntadores te van a dar el poder de modificar variables desde dentro de las funciones, y todo lo que aprendiste aquí (prototipos, parámetros por valor, alcance, la pila de llamadas) va a encajar como piezas de rompecabezas.

UNIDAD 7: Arreglos y cadenas de texto

Introducción

Imaginate que eres el encargado de un salón de clases y tienes que guardar la calificación de cada alumno del grupo. Son 40 alumnos. Hasta ahora, en este curso, sabes crear variables: una para cada dato. Así que te sientas y escribes algo así:

```
int calif1 = 85;
int calif2 = 92;
int calif3 = 78;
...
int calif40 = 90;
```

Cuarenta variables. Cuarenta líneas. Y ahora te piden calcular el promedio. Tendrías que sumar las cuarenta variables una por una, a mano, en el código. Después te piden decir cuál fue la calificación más alta. Tendrías que comparar cuarenta variables con un montón de ifs. Y si el grupo crece a 200 alumnos, necesitas 200 variables y 200 líneas. Es una locura, verdad? Pues esa locura es exactamente lo que viven los programadores principiantes cuando no conocen los arreglos.

Ahora piensa en la alternativa. Piensa en los casilleros de una escuela: una pared con 40 casilleros numerados, todos iguales, todos juntos, uno al lado del otro. Para guardar la calificación del alumno número 7, abres el casillero 7 y la metes ahí. Para leerla, abres el casillero 7 y la sacas. Para sumar todas las calificaciones, pasas del casillero 0 al casillero 39, uno por uno, y vas acumulando. No necesitas 40 nombres de variables: necesitas UN solo nombre (el del conjunto de casilleros) y un número que diga cuál casillero quieres. Eso, mi amigo, es un arreglo.

Y cuando necesitas algo más grande, como una tabla completa con filas y columnas (piensa en la hoja de cálculo donde llevas las calificaciones de todo el semestre: cada fila es un alumno, cada columna es un examen), usas un archivero con varios cajones y cada cajón tiene compartimentos: eso es un arreglo multidimensional, una matriz. La analogía de los casilleros y el archivero te va a acompañar toda esta unidad, porque es la forma más natural de entender lo que la computadora hace con la memoria.

En esta unidad vas a aprender una de las herramientas más importantes de C y de la programación en general: los arreglos. Verás como declararlos, como llenarlos con datos, como recorrerlos con ciclos for (vas a descubrir que los arreglos y los ciclos for nacieron el uno

para el otro), como calcular su tamaño con el operador sizeof, y como trabajar con matrices de dos dimensiones, que no son más que arreglos de arreglos. Vas a ver también que en C no existe un tipo de dato llamado "texto" o "cadena" como en otros lenguajes: las cadenas de texto SON arreglos de caracteres, con un carácter especial al final que marca donde terminan. Entender esto te va a explicar un monton de cosas raras que antes parecian magia.

Por qué importa tanto este tema? Porque casi ningún programa real trabaja con un solo dato a la vez. Los programas trabajan con listas: la lista de contactos de tu celular, las canciones de tu playlist, los productos del carrito de compras, los mensajes del chat, las calificaciones del grupo, los pixeles de una imagen (qué es una matriz gigante de colores), los caracteres de un texto. Todo eso, en C, se maneja con arreglos y cadenas. Dominar esta unidad es dar un salto enorme: vas a dejar de escribir programas de juguete que manejan tres variables y vas a empezar a escribir programas que manejan cientos o miles de datos con unas cuantas líneas de código.

Además, lo que aprendas aquí es la base directa de la Unidad 8, donde entran los apuntadores. Los apuntadores y los arreglos están tan relacionados en C que casi son primos hermanos, y entender bien los arreglos ahora te va a ahorrar mucho dolor de cabeza después. También vas a tocar por primera vez un tema serio de seguridad: el desbordamiento de buffer, que es uno de los errores más famosos y peligrosos de la historia de la computación. Vas a entender por qué pasa y como evitarlo.

Un aviso honesto antes de empezar: en esta unidad vas a cometer errores. Todos los cometemos. Vas a salirte del rango del arreglo (pedir el casillero 40 cuando solo hay 40, del 0 al 39), vas a olvidar el carácter de terminación de las cadenas, vas a comparar textos con == y te vas a preguntar por que no funciona. No te frustres: cada uno de esos errores te va a enseñar algo profundo sobre cómo funciona la memoria. Al final de la unidad vas a mirar atras y te vas a reir de lo que antes te parecía imposible.

Esta unidad cubre las clases 19, 20 y 21 del curso de video de lucasegaacademy.com: Arreglos unidimensionales, Arreglos multidimensionales, y Cadenas de texto y string.h. En el curso en video ves cada tema en clips cortos; aquí en el libro lo tienes todo desarrollado con más detalle, más analogias, más errores comunes y más práctica. Si vienes del video, esto es tu repaso profundo. Si solo lees el libro, no te falta nada: cada concepto esta explicado desde cero.

Vamos a abrir los casilleros.

Explicación detallada

Arreglos unidimensionales: tu pared de casilleros

Un arreglo es una colección de variables del MISMO tipo que viven juntas en la memoria, una detrás de otra, y que comparten un solo nombre. En lugar de tener calif1, calif2, calif3, tienes un solo nombre, por ejemplo calificaciones, y accedes a cada elemento con un número entre corchetes que se llama índice.

La declaración de un arreglo tiene esta forma:

```
tipo nombre[tamano];
```

Por ejemplo:

```
int calificaciones[40];
```

Eso crea 40 variables enteras, todas juntas, bajo el nombre calificaciones. Fíjate en algo importante: el 40 entre corchetes es el TAMAÑO, la cantidad de elementos. No es el último índice. Este detalle confunde a todo el mundo al principio, así que grabatelo bien.

Los índices en C empiezan en CERO. Siempre. El primer elemento es calificaciones[0], el segundo es calificaciones[1], y el último de un arreglo de 40 elementos es calificaciones[39]. No existe calificaciones[40]: si lo pides, estás abriendo un casillero que no es tuyo, y C no te va a detener. Te va a dejar leer o escribir ahí, y eso puede corromper otros datos de tu programa o hacerlo fallar de formas muy extrañas. A este error se le llama salir del rango o desbordar el índice, y es el error número uno de los principiantes con arreglos.

Por que los índices empiezan en cero? La explicación técnica completa la verás en la Unidad 8 con los apuntadores, pero la versión corta es esta: el índice no es "el número del casillero" sino "cuántos casilleros me muevo desde el inicio". El primer elemento está a cero casilleros de distancia del inicio, por eso su índice es 0. El segundo está a un casillero de distancia, índice 1. Cuando veas apuntadores, esta idea te va a hacer clic perfecto.

Para guardar un valor en un elemento, lo tratas como una variable normal:

```
calificaciones[0] = 85;  
calificaciones[1] = 92;  
calificaciones[7] = 78;
```

Para leerlo, igual:

```
printf("La primera calificacion es: %d\n", calificaciones[0]);
```

Cada elemento del arreglo es una variable completa: puedes sumarla, compararla, pasarla a una función, hacer con ella todo lo que haces con una variable normal. La magia es que el índice puede ser una VARIABLE, y ahí es donde los arreglos se vuelven poderosos:

```
int i = 3;
calificaciones[i] = 100;    // guarda 100 en el elemento 3
printf("%d\n", calificaciones[i]); // imprime el elemento 3
```

Como el índice puede ser una variable, puedes usarlo dentro de un ciclo for para recorrer todo el arreglo elemento por elemento. Esto es tan común que casi siempre que veas un for en un programa de C, hay un arreglo cerca.

Inicializar un arreglo: llenar los casilleros desde el inicio

Cuando declaras un arreglo sin inicializarlo, sus elementos contienen basura: lo que sea que había antes en esa parte de la memoria. Es como abrir casilleros nuevos y encontrar papeles viejos de quien los uso antes. Nunca asumas que un arreglo empieza en ceros. Si necesitas ceros, ponlos tu.

Hay varias formas de inicializar un arreglo al declararlo:

```
int numeros[5] = {10, 20, 30, 40, 50};
```

Eso pone 10 en números[0], 20 en números[1], y así. Si das menos valores que el tamaño, el resto se llena con ceros automáticamente:

```
int datos[10] = {1, 2, 3}; // datos[0]=1, datos[1]=2, datos[2]=3, el resto es 0
```

Esa es una técnica muy usada: inicializar todo en cero de golpe:

```
int conteo[100] = {0}; // los 100 elementos quedan en cero
```

Con solo poner {0}, C llena TODO el arreglo con ceros. Es un truco que vas a usar muchísimo.

También puedes dejar que el compilador cuente por ti:

```
int primos[] = {2, 3, 5, 7, 11}; // el compilador deduce que el tamaño es 5
```

Esto funciona solo al declarar e inicializar en la misma línea. Después ya no puedes cambiar el tamaño: en C clásico, el tamaño de un arreglo es fijo para siempre. Si necesitas un tamaño

que cambie mientras el programa corre, eso se hace con memoria dinámica, que verás en la clase 27. Por ahora: declaras el tamaño una vez y es para siempre.

Ojo con un error clásico de principiantes: intentar asignar un arreglo completo de un golpe después de declararlo:

```
int a[5];  
a = {1, 2, 3, 4, 5}; // ERROR: esto no compila
```

No se puede. La inicialización con llaves solo funciona en la declaración. Después tienes que llenar elemento por elemento, normalmente con un ciclo.

Recorrer un arreglo con for: el duo dinámico

El ciclo for y los arreglos son la pareja perfecta de C. El for te da una variable que va de 0 hasta el último índice, y esa variable es exactamente lo que necesitas como índice del arreglo. Mira el patrón más importante de esta unidad:

```
int calificaciones[5] = {85, 92, 78, 90, 88};  
int i;  
  
for (i = 0; i < 5; i++) {  
    printf("Calificacion %d: %d\n", i + 1, calificaciones[i]);  
}
```

Fíjate en la condición: $i < 5$, NO $i \leq 5$. Como los índices van de 0 a 4, el ciclo debe detenerse cuando i llega a 5. Usar $i \leq 5$ leería `calificaciones[5]`, que no existe. Este patrón (empezar en 0, condición con $<$ tamaño, incrementar de uno en uno) lo vas a escribir miles de veces en tu vida como programador. Aprendetelo de memoria: `for (i = 0; i < N; i++)`.

Llenar un arreglo con datos del usuario es igual de natural:

```
int edades[10];  
int i;  
  
printf("Dame 10 edades:\n");  
for (i = 0; i < 10; i++) {  
    printf("Edad %d: ", i + 1);  
    scanf("%d", &edades[i]);  
}
```

Fíjate en el &: scanf necesita la DIRECCIón de donde guardar el dato, y &edades[i] es la dirección del elemento i. Cada elemento tiene su propia dirección, como cada casillero tiene su propia ubicación física. (En la Unidad 8 entenderás a fondo que significa esa dirección.)

Y para procesar los datos, otro ciclo:

```
int suma = 0;
for (i = 0; i < 10; i++) {
    suma = suma + edades[i];
}
printf("Suma total: %d\n", suma);
```

Tres ciclos casi idénticos: uno para leer, uno para procesar, uno para mostrar. Este esqueleto (leer, procesar, mostrar) aparece en una cantidad enorme de programas reales. Cuando lo domines, vas a sentir que puedes con cualquier lista de datos.

sizeof: preguntarle al arreglo cuánto mide

El operador sizeof te dice cuántos BYTES ocupa algo en memoria. Con arreglos es útil porque te permite calcular cuántos elementos tiene sin contarlos a mano:

```
int numeros[10];
int n = sizeof(numeros) / sizeof(numeros[0]);
```

Cómo funciona? sizeof(números) te da el total de bytes del arreglo completo (10 enteros por 4 bytes cada uno = 40 bytes). sizeof(números[0]) te da los bytes de UN elemento (4 bytes). Divides 40 / 4 y obtienes 10: el número de elementos. Esta división es un truco tan común que muchos programadores lo escriben hasta dormidos.

Pero OJO, hay una trampa importante que debes conocer desde ya: este truco SOLO funciona con el arreglo original, en el mismo lugar donde se declaró. Si pasas el arreglo a una función, ahí dentro el truco ya no sirve, porque lo que recibe la función no es el arreglo completo sino solo su dirección. En la clase 24 (apuntadores como parámetros) verás esto a fondo, y por eso la buena práctica es: cuando pases un arreglo a una función, pasa también su tamaño como otro parámetro. Grabatelo: arreglo y tamaño viajan juntos.

También puedes usar sizeof para no hardcodear el tamaño en tus ciclos:


```
int valores[] = {5, 10, 15, 20, 25};
int n = sizeof(valores) / sizeof(valores[0]);
int i;

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

Así, si mañana agregas más valores a la inicialización, el ciclo se ajusta solo. Menos números mágicos en el código, menos errores tontos.

Arreglos multidimensionales: el archivero con cajones y compartimentos

Un arreglo de una dimensión es una fila de casilleros. Pero que pasa si necesitas una TABLA, con filas y columnas? Por ejemplo, las calificaciones de 5 alumnos en 4 exámenes cada uno. Podrías hacer 5 arreglos separados, pero es incomodo. La solución es un arreglo de dos dimensiones, también llamado matriz:

```
int califs[5][4];
```

Leelo así: un arreglo de 5 filas, donde cada fila es un arreglo de 4 columnas. En total son 5 por 4 = 20 enteros. La analogía del archivero: tienes 5 cajones (filas) y cada cajón tiene 4 compartimentos (columnas). Para llegar a un dato necesitas DOS índices: `califs[fila][columna]`.

```
califs[0][0] = 85;    // alumno 0, examen 0
califs[0][1] = 90;    // alumno 0, examen 1
califs[2][3] = 77;    // alumno 2, examen 3
```

Los índices también empiezan en cero en cada dimensión: las filas van de 0 a 4 y las columnas de 0 a 3. Y también puedes salirte del rango en cualquiera de las dos dimensiones, con las mismas consecuencias desagradables.

La inicialización de una matriz se hace con llaves anidadas, una por fila:

```
int m[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Visualmente queda clarísimo: cada renglon de llaves es una fila. `m[0][0]` es 1, `m[0][2]` es 3, `m[2][1]` es 8. Dibujalo en papel la primera vez: haz una cuadrícula de 3x3 y escribe los números. Ver la matriz dibujada te ayuda a no confundir filas con columnas.

Si omites las llaves internas, también funciona, pero es menos legible:

```
int m[2][3] = {1, 2, 3, 4, 5, 6}; // fila 0: 1,2,3  fila 1: 4,5,6
```

C llena la matriz por filas: primero toda la fila 0, luego la fila 1, y así. A esto se le llama orden por filas, y es importante saberlo.

Para recorrer una matriz necesitas DOS ciclos for anidados: uno para las filas y otro para las columnas. El patrón es:

```
int i, j;
for (i = 0; i < 3; i++) {           // recorre las filas
    for (j = 0; j < 3; j++) {       // recorre las columnas de esa fila
        printf("%d ", m[i][j]);
    }
    printf("\n"); // salto de linea al terminar cada fila
}
```

El ciclo de afuera (i) elige la fila y se queda quieto mientras el ciclo de adentro (j) pasa por todas las columnas. Es cómo leer un libro: tu dedo baja renglon por renglon (ciclo externo), y tus ojos recorren cada renglon de izquierda a derecha (ciclo interno). Si inviertes el orden de los ciclos, recorres por columnas, que a veces es lo que quieres, pero para mostrar una matriz en pantalla el orden natural es filas afuera, columnas adentro.

Las matrices aparecen en todas partes: una imagen es una matriz de pixeles (cada pixel tiene fila y columna), un tablero de ajedrez o de gato es una matriz de 8x8 o 3x3, una hoja de cálculo es una matriz gigante, y en matemáticas e ingeniería las matrices son la herramienta diaria. Aprender a manejarlas con ciclos anidados te abre la puerta a graficos, juegos y simulaciones.

C también permite arreglos de 3 o más dimensiones (`int cubo[4][4][4]`), que sirven para cosas como volúmenes en 3D o datos por año, mes y día. Funcionan igual, con un índice por dimensión, pero son menos comunes. Con dominar bien las dos dimensiones tienes el 95 por ciento de los casos cubiertos.

Cadenas de texto: no hay tipo string, hay arreglos de char

Aquí viene una de las revelaciones más importantes de C: en este lenguaje NO existe un tipo de dato para texto. No hay un tipo string como en Python o Java. Lo que hay son arreglos de caracteres (char), y una convención para saber dónde termina el texto.

Un char guarda UN solo carácter: 'A', '7', '\$', ' '. Una cadena como "Hola" es simplemente un arreglo de 5 chars: 'H', 'o', 'l', 'a' y un carácter especial extra que marca el final. Ese carácter especial es el carácter nulo, que se escribe '\0' y que vale cero. NO es el carácter '0' (el dígito cero, que vale 48): es el valor cero, el "nada", la marca de "aquí se acabo el texto".

```
char saludo[6] = {'H', 'o', 'l', 'a', '\0'};
```

Fíjate: la palabra "Hola" tiene 4 letras, pero el arreglo necesita 5 espacios: 4 para las letras y 1 para el '\0'. Esta es LA regla de oro de las cadenas en C: siempre necesitas un espacio extra para el terminador. Si declaras char nombre[10], puedes guardar como máximo 9 letras más el '\0'. Olvidar ese espacio extra es fuente de una cantidad epica de errores.

Por suerte, C te da una forma más cómoda de inicializar cadenas:

```
char saludo[] = "Hola";
```

Cuando usas comillas DOBLES, el compilador crea el arreglo con el tamaño exacto y le agrega el '\0' automáticamente. "Hola" entre comillas dobles ocupa 5 bytes: H, o, l, a, '\0'. Nota la diferencia crucial: 'A' con comillas SIMPLES es un solo carácter; "A" con comillas DOBLES es una cadena de dos bytes ('A' y '\0'). Confundirlas es un error típico.

Cada elemento de la cadena es un char normal, así que puedes manipularlo letra por letra:

```
char palabra[] = "Hola";  
palabra[0] = 'h'; // ahora dice "hola"  
printf("%c\n", palabra[2]); // imprime 'l'
```

Y con %s en printf imprimes la cadena completa: printf imprime carácter por carácter hasta encontrar el '\0':

```
printf("Saludo: %s\n", saludo);
```

Fíjate que con %s NO pones corchetes: le pasas el nombre del arreglo y printf se encarga de recorrerlo hasta el terminador. Si por algún accidente tu cadena no tiene '\0', printf seguirá leyendo memoria más allá del arreglo, imprimiendo basura hasta que por suerte encuentre un

cero. Es otro sintoma clásico del terminador olvidado: en pantalla aparece tu texto seguido de símbolos raros.

Leer texto del teclado: por que fgets y no scanf

Para leer un número usas scanf con %d. Para leer una PALABRA también puedes usar scanf con %s:

```
char nombre[30];
scanf("%s", nombre); // lee una palabra (sin espacios)
```

Pero %s tiene dos problemas graves. Primero: se detiene en el primer espacio. Si el usuario escribe "Juan Perez", scanf solo guarda "Juan" y deja "Perez" ahí atorado en el buffer de entrada, lo que puede arruinar tus siguientes lecturas. Segundo y más grave: si el usuario escribe más caracteres de los que caben en el arreglo, scanf los escribe de todos modos, pasandose del tamaño y corrompiendo memoria. Es el desbordamiento de buffer del que hablaremos al final.

La forma segura y profesional de leer una línea completa de texto (con espacios incluidos) es fgets:

```
char nombre[50];
printf("Escribe tu nombre completo: ");
fgets(nombre, 50, stdin);
```

fgets recibe tres cosas: el arreglo donde guardar, el tamaño MÁXIMO (incluyendo el '\0'), y stdin que significa "el teclado". Lee todo lo que el usuario escriba hasta que presione Enter, pero NUNCA escribe más de tamaño-1 caracteres: si el usuario escribe de más, corta ahí. Eso elimina el desbordamiento. Además guarda los espacios: "Juan Perez" se guarda completo.

fgets tiene una peculiaridad: guarda también el Enter (el salto de línea '\n') al final del texto. Si el usuario escribe "Ana" y presiona Enter, el arreglo contiene 'A', 'n', 'a', '\n', '\0'. Ese '\n' extra suele estorbar (por ejemplo, al comparar cadenas o al imprimir con formato). La forma estándar de quitarlo es:

```
nombre[strcspn(nombre, "\n")] = '\0';
```

Que hace esto? `strcspn` busca la posición del primer `'\n'` en la cadena y ahí pone el `'\0'`, cortando el salto de línea. Es una línea que vas a copiar tal cual muchas veces; con el tiempo entenderás cada parte. Por ahora, usala como receta: después de cada `fgets`, quita el salto de línea así. Necesitas `#include <string.h>` para usar `strcspn`.

Un detalle práctico: no mezcles `scanf` y `fgets` sin cuidado. Si antes hiciste un `scanf("%d")` y luego un `fgets`, el `fgets` puede leer el Enter que quedó pendiente del `scanf` y parecer que "se salta" la lectura. La solución más simple para principiantes: después de un `scanf`, limpia el buffer consumiendo caracteres hasta el salto de línea, o usa `fgets` para todo y convierte con funciones como `atoi`. No te preocupes si esto te confunde ahora: es uno de esos detalles que se dominan con práctica.

string.h: la caja de herramientas para texto

Como las cadenas son arreglos, C no te da operadores mágicos para trabajar con ellas: no puedes sumar dos cadenas con `+`, ni compararlas con `==`, ni asignarlas con `=`. Para eso existe la biblioteca `string.h`, con funciones que hacen el trabajo pesado. Tienes que incluirla con `#include <string.h>`. Vamos una por una, porque cada una tiene su truco.

`strlen`: mide la longitud de la cadena, SIN contar el `'\0'`.

```
char texto[] = "Hola";  
int n = strlen(texto);    // n vale 4
```

`strlen` cuenta caracteres hasta encontrar el terminador. Ojo: `strlen` te dice cuánto mide el TEXTO, no cuánto mide el ARREGLO. Si declaras `char buf[100]` y guardas "Hola", `strlen` dice 4 pero el arreglo mide 100. Esa diferencia (texto vs contenedor) es clave para no desbordar nada.

`strcpy`: copia una cadena en otra. Como no puedes hacer destino = origen con arreglos, usas:

```
char origen[] = "Hola mundo";  
char destino[20];  
strcpy(destino, origen);    // destino ahora contiene "Hola mundo"
```

El orden es: primero el DESTINO, luego el ORIGEN. Un truco para recordarlo: es como una asignación destino = origen, el destino va a la izquierda. `strcpy` copia carácter por carácter incluyendo el `'\0'`. El peligro: si el origen no cabe en el destino, `strcpy` lo copia de todos modos y desborda. Por eso el destino SIEMPRE debe ser más grande que lo que vas a copiar. Existe una versión más segura, `strncpy`, que limita la cantidad de caracteres, pero tiene sus propios

detalles (no siempre pone el '\0'), así que muchos programadores prefieren asegurarse del tamaño y usar strcpy con cuidado.

strcat: concatena, o sea, pega una cadena al final de otra.

```
char saludo[30] = "Hola, ";  
strcat(saludo, "mundo"); // saludo ahora es "Hola, mundo"
```

strcat busca el '\0' del destino y empieza a pegar ahí, y pone un nuevo '\0' al final. El destino debe tener espacio suficiente para ambas cadenas más el terminador: "Hola, " (6) + "mundo" (5) + '\0' (1) = 12, y el arreglo es de 30, así que cabe sobrado. Si no cabe, desbordamiento otra vez. El patrón seguro: declara el destino con holgura generosa.

strcmp: compara dos cadenas y te dice si son iguales y, si no, cuál va primero en orden alfabético. NO uses == para comparar cadenas: == compara las DIRECCIONES de los arreglos, no su contenido, y dos arreglos distintos siempre tienen direcciones distintas aunque digan lo mismo.

```
int r = strcmp("manzana", "manzana"); // r vale 0: son iguales  
int r2 = strcmp("abc", "abd"); // r2 es negativo: "abc" va antes  
int r3 = strcmp("abd", "abc"); // r3 es positivo: "abd" va despues
```

La regla: strcmp devuelve 0 si son iguales, un número negativo si la primera es "menor" (va antes en el diccionario), y un número positivo si es "mayor". Para verificar igualdad, comparas con cero:

```
if (strcmp(password, "secreto123") == 0) {  
    printf("Acceso permitido\n");  
}
```

Ese == 0 confunde al principio ("si son iguales, por qué comparo con cero?"), pero tiene lógica: strcmp devuelve la DIFERENCIA, y diferencia cero significa iguales. Con la práctica se vuelve natural.

Hay más funciones útiles en string.h: strchr busca un carácter dentro de una cadena, strstr busca una subcadena dentro de otra, y memset llena un bloque de memoria con un valor (muy usado para limpiar arreglos). No necesitas memorizarlas todas ahora; con strlen, strcpy, strcat y strcmp cubres el 90 por ciento de lo que harás con texto en tus primeros programas.

El desbordamiento de buffer: el error más famoso de C

Un buffer es simplemente un arreglo donde guardas datos temporalmente. El desbordamiento de buffer (buffer overflow) pasa cuando escribes MÁS datos de los que caben en el arreglo. C no verifica los límites: si tu arreglo es de 10 y escribes 15 caracteres, C escribe los 15, pisando la memoria que sigue. Esa memoria pisada puede contener otras variables de tu programa, o direcciones de retorno de funciones. El resultado va desde un programa que falla de forma misteriosa hasta una vulnerabilidad de seguridad que un atacante puede explotar para tomar control de la computadora. No es exageración: algunos de los ataques informáticos más famosos de la historia usaron desbordamientos de buffer en programas escritos en C.

Mira lo fácil que es provocarlo:

```
char nombre[8];
strcpy(nombre, "Alejandro"); // "Alejandro" tiene 9 letras + '\0' = 10 bytes en un
arreglo de 8
```

Eso escribe 2 bytes de más. Tal vez no pase nada visible hoy, y tal vez mañana tu programa falle en un lugar completamente distinto, porque la corrupción de memoria es traicionera: el error aparece lejos de donde se origina.

Como te proteges? Con disciplina, porque el compilador no lo hará por ti:

1. Dimensiona con holgura: si esperas nombres de hasta 20 letras, declara [50] o [100]. La memoria es barata; los errores son caros.
2. Usa fgets en lugar de scanf("%s") para leer texto, porque fgets respeta el tamaño máximo.
3. Antes de strcpy o strcat, asegurate de que el destino tiene espacio. Puedes verificar con strlen: si $\text{strlen}(\text{origen}) + 1 \leq \text{tamano_destino}$, cabe.
4. En ciclos, verifica siempre que el índice este en rango: $i \geq 0 \ \&\& \ i < N$.
5. Compila con advertencias activadas: gcc -Wall -Wextra te avisa de varios problemas. (En la clase 29 verás depuración a fondo.)

Graba esto: en C, TU eres el responsable de los límites. El lenguaje te da velocidad y control total, y a cambio te pide disciplina total. Es el trato.

Errores comunes (y como detectarlos)

Índice fuera de rango: pedir `a[10]` en un arreglo de 10 (índices válidos 0 a 9), o usar `<=` en el for en lugar de `<`. Síntoma: el programa funciona "casi bien" pero a veces falla o da valores locos. Detección: revisa que tus ciclos usen $i < N$ y que ningún índice pueda llegar a N.

Olvidar el '\0': construir una cadena carácter por carácter y no poner el terminador al final. Sintoma: `printf("%s")` imprime tu texto seguido de basura. Detección: después de armar la cadena a mano, pon `cadena[posición] = '\0'` siempre.

Comparar cadenas con `==`: `if (nombre == "Juan")` nunca hace lo que esperas, porque compara direcciones. Sintoma: la comparación siempre da falso aunque los textos sean iguales. Solución: usa `strcmp(nombre, "Juan") == 0`.

Confundir 'A' con "A": comillas simples es un char, comillas dobles es una cadena (con '\0' escondido). Sintoma: warnings del compilador o comportamiento raro al asignar. Solución: `char c = 'A'; char s[] = "A";` cada cosa en su lugar.

Usar `scanf("%s")` para frases con espacios: solo lee hasta el primer espacio. Sintoma: "se come" parte del nombre. Solución: `fgets` para líneas completas.

No dejar espacio para el '\0': `char código[5] = "hola!";` necesita 6 bytes (5 letras + terminador). Sintoma: el compilador te avisa (warning) o la cadena queda sin terminador. Solución: cuenta siempre +1, o deja que el compilador deduzca con `[]`.

Mezclar `strlen` con el tamaño del arreglo: usar `strlen` para dimensionar un ciclo sobre el CONTENEDOR en lugar del contenido, o viceversa. Sintoma: ciclos que no recorren todo o que leen de más. Solución: ten claro cuál de los dos necesitas en cada caso.

Buenas prácticas con arreglos y cadenas

Nombra los tamaños con `#define` o `const`: en lugar de repetir el 40 por todo el código, escribe `#define MAX_ALUMNOS 40` y usa `MAX_ALUMNOS` en la declaración y en los ciclos. Si mañana cambia el tamaño, lo cambias en un solo lugar. (Viste `#define` en la clase 5; aquí brilla de verdad.)

Inicializa siempre: `int datos[100] = {0};` para números, y para cadenas asegurate de que empiezan con un '\0' (`char texto[50] = "";` deja la cadena vacía pero válida). Un arreglo inicializado es un arreglo predecible.

Pasa el tamaño junto con el arreglo a las funciones: como verás en la clase 24, dentro de una función el arreglo "pierde" su tamaño, así que la función debe recibirlo como parámetro: `void imprimir(int arr[], int n)`. Acostumbrate desde ya.

Verifica límites en funciones que reciben índices: si una función recibe una posición, comprueba que este entre 0 y n-1 antes de usarla. Es barato y evita desastres.

Prefiere `fgets` sobre `scanf` para texto, quita el '\n' con la receta de `strcspn`, y dimensiona los buffers con holgura. Tu yo del futuro te lo va a agradecer.

Comenta la intención de tus índices: en matrices, usa nombres como fila y col en lugar de i y j cuando el código se complique. `for (fila = 0; fila < FILAS; fila++)` se lee solo.

Y práctica dibujando: cuando trabajes con matrices o con cadenas, dibuja en papel los casilleros con sus índices y ve llenandolos a mano. Los programadores profesionales hacen esto para depurar; no es cosa de principiantes, es cosa de inteligentes.

Arreglos de cadenas: listas de palabras

Si una cadena es un arreglo de chars, una LISTA de cadenas es un arreglo de dos dimensiones de chars: cada fila es una cadena completa.

```
char alumnos[5][30];
```

Leelo así: 5 cadenas, cada una con espacio para 29 letras más el '\0'. `alumnos[0]` es la primera cadena ("Ana"), `alumnos[1]` la segunda, y así. Para leer una cadena específica usas solo el primer índice:

```
strcpy(alumnos[0], "Ana");
strcpy(alumnos[1], "Beto");
printf("Alumno 2: %s\n", alumnos[1]);
```

Y para acceder a UNA LETRA de una cadena específica usas los dos índices: `alumnos[1][0]` es la 'B' de "Beto". Piensa en el archivero: el primer índice elige el cajón (que palabra), el segundo elige el compartimento dentro del cajón (que letra).

Llenar la lista con un ciclo es natural:

```
int i;
for (i = 0; i < 5; i++) {
    printf("Nombre del alumno %d: ", i + 1);
    fgets(alumnos[i], 30, stdin);
    alumnos[i][strcspn(alumnos[i], "\n")] = '\0';
}
```

Fíjate: `fgets` recibe `alumnos[i]`, que es la fila i completa (una cadena de 30 chars). El tamaño que le pasas a `fgets` es 30, el de cada cadena individual, no el del arreglo completo. Este es el error más común con arreglos de cadenas: pasar el tamaño equivocado. Cada fila es su propio buffer con su propio límite.

Los arreglos de cadenas aparecen en todas partes: listas de nombres, menus de opciones, días de la semana, mensajes de error. Y son el puente natural hacia los structs de la clase 25, donde podrás guardar juntos el nombre, la edad y el teléfono de cada persona en lugar de usar arreglos paralelos.

Arreglos como parámetros de funciones: un adelanto

En la clase 24 verás esto a fondo, pero necesitas la idea desde ya porque el ejercicio 7 la usa: cuando pasas un arreglo a una función, no viaja el arreglo completo, viaja solo su dirección. Eso tiene dos consecuencias prácticas que debes conocer ahora.

Primera: dentro de la función, el truco de sizeof para calcular el número de elementos YA NO FUNCIONA. Por eso la función debe recibir el tamaño como parámetro extra:

```
void imprimir(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int datos[] = {10, 20, 30, 40, 50};
    int n = sizeof(datos) / sizeof(datos[0]); // aqui SI funciona
    imprimir(datos, n); // el arreglo y su tamano viajan juntos
    return 0;
}
```

Segunda: como la función recibe la dirección del arreglo original (no una copia), si la función modifica los elementos, los cambios SE QUEDAN. El arreglo no se copia al llamar a la función: la función trabaja directamente sobre tus casilleros. Esto es eficiente (no se desperdicia memoria copiando cientos de datos) pero exige cuidado: una función descuidada puede modificar tu arreglo sin que te des cuenta. Cuando quieras protegerlo, la buena práctica es documentar en un comentario si la función modifica el arreglo o solo lo lee.

Para matrices, la función debe conocer el número de columnas:

```
void mostrarMatriz(int m[][4], int filas) {
    // ...
}
```

El número de filas puede variar (por eso se pasa como parámetro), pero las columnas deben estar fijas en la declaración. La razón técnica completa la entenderás con los apuntadores en la Unidad 8; por ahora, tomalo como regla.

gets: la función prohibida

Si buscas en tutoriales viejos o en libros antiguos, vas a encontrar la función `gets` para leer cadenas:

```
char nombre[20];  
gets(nombre);    // NUNCA HAGAS ESTO
```

`gets` lee una línea completa pero NO recibe el tamaño del arreglo: si el usuario escribe más de lo que cabe, desborda el buffer sin piedad. Es tan peligrosa que el estándar de C la ELIMINÓ por completo: los compiladores modernos se niegan a compilarla o lanzan advertencias graves. Si la ves en código ajeno, reemplazala por `fgets`. Conocer `gets` solo sirve para una cosa: reconocerla y no usarla jamás. Es el ejemplo perfecto de por qué en C la seguridad está en tus manos.

Ejemplos paso a paso

Todos estos ejemplos están listos para copiar, compilar con `gcc` y correr. Escríbelos tú mismo en lugar de copiar y pegar: la práctica de escribir es parte del aprendizaje.

Ejemplo 1: Calificaciones y promedio

El ejemplo clásico: leer las calificaciones de un grupo y calcular el promedio. Fíjate en el esqueleto leer-procesar-mostrar con tres ciclos.

```

#include <stdio.h>
#define ALUMNOS 5

int main() {
    float califs[ALUMNOS];
    float suma = 0;
    int i;

    printf("Dame las %d calificaciones:\n", ALUMNOS);
    for (i = 0; i < ALUMNOS; i++) {
        printf("Calificacion %d: ", i + 1);
        scanf("%f", &califs[i]);
    }

    for (i = 0; i < ALUMNOS; i++) {
        suma = suma + califs[i];
    }

    printf("\nPromedio del grupo: %.2f\n", suma / ALUMNOS);

    printf("Calificaciones arriba del promedio:\n");
    for (i = 0; i < ALUMNOS; i++) {
        if (califs[i] > suma / ALUMNOS) {
            printf("  Alumno %d: %.1f\n", i + 1, califs[i]);
        }
    }

    return 0;
}

```

Por que tres ciclos y no uno solo? Porque primero necesitas TODAS las calificaciones para calcular el promedio, y solo después puedes comparar cada una contra el promedio. Este patrón (primero junto los datos, luego los proceso) es super común. Nota también el #define ALUMNOS 5: si quieres 40 alumnos, cambias un número y todo el programa se ajusta.

Ejemplo 2: Buscar el máximo y el mínimo

Recorres el arreglo una vez llevando la cuenta del valor más grande y el más chico vistos hasta el momento. La clave es inicializar máximo y mínimo con el PRIMER elemento, no con cero.

```

#include <stdio.h>

int main() {
    int nums[8] = {34, 12, 89, 5, 67, 23, 91, 45};
    int maximo = nums[0];
    int minimo = nums[0];
    int i;

    for (i = 1; i < 8; i++) {
        if (nums[i] > maximo) {
            maximo = nums[i];
        }
        if (nums[i] < minimo) {
            minimo = nums[i];
        }
    }

    printf("Maximo: %d\n", maximo);
    printf("Minimo: %d\n", minimo);

    return 0;
}

```

Por qué empezar con `nums[0]` y el ciclo en `i = 1`? Si inicializaras `máximo = 0` y todos los números fueran negativos, el "máximo" quedaría en 0, que ni siquiera está en el arreglo. Empezando con el primer elemento, el algoritmo funciona con cualquier conjunto de números. El ciclo empieza en 1 porque el elemento 0 ya se tomó en cuenta. Este algoritmo es la base de muchísimos otros: buscar, filtrar, contar.

Ejemplo 3: Matriz y suma de sus elementos

Una matriz de 3x4 con ciclos anidados: uno para llenarla, otro para sumar todo y mostrarla como tabla.

```

#include <stdio.h>
#define FILAS 3
#define COLS 4

int main() {
    int m[FILAS][COLS];
    int suma = 0;
    int fila, col;

    printf("Llena la matriz de %dx%d:\n", FILAS, COLS);
    for (fila = 0; fila < FILAS; fila++) {
        for (col = 0; col < COLS; col++) {
            printf("Elemento [%d][%d]: ", fila, col);
            scanf("%d", &m[fila][col]);
        }
    }

    printf("\nTu matriz:\n");
    for (fila = 0; fila < FILAS; fila++) {
        for (col = 0; col < COLS; col++) {
            printf("%4d", m[fila][col]);
            suma = suma + m[fila][col];
        }
        printf("\n");
    }

    printf("\nSuma de todos los elementos: %d\n", suma);

    return 0;
}

```

Observa el %4d: imprime cada número ocupando 4 espacios, así las columnas quedan alineadas y la matriz se ve como tabla de verdad. Y fíjate como la suma se acumula dentro del mismo ciclo que muestra: cuando el recorrido ya pasa por cada elemento, aprovecha para hacer todo lo que necesites en esa pasada.

Ejemplo 4: Tabla de multiplicar guardada en una matriz

En lugar de imprimir la tabla de multiplicar directamente, la guardamos en una matriz de 10x10 y luego la mostramos. Así ves que una matriz puede ser una "memoria" de resultados calculados.

```

#include <stdio.h>

int main() {
    int tabla[10][10];
    int fila, col;

    // Llenar: la celda [f][c] guarda (f+1) * (c+1)
    for (fila = 0; fila < 10; fila++) {
        for (col = 0; col < 10; col++) {
            tabla[fila][col] = (fila + 1) * (col + 1);
        }
    }

    // Mostrar como tabla bonita
    printf("Tabla de multiplicar:\n");
    for (fila = 0; fila < 10; fila++) {
        for (col = 0; col < 10; col++) {
            printf("%4d", tabla[fila][col]);
        }
        printf("\n");
    }

    // Consultar un valor específico
    printf("\n7 x 8 = %d\n", tabla[6][7]);

    return 0;
}

```

El truco: los índices van de 0 a 9 pero las tablas de multiplicar van de 1 a 10, por eso usamos $(fila + 1) * (col + 1)$. Y 7×8 está en `tabla[6][7]`: fila 6 = número 7, columna 7 = número 8. Este desfase entre "número humano" e "índice de máquina" es algo con lo que vas a convivir siempre; acostúmbrate a sumar o restar 1 con naturalidad.

Ejemplo 5: Contar caracteres de una cadena

Leemos una frase con `fgets` y contamos cuántos caracteres tiene de dos formas: con `strlen` y a mano, recorriendo hasta el `'\0'`. Así ves con tus propios ojos que `strlen` no es magia.

```

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

int main() {
    char frase[100];
    int i, contador = 0;

    printf("Escribe una frase: ");
    fgets(frase, 100, stdin);
    frase[strcspn(frase, "\n")] = '\0'; // quitar el Enter

    printf("Con strlen: %lu caracteres\n", strlen(frase));

    // A mano: contar hasta encontrar el '\0'
    for (i = 0; frase[i] != '\0'; i++) {
        contador++;
    }
    printf("A mano: %d caracteres\n", contador);

    return 0;
}

```

El ciclo `for (i = 0; frase[i] != '\0'; i++)` es EL patrón para recorrer cadenas: avanzas mientras no llegues al terminador. Aprendetelo igual que el `for` de arreglos. Nota el `%lu` para `strlen`: `strlen` devuelve un tipo llamado `size_t`, que se imprime con `%lu`. Si usas `%d`, en la mayoría de los casos funciona, pero `%lu` es lo correcto y evita warnings.

Ejemplo 6: Invertir una cadena

Para invertir "Hola" y obtener "aloH", usamos dos índices: uno que avanza desde el inicio y otro que retrocede desde el final, intercambiando caracteres hasta que se cruzan. Es el algoritmo clásico de intercambio con variable temporal.


```

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

int main() {
    char texto[100];
    int inicio, fin;
    char temp;

    printf("Escribe una palabra: ");
    fgets(texto, 100, stdin);
    texto[strcspn(texto, "\n")] = '\0';

    inicio = 0;
    fin = strlen(texto) - 1; // indice del ultimo caracter (no el '\0')

    while (inicio < fin) {
        temp = texto[inicio];
        texto[inicio] = texto[fin];
        texto[fin] = temp;
        inicio++;
        fin--;
    }

    printf("Invertida: %s\n", texto);

    return 0;
}

```

fin empieza en `strlen - 1` porque el último carácter real esta justo antes del `'\0'`. El intercambio necesita la variable `temp`: si hicieras `texto[inicio] = texto[fin]` y luego `texto[fin] = texto[inicio]`, el segundo ya estaria pisado y perderias la letra. Es como intercambiar el contenido de dos vasos: necesitas un tercer vaso. El ciclo se detiene cuando `inicio >= fin`, o sea, cuando los índices se cruzan en el centro. Este algoritmo de "dos índices que se acercan" aparece en muchos problemas: palindromos, ordenamientos, busquedas.

Ejemplo 7: Comparar passwords con `strcmp`

Un mini sistema de acceso: el usuario escribe una contrasena y la comparamos con la guardada usando `strcmp`. Aquí ves por que `==` no sirve para cadenas.

```

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

int main() {
    char correcta[] = "cLang2026";
    char intento[50];
    int intentos = 3;

    while (intentos > 0) {
        printf("Password (%d intentos restantes): ", intentos);
        fgets(intento, 50, stdin);
        intento[strcspn(intento, "\n")] = '\0';

        if (strcmp(intento, correcta) == 0) {
            printf("Acceso permitido. Bienvenido!\n");
            return 0;
        } else {
            printf("Password incorrecto.\n");
            intentos--;
        }
    }

    printf("Demasiados intentos. Acceso bloqueado.\n");
    return 1;
}

```

Tres detalles a notar. Primero: `strcmp(intento, correcta) == 0` significa "son iguales". Segundo: quitamos el `\n` del `fgets` ANTES de comparar, porque si no, "cLang2026\n" nunca sería igual a "cLang2026" y el acceso fallaría siempre: un error sutil y muy común. Tercero: `return 1` al final indica que el programa terminó con error, una convención que los sistemas operativos entienden.

Ejemplo 8: Agenda simple con dos arreglos paralelos

Guardamos hasta 10 contactos usando dos arreglos: uno para nombres y otro para telefonos. La posición `i` del nombre corresponde a la posición `i` del teléfono: son "arreglos paralelos". Es una técnica sencilla que anticipa la idea de los structs de la clase 25.

```

#include <stdio.h>
#include <string.h>
#define MAX 10

int main() {

```

```

char nombres[MAX][50];
char telefonos[MAX][20];
int total = 0;
int opcion, i;

do {
    printf("\n--- AGENDA ---\n");
    printf("1. Agregar contacto\n");
    printf("2. Ver contactos\n");
    printf("3. Buscar por nombre\n");
    printf("0. Salir\n");
    printf("Opcion: ");
    scanf("%d", &opcion);
    getchar(); // consumir el Enter que deja scanf

    if (opcion == 1) {
        if (total < MAX) {
            printf("Nombre: ");
            fgets(nombres[total], 50, stdin);
            nombres[total][strcspn(nombres[total], "\n")] = '\0';
            printf("Telefono: ");
            fgets(telefonos[total], 20, stdin);
            telefonos[total][strcspn(telefonos[total], "\n")] = '\0';
            total++;
            printf("Contacto guardado.\n");
        } else {
            printf("Agenda llena!\n");
        }
    } else if (opcion == 2) {
        printf("\nContactos (%d):\n", total);
        for (i = 0; i < total; i++) {
            printf("%d. %s - %s\n", i + 1, nombres[i], telefonos[i]);
        }
    } else if (opcion == 3) {
        char buscar[50];
        int encontrado = 0;
        printf("Nombre a buscar: ");
        fgets(buscar, 50, stdin);
        buscar[strcspn(buscar, "\n")] = '\0';
        for (i = 0; i < total; i++) {
            if (strcmp(nombres[i], buscar) == 0) {
                printf("Encontrado: %s - %s\n", nombres[i], telefonos[i]);
                encontrado = 1;
            }
        }
    }
}

```

```

        if (!encontrado) {
            printf("No se encontro ese contacto.\n");
        }
    }
} while (opcion != 0);

return 0;
}

```

Este ejemplo junta casi todo lo de la unidad: arreglo bidimensional de chars (nombres es un arreglo de 10 cadenas de 50), fgets con su limpieza, strcmp para buscar, ciclos para mostrar, y validación de límites (total < MAX para no desbordar la agenda). El getchar() después del scanf es el truco para consumir el Enter pendiente y que el siguiente fgets no lea una línea vacía. La variable total lleva la cuenta de cuántos contactos hay de verdad, porque el arreglo tiene espacio para 10 pero puede no estar lleno: otra vez la diferencia entre el contenedor y el contenido.

Ejemplo 9: Ordenamiento burbuja

Ordenar un arreglo de menor a mayor con el método de la burbuja: pasas por el arreglo comparando parejas vecinas e intercambiando las que esten en desorden. Con cada pasada, el número más grande "flota" hasta el final, como una burbuja.

```

#include <stdio.h>
#define N 8

int main() {
    int nums[N] = {34, 12, 89, 5, 67, 23, 91, 45};
    int i, j, temp;

    printf("Original: ");
    for (i = 0; i < N; i++) {
        printf("%d ", nums[i]);
    }
    printf("\n");

    // Burbuja: N-1 pasadas
    for (i = 0; i < N - 1; i++) {
        for (j = 0; j < N - 1 - i; j++) {
            if (nums[j] > nums[j + 1]) {
                temp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = temp;
            }
        }
    }

    printf("Ordenado: ");
    for (i = 0; i < N; i++) {
        printf("%d ", nums[i]);
    }
    printf("\n");

    return 0;
}

```

Por que $N - 1 - i$ en el ciclo interno? Después de la primera pasada, el mayor ya quedó al final y no hace falta volver a compararlo; después de la segunda, los dos mayores están en su lugar, y así. Cada pasada necesita comparar uno menos. Es una optimización pequeña pero muestra algo importante: pensar en que parte del arreglo YA esta resuelta te ahorra trabajo. La burbuja no es el método más rápido (hay mejores, como quicksort), pero es el más didactico: con el entiendes la mecánica de comparar e intercambiar que usan todos los ordenamientos. Y nota que modifica el arreglo original: recuerda lo que viste en la sección de arreglos como parámetros, los cambios a un arreglo se quedan.

Práctica

Haz cada ejercicio antes de ver su solución. Si te atorras más de treinta minutos, mira la solución, entiendela y luego escríbela tu solo sin verla. Todos se compilan con gcc.

1. Pide 10 calificaciones, calcula el promedio e indica cuántas son mayores al promedio, cuál fue la más alta y cuál la más baja.
2. Llena un arreglo de 8 enteros e imprimelo al revés, sin modificar el arreglo original. Después haz una segunda versión que si invierta el arreglo original.
3. Llena una matriz de 4x4 y calcula la suma de su diagonal principal (los elementos [0][0], [1][1], [2][2], [3][3]) y la suma de su diagonal secundaria ([0][3], [1][2], [2][1], [3][0]).
4. Pide una frase al usuario y cuenta cuántas vocales tiene (mayúsculas y minúsculas cuentan). No uses nada más que un ciclo y comparaciones.
5. Pide una palabra y determina si es palindromo, es decir, si se lee igual al derecho y al revés (como "oso", "reconocer" o "anilina"). Ignora mayúsculas y minúsculas: "Oso" también cuenta.
6. Pide una frase y cuenta cuántas palabras tiene. Considera que las palabras están separadas por uno o más espacios, y que la frase puede empezar o terminar con espacios.
7. Genera la matriz identidad de 5x5 (unos en la diagonal principal y ceros en el resto) y muéstrala en pantalla. Después pide un número al usuario y multiplica toda la matriz por ese número (matriz escalar) y vuelve a mostrarla.

Solución del ejercicio 1

La idea es la misma del ejemplo de calificaciones: un ciclo para leer y acumular la suma, luego calcular el promedio, y un segundo ciclo para contar cuántas superan el promedio mientras rastreamos máximo y mínimo. El máximo y mínimo se inicializan con el primer elemento.

```

#include <stdio.h>
#define N 10

int main() {
    float califs[N];
    float suma = 0, promedio;
    float mayor, menor;
    int arriba = 0;
    int i;

    for (i = 0; i < N; i++) {
        printf("Calificacion %d: ", i + 1);
        scanf("%f", &califs[i]);
        suma += califs[i];
    }

    promedio = suma / N;
    mayor = califs[0];
    menor = califs[0];

    for (i = 0; i < N; i++) {
        if (califs[i] > promedio) {
            arriba++;
        }
        if (califs[i] > mayor) {
            mayor = califs[i];
        }
        if (califs[i] < menor) {
            menor = califs[i];
        }
    }

    printf("\nPromedio: %.2f\n", promedio);
    printf("Arriba del promedio: %d\n", arriba);
    printf("Mas alta: %.1f\n", mayor);
    printf("Mas baja: %.1f\n", menor);

    return 0;
}

```

Explicación: el primer ciclo hace dos cosas a la vez (leer y sumar) porque ambas necesitan pasar por cada elemento una sola vez. El segundo ciclo también hace tres cosas en una pasada: contar, buscar mayor y buscar menor. Hacer varias cosas en el mismo recorrido es eficiente y es una habilidad que se desarrolla con práctica: cada vez que recorras un arreglo, pregúntate "¿qué más puedo calcular en esta pasada?".

Solución del ejercicio 2

Para imprimir al revés sin modificar nada, basta recorrer el arreglo desde el último índice hasta el primero. Para invertirlo de verdad, usamos el intercambio con dos índices como en el ejemplo de la cadena invertida.


```

#include <stdio.h>
#define N 8

int main() {
    int nums[N] = {5, 12, 3, 27, 9, 41, 6, 18};
    int i, inicio, fin, temp;

    printf("Al reves (sin modificar): ");
    for (i = N - 1; i >= 0; i--) {
        printf("%d ", nums[i]);
    }
    printf("\n");

    // Invertir el arreglo original
    inicio = 0;
    fin = N - 1;
    while (inicio < fin) {
        temp = nums[inicio];
        nums[inicio] = nums[fin];
        nums[fin] = temp;
        inicio++;
        fin--;
    }

    printf("Arreglo invertido: ");
    for (i = 0; i < N; i++) {
        printf("%d ", nums[i]);
    }
    printf("\n");

    return 0;
}

```

Explicación: la primera parte muestra que "recorrer al reves" y "invertir" son cosas distintas: recorrer al reves solo cambia el orden en que MIRAS los datos; invertir cambia el orden en que están GUARDADOS. El ciclo for ($i = N - 1$; $i \geq 0$; $i--$) es el patrón de recorrido inverso: empieza en el último índice válido ($N - 1$, no N) y baja hasta 0 inclusive. La segunda parte reutiliza el algoritmo de intercambio con variable temporal que ya viste en cadenas: funciona igual con enteros porque la idea no depende del tipo de dato.

Solución del ejercicio 3

La diagonal principal cumple que $\text{fila} == \text{columna}$, así que con un solo ciclo basta: $m[i][i]$. La secundaria cumple que $\text{fila} + \text{columna} == N - 1$, o sea, $m[i][N - 1 - i]$.

```
#include <stdio.h>
#define N 4

int main() {
    int m[N][N];
    int sumaP = 0, sumaS = 0;
    int fila, col, i;

    printf("Llena la matriz de %dx%d:\n", N, N);
    for (fila = 0; fila < N; fila++) {
        for (col = 0; col < N; col++) {
            printf("[%d][%d]: ", fila, col);
            scanf("%d", &m[fila][col]);
        }
    }

    for (i = 0; i < N; i++) {
        sumaP += m[i][i];           // diagonal principal
        sumaS += m[i][N - 1 - i]; // diagonal secundaria
    }

    printf("\nMatriz:\n");
    for (fila = 0; fila < N; fila++) {
        for (col = 0; col < N; col++) {
            printf("%4d", m[fila][col]);
        }
        printf("\n");
    }

    printf("\nSuma diagonal principal: %d\n", sumaP);
    printf("Suma diagonal secundaria: %d\n", sumaS);

    return 0;
}
```

Explicación: lo bonito de este ejercicio es que NO necesitas ciclos anidados para las diagonales: cada diagonal tiene exactamente N elementos y cada uno esta determinado por un solo índice i. Para la secundaria, cuando i = 0 la columna es 3, cuando i = 1 la columna es 2: la fórmula $N - 1 - i$ genera esa secuencia. Aprender a expresar posiciones de una matriz con fórmulas de i es una habilidad clave para graficos y juegos.

Solución del ejercicio 4

Recorremos la cadena hasta el '\0' y comparamos cada carácter contra las 10 vocales (5 minúsculas y 5 mayúsculas). Se puede escribir con una condición larga o con una función auxiliar; aquí va la versión directa.

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

int main() {
    char frase[200];
    int i, vocales = 0;
    char c;

    printf("Escribe una frase: ");
    fgets(frase, 200, stdin);
    frase[strcspn(frase, "\n")] = '\0';

    for (i = 0; frase[i] != '\0'; i++) {
        c = frase[i];
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' ||
            c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U') {
            vocales++;
        }
    }

    printf("La frase tiene %d vocales.\n", vocales);

    return 0;
}
```

Explicación: guardamos frase[i] en la variable c para no repetir frase[i] diez veces: el código queda más legible. La condición usa || porque basta con que el carácter sea IGUAL a una de las vocales. Una mejora elegante sería convertir cada carácter a minúscula antes de comparar (con tolower de ctype.h) para comparar solo contra 5 vocales; si quieres practicar más, intenta esa versión. Este ejercicio refuerza el patrón de recorrido de cadenas hasta el '\0'.

Solución del ejercicio 5

Es el mismo algoritmo de dos índices que se acercan, pero en lugar de intercambiar, COMPARAMOS: si alguna pareja no coincide, no es palindromo. Para ignorar mayúsculas, convertimos todo a minúsculas primero (o comparamos con tolerancia).

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

int main() {
    char palabra[100];
    int inicio, fin;
    int esPalindromo = 1; // suponemos que si, hasta demostrar lo contrario

    printf("Escribe una palabra: ");
    fgets(palabra, 100, stdin);
    palabra[strcspn(palabra, "\n")] = '\0';

    // Pasar todo a minusculas
    for (inicio = 0; palabra[inicio] != '\0'; inicio++) {
        palabra[inicio] = tolower(palabra[inicio]);
    }

    inicio = 0;
    fin = strlen(palabra) - 1;

    while (inicio < fin) {
        if (palabra[inicio] != palabra[fin]) {
            esPalindromo = 0;
            break; // ya no tiene caso seguir comparando
        }
        inicio++;
        fin--;
    }

    if (esPalindromo) {
        printf("Si es palindromo.\n");
    } else {
        printf("No es palindromo.\n");
    }

    return 0;
}
```

Explicación: la variable `esPalindromo` es una "bandera": empieza en 1 (verdadero) y solo cambia a 0 si encontramos una pareja distinta. El `break` sale del ciclo en cuanto sabemos la respuesta, sin comparar de más. `tolower` (de `ctype.h`) convierte un carácter a minúscula; aplicarlo a toda la palabra antes de comparar resuelve lo de "Oso". Fíjate como este ejercicio combina tres ideas de la unidad: recorrido de cadenas, el patrón de dos índices, y manipulación carácter por carácter.

Solución del ejercicio 6

Contamos palabras detectando TRANSICiones: una palabra empieza cuando encontramos un carácter que no es espacio y el anterior si era espacio (o es el inicio de la cadena). Así los espacios múltiples no cuentan de más.

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

int main() {
    char frase[200];
    int i, palabras = 0;
    int dentro = 0; // 0 = estamos en espacios, 1 = estamos dentro de una palabra

    printf("Escribe una frase: ");
    fgets(frase, 200, stdin);
    frase[strlen(frase)] = '\0';

    for (i = 0; frase[i] != '\0'; i++) {
        if (frase[i] != ' ') {
            if (dentro == 0) {
                palabras++; // empieza una palabra nueva
                dentro = 1;
            }
        } else {
            dentro = 0; // salimos de la palabra
        }
    }

    printf("La frase tiene %d palabras.\n", palabras);

    return 0;
}
```

Explicación: la variable `dentro` es una máquina de dos estados: "estoy en una palabra" o "estoy en espacios". Solo contamos cuando ENTRAMOS a una palabra, no por cada letra. Por eso

"hola mundo" (con tres espacios) cuenta 2 y no 4, y los espacios al inicio o al final no cuentan nada. Este patrón de "detectar el borde" (el momento en que algo cambia) es muy poderoso: sirve para contar palabras, para procesar sensores, para analizar datos. Si lo entiendes bien, lo usarás toda tu vida.

Solución del ejercicio 7

La matriz identidad tiene 1 donde $\text{fila} == \text{columna}$ y 0 en el resto. La multiplicación escalar multiplica cada elemento por el número dado, con ciclos anidados.

```

#include <stdio.h>
#define N 5

void mostrar(int m[N][N]) {
    int fila, col;
    for (fila = 0; fila < N; fila++) {
        for (col = 0; col < N; col++) {
            printf("%4d", m[fila][col]);
        }
        printf("\n");
    }
}

int main() {
    int m[N][N];
    int fila, col, k;

    for (fila = 0; fila < N; fila++) {
        for (col = 0; col < N; col++) {
            if (fila == col) {
                m[fila][col] = 1;
            } else {
                m[fila][col] = 0;
            }
        }
    }

    printf("Matriz identidad:\n");
    mostrar(m);

    printf("\nNumero para multiplicar: ");
    scanf("%d", &k);

    for (fila = 0; fila < N; fila++) {
        for (col = 0; col < N; col++) {
            m[fila][col] = m[fila][col] * k;
        }
    }

    printf("\nMatriz por %d:\n", k);
    mostrar(m);

    return 0;
}

```

Explicación: este ejercicio introduce algo nuevo: una FUNCIÓN que recibe una matriz (mostrar). Fíjate en la declaración `void mostrar(int m[N][N])`: cuando pasas una matriz a una función, debes indicar el tamaño de las columnas (aquí con el `#define N`). Es un adelanto de la clase 24, donde verás a fondo como viajan los arreglos entre funciones. La identidad se genera con la condición `fila == col`, la misma idea de la diagonal principal del ejercicio 3. La multiplicación escalar es el recorrido anidado más simple: operas cada celda sin importar su posición.

8. Pide 12 números enteros, luego pide un valor a buscar e indica en que posiciones aparece y cuántas veces. Si no aparece, dilo claramente.

Solución del ejercicio 8

Es la búsqueda lineal: recorrer el arreglo comparando cada elemento con el valor buscado. Guardamos las posiciones encontradas y un contador.


```

#include <stdio.h>
#define N 12

int main() {
    int nums[N];
    int buscado, i, veces = 0;

    printf("Dame %d numeros:\n", N);
    for (i = 0; i < N; i++) {
        printf("Numero %d: ", i + 1);
        scanf("%d", &nums[i]);
    }

    printf("\nValor a buscar: ");
    scanf("%d", &buscado);

    printf("El valor %d aparece en las posiciones: ", buscado);
    for (i = 0; i < N; i++) {
        if (nums[i] == buscado) {
            printf("%d ", i);
            veces++;
        }
    }

    if (veces == 0) {
        printf("(ninguna)\nNo se encontro el valor.\n");
    } else {
        printf("\nTotal de apariciones: %d\n", veces);
    }

    return 0;
}

```

Explicación: la búsqueda lineal es el algoritmo de búsqueda más simple: miras cada casillero en orden hasta terminar. Es lenta para arreglos gigantescos (ahí entran técnicas como la búsqueda binaria, que requiere el arreglo ordenado), pero para tamaños normales es perfecta y es la base para entender las demás. Nota que mostramos las posiciones como ÍNDICES (empezando en 0): si tu usuario final espera posiciones "humanas" (empezando en 1), suma 1 al mostrar. Ese detalle de comunicación con el usuario es parte del oficio: la máquina cuenta desde 0, las personas desde 1, y tú decides en que momento traducir.

Resumen de la unidad

Llegaste al final de la Unidad 7. Si la trabajaste en serio (leiste, escribiste los ejemplos, hiciste los ejercicios), ahora manejas una de las herramientas más poderosas de la programación. Repasemos lo esencial.

Un arreglo es una colección de elementos del mismo tipo, juntos en memoria, bajo un solo nombre. Se declara con `tipo nombre[tamaño]`; y se accede con índices que SIEMPRE empiezan en cero: en un arreglo de N elementos, los índices válidos van de 0 a $N-1$. El índice puede ser una variable o una expresión, y esa es la llave que abre todo el poder de los arreglos.

Los arreglos y el ciclo `for` nacieron el uno para el otro. El patrón `for (i = 0; i < N; i++)` recorre cualquier arreglo de principio a fin; con la condición `< N` (nunca `<= N`) te mantienes dentro del rango. Con ese patrón lees datos, los procesas y los muestras: el esqueleto leer-procesar-mostrar resuelve una enorme cantidad de problemas reales.

Inicializa tus arreglos: con llaves al declararlos (`{1, 2, 3}`), con `{0}` para poner todo en ceros, o dejando que el compilador deduzca el tamaño con `[]`. Un arreglo sin inicializar contiene basura. El operador `sizeof` te dice cuántos bytes ocupa algo, y `sizeof(arr) / sizeof(arr[0])` te da el número de elementos, pero solo donde el arreglo se declaró.

Las matrices son arreglos de dos dimensiones: `int m[FILAS][COLS]`; un archivero con cajones y compartimentos. Se recorren con dos ciclos `for` anidados (filas afuera, columnas adentro) y se inicializan con llaves anidadas. La diagonal principal cumple `fila == columna`; la secundaria, `fila + columna == N - 1`.

En C no existe el tipo `string`: las cadenas son arreglos de `char` terminados con el carácter nulo `'\0'`. "Hola" ocupa 5 bytes, no 4: siempre reserva un espacio extra para el terminador. `'A'` (comillas simples) es un carácter; `"A"` (comillas dobles) es una cadena de dos bytes. Para imprimir una cadena completa usa `%s`; para leer una línea con espacios usa `fgets(cadena, tamaño, stdin)` y quita el Enter final con `cadena[strcspn(cadena, "\n")] = '\0'`. Evita `scanf("%s")` para frases: se detiene en el primer espacio y no respeta el tamaño.

La biblioteca `string.h` te da las herramientas para texto: `strlen` mide la longitud sin contar el `'\0'`; `strcpy` copia (destino primero, origen después); `strcat` pega al final; `strcmp` compara y devuelve 0 si son iguales, negativo si la primera va antes, positivo si va después. Nunca compares cadenas con `==`: eso compara direcciones, no contenido.

El desbordamiento de buffer es escribir más datos de los que caben en el arreglo. C no verifica límites: TU eres el responsable. Protegete dimensionando con holgura, usando `fgets`,

verificando tamaños antes de strcpy/strcat, cuidando los índices en los ciclos y compilando con gcc -Wall -Wextra. La corrupción de memoria es traicionera: el error aparece lejos de donde se origino.

Errores que ya sabes detectar: índice fuera de rango (ciclos con `<=` en lugar de `<`), olvidar el `'\0'` (basura después del texto en printf), comparar con `==` (siempre falso), confundir `'A'` con `"A"`, y no dejar espacio para el terminador. Cada uno tiene síntomas claros y soluciones claras.

Buenas prácticas que te van a acompañar siempre: nombra los tamaños con `#define`, inicializa todo, pasa el tamaño junto con el arreglo a las funciones, verifica límites, prefiere `fgets` para texto, y dibuja en papel cuando algo se complique. Los profesionales dibujan; no es de principiantes, es de inteligentes.

Lo que sigue: en la Unidad 8 entran los apuntadores, y vas a descubrir que los arreglos y los apuntadores están íntimamente relacionados: el nombre de un arreglo ES una dirección, y el índice `[i]` esconde aritmética de apuntadores. Todo lo que aprendiste aquí (índices desde cero, contenedores vs contenido, direcciones con `&`) va a hacer clic de forma espectacular. Si esta unidad te costó, repasa los ejemplos antes de seguir: los apuntadores se construyen directamente sobre estos cimientos.

Y recuerda: cada curso de esta serie nace de los videos de lucasegaacademy.com, donde puedes ver estos mismos temas explicados en clips cortos frente al pizarron. El libro y los videos se complementan: el video te muestra el ritmo, el libro te da la profundidad. Nos vemos en la Unidad 8: los apuntadores te están esperando, y esta vez llegas preparado.

UNIDAD 8: Apuntadores

Introducción

Bienvenido a la unidad más famosa, más temida y más poderosa de todo el lenguaje C. Los apuntadores son el tema que separa a quienes apenas usan C de quienes realmente lo dominan. Si has escuchado por ahí que C es difícil, probablemente quien lo dijo estaba pensando en esta unidad.

Te voy a ser honesto desde el inicio: este tema va a exigirte más paciencia que todos los anteriores. También te voy a decir algo que suena a promesa de película pero es verdad: cuando termines esta unidad, vas a entender algo que el noventa por ciento de los programadores que usan otros lenguajes nunca han entendido. Vas a ver por debajo del capot de la computadora. Vas a tocar la memoria con las manos, por decirlo de alguna manera, y vas a entender cómo funcionan por dentro las listas, los arreglos dinámicos, los árboles y hasta como el sistema operativo reparte la memoria entre programas.

Por que asustan tanto los apuntadores? Primero, por el mito. Hay profesores y videos que los presentan como si fueran un monstruo, y esa fama hace que mucha gente llegue a este tema con miedo antes de leer una sola línea. Segundo, porque los errores con apuntadores son de los peores: un apuntador mal usado no siempre te da un error amable en pantalla; a veces el programa se traba, a veces te da un número raro, a veces funciona cinco veces y a la sexta explota. Eso confunde. Tercero, porque los apuntadores exigen pensar en dos niveles al mismo tiempo: el valor y la dirección donde vive el valor. Es como si de repente tuvieras que preocuparte no solo por el contenido de las cajas, sino por el número de cada caja en un almacén gigante.

Pero aquí esta la otra cara, la que nadie te dice con suficiente fuerza: los apuntadores son poderosos porque te dan control total. C fue creado para escribir sistemas operativos, y un sistema operativo necesita decidir exactamente donde va cada byte de memoria. Los apuntadores son la herramienta que hace posible eso. Con apuntadores puedes pedir memoria prestada cuando tu programa está corriendo (memoria dinámica, que verás en la unidad 9), puedes hacer que una función cambie varias variables a la vez, puedes recorrer millones de datos sin copiarlos (imaginate copiar un arreglo de un millón de números solo para imprimirlo: con apuntadores no copias nada, solo pasas la dirección), y puedes construir estructuras de datos que crecen y se encogen a voluntad, como listas enlazadas y árboles.

Piensa en la memoria de tu computadora como un vecindario enorme. Cada casa tiene un número: la casa 1000, la casa 1001, la casa 1002, y así hasta millones. En cada casa vive un valor: un número, una letra, lo que sea. Una variable normal es como tener la llave de una casa y guardar cosas adentro. Un apuntador, en cambio, es como tener un papelito donde anotaste la dirección de la casa. El papelito no es la casa: es la dirección de la casa. Con el papelito puedes ir a la casa, abrir la puerta y cambiar lo que hay adentro. Y lo más importante: puedes pasarle el papelito a otra persona (una función) para que ella también pueda ir a esa casa y modificarla. Sin el papelito, la otra persona solo podría trabajar con una copia de lo que había en la casa, y los cambios se perderían.

Esa analogía de las direcciones postales es la clave de toda esta unidad. Guardala en la cabeza: un apuntador no es un valor, es la dirección donde vive un valor. Cuando la entiendas de verdad, todo lo demás se vuelve consecuencia lógica.

El panorama de la unidad es este. Empezaremos por el operador `&`, que te dice la dirección de una variable. Luego veremos cómo declarar un apuntador (`int *p`), cómo hacer que apunte a una variable y cómo usar el operador `*` para entrar a la casa y leer o cambiar su contenido. Veremos por qué el tipo importa: un apuntador a `int` no es lo mismo que un apuntador a `char`, porque cada tipo ocupa un tamaño distinto de casa. Conoceremos al apuntador `NULL`, que significa "no apunto a ningún lado", y por qué es tu mejor amigo para evitar errores. Después entraremos a la aritmética de apuntadores: `p + 1` no suma uno al número de la casa, sino que salta a la siguiente casa completa del tipo, y entender esto te va a abrir los ojos. Descubriremos la relación íntima entre arreglos y apuntadores: el nombre de un arreglo ES un apuntador a su primer elemento, así que todo lo que sabes de arreglos se conecta con todo lo que aprendas aquí. Aprenderemos a pasar apuntadores a funciones para que una función pueda modificar las variables de quien la llamó, con el ejemplo clásico de intercambiar dos valores (`swap`). Y cerraremos con los apuntadores a apuntadores, que suenan a broma pero son reales y muy usados.

Un consejo antes de empezar, de programador a estudiante: no corras. Esta es la unidad donde más vale la pena escribir cada ejemplo con tus propias manos y ejecutarlo. No copies y pegues: escribe, compila, ejecuta, rompe cosas a propósito y observa que pasa. Cada experimento que hagas aquí te va a dar una intuición que ningún libro puede regalarte. Y si algo no te queda claro a la primera, es normal: este tema se entiende por capas. Hoy entiendes el sesenta por ciento, mañana repasas y entiendes el ochenta, y en una semana, de repente, todo encaja.

Como siempre, en lucasegaacademy.com tienes esta unidad completa con ejercicios interactivos y el tutor Profesor Buho para preguntarle tus dudas a cualquier hora. Vamos a ello: es hora de conocer al tema más famoso de C.

Explicación detallada

Que es una dirección de memoria: el operador &

Vamos a empezar por lo más básico y lo más concreto. Toda variable que declaras en C vive en algun lugar de la memoria de la computadora, y ese lugar tiene una dirección, un número. Es el número de la casa en la analogia del vecindario.

El operador & (ampersand) te dice la dirección de una variable. Se lee "dirección de". Mira este programa minusculo:

```
#include <stdio.h>

int main() {
    int edad = 25;
    printf("El valor de edad es: %d\n", edad);
    printf("La direccion de edad es: %p\n", (void*)&edad);
    return 0;
}
```

Si lo compilas y lo ejecutas, vas a ver algo como:

```
El valor de edad es: 25
La direccion de edad es: 0x7ffe9a2b4c1c
```

Ese número raro en hexadecimal (empieza con 0x) es la dirección. No te preocupes por entender el hexadecimal a fondo: lo único que importa es que es un número que identifica el lugar donde vive la variable. Cada vez que ejecutes el programa, la dirección probablemente será distinta, porque el sistema operativo coloca tu programa donde le conviene. Eso es normal.

Fíjate en dos detalles del printf. Primero, para imprimir una dirección se usa el especificador %p. Segundo, se hace un cast a (void*) porque %p espera un apuntador a void, que es el tipo generico de dirección. En esta unidad verás mucho ese cast; no te asuste, es solo para que el compilador este contento.

Haz este experimento mental: declara tres variables, int a, int b, int c, e imprime sus direcciones. Vas a notar que las direcciones son consecutivas o casi consecutivas, separadas por 4 bytes (porque un int ocupa 4 bytes en la mayoría de las computadoras). Eso te confirma la imagen del vecindario: las casas están una junto a otra, numeradas en orden.

Un punto importante: & solo funciona con algo que tenga una dirección real en memoria, es decir, con una variable (o con un elemento de arreglo, o con un campo de struct, como verás después). No puedes pedir la dirección de un número literal: &5 no tiene sentido, porque el 5 no es una casa, es solo un valor suelto. Tampoco puedes pedir la dirección del resultado de una expresión como &(a + b): el resultado de a + b existe un instante en el procesador y luego desaparece, no tiene casa propia.

Que es un apuntador: declaración con *

Ahora que sabes pedir direcciones, viene la pregunta natural: donde guardo una dirección? No la puedes guardar en un int normal, o al menos no deberías, porque una dirección no es un número cualquiera: es una dirección. Para guardarla existe un tipo de variable especial: el apuntador.

Un apuntador es una variable que guarda una dirección de memoria. Declararlo se hace con un asterisco:

```
int *p;
```

Esa línea dice: "p es un apuntador a int". Léelo así, de derecha a izquierda: p es un apuntador, y apunta a un int. El asterisco en la declaración no significa multiplicación ni significa "el valor de": significa "apuntador a". Es la etiqueta que dice que esta variable guarda direcciones, no números normales.

Ojo con un detalle que confunde a todo el mundo al inicio: el asterisco va pegado al nombre de la variable, no al tipo. Estas tres formas son equivalentes:

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

Pero la primera es la más común y la que te recomiendo. Y hay una trampa clásica: si declaras varias variables en una línea, el asterisco solo aplica a la que lo tiene pegado:

```
int *p, q;
```

Aquí p es un apuntador a int, pero q es un int normal. Por eso muchos programadores prefieren declarar un apuntador por línea. Es más claro y evita errores tontos.

Como hago que un apuntador apunte a una variable? Con el operador & que acabas de conocer:

```
int edad = 25;
int *p;
p = &edad;
```

Después de esas líneas, p guarda la dirección de edad. Dibujemos la memoria en texto para que lo veas:

Dirección	Contenido	Variable
0x7ffe9a2b4c1c	25	edad
0x7ffe9a2b4c10	0x7ffe9a2b4c1c	p

La variable edad vive en la casa 0x7ffe9a2b4c1c y adentro hay un 25. La variable p vive en otra casa (0x7ffe9a2b4c10) y adentro hay... la dirección de edad. El papelito con la dirección anotada. Eso es un apuntador: una variable cuyo contenido es la dirección de otra variable.

El operador * para desreferenciar: entrar a la casa

Ya tienes el papelito con la dirección. Ahora, como entras a la casa? Con el operador * usado sobre un apuntador. Cuando el asterisco aparece en una expresión (no en una declaración), significa "ve a la dirección que guarda este apuntador y dame lo que hay ahí". A eso se le llama desreferenciar.

```
int edad = 25;
int *p = &edad;

printf("%d\n", *p);
```

Que imprime? 25. Porque *p significa: ve a la dirección guardada en p (qué es la dirección de edad) y trae lo que hay adentro (25). Fíjate en la diferencia brutal:

- p es la dirección (el papelito).
- *p es el valor que vive en esa dirección (el contenido de la casa).

Y lo más poderoso: *p no solo sirve para leer, también sirve para escribir. Puedes usar *p como si fuera la variable misma:


```
*p = 30;
printf("%d\n", edad);
```

Esto imprime 30. Cambiaste el valor de edad sin tocar la variable edad directamente: fuiste a su casa usando el papelito y cambiaste lo de adentro. Esta es la idea central de los apuntadores como parámetros de funciones, que veremos más adelante: le pasas a una función el papelito, y la función puede modificar tu variable.

Ahora, el punto que más confunde a los principiantes: el asterisco significa dos cosas distintas según donde aparezca.

- En la declaración: `int *p;` significa "p es un apuntador". Es una etiqueta.
- En una expresión: `*p` significa "el valor al que apunta p". Es una acción: ir a la casa.

Mira este ejemplo que junta los dos usos y léelo con calma:

```
int x = 10;
int *p = &x;    // * en declaracion: p es apuntador; &x es la direccion de x
int y = *p;     // * en expresion: y recibe el valor que esta en la direccion de p
*p = 99;        // * en expresion del lado izquierdo: cambia el valor de x a 99
```

Después de esto, x vale 99 e y vale 10. Si logras leer esas cuatro líneas sin dudar, ya tienes el cincuenta por ciento de la unidad dominado. Si todavía dudas, vuelve a leerlas despacio: no hay prisa, y este es exactamente el punto donde vale la pena detenerse.

Apuntadores y tipos: por que el tipo importa

Te preguntarán: si un apuntador solo guarda una dirección, que es un número, por que tengo que decirle a que tipo apunta? Por que `int *p` y no simplemente "apuntador p"? La respuesta es que el tipo le dice al compilador dos cosas fundamentales.

Primero, el tipo dice cuánto mide lo que hay en la dirección. Un `int` ocupa (normalmente) 4 bytes, un `char` ocupa 1 byte, un `double` ocupa 8 bytes. Cuando haces `*p`, el compilador necesita saber cuántos bytes leer a partir de esa dirección. Si p es `int*`, lee 4 bytes y los interpreta como un entero. Si p es `char*`, lee 1 byte y lo interpreta como un carácter. La misma dirección, interpretada con tipos distintos, da valores distintos. El tipo es el lente con el que miras la memoria.

Segundo, el tipo dice cuánto avanzar cuando haces aritmética de apuntadores (`p + 1`), como verás en la siguiente sección. Un `int*` que avanza 1 salta 4 bytes; un `char*` que avanza 1 salta 1 byte.

Por eso no puedes mezclar tipos a lo loco:

```
int x = 65;
int *pi = &x;
char *pc = &x;    // El compilador se queja: tipos incompatibles
```

Bueno, en realidad C te deja hacerlo con un cast, pero el compilador te lanza una advertencia (warning), y en esta unidad vas a aprender a tomar los warnings en serio (la unidad 10 habla de eso a fondo). Si fuerzas un char* a apuntar a un int y luego haces *pc, solo lees 1 byte de los 4 del int, y obtienes un valor parcial que probablemente no es lo que esperabas. Moraleja: el tipo del apuntador debe corresponder al tipo de la variable a la que apunta.

Hay un tipo especial: void*. Un apuntador a void es un apuntador generico: puede guardar la dirección de cualquier cosa, pero no puedes desreferenciarlo directamente porque el compilador no sabe cuántos bytes leer. Es como un papelito donde anotaste una dirección pero no anotaste que tipo de casa es. Se usa mucho en funciones genericas (como malloc, que verás en la unidad 9) y siempre hay que convertirlo al tipo correcto antes de usarlo. Ya lo viste en acción: printf con %p pide un void*.

El apuntador NULL: no apunto a ningún lado

Que pasa si declaras un apuntador pero todavia no tienes a dónde apuntarlo? Esto:

```
int *p;
```

En ese momento p existe, ocupa su espacio en memoria, pero su contenido es basura: lo que haya quedado ahí antes. Es un papelito con una dirección inventada, escrita al azar. Si por accidente haces *p con ese apuntador sin inicializar, vas a ir a una dirección aleatoria y vas a leer o escribir en un lugar que no te corresponde. Eso es uno de los errores más comunes y más peligrosos en C, y lo veremos en la sección de errores.

La solución es NULL: un valor especial que significa "este apuntador no apunta a ningún lado". Para usarlo incluyes <stddef.h> o <stdio.h> (que ya lo trae en la práctica):

```
int *p = NULL;
```

Piensa en NULL como un papelito en blanco con una nota que dice "todavía sin dirección". Es honesto: el programa sabe que no debe seguir ese papelito. Si intentas desreferenciar NULL (*p cuando p es NULL), el programa se detiene con un error (segmentation fault). Eso suena mal, pero en realidad es bueno: es un error claro e inmediato, mucho mejor que escribir en una dirección aleatoria y que el programa siga funcionando mal en silencio durante horas.

La buena práctica de oro es esta: todo apuntador que declares debe nacer apuntando a algo válido o nacer siendo NULL. Y antes de desreferenciar un apuntador que recibiste (por ejemplo, como parámetro de una función), verifica que no sea NULL:

```
if (p != NULL) {
    *p = 10;
}
```

Este hábito te va a ahorrar noches enteras de depuración. Grábatelo.

Aritmética de apuntadores: $p + 1$ no es lo que crees

Aquí viene una de las ideas más elegantes y más sorprendentes de C. Si p es un apuntador a int y haces $p + 1$, no obtienes la dirección más uno: obtienes la dirección del siguiente int completo. Es decir, el compilador multiplica automáticamente por el tamaño del tipo.

```
int *p = /* direccion 1000, supongamos */;
int *q = p + 1; // q es la direccion 1004, no la 1001
```

¿Por qué? Porque $p + 1$ significa "el siguiente elemento", y un int ocupa 4 bytes, así que el siguiente int empieza 4 bytes después. Si fuera un char* (1 byte), $p + 1$ sí sería la dirección más uno. Si fuera un double* (8 bytes), $p + 1$ saltaría 8 bytes. El compilador hace la cuenta por ti: $p + n$ avanza n elementos completos, no n bytes.

Dibujemos la memoria para un arreglo de enteros. Supon que tienes int nums[4] que empieza en la dirección 1000:

Dirección	Contenido	Elemento
1000	10	nums[0]
1004	20	nums[1]
1008	30	nums[2]
1012	40	nums[3]

Si p apunta a nums[0] (dirección 1000):

- `p` apunta a 1000, `*p` es 10.
- `p + 1` apunta a 1004, `*(p + 1)` es 20.
- `p + 2` apunta a 1008, `*(p + 2)` es 30.
- `p - 1` apuntaría a 996, que está ANTES del arreglo: peligro, no lo hagas.

También puedes restar dos apuntadores del mismo tipo: `q - p` te dice cuántos elementos hay entre ellos, no cuántos bytes. Y puedes comparar apuntadores con `<`, `>`, `==` para saber cuál dirección es menor o si dos apuntadores apuntan al mismo lugar.

Lo que NO puedes hacer es sumar dos apuntadores (`p + q` no tiene sentido: qué significaría sumar dos direcciones?) ni multiplicarlos ni dividirlos. La aritmética de apuntadores está limitada a: apuntador + entero, apuntador - entero, apuntador - apuntador, y comparaciones. Todo lo demás es error.

Esta aritmética solo tiene sentido dentro de un mismo bloque de memoria, como un arreglo. Si tomas la dirección de una variable suelta `int x` y haces `(&x) + 1`, estás apuntando a memoria que no te pertenece. El compilador no te detiene, pero es un error lógico grave.

La relación íntima entre arreglos y apuntadores

Preparate para el momento "aha" más grande de la unidad: en C, el nombre de un arreglo ES un apuntador a su primer elemento. No es parecido a un apuntador, no funciona como un apuntador: en casi todos los contextos, el nombre del arreglo se convierte automáticamente en la dirección de su primer elemento.

```
int nums[4] = {10, 20, 30, 40};
int *p = nums;    // p apunta a nums[0]. No necesitas el &: nums YA es una
dirección.
```

Después de eso, `p[0]`, `p[1]`, `p[2]`, `p[3]` funcionan igual que `nums[0]`, `nums[1]`, etc. Y al revés: `nums[2]` es exactamente lo mismo que `*(nums + 2)`. La notación de corchetes es azúcar sintáctica: el compilador convierte `a[i]` en `*(a + i)` siempre. Por eso puedes escribir `2[nums]` y funciona (es `*(2 + nums)`, lo mismo que `*(nums + 2)`): es un truco curioso que demuestra que los corchetes son solo una forma cómoda de escribir aritmética de apuntadores. No lo uses en código real, pero entiende por qué funciona.

Hay una diferencia importante que debes conocer: el nombre del arreglo es un apuntador constante, no le puedes asignar otra dirección. Esto es válido:

```
int *p = nums;
p = p + 1;    // Bien: p es una variable apuntador normal
```

Pero esto NO es válido:

```
nums = nums + 1;    // Error: no puedes reasignar el nombre del arreglo
```

El nombre del arreglo siempre apunta al inicio; es como un papelito pegado con cinta adhesiva a la primera casa. Puedes leer la dirección que tiene anotada, pero no puedes cambiar lo que dice.

Esta relación explica algo que quizá te intrigaba desde la unidad 6: por que cuando pasas un arreglo a una función no necesitas el &, y por que la función puede modificar el arreglo original. Mira:

```
void llenar(int arr[], int n) { ... }

int nums[5];
llenar(nums, 5);
```

Cuando escribes `int arr[]` como parámetro, el compilador lo trata como `int *arr`: un apuntador. Y al llamar `llenar(nums, 5)`, le estás pasando la dirección del primer elemento. La función recibe el papelito con la dirección, así que cuando hace `arr[0] = 99` está escribiendo directamente en tu arreglo original. Por eso los arreglos siempre se pasan "por referencia" en C, aunque C oficialmente solo tenga paso por valor: lo que se pasa por valor es la dirección, y con la dirección puedes modificar el original. Esta idea es el puente directo a la siguiente sección.

Recorrer arreglos con apuntadores

Con lo que sabes, ya puedes recorrer un arreglo de dos formas equivalentes. La forma clásica con índices:

```
int nums[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
    printf("%d ", nums[i]);
}
```

Y la forma con apuntadores:

```
int nums[5] = {10, 20, 30, 40, 50};
int *p = nums;
for (int i = 0; i < 5; i++) {
    printf("%d ", *(p + i));
}
```

Pero hay una tercera forma, la más "puntera" de todas, que avanza el apuntador en cada vuelta:

```
int nums[5] = {10, 20, 30, 40, 50};
for (int *p = nums; p < nums + 5; p++) {
    printf("%d ", *p);
}
```

Lee ese for con calma: p empieza apuntando al primer elemento; mientras p sea menor que la dirección justo después del último elemento (nums + 5), imprime *p y avanza p al siguiente elemento con p++. Es elegante, es rápido y es el estilo que vas a ver en código C profesional. Fíjate que nums + 5 es una dirección válida para comparar aunque no exista un sexto elemento: C permite apuntar "uno más allá" del final para hacer comparaciones, pero no permite desreferenciarlo.

Cuál forma debes usar? La que te resulte más clara. Los índices son más legibles para la mayoría; los apuntadores pueden ser más eficientes en ciertos casos y son indispensables cuando trabajas con memoria dinámica (unidad 9). Lo importante es que entiendas que las tres hacen exactamente lo mismo por dentro.

Apuntadores como parámetros: el famoso swap

En la unidad 5 aprendiste que C pasa los argumentos por valor: la función recibe una copia. Eso significa que una función normal no puede modificar las variables de quien la llamó. Mira el problema:

```
void intercambiar_mal(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int x = 5, y = 9;
intercambiar_mal(x, y);
// x sigue siendo 5, y sigue siendo 9. No paso nada.
```

La función recibió copias de x e y, las intercambio entre ellas, y las copias se destruyeron al terminar. Las originales ni se enteraron. Es como si le hubieras dado a alguien fotocopias de tus papelitos: puede rayar las fotocopias todo lo que quiera, tus originales siguen intactos.

La solución es pasarle a la función los papelitos de verdad: las direcciones. Es decir, pasar apuntadores:

```
void intercambiar(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int x = 5, y = 9;
intercambiar(&x, &y);
// Ahora x es 9 y y es 5. Si funciona.
```

Que pasó aquí? Al llamar `intercambiar(&x, &y)`, le pasaste las direcciones de x e y. Dentro de la función, a es un apuntador a x y b es un apuntador a y. Cuando la función hace `*a = *b`, está diciendo: "ve a la casa de x y guarda ahí lo que hay en la casa de y". Como las casas son las originales, los cambios son permanentes.

Dibujemos la memoria durante la llamada:

En main:	En intercambiar:
x vive en 0x100, vale 5	a guarda 0x100 (apunta a x)
y vive en 0x104, vale 9	b guarda 0x104 (apunta a y)
	temp es una variable local

Paso a paso dentro de la función: `temp = *a` guarda 5 en temp. `*a = *b` escribe 9 en la casa 0x100 (x ahora vale 9). `*b = temp` escribe 5 en la casa 0x104 (y ahora vale 5). Intercambio completo.

A esto se le llama "paso por referencia simulado": C no tiene paso por referencia de verdad como otros lenguajes, pero pasar la dirección logra el mismo efecto. Es una de las técnicas más usadas en C: cada vez que veas una función que recibe un apuntador, pregunta "¿qué va a modificar esta función?".

Este mismo truco permite que una función "regrese" varios valores. Como return solo puede devolver una cosa, si necesitas que una función calcule dos o tres resultados, le pasas apuntadores donde escribirlos:

```
void dividir(int dividendo, int divisor, int *cociente, int *residuo) {
    *cociente = dividendo / divisor;
    *residuo = dividendo % divisor;
}

int c, r;
dividir(17, 5, &c, &r);
// c es 3, r es 2
```

La función no regresa nada (void), pero dejó los dos resultados escritos en c y r. Este patrón lo verás por todas partes en C profesional, así que acostúmbrate a leerlo.

Apuntadores a apuntadores: la idea básica

Si un apuntador es una variable que guarda una dirección, y toda variable tiene una dirección... entonces un apuntador también tiene una dirección, y puede existir un apuntador que apunte a otro apuntador. Se declara con dos asteriscos:

```
int x = 10;
int *p = &x;    // p apunta a x
int **pp = &p;  // pp apunta a p (apuntador a apuntador a int)
```

Dibujemos:

Dirección	Contenido	Variable
0x200	10	x
0x208	0x200	p (guarda la dirección de x)
0x210	0x208	pp (guarda la dirección de p)

Para llegar al valor 10 desde pp necesitas dos saltos: *pp te lleva a p (el papelito que apunta a x), y **pp te lleva al 10. Es como un papelito que dice "la dirección está anotada en otro papelito que está en tal cajón"**.

Cuando se usa esto en la vida real? El caso más común es cuando una función necesita modificar un apuntador (no el valor al que apunta, sino el apuntador mismo). Por ejemplo, una función que reserva memoria y quiere dejar el apuntador apuntando a la memoria nueva: como el apuntador se pasaría por valor (una copia del papelito), la función necesita recibir la dirección del papelito, es decir, un apuntador al apuntador. Lo verás en acción en la unidad 9 con malloc. Por ahora, quedate con la idea y con saber leer la sintaxis: dos asteriscos, dos saltos.

Dibujar la memoria en papel: tu superpoder

Te voy a enseñar la técnica que usan hasta los programadores con años de experiencia cuando un código con apuntadores se les atora: dibujar la memoria en papel. No es cosa de principiantes, es cosa de profesionales. La regla es simple: por cada variable dibujas una cajita con tres datos: el nombre, la dirección (inventada, no importa el número exacto) y el contenido. Y por cada apuntador dibujas una flecha desde su cajita hasta la cajita a la que apunta.

Vamos a hacerlo juntos con el swap, paso a paso. Código:

```
int x = 5, y = 9;
intercambiar(&x, &y);
```

Antes de la llamada, dibujas:

```
x: dir 100, contenido 5
y: dir 104, contenido 9
```

Al entrar a `intercambiar(int *a, int *b)`, se crean `a` y `b` como variables locales que reciben las direcciones:

```
a: dir 200, contenido 100 ---> flecha a la caja x
b: dir 204, contenido 104 ---> flecha a la caja y
```

Línea `temp = *a`: sigues la flecha de `a`, llegas a `x`, lees 5, lo guardas en `temp`:

```
temp: dir 208, contenido 5
```

Línea `*a = *b`: sigues la flecha de `b`, lees 9 de la caja `y`; sigues la flecha de `a`, llegas a la caja `x` y escribes 9 ahí. Ahora `x` vale 9. La flecha no cambio, cambio el contenido de la casa.

Línea `*b = temp`: sigues la flecha de `b`, llegas a la caja `y` y escribes 5. Ahora `y` vale 5.

Resultado final en el dibujo: `x` vale 9, `y` vale 5. Las flechas de `a` y `b` nunca se movieron; solo cambiaron los contenidos de las casas. Esa es la lección visual más importante de toda la unidad: un apuntador es una flecha, y desreferenciar es seguir la flecha.

Haz este dibujo cada vez que te atores con un ejercicio de esta unidad. Te toma dos minutos y te ahorra dos horas de mirar el código sin entenderlo. En serio: el papel no falla.

Apuntadores y cadenas: un adelanto

En la unidad 7 (clase 21) verás las cadenas de texto a fondo, pero aquí te adelanto lo esencial porque las cadenas en C SON apuntadores en la práctica. Una cadena es un arreglo de char terminado en el carácter especial '\0' (el cero). Y como el nombre de un arreglo es un apuntador, una cadena se maneja con char*.

```
char *saludo = "Hola";
```

Esta línea crea la cadena "Hola" en memoria (5 cajitas: 'H', 'o', 'l', 'a', '\0') y el apuntador saludo apunta a la primera. Con aritmética de apuntadores puedes caminar por la cadena:

```
char *p = saludo;
while (*p != '\0') {
    printf("%c", *p);
    p++;
}
```

Este ciclo imprime Hola: p empieza en la 'H', imprime *p, avanza a la 'o', y así hasta que *p es '\0', que marca el final. Observa que no usamos ningún índice ni la longitud de la cadena: el '\0' es el que dice "aquí termina". Así funcionan por dentro strlen, strcpy y todas las funciones de string.h: son ciclos con apuntadores que caminan hasta el '\0'.

Un detalle importante: un literal como "Hola" vive en una zona de memoria de solo lectura. Puedes leerlo con el apuntador, pero no debes modificarlo:

```
char *s = "Hola";
*s = 'h'; // MAL: la cadena literal es de solo lectura
```

Si necesitas una cadena modificable, declárala como arreglo: char s[] = "Hola"; ahí sí puedes cambiar letras. Esta distinción te va a salvar de errores extraños cuando llegues a la unidad de cadenas.

Errores comunes con apuntadores (y como evitarlos)

Esta sección vale oro. La mayoría de los dolores de cabeza con apuntadores vienen de una lista corta de errores que todo programador de C comete al menos una vez. Conocelos de antemano y te ahorras horas.

Error 1: usar un apuntador sin inicializar. El papelito con dirección al azar.

```
int *p;
*p = 42;    // PELIGRO: p no apunta a ningun lugar valido
```

p contiene basura, así que `*p = 42` escribe un 42 en una dirección aleatoria. Puede que no pase nada visible, puede que el programa se cierre, puede que corrompa otra variable. Solución: inicializa siempre. `int *p = NULL;` o `int *p = &x;`.

Error 2: desreferenciar NULL.

```
int *p = NULL;
printf("%d", *p);    // El programa se detiene: segmentation fault
```

Al menos este error es ruidoso: el programa muere de inmediato y sabes dónde. Pero mejor evitarlo verificando `if (p != NULL)` antes de usar apuntadores que vienen de fuera.

Error 3: confundir `*` en la declaración con `*` en el uso. Este es un error de lectura, no de ejecución, pero causa muchos bugs:

```
int x = 5;
int *p = &x;
int y = *p;    // Bien: y = 5
int z = p;     // MAL: z recibe la direccion, no el 5 (y el compilador advierte)
```

Si quieres el valor, usa `*p`. Si quieres la dirección, usa `p`. Parece obvio escrito así, pero cuando el código crece es fácil confundirse.

Error 4: apuntador colgando (dangling pointer). Apuntar a algo que ya no existe:

```
int *p;
{
    int x = 99;
    p = &x;
}    // x muere aqui: su casa se libera
*p = 1;    // PELIGRO: la casa de x ya tiene otro dueño
```

Las variables locales mueren cuando termina su bloque. Si guardaste su dirección en un apuntador que sobrevive al bloque, el apuntador queda "colgando": apunta a una casa que ya se rentó a otro. Nunca regreses la dirección de una variable local desde una función; es el error clásico:

```
int *crear(void) {  
    int x = 10;  
    return &x;    // MAL: x muere al salir de la funcion  
}
```

El compilador moderno te advierte de esto. Hazle caso.

Error 5: salirse del arreglo con aritmética de apuntadores.

```
int nums[3] = {1, 2, 3};  
int *p = nums;  
*(p + 10) = 99;    // PELIGRO: escribes fuera del arreglo
```

El compilador no te detiene. Tu programa escribe en memoria ajena y luego pasan cosas raras en lugares que no tienen nada que ver. La regla es simple: la aritmética de apuntadores solo es válida dentro del bloque de memoria al que apunta.

Error 6: olvidar el & al llamar una función que espera apuntador, o poner * de más:

```
void leer(int *n);  
int edad;  
leer(edad);    // MAL: la funcion espera una direccion, le pasaste un entero  
leer(&edad);    // Bien
```

El compilador te avisa con un warning. Otra razon para compilar siempre con -Wall y no ignorar advertencias.

Buenas prácticas con apuntadores

Cierra la teoria con estos habitos. No son opcionales: son lo que separa al programador que sufre con apuntadores del que los disfruta.

1. Inicializa todo apuntador al declararlo: a una dirección válida o a NULL. Sin excepciones.
2. Verifica NULL antes de desreferenciar apuntadores que recibes como parámetros o que vienen de funciones que pueden fallar (como malloc en la unidad 9).
3. Compila con gcc -Wall -Wextra y lee cada warning como si fuera un error. La mayoría de los bugs de apuntadores avisan antes de morder.
4. Dibuja la memoria en papel cuando te atores. Casas, direcciones, flechas de quien apunta a quien. Funciona siempre; hasta los programadores con experiencia lo hacen con casos difíciles.

5. No hagas trucos elegantes con apuntadores hasta dominar lo básico. El código claro con apuntadores es mejor que el código ingenioso con apuntadores.
6. Respeta los tipos: no mezcles `int*` con `char*` sin una razón de peso y un cast explícito.
7. Recuerda que el nombre de un arreglo no se puede reasignar, y que la aritmética solo vale dentro del bloque.

Con esto tienes la teoría completa. Ahora viene la parte donde todo se vuelve concreto: los ejemplos. Escríbelos, ejecútalos, rompelos a propósito y obsérvalos. En lucasegaacademy.com puedes correr cada ejemplo en el navegador y preguntarle al Profesor Buho lo que no te quede claro.

Preguntas que todos se hacen (y sus respuestas honestas)

Antes de pasar a los ejemplos, respondo las preguntas que me hacen todos los estudiantes al llegar a este punto. Si alguna ronda por tu cabeza, aquí está.

"Puedo vivir sin apuntadores y programar en C de todos modos?" Técnicamente puedes escribir programas sencillos sin tocar un apuntador de forma consciente, pero en cuanto quieras hacer algo serio (pedir memoria dinámica, usar cadenas de verdad, trabajar con archivos de forma eficiente, usar bibliotecas) los apuntadores aparecen por todas partes. Además, incluso el `scanf` que ya usas recibe apuntadores: `scanf("%d", &edad)` le pasa la dirección de `edad` para poder escribir en ella. Ya estabas usando apuntadores sin saberlo.

"Por que otros lenguajes como Python no tienen apuntadores visibles?" Si los tienen, solo que escondidos. Cuando en Python haces `lista2 = lista1`, ambas variables apuntan a la misma lista en memoria, y si modificas una se modifica la otra. Es exactamente el comportamiento de apuntadores, pero sin que veas las direcciones. C te muestra la maquinaria; otros lenguajes te la ocultan para que no te equivoques. La ventaja de C es que entiendes lo que pasa de verdad; el precio es que los errores son tuyos.

"El `segmentation fault` significa que rompi la computadora?" No. Significa que tu programa intentó tocar memoria que no le corresponde y el sistema operativo lo detuvo antes de hacer dano. Es el sistema protegiéndose, y a ti te sirve como alarma: si ves `"segmentation fault"`, casi seguro desreferenciaste un apuntador malo (`NULL`, sin inicializar o fuera de rango). Celebra el error: te está diciendo exactamente donde buscar.

"Debo usar apuntadores para todo ahora que se usarlos?" No. Esa es la trampa del principiante entusiasmado. Usa apuntadores donde aportan algo: para modificar variables desde funciones, para recorrer bloques de memoria, para memoria dinámica. Para una suma sencilla dentro de una función, los valores normales están perfectos. El buen programador de C no es el que más asteriscos escribe, sino el que usa cada herramienta en su lugar.

"Cuánto tarda en hacer clic este tema?" Para la mayoría, entre una y tres semanas de práctica intermitente. Hay un momento específico que todos recuerdan: de repente estás depurando algo, dibujas las flechas en papel, y todo encaja. No puedes forzar ese momento, pero si puedes provocarlo: escribe código, rompe cosas, dibuja memoria. Llega solo.

"Que sigue después de dominar apuntadores?" La unidad 9: memoria dinámica con malloc y archivos. Ahí los apuntadores dejan de ser un tema de examen y se convierten en tu herramienta diaria. Y después, cuando veas listas enlazadas, árboles y tablas hash en estructuras de datos, vas a descubrir que ya sabías el noventa por ciento: esas estructuras son solo apuntadores organizados con imaginación.

Ejemplos paso a paso

Ejemplo 1: imprimir direcciones con & y %p

El primer programa de todo estudiante de apuntadores: ver las direcciones con tus propios ojos.

```
#include <stdio.h>

int main() {
    int edad = 25;
    char letra = 'A';
    double precio = 99.9;

    printf("edad:   valor=%d   direccion=%p\n", edad, (void*)&edad);
    printf("letra:  valor=%c   direccion=%p\n", letra, (void*)&letra);
    printf("precio: valor=%f   direccion=%p\n", precio, (void*)&precio);

    return 0;
}
```

Ejecuta y observa: cada variable tiene su propia dirección. Fíjate en cuanto se separan: la dirección de letra y la de edad probablemente difieren en 4, y la de precio en 8 respecto a la

anterior, reflejando los tamaños de cada tipo. Ejecuta el programa dos veces y compara: las direcciones cambian entre ejecuciones. Eso es normal y no afecta nada: lo importante son las relaciones, no los números exactos.

Ejemplo 2: declarar un apuntador y modificar la variable a través del

Aquí ves el ciclo completo: declarar, apuntar, leer y escribir mediante el apuntador.

```
#include <stdio.h>

int main() {
    int x = 10;
    int *p = &x;    // p guarda la direccion de x

    printf("x vale %d\n", x);
    printf("p guarda la direccion %p\n", (void*)p);
    printf("*p (lo que hay en esa direccion) vale %d\n", *p);

    *p = 42;    // Escribo en la casa de x usando el papelito

    printf("Despues de *p = 42, x vale %d\n", x);

    return 0;
}
```

Salida esperada:

```
x vale 10
p guarda la direccion 0x7ffd... (la que sea)
*p (lo que hay en esa direccion) vale 10
Despues de *p = 42, x vale 42
```

La última línea es la magia: cambiaste x sin mencionar x. Si entiendes por qué funciona, entiendes el corazón de los apuntadores.

Ejemplo 3: swap, intercambiar dos valores con una función

El ejemplo clásico que demuestra para que sirven los apuntadores como parámetros. Primero la versión que no funciona, para que veas el problema con tus ojos, y luego la que sí.

```

#include <stdio.h>

void intercambiar_mal(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    printf(" Dentro de intercambiar_mal: a=%d b=%d\n", a, b);
}

void intercambiar(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 9;

    printf("Antes: x=%d y=%d\n", x, y);
    intercambiar_mal(x, y);
    printf("Despues de la version mala: x=%d y=%d (no cambio nada)\n", x, y);

    intercambiar(&x, &y);
    printf("Despues de la version buena: x=%d y=%d (si cambio)\n", x, y);

    return 0;
}

```

La versión mala imprime a=9 b=5 adentro, pero afuera x e y siguen igual: trabajo con fotocopias. La versión buena recibe las direcciones y modifica las casas originales. Este patrón (recibir int* y modificar *a) lo vas a ver cientos de veces en tu vida como programador de C.

Ejemplo 4: recorrer un arreglo con un apuntador

Las tres formas de recorrer, una junto a otra, para que compares.


```

#include <stdio.h>

int main() {
    int nums[5] = {10, 20, 30, 40, 50};

    printf("Con indices:      ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", nums[i]);
    }
    printf("\n");

    printf("Con *(p + i):      ");
    int *p = nums;
    for (int i = 0; i < 5; i++) {
        printf("%d ", *(p + i));
    }
    printf("\n");

    printf("Avanzando el apuntador: ");
    for (int *q = nums; q < nums + 5; q++) {
        printf("%d ", *q);
    }
    printf("\n");

    return 0;
}

```

Las tres imprimen 10 20 30 40 50. La tercera es la favorita de los programadores de C veteranos: el apuntador camina por el arreglo casa por casa. Observa la condición `q < nums + 5`: compara direcciones, y `nums + 5` es "uno más allá del final", válido para comparar pero no para desreferenciar.

Ejemplo 5: aritmética de apuntadores visualizada

Este ejemplo imprime direcciones y valores lado a lado para que veas con tus ojos como `p + 1` salta según el tipo.

```

#include <stdio.h>

int main() {
    int nums[4] = {10, 20, 30, 40};
    int *p = nums;

    for (int i = 0; i < 4; i++) {
        printf("p + %d -> direccion %p, valor %d\n", i, (void*)(p + i), *(p + i));
    }

    printf("\nAhora con char:\n");
    char letras[4] = {'a', 'b', 'c', 'd'};
    char *q = letras;

    for (int i = 0; i < 4; i++) {
        printf("q + %d -> direccion %p, valor %c\n", i, (void*)(q + i), *(q + i));
    }

    return 0;
}

```

Observa las direcciones: en el arreglo de int, cada paso avanza 4 (en hexadecimal verás saltos de 4 en 4); en el de char, cada paso avanza 1. El mismo + 1, distinto salto, según el tipo. Esa es toda la aritmética de apuntadores en una imagen.

Ejemplo 6: una función que regresa varios valores con apuntadores

Como return solo devuelve una cosa, usamos parámetros de salida para devolver más resultados.

```

#include <stdio.h>

void analizar(int n, int *doble, int *triple, int *es_par) {
    *doble = n * 2;
    *triple = n * 3;
    *es_par = (n % 2 == 0);    // 1 si es par, 0 si no
}

int main() {
    int d, t, par;

    analizar(7, &d, &t, &par);
    printf("n=7: doble=%d triple=%d es_par=%d\n", d, t, par);

    analizar(10, &d, &t, &par);
    printf("n=10: doble=%d triple=%d es_par=%d\n", d, t, par);

    return 0;
}

```

La función es void pero "regresa" tres valores escribiendolos en las direcciones que recibió. En main declaras las variables, pasas sus direcciones con &, y al volver de la función las variables ya tienen los resultados. Este patrón es estándar en C: cuando veas parámetros que son apuntadores, casi siempre son valores de salida.

Ejemplo 7: apuntador a apuntador

Dos niveles de indirección, paso a paso y sin miedo.

```

#include <stdio.h>

int main() {
    int x = 10;
    int *p = &x;    // p apunta a x
    int **pp = &p;  // pp apunta a p

    printf("x = %d\n", x);
    printf("*p = %d (un salto desde p)\n", *p);
    printf("**pp = %d (dos saltos desde pp)\n", **pp);

    **pp = 99;    // Cambio x dando dos saltos
    printf("Despues de **pp = 99, x = %d\n", x);

    printf("Direccion de x: %p\n", (void*)&x);
    printf("p guarda:      %p (la direccion de x)\n", (void*)p);
    printf("pp guarda:     %p (la direccion de p)\n", (void*)pp);

    return 0;
}

```

Sigue la cadena: pp guarda la dirección de p, p guarda la dirección de x, x guarda el 10. **pp hace los dos saltos y llega al 10. Cuando cambias pp, cambias x.** En la unidad 9 verás el uso real de esto: funciones que necesitan modificar un apuntador.

Ejemplo 8: mini programa completo, agenda con busqueda por apuntador

Un programa que junta todo: arreglos, apuntadores como parámetros, recorrido con aritmética y valores de salida.

```

#include <stdio.h>

int buscar(int *arr, int n, int objetivo, int *posicion) {
    for (int *p = arr; p < arr + n; p++) {
        if (*p == objetivo) {
            *posicion = p - arr;    // cuantos elementos hay entre arr y p
            return 1;                // encontrado
        }
    }
    return 0;    // no encontrado
}

int main() {
    int datos[8] = {15, 42, 7, 23, 99, 4, 61, 30};
    int objetivo, pos;

    printf("Arreglo: ");
    for (int *p = datos; p < datos + 8; p++) {
        printf("%d ", *p);
    }
    printf("\nQue numero buscas? ");
    scanf("%d", &objetivo);

    if (buscar(datos, 8, objetivo, &pos)) {
        printf("Encontrado en la posicion %d\n", pos);
    } else {
        printf("Ese numero no esta en el arreglo\n");
    }

    return 0;
}

```

Observa todo lo que este programa usa sin que te des cuenta: `datos` se pasa como `int*` (el nombre del arreglo es la dirección), dentro de `buscar` el parámetro `arr` es un apuntador que se recorre con `p++`, `p - arr` calcula la posición restando apuntadores, y `&pos` le da a la función donde escribir el resultado. Si puedes leer este programa de arriba a abajo sin perderte, felicidades: ya piensas con apuntadores.

Ejemplo 9: medir una cadena con un apuntador (tu propio `strlen`)

La biblioteca estándar tiene `strlen` para medir cadenas, pero vamos a escribirla nosotros para ver como camina un apuntador hasta el `'\0'`.

```

#include <stdio.h>

int mi_strlen(char *s) {
    int contador = 0;
    while (*s != '\0') {
        contador++;
        s++;
    }
    return contador;
}

int main() {
    char *texto = "Programar en C";

    printf("La cadena \"%s\" mide %d caracteres\n", texto, mi_strlen(texto));
    printf("La cadena vacia mide %d caracteres\n", mi_strlen(""));

    return 0;
}

```

Explicación: `mi_strlen` recibe un `char*` que apunta al primer carácter. El ciclo mira `*s`: mientras no sea el terminador `'\0'`, cuenta uno y avanza el apuntador al siguiente carácter con `s++`. Cuando `*s` es `'\0'`, el ciclo termina y regresamos el contador. Con "Programar en C" el resultado es 13 (cuenta los espacios, pero no el `'\0'`). Con la cadena vacía "", el primer carácter ya es `'\0'`, así que el resultado es 0. Fíjate en algo elegante: la función no necesita saber el tamaño de antemano; el `'\0'` es su guía. Así funciona la `strlen` real por dentro.

Ejemplo 10: copiar una cadena con apuntadores (tu propio `strcpy`)

Ahora copiamos una cadena carácter por carácter, incluyendo el `'\0'` final, que es el detalle que más se olvida.

```

#include <stdio.h>

void mi_strcpy(char *destino, char *origen) {
    while (*origen != '\0') {
        *destino = *origen;
        destino++;
        origen++;
    }
    *destino = '\0';    // no olvides el terminador
}

int main() {
    char original[] = "Hola, mundo";
    char copia[20];

    mi_strcpy(copia, original);

    printf("Original: %s\n", original);
    printf("Copia:    %s\n", copia);

    return 0;
}

```

Explicación: dos apuntadores caminan en paralelo: origen lee y destino escribe. En cada vuelta, `*destino = *origen` copia el carácter actual, y ambos apuntadores avanzan. Cuando origen llega al `'\0'`, el ciclo termina, pero falta un paso crucial: escribir el `'\0'` en el destino. Sin esa última línea, copia no tendría terminador y printf imprimiría basura después del texto. Ese olvido es uno de los bugs clásicos en C. Nota que `original` es un arreglo (`char original[]`), así que es modificable y su nombre se pasa como dirección; `copia` es un arreglo de 20 con espacio suficiente. Si el destino fuera más chico que el origen, escribiríamos fuera de su memoria: por eso la `strcpy` real es peligrosa y existen versiones seguras. Pero como ejercicio para entender apuntadores, es perfecto.

Práctica

Antes de ver las soluciones, intenta cada ejercicio por tu cuenta. Escribe el código, compilalo con `gcc -Wall -Wextra` y ejecutalo. Si algo no sale, lee el error, piensa, y vuelve a intentar antes de mirar la solución. Los apuntadores se aprenden haciendo, no leyendo.

Ejercicio 1: el detective de direcciones

Declara tres variables de distinto tipo: un int, un char y un double. Imprime el valor y la dirección de cada una. Luego declara un apuntador para cada variable, haz que apunte a ella e imprime el valor al que apunta usando el operador *. Verifica que los valores coincidan.

Ejercicio 2: modifica sin tocar

Declara una variable int con valor inicial 100. Declara un apuntador a int que apunte a ella. Sin usar el nombre de la variable (solo el apuntador), cambia su valor a 777 y luego imprimelo usando el nombre de la variable para comprobar que cambio. Explica en un comentario por qué funciona.

Ejercicio 3: swap de tres valores (rotación)

Escribe una función rotar que reciba tres apuntadores a int (int *a, int *b, int *c) y rote sus valores: el valor de a pasa a b, el de b pasa a c, y el de c pasa a a. Pruebala en main con x=1, y=2, z=3 e imprime antes y después. El resultado debe ser x=3, y=1, z=2.

Ejercicio 4: suma con aritmética de apuntadores

Escribe una función suma_arreglo que reciba un apuntador a int y el tamaño, y regrese la suma de los elementos. Pero con una condición: dentro de la función tienes prohibido usar corchetes []. Debes recorrer el arreglo solo con aritmética de apuntadores. Pruebala con un arreglo de 6 números.

Ejercicio 5: máximo y mínimo con parámetros de salida

Escribe una función extremos que reciba un arreglo de int, su tamaño, y dos apuntadores a int (int *max, int *min). La función debe encontrar el valor máximo y el mínimo del arreglo y escribirlos en *max y *min. No puede usar return para los resultados (la función es void). Pruebala en main e imprime los resultados.

Ejercicio 6: invierte el arreglo en su lugar

Escribe una función invertir que reciba un arreglo de int y su tamaño, e invierta el orden de los elementos sin usar un segundo arreglo. Pista: usa dos apuntadores, uno al inicio y otro al final, e intercambia los valores mientras los apuntadores se acercan. Pruebala con {1, 2, 3, 4, 5} y verifica que quede {5, 4, 3, 2, 1}. También pruebala con un arreglo de tamaño par.

Solución del ejercicio 1

```
#include <stdio.h>

int main() {
    int entero = 42;
    char letra = 'Z';
    double real = 3.14;

    int *pe = &entero;
    char *pl = &letra;
    double *pr = &real;

    printf("entero: valor=%d direccion=%p *pe=%d\n",
           entero, (void*)&entero, *pe);
    printf("letra: valor=%c direccion=%p *pl=%c\n",
           letra, (void*)&letra, *pl);
    printf("real: valor=%f direccion=%p *pr=%f\n",
           real, (void*)&real, *pr);

    return 0;
}
```

Explicación detallada: cada variable tiene su dirección, obtenida con &. Cada apuntador se declara con el tipo correspondiente: int* para el int, char* para el char, double* para el double. Esto es importante porque el tipo le dice al compilador cuántos bytes leer al desreferenciar: 4 para el int, 1 para el char, 8 para el double. Si mezclaras los tipos, *p leería una cantidad incorrecta de bytes y obtendrías basura. Al imprimir *pe, *pl y *pr obtienes exactamente los valores originales, lo que confirma que cada apuntador apunta a su variable. Observa también las direcciones: probablemente estén separadas según el tamaño de cada tipo, y cambiarían cada vez que ejecutes el programa.

Solución del ejercicio 2

```
#include <stdio.h>

int main() {
    int numero = 100;
    int *p = &numero;

    *p = 777;    // Modifico la variable sin mencionar su nombre

    printf("numero vale %d\n", numero);

    return 0;
}
```

Explicación detallada: p guarda la dirección de número, es el papelito con la dirección de la casa. La expresión *p significa "lo que hay en la dirección guardada en p", es decir, el contenido de la casa de número. Al escribir *p = 777, no estamos cambiando p (el papelito sigue diciendo lo mismo), estamos cambiando el contenido de la casa a la que apunta el papelito. Por eso, cuando después imprimimos número con su nombre, vale 777: es la misma casa, solo que entramos por la puerta del apuntador en lugar de por la puerta del nombre. Esta es exactamente la mecánica que permite que una función modifique variables que no son suyas: si le pasas la dirección, puede entrar a la casa.

Solución del ejercicio 3

```
#include <stdio.h>

void rotar(int *a, int *b, int *c) {
    int temp = *a;    // guardo el valor de a
    *a = *c;          // a recibe lo que tenia c
    *c = *b;          // c recibe lo que tenia b
    *b = temp;        // b recibe lo que tenia a (guardado en temp)
}

int main() {
    int x = 1, y = 2, z = 3;

    printf("Antes:  x=%d y=%d z=%d\n", x, y, z);
    rotar(&x, &y, &z);
    printf("Despues: x=%d y=%d z=%d\n", x, y, z);

    return 0;
}
```

Explicación detallada: queremos que el valor de a vaya a b, el de b vaya a c y el de c vaya a a. El problema es que al escribir `*a = *c` perdemos el valor original de a, por eso lo guardamos primero en temp. Sigamos el ejemplo con x=1, y=2, z=3: temp guarda 1 (el valor de *a). Luego `*a = *c` escribe 3 en x. Después `*c = *b` escribe 2 en z. Finalmente `*b = temp` escribe 1 en y. Resultado: x=3, y=1, z=2. La rotación quedó completa. Fíjate en el orden de las asignaciones: es el orden inverso al de la rotación, y eso no es casualidad. Si intentaras hacerlo en el orden "natural" (`*b = *a; *c = *b; *a = *c`) perderías valores por el camino, porque `*b = *a` destruye el valor original de b antes de que puedas pasarlo a c. La variable temporal es la que rompe ese problema, igual que en el swap de dos valores. Este ejercicio te entrena a pensar en que valor se pierde en cada paso, que es la habilidad central para manipular datos con apuntadores.

Solución del ejercicio 4

```
#include <stdio.h>

int suma_arreglo(int *arr, int n) {
    int suma = 0;
    for (int *p = arr; p < arr + n; p++) {
        suma += *p;
    }
    return suma;
}

int main() {
    int nums[6] = {5, 10, 15, 20, 25, 30};

    printf("La suma es %d\n", suma_arreglo(nums, 6));

    return 0;
}
```

Explicación detallada: la restricción de no usar corchetes te obliga a pensar como piensa el compilador. Cuando llamas `suma_arreglo(nums, 6)`, el nombre del arreglo se convierte en la dirección de su primer elemento, así que el parámetro `arr` es un `int*` que apunta a `nums[0]`. Dentro de la función declaramos un apuntador `p` que empieza en `arr` y avanza con `p++` en cada iteración. La condición `p < arr + n` compara direcciones: seguimos mientras `p` no haya llegado a la dirección justo después del último elemento. En cada vuelta, `*p` es el valor del elemento actual y lo sumamos. Cuando `p` llega a `arr + n`, el ciclo termina. Nota que nunca desreferenciamos `arr + n`, solo lo usamos para comparar, lo cual es válido. La suma de $5+10+15+20+25+30$ es 105. Este estilo de recorrido es el que usan las funciones de la biblioteca estándar de C por dentro, así que ya estás leyendo código profesional.

Solución del ejercicio 5

```
#include <stdio.h>

void extremos(int *arr, int n, int *max, int *min) {
    *max = arr[0];
    *min = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > *max) {
            *max = arr[i];
        }
        if (arr[i] < *min) {
            *min = arr[i];
        }
    }
}

int main() {
    int datos[7] = {34, 12, 89, 4, 67, 23, 51};
    int mayor, menor;

    extremos(datos, 7, &mayor, &menor);

    printf("Maximo: %d, Minimo: %d\n", mayor, menor);

    return 0;
}
```

Explicación detallada: la función necesita devolver dos resultados pero `return` solo puede devolver uno, así que usamos parámetros de salida: dos apuntadores donde la función escribiera las respuestas. En `main` declaramos `mayor` y `menor` sin inicializarlas (no hace falta,

porque la función las va a escribir antes de que las leamos) y pasamos sus direcciones con `&mayor` y `&menor`. Dentro de `extremos`, `*max` y `*min` son las variables originales de `main`, así que cada asignación las modifica directamente. Inicializamos ambos con `arr[0]` (el primer elemento) y luego recorremos desde el índice 1 comparando: si encontramos algo mayor que `*max`, lo actualizamos; si encontramos algo menor que `*min`, también. Con el ejemplo, el máximo es 89 y el mínimo es 4. Un detalle importante: esta función supone que `n` es al menos 1, porque lee `arr[0]` sin verificar. En código real agregarías una verificación al inicio. Y observa que aquí si usamos corchetes dentro de la función porque el ejercicio no lo prohibía: `arr[i]` es lo mismo que `*(arr + i)`, y mezclar ambos estilos es perfectamente normal en C.

Solución del ejercicio 6

```

#include <stdio.h>

void invertir(int *arr, int n) {
    int *inicio = arr;          // apunta al primer elemento
    int *fin = arr + n - 1;      // apunta al ultimo elemento
    while (inicio < fin) {
        int temp = *inicio;
        *inicio = *fin;
        *fin = temp;
        inicio++; // avanza hacia el centro
        fin--;    // retrocede hacia el centro
    }
}

void imprimir(int *arr, int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    int b[4] = {10, 20, 30, 40};

    invertir(a, 5);
    printf("Invertido (impar): ");
    imprimir(a, 5);

    invertir(b, 4);
    printf("Invertido (par):  ");
    imprimir(b, 4);

    return 0;
}

```

Explicación detallada: la idea es tener dos apuntadores que caminan uno hacia el otro desde los extremos, intercambiando valores a cada paso. inicio empieza en arr (el primer elemento) y fin empieza en arr + n - 1 (el último elemento: n - 1 porque los índices van de 0 a n-1). Mientras inicio este antes que fin, intercambiamos *inicio con *fin usando una variable temporal, y luego movemos inicio una posición adelante y fin una posición atrás. Sigamos el ejemplo {1,2,3,4,5}: primera vuelta intercambia 1 y 5, queda {5,2,3,4,1}; segunda vuelta intercambia 2 y 4, queda {5,4,3,2,1}; ahora inicio y fin apuntan ambos al 3 del centro, la condición inicio < fin ya no se

cumple y el ciclo termina. El elemento del centro en arreglos impares no necesita moverse, y el algoritmo lo maneja solo. Con el arreglo par {10,20,30,40}: primera vuelta intercambia 10 y 40, segunda intercambia 20 y 30, y luego inicio supera a fin y termina. Queda {40,30,20,10}. Fíjate que no usamos ningún arreglo auxiliar: todo se hace en el mismo espacio de memoria, intercambiando casa por casa. Esta técnica de "dos apuntadores que se acercan" es un patrón clásico que verás en muchos algoritmos, y aquí la aprendiste en su forma más pura.

Resumen de la unidad

Felicidades: acabas de terminar la unidad más famosa de C. Vamos a repasar lo esencial para que quede bien amarrado.

Una dirección de memoria es el número que identifica el lugar donde vive una variable, y el operador & te la da. Un apuntador es una variable que guarda una dirección; se declara con * (int *p significa "p es apuntador a int") y se hace apuntar a una variable con p = &edad.

El operador * tiene dos significados según el contexto: en la declaración dice "esto es un apuntador", y en una expresión significa "ve a la dirección y dame lo que hay ahí" (desreferenciar). Con *p puedes leer el valor apuntado y también modificarlo: *p = 30 cambia la variable original sin mencionar su nombre.

El tipo del apuntador importa porque dice cuántos bytes leer al desreferenciar y cuánto avanzar en la aritmética: p + 1 salta un elemento completo del tipo (4 bytes para int*, 1 byte para char*). NULL significa "no apunto a ningún lado" y es el valor seguro para un apuntador sin destino; verifica siempre antes de desreferenciar.

El nombre de un arreglo es un apuntador a su primer elemento (constante, no reasignable), y a[i] es exactamente *(a + i). Por eso las funciones reciben arreglos como apuntadores y pueden modificarlos. Pasar la dirección de una variable a una función (&x) permite que la función modifique la original mediante *p: es el paso por referencia simulado, y con el puedes hacer el swap clásico y regresar varios valores por parámetros de salida. Un apuntador a apuntador (int **pp**) **guarda la dirección de otro apuntador y se desreferencia con dos saltos (pp)**.

Los errores que debes evitar: apuntadores sin inicializar, desreferenciar NULL, apuntadores colgando a variables que ya murieron, salirte del arreglo con aritmética, olvidar el terminador '\0' al trabajar con cadenas, y confundir p (dirección) con *p (valor). Tus defensas: inicializar siempre, verificar NULL, compilar con -Wall -Wextra, dibujar la memoria en papel y respetar los tipos.

Y si tuviera que resumir toda la unidad en una sola frase para que la repitas antes de dormir, sería esta: un apuntador no es un valor, es la dirección donde vive un valor, y con esa dirección puedes ir, mirar y cambiar lo que hay ahí. Todo lo demás (la aritmética, los arreglos, el swap, los parámetros de salida, los apuntadores a apuntadores) son consecuencias de esa idea.

Un último consejo de esta unidad, y es en serio: no la des por terminada hoy. Vuelve a los ejemplos en tres días y ejecútalos otra vez. Los apuntadores son de esos temas que se asientan con el repaso: la primera vez entiendes la mecánica, la segunda vez entiendes el porqué, y la tercera vez ya piensas con ellos sin esfuerzo. Todos los programadores de C pasamos por ese proceso, sin excepción.

En la siguiente unidad (unidad 9) usarás todo esto para algo espectacular: pedir memoria prestada mientras el programa corre, con malloc, calloc, realloc y free, y para leer y escribir archivos. Los apuntadores dejan de ser teoría y se vuelven la herramienta con la que construyes programas de verdad. Nos vemos ahí, y recuerda que en lucasegaacademy.com tienes más ejercicios y al Profesor Buho para seguir practicando.

UNIDAD 9: Structs y memoria dinámica

Introducción

Imaginate que es tu primer día de trabajo en la recepción de una escuela. Te sientan frente a un escritorio y te ponen delante una montaña de papeles: las fichas de inscripción de todos los alumnos. Tomas una y la lees: nombre completo, edad, matrícula, promedio y carrera. Todo junto en una sola hoja, ordenado, fácil de encontrar. Nadie en su sano juicio guardaría el nombre en un cajón, la edad en otro cajón y el promedio en un tercero, para luego andar cazando los pedazos por toda la oficina cada vez que un alumno pide un trámite. Sería un caos lento, frágil y lleno de errores.

Pues bien: eso es justo lo que has estado haciendo en tus programas hasta este momento del curso. Cada dato vive en su propia variable, suelto, sin relación con los demás. Un `int` para la edad, un `float` para el promedio, un arreglo de `char` para el nombre. Para manejar a un solo alumno necesitas tres o cuatro variables que tú tienes que acordarte de mover juntas, pasar juntas a las funciones y mantener sincronizadas a mano. Con un alumno se puede. Con cincuenta se vuelve una pesadilla. Con un sistema real es imposible.

En esta unidad vas a aprender a fabricar tus propios tipos de datos. Así como C te regaló el `int`, el `float` y el `char`, tú vas a crear el tipo `Alumno`, el tipo `Producto`, el tipo `Fecha` o el que tu programa necesite. Esa es la primera mitad de la unidad: el `struct`, la ficha de registro del mundo de la programación. Un `struct` te permite meter varios datos distintos bajo un mismo techo y tratarlos como una sola cosa. El nombre, la edad, la matrícula y el promedio dejan de ser cuatro variables sueltas y se convierten en una sola ficha llamada `alumno`. La pasas a una función como una sola cosa, la guardas en un arreglo como una sola cosa, la piensas como una sola cosa. Tu código se vuelve más corto, más claro y mucho más parecido a cómo es el mundo real, porque en el mundo real las cosas vienen en paquete: un cliente tiene nombre, teléfono y dirección; un producto tiene código, descripción, precio y existencias; una fecha tiene día, mes y año. El `struct` es la herramienta que te deja modelar esa realidad dentro de tu programa.

Pero hay una segunda mitad, y es igual de poderosa. Hasta ahora toda la memoria que usas la decides antes de que el programa arranque: "este arreglo va a tener 100 espacios". Y si llegan 101 alumnos? O desperdicias memoria reservando de más por si acaso, o tu programa se queda corto y truena. La vida real no te dice por adelantado cuántos datos van a llegar. Por eso existe la memoria dinámica: la capacidad de pedirle memoria al sistema mientras el programa está corriendo, del tamaño exacto que necesitas, y devolverla cuando terminas. Es

como la diferencia entre comprar un camion de mudanza "por si acaso" y rentar la bodega del tamaño justo cuando ya sabes cuántas cajas tienes. Con malloc pides, con free devuelves, con calloc pides ya limpio y con realloc agrandas o achicas lo que ya pediste.

Piensalo con ejemplos de todos los días. La lista de contactos de tu celular no tiene un límite fijo de 100 nombres: crece cada vez que agregas a alguien. Un videojuego no sabe cuántos enemigos van a aparecer en el nivel: los crea sobre la marcha y los borra cuando los derrotas. Un editor de texto no reserva memoria para un millon de letras desde que lo abres: pide más conforme escribes. Todo eso es memoria dinámica por dentro. Cuando la domines, tus programas van a dejar de ser cajitas rígidas y van a volverse elasticos, capaces de crecer y encogerse según lo que el usuario necesite.

Estas dos herramientas juntas, structs y memoria dinámica, son las que separan a quien escribe programitas de quien construye sistemas. Una lista de alumnos que crece sola, un inventario que se agranda sin límite fijo, un catalogo de productos que se carga desde un archivo sin saber cuántos vienen: todo eso se construye con lo que vas a aprender aquí. No es casualidad que este tema llegue casi al final del curso: necesitas todo lo anterior (variables, arreglos, funciones, apuntadores) porque aquí todo se junta. Los apuntadores de la unidad pasada dejan de ser teoria abstracta y se vuelven la llave que abre la memoria dinámica. Si sentiste que los apuntadores eran difíciles, aquí por fin vas a ver para que sirven de verdad, y todo va a hacer clic.

El plan de la unidad es este. Primero dominas el struct: qué es, cómo se declara, cómo se accede a sus campos con el punto, cómo se inicializa, cómo se hacen arreglos de structs y cómo se pasan a funciones por valor y por apuntador (ahí entra en escena el operador flecha ->, que te va a encantar). Luego pasas por sus primos: la union, el enum y el typedef, tres herramientas que te ahorran dolores de cabeza y hacen tu código más legible. Y cierras con la memoria dinámica: el heap contra el stack, malloc y free, calloc y realloc, y los errores clásicos que todo programador de C comete una vez (y solo una vez, si aprende la lección): olvidar liberar memoria, usar memoria ya liberada y no verificar si malloc te regreso NULL.

Para cuando termines esta unidad vas a poder diseñar tus propios tipos de datos y manejar memoria como un profesional. Es un salto grande, así que tomalo con calma, escribe cada ejemplo con tus manos y rompe cosas a propósito para ver qué pasa. Así se aprende C de verdad. Y recuerda que en lucasegaacademy.com tienes todo el curso completo gratis para repasar cualquier tema cuando lo necesites. Vamos a darle.

Explicación detallada

Que es un struct: tu ficha de registro personal

Un struct es un tipo de dato que TU defines, formado por varias variables agrupadas bajo un mismo nombre. Cada una de esas variables internas se llama campo o miembro. La gracia es que los campos pueden ser de tipos distintos: un struct Alumno puede tener un arreglo de char para el nombre, un int para la edad, otro int para la matricula y un float para el promedio. Todos esos datos viajan juntos, se copian juntos y se pasan a funciones juntos.

La sintaxis para definir un struct es asi:

```
struct Alumno {  
    char nombre[50];  
    int edad;  
    int matricula;  
    float promedio;  
};
```

Fíjate en tres detalles. Primero: la palabra reservada struct seguida del nombre que tu eliges para tu nuevo tipo. Por convención se escribe con mayúscula inicial, como Alumno, Producto o Fecha, para distinguirlo de las variables. Segundo: entre llaves van los campos, cada uno con su tipo y su nombre, terminados en punto y coma como cualquier declaración. Tercero y el que más se olvida: el punto y coma después de la llave de cierre. Sin ese punto y coma el compilador te va a marcar error y te vas a tardar un rato en encontrarlo, así que grabatelo desde hoy: todo struct termina en };

Es importante entender que definir un struct NO crea ninguna variable todavía. Solo le estás diciendo al compilador: "mira, existe un molde llamado Alumno que tiene estos campos". Es como diseñar el formato de la ficha de inscripción en papel: el formato solo no guarda a ningún alumno, pero ya sabes exactamente que datos va a llevar cada ficha que imprimas después.

Declarar variables struct y acceder con el punto

Una vez definido el molde, declaras variables de ese tipo así:

```
struct Alumno a1;  
struct Alumno a2;
```

Cada variable es una ficha completa e independiente: a1 tiene su propio nombre, su propia edad, su propio promedio, y a2 tiene los suyos. Para entrar a un campo específico usas el operador punto, que se lee como "el campo tal de la variable tal":

```
strcpy(a1.nombre, "Maria Lopez");
a1.edad = 20;
a1.matricula = 2024001;
a1.promedio = 9.2;

printf("Alumno: %s, edad: %d, promedio: %.1f\n", a1.nombre, a1.edad, a1.promedio);
```

El punto es tu llave de acceso. `a1.nombre` significa "el campo nombre que vive dentro de `a1`". Sin el punto no hay forma de decirle al compilador a que pedazo del struct te refieres. Fíjate también en que para el nombre usamos `strcpy` y no una asignación directa: recuerda que nombre es un arreglo de char, y los arreglos no se pueden asignar con `=` en C. Ese detalle de la unidad de cadenas sigue aplicando aquí adentro.

Puedes declarar la variable y definir el struct al mismo tiempo si quieres, aunque lo común es definir el struct una vez (normalmente arriba del programa o en un archivo de encabezado) y declarar todas las variables que necesites después.

Inicialización de structs

Hay varias formas de darle valores iniciales a un struct. La más directa es asignar campo por campo, como viste arriba. Pero también puedes inicializarlo todo de un jalon al declararlo, poniendo los valores entre llaves y en el mismo orden en que definiste los campos:

```
struct Alumno a3 = {"Carlos Ruiz", 19, 2024002, 8.7};
```

El compilador reparte los valores en orden: "Carlos Ruiz" va a nombre, 19 va a edad, 2024002 va a matricula y 8.7 va a promedio. Es compacto y legible, perfecto cuando ya conoces los datos de antemano. Si das menos valores de los que tiene el struct, los campos restantes se inicializan en cero automáticamente.

Desde el estándar C99 también puedes usar inicializadores designados, donde nombras cada campo explícitamente y el orden ya no importa:

```
struct Alumno a4 = {.promedio = 9.8, .nombre = "Ana Torres", .edad = 21, .matricula = 2024003};
```

Esta forma es más clara cuando el struct tiene muchos campos, porque cualquiera que lea el código sabe exactamente que valor va a cada campo sin tener que contar posiciones. Cualquiera de las tres formas es válida; elige la que haga tu código más fácil de leer.

Arreglos de structs: tu archivo completo

Aquí es donde el struct empieza a pagar su precio. Un solo struct es una ficha; un arreglo de structs es el archivo completo con todas las fichas ordenadas. Declararlo es igual que declarar un arreglo de cualquier otro tipo:

```
struct Alumno grupo[30];
```

Ahora tienes 30 fichas de alumno, de grupo[0] a grupo[29]. Para llegar a un campo de una ficha específica combinas el índice del arreglo con el punto:

```
strcpy(grupo[0].nombre, "Maria Lopez");
grupo[0].edad = 20;
grupo[0].promedio = 9.2;

strcpy(grupo[1].nombre, "Carlos Ruiz");
grupo[1].edad = 19;
grupo[1].promedio = 8.7;
```

Se lee de izquierda a derecha: grupo[0] es la primera ficha, y .nombre es el campo nombre de esa ficha. Con un ciclo for puedes recorrer todo el archivo: pedir los datos de cada alumno, imprimir la lista completa, buscar al de mejor promedio o contar cuántos reprobaron. Todo lo que aprendiste de arreglos en la unidad 4 se aplica igualito, solo que ahora cada casilla del arreglo es una ficha completa en lugar de un número suelto.

```
int i;
for (i = 0; i < 30; i++) {
    printf("Alumno %d: %s, promedio %.1f\n", i + 1, grupo[i].nombre,
    grupo[i].promedio);
}
```

Piensa en lo horrible que sería hacer esto mismo sin structs: necesitarías cuatro arreglos paralelos (nombres[30], edades[30], matriculas[30], promedios[30]) y tendrías que mantenerlos sincronizados a mano. Si ordenas un arreglo y se te olvida ordenar los otros tres, los datos se revuelven y ya no sabes de quien es cada promedio. Con un arreglo de structs ese problema desaparece: cada ficha viaja completa a donde vaya, y ordenar el arreglo ordena las fichas enteras sin romper nada. Menos variables sueltas, menos errores tontos, código más profesional.

Structs y funciones: pasar por valor vs por apuntador

Ya sabes pasar un struct a una función igual que cualquier variable: lo pones como parámetro y listo. Pero aquí hay una decisión importante que tomar, porque hay dos formas de hacerlo y cada una tiene consecuencias.

La primera forma es pasar por valor, es decir, pasar una copia del struct:

```
void imprimirAlumno(struct Alumno a) {
    printf("Nombre: %s\n", a.nombre);
    printf("Edad: %d, Promedio: %.1f\n", a.edad, a.promedio);
}

// y se llama así:
imprimirAlumno(a1);
```

Cuando pasas por valor, la función recibe un CLON de tu ficha. Puede leerla y hasta modificarla, pero esos cambios mueren cuando la función termina: tu variable original ni se entera. Esto es seguro (la función no puede romper tus datos por accidente), pero tiene un costo: si el struct es grande, copiarlo completo cada vez que llamas a la función desperdicia tiempo y memoria. Para un struct pequeño no importa; para uno con muchos campos o arreglos grandes adentro, sí.

La segunda forma es pasar por apuntador, es decir, pasar la dirección del struct original:

```
void subirPromedio(struct Alumno *a) {
    a->promedio = a->promedio + 0.5;
}

// y se llama así:
subirPromedio(&a1);
```

Aquí la función recibe la dirección de tu ficha real, no una copia. Cualquier cambio que haga se refleja en tu variable original. Esto es rapidísimo (solo se copia una dirección, sin importar que tan grande sea el struct) y es la forma estándar de trabajar cuando la función necesita MODIFICAR el struct. La desventaja es el riesgo: si la función tiene un error, puede danar tus datos originales. Por eso, cuando pasas un apuntador solo para LEER (no para modificar), es buena práctica agregar `const` al parámetro:

```
void imprimirAlumno(const struct Alumno *a) {  
    printf("Nombre: %s\n", a->nombre);  
}
```

Con `const` le dices al compilador: "esta función promete no modificar el struct". Si por accidente intentas cambiar un campo adentro, el compilador te frena. Es como prestarle tu ficha a alguien dentro de una mica protectora: puede leerla, pero no rayarla.

Regla práctica para decidir: si la función solo va a leer el struct y el struct es pequeño, pasalo por valor, es lo más simple. Si la función necesita modificarlo, o si el struct es grande, pasalo por apuntador. Cuando dudes, el apuntador con `const` es casi siempre la mejor opción.

El operador flecha ->

Fíjate que en los ejemplos anteriores, cuando teníamos un apuntador al struct, ya no usamos el punto sino una flechita: `a->promedio`. Ese es el operador `->` y existe por una razón muy simple: comodidad. Técnicamente `a->promedio` es lo mismo que `(*a).promedio`, es decir, "sigue el apuntador hasta el struct y luego entra al campo promedio". Pero escribir `(*a).promedio` es feo, fácil de equivocar (los paréntesis son obligatorios porque el punto tiene más prioridad que el asterisco) y cansado de leer. El `->` resume todo eso en dos caracteres.

La regla de oro es fácil de memorizar:

- Si tienes el struct DIRECTO (una variable normal), usas el punto: `a1.promedio`
- Si tienes un APUNTADOR al struct, usas la flecha: `a->promedio`

```
struct Alumno a1;           // variable directa  
struct Alumno *p = &a1;     // apuntador al struct  
  
a1.promedio = 9.2;          // punto: variable directa  
p->promedio = 9.2;          // flecha: a través del apuntador
```

Vas a ver el `->` por todos lados de aquí en adelante, porque casi todo el código profesional en C pasa structs por apuntador. Cuando lo veas, lee mentalmente "el campo tal del struct apuntado por tal". Con un poco de práctica se vuelve automático.

Structs anidados: fichas dentro de fichas

Un campo de un struct puede ser, a su vez, otro struct. Esto se llama anidamiento y sirve para modelar cosas que naturalmente tienen partes. Por ejemplo, un alumno tiene una fecha de nacimiento, y una fecha tiene día, mes y año. En lugar de meter día, mes y año sueltos dentro de Alumno, defines un struct Fecha y lo metes como campo:

```
struct Fecha {
    int dia;
    int mes;
    int ano;
};

struct Alumno {
    char nombre[50];
    int edad;
    int matricula;
    float promedio;
    struct Fecha nacimiento;
};
```

Ojo con el orden: struct Fecha tiene que estar definido ANTES de struct Alumno, porque el compilador necesita saber que es una Fecha antes de meterla dentro de un Alumno. Para acceder a los campos del struct interno encadenas los puntos:

```
struct Alumno a1;
a1.nacimiento.dia = 15;
a1.nacimiento.mes = 3;
a1.nacimiento.ano = 2004;

printf("Nacio el %d/%d/%d\n", a1.nacimiento.dia, a1.nacimiento.mes,
a1.nacimiento.ano);
```

Se lee de izquierda a derecha: "de a1, su nacimiento, el día". Puedes anidar tantos niveles como necesites (una Empresa tiene un Domicilio que tiene una Colonia...), pero no te pases: si necesitas más de dos o tres niveles, probablemente tu diseño se puede simplificar.

El anidamiento también funciona con apuntadores y con inicialización entre llaves. Para inicializar un struct anidado de un jalon, anidas las llaves:

```
struct Alumno a2 = {"Luis Perez", 20, 2024005, 8.9, {22, 7, 2003}};
```

Los valores {22, 7, 2003} van a los campos de nacimiento en orden. Y si tienes un apuntador al struct externo, la flecha se combina con el punto: p->nacimiento.ano te da el año de

nacimiento del alumno apuntado por p. Punto para lo directo, flecha para lo apuntado, en cada nivel.

Union: una memoria, varios disfraces

La union es la prima extrana del struct. Se define casi igual, pero con una diferencia radical: mientras que en un struct cada campo tiene su propio espacio de memoria, en una union TODOS los campos comparten el mismo espacio. Solo uno de ellos es válido a la vez.

```
union Dato {  
    int entero;  
    float decimal;  
    char texto[20];  
};
```

Esta union ocupa en memoria lo que ocupe su campo más grande (el arreglo texto[20], o sea 20 bytes), no la suma de todos. Si guardas un entero en dato.entero y luego guardas un float en dato.decimal, el float SOBREESCRIBE la memoria donde estaba el entero. El valor viejo se pierde. Por eso se dice que la union es como un disfraz: la misma persona (la misma memoria) puede vestirse de entero, de float o de texto, pero no de los tres al mismo tiempo.

Cuando se usa esto en la vida real? Cuando un dato puede ser de varios tipos pero sabes que solo será de uno a la vez, y quieres ahorrar memoria. El ejemplo clásico es un campo de "valor" en una calculadora o en un lector de archivos: a veces el valor es un entero, a veces es un decimal. En lugar de guardar los dos siempre (desperdiciando espacio), usas una union y guardas solo el que toca.

El truco profesional es combinar la union con un struct y un campo "tipo" que te diga cuál de los disfraces está activo. Así no andas adivinando:

```

struct Valor {
    int tipo; // 0 = entero, 1 = decimal
    union {
        int entero;
        float decimal;
    } dato;
};

struct Valor v;
v.tipo = 1;
v.dato.decimal = 3.14;

if (v.tipo == 1) {
    printf("El valor es %.2f\n", v.dato.decimal);
}

```

El campo tipo funciona como la etiqueta de la caja: antes de abrirla, lees la etiqueta y sabes que hay adentro. Sin esa etiqueta, usar una union es jugar a la ruleta rusa con tus datos. Regla de oro: nunca leas de una union un campo distinto al último que escribiste, a menos que sepas EXACTAMENTE lo que estás haciendo.

Enum: constantes con nombre

Hay datos que solo pueden tomar un punado de valores fijos: los días de la semana, los meses, los estados de un pedido (pendiente, enviado, entregado), las opciones de un menu. Podrias representarlos con números sueltos (0 = lunes, 1 = martes...), pero entonces tu código se llena de números magicos que nadie entiende:

```

int dia = 2; // 2 que? martes? febrero? quien sabe

```

El enum resuelve esto dandole NOMBRE a cada valor posible:

```

enum Dia { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO };

```

Por dentro, el compilador les asigna números consecutivos empezando en 0: LUNES vale 0, MARTES vale 1, y así. Pero en tu código tú usas los nombres, que se explican solos:

```
enum Dia hoy = MIERCOLES;

if (hoy == SABADO || hoy == DOMINGO) {
    printf("Es fin de semana\n");
} else {
    printf("A darle, es dia de chamba\n");
}
```

Compara eso con `if (hoy == 5 || hoy == 6)`. El enum convierte tu código en algo que se lee casi como español. Si necesitas valores específicos en lugar de 0, 1, 2..., los puedes asignar a mano:

```
enum Estado { PENDIENTE = 1, ENVIADO = 2, ENTREGADO = 3, CANCELADO = 9 };
```

Los que no tienen valor asignado siguen la secuencia desde el último asignado. Los enums no crean variables nuevas ni gastan memoria extra: son simplemente una forma elegante de ponerle nombre a los enteros. Usalos cada vez que un dato tenga un conjunto cerrado de valores posibles. Tu yo del futuro, leyendo ese código a las 2 de la mañana, te lo va a agradecer.

Typedef: alias para tus tipos

Escribir `struct Alumno` cada vez que declaras una variable es correcto, pero es tedioso. El `typedef` te permite crear un ALIAS, un nombre corto para un tipo que ya existe:

```
typedef struct Alumno Alumno;
```

Después de esa línea, `Alumno` (a secas) significa lo mismo que `struct Alumno`. Puedes declarar variables así:

```
Alumno a1;
Alumno grupo[30];
```

Mucho más limpio, verdad? Pero el `typedef` no es solo para structs: sirve para cualquier tipo. Por ejemplo, `unsigned long long` es un tecleadero; con `typedef` lo acortas:

```
typedef unsigned long long ULL;
ULL poblacion = 128000000ULL;
```

El patrón más común en el mundo real combina la definición del struct con el typedef en una sola jugada:

```
typedef struct {  
    char nombre[50];  
    int edad;  
    int matricula;  
    float promedio;  
} Alumno;
```

Fíjate bien: el struct ya no lleva nombre entre struct y la llave (es anonimo), y el nombre Alumno va después de la llave de cierre, justo antes del punto y coma. Esa línea hace dos cosas a la vez: define el molde y crea el alias. De aquí en adelante, en todo tu programa, Alumno es un tipo como int o float. Casi todo el código C profesional que vas a leer usa este patrón, así que acostúmbrate a reconocerlo.

Un detalle importante: typedef NO crea un tipo nuevo de verdad, solo un apodo. Para el compilador, Alumno y struct Alumno son identicos e intercambiables. El beneficio es pura legibilidad: menos tecleo, código más limpio y nombres que expresan la intención. Cuando defines structs de aquí en adelante, usa este patrón siempre.

El heap vs el stack: dos almacenes de memoria

Para entender la memoria dinámica primero tienes que saber que tu programa maneja la memoria en dos zonas distintas: el stack y el heap. Ya has usado el stack todo el curso sin saberlo.

El stack (la pila) es donde viven las variables locales que declaras dentro de las funciones. Es rapidísimo y automatico: cuando entras a una función, el sistema aparta espacio para sus variables; cuando sales, ese espacio se libera solo, sin que muevas un dedo. El problema es que el stack es pequeño y rígido: el tamaño de todo lo que guardas ahí tiene que decidirse en tiempo de compilación. Por eso `int arreglo[100]` funciona (el 100 es fijo y conocido) pero `int arreglo[n]` con `n` variable es problemático en muchos compiladores: el stack no está hecho para tamaños que se deciden sobre la marcha. Además, lo que guardas en el stack muere cuando la función termina; no puedes crear algo en una función y seguir usándolo después de que retorne.

El heap (el monton) es la zona de memoria para lo dinámico. Es mucho más grande y flexible: puedes pedirle al sistema exactamente los bytes que necesitas EN TIEMPO DE EJECUCIÓN, cuando tu programa ya está corriendo y ya sabe cuántos datos van a llegar. Lo que guardas en el heap sobrevive a las funciones: sigue ahí hasta que TU lo liberes explícitamente. Esa es

su gran ventaja y su gran responsabilidad: el sistema no lo limpia por ti. Si pides memoria y no la devuelves, se queda apartada hasta que tu programa termine. A eso se le llama fuga de memoria, y es el error más famoso de C.

La analogía clásica: el stack es como tu mochila. Lo que cabe, cabe; es tuya, la cargas a todos lados y cuando llegas a casa la vacías sola. El heap es como una bodega que rentas: pides los metros cuadrados que necesites, guardas lo que quieras por el tiempo que quieras, pero pagas la renta (la memoria ocupada) y tienes que avisar cuando la desocupas. Si rentas bodegas y nunca las devuelves, tarde o temprano te quedas sin dinero, o en este caso, sin memoria.

malloc: pedir memoria prestada al sistema

La función malloc (de "memory allocation") es como llenas la solicitud de la bodega. Le dices cuántos BYTES quieres y te regresa un apuntador al primer byte de un bloque de ese tamaño en el heap. Vive en `stdlib.h`, así que necesitas incluirla.

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

int main() {
    int n;
    printf("Cuántos números vas a capturar? ");
    scanf("%d", &n);

    int *números = (int *)malloc(n * sizeof(int));

    if (números == NULL) {
        printf("No hay memoria suficiente\n");
        return 1;
    }

    // usar el arreglo...
    free(números);
    return 0;
}
```

Vamos por partes, porque cada línea tiene su razón. Primero: `n * sizeof(int)`. malloc no sabe de ints ni de structs, solo entiende bytes. Si quieres espacio para `n` enteros, tienes que pedir `n` multiplicado por el tamaño de un entero. `sizeof(int)` te da ese tamaño (normalmente 4 bytes, pero `sizeof` lo calcula por ti para que tu código funcione en cualquier máquina). Este es EL patrón que vas a repetir mil veces: `malloc(cantidad * sizeof(tipo))`.

Segundo: el `(int *)` del principio es un cast, una conversión de tipo. `malloc` regresa un apuntador generico (`void *`), que significa "apuntador a algo sin tipo definido". El cast le dice al compilador: "tratame ese bloque como un arreglo de `int`". En C moderno el cast es opcional, pero ponerlo hace el código más claro y te obliga a pensar en que tipo de datos vas a guardar ahí.

Tercero y MUY importante: verificar si regreso `NULL`. Si el sistema no tiene memoria suficiente (o pediste una cantidad absurda), `malloc` no truena tu programa: regresa `NULL`, el apuntador nulo. Si ignoras esa posibilidad y usas el apuntador como si todo estuviera bien, tu programa va a intentar escribir en la dirección cero y se va a caer de la forma más fea. La verificación con `if (números == NULL)` es obligatoria en código serio. Casi nunca va a pasar en tus programas de práctica, pero el día que pase en un sistema real, esa verificación es la diferencia entre un mensaje de error elegante y una caída misteriosa.

Y cuarto: `free(números)` al final. Todo lo que se pide se devuelve. Siempre.

Una vez que tienes el apuntador, lo usas EXACTAMENTE como un arreglo normal: `números[0]`, `números[1]`, ..., `números[n-1]`. Esa es la belleza del asunto: para el resto de tu código no hay diferencia entre un arreglo del stack y uno del heap. La diferencia está en quien decide el tamaño y cuando (tu, en tiempo de ejecución) y en quien lo libera (tu, con `free`).

free: devolver lo que pediste

La función `free` toma el apuntador que te dio `malloc` y le dice al sistema: "ya termine con este bloque, puedes reusarlo". Así de simple, y así de fácil de olvidar.

```
free(numeros);
```

Tres reglas de oro con `free`. Primera: libera EXACTAMENTE lo que `malloc` te dio. No liberes un apuntador que apunta a la mitad del bloque, ni una variable del stack, ni un apuntador que ya liberaste. `free` espera recibir el mismo valor que salió de `malloc`, `calloc` o `realloc`. Si le pasas cualquier otra cosa, el comportamiento es indefinido, que en C es la forma elegante de decir "puede pasar cualquier cosa mala".

Segunda: después de liberar, el apuntador queda colgando. Sigue guardando la dirección vieja, pero esa memoria ya no es tuya. Si por accidente la usas, estás escribiendo en memoria que el sistema ya pudo haberle dado a alguien más. La buena práctica es poner el apuntador en `NULL` inmediatamente después de liberarlo:

```
free(numeros);
numeros = NULL;
```

Así, si por error intentas usarlo después, tu programa truena de inmediato en lugar de corromper datos en silencio. Un trueno inmediato es mil veces mejor que un error silencioso que aparece tres días después.

Tercera: cada malloc necesita su free. Ni más ni menos. Si pides tres bloques, liberas tres bloques. Si pides memoria dentro de un ciclo, asegúrate de liberarla dentro del ciclo o tendrás una fuga multiplicada por cada vuelta.

calloc y realloc: los otros dos de la familia

malloc tiene dos hermanos que completan la familia de la memoria dinámica.

calloc (de "clear allocation") es como malloc, pero con dos diferencias: recibe la cantidad y el tamaño POR SEPARADO, e inicializa toda la memoria en ceros.

```
int *numeros = (int *)calloc(n, sizeof(int));
```

Fíjate: calloc(n, sizeof(int)) en lugar de malloc(n * sizeof(int)). El resultado es el mismo bloque, pero con todos los bytes en cero. Eso significa que cada entero del arreglo empieza valiendo 0, cada apuntador empieza en NULL y cada char en '\0'. Cuando necesitas un arreglo que empiece limpio (contadores, tablas de frecuencias, buffers), calloc te ahorra el ciclo de inicialización. Cuesta un poquito más de tiempo que malloc por la limpieza, pero la seguridad lo vale.

realloc (de "re-allocation") es la más interesante: te permite CAMBIAR EL TAMAÑO de un bloque que ya pediste. Le pasas el apuntador viejo y el nuevo tamaño en bytes, y te regresa un apuntador al bloque redimensionado, con los datos viejos conservados.

```
int *tmp = (int *)realloc(numeros, nuevo_n * sizeof(int));
if (tmp == NULL) {
    printf("No se pudo agrandar el bloque\n");
    // numeros SIGUE valido: no lo pierdas
} else {
    numeros = tmp;
}
```

Este ejemplo merece atención porque aquí vive uno de los errores más sutiles de C. Fíjate que NO hacemos `numeros = realloc(numeros, ...)` directamente. ¿Por qué? Porque si realloc falla y

regresa NULL, habrias SOBRESERITO tu apuntador original con NULL y perdido para siempre la direcci3n del bloque viejo: una fuga de memoria instantanea, y adem3s ya no podrias liberar lo que si tenias. El patr3n correcto es guardar el resultado en un apuntador temporal, verificar que no sea NULL, y solo entonces actualizar tu apuntador.

Qu3 hace realloc por dentro? Si hay espacio libre justo despu3s de tu bloque, lo agranda ah3 mismo y te regresa el mismo apuntador. Si no hay espacio, pide un bloque nuevo m3s grande en otro lado, COPIA tus datos viejos, libera el bloque viejo y te regresa el apuntador nuevo. Tu no tienes que preocuparte por cu3l de las dos cosas hizo: tus datos se conservan en ambos casos. Tambi3n puedes usar realloc para ACHICAR un bloque, y si le pasas NULL como apuntador, se comporta como malloc. Es la navaja suiza de la memoria din3mica.

El uso estrella de realloc es el arreglo que crece solo: empiezas pidiendo espacio para 5 elementos, y cada vez que se llena, lo duplicas con realloc. Asi tu programa maneja 3 datos o 3 millones sin que tu tengas que adivinar el tama3o por adelantado. Este patr3n aparece en los ejemplos paso a paso y es una de las t3cnicas m3s valiosas que te vas a llevar de esta unidad.

Structs en el heap: malloc de structs y liberaci3n en orden

Ya sabes pedir arreglos dinamicos de ints y de floats. Pero la combinaci3n m3s poderosa es pedir STRUCTS completos en el heap: malloc(n * sizeof(Alumno)) te da un archivo de fichas del tama3o exacto, como viste en el ejemplo 8. Hay un caso m3s que tienes que dominar: un struct INDIVIDUAL pedido con malloc, y structs que tienen apuntadores ADENTRO.

Pedir un solo struct din3mico es com3n cuando una funci3n necesita crear una ficha y regresarla:


```

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

typedef struct {
    char nombre[50];
    int edad;
    float promedio;
} Alumno;

Alumno *crearAlumno(const char *nombre, int edad, float promedio) {
    Alumno *a = (Alumno *)malloc(sizeof(Alumno));
    if (a == NULL) {
        return NULL;
    }
    strcpy(a->nombre, nombre);
    a->edad = edad;
    a->promedio = promedio;
    return a;
}

int main() {
    Alumno *a1 = crearAlumno("Sofia Vega", 21, 9.4);
    if (a1 == NULL) {
        printf("No hay memoria\n");
        return 1;
    }

    printf("%s, %d anos, %.1f\n", a1->nombre, a1->edad, a1->promedio);

    free(a1);
    a1 = NULL;
    return 0;
}

```

Fíjate en el patrón: `malloc(sizeof(Alumno))` pide exactamente una ficha (sin multiplicar por n, porque es una sola). La función regresa el apuntador y quien la llama se vuelve responsable de liberarlo. Y adentro de la función usamos la flecha (`a->nombre`) porque `a` es un apuntador. Este patrón de "función que crea y regresa" es la base de las estructuras de datos avanzadas: listas ligadas, árboles y grafos se construyen creando nodos con `malloc` uno por uno.

Ahora el caso delicado: un struct con un apuntador adentro. Imagina que el nombre no es un arreglo fijo de 50 sino un apuntador, para no desperdiciar espacio con nombres cortos:

```
typedef struct {
    char *nombre;
    int edad;
    float promedio;
} AlumnoFlexible;
```

Aquí hay DOS reservas de memoria: una para el struct (`malloc(sizeof(...))`) y otra para el nombre (`malloc(strlen(nombre) + 1)`). Y la liberación tiene que hacerse EN ORDEN INVERSO: primero liberas lo de adentro (el nombre), y luego el contenedor (el struct). Si liberas el struct primero, pierdes el apuntador al nombre y esa memoria queda fugada para siempre:

```
// ORDEN CORRECTO:
free(a->nombre); // primero lo de adentro
a->nombre = NULL;
free(a);          // luego el contenedor
a = NULL;
```

Piénsalo como una caja con cosas adentro: primero sacas las cosas, luego desechas la caja. Si tiras la caja con las cosas adentro, las cosas se pierden. Cada vez que disenes un struct con apuntadores internos, dibuja mentalmente este orden de liberación. Es el error número 6 de la lista, y ahora ya sabes exactamente por qué pasa y como evitarlo.

Fugas de memoria y el doble free: los dos pecados capitales

Ya los mencione, pero merecen su propia sección porque son los errores que definen la reputación de C.

La FUGA DE MEMORIA (memory leak) pasa cuando pides memoria con `malloc` y nunca la liberas. Cada fuga es poca cosa: unos bytes aquí, un struct allá. El problema es que se acumulan. Un programa que corre unos segundos ni lo nota. Un servidor que corre meses, un videojuego que crea enemigos sin parar o una app que procesa fotos en ciclo, se va quedando sin memoria gota a gota hasta que el sistema se arrastra o el programa muere. Es como dejar la llave del agua goteando: una gota no es nada, un mes de goteo inunda la casa.

Las fugas más comunes tienen tres orígenes. Uno: olvidar el `free` porque el código tiene varios caminos de salida (returns tempranos) y en alguno se te paso. Dos: perder el apuntador, por ejemplo reasignandolo antes de liberar: `p = malloc(...); p = otra_cosa;` <- el primer bloque ya es inalcanzable, nadie lo va a liberar jamás. Tres: el error de `realloc` que vimos arriba, sobrescribir el apuntador con `NULL` cuando falla.

El DOBLE FREE pasa cuando liberas DOS VECES el mismo bloque. La primera vez todo bien; la segunda, le estás diciendo al sistema "libera esto" sobre memoria que ya esta libre y posiblemente reasignada a otra cosa. El resultado es corrupción de la memoria interna del administrador del heap, y los síntomas son de lo más traicionero: el programa puede seguir funcionando un rato y tronar mucho después, en un lugar que no tiene nada que ver con el error real. Por eso la práctica de poner el apuntador en NULL después de liberarlo es tan valiosa: liberar NULL es seguro (`free(NULL)` no hace nada), así que si por accidente llamas `free` dos veces sobre un apuntador que pusiste en NULL, la segunda llamada es inofensiva.

Errores comunes y buenas prácticas

Cierro la explicación con la lista de los tropiezos clásicos. Todos los programadores de C se pegan con al menos uno; la diferencia entre el novato y el profesional es que el profesional ya sabe que existen y los huele a distancia.

Error 1: usar memoria después de liberarla. Liberas el bloque y luego, unas líneas más abajo, lees o escribes ahí. La memoria ya no es tuya; puede tener basura o pertenecerle a otro bloque. Solución: pon el apuntador en NULL justo después del `free` y disena tu código para no necesitar el bloque después de liberarlo.

Error 2: no verificar el resultado de `malloc`. Ya lo vimos: si `malloc` regresa NULL y no lo checas, el trueno es seguro. Verifica SIEMPRE. Es una línea de código que te salva de horas de depuración.

Error 3: olvidar el `free` en algun camino del código. Si tu función tiene tres `return` y solo dos liberan la memoria, tienes una fuga en el tercero. Solución: centraliza la liberación o revisa cada salida. Con la práctica desarrollas el reflejo de escribir el `free` casi al mismo tiempo que el `malloc`.

Error 4: pasarte del tamaño pedido. Pediste espacio para 10 enteros y escribes en la posición 10 (qué es la número 11, recuerda que empiezan en 0). Esa escritura cae fuera de tu bloque y corrompe lo que este al lado. C no te avisa: no hay límites verificados como en otros lenguajes. Tú eres el guardian de tus propios índices.

Error 5: mezclar `malloc` con `delete` o `new`. Eso es de C++, no de C. En C es `malloc/free`, punto. Y nunca liberes con `free` algo que no salió de `malloc/calloc/realloc` (como una variable local del stack).

Error 6: en structs con apuntadores adentro, liberar solo el struct y olvidar lo de adentro. Si tu struct `Alumno` tuviera un `char *nombre` con memoria pedida por `malloc`, hacer `free` del struct NO libera el nombre: tienes que liberar primero los campos internos y luego el struct. Es la

liberación en orden inverso: primero lo de adentro, luego el contenedor. Lo verás en los ejemplos.

Buenas prácticas para llevarte a la tumba: cada malloc con su free, como abrir y cerrar paréntesis. Verifica NULL siempre. Pon en NULL después de liberar. Usa sizeof en lugar de números mágicos para los tamaños. Prefiere calloc cuando necesites memoria limpia. Usa el patrón del apuntador temporal con realloc. Y cuando algo truene de forma rara, sospecha primero de la memoria dinámica: nueve de cada diez errores misteriosos en C viven ahí.

Ejemplos paso a paso

Ejemplo 1: ficha de alumno con struct

El ejemplo más básico y el que tienes que dominar primero: definir un struct, llenar una ficha y mostrarla. Es el "hola mundo" de esta unidad.

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

struct Alumno {
    char nombre[50];
    int edad;
    int matricula;
    float promedio;
};

int main() {
    struct Alumno a1;

    strcpy(a1.nombre, "Maria Lopez");
    a1.edad = 20;
    a1.matricula = 2024001;
    a1.promedio = 9.2;

    printf("=== FICHA DEL ALUMNO ===\n");
    printf("Nombre: %s\n", a1.nombre);
    printf("Edad: %d\n", a1.edad);
    printf("Matricula: %d\n", a1.matricula);
    printf("Promedio: %.1f\n", a1.promedio);

    return 0;
}
```

Paso a paso: primero defines el molde struct Alumno con sus cuatro campos. Luego declaras a1, una ficha real. Llenas cada campo con el operador punto (y strcpy para el nombre, porque es un arreglo de char). Finalmente imprimes la ficha completa. Compilalo, correlo y luego modifica los datos: cambia el nombre, la edad, el promedio. Juega con dos variables, a1 y a2, y comprueba que son independientes: cambiar a2 no afecta a a1. Esa independencia es toda la gracia del struct.

Ejemplo 2: arreglo de productos e inventario

Ahora subimos de nivel: un arreglo de structs que modela el inventario de una tiendita. Cada producto es una ficha con código, nombre, precio y existencias.

```

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

struct Producto {
    int codigo;
    char nombre[40];
    float precio;
    int existencias;
};

int main() {
    struct Producto inventario[3];

    inventario[0].codigo = 101;
    strcpy(inventario[0].nombre, "Tortillas");
    inventario[0].precio = 18.50;
    inventario[0].existencias = 100;

    inventario[1].codigo = 102;
    strcpy(inventario[1].nombre, "Frijol");
    inventario[1].precio = 32.00;
    inventario[1].existencias = 50;

    inventario[2].codigo = 103;
    strcpy(inventario[2].nombre, "Arroz");
    inventario[2].precio = 28.75;
    inventario[2].existencias = 40;

    printf("=== INVENTARIO ===\n");
    float total = 0;
    int i;
    for (i = 0; i < 3; i++) {
        float valor = inventario[i].precio * inventario[i].existencias;
        printf("%d %s: $%.2f x %d = $%.2f\n",
            inventario[i].codigo, inventario[i].nombre,
            inventario[i].precio, inventario[i].existencias, valor);
        total += valor;
    }
    printf("Valor total del inventario: $%.2f\n", total);

    return 0;
}

```

Fíjate en el patrón `inventario[i].precio`: índice del arreglo + punto + campo. El ciclo recorre las tres fichas y calcula el valor de cada producto (precio por existencias) más el total. Este es el esqueleto de cualquier sistema de inventario real: la única diferencia con el de una tienda de verdad es la cantidad de productos y de donde vienen los datos. Como ejercicio mental, imagina agregar un campo `int vendidos` y calcular las ganancias: con structs es trivial, con arreglos paralelos sería un enredo.

Ejemplo 3: struct pasado por apuntador y el operador ->

Aquí ves la diferencia entre pasar por valor y pasar por apuntador, y el operador flecha en acción. La función `aplicarDescuento` MODIFICA el precio original, así que recibe un apuntador.

```

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

struct Producto {
    int codigo;
    char nombre[40];
    float precio;
    int existencias;
};

void mostrar(struct Producto p) {
    printf("%s: $%.2f\n", p.nombre, p.precio);
}

void aplicarDescuento(struct Producto *p, float porcentaje) {
    p->precio = p->precio * (1 - porcentaje / 100);
}

int main() {
    struct Producto prod;
    prod.codigo = 201;
    strcpy(prod.nombre, "Cafe molido");
    prod.precio = 120.00;
    prod.existencias = 25;

    printf("Antes: ");
    mostrar(prod);

    aplicarDescuento(&prod, 10);

    printf("Despues del 10%% de descuento: ");
    mostrar(prod);

    return 0;
}

```

Observa los detalles: mostrar recibe el struct POR VALOR (struct Producto p) y usa el punto, porque solo lee. aplicarDescuento recibe un APUNTAADOR (struct Producto *p) y usa la flecha, porque modifica el precio real. En main, la llamada es aplicarDescuento(&prod, 10): pasamos la dirección con &. Después de la llamada, el precio de prod cambio de 120.00 a 108.00, y mostrar lo confirma. Si aplicarDescuento hubiera recibido el struct por valor, el descuento se habría aplicado a una copia y el precio original seguiria en 120.00. Esa es la diferencia que tienes que sentir en los huesos.

Ejemplo 4: enum de días para un control de asistencia

Un enum convierte números mágicos en nombres legibles. Este programa registra en qué días hubo clase usando un enum.

```
#include <stdio.h>

enum Dia { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO };

const char *nombreDia(enum Dia d) {
    switch (d) {
        case LUNES: return "Lunes";
        case MARTES: return "Martes";
        case MIERCOLES: return "Miercoles";
        case JUEVES: return "Jueves";
        case VIERNES: return "Viernes";
        case SABADO: return "Sabado";
        case DOMINGO: return "Domingo";
        default: return "Desconocido";
    }
}

int main() {
    enum Dia hoy = VIERNES;
    printf("Hoy es %s\n", nombreDia(hoy));

    if (hoy == SABADO || hoy == DOMINGO) {
        printf("No hay clases, a descansar\n");
    } else {
        printf("Hay clases, no faltes\n");
    }

    return 0;
}
```

El enum Día le pone nombre a los valores 0 a 6. La función nombreDia traduce el enum a texto para imprimirlo (un switch con un case por cada valor del enum es el patrón clásico). En main, comparar hoy == SÁBADO se lee solo, sin comentarios que expliquen que significa el 5. Prueba cambiar hoy = VIERNES por hoy = DOMINGO y corre de nuevo: el mensaje cambia. Así de fácil es trabajar con enums.

Ejemplo 5: typedef para structs limpios

Este ejemplo muestra el patrón típico: definir el struct y crear el alias en una sola línea, y como eso simplifica todo el código que sigue.

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

typedef struct {
    char nombre[50];
    int edad;
    int matricula;
    float promedio;
} Alumno;

void imprimirFicha(Alumno a) {
    printf("%s, %d anos, promedio %.1f\n", a.nombre, a.edad, a.promedio);
}

int main() {
    Alumno a1 = {"Pedro Sanchez", 22, 2024010, 8.5};
    Alumno a2 = {"Lucia Fernandez", 19, 2024011, 9.6};

    imprimirFicha(a1);
    imprimirFicha(a2);

    return 0;
}
```

Compara con los ejemplos anteriores: ya no aparece la palabra struct en todo el programa después del typedef. Alumno se usa como cualquier tipo: para declarar variables, para parámetros de funciones, para inicializar con llaves. Es menos ruido visual y el código expresa mejor la intención. De aquí en adelante, en tus programas, define tus structs con este patrón. Cuando leas código ajeno (que casi siempre usa typedef), ya sabes exactamente que está pasando.

Ejemplo 6: arreglo dinámico con malloc

El primer programa de verdad con memoria dinámica: el usuario dice cuántos números va a capturar, y el programa pide EXACTAMENTE esa memoria. Sin desperdicio, sin límite arbitrario.

```

#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, i;
    printf("Cuantos numeros vas a capturar? ");
    scanf("%d", &n);

    int *numeros = (int *)malloc(n * sizeof(int));

    if (numeros == NULL) {
        printf("Error: no hay memoria suficiente\n");
        return 1;
    }

    for (i = 0; i < n; i++) {
        printf("Dame el numero %d: ", i + 1);
        scanf("%d", &numeros[i]);
    }

    int suma = 0;
    for (i = 0; i < n; i++) {
        suma += numeros[i];
    }

    printf("Suma: %d, Promedio: %.2f\n", suma, (float)suma / n);

    free(numeros);
    numeros = NULL;

    return 0;
}

```

El corazon del programa son tres líneas: `malloc(n * sizeof(int))` pide el bloque, el `if` verifica `NULL`, y `free(números)` lo devuelve al final. Todo lo demás (los dos ciclos, la suma, el promedio) es código normal que trata a números como un arreglo cualquiera. Esa es la lección: la memoria dinámica solo cambia COMO se crea el arreglo, no cómo se usa. Correló con 3 números y luego con 1000: funciona igual, porque el tamaño se decide en tiempo de ejecución. Intenta quitar el `free` y piensa que pasaria si este programa corriera mil veces seguidas: mil bloques abandonados. Así nacen las fugas.

Ejemplo 7: lista que crece sola con realloc

El patrón más valioso de la unidad: un arreglo que se agranda automáticamente cada vez que se llena. El usuario captura números sin saber cuántos serán; cuando dice 0, termina.

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

int main() {
    int capacidad = 5;
    int total = 0;
    int *lista = (int *)malloc(capacidad * sizeof(int));

    if (lista == NULL) {
        printf("Error: no hay memoria\n");
        return 1;
    }

    printf("Captura numeros (0 para terminar):\n");
    int valor;
    while (1) {
        scanf("%d", &valor);
        if (valor == 0) break;

        if (total == capacidad) {
            capacidad *= 2;
            int *tmp = (int *)realloc(lista, capacidad * sizeof(int));
            if (tmp == NULL) {
                printf("Error: no se pudo agrandar\n");
                break;
            }
            lista = tmp;
            printf("(lista agrandada a %d espacios)\n", capacidad);
        }

        lista[total] = valor;
        total++;
    }

    printf("Capturaste %d numeros: ", total);
    int i;
    for (i = 0; i < total; i++) {
        printf("%d ", lista[i]);
    }
    printf("\n");
}
```

```

    free(lista);
    lista = NULL;

    return 0;
}

```

Sigue la lógica con calma: capacidad es el tamaño actual del bloque, total es cuántos números llevamos guardados. Cada vez que llega un número nuevo, preguntamos si `total == capacidad`, o sea, si ya no cabe. Si no cabe, duplicamos la capacidad y pedimos el bloque más grande con `realloc`, usando el patrón del apuntador temporal `tmp` para no perder el bloque viejo si falla. Los datos anteriores se conservan solos. Correló y captura más de 5 números para ver el mensaje de agrandamiento en acción. Este patrón (empezar chico y duplicar cuando se llena) es exactamente cómo funcionan por dentro las listas dinámicas de otros lenguajes. Ahora sabes el truco.

Ejemplo 8: programa completo que combina struct + malloc

El examen final de la unidad: un registro de alumnos donde el NÚMERO de alumnos se decide en tiempo de ejecución. Structs para las fichas, malloc para el archivo completo. Todo lo aprendido, junto.

```

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

typedef struct {
    char nombre[50];
    int matricula;
    float promedio;
} Alumno;

int main() {
    int n, i;
    printf("Cuántos alumnos vas a registrar? ");
    scanf("%d", &n);

    Alumno *grupo = (Alumno *)malloc(n * sizeof(Alumno));

    if (grupo == NULL) {
        printf("Error: no hay memoria suficiente\n");
        return 1;
    }

    for (i = 0; i < n; i++) {

```

```

        printf("\n--- Alumno %d ---\n", i + 1);
        printf("Nombre: ");
        scanf(" %[^\\n]", grupo[i].nombre);
        printf("Matricula: ");
        scanf("%d", &grupo[i].matricula);
        printf("Promedio: ");
        scanf("%f", &grupo[i].promedio);
    }

    printf("\n=== LISTA DE ALUMNOS ===\n");
    float suma = 0;
    int mejor = 0;
    for (i = 0; i < n; i++) {
        printf("%s (mat. %d): %.1f\n",
               grupo[i].nombre, grupo[i].matricula, grupo[i].promedio);
        suma += grupo[i].promedio;
        if (grupo[i].promedio > grupo[mejor].promedio) {
            mejor = i;
        }
    }

    printf("\nPromedio del grupo: %.2f\n", suma / n);
    printf("Mejor promedio: %s con %.1f\n", grupo[mejor].nombre,
    grupo[mejor].promedio);

    free(grupo);
    grupo = NULL;

    return 0;
}

```

Analízalo por capas. La capa struct: el typedef define la ficha Alumno con nombre, matrícula y promedio. La capa dinámica: `malloc(n * sizeof(Alumno))` pide espacio para exactamente `n` fichas, ni una más. La capa lógica: dos ciclos, uno para capturar y otro para mostrar, calcular el promedio del grupo y encontrar el mejor. Fíjate en el truco de `scanf(" %[^\\n]", ...)`: el espacio antes del `%` consume el Enter que quedó del `scanf` anterior, y `%[^\\n]` lee el nombre completo con espacios. Y al final, como siempre, `free` y `NULL`.

Este programa es, en miniatura, un sistema de verdad: modela entidades del mundo real con structs y maneja una cantidad desconocida de ellas con memoria dinámica. Si entiendes cada línea de este ejemplo, entiendes la unidad completa. Modifícalo: agrega el campo edad, agrega la búsqueda de un alumno por matrícula, ordena la lista por promedio. Cada modificación te va a enseñar algo nuevo.

Práctica

1. Define un struct Libro con los campos título (50 caracteres), autor (50 caracteres), ano (int) y precio (float). En el main crea tres libros con los datos que quieras, imprimelos en formato de ficha y al final muestra cuál es el más caro.
2. Usando typedef, define el tipo Punto con los campos x y y (ambos float). Escribe la función distancia que reciba dos puntos POR APUNTAADOR (con const, porque solo los lee) y regrese la distancia entre ellos. Pruebala en el main con dos puntos capturados por teclado. Pista: la distancia es la raíz cuadrada de $(x_2-x_1)^2 + (y_2-y_1)^2$ al cuadrado; necesitas math.h y compilar con -lm.
3. Define un enum Mes con los doce meses del año (ENERO a DICIEMBRE). Escribe un programa que pida al usuario un número del 1 al 12 e imprima el nombre del mes y cuántos días tiene (febrero: 28). Si el número no está entre 1 y 12, muestra un mensaje de error. Usa un switch.
4. Escribe un programa que pregunte cuántas calificaciones se van a capturar, pida la memoria exacta con malloc, lea las calificaciones, y al final muestre la más alta, la más baja y el promedio. Verifica que malloc no regrese NULL y libera la memoria al final.
5. Modifica el programa anterior para que NO pregunte cuántas calificaciones serán: el usuario captura calificaciones una por una y escribe -1 para terminar. Usa malloc para empezar con espacio para 4 y realloc para duplicar la capacidad cada vez que se llene. Al final muestra el promedio y libera la memoria.
6. Programa completo: define con typedef el struct Empleado con nombre (50 caracteres), puesto (30 caracteres) y salario (float). Pide al usuario cuántos empleados va a registrar, reserva el arreglo con malloc, captura los datos, y muestra: la lista completa, el salario promedio y los datos del empleado mejor pagado. Libera la memoria antes de terminar.

Solución del ejercicio 1

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

struct Libro {
    char titulo[50];
    char autor[50];
    int ano;
    float precio;
};

void imprimirLibro(struct Libro l) {
```

```

    printf("Titulo: %s\n", l.titulo);
    printf("Autor: %s\n", l.autor);
    printf("Año: %d, Precio: $%.2f\n", l.ano, l.precio);
    printf("-----\n");
}

int main() {
    struct Libro libros[3];

    strcpy(libros[0].titulo, "Cien años de soledad");
    strcpy(libros[0].autor, "Gabriel García Márquez");
    libros[0].ano = 1967;
    libros[0].precio = 299.00;

    strcpy(libros[1].titulo, "Don Quijote de la Mancha");
    strcpy(libros[1].autor, "Miguel de Cervantes");
    libros[1].ano = 1605;
    libros[1].precio = 349.00;

    strcpy(libros[2].titulo, "La sombra del viento");
    strcpy(libros[2].autor, "Carlos Ruiz Zafón");
    libros[2].ano = 2001;
    libros[2].precio = 279.00;

    int i, caro = 0;
    for (i = 0; i < 3; i++) {
        imprimirLibro(libros[i]);
        if (libros[i].precio > libros[caro].precio) {
            caro = i;
        }
    }

    printf("El libro mas caro es: %s ($%.2f)\n",
           libros[caro].titulo, libros[caro].precio);

    return 0;
}

```

Explicación: el struct Libro agrupa los cuatro datos de cada libro en una ficha. El arreglo libros[3] es el catalogo. La función imprimirLibro recibe el struct por valor porque solo lo lee. Para encontrar el más caro usamos la técnica del "mejor hasta ahora": la variable caro guarda el ÍNDICE del libro más caro encontrado, empezando en 0, y cada vez que encontramos un precio mayor actualizamos el índice. Al final, libros[caro] es el ganador. Esta técnica de guardar el índice del mejor te va a servir en cientos de programas.

Solución del ejercicio 2

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

typedef struct {
    float x;
    float y;
} Punto;

float distancia(const Punto *p1, const Punto *p2) {
    float dx = p2->x - p1->x;
    float dy = p2->y - p1->y;
    return sqrt(dx * dx + dy * dy);
}

int main() {
    Punto a, b;

    printf("Punto A (x y): ");
    scanf("%f %f", &a.x, &a.y);
    printf("Punto B (x y): ");
    scanf("%f %f", &b.x, &b.y);

    printf("Distancia: %.2f\n", distancia(&a, &b));

    return 0;
}
```

Explicación: el typedef crea el tipo Punto limpio, sin struct por todos lados. La función distancia recibe APUNTADORES constantes: const Punto *p1 significa "puedo leer el punto, pero no modificarlo". Adentro usamos la flecha (p2->x) porque trabajamos a través de apuntadores. La fórmula es la distancia euclidiana: raíz cuadrada de la suma de los cuadrados de las diferencias. sqrt vive en math.h, y al compilar necesitas agregar -lm al final: gcc programa.c -o programa -lm. En el main pasamos las direcciones con &a y &b. Fíjate como este ejemplo junta tres temas de la unidad: typedef, paso por apuntador con const y el operador ->.

Solución del ejercicio 3

```

#include <stdio.h>

enum Mes { ENERO = 1, FEBRERO, MARZO, ABRIL, MAYO, JUNIO,
           JULIO, AGOSTO, SEPTIEMBRE, OCTUBRE, NOVIEMBRE, DICIEMBRE };

int main() {
    int numero;
    printf("Dame un numero de mes (1-12): ");
    scanf("%d", &numero);

    switch (numero) {
        case ENERO: printf("Enero, 31 dias\n"); break;
        case FEBRERO: printf("Febrero, 28 dias\n"); break;
        case MARZO: printf("Marzo, 31 dias\n"); break;
        case ABRIL: printf("Abril, 30 dias\n"); break;
        case MAYO: printf("Mayo, 31 dias\n"); break;
        case JUNIO: printf("Junio, 30 dias\n"); break;
        case JULIO: printf("Julio, 31 dias\n"); break;
        case AGOSTO: printf("Agosto, 31 dias\n"); break;
        case SEPTIEMBRE: printf("Septiembre, 30 dias\n"); break;
        case OCTUBRE: printf("Octubre, 31 dias\n"); break;
        case NOVIEMBRE: printf("Noviembre, 30 dias\n"); break;
        case DICIEMBRE: printf("Diciembre, 31 dias\n"); break;
        default: printf("Error: el mes debe estar entre 1 y 12\n");
    }

    return 0;
}

```

Explicación: el truco está en ENERO = 1. Al asignarle 1 al primer valor, los demás siguen la secuencia solos: FEBRERO vale 2, MARZO 3, hasta DICIEMBRE que vale 12. Así los valores del enum coinciden con los números que el usuario escribe. El switch usa los NOMBRES del enum en los case, no los números: case ENERO se entiende solo, case 1 obligaría al lector a adivinar. El default atrapa cualquier número fuera de rango. Este es el uso más común del enum en programas reales: opciones de menú, estados y categorías con nombre.

Solución del ejercicio 4

```

#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, i;
    printf("Cuántas calificaciones vas a capturar? ");
    scanf("%d", &n);

    float *califs = (float *)malloc(n * sizeof(float));

    if (califs == NULL) {
        printf("Error: no hay memoria suficiente\n");
        return 1;
    }

    for (i = 0; i < n; i++) {
        printf("Calificacion %d: ", i + 1);
        scanf("%f", &califs[i]);
    }

    float mayor = califs[0], menor = califs[0], suma = 0;
    for (i = 0; i < n; i++) {
        if (califs[i] > mayor) mayor = califs[i];
        if (califs[i] < menor) menor = califs[i];
        suma += califs[i];
    }

    printf("Mayor: %.1f, Menor: %.1f, Promedio: %.2f\n",
        mayor, menor, suma / n);

    free(califs);
    califs = NULL;

    return 0;
}

```

Explicación: el patrón completo de malloc en cuatro pasos: pedir con malloc(n * sizeof(float)), verificar NULL, usar el bloque como un arreglo normal y liberar con free al final. Para mayor y menor usamos la técnica de inicializar con el primer elemento (califs[0]) en lugar de un valor inventado como 0 o 100: así funciona aunque todas las calificaciones sean, por ejemplo, menores que 5. El promedio es la suma entre n. Nota que el programa funciona con 5 calificaciones o con 5000 sin cambiar una sola línea: el tamaño lo decide el usuario en tiempo de ejecución. Esa flexibilidad es imposible con un arreglo fijo del stack.

Solución del ejercicio 5

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

int main() {
    int capacidad = 4;
    int total = 0;
    float *califs = (float *)malloc(capacidad * sizeof(float));

    if (califs == NULL) {
        printf("Error: no hay memoria\n");
        return 1;
    }

    printf("Captura calificaciones (-1 para terminar):\n");
    float valor;
    while (1) {
        scanf("%f", &valor);
        if (valor == -1) break;

        if (total == capacidad) {
            capacidad *= 2;
            float *tmp = (float *)realloc(califs, capacidad * sizeof(float));
            if (tmp == NULL) {
                printf("Error: no se pudo agrandar, se conservan %d datos\n", total);
                break;
            }
            califs = tmp;
        }

        califs[total] = valor;
        total++;
    }

    if (total > 0) {
        float suma = 0;
        int i;
        for (i = 0; i < total; i++) {
            suma += califs[i];
        }
        printf("Capturaste %d calificaciones. Promedio: %.2f\n", total, suma / total);
    } else {
        printf("No capturaste ninguna calificacion\n");
    }
}
```

```

    free(califs);
    califs = NULL;

    return 0;
}

```

Explicación: este es el patrón de crecimiento con realloc aplicado a un caso real. Empezamos con capacidad 4 y cada vez que el arreglo se llena (total == capacidad), duplicamos y pedimos el bloque más grande. El apuntador temporal tmp protege contra el error clásico: si realloc falla, califs sigue apuntando al bloque viejo válido y podemos terminar con dignidad en lugar de perderlo todo. Fíjate en dos detalles defensivos: verificamos total > 0 antes de dividir (evitamos división entre cero si el usuario escribe -1 de inmediato) y el free está en un solo lugar al final, por donde pasan todos los caminos del programa. Con este patrón ya no necesitas preguntar cuántos datos vienen: tu programa se adapta solo.

Solución del ejercicio 6

```

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

typedef struct {
    char nombre[50];
    char puesto[30];
    float salario;
} Empleado;

int main() {
    int n, i;
    printf("Cuantos empleados vas a registrar? ");
    scanf("%d", &n);

    Empleado *planta = (Empleado *)malloc(n * sizeof(Empleado));

    if (planta == NULL) {
        printf("Error: no hay memoria suficiente\n");
        return 1;
    }

    for (i = 0; i < n; i++) {
        printf("\n--- Empleado %d ---\n", i + 1);
        printf("Nombre: ");
        scanf(" %s", planta[i].nombre);
        printf("Puesto: ");
    }
}

```

```

        scanf(" %[^\\n]", planta[i].puesto);
        printf("Salario: ");
        scanf("%f", &planta[i].salario);
    }

    printf("\\n=== PLANTILLA ===\\n");
    float suma = 0;
    int top = 0;
    for (i = 0; i < n; i++) {
        printf("%s - %s: %.2f\\n",
            planta[i].nombre, planta[i].puesto, planta[i].salario);
        suma += planta[i].salario;
        if (planta[i].salario > planta[top].salario) {
            top = i;
        }
    }

    printf("\\nSalario promedio: %.2f\\n", suma / n);
    printf("Mejor pagado: %s (%s) con %.2f\\n",
        planta[top].nombre, planta[top].puesto, planta[top].salario);

    free(planta);
    planta = NULL;

    return 0;
}

```

Explicación: el ejercicio que junta toda la unidad. El typedef define la ficha Empleado; malloc(n * sizeof(Empleado)) crea el archivo dinámico del tamaño exacto; los ciclos capturan y procesan; la técnica del índice del mejor (top) encuentra al mejor pagado; y el free final devuelve la memoria. El scanf(" %[^\\n]", ...) lee nombres y puestos con espacios, y el espacio antes del % limpia el Enter pendiente del scanf anterior. Si llegaste hasta aquí y este programa te parece claro de principio a fin, felicidades: ya piensas como programador de C. Ya no ves variables sueltas, ves entidades; ya no ves límites fijos, ves memoria que se adapta.

Resumen de la unidad

Llegaste al final de la unidad más transformadora del curso. Hagamos el recuento de las armas nuevas que agregaste a tu arsenal.

El struct es tu molde para crear tipos de datos a la medida: agrupa campos de tipos distintos bajo un solo nombre y te deja modelar el mundo real (alumnos, productos, empleados, fechas) dentro de tu programa. Lo defines con struct y llaves, declaras variables de ese tipo, accedes a

los campos con el punto y lo inicializas con llaves en orden o con inicializadores designados. Un arreglo de structs es tu archivo de fichas: reemplaza a los arreglos paralelos y elimina de un golpe toda una familia de errores de sincronización.

Cuando pasas structs a funciones tienes dos caminos: por valor (una copia segura pero costosa, con el punto adentro) y por apuntador (rápido y modificable, con la flecha -> adentro). Si solo vas a leer, agrega const al apuntador y duerme tranquilo. Los structs se pueden anidar para modelar cosas con partes, encadenando los puntos nivel por nivel.

La union es el struct con memoria compartida: todos sus campos usan el mismo espacio y solo uno es válido a la vez; usala con un campo etiqueta que diga cuál está activo. El enum le pone nombre a los enteros de un conjunto cerrado de valores (días, meses, estados) y vuelve tu código legible sin números magicos. El typedef crea alias para tus tipos, y el patrón typedef struct { ... } Nombre es el estándar para definir structs limpios en código profesional.

La memoria dinámica vive en el heap: la pides en tiempo de ejecución con malloc(cantidad * sizeof(tipo)), SIEMPRE verificas que no regrese NULL, la usas como un arreglo normal y la devuelves con free. Después de liberar, pon el apuntador en NULL. calloc es como malloc pero con la memoria inicializada en ceros; realloc cambia el tamaño de un bloque conservando los datos, y con el patrón del apuntador temporal y la duplicación de capacidad construyes arreglos que crecen solos.

Y los pecados capitales, grabados a fuego: la fuga de memoria (pedir y no devolver), el doble free (liberar dos veces lo mismo), usar memoria ya liberada y pasarte de los límites del bloque. Cada malloc quiere su free, como cada paréntesis que abres quiere su cierre.

Con structs y memoria dinámica ya puedes construir sistemas de verdad: inventarios, plantillas, catalogos y listas que crecen sin límite prefijado. Solo te quedan dos unidades: archivos (para que tus datos sobrevivan al apagar la computadora) y el proyecto final. Estás a un paso de cerrar el curso completo. Repasa esta unidad las veces que necesites, porque todo lo que sigue se apoya en ella, y cuando quieras volver a cualquier tema, ya sabes dónde encontrarlo: en lucasegaacademy.com, gratis, como siempre. A darle con la que sigue.

UNIDAD 10: Archivos, buenas prácticas y proyecto final

Introducción

Hay un momento en la vida de todo programador en que su programa hace algo maravilloso y luego se apaga la computadora y todo se pierde. Capturaste cien contactos en tu agenda, el programa funcionó perfecto, los viste en pantalla, todo bien, y después cerraste la terminal y adios: los cien contactos desaparecieron como si nunca hubieran existido. Eso no es un programa de verdad, es un juguete. Y no es tu culpa: hasta ahora todo lo que guardaste vivió en la memoria RAM, y la RAM es como una pizarra que alguien borra cada vez que apagas la máquina. Los datos que importan tienen que sobrevivir al apagón. Un programa profesional no solo procesa información: la conserva. Un banco no puede perder tus movimientos cuando se va la luz, un hospital no puede perder tu historial médico porque alguien reinicie el servidor, y tu agenda no puede olvidar a tus contactos porque cerraste la ventana. En esta unidad vas a cruzar esa frontera: vas a aprender a leer y escribir archivos en C, que es como el lenguaje guarda información en el disco duro para que siga ahí mañana, la próxima semana y el próximo año.

Pero los archivos son solo la mitad de esta unidad. La otra mitad es algo que separa a los estudiantes de los profesionales: la disciplina del código bien hecho. Hasta la clase 27 aprendiste a hacer que las cosas funcionen, y eso está muy bien porque es lo primero. Pero en el mundo real, funcionar no basta. El código que escribes hoy lo vas a leer tú mismo dentro de seis meses y no lo vas a entender si está mal escrito. El código que escribes en un trabajo lo van a leer tus compañeros, y si está lleno de trucos raros y variables con nombres como `x`, `xx` y `xxx2`, nadie lo va a querer tocar. Y peor aun: un programa que compila con veinte advertencias es un programa que tiene veinte lugares donde puede romperse en el peor momento. En esta unidad vas a aprender a compilar como un profesional con las banderas `-Wall` y `-Wextra`, que le dicen al compilador: "dime TODO lo que este mal, aunque sea pequeño". Vas a aprender a leer los mensajes del compilador como lo que son: el mejor maestro gratis que existe. Vas a aprender buenas prácticas de verdad, de las que se usan en empresas: nombres claros, funciones cortas, comentarios que explican el porqué y no el que, y pruebas de casos borde, esos datos raros y extremos que rompen los programas descuidados. Y vas a aprender a depurar, que es el arte de encontrar errores como un detective: con paciencia, con método y con `printf` estratégicos en los lugares correctos.

Y al final de esta unidad viene la corona de todo el curso: el proyecto final. Vas a construir un sistema completo en C, con menu interactivo, estructuras de datos, memoria dinámica y guardado en archivo: una agenda de contactos o un sistema de inventario, tu eliges, con todo lo que aprendiste en las 30 clases. No va a ser un ejercicio de juguete de veinte líneas: va a ser un programa de verdad, de esos que puedes presumir, que puedes poner en tu portafolio y que puedes mostrarle a alguien diciendo "esto lo hice yo, en C, desde cero". Porque eso es lo que este curso prometió desde la clase 1: convertirte en alguien que programa en C de verdad. Y cuando termines, no te voy a dejar colgado: te voy a decir exactamente que sigue después de C, cuál es el camino natural hacia C++, estructuras de datos, sistemas operativos y sistemas embebidos, y como lucasegaacademy.com te acompaña en cada paso.

Así que respira hondo. Esta es la última unidad, pero también es la más práctica de todas. Lo que aprendas aquí es lo que vas a usar todos los días como programador. Vamos con todo.

Explicación detallada

Archivos en C: tus datos sobreviven al apagón

Todo lo que has guardado hasta ahora en variables, arreglos y estructuras vivió en la memoria RAM. La RAM es rápida pero volátil: cuando el programa termina o la computadora se apaga, todo se borra. Para que los datos persistan necesitas escribirlos en un almacenamiento permanente: el disco duro o la memoria SSD. En C, la herramienta para eso son los archivos, y el lenguaje te da una interfaz limpia y poderosa para manejarlos.

El concepto central es el tipo `FILE`, que se define en la biblioteca `stdio.h` (la misma de `printf` y `scanf`, así que ya la conoces). `FILE` no es un tipo de dato normal: es una estructura opaca que representa un archivo abierto. Tu nunca manipulas su contenido directamente; solo guardas un apuntador a `FILE`, es decir, un `FILE *`. Piensalo así: el `FILE *` es como el control remoto del archivo. No necesitas saber cómo funciona el televisor por dentro; con el control lo prendes, cambias de canal y lo apagas. Igual con el archivo: con el apuntador `FILE *` lo abres, lees, escribes y lo cierras.

El ciclo de vida de un archivo en C tiene cuatro pasos, y te los tienes que aprender de memoria porque los vas a usar miles de veces en tu carrera:

Paso 1: abrir el archivo con `fopen`. La función `fopen` recibe dos cosas: el nombre del archivo y el modo de apertura, y te devuelve un `FILE *`. Si algo sale mal (por ejemplo, intentas leer un archivo que no existe), `fopen` devuelve `NULL`, y tienes que verificarlo. Siempre. Sin excepciones. Más adelante veremos por que no verificarlo es uno de los errores más comunes y más tontos.

Paso 2: leer o escribir usando funciones como `fprintf`, `fscanf`, `fgets`, `fputs`, `fread` o `fwrite`.

Paso 3: cerrar el archivo con `fclose`. Este paso es OBLIGATORIO. Cerrar el archivo libera los recursos del sistema y, muy importante, asegura que todo lo que escribiste realmente llegue al disco. C tiene un bufer de escritura: cuando escribes con `fprintf`, los datos primero van a un bufer en memoria y solo se vacían al disco cuando el bufer se llena o cuando cierras el archivo. Si tu programa termina sin llamar a `fclose`, puedes perder datos. Es cómo escribir una carta y no meterla al sobre: el trabajo se hizo a medias.

Paso 4 (implícito): verificar errores en cada paso. Un programa profesional nunca asume que el archivo se abrió bien.

Los modos de apertura de `fopen` son la primera decisión importante. Los básicos para archivos de texto son:

- "r" (read, lectura): abre un archivo existente para leerlo. Si el archivo no existe, `fopen` devuelve `NULL`. El cursor de lectura empieza al principio del archivo. Es el modo más seguro porque no modifica nada.
- "w" (write, escritura): crea un archivo nuevo para escribir, o si ya existe, BORRA todo su contenido y empieza desde cero. Cuidado con este modo: es destructivo. Si abres en "w" un archivo que tenía información valiosa, esa información se pierde para siempre. Usalo cuando quieres crear un archivo nuevo o reemplazar por completo el contenido.
- "a" (append, agregar): abre el archivo para agregar datos al final. Si el archivo no existe, lo crea. Si existe, conserva su contenido y todo lo que escribas se pega al final. Es el modo ideal para registros, bitácoras y logs: cada ejecución del programa agrega líneas nuevas sin borrar las anteriores.

Hay más modos ("r+", "w+", "a+", "rb", "wb"...), donde el signo + permite leer y escribir, y la b indica modo binario (para archivos que no son texto, como imágenes). Para esta unidad nos concentramos en r, w y a con archivos de texto, que es lo que más vas a usar, y mencionaremos el modo binario cuando guardemos estructuras completas.

Veamos el patrón básico de escritura, que tienes que memorizar:

```
#include <stdio.h>

int main() {
    FILE *archivo = fopen("saludo.txt", "w");
    if (archivo == NULL) {
        printf("Error: no se pudo abrir el archivo.\n");
        return 1;
    }
    fprintf(archivo, "Hola, mundo desde un archivo.\n");
    fprintf(archivo, "Segunda línea.\n");
    fclose(archivo);
    printf("Archivo escrito correctamente.\n");
    return 0;
}
```

Observa tres cosas. Primera: siempre verificamos si `archivo == NULL` después de `fopen`. Segunda: `fprintf` funciona EXACTAMENTE igual que `printf`, solo que el primer argumento es el apuntador al archivo en lugar de la pantalla. Todo lo que sabes de `printf` (formatos `%d`, `%s`, `%f`) aplica igual a `fprintf`. Tercera: cerramos con `fclose` al final. Este esqueleto (abrir, verificar, usar, cerrar) lo vas a repetir en cada programa con archivos que escribas en tu vida.

Para leer, el patrón es simétrico:

```
#include <stdio.h>

int main() {
    FILE *archivo = fopen("saludo.txt", "r");
    if (archivo == NULL) {
        printf("Error: no se pudo abrir el archivo.\n");
        return 1;
    }
    char linea[256];
    while (fgets(linea, sizeof(linea), archivo) != NULL) {
        printf("%s", linea);
    }
    fclose(archivo);
    return 0;
}
```

Aquí aparece `fgets`, que lee una línea completa del archivo (incluyendo espacios, a diferencia de `scanf` que se detiene en el primer espacio). `fgets` recibe el bufer donde guardar, el tamaño máximo y el archivo. Devuelve `NULL` cuando llega al final del archivo, por eso el `while` funciona perfecto para leer línea por línea hasta terminar. Este patrón de "`while fgets != NULL`" es uno de los más usados en C para procesar archivos de texto.

Su compañera para escribir líneas es `fputs`, que escribe una cadena en el archivo. Y para lectura con formato existe `fscanf`, que funciona igual que `scanf` pero leyendo del archivo en lugar del teclado. La regla mnemotécnica es sencilla: la `f` inicial significa "file". `printf` -> `fprintf`, `scanf` -> `fscanf`, `puts` -> `fputs`, `gets` -> `fgets` (bueno, `gets` está prohibido por inseguro, pero la idea se entiende).

Dos funciones más que debes conocer: `fseek` y `rewind`. Cuando lees o escribes un archivo, hay un cursor invisible que avanza. `fseek` te permite mover ese cursor a una posición específica, y `rewind` lo regresa al principio. No las vas a usar todos los días, pero cuando necesites releer un archivo sin cerrarlo y volverlo a abrir, `rewind(archivo)` te ahorra ese trabajo. La idea básica es: el archivo tiene una posición actual, y tu la puedes controlar.

Ejemplo de agenda que guarda en disco

Vamos a juntar todo en un ejemplo pequeño pero real: una agenda que guarda contactos en un archivo de texto. Cada vez que agregas un contacto, se anexa al final del archivo con modo "`a`". Cuando quieres ver los contactos, lees el archivo completo con modo "`r`". Así los datos sobreviven entre ejecuciones del programa: puedes cerrar la terminal, apagar la computadora, volver mañana, y tus contactos siguen ahí.

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

void agregar_contacto() {
    char nombre[100], telefono[30];
    printf("Nombre: ");
    fgets(nombre, sizeof(nombre), stdin);
    nombre[strcspn(nombre, "\n")] = '\0'; // quita el salto de línea
    printf("Telefono: ");
    fgets(telefono, sizeof(telefono), stdin);
    telefono[strcspn(telefono, "\n")] = '\0';

    FILE *f = fopen("agenda.txt", "a");
    if (f == NULL) {
        printf("Error al abrir el archivo.\n");
        return;
    }
}
```

```

    }
    fprintf(f, "%s|%s\n", nombre, telefono);
    fclose(f);
    printf("Contacto guardado.\n");
}

void mostrar_contactos() {
    FILE *f = fopen("agenda.txt", "r");
    if (f == NULL) {
        printf("Aun no hay contactos guardados.\n");
        return;
    }
    char linea[256];
    printf("\n--- CONTACTOS ---\n");
    while (fgets(linea, sizeof(linea), f) != NULL) {
        printf("%s", linea);
    }
    fclose(f);
}

int main() {
    int opcion;
    do {
        printf("\n1. Agregar contacto\n2. Ver contactos\n0. Salir\nOpcion: ");
        scanf("%d", &opcion);
        getchar(); // consume el salto de linea que deja scanf
        if (opcion == 1) agregar_contacto();
        else if (opcion == 2) mostrar_contactos();
    } while (opcion != 0);
    return 0;
}

```

Fíjate en los detalles profesionales de este ejemplo. Usamos el carácter | como separador entre nombre y teléfono, así cada línea del archivo tiene formato "nombre|teléfono" y después podríamos separar los campos si necesitáramos buscar o borrar. Usamos fgets para leer el nombre porque los nombres tienen espacios y scanf los cortaría. Quitamos el salto de línea final con el truco de strcspn, que ya conoces de la unidad de cadenas. Abrimos en modo "a" para agregar sin borrar los contactos anteriores. Y verificamos NULL en ambas funciones. Este programita de menos de 70 líneas ya es más útil que muchos programas de estudiante, porque sus datos persisten. En la sección de ejemplos vamos a construir la versión completa con bajas, búsqueda y guardado de structs.

Archivos binarios: `fwrite` y `fread` en breve

Hasta ahora trabajamos con archivos de texto, que puedes abrir y leer con cualquier editor. Pero C también puede guardar datos en formato binario: los bytes exactos de la memoria, sin convertirlos a texto. La ventaja es velocidad y precisión (un float se guarda con todos sus decimales exactos, sin pérdidas de conversión), y la desventaja es que el archivo resultante no es legible para humanos.

Las funciones son `fwrite` para escribir y `fread` para leer. Ambas trabajan con bloques de memoria: les dices que escribir, cuánto mide cada elemento y cuántos elementos son. Para archivos binarios se agregan los modos "wb" (write binary), "rb" (read binary) y "ab" (append binary).

```

#include <stdio.h>

typedef struct {
    char nombre[50];
    int edad;
    float promedio;
} Alumno;

int main() {
    Alumno grupo[3] = {
        {"Ana Lopez", 20, 9.5f},
        {"Luis Perez", 22, 8.7f},
        {"Maria Ruiz", 19, 9.1f}
    };

    // guardar: vuelca los structs tal cual estan en memoria
    FILE *f = fopen("alumnos.bin", "wb");
    if (f == NULL) {
        printf("Error al escribir.\n");
        return 1;
    }
    size_t escritos = fwrite(grupo, sizeof(Alumno), 3, f);
    fclose(f);
    printf("Structs guardados: %zu\n", escritos);

    // leer: reconstruye los structs directo a memoria
    Alumno leidos[3];
    f = fopen("alumnos.bin", "rb");
    if (f == NULL) {
        printf("Error al leer.\n");
        return 1;
    }
    size_t n = fread(leidos, sizeof(Alumno), 3, f);
    fclose(f);

    for (size_t i = 0; i < n; i++) {
        printf("%s, %d anios, %.2f\n",
            leidos[i].nombre, leidos[i].edad, leidos[i].promedio);
    }
    return 0;
}

```

Observa la elegancia: `fwrite(grupo, sizeof(Alumno), 3, f)` guarda los tres structs de un solo golpe, sin ciclos ni formatos ni separadores. `fread` hace lo inverso y reconstruye los structs byte por byte. No hay que convertir nada ni parsear nada: lo que sale es lo que entra.

Pero ojo con las limitaciones, porque un profesional las conoce: el archivo binario solo lo entiende un programa que conozca exactamente el mismo struct (mismo orden de campos, mismos tamanos). Si cambias el struct en una versión futura del programa, los archivos viejos pueden dejar de leerse bien. Además, los binarios no son portables entre arquitecturas con distinto orden de bytes, aunque para tus proyectos en una sola computadora eso no te afecta. Por eso en el proyecto final usamos texto: es legible, depurable y tolerante a cambios. Usa binario cuando necesites velocidad o precisión total, y texto cuando necesites claridad y compatibilidad. Saber elegir entre ambos es criterio de profesional.

Buenas prácticas: del código que funciona al código profesional

Hay una frase famosa en programación: "cualquier tonto puede escribir código que una computadora entienda; los buenos programadores escriben código que los humanos entienden". Tu ya sabes hacer que las cosas funcionen. Ahora toca el siguiente nivel: escribir código que otros (y tu yo del futuro) puedan leer, mantener y confiar.

Primera práctica, y la más importante de todas: compila siempre con `-Wall -Wextra`. Cuando compilas con `gcc programa.c -o programa`, el compilador por defecto es callado: solo se queja de errores graves que impiden compilar. Pero hay una categoría intermedia de problemas: cosas que compilan pero que huelen mal, que probablemente son errores. Las banderas `-Wall` (todos los warnings comunes) y `-Wextra` (warnings extra) le ordenan al compilador que te avise de todo lo sospechoso. Un profesional compila así:

```
gcc -Wall -Wextra programa.c -o programa
```

Y la regla de oro es: cero warnings. Un programa que compila con advertencias es un programa con deudas. Cada warning es el compilador diciendote "esto compiló, pero creo que no quisiste hacer esto". Ignorarlos es como ignorar el ruido raro del motor de tu carro: tarde o temprano te deja tirado. Acostumbrate desde hoy: si hay un warning, lo corriges antes de seguir.

Segunda práctica: lee los mensajes del compilador con calma. Los principiantes ven un error del compilador y entran en panico; los profesionales los leen como instrucciones. El compilador te dice el archivo, la línea y que cree que está mal. El truco es empezar por el PRIMER error, porque un solo error al principio puede generar una cascada de errores falsos después. Corrige el primero, recompila, y mágicamente la mitad de los errores desaparecen.

Tercera práctica: nombres claros. Una variable llamada `x` no dice nada; una llamada `total_ventas` lo dice todo. Las funciones deben llamarse como verbos que describen lo que hacen: `calcular_promedio`, `guardar_en_archivo`, `buscar_contacto`. Los nombres largos y descriptivos no son un defecto: son documentación gratis. Tu código se lee muchas más veces de las que se escribe; invierte en nombres.

Cuarta práctica: comenta lo no obvio. Un mal comentario repite lo que el código ya dice ("`// suma a y b`" sobre la línea `suma = a + b`). Un buen comentario explica el porqué: por que se eligio este algoritmo, por que hay una excepción a la regla, que significa ese número mágico. Si tu código necesita un comentario para explicar QUE hace, probablemente el código está mal escrito y deberias mejorarlo en lugar de comentarlo.

Quinta práctica: funciones cortas. Si una función tiene más de 40 o 50 líneas, casi seguro está haciendo más de una cosa y debería dividirse. Cada función debe hacer una sola cosa y hacerla bien. Esto no es estética: las funciones cortas son más fáciles de probar, más fáciles de entender y más fáciles de reutilizar. Cuando en el proyecto final veas el menu principal, notarás que cada opción llama a una función pequena y clara.

Sexta práctica: prueba los casos borde. Los casos borde son las entradas extremas o raras: el archivo vacío, el número cero, el negativo, la cadena vacía, el arreglo lleno, el usuario que escribe letras donde van números. Los programas de estudiante solo se prueban con datos bonitos; los programas profesionales se prueban con datos feos. Antes de dar por terminado cualquier programa, preguntate: qué pasa si el archivo no existe, que pasa si el usuario escribe cero contactos, que pasa si el disco está lleno. Esa paranoia sana es lo que distingue al profesional.

Septima práctica: inicializa tus variables. En C, una variable local no inicializada contiene basura: lo que haya quedado en esa posición de memoria. Usar esa basura produce resultados impredecibles y es fuente de bugs dificiles de encontrar. Acostumbrate a darles valor inicial: `int contador = 0`; `char nombre[100] = ""`; `struct Contacto c = {0}`;. Cuesta una línea y te ahorra horas de depuración.

Depuración: el arte de cazar bugs como detective

Depurar (debugging) es el proceso de encontrar y corregir errores en un programa que compila pero se comporta mal. Aquí va una verdad incomoda: vas a pasar más tiempo depurando que escribiendo código nuevo. Todos los programadores, desde el principiante hasta el que lleva 30 años, depuran todos los días. La diferencia es que el profesional depura con método y el principiante depura con panico y cambios al azar.

El método tiene pasos. Paso uno: reproduce el error de forma confiable. Si no puedes provocar el error cuando quieres, no puedes comprobar que lo arreglaste. Anota exactamente que hiciste para que fallara: que datos metiste, en que orden, que opción elegiste.

Paso dos: lee el mensaje de error completo. Si el programa se cae, el sistema operativo a veces te dice donde (segmentation fault). Si el compilador te dio warnings y los ignoraste, vuelve a ellos: muchas veces el bug estaba avisado desde la compilación.

Paso tres: divide y vencerás. Si el programa es grande, aísla la parte sospechosa. Comenta bloques de código temporalmente o escribe un programita mínimo que reproduzca solo esa parte. Un bug en 20 líneas se encuentra en minutos; el mismo bug escondido en 500 líneas puede tomar horas.

Paso cuatro: los printf estratégicos. Es la técnica de depuración más usada en la historia de C, y no tiene nada de vergüenza. Consiste en imprimir el valor de las variables en puntos clave para ver dónde el programa se desvía de lo esperado. La estrategia es imprimir a la entrada y salida de funciones sospechosas, y los valores de las variables justo antes de donde crees que falla. Cuando encuentres el punto exacto donde un valor se vuelve incorrecto, encontraste tu bug. Eso sí: quita o comenta esos printf cuando termines, porque un programa profesional no escupe mensajes de depuración.

Paso cinco: verifica tus suposiciones. La mayoría de los bugs nacen de algo que asumiste sin comprobar: asumiste que el archivo se abrió, asumiste que el índice no se sale del arreglo, asumiste que el apuntador no es NULL, asumiste que scanf leyó lo que creías. Cada suposición es un sospechoso. Verificalas una por una con printf o con condiciones.

Paso seis: descansa si te atorras. Suena a broma pero es ciencia: después de una hora mirando el mismo código, tu cerebro deja de ver lo obvio. Levantate, toma agua, camina cinco minutos. La cantidad de bugs que se resuelven en la regadera es legendaria entre programadores.

Y una regla de oro de la depuración: cambia una sola cosa a la vez. Si cambias tres cosas y el bug desaparece, no sabes cuál de las tres lo arregló, y probablemente metiste un bug nuevo sin darte cuenta. Un cambio, una prueba, un resultado. Método científico aplicado al código.

Errores comunes con archivos (y como evitarlos)

Vamos a listar los errores más frecuentes que cometen los estudiantes con archivos, para que tu no los cometas:

1. Olvidar fclose. Es el error número uno. Sin fclose, los datos pueden quedarse en el bufer y nunca llegar al disco, y además agotas los descriptores de archivo del sistema si abres

muchos archivos. Regla: cada fopen tiene su fclose, como cada paréntesis que abres lo cierras.

2. No verificar el resultado de fopen. Si el archivo no existe (en modo "r") o no tienes permiso de escritura, fopen devuelve NULL. Si después usas ese NULL en fprintf, el programa se cae con segmentation fault. Verificar NULL no es opcional: es lo que separa un programa robusto de uno frágil. En el proyecto final verás verificaciones en cada apertura.
3. Usar el modo de apertura equivocado. Abrir en "w" cuando querías "a" borra todo el archivo: pérdida de datos irreversible. Abrir en "r" un archivo que no existe devuelve NULL. Antes de escribir fopen, preguntate: quiero leer, reemplazar o agregar. Esa pregunta de tres segundos evita desastres.
4. Mezclar scanf con fgets sin limpiar el bufer. scanf("%d") deja el salto de línea en el bufer de entrada, y el siguiente fgets lo lee como una línea vacía. La solución es consumir ese salto de línea con getchar() después de scanf, como hicimos en el ejemplo de la agenda. Este bug confunde a todos los principiantes; ahora tu ya sabes que existe.
5. Olvidar el salto de línea al escribir registros. Si escribes fprintf(f, "%s|s", ...) sin \n al final, todos los registros quedan pegados en una sola línea gigante y después no puedes leerlos uno por uno con fgets. Cada registro, una línea: es la convención más simple y la que usaremos en el proyecto final.
6. Leer con fscanf datos que contienen espacios. fscanf con %s se detiene en el primer espacio, igual que scanf. Para líneas completas usa fgets. Elige la herramienta según tus datos: campos simples sin espacios -> fscanf; líneas con espacios -> fgets.

Proyecto final: especificación del sistema

El proyecto final del curso es un sistema completo de agenda de contactos (y te dare la variante de inventario como ejercicio). Aquí esta la especificación formal, como la escribiría un cliente real:

El sistema debe permitir al usuario gestionar una lista de contactos, donde cada contacto tiene nombre, teléfono y correo electrónico. El programa presenta un menú con las siguientes opciones: 1) agregar contacto, 2) mostrar todos los contactos, 3) buscar un contacto por nombre, 4) eliminar un contacto, 5) guardar y 6) salir. Los datos deben persistir en un archivo de texto en disco, de modo que al cerrar y volver a abrir el programa, los contactos sigan ahí. El programa debe manejar memoria dinámica para la lista de contactos (que pueda crecer), debe verificar todos los errores de apertura de archivo, no debe tener warnings al compilar con -Wall -Wextra, y debe estar dividido en funciones cortas y claras.

Requisitos técnicos que demuestran todo lo aprendido en el curso:

- Uso de struct para modelar el contacto (clase 25).
- Uso de typedef para el tipo Contacto (clase 26).
- Arreglo dinámico con malloc y realloc que crece según se agregan contactos (clase 27).
- Archivos de texto para persistencia: guardar al salir y cargar al iniciar (clase 28).
- Menu con switch (clase 11) y ciclo do-while (clase 15).
- Funciones separadas para cada operación (clases 16 y 17).
- Validación de entradas y manejo de casos borde (clase 29).
- Cero warnings con gcc -Wall -Wextra (clase 29).

Guía de diseño paso a paso

Antes de escribir una sola línea del proyecto, un profesional diseña. Aquí está el proceso aplicado a nuestra agenda:

Paso 1: define los datos. Que información guarda un contacto. Respuesta: nombre (cadena de hasta 99 caracteres), teléfono (cadena de hasta 29), email (cadena de hasta 99). Eso se convierte directamente en un struct. También necesitas la colección: un arreglo dinámico de contactos con su tamaño actual y su capacidad.

Paso 2: define las operaciones. Que puede hacer el usuario con esos datos. Respuesta: agregar, mostrar, buscar, eliminar, guardar, cargar, salir. Cada operación será una función. Esta es la parte más importante del diseño: si tus operaciones están claras, el código casi se escribe solo.

Paso 3: define la persistencia. Cómo se guardan los datos en disco. Decisión: archivo de texto "agenda.dat", un contacto por línea, campos separados por |. Al iniciar, el programa carga el archivo a memoria; al elegir guardar (o al salir), escribe todo el arreglo al archivo en modo "w" (reescritura completa, porque es más simple y segura que intentar actualizar líneas sueltas).

Paso 4: define el crecimiento de memoria. Empiezas con capacidad para 10 contactos (malloc). Cuando se llena, duplicas la capacidad con realloc. Duplicar (en lugar de crecer de uno en uno) es la estrategia estándar porque realloc es costoso y duplicar mantiene el número de realocaciones bajo.

Paso 5: define los casos borde. Qué pasa si el archivo no existe al iniciar (respuesta: se empieza con lista vacía, no es error). Qué pasa si se elimina un contacto que no existe (respuesta: mensaje de aviso). Qué pasa si realloc falla (respuesta: mensaje de error y no se pierde la memoria original). Qué pasa si el usuario busca con cadena vacía.

Paso 6: escribe el esqueleto primero. Las firmas de las funciones y el menu, con cuerpos vacíos o minimos, y compila. Un esqueleto que compila te da una base solida sobre la cual construir pieza por pieza, probando cada función conforme la completas. En la sección de ejemplos verás exactamente este esqueleto antes de la versión completa.

Esta metodologia (datos, operaciones, persistencia, memoria, casos borde, esqueleto) te sirve para cualquier proyecto en C, no solo para la agenda. Es la forma profesional de empezar.

Ejemplos paso a paso

Ejemplo 1: escribir un archivo de texto

El programa más básico de archivos: crear un archivo y escribir en el tres líneas con formato. Es el "hola mundo" de la persistencia. Compilalo, ejecutalo y luego abre el archivo generado con cualquier editor de texto para comprobar que ahí esta.

```
#include <stdio.h>

int main() {
    FILE *f = fopen("notas.txt", "w");
    if (f == NULL) {
        printf("Error: no se pudo crear el archivo.\n");
        return 1;
    }

    fprintf(f, "Mis notas de programacion en C\n");
    fprintf(f, "Clase 28: aprendi a manejar archivos.\n");
    fprintf(f, "Calificacion esperada: %d\n", 10);

    fclose(f);
    printf("Archivo 'notas.txt' creado con exito.\n");
    return 0;
}
```

Paso a paso: incluimos stdio.h, que trae FILE, fopen, fprintf y fclose. Abrimos "notas.txt" en modo "w": si no existe lo crea, si existe lo vacía. Verificamos NULL por si no hay permiso de escritura. Escribimos tres líneas con fprintf, usando %d igual que en printf. Cerramos con fclose para asegurar que los datos lleguen al disco. Si ejecutas el programa dos veces, el archivo se reescribe desde cero cada vez: eso es el comportamiento de "w". Prueba cambiar a "a" y ejecuta dos veces para ver la diferencia con tus propios ojos.

Ejemplo 2: leer un archivo línea por línea

Ahora el complemento: leer el archivo que creamos y mostrarlo en pantalla numerando las líneas. Este patrón de lectura lo usarás para cargar datos al iniciar tus programas.

```
#include <stdio.h>

int main() {
    FILE *f = fopen("notas.txt", "r");
    if (f == NULL) {
        printf("Error: el archivo no existe. Ejecuta primero el ejemplo 1.\n");
        return 1;
    }

    char linea[256];
    int numero = 1;
    while (fgets(linea, sizeof(linea), f) != NULL) {
        printf("%d: %s", numero, linea);
        numero++;
    }

    fclose(f);
    printf("Total de lineas leidas: %d\n", numero - 1);
    return 0;
}
```

Paso a paso: abrimos en modo "r" porque solo vamos a leer. Si el archivo no existe, damos un mensaje útil que le dice al usuario que hacer (buena práctica: los mensajes de error deben orientar, no solo quejarse). El ciclo while con fgets lee una línea por iteración y se detiene solo al llegar al final del archivo, cuando fgets devuelve NULL. Nota que printf("%d: %s", ...) no agrega salto de línea extra porque línea ya trae su \n al final. Al terminar cerramos el archivo e informamos cuántas líneas leímos. Este es exactamente el patrón que usa la función de carga del proyecto final.

Ejemplo 3: guardar un struct en archivo de texto

Aquí subimos de nivel: en lugar de cadenas sueltas, guardamos registros estructurados. Definimos un struct Alumno y guardamos varios en un archivo, un registro por línea con campos separados por |. Después los leemos de vuelta. Este ejemplo es el corazón del proyecto final en miniatura.

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

```

typedef struct {
    char nombre[50];
    int edad;
    float promedio;
} Alumno;

void guardar_alumnos(Alumno grupo[], int n) {
    FILE *f = fopen("alumnos.txt", "w");
    if (f == NULL) {
        printf("Error al abrir el archivo para escritura.\n");
        return;
    }
    for (int i = 0; i < n; i++) {
        fprintf(f, "%s|%d|%.2f\n", grupo[i].nombre, grupo[i].edad, grupo[i].promedio);
    }
    fclose(f);
    printf("Se guardaron %d alumnos.\n", n);
}

int cargar_alumnos(Alumno grupo[], int max) {
    FILE *f = fopen("alumnos.txt", "r");
    if (f == NULL) {
        return 0; // no hay archivo: se empieza vacio, no es error
    }
    int n = 0;
    char linea[256];
    while (n < max && fgets(linea, sizeof(linea), f) != NULL) {
        // separa la linea por el caracter |
        char *campo = strtok(linea, "|");
        if (campo == NULL) continue;
        strcpy(grupo[n].nombre, campo);

        campo = strtok(NULL, "|");
        if (campo == NULL) continue;
        grupo[n].edad = atoi(campo);

        campo = strtok(NULL, "|\n");
        if (campo == NULL) continue;
        grupo[n].promedio = atof(campo);

        n++;
    }
    fclose(f);
    return n;
}

```

```

}

int main() {
    Alumno grupo[100];
    int n = 0;

    // cargamos lo que hubiera de ejecuciones anteriores
    n = cargar_alumnos(grupo, 100);
    printf("Alumnos cargados del archivo: %d\n", n);

    // agregamos dos nuevos
    strcpy(grupo[n].nombre, "Ana Lopez");
    grupo[n].edad = 20;
    grupo[n].promedio = 9.5;
    n++;
    strcpy(grupo[n].nombre, "Luis Perez");
    grupo[n].edad = 22;
    grupo[n].promedio = 8.7;
    n++;

    guardar_alumnos(grupo, n);

    printf("\nLista completa:\n");
    for (int i = 0; i < n; i++) {
        printf("- %s, %d años, promedio %.2f\n",
            grupo[i].nombre, grupo[i].edad, grupo[i].promedio);
    }
    return 0;
}

```

Paso a paso: definimos el struct con typedef como aprendiste en la clase 26. `guardar_alumnos` abre en "w" y escribe cada struct en una línea con formato `nombre|edad|promedio`. `cargar_alumnos` abre en "r"; si el archivo no existe devuelve 0 sin quejarse, porque en la primera ejecución es normal que no exista. Para reconstruir cada struct usamos `strtok` para partir la línea por el separador `|`, y convertimos los campos numericos con `atoi` y `atof`. El main demuestra el ciclo completo de persistencia: carga lo anterior, agrega datos nuevos, guarda todo y muestra. Ejecuta el programa tres veces y verás como la lista crece en cada ejecución: tus datos sobreviven al apagon. Este es el mecanismo exacto que usa la agenda del proyecto final.

Ejemplo 4: contador de visitas persistente

Un clásico de la persistencia: un programa que recuerda cuántas veces se ha ejecutado. Cada vez que corre, lee el contador del archivo, le suma uno, lo guarda y lo muestra. Es el mismo principio que usan las páginas web para contar visitas.

```
#include <stdio.h>

int main() {
    int visitas = 0;

    // intenta leer el contador anterior
    FILE *f = fopen("contador.txt", "r");
    if (f != NULL) {
        fscanf(f, "%d", &visitas);
        fclose(f);
    }
    // si el archivo no existia, visitas se queda en 0: primera visita

    visitas++;
    printf("Esta es la visita numero %d.\n", visitas);

    // guarda el nuevo valor
    f = fopen("contador.txt", "w");
    if (f == NULL) {
        printf("Error: no se pudo guardar el contador.\n");
        return 1;
    }
    fprintf(f, "%d\n", visitas);
    fclose(f);

    return 0;
}
```

Paso a paso: intentamos abrir en "r". Si el archivo existe, leemos el número con `fscanf` y cerramos. Si no existe (primera ejecución), simplemente empezamos en cero: observa que aquí NO tratamos el NULL como error fatal, porque la ausencia del archivo es un caso normal y esperado. Incrementamos, mostramos y guardamos en modo "w". Ejecuta el programa cinco veces seguidas y verás el número subir: 1, 2, 3, 4, 5. Este ejemplo te enseña una lección sutil pero importante: no todo NULL de `fopen` es un error; a veces es una condición normal que tu programa debe manejar con elegancia. La diferencia entre un error y un caso borde esperado está en tu diseño.

Ejemplo 5: programa con -Wall corregido

Vamos a ver la diferencia entre compilar "a la brava" y compilar como profesional. Este programa tiene tres problemas que el compilador normal no reporta pero -Wall si. Primero mira el código con errores:

```
#include <stdio.h>

int main() {
    int edad;
    printf("Tu edad: ");
    scanf("%d", &edad);
    if (edad = 18) {
        printf("Tienes 18 años.\n");
    }
    return 0;
}
```

Si compilas con gcc programa.c -o programa, compila sin quejarse. Pero si compilas con gcc -Wall -Wextra programa.c -o programa, el compilador te avisa:

```
warning: suggest parentheses around assignment used as truth value [-Wparentheses]
```

Te está diciendo: "oye, escribiste if (edad = 18), eso es una ASIGNACIÓN, no una comparación; siempre será verdadero". El programa tenía un bug lógico grave y el compilador callado no dijo nada. La versión corregida:

```
#include <stdio.h>

int main() {
    int edad = 0;
    printf("Tu edad: ");
    if (scanf("%d", &edad) != 1) {
        printf("Error: eso no es un numero.\n");
        return 1;
    }
    if (edad == 18) {
        printf("Tienes 18 años.\n");
    } else {
        printf("No tienes 18 años.\n");
    }
    return 0;
}
```

Cambios aplicados: == en lugar de = (el bug original), variable inicializada en cero, y verificación del valor de retorno de scanf para detectar entradas invalidas. Compila esta versión con -Wall -Wextra y obtendras silencio total: cero warnings. Ese silencio es música para un profesional. Acostumbrate a buscarlo: compila con las banderas siempre, y no des por terminado un programa hasta que el compilador no tenga nada que decirte. En lucasegaacademy.com insistimos mucho en esto porque es el habito que más rápido te hace ver como programador serio.

Ejemplo 6: sesión de depuración narrada

Vamos a depurar juntos un programa con un bug escondido, paso a paso, como lo haría un profesional. El programa debería calcular el promedio de un arreglo, pero siempre da un resultado raro:

```
#include <stdio.h>

float promedio(int datos[], int n) {
    int suma = 0;
    for (int i = 0; i <= n; i++) {
        suma += datos[i];
    }
    return (float)suma / n;
}

int main() {
    int califs[] = {8, 9, 7, 10, 6};
    printf("Promedio: %.2f\n", promedio(califs, 5));
    return 0;
}
```

El promedio real de 8, 9, 7, 10 y 6 es 8.0, pero el programa imprime algo como 8.40 o cualquier otro número extraño, y a veces hasta se cae. Apliquemos el método:

Paso uno, reproducir: lo compilamos y ejecutamos varias veces. El resultado es inconsistente: a veces da un número, a veces otro. Inconsistencia entre ejecuciones es la firma de un bug de memoria: estamos leyendo basura de algún lado.

Paso dos, leer warnings: compilamos con -Wall -Wextra... silencio. El compilador no lo detecta. No todos los bugs los atrapa el compilador; por eso existe la depuración.

Paso tres, dividir y vencer: la función es pequeña, solo el ciclo es sospechoso. Agregamos printf estratégicos dentro del ciclo para ver qué está sumando:

```
float promedio(int datos[], int n) {
    int suma = 0;
    for (int i = 0; i <= n; i++) {
        printf("[debug] i=%d, datos[i]=%d, suma=%d\n", i, datos[i], suma);
        suma += datos[i];
    }
    return (float)suma / n;
}
```

Al ejecutar vemos:

```
[debug] i=0, datos[i]=8, suma=0
[debug] i=1, datos[i]=9, suma=8
[debug] i=2, datos[i]=7, suma=17
[debug] i=3, datos[i]=10, suma=24
[debug] i=4, datos[i]=6, suma=34
[debug] i=5, datos[i]=32764, suma=40
```

Ahí está el culpable: cuando *i* llega a 5, leemos `datos[5]`, pero el arreglo solo tiene índices válidos del 0 al 4. `datos[5]` es memoria basura (32764 o lo que sea), y contamina la suma. El bug es la condición del `for`: dice `i <= n` cuando debería decir `i < n`. Es el error "off by one", uno de los más clásicos de la historia de la programación.

Paso cuatro, corregir y verificar: cambiamos `<=` por `<`, quitamos los `printf` de depuración, recompilamos y el programa imprime 8.00 de forma consistente. Bug cazado.

La moraleja: el `printf` estratégico te mostró exactamente en qué iteración el valor se volvió loco. Sin él, habrías mirado el código por horas sin ver el `<=` porque tu cerebro lee lo que espera ver, no lo que está escrito. Depurar con método siempre le gana a depurar con esperanza.

Ejemplo 7: esqueleto del proyecto final

Antes del programa completo, el esqueleto: todas las funciones declaradas, el menú funcionando y cuerpos mínimos que compilan. Este es el punto de partida profesional: primero la estructura, después el relleno.

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

#define NOMBRE_ARCHIVO "agenda.dat"
#define CAPACIDAD_INICIAL 10

typedef struct {
    char nombre[100];
    char telefono[30];
    char email[100];
} Contacto;

typedef struct {
    Contacto *datos;
    int cantidad;
    int capacidad;
} Agenda;
```

```

void agenda_init(Agenda *a);
void agenda_liberar(Agenda *a);
void agenda_agregar(Agenda *a);
void agenda_mostrar(const Agenda *a);
void agenda_buscar(const Agenda *a);
void agenda_eliminar(Agenda *a);
int agenda_guardar(const Agenda *a);
int agenda_cargar(Agenda *a);
void limpiar_linea(char *texto);

int main() {
    Agenda agenda;
    agenda_init(&agenda);
    agenda_cargar(&agenda);

    int opcion;
    do {
        printf("\n=== AGENDA DE CONTACTOS ===\n");
        printf("1. Agregar contacto\n");
        printf("2. Mostrar contactos\n");
        printf("3. Buscar contacto\n");
        printf("4. Eliminar contacto\n");
        printf("5. Guardar\n");
        printf("0. Salir\n");
        printf("Opcion: ");
        if (scanf("%d", &opcion) != 1) {
            opcion = -1;
            while (getchar() != '\n');
        }
        getchar();

        switch (opcion) {
            case 1: agenda_agregar(&agenda); break;
            case 2: agenda_mostrar(&agenda); break;
            case 3: agenda_buscar(&agenda); break;
            case 4: agenda_eliminar(&agenda); break;
            case 5:
                if (agenda_guardar(&agenda))
                    printf("Agenda guardada.\n");
                break;
            case 0: printf("Saliendo...\n"); break;
            default: printf("Opcion no valida.\n");
        }
    } while (opcion != 0);
}

```

```

    agenda_guardar(&agenda);
    agenda_liberar(&agenda);
    return 0;
}

// Cuerpos minimos: compilan, falta implementarlos
void agenda_init(Agenda *a) { a->datos = NULL; a->cantidad = 0; a->capacidad = 0; }
void agenda_liberar(Agenda *a) { free(a->datos); }
void agenda_agregar(Agenda *a) { printf("(agregar: pendiente)\n"); }
void agenda_mostrar(const Agenda *a) { printf("(mostrar: pendiente)\n"); }
void agenda_buscar(const Agenda *a) { printf("(buscar: pendiente)\n"); }
void agenda_eliminar(Agenda *a) { printf("(eliminar: pendiente)\n"); }
int agenda_guardar(const Agenda *a) { return 0; }
int agenda_cargar(Agenda *a) { return 0; }
void limpiar_linea(char *texto) { texto[strcspn(texto, "\n")] = '\0'; }

```

Este esqueleto ya compila con -Wall -Wextra sin warnings y el menu ya navega. A partir de aquí implementas una función a la vez, compilando y probando después de cada una. Así nunca tienes 200 líneas nuevas sin probar: siempre avanzas sobre terreno firme.

Ejemplo 8: proyecto final completo y funcional

Aquí esta: la agenda de contactos completa, con todo lo aprendido en el curso. Memoria dinámica que crece sola, persistencia en archivo, menu con switch, búsqueda, eliminación, validaciones y cero warnings con -Wall -Wextra. Estúdialo con calma: cada función es corta y hace una sola cosa.

```

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

#define NOMBRE_ARCHIVO "agenda.dat"
#define CAPACIDAD_INICIAL 10

typedef struct {
    char nombre[100];
    char telefono[30];
    char email[100];
} Contacto;

typedef struct {
    Contacto *datos;
    int cantidad;
    int capacidad;
}

```

```

} Agenda;

void agenda_init(Agenda *a);
void agenda_liberar(Agenda *a);
int agenda_crecer(Agenda *a);
void agenda_agregar(Agenda *a);
void agenda_mostrar(const Agenda *a);
void agenda_buscar(const Agenda *a);
void agenda_eliminar(Agenda *a);
int agenda_guardar(const Agenda *a);
int agenda_cargar(Agenda *a);
void limpiar_linea(char *texto);
void leer_texto(const char *mensaje, char *destino, int tam);

int main() {
    Agenda agenda;
    agenda_init(&agenda);

    if (!agenda_crecer(&agenda)) {
        printf("Error: no hay memoria suficiente.\n");
        return 1;
    }
    int cargados = agenda_cargar(&agenda);
    printf("Se cargaron %d contactos del archivo.\n", cargados);

    int opcion;
    do {
        printf("\n=== AGENDA DE CONTACTOS ===\n");
        printf("Contactos registrados: %d\n", agenda.cantidad);
        printf("1. Agregar contacto\n");
        printf("2. Mostrar contactos\n");
        printf("3. Buscar contacto\n");
        printf("4. Eliminar contacto\n");
        printf("5. Guardar\n");
        printf("0. Salir\n");
        printf("Opcion: ");
        if (scanf("%d", &opcion) != 1) {
            opcion = -1;
            while (getchar() != '\n');
        }
        getchar();

        switch (opcion) {
            case 1: agenda_agregar(&agenda); break;
            case 2: agenda_mostrar(&agenda); break;

```



```

        case 3: agenda_buscar(&agenda); break;
        case 4: agenda_eliminar(&agenda); break;
        case 5:
            if (agenda_guardar(&agenda))
                printf("Agenda guardada correctamente.\n");
            else
                printf("Error al guardar la agenda.\n");
            break;
        case 0: printf("Guardando y saliendo...\n"); break;
        default: printf("Opcion no valida, intenta de nuevo.\n");
    }
} while (opcion != 0);

agenda_guardar(&agenda);
agenda_liberar(&agenda);
return 0;
}

void agenda_init(Agenda *a) {
    a->datos = NULL;
    a->cantidad = 0;
    a->capacidad = 0;
}

void agenda_liberar(Agenda *a) {
    free(a->datos);
    a->datos = NULL;
    a->cantidad = 0;
    a->capacidad = 0;
}

// Duplica la capacidad cuando el arreglo se llena. Devuelve 1 si ok, 0 si falla.
int agenda_crecer(Agenda *a) {
    int nueva_capacidad = (a->capacidad == 0) ? CAPACIDAD_INICIAL : a->capacidad * 2;
    Contacto *nuevo = realloc(a->datos, nueva_capacidad * sizeof(Contacto));
    if (nuevo == NULL) {
        return 0;
    }
    a->datos = nuevo;
    a->capacidad = nueva_capacidad;
    return 1;
}

void leer_texto(const char *mensaje, char *destino, int tam) {
    printf("%s", mensaje);

```

```

    fgets(destino, tam, stdin);
    limpiar_linea(destino);
}

void limpiar_linea(char *texto) {
    texto[strcspn(texto, "\n")] = '\0';
}

void agenda_agregar(Agenda *a) {
    if (a->cantidad >= a->capacidad) {
        if (!agenda_creecer(a)) {
            printf("Error: no se pudo ampliar la agenda.\n");
            return;
        }
    }
    Contacto *c = &a->datos[a->cantidad];
    leer_texto("Nombre: ", c->nombre, sizeof(c->nombre));
    if (strlen(c->nombre) == 0) {
        printf("El nombre no puede estar vacio.\n");
        return;
    }
    leer_texto("Telefono: ", c->telefono, sizeof(c->telefono));
    leer_texto("Email: ", c->email, sizeof(c->email));
    a->cantidad++;
    printf("Contacto agregado.\n");
}

void agenda_mostrar(const Agenda *a) {
    if (a->cantidad == 0) {
        printf("La agenda esta vacia.\n");
        return;
    }
    printf("\n--- CONTACTOS (%d) ---\n", a->cantidad);
    for (int i = 0; i < a->cantidad; i++) {
        printf("%d. %s | Tel: %s | Email: %s\n",
            i + 1,
            a->datos[i].nombre,
            a->datos[i].telefono,
            a->datos[i].email);
    }
}

void agenda_buscar(const Agenda *a) {
    char busqueda[100];
    leer_texto("Nombre a buscar: ", busqueda, sizeof(busqueda));

```

```

int encontrados = 0;
for (int i = 0; i < a->cantidad; i++) {
    if (strstr(a->datos[i].nombre, busqueda) != NULL) {
        printf("%d. %s | Tel: %s | Email: %s\n",
            i + 1,
            a->datos[i].nombre,
            a->datos[i].telefono,
            a->datos[i].email);
        encontrados++;
    }
}
if (encontrados == 0) {
    printf("No se encontraron contactos con ese nombre.\n");
} else {
    printf("Se encontraron %d coincidencia(s).\n", encontrados);
}
}

void agenda_eliminar(Agenda *a) {
    if (a->cantidad == 0) {
        printf("La agenda esta vacia, no hay nada que eliminar.\n");
        return;
    }
    agenda_mostrar(a);
    printf("Numero del contacto a eliminar (0 para cancelar): ");
    int num;
    if (scanf("%d", &num) != 1) {
        while (getchar() != '\n');
        printf("Entrada no valida.\n");
        return;
    }
    getchar();
    if (num == 0) {
        printf("Operacion cancelada.\n");
        return;
    }
    if (num < 1 || num > a->cantidad) {
        printf("Numero fuera de rango.\n");
        return;
    }
    // recorre los elementos posteriores una posicion hacia atras
    for (int i = num - 1; i < a->cantidad - 1; i++) {
        a->datos[i] = a->datos[i + 1];
    }
    a->cantidad--;
}

```

```

    printf("Contacto eliminado.\n");
}

int agenda_guardar(const Agenda *a) {
    FILE *f = fopen(NOMBRE_ARCHIVO, "w");
    if (f == NULL) {
        return 0;
    }
    for (int i = 0; i < a->cantidad; i++) {
        fprintf(f, "%s|%s|%s\n",
            a->datos[i].nombre,
            a->datos[i].telefono,
            a->datos[i].email);
    }
    fclose(f);
    return 1;
}

int agenda_cargar(Agenda *a) {
    FILE *f = fopen(NOMBRE_ARCHIVO, "r");
    if (f == NULL) {
        return 0; // primera ejecucion: no hay archivo, lista vacia
    }
    char linea[256];
    int cargados = 0;
    while (fgets(linea, sizeof(linea), f) != NULL) {
        if (a->cantidad >= a->capacidad) {
            if (!agenda_creer(a)) {
                break; // sin memoria: conserva lo cargado hasta aqui
            }
        }
        Contacto *c = &a->datos[a->cantidad];
        char *campo = strtok(linea, "|");
        if (campo == NULL) continue;
        strncpy(c->nombre, campo, sizeof(c->nombre) - 1);
        c->nombre[sizeof(c->nombre) - 1] = '\0';

        campo = strtok(NULL, "|");
        if (campo == NULL) continue;
        strncpy(c->telefono, campo, sizeof(c->telefono) - 1);
        c->telefono[sizeof(c->telefono) - 1] = '\0';

        campo = strtok(NULL, "\\n");
        if (campo == NULL) continue;
        strncpy(c->email, campo, sizeof(c->email) - 1);
    }
}

```

```

        c->email[sizeof(c->email) - 1] = '\0';

        a->cantidad++;
        cargados++;
    }
    fclose(f);
    return cargados;
}

```

Cómo usarlo: compilalo con `gcc -Wall -Wextra agenda.c -o agenda`. Ejecuta, agrega varios contactos, elige guardar, sal, y vuelve a ejecutar: tus contactos siguen ahí. Abre `agenda.dat` con un editor de texto y verás el formato `nombre|teléfono|email`, una línea por contacto.

Observa como cada pieza del curso aparece: structs y typedef (clase 25 y 26), malloc/realloc/free (clase 27), archivos con fopen/fprintf/fgets/fclose y verificación de NULL (clase 28), switch y do-while (clases 11 y 15), funciones separadas y parámetros por apuntador (clases 16, 17 y 24), validación de entradas y casos borde (clase 29). Este programa es la prueba de que completaste el curso de verdad. Guardalo, porque en la sección de práctica lo vas a ampliar.

Práctica

1. Registro de temperaturas: escribe un programa que pida al usuario 7 temperaturas (una por día de la semana), las guarde en un archivo `temperaturas.txt` (una por línea), y después lea el archivo para calcular y mostrar el promedio semanal, la máxima y la mínima. Verifica NULL en cada fopen.
2. Copiador de archivos: escribe un programa que reciba dos nombres de archivo (origen y destino) y copie el contenido del primero al segundo, línea por línea. Debe manejar los errores: si el origen no existe, avisar; si el destino no se puede crear, avisar. Pista: es como el comando `cp` de la terminal, pero hecho por ti en C.
3. Agenda sin duplicados: modifica la función `agenda_agregar` del proyecto final para que no permita agregar un contacto cuyo nombre ya exista en la agenda (comparación insensible a mayúsculas y minúsculas usando `tolower` de `ctype.h`). Si el nombre ya existe, muestra un aviso y no lo agregues.
4. Caza de warnings: toma el siguiente programa, compilalo con `gcc -Wall -Wextra`, lee todos los warnings, corrígelos uno por uno hasta lograr cero warnings, y explica qué estaba mal en cada caso:

```
#include <stdio.h>
int main() {
    int n;
    printf("Dame un numero: ");
    scanf("%d", &n);
    float mitad = n / 2;
    printf("La mitad es: %f\n", mitad);
    return 0;
}
```

5. Editar contacto (ampliación del proyecto): agrega al menu del proyecto final la opción 6 "Editar contacto", que muestre la lista, pida el número del contacto, y permita modificar su teléfono y su email (el nombre no se toca). Guarda los cambios en el archivo. Recuerda validar el número ingresado.
6. Sistema de inventario (variante del proyecto): adapta el proyecto final para que en lugar de contactos maneje productos de una tienda. Cada producto tiene nombre, cantidad en existencia y precio unitario. El menu debe permitir agregar producto, mostrar inventario, buscar producto, eliminar producto, mostrar el valor total del inventario (suma de cantidad * precio de cada producto) y guardar/salir. Reutiliza la estructura del proyecto: solo cambian los campos del struct y las funciones específicas.

Solución del ejercicio 1

```

#include <stdio.h>

int main() {
    float temps[7];

    FILE *f = fopen("temperaturas.txt", "w");
    if (f == NULL) {
        printf("Error: no se pudo crear el archivo.\n");
        return 1;
    }
    for (int i = 0; i < 7; i++) {
        printf("Temperatura del dia %d: ", i + 1);
        if (scanf("%f", &temps[i]) != 1) {
            printf("Entrada no valida.\n");
            fclose(f);
            return 1;
        }
        fprintf(f, "%.2f\n", temps[i]);
    }
    fclose(f);

    f = fopen("temperaturas.txt", "r");
    if (f == NULL) {
        printf("Error: no se pudo leer el archivo.\n");
        return 1;
    }
    float valor, suma = 0, max, min;
    int n = 0;
    int primero = 1;
    while (fscanf(f, "%f", &valor) == 1) {
        suma += valor;
        if (primero || valor > max) max = valor;
        if (primero || valor < min) min = valor;
        primero = 0;
        n++;
    }
    fclose(f);

    if (n > 0) {
        printf("Promedio semanal: %.2f\n", suma / n);
        printf("Maxima: %.2f, Minima: %.2f\n", max, min);
    }
    return 0;
}

```

Explicación: primero escribimos las 7 temperaturas en modo "w", verificando scanf y cerrando el archivo. Después lo reabrimos en modo "r" y lo leemos con fscanf, que devuelve 1 por cada número leído correctamente y algo distinto al llegar al final. Usamos la bandera primero para inicializar max y min con el primer valor real en lugar de un número arbitrario: así funciona aunque todas las temperaturas sean negativas. Este patrón de "escribir, cerrar, reabrir, leer" es el ciclo básico de persistencia.

Solución del ejercicio 2

```
#include <stdio.h>

int main() {
    char origen[256], destino[256];
    printf("Archivo origen: ");
    scanf("%255s", origen);
    printf("Archivo destino: ");
    scanf("%255s", destino);

    FILE *fo = fopen(origen, "r");
    if (fo == NULL) {
        printf("Error: no se pudo abrir '%s' para lectura.\n", origen);
        return 1;
    }
    FILE *fd = fopen(destino, "w");
    if (fd == NULL) {
        printf("Error: no se pudo crear '%s'.\n", destino);
        fclose(fo);
        return 1;
    }

    char linea[1024];
    long lineas = 0;
    while (fgets(linea, sizeof(linea), fo) != NULL) {
        fputs(linea, fd);
        lineas++;
    }

    fclose(fo);
    fclose(fd);
    printf("Copia terminada: %ld lineas copiadas.\n", lineas);
    return 0;
}
```


Explicación: abrimos el origen en "r" y el destino en "w". Si el destino falla, cerramos el origen antes de salir: dejar archivos abiertos al terminar con error es un descuido común. Copiamos línea por línea con fgets/fputs, que preserva el contenido exacto incluyendo espacios y saltos de línea. Contamos las líneas para informar al usuario. Nota el %255s en scanf: limita la lectura a 255 caracteres para no desbordar el bufer de 256. Ese detalle es buena práctica de seguridad que un profesional siempre aplica.

Solución del ejercicio 3

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

// Devuelve 1 si el nombre ya existe (ignorando mayusculas/minusculas)
int nombre_existe(const Agenda *a, const char *nombre) {
    for (int i = 0; i < a->cantidad; i++) {
        const char *p1 = a->datos[i].nombre;
        const char *p2 = nombre;
        int iguales = 1;
        while (*p1 && *p2) {
            if (tolower((unsigned char)*p1) != tolower((unsigned char)*p2)) {
                iguales = 0;
                break;
            }
            p1++;
            p2++;
        }
        if (iguales && *p1 == '\0' && *p2 == '\0') {
            return 1;
        }
    }
    return 0;
}

void agenda_agregar(Agenda *a) {
    char nombre[100];
    leer_texto("Nombre: ", nombre, sizeof(nombre));
    if (strlen(nombre) == 0) {
        printf("El nombre no puede estar vacio.\n");
        return;
    }
    if (nombre_existe(a, nombre)) {
        printf("Ese contacto ya existe en la agenda.\n");
        return;
    }
}
```

```

    }
    if (a->cantidad >= a->capacidad) {
        if (!agenda_crecer(a)) {
            printf("Error: no se pudo ampliar la agenda.\n");
            return;
        }
    }
    Contacto *c = &a->datos[a->cantidad];
    strcpy(c->nombre, nombre);
    leer_texto("Telefono: ", c->telefono, sizeof(c->telefono));
    leer_texto("Email: ", c->email, sizeof(c->email));
    a->cantidad++;
    printf("Contacto agregado.\n");
}

```

Explicación: la función `nombre_existe` recorre la agenda comparando carácter por carácter con `tolower`, así "Ana" y "ANA" se consideran el mismo nombre. Pedimos el nombre ANTES de pedir los demás datos y ANTES de crecer el arreglo: si el nombre está duplicado o vacío, salimos temprano sin haber hecho trabajo innecesario. Este orden (validar primero, actuar después) es un principio de diseño que mantiene el código limpio y evita estados inconsistentes.

Solución del ejercicio 4

Al compilar con `gcc -Wall -Wextra` aparecen estos warnings:

1. warning: ignoring return value of 'scanf' [-Wunused-result]: `scanf` puede fallar y estamos ignorando su resultado. Se corrige verificando que devuelva 1.
2. No hay warning directo para la división, pero hay un bug lógico: $n / 2$ es división ENTERA porque ambos operandos son `int`. Si n es 7, $n / 2$ da 3, no 3.5, y ese 3 se convierte a `float` después. El compilador con `-Wextra` a veces avisa de conversiones, pero este bug lógico hay que cazarlo con la mente. Se corrige con $n / 2.0$.

Versión corregida, cero warnings:

```
#include <stdio.h>

int main() {
    int n = 0;
    printf("Dame un numero: ");
    if (scanf("%d", &n) != 1) {
        printf("Error: eso no es un numero.\n");
        return 1;
    }
    float mitad = n / 2.0f;
    printf("La mitad es: %.2f\n", mitad);
    return 0;
}
```

Explicación: inicializamos n en cero (buena práctica), verificamos el retorno de scanf (corrige el warning), y usamos 2.0f para forzar división flotante (corrige el bug lógico silencioso). La lección grande: -Wall atrapa muchos problemas, pero no todos; la división entera accidental es un bug que el compilador no siempre detecta y que tú tienes que vigilar. Por eso las buenas prácticas y las pruebas son complementos del compilador, no reemplazos.

Solución del ejercicio 5

Agrega esta función al proyecto final y la opción 6 en el menu:

```

void agenda_editar(Agenda *a) {
    if (a->cantidad == 0) {
        printf("La agenda esta vacia.\n");
        return;
    }
    agenda_mostrar(a);
    printf("Numero del contacto a editar (0 para cancelar): ");
    int num;
    if (scanf("%d", &num) != 1) {
        while (getchar() != '\n');
        printf("Entrada no valida.\n");
        return;
    }
    getchar();
    if (num == 0) {
        printf("Operacion cancelada.\n");
        return;
    }
    if (num < 1 || num > a->cantidad) {
        printf("Numero fuera de rango.\n");
        return;
    }
    Contacto *c = &a->datos[num - 1];
    printf("Editando: %s (telefono actual: %s, email actual: %s)\n",
           c->nombre, c->telefono, c->email);
    leer_texto("Nuevo telefono (enter para no cambiar): ", c->telefono,
               sizeof(c->telefono));
    // si dejo vacio, conservamos el anterior: necesitamos respaldo
    printf("Contacto actualizado.\n");
}

```

Espera: hay un detalle. Si el usuario deja el teléfono vacío para no cambiarlo, leer_texto ya sobrescribio el valor original con una cadena vacía. La versión correcta respalda primero:

```

void agenda_editar(Agenda *a) {
    if (a->cantidad == 0) {
        printf("La agenda esta vacia.\n");
        return;
    }
    agenda_mostrar(a);
    printf("Numero del contacto a editar (0 para cancelar): ");
    int num;
    if (scanf("%d", &num) != 1) {
        while (getchar() != '\n');
        printf("Entrada no valida.\n");
        return;
    }
    getchar();
    if (num == 0) {
        printf("Operacion cancelada.\n");
        return;
    }
    if (num < 1 || num > a->cantidad) {
        printf("Numero fuera de rango.\n");
        return;
    }
    Contacto *c = &a->datos[num - 1];
    char nuevo[100];
    printf("Editando a: %s\n", c->nombre);
    leer_texto("Nuevo telefono (enter para conservar el actual): ",
               nuevo, sizeof(nuevo));
    if (strlen(nuevo) > 0) {
        strcpy(c->telefono, nuevo);
    }
    leer_texto("Nuevo email (enter para conservar el actual): ",
               nuevo, sizeof(nuevo));
    if (strlen(nuevo) > 0) {
        strcpy(c->email, nuevo);
    }
    printf("Contacto actualizado. No olvides guardar (opcion 5).\n");
}

```

Explicación: leemos el dato nuevo en un bufer temporal y solo lo copiamos al contacto si el usuario escribió algo. Así "enter" significa conservar el valor anterior, que es el comportamiento que los usuarios esperan. Validamos el número igual que en eliminar: rango, cero para cancelar, entrada no numerica. Y avisamos que hay que guardar, porque los cambios viven en memoria hasta que se escriben al archivo. En el menu agregas case 6: agenda_editar(&agenda); break; y la línea "6. Editar contacto".

Solución del ejercicio 6

El sistema de inventario reutiliza toda la arquitectura de la agenda; solo cambian los datos y un par de funciones. Aquí esta el programa completo:

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

#define NOMBRE_ARCHIVO "inventario.dat"
#define CAPACIDAD_INICIAL 10

typedef struct {
    char nombre[100];
    int cantidad;
    float precio;
} Producto;

typedef struct {
    Producto *datos;
    int total;
    int capacidad;
} Inventario;

void inv_init(Inventario *inv);
void inv_liberar(Inventario *inv);
int inv_crecer(Inventario *inv);
void inv_agregar(Inventario *inv);
void inv_mostrar(const Inventario *inv);
void inv_buscar(const Inventario *inv);
void inv_eliminar(Inventario *inv);
void inv_valor_total(const Inventario *inv);
int inv_guardar(const Inventario *inv);
int inv_cargar(Inventario *inv);
void limpiar_linea(char *t);
void leer_texto(const char *msg, char *dest, int tam);
```

```

int main() {
    Inventario inv;
    inv_init(&inv);
    if (!inv_crecer(&inv)) {
        printf("Error de memoria.\n");
        return 1;
    }
    printf("Productos cargados: %d\n", inv_cargar(&inv));

    int opcion;
    do {
        printf("\n=== INVENTARIO DE TIENDA ===\n");
        printf("Productos: %d\n", inv.total);
        printf("1. Agregar producto\n");
        printf("2. Mostrar inventario\n");
        printf("3. Buscar producto\n");
        printf("4. Eliminar producto\n");
        printf("5. Valor total del inventario\n");
        printf("6. Guardar\n");
        printf("0. Salir\n");
        printf("Opcion: ");
        if (scanf("%d", &opcion) != 1) {
            opcion = -1;
            while (getchar() != '\n');
        }
        getchar();
        switch (opcion) {
            case 1: inv_agregar(&inv); break;
            case 2: inv_mostrar(&inv); break;
            case 3: inv_buscar(&inv); break;
            case 4: inv_eliminar(&inv); break;
            case 5: inv_valor_total(&inv); break;
            case 6:
                if (inv_guardar(&inv)) printf("Inventario guardado.\n");
                else printf("Error al guardar.\n");
                break;
            case 0: printf("Guardando y saliendo...\n"); break;
            default: printf("Opcion no valida.\n");
        }
    } while (opcion != 0);

    inv_guardar(&inv);
    inv_liberar(&inv);
    return 0;
}

```

```

void inv_init(Inventario *inv) {
    inv->datos = NULL;
    inv->total = 0;
    inv->capacidad = 0;
}

void inv_liberar(Inventario *inv) {
    free(inv->datos);
    inv->datos = NULL;
    inv->total = 0;
    inv->capacidad = 0;
}

int inv_crecer(Inventario *inv) {
    int nueva = (inv->capacidad == 0) ? CAPACIDAD_INICIAL : inv->capacidad * 2;
    Producto *p = realloc(inv->datos, nueva * sizeof(Producto));
    if (p == NULL) return 0;
    inv->datos = p;
    inv->capacidad = nueva;
    return 1;
}

void limpiar_linea(char *t) { t[strcspn(t, "\n")] = '\0'; }

void leer_texto(const char *msg, char *dest, int tam) {
    printf("%s", msg);
    fgets(dest, tam, stdin);
    limpiar_linea(dest);
}

void inv_agregar(Inventario *inv) {
    if (inv->total >= inv->capacidad && !inv_crecer(inv)) {
        printf("Error: no se pudo ampliar el inventario.\n");
        return;
    }
    Producto *p = &inv->datos[inv->total];
    leer_texto("Nombre del producto: ", p->nombre, sizeof(p->nombre));
    if (strlen(p->nombre) == 0) {
        printf("El nombre no puede estar vacio.\n");
        return;
    }
    printf("Cantidad en existencia: ");
    if (scanf("%d", &p->cantidad) != 1 || p->cantidad < 0) {
        while (getchar() != '\n');
    }
}

```



```

        printf("Cantidad no valida.\n");
        return;
    }
    printf("Precio unitario: ");
    if (scanf("%f", &p->precio) != 1 || p->precio < 0) {
        while (getchar() != '\n');
        printf("Precio no valido.\n");
        return;
    }
    getchar();
    inv->total++;
    printf("Producto agregado.\n");
}

void inv_mostrar(const Inventario *inv) {
    if (inv->total == 0) {
        printf("El inventario esta vacio.\n");
        return;
    }
    printf("\n--- INVENTARIO (%d productos) ---\n", inv->total);
    for (int i = 0; i < inv->total; i++) {
        printf("%d. %s | Cant: %d | Precio: $%.2f | Subtotal: $%.2f\n",
            i + 1, inv->datos[i].nombre, inv->datos[i].cantidad,
            inv->datos[i].precio,
            inv->datos[i].cantidad * inv->datos[i].precio);
    }
}

void inv_buscar(const Inventario *inv) {
    char b[100];
    leer_texto("Producto a buscar: ", b, sizeof(b));
    int n = 0;
    for (int i = 0; i < inv->total; i++) {
        if (strstr(inv->datos[i].nombre, b) != NULL) {
            printf("%d. %s | Cant: %d | Precio: $%.2f\n",
                i + 1, inv->datos[i].nombre,
                inv->datos[i].cantidad, inv->datos[i].precio);
            n++;
        }
    }
    if (n == 0) printf("Sin coincidencias.\n");
}

void inv_eliminar(Inventario *inv) {
    if (inv->total == 0) {

```

```

        printf("Inventario vacio.\n");
        return;
    }
    inv_mostrar(inv);
    printf("Numero a eliminar (0 cancela): ");
    int num;
    if (scanf("%d", &num) != 1) {
        while (getchar() != '\n');
        printf("Entrada no valida.\n");
        return;
    }
    getchar();
    if (num == 0) { printf("Cancelado.\n"); return; }
    if (num < 1 || num > inv->total) {
        printf("Fuera de rango.\n");
        return;
    }
    for (int i = num - 1; i < inv->total - 1; i++) {
        inv->datos[i] = inv->datos[i + 1];
    }
    inv->total--;
    printf("Producto eliminado.\n");
}

void inv_valor_total(const Inventario *inv) {
    float total = 0;
    for (int i = 0; i < inv->total; i++) {
        total += inv->datos[i].cantidad * inv->datos[i].precio;
    }
    printf("Valor total del inventario: $%.2f\n", total);
}

int inv_guardar(const Inventario *inv) {
    FILE *f = fopen(NOMBRE_ARCHIVO, "w");
    if (f == NULL) return 0;
    for (int i = 0; i < inv->total; i++) {
        fprintf(f, "%s|%d|%.2f\n",
            inv->datos[i].nombre,
            inv->datos[i].cantidad,
            inv->datos[i].precio);
    }
    fclose(f);
    return 1;
}

```

```

int inv_cargar(Inventario *inv) {
    FILE *f = fopen(NOMBRE_ARCHIVO, "r");
    if (f == NULL) return 0;
    char linea[256];
    int n = 0;
    while (fgets(linea, sizeof(linea), f) != NULL) {
        if (inv->total >= inv->capacidad && !inv_creer(inv)) break;
        Producto *p = &inv->datos[inv->total];
        char *c = strtok(linea, "|");
        if (c == NULL) continue;
        strncpy(p->nombre, c, sizeof(p->nombre) - 1);
        p->nombre[sizeof(p->nombre) - 1] = '\0';
        c = strtok(NULL, "|");
        if (c == NULL) continue;
        p->cantidad = atoi(c);
        c = strtok(NULL, "|\n");
        if (c == NULL) continue;
        p->precio = atof(c);
        inv->total++;
        n++;
    }
    fclose(f);
    return n;
}

```

Explicación: la estructura es idéntica a la agenda porque el patrón es el mismo: una colección dinámica de structs con persistencia en archivo. Lo único que cambió fue el struct (nombre, cantidad, precio en lugar de nombre, teléfono, email) y las funciones que dependen de esos campos. La función nueva es `inv_valor_total`, que recorre el inventario sumando `cantidad * precio`: es el reporte que el dueño de la tienda consultaría cada mañana. Las validaciones rechazan cantidades y precios negativos, porque un inventario con -5 manzanas no tiene sentido. Si entendiste la agenda, este programa te tomó minutos adaptarlo: esa es la magia de diseñar con patrones reutilizables. Compíllalo con `gcc -Wall -Wextra inventario.c -o inventario` y comprueba el cero warnings.

Resumen de la unidad

Llegamos al final. Vamos a repasar todo lo que conquistaste en esta unidad, porque es mucho:

En archivos aprendiste que los datos de la RAM se pierden al apagar la computadora y que los archivos en disco son la respuesta. Conoces el tipo `FILE *` como el control remoto del archivo, y el ciclo sagrado de cuatro pasos: abrir con `fopen`, verificar que no sea `NULL`, leer o escribir, y

cerrar siempre con `fclose`. Dominas los modos "r" para leer, "w" para crear o reemplazar, y "a" para agregar sin borrar. Sabes que `fprintf` y `fscanf` son los primos de `printf` y `scanf` que trabajan con archivos, que `fgets` lee líneas completas y `fputs` las escribe, y que `fseek` y `rewind` controlan la posición del cursor. Construiste una agenda que guarda contactos en disco y un cargador de structs que reconstruye registros desde texto con `strtok`, `atoi` y `atof`.

En buenas prácticas aprendiste a compilar como profesional con `gcc -Wall -Wextra` y la regla de oro de cero warnings. Aprendiste a leer los mensajes del compilador empezando por el primer error, a poner nombres claros que documentan solos, a comentar el porqué y no el que, a escribir funciones cortas que hacen una sola cosa, a probar los casos borde con datos feos, y a inicializar siempre tus variables. En depuración aprendiste el método del detective: reproducir el error, leer los mensajes, dividir y vencer, usar `printf` estratégicos, verificar tus suposiciones, cambiar una sola cosa a la vez, y descansar cuando te atoras. Cazamos juntos el bug del off by one con `i <= n` y viste como los `printf` te mostraron al culpable en la iteración exacta.

En el proyecto final diseñaste como profesional antes de programar: datos, operaciones, persistencia, memoria, casos borde y esqueleto compilable. Construiste una agenda de contactos completa y funcional con menu, altas, bajas, búsqueda, edición, memoria dinámica que se duplica con `realloc`, y guardado en archivo con formato `nombre|teléfono|email`. La adaptaste a un sistema de inventario con valor total, le agregaste edición de contactos y validación de duplicados. Cada programa compila sin warnings y maneja sus errores con elegancia.

Y aquí va el cierre, de programador a programador: terminaste un curso completo de C de 30 clases. Eso no es poca cosa. C es el lenguaje que esta debajo de todo: los sistemas operativos, las bases de datos, los videojuegos, los microcontroladores y hasta los interpretes de otros lenguajes están escritos en C o en sus descendientes. Entender C es entender como piensa realmente una computadora: la memoria, los apuntadores, el manejo manual de recursos. Esa comprensión te acompaña a cualquier lenguaje que aprendas después, porque ahora sabes lo que pasa debajo del cofre.

Que sigue después de C. Tienes varios caminos, y todos son buenos. El más natural es C++: es C con superpoderes (programación orientada a objetos, plantillas, la biblioteca estándar), y como ya dominas C, la curva es suave. Otro camino es estructuras de datos y algoritmos: listas ligadas, árboles, grafos, tablas hash y ordenamientos, implementados por ti en C, que es la mejor forma de entenderlos a fondo. Si te atrae el hardware, los sistemas embebidos te esperan: programar microcontroladores en C para robots, sensores y dispositivos del internet de las cosas. Y si te gusta saber cómo funcionan las cosas por dentro, los sistemas operativos y el desarrollo de compiladores son territorio C por excelencia.

Mi recomendación: sigue construyendo. El proyecto final no es el final de tu práctica, es el principio. Agregale a tu agenda una función de respaldo automatico, un registro de cambios con fecha y hora, o una versión que guarde en formato binario con fwrite y fread. Cada mejora te ensena algo nuevo. Y cuando estes listo para el siguiente paso, en lucasegaacademy.com te esperan los siguientes cursos del camino: C++, estructuras de datos y mucho más, con la misma filosofia de este libro, aprender haciendo, con código real desde la primera clase.

Gracias por llegar hasta aquí. Empezaste sin saber que era un compilador y terminas con un sistema completo que guarda datos en disco, maneja memoria dinámica y compila sin warnings. Eso lo hiciste tu, línea por línea. El pizarron se queda encendido y la puerta abierta: nos vemos en lucasegaacademy.com para lo que sigue. A programar se ha dicho.

EXAMEN FINAL

Este es tu momento de demostrar todo lo que aprendiste en el Curso Completo de C de Lucas EGA Academy. Resuelve las 50 preguntas sin ver las respuestas: primero las de opción múltiple, después las de verdadero o falso y al final las de código. Trabaja a tu ritmo, pero hazlo honesto, sin copiar y sin consultar las unidades del libro, porque este examen mide lo que de verdad se te quedó en la cabeza.

Cuando termines, compara con la sección de respuestas y soluciones y cuenta tu puntaje: 90 puntos o más significa que dominas C de principio a fin; entre 70 y 89 estas sólido pero debes repasar las unidades donde fallaste; menos de 70 significa que conviene volver a leer las unidades con calma antes de seguir adelante. Cada pregunta vale 2 puntos, así que el máximo es 100. Anota tus errores en una libreta, porque ahí están tus puntos débiles y ahí es donde debes atacar.

Preguntas

PREGUNTA 1:

C es un lenguaje que nació en los laboratorios Bell a principios de los años 70. Quién fue su creador principal? a) Bjarne Stroustrup b) Dennis Ritchie c) Linus Torvalds d) Brian Kernighan

PREGUNTA 2:

Tienes un archivo llamado programa.c y quieres compilarlo con gcc para obtener un ejecutable llamado programa. Cuál comando es el correcto? a) gcc programa.c b) gcc -o programa programa.c c) gcc -run programa.c d) compile programa.c -o programa

PREGUNTA 3:

En la función printf, que especificador de formato se usa para imprimir un número entero con signo? a) %f b) %c c) %d d) %s

PREGUNTA 4:

Que palabra reservada se usa en C para declarar una constante simbólica con el preprocesador, de modo que PI valga 3.14 en todo el programa? a) const PI = 3.14; b) #define PI 3.14 c) final PI = 3.14; d) static PI 3.14

PREGUNTA 5:

En el siguiente fragmento, que valor se imprime?

```
int a = 5, b = 2;
printf("%d", a / b);
```

a) 2.5 b) 2 c) 3 d) 0

PREGUNTA 6:

Para leer un entero desde el teclado y guardarlo en la variable edad, cuál llamada a scanf es correcta? a) scanf("%d", edad); b) scanf("%d", &edad); c) scanf(edad, "%d"); d) scanf("&d", edad);

PREGUNTA 7:

Que operador lógico en C significa "Y" (AND), es decir, solo es verdadero si ambas condiciones son verdaderas? a) || b) ! c) && d) ==

PREGUNTA 8:

Cuál es la diferencia entre = y == en C? a) No hay diferencia, se pueden usar indistintamente b) = asigna un valor y == compara dos valores c) == asigna un valor y = compara dos valores d) = solo sirve para textos y == solo para números

PREGUNTA 9:

Observa este código:

```
int x = 10;
if (x > 5)
    printf("A");
else if (x > 8)
    printf("B");
else
    printf("C");
```

Que se imprime? a) A b) B c) C d) AB

PREGUNTA 10:

Cuál es la salida del siguiente programa?

```
int i;
for (i = 0; i < 3; i++)
    printf("%d ", i);
```

a) 1 2 3 b) 0 1 2 c) 0 1 2 3 d) 1 2

PREGUNTA 11:

En un bucle while, que hace la instrucción break? a) Salta a la siguiente iteración sin ejecutar el resto del cuerpo b) Termina el bucle por completo y continua con lo que sigue c) Reinicia el bucle desde el principio d) Pausa el programa hasta que el usuario presione una tecla

PREGUNTA 12:

Cuál es la forma correcta de declarar una función en C que recibe dos enteros y regresa su suma como entero? a) `int suma(int a, int b);` b) `void suma(int a, int b);` c) `suma(int a, int b) int;` d) `function int suma(int a, int b);`

PREGUNTA 13:

En C, las variables declaradas dentro de una función tienen un alcance local. Qué pasa si intentas usar una variable local fuera de su función? a) El programa la crea automáticamente con valor cero b) Funciona sin problema porque todas las variables son globales c) El compilador marca error porque la variable no existe en ese alcance d) Toma el valor de otra variable con el mismo nombre

PREGUNTA 14:

Cómo se declara correctamente un arreglo de 10 enteros en C? a) `int arr[10];` b) `array int arr = 10;` c) `int arr = new int[10];` d) `int arr(10);`

PREGUNTA 15:

En C, las cadenas de texto son arreglos de caracteres que terminan con un carácter especial.Cuál es ese carácter? a) `\n` b) `\0` c) `\t` d) `\e`

PREGUNTA 16:

Si `int *p;` declara un apuntador a entero, que significa la expresión `*p` cuando `p` ya apunta a una variable válida? a) La dirección de memoria donde esta `p` b) El valor almacenado en la variable a la que apunta `p` c) El tamaño en bytes del apuntador d) Un error de compilación

PREGUNTA 17:

Que operador se usa para obtener la dirección de memoria de una variable en C? a) `*` b) `&` c) `->` d) `#`

PREGUNTA 18:

Cuál es la forma correcta de definir un struct llamado Punto con dos miembros enteros x e y?
a) struct Punto { int x; int y; }; b) struct Punto(int x, int y); c) Punto = { int x; int y; }; d) struct { Punto x; Punto y; };

PREGUNTA 19:

Para reservar memoria dinámica para 20 enteros durante la ejecución del programa, cuál llamada es correcta? a) int *p = malloc(20 * sizeof(int)); b) int *p = new int[20]; c) int *p = alloc(20); d) int p[20] = malloc();

PREGUNTA 20:

Quieres abrir un archivo de texto llamado datos.txt para lectura. Cuál llamada a fopen es correcta? a) fopen("datos.txt", "r"); b) fopen("datos.txt", "w"); c) open("datos.txt", "lectura"); d) fopen("r", "datos.txt");

PREGUNTA 21:

En C, toda sentencia termina con punto y coma, y si lo olvidas el compilador marca error.

PREGUNTA 22:

Una variable de tipo float puede guardar exactamente cualquier número decimal sin pérdida de precisión.

PREGUNTA 23:

El siguiente código imprime los números del 1 al 5:

```
int n = 1;
while (n <= 5) {
    printf("%d ", n);
    n++;
}
```

PREGUNTA 24:

En C puedes declarar una función después de usarla en main sin ningún prototipo previo y el programa compila sin advertencias.

PREGUNTA 25:

Los índices de un arreglo en C empiezan en 0, así que el primer elemento de int v[5] es v[0].

PREGUNTA 26:

La función `strlen` cuenta los caracteres de una cadena sin incluir el carácter nulo final.

PREGUNTA 27:

Si `p` es un apuntador a `int`, la expresión `p++` avanza el apuntador al siguiente entero en memoria, no un solo byte.

PREGUNTA 28:

Un apuntador que no ha sido inicializado puede usarse sin problema porque C lo pone automáticamente en `NULL`.

PREGUNTA 29:

Los miembros de un `struct` se acceden con el operador punto cuando tienes la variable directa, y con flecha (`->`) cuando tienes un apuntador al `struct`.

PREGUNTA 30:

La memoria reservada con `malloc` se libera sola cuando termina la función que la reservo.

PREGUNTA 31:

Después de abrir un archivo con `fopen`, es buena práctica cerrarlo con `fclose` cuando ya no lo necesitas.

PREGUNTA 32:

El modo "`w`" en `fopen` conserva el contenido previo del archivo y escribe al final de él.

PREGUNTA 33:

En C, el operador `%` calcula el residuo de una división entre enteros.

PREGUNTA 34:

Una función recursiva es aquella que se llama a si misma, y siempre debe tener un caso base para no repetirse hasta el infinito.

PREGUNTA 35:

El preprocesador (`#include`, `#define`) se ejecuta antes de la compilación y no genera código máquina por si mismo.

PREGUNTA 36:

Predice la salida exacta del siguiente programa:

```
#include <stdio.h>
int main() {
    int a = 7;
    float b = 2.0;
    printf("%d %d", a / 2, (int)(a / b));
    return 0;
}
```

PREGUNTA 37:

Encuentra el error en este programa y escribe la versión corregida:

```
#include <stdio.h>
int main() {
    int edad;
    printf("Dame tu edad: ");
    scanf("%d", edad);
    printf("Tienes %d años", edad);
    return 0;
}
```

PREGUNTA 38:

Escribe un programa completo que pida al usuario un número entero y diga si es par o impar usando el operador %.

PREGUNTA 39:

Predice la salida exacta:

```
#include <stdio.h>
int main() {
    int i = 5;
    do {
        printf("%d ", i);
        i -= 2;
    } while (i > 0);
    return 0;
}
```

PREGUNTA 40:

Encuentra el error lógico en este bucle y corrígelo. El programa debe sumar los números del 1 al 10:

```
#include <stdio.h>
int main() {
    int suma = 0, i;
    for (i = 1; i <= 10; i++);
        suma += i;
    printf("%d", suma);
    return 0;
}
```

PREGUNTA 41:

Escribe una función llamada mayor que reciba tres enteros y regrese el mayor de los tres. Incluye un main pequeño que la pruebe con los valores 4, 9 y 2.

PREGUNTA 42:

Predice la salida exacta:

```
#include <stdio.h>
void cambiar(int x) {
    x = 100;
}
int main() {
    int n = 5;
    cambiar(n);
    printf("%d", n);
    return 0;
}
```

Explica por que el valor no cambia.

PREGUNTA 43:

Escribe un programa que llene un arreglo de 5 enteros con valores leídos del teclado y después imprima el promedio de esos valores con dos decimales.

PREGUNTA 44:

Predice la salida exacta:

```
#include <stdio.h>
#include <string.h>
int main() {
    char texto[] = "Hola";
    printf("%d %d", strlen(texto), sizeof(texto));
    return 0;
}
```

PREGUNTA 45:

Encuentra el error y corrígelo:

```
#include <stdio.h>
int main() {
    int a = 10;
    int *p;
    *p = a;
    printf("%d", *p);
    return 0;
}
```

PREGUNTA 46:

Predice la salida exacta:

```
#include <stdio.h>
int main() {
    int v[] = {10, 20, 30, 40};
    int *p = v;
    printf("%d %d", *(p + 2), p[3]);
    return 0;
}
```

PREGUNTA 47:

Escribe un programa que defina un struct Alumno con nombre (cadena de 30 caracteres) y calificación (float), cree un arreglo de 3 alumnos, lea sus datos con scanf y después imprima el nombre del alumno con la calificación más alta.

PREGUNTA 48:

Encuentra los dos errores en este programa que usa memoria dinámica y escribe la versión corregida:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *datos = malloc(5 * sizeof(int));
    int i;
    for (i = 0; i < 5; i++)
        datos[i] = i * 10;
    for (i = 0; i < 5; i++)
        printf("%d ", datos[i]);
    return 0;
}
```

PREGUNTA 49:

Escribe un programa que abra el archivo "notas.txt" en modo lectura, lea enteros uno por uno hasta el fin del archivo e imprima cuántos números leyo y la suma de todos ellos. Si el archivo no existe, debe mostrar un mensaje de error.

PREGUNTA 50:

Este programa debería copiar una cadena en otra usando apuntadores, pero tiene errores. Encuéntralos y escribe la versión corregida:

```
#include <stdio.h>
void copiar(char *dest, char *orig) {
    while (*orig != '\0') {
        *dest = *orig;
        dest++;
        orig++;
    }
}
int main() {
    char a[] = "Programar";
    char b[20];
    copiar(b, a);
    printf("%s", b);
    return 0;
}
```

Respuestas y soluciones

RESPUESTA 1: b

La respuesta correcta es b) Dennis Ritchie, quien creó C en los laboratorios Bell entre 1972 y 1973 como evolución del lenguaje B. La opción a) es incorrecta porque Bjarne Stroustrup creó C++, que es un lenguaje derivado de C pero posterior. La opción c) es incorrecta porque Linus Torvalds creó el núcleo Linux, escrito en C pero no es su creador. La opción d) es incorrecta porque Brian Kernighan fue coautor del famoso libro de C junto a Ritchie, pero no el creador del lenguaje.

Dato extra de la unidad 1: Ritchie escribió junto a Kernighan el libro *The C Programming Language*, conocido como K&R, que por décadas fue la biblia del lenguaje. Casi todos los lenguajes modernos que usas hoy, como C++, Java, Python y JavaScript, heredaron su sintaxis de C, así que aprenderlo te da ventaja en todos los demás.

RESPUESTA 2: b

La respuesta correcta es b) `gcc -o programa programa.c`, porque la bandera `-o` indica el nombre del archivo ejecutable de salida. La opción a) también compila, pero genera el ejecutable con el nombre por defecto `a.out`, no con el nombre `programa` que se pide. La opción c) es incorrecta porque no existe una bandera `-run` en `gcc`. La opción d) es incorrecta porque `compile` no es un comando del compilador de C.

Tip de la unidad 1: acostúmbrate a compilar siempre con `gcc -Wall -o programa programa.c`. La bandera `-Wall` activa todas las advertencias y te avisa de errores lógicos comunes antes de que se conviertan en fallos raros en ejecución. Un programador que compila sin advertencias duerme tranquilo.

RESPUESTA 3: c

La respuesta correcta es c) `%d`, que es el especificador para enteros con signo en formato decimal. La opción a) `%f` es incorrecta porque se usa para números de punto flotante como `float` y `double`. La opción b) `%c` es incorrecta porque imprime un solo carácter. La opción d) `%s` es incorrecta porque se usa para cadenas de texto terminadas en nulo.

Para no confundirte, memoriza la familia completa: `%d` enteros, `%f` flotantes, `%c` caracteres, `%s` cadenas y `%p` direcciones de memoria. Si usas el especificador equivocado, `printf` lee los bytes con el tipo incorrecto e imprime basura. El formato y el tipo de dato siempre deben ir de la mano.

RESPUESTA 4: b

La respuesta correcta es b) `#define PI 3.14`, que es la directiva del preprocesador para crear una constante simbólica: antes de compilar, cada aparición de `PI` se reemplaza por `3.14`. La opción a) es incorrecta porque `const` declara una variable de solo lectura, no una constante del

preprocesador, y además esa es otra técnica válida pero no es lo que pregunta el enunciado. La opción c) es incorrecta porque `final` no existe en C, es palabra de Java. La opción d) es incorrecta porque `static` es un especificador de almacenamiento y esa sintaxis ni siquiera compila.

Ojo con un detalle tramposo: `#define` no lleva punto y coma al final porque no es una sentencia de C sino una orden del preprocesador. Si escribes `#define PI 3.14`; el punto y coma se pega al reemplazo y te genera errores extraños. La alternativa moderna es `const float PI = 3.14`;, que si es una variable de solo lectura con tipo.

RESPUESTA 5: b

La respuesta correcta es b) 2, porque al dividir dos enteros en C el resultado es entero y la parte decimal se trunca, no se redondea: `5 / 2` da 2. La opción a) 2.5 es incorrecta porque ese sería el resultado si al menos uno de los operandos fuera flotante. La opción c) 3 es incorrecta porque C no redondea hacia arriba en división entera. La opción d) 0 es incorrecta porque la división si se realiza y da un cociente de 2.

Si necesitas el resultado decimal, convierte uno de los operandos a flotante con un cast: `(float)a / b` daría 3.5. Esta conversión se llama promoción de tipos y es una de las herramientas más usadas en la unidad 2. Acuerdate de la regla de oro: en C, el tipo del resultado lo deciden los operandos, nunca lo que esperas obtener.

RESPUESTA 6: b

La respuesta correcta es b) `scanf("%d", &edad)`;, porque `scanf` necesita la dirección de memoria donde guardar el dato leído, y el operador `&` la proporciona. La opción a) es incorrecta porque pasa el valor de edad en lugar de su dirección, lo que provoca comportamiento indefinido y casi seguro un fallo. La opción c) es incorrecta porque invierte el orden de los argumentos: primero va la cadena de formato. La opción d) es incorrecta porque `"&d"` no es un especificador válido y además falta el `&` en la variable.

Este es el error que más programas rompe en la unidad 3: olvidar el `&` hace que `scanf` escriba en una dirección de memoria inválida y el sistema operativo mata tu programa con un fallo de segmentación. Cada vez que tu programa se cierre de golpe después de un `scanf`, lo primero que debes revisar es si todos los argumentos llevan su `&`.

RESPUESTA 7: c

La respuesta correcta es c) `&&`, que es el operador AND lógico: el resultado solo es verdadero cuando ambos operandos son verdaderos. La opción a) `||` es incorrecta porque es el OR lógico, verdadero si al menos uno lo es. La opción b) `!` es incorrecta porque es la negación

lógica, invierte un solo valor. La opción d) == es incorrecta porque es el operador de comparación de igualdad, no un operador lógico.

El && tiene una propiedad elegante llamada cortocircuito: si la primera condición es falsa, la segunda ni se evalúa porque el resultado ya se sabe. Esto sirve para escribir verificaciones seguras como `if (p != NULL && *p > 0)`, donde la segunda parte solo se ejecuta si el apuntador es válido. El || hace lo mismo pero al revés.

RESPUESTA 8: b

La respuesta correcta es b) = asigna un valor y == compara dos valores. Confundirlos es uno de los errores más comunes: escribir `if (x = 5)` asigna 5 a x en lugar de comparar. La opción a) es incorrecta porque tienen significados totalmente distintos y no son intercambiables. La opción c) invierte los papeles y por eso es incorrecta. La opción d) es incorrecta porque ambos operadores trabajan con cualquier tipo de dato, no están limitados a textos o números.

Un truco de programadores experimentados para evitar este error es la condición Yoda: escribir `if (5 == x)` en lugar de `if (x == 5)`. Si por accidente pones un solo =, el compilador marca error porque no puedes asignar a una constante, y el bug se detecta solo. Con la práctica dejarás de necesitarlo, pero al inicio salva vidas.

RESPUESTA 9: a

La respuesta correcta es a) A, porque x vale 10 y la primera condición `x > 5` ya es verdadera, así que se ejecuta su bloque y se saltan todos los `else if` y `else` siguientes. La opción b) es incorrecta porque aunque `x > 8` también sería verdadero, esa rama nunca se evalúa al haberse cumplido la primera. La opción c) es incorrecta porque el `else` solo se ejecuta si ninguna condición anterior se cumple. La opción d) es incorrecta porque en una cadena `if-else if-else` solo se ejecuta una rama, nunca varias.

La cadena `if-else if-else` evalúa las condiciones en orden y ejecuta solo la primera que sea verdadera, saltándose todo lo demás. Si necesitas que varias condiciones se revisen de forma independiente, usa `ifs` separados en lugar de `else if`. Esta diferencia entre exclusión mutua y revisión independiente aparece en casi todos los programas reales.

RESPUESTA 10: b

La respuesta correcta es b) 0 1 2, porque el bucle empieza con i en 0, se repite mientras `i < 3` e incrementa de uno en uno, imprimiendo 0, 1 y 2. La opción a) es incorrecta porque el contador empieza en 0, no en 1. La opción c) es incorrecta porque cuando i llega a 3 la condición `i < 3` es falsa y el bucle termina antes de imprimir el 3. La opción d) es incorrecta porque omite el 0 inicial.

El patrón for ($i = 0; i < n; i++$) que recorre n elementos es el más usado en C porque combina perfecto con los arreglos, cuyos índices van de 0 a $n-1$. El error clásico es escribir $i \leq n$ y leer un elemento de más, provocando el famoso error de uno de más (off-by-one) que rompe programas enteros.

RESPUESTA 11: b

La respuesta correcta es b) break termina el bucle por completo y el programa continua con la primera instrucción después del bucle. La opción a) describe a continue, no a break, por eso es incorrecta. La opción c) es incorrecta porque break no reinicia nada, sale del bucle. La opción d) es incorrecta porque break no interactúa con el usuario ni pausa el programa.

No confundas break con continue: break abandona el bucle por completo, mientras que continue salta a la siguiente iteración sin ejecutar el resto del cuerpo actual. Ambos también funcionan en for y en do-while, y break además sirve para salir de un switch. Dominar estas dos palabras te da control total sobre cualquier ciclo.

RESPUESTA 12: a

La respuesta correcta es a) `int suma(int a, int b);`, que declara una función que regresa int y recibe dos parámetros int, con la sintaxis correcta de C. La opción b) es incorrecta porque void indica que no regresa nada, pero se pide que regrese la suma. La opción c) es incorrecta porque la sintaxis está invertida y no compila. La opción d) es incorrecta porque function no es palabra reservada en C.

En C se distingue entre declarar una función (el prototipo, que dice cómo se llama) y definirla (el cuerpo, que dice que hace). El prototipo va arriba del main o en un archivo de encabezado .h, y la definición puede ir abajo. Esta separación es la base para organizar programas grandes en varios archivos.

RESPUESTA 13: c

La respuesta correcta es c) el compilador marca error porque la variable local solo existe dentro del bloque de su función y fuera de ahí ese nombre no está declarado. La opción a) es incorrecta porque C no crea variables automáticamente. La opción b) es incorrecta porque no todas las variables son globales, solo las declaradas fuera de las funciones. La opción d) es incorrecta porque cada variable local es independiente aunque comparta nombre con otra.

Existe un fenómeno llamado sombreado (shadowing): puedes declarar una variable local con el mismo nombre que una global, y dentro de la función la local oculta a la global. Compila sin error pero confunde muchísimo al leer el código. La buena práctica de la unidad 6 es evitar nombres duplicados y mantener las variables globales al mínimo.

RESPUESTA 14: a

La respuesta correcta es a) `int arr[10];`, que declara un arreglo de 10 enteros con la sintaxis propia de C: tipo, nombre y tamaño entre corchetes. La opción b) es incorrecta porque `array` no es palabra reservada y esa sintaxis no existe. La opción c) es incorrecta porque `new` es operador de C++, no de C. La opción d) es incorrecta porque los paréntesis no se usan para declarar arreglos.

C no verifica que tus índices estén dentro del arreglo: si escribes `arr[20]` en un arreglo de 10, el compilador no se queja y el programa lee o escribe memoria ajena. Esta falta de verificación es lo que hace a C rapidísimo y a la vez peligroso. Tú eres el responsable de cuidar los límites, nadie lo hará por ti.

RESPUESTA 15: b

La respuesta correcta es b) `\0`, el carácter nulo con valor cero, que marca el fin de toda cadena en C y es lo que permite a funciones como `printf` saber dónde termina el texto. La opción a) `\n` es incorrecta porque es el salto de línea. La opción c) `\t` es incorrecta porque es la tabulación. La opción d) `\e` es incorrecta porque ni siquiera es una secuencia de escape estándar en C.

Todas las funciones de cadenas de `string.h` dependen del `\0` para saber dónde termina el texto: `strcpy`, `strcat`, `strcmp` y `printf` con `%s` recorren la memoria hasta encontrarlo. Si una cadena pierde su terminador, estas funciones siguen leyendo basura hasta chocar con algo. Por eso al crear cadenas a mano siempre debes reservar un byte extra para el `\0`.

RESPUESTA 16: b

La respuesta correcta es b) `*p` es el valor guardado en la variable a la que apunta `p`, porque el asterisco como operador unario desreferencia el apuntador. La opción a) es incorrecta porque la dirección donde está `p` sería `&p`, y la dirección a la que apunta es simplemente `p`. La opción c) es incorrecta porque el tamaño se obtiene con `sizeof`, no con `*`. La opción d) es incorrecta porque desreferenciar un apuntador válido es una operación perfectamente legal.

El `*p` no solo sirve para leer: también puedes escribir con `*p = 20;`, lo que modifica la variable original a través del apuntador. Esta capacidad de modificar datos ajenos es lo que hace poderosos a los apuntadores y es la base para que las funciones puedan cambiar variables del `main`, como verás en la unidad 8.

RESPUESTA 17: b

La respuesta correcta es b) `&`, el operador de dirección, que aplicado a una variable regresa su dirección de memoria. La opción a) `*` es incorrecta porque es el operador de desreferencia, hace lo contrario: dado un apuntador obtiene el valor. La opción c) `->` es incorrecta porque se

usa para acceder a miembros de un struct a través de un apuntador. La opción d) # es incorrecta porque se usa en directivas del preprocesador.

No confundas el & de dirección con el & de scanf que ya viste ni con el && lógico: son tres usos distintos del mismo símbolo según el contexto. Además existe el & a nivel de bits para operaciones binarias, que verás en cursos más avanzados. El contexto siempre te dice cuál es cuál: &variable es dirección, a && b es lógica.

RESPUESTA 18: a

La respuesta correcta es a) struct Punto { int x; int y; }, con la palabra struct, el nombre, los miembros entre llaves y el punto y coma final obligatorio. La opción b) es incorrecta porque usa paréntesis como si fuera una función. La opción c) es incorrecta porque esa sintaxis de asignación no define un tipo. La opción d) es incorrecta porque declara miembros de tipo Punto dentro del propio struct sin haberlo definido, lo cual no compila.

Un truco muy usado es combinar struct con typedef para no escribir la palabra struct cada vez: typedef struct { int x; int y; } Punto; te permite declarar variables simplemente como Punto p;. Es azucar sintáctica que hace el código más limpio y la verás en casi todo código C profesional.

RESPUESTA 19: a

La respuesta correcta es a) int *p = malloc(20 * sizeof(int));, que pide al sistema memoria para 20 enteros y regresa un apuntador a esa zona. La opción b) es incorrecta porque new es de C++, no existe en C. La opción c) es incorrecta porque alloc no es una función estándar. La opción d) es incorrecta porque mezcla la declaración de un arreglo estático con malloc de forma inválida y además no indica cuántos bytes reservar.

Memoriza el patrón completo: Tipo *p = malloc(n * sizeof(Tipo));. Usar sizeof en lugar de un número fijo hace tu código portable, porque el tamaño de int puede variar entre sistemas. Y nunca olvides su pareja: cada malloc debe tener su free cuando la memoria ya no se necesite.

RESPUESTA 20: a

La respuesta correcta es a) fopen("datos.txt", "r");, porque el primer argumento es el nombre del archivo y el segundo el modo, siendo "r" el modo de lectura. La opción b) es incorrecta porque "w" abre en modo escritura y borra el contenido previo. La opción c) es incorrecta porque open no es la función estándar con esa firma ni "lectura" un modo válido. La opción d) es incorrecta porque invierte el orden de los argumentos.

Aprende los tres modos básicos: "r" lee sin modificar, "w" escribe borrando todo lo previo, y "a" agrega al final conservando el contenido. Existe también "r+" para leer y escribir. Elegir mal

el modo es una de las formas más rápidas de perder datos, así que piensa dos veces antes de abrir con "w".

RESPUESTA 21: Verdadero

Es verdadero porque en C cada instrucción debe terminar con punto y coma, que es el delimitador de sentencias del lenguaje. Si lo omites, el compilador no sabe dónde termina una instrucción y donde empieza la siguiente, así que marca error de sintaxis. La única excepción son las directivas del preprocesador como `#include`, que no llevan punto y coma, y las llaves de bloques y funciones.

La única excepción importante a la regla del punto y coma son las directivas del preprocesador como `#include` y `#define`, que ocupan su propia línea y no lo llevan. Tampoco lo llevan las llaves que delimitan bloques ni los encabezados de funciones. Con el tiempo lo harás en automático, pero al inicio conviene revisar cada error de sintaxis buscando el ; faltante.

RESPUESTA 22: Falso

Es falso porque float usa 32 bits con precisión limitada, así que muchos decimales no se pueden representar con exactitud y se guardan como aproximaciones. Por ejemplo, $0.1 + 0.2$ en float no da exactamente 0.3 por la forma en que los binarios representan las fracciones decimales. Si necesitas más precisión debes usar double, y aún así ningún tipo flotante representa "cualquier" decimal con exactitud total.

Por la precisión limitada, nunca compares flotantes con `==` directamente: en lugar de `if (a == b)` usa `if (fabs(a - b) < 0.00001)`. Esta técnica del epsilon es estándar en la industria y evita que tu programa falle por diferencias microscópicas de representación. Es un detalle fino que separa al principiante del programador cuidadoso.

RESPUESTA 23: Verdadero

Es verdadero porque el bucle empieza con `n` en 1, la condición `n <= 5` se cumple para 1, 2, 3, 4 y 5, y en cada vuelta imprime el valor actual y luego lo incrementa. Cuando `n` llega a 6 la condición se vuelve falsa y el bucle termina. La salida exacta es "1 2 3 4 5 ", con el incremento dentro del cuerpo garantizando que el bucle no sea infinito.

Todo while necesita que algo dentro del cuerpo acerque la condición a su fin; si olvidas el `n++` el bucle se vuelve infinito y tu programa se congela. Cuando eso pase, no entres en pánico: Ctrl+C lo detiene en la terminal. Los bucles infinitos intencionales como `while(1)` si existen, pero siempre con un break de salida.

RESPUESTA 24: Falso

Es falso porque C exige que una función este declarada antes de usarse: si la llamas en main sin prototipo previo, los compiladores modernos marcan al menos una advertencia y en el estándar C99 en adelante es un error. El prototipo le dice al compilador que tipos recibe y regresa la función para que pueda verificar la llamada. La forma correcta es escribir el prototipo arriba de main o definir la función completa antes de main.

En los compiladores viejos existía la declaración implícita: si usabas una función sin prototipo, C asumía que regresaba int. El estándar C99 eliminó esa permisividad y hoy es un error. La lección es clara: los prototipos no son opcionales ni decoración, son el contrato que le permite al compilador revisar que llamas bien a cada función.

RESPUESTA 25: Falso

Es falso porque en C los índices empiezan en 0, así que el primer elemento de `int v[5]` es `v[0]` y el último es `v[4]`. Usar `v[5]` accede a memoria fuera del arreglo, lo que provoca comportamiento indefinido y es una de las fuentes más comunes de errores difíciles de rastrear. Esta convención viene de que el índice representa el desplazamiento respecto al inicio del arreglo.

La consecuencia práctica es que el último elemento de un arreglo de tamaño `n` siempre es `v[n-1]`. Para recorrerlo completo usa `for (i = 0; i < n; i++)`, nunca con `<=`. Este patrón aparece en cada programa que maneje arreglos, así que grabatelo: índices de 0 a `n-1`, condición con `<` estricto.

RESPUESTA 26: Verdadero

Es verdadero porque `strlen` recorre la cadena contando caracteres hasta encontrar el `\0`, pero no lo incluye en el conteo. Por eso `strlen("Hola")` regresa 4 aunque el arreglo ocupe 5 bytes en memoria. Esta diferencia entre longitud lógica y tamaño físico es clave: `sizeof` si cuenta el byte del carácter nulo.

La diferencia entre `strlen` y `sizeof` es favorita en entrevistas de trabajo: `strlen("Hola")` da 4 pero `sizeof("Hola")` da 5 por el `\0`. Si declaras `char buff[100]` y guardas "Hola", `strlen` da 4 y `sizeof` da 100. Una mide el texto, la otra mide el contenedor. Confundirlas provoca cálculos de memoria incorrectos.

RESPUESTA 27: Verdadero

Es verdadero porque la aritmética de punteros en C avanza en unidades del tipo apuntado: si `p` es `int*` y un `int` ocupa 4 bytes, `p++` suma 4 a la dirección para llegar al siguiente entero completo. El compilador conoce el tamaño del tipo gracias a la declaración del puntero. Esto permite recorrer arreglos con punteros de forma natural, sin calcular desplazamientos en bytes a mano.

El caso especial es `char*`: como un `char` ocupa 1 byte, `p++` si avanza de byte en byte, y por eso los apuntadores a `char` son ideales para recorrer cadenas. En cambio con `double*`, que ocupa 8 bytes, `p++` salta de 8 en 8. La aritmética de apuntadores siempre piensa en elementos, nunca en bytes crudos.

RESPUESTA 28: Falso

Es falso y además peligroso: C no inicializa los apuntadores automáticamente, así que un apuntador sin inicializar contiene basura, una dirección aleatoria. Usarlo para leer o escribir provoca comportamiento indefinido y normalmente un fallo de segmentación. La buena práctica es inicializar todo apuntador en `NULL` si aún no apunta a algo válido, y verificar antes de desreferenciar.

Adopta el hábito de inicializar cada apuntador al declararlo: `int *p = NULL;`. Así, si por error lo usas antes de asignarlo, el fallo es inmediato y obvio en lugar de un comportamiento aleatorio difícil de rastrear. Un `NULL` es un error ruidoso y fácil de encontrar; un apuntador con basura es un error silencioso y traicionero.

RESPUESTA 29: Verdadero

Es verdadero porque son las dos formas de acceso a miembros de un struct: con una variable directa usas `variable.miembro`, y con un apuntador usas `apuntador->miembro`, que es azúcar sintáctica para `(*apuntador).miembro`. Confundirlos es un error típico de compilación que el compilador detecta de inmediato, así que conviene memorizar la pareja punto-variable y flecha-apuntador.

Recuerda que `p->x` es exactamente lo mismo que `(*p).x`: los paréntesis son obligatorios en la segunda forma porque el punto tiene mayor precedencia que el asterisco. La flecha existe para que no tengas que escribir esos paréntesis a cada rato. Cuando veas `->`, piensa automáticamente apuntador a struct.

RESPUESTA 30: Falso

Es falso porque la memoria del heap reservada con `malloc` vive hasta que el programador la libere explícitamente con `free`, sin importar cuántas funciones terminen en el camino. Solo las variables locales de la pila se destruyen al salir de la función. Olvidar el `free` provoca fugas de memoria: el programa va consumiendo RAM que nunca regresa al sistema hasta que termina.

El error gemelo de olvidar el `free` es liberar dos veces la misma memoria (`double free`), que también corrompe el programa. La disciplina es: un `malloc`, un `free`, y después del `free` asigna `NULL` al apuntador para evitar usarlo por accidente. Estas tres reglas previenen el 90 por ciento de los bugs de memoria dinámica.

RESPUESTA 31: Verdadero

Es verdadero porque cada archivo abierto consume recursos del sistema y hay un límite de archivos abiertos simultáneamente. Si no cierras con `fclose`, puedes agotar esos recursos y además los datos escritos en modo escritura podrían no guardarse por completo, ya que `fclose` vacía el buffer al disco. Abrir, trabajar y cerrar es el ciclo de vida correcto de todo archivo.

Además de liberar recursos, `fclose` vacía el buffer de escritura al disco: los datos que escribes con `fprintf` no van directo al archivo sino a un buffer temporal, y solo `fclose` o `fflush` los guardan de verdad. Si tu programa termina de golpe sin cerrar, puedes perder lo último que escribiste.

RESPUESTA 32: Falso

Es falso porque el modo "w" trunca el archivo: borra todo su contenido previo y empieza a escribir desde cero. Si quieres conservar lo que ya hay y agregar al final debes usar el modo "a" (append). Esta es una trampa clásica: abrir con "w" un archivo con datos importantes los destruye de inmediato, así que hay que elegir el modo con cuidado.

El modo "a" tiene una ventaja extra: si el archivo no existe, lo crea automáticamente, igual que "w". Así que "a" es la opción segura cuando quieres agregar registros a un archivo que debe conservar su historial, como un log o una bitacora de calificaciones.

RESPUESTA 33: Verdadero

Es verdadero porque `%` es el operador módulo, que regresa el residuo de la división entera: $7 \% 3$ da 1 porque 7 entre 3 cabe 2 veces y sobra 1. Solo funciona con operandos enteros, no con flotantes. Es muy útil para saber si un número es par ($n \% 2 == 0$), para ciclos repetitivos y para limitar valores dentro de un rango.

Cuidado con operandos negativos: en C el signo del resultado de `%` sigue al dividendo, así que $-7 \% 3$ da -1. Además, `%` solo acepta enteros; para flotantes existe la función `fmod` de `math.h`. El módulo es tan útil que aparece en relojes, juegos, paginación y criptografía básica.

RESPUESTA 34: Verdadero

Es verdadero porque la recursión consiste en que una función se invoque a sí misma con un problema más pequeño, y el caso base es la condición que detiene las llamadas, por ejemplo $n == 0$ en el factorial. Sin caso base, la función se llamaría sin fin hasta agotar la pila y provocar un desbordamiento de pila (stack overflow). Todo algoritmo recursivo necesita tanto el caso base como el paso recursivo que se acerque a él.

Cada llamada recursiva consume un marco en la pila de llamadas, así que una recursión muy profunda puede agotar la pila aunque tenga caso base correcto. Por eso la recursión es

elegante para problemas como factorial o torres de Hanoi, pero para procesar millones de datos un bucle suele ser más seguro y rápido.

RESPUESTA 35: Verdadero

Es verdadero porque el preprocesador corre antes que el compilador: procesa las líneas que empiezan con #, expande `#include` trayendo el contenido de los encabezados y reemplaza las macros de `#define` por su texto. El resultado es un archivo de código fuente "expandido" que luego si se compila a código máquina. Por eso las directivas no terminan en punto y coma: no son sentencias de C, son ordenes para esta etapa previa.

Otro detalle del preprocesador: `#include <stdio.h>` busca el encabezado en las carpetas del sistema, mientras que `#include "misfunciones.h"` busca primero en tu carpeta de proyecto. Los corchetes angulares son para librerías estándar, las comillas para tus propios archivos. Así se organizan los programas grandes en varios módulos.

RESPUESTA 36: 3 3

La salida es "3 3" porque en `a / 2` ambos operandos son enteros (7 y 2), así que la división es entera y trunca el .5, dando 3. En la segunda expresión, `a / b` divide un int entre un float, así que `a` se promueve a float y el resultado es 3.5; el cast `(int)` trunca la parte decimal y queda 3. El especificador `%d` imprime ambos como enteros. El error típico aquí es pensar que la primera división da 3.5, pero en C el tipo del resultado lo deciden los operandos, no el formato de impresión.

Este ejercicio junta dos trampas clásicas: la división entera y el cast. Fíjate que `(int)(a / b)` primero divide en flotante y luego trunca, dando 3, pero si el cast estuviera dentro como `(int)a / b` el resultado sería el mismo aquí por coincidencia. La lección general: los paréntesis y el orden de las conversiones cambian el resultado, así que escribe las expresiones paso a paso cuando dudes.

RESPUESTA 37:

El error es que a `scanf` le falta el operador `&` en la variable `edad`: sin la dirección de memoria, `scanf` escribe en una dirección inválida y el programa falla. Versión corregida:

```
#include <stdio.h>
int main() {
    int edad;
    printf("Dame tu edad: ");
    scanf("%d", &edad);
    printf("Tienes %d años", edad);
    return 0;
}
```

La explicación es sencilla pero fundamental: `scanf` no recibe variables, recibe direcciones donde guardar lo que lee del teclado. El `&` convierte la variable `edad` en su dirección. Este es el error número uno de los principiantes en la unidad 3, porque `printf` si recibe valores directos y la asimetría confunde. Acuerdate: `printf` imprime valores, `scanf` necesita direcciones.

Un programador cuidadoso además revisa lo que `scanf` regresa: `scanf("%d", &edad)` regresa 1 si leyó un entero correctamente. Puedes usar ese valor para validar la entrada con `if (scanf("%d", &edad) != 1)`. Validar la entrada del usuario es la diferencia entre un programa de tarea y un programa robusto que no se rompe con datos raros.

RESPUESTA 38:

Una solución válida es:

```
#include <stdio.h>
int main() {
    int n;
    printf("Dame un número: ");
    scanf("%d", &n);
    if (n % 2 == 0)
        printf("Es par");
    else
        printf("Es impar");
    return 0;
}
```

El operador `%` regresa el residuo de la división entera, así que `n % 2` vale 0 solo cuando `n` es divisible entre 2, es decir, par. El `if-else` decide el mensaje según ese resultado. También sería válido usar el operador ternario o verificar `n % 2 != 0` para el impar. Lo importante es que combines lectura con `scanf`, el operador módulo y una estructura condicional, que son los tres temas de las unidades 2 a 4.

También puedes resolverlo con el operador ternario en una sola línea: `printf(n % 2 == 0 ? "Es par" : "Es impar");`. El ternario es compacto pero no abuses de él: si la lógica crece, el `if-else` es más legible. Elegir la estructura más clara para cada caso es parte de las buenas prácticas de la unidad 10.

RESPUESTA 39: 5 3 1

La salida es "5 3 1 " porque el `do-while` ejecuta el cuerpo al menos una vez: imprime 5, resta 2 y queda 3; como $3 > 0$ repite, imprime 3, resta 2 y queda 1; como $1 > 0$ repite, imprime 1, resta 2 y queda -1; ahora $-1 > 0$ es falso y termina. La clave del `do-while` frente al `while` normal es que la condición se evalúa al final, así que el cuerpo siempre corre mínimo una vez. El error común es pensar que imprime también el -1 o que se detiene en 1 sin imprimirlo.

El `do-while` brilla en menús interactivos: quieres mostrar el menú al menos una vez y repetir mientras el usuario no elija salir. El patrón es `do { mostrar menú; leer opción; } while (opción != 0);`. Si usarás un `while` normal tendrías que duplicar código o inventar una condición inicial artificial.

RESPUESTA 40:

El error es el punto y coma después del `for`: ese `;` forma un cuerpo vacío, así que el bucle gira 10 veces sin hacer nada y luego suma `+= i` y se ejecuta una sola vez con `i` valiendo 11, imprimiendo 11 en lugar de 55. Versión corregida:

```
#include <stdio.h>
int main() {
    int suma = 0, i;
    for (i = 1; i <= 10; i++)
        suma += i;
    printf("%d", suma);
    return 0;
}
```

Quitando el punto y coma, `suma += i` queda como cuerpo del `for` y se acumula $1 + 2 + \dots + 10 = 55$. Este es un error lógico traicionero porque el programa compila sin advertencias y el resultado parece casi plausible. Moraleja de la unidad 5: un `;` solitario después de un `for` o un `while` cambia por completo el significado del programa.

Compilar con `-Wall` te habría avisado de este bug: `gcc` detecta el cuerpo vacío del `for` y muestra una advertencia de sentencia sin efecto. Este es el mejor argumento para activar siempre las advertencias: el compilador es tu primer revisor de código y encuentra errores que tus ojos ya no ven después de horas programando.

RESPUESTA 41:

Una solución válida es:

```
#include <stdio.h>
int mayor(int a, int b, int c) {
    int m = a;
    if (b > m) m = b;
    if (c > m) m = c;
    return m;
}
int main() {
    printf("%d", mayor(4, 9, 2));
    return 0;
}
```

La función guarda el primer valor en m y lo compara con los otros dos, quedándose siempre con el más grande, y lo regresa con return. En el main se prueba con 4, 9 y 2 y debe imprimir 9. También es válido resolverlo con if-else anidados. Lo esencial de la unidad 6 aquí es: tipo de regreso int, tres parámetros int, y que el valor regresado se usa en el printf del main.

Otra forma válida es con if-else anidados comparando de dos en dos, o incluso llamando a una función menor de dos números dos veces. En programación casi siempre hay varios caminos correctos: lo que importa es que la función sea clara, tenga un solo punto de regreso bien definido y se pruebe con casos como números iguales o negativos.

RESPUESTA 42: 5

La salida es "5" porque en C los argumentos se pasan por valor: cuando llamas cambiar(n), la función recibe una copia de n en su parámetro x, así que x = 100 solo modifica la copia local y la variable n del main sigue valiendo 5. Para que el cambio se reflejara afuera habría que pasar la dirección con cambiar(&n) y recibir un apuntador. Esta es la razón por la que la unidad 8 existe: sin apuntadores, una función jamás puede modificar las variables de quien la llama.

Si quieres que la función si modifique la variable original, pasa su dirección: cambiar(&n) con parámetro int *x y cuerpo *x = 100;. Este patrón de paso por referencia simulada con apuntadores es omnipresente en C: scanf lo usa, y funciones como swap lo necesitan. Entenderlo es dominar la unidad 8.

RESPUESTA 43:

Una solución válida es:

```
#include <stdio.h>
int main() {
    int v[5], i, suma = 0;
    for (i = 0; i < 5; i++) {
        printf("Dame el valor %d: ", i + 1);
        scanf("%d", &v[i]);
    }
    for (i = 0; i < 5; i++)
        suma += v[i];
    printf("Promedio: %.2f", suma / 5.0);
    return 0;
}
```

El primer bucle llena el arreglo leyendo con `scanf` en la dirección de cada elemento `&v[i]`. El segundo acumula la suma. Para el promedio se divide entre 5.0 (no entre 5) para forzar división flotante, y `%.2f` muestra dos decimales. Si dividieras entre 5 entero, el promedio se truncaría y perderías los decimales. Aquí se juntan arreglos de la unidad 7 con la división entera contra flotante de la unidad 2.

Si los valores pueden ser decimales, declara el arreglo como `float` y acumula en `float` para no perder precisión desde la lectura. Y si el tamaño del arreglo cambiara, calcula el promedio dividiendo entre una constante o una variable `n` en lugar del 5 fijo. Escribir código que se adapta a cambios es programar pensando en el futuro.

RESPUESTA 44: 4 5

La salida es "4 5" porque `strlen` cuenta los caracteres hasta el `\0` sin incluirlo: "Hola" tiene 4 letras. En cambio `sizeof` mide el tamaño en bytes del arreglo completo, que incluye el carácter nulo final, así que son 5 bytes. Esta pregunta resume una distinción clave de la unidad 7: `strlen` es longitud lógica del texto y `sizeof` es tamaño físico en memoria. Si declararas `char texto[20] = "Hola";`, `strlen` seguiría dando 4 pero `sizeof` daría 20.

Al planear buffers para cadenas, la fórmula es: longitud máxima del texto más 1 para el `\0`. Si el nombre más largo mide 29 caracteres, el arreglo debe ser de 30. Quedarte corto por un byte es el famoso desbordamiento de buffer, la vulnerabilidad de seguridad más clásica de la historia de C.

RESPUESTA 45:

El error es que `p` es un apuntador sin inicializar: no apunta a ninguna variable válida, así que `*p = a` escribe en una dirección aleatoria y el programa falla. Hay que hacerlo apuntar a `a` primero. Versión corregida:

```
#include <stdio.h>
int main() {
    int a = 10;
    int *p;
    p = &a;
    printf("%d", *p);
    return 0;
}
```

Con `p = &a`, el apuntador guarda la dirección de `a`, y `*p` desreferencia esa dirección para leer el 10. También sería válido escribir `int *p = &a;` en una sola línea. La lección de la unidad 8: un apuntador solo es útil cuando apunta a algo; declararlo no reserva la variable destino, solo la flecha.

La forma más segura es inicializar al declarar: `int *p = &a;`. Así el apuntador nace apuntando a algo válido y eliminas la ventana de tiempo en que existe pero no sirve. Esta costumbre de inicializar todo al declarar aplica a variables normales también y previene una familia entera de bugs.

RESPUESTA 46: 30 40

La salida es "30 40" porque `p` apunta al inicio del arreglo `v`, así que `p + 2` apunta al tercer elemento y `*(p + 2)` vale 30. La notación `p[3]` es equivalente a `*(p + 3)`, es decir, el cuarto elemento, que vale 40. Esta pregunta demuestra la equivalencia entre apuntadores y arreglos de la unidad 8: el nombre de un arreglo decae a apuntador a su primer elemento, y la sintaxis de corchetes funciona igual sobre apuntadores.

Memoriza la equivalencia fundamental: `p[i]` es exactamente `*(p + i)` para cualquier apuntador `p` e índice `i`. Incluso funciona al revés de forma curiosa: `i[p]` también compila porque es `*(i + p)`. No lo uses así en código real, pero entender por qué funciona demuestra que de verdad captaste la aritmética de apuntadores.

RESPUESTA 47:

Una solución válida es:

```

#include <stdio.h>
struct Alumno {
    char nombre[30];
    float calificacion;
};
int main() {
    struct Alumno grupo[3];
    int i, mejor = 0;
    for (i = 0; i < 3; i++) {
        printf("Nombre: ");
        scanf("%29s", grupo[i].nombre);
        printf("Calificacion: ");
        scanf("%f", &grupo[i].calificacion);
    }
    for (i = 1; i < 3; i++) {
        if (grupo[i].calificacion > grupo[mejor].calificacion)
            mejor = i;
    }
    printf("Mejor alumno: %s", grupo[mejor].nombre);
    return 0;
}

```

Se define el struct con sus dos miembros, se crea un arreglo de 3 structs y se leen los datos accediendo con el operador punto: `grupo[i].nombre` y `grupo[i].calificación`. Para el nombre se usa `%29s` y se pasa el arreglo directamente porque el nombre de un arreglo ya es su dirección; para el float se usa `%f` con `&`. El segundo bucle rastrea el índice con la calificación más alta. Aquí se combinan structs de la unidad 9 con arreglos y cadenas de la unidad 7.

El `%29s` es programación defensiva: limita la lectura a 29 caracteres más el `\0` para no desbordar el arreglo de 30. Sin ese límite, un usuario podría escribir un nombre larguísimo y corromper la memoria vecina. Nunca confíes en que el usuario escribiera datos razonables: limita siempre tus lecturas.

RESPUESTA 48:

Los dos errores son: no se verifica si `malloc` regresa `NULL` (si no hay memoria, usar datos provoca fallo) y no se libera la memoria con `free` al final (fuga de memoria). Versión corregida:

```

#include <stdio.h>
#include <stdlib.h>
int main() {
    int *datos = malloc(5 * sizeof(int));
    int i;
    if (datos == NULL) {
        printf("Sin memoria");
        return 1;
    }
    for (i = 0; i < 5; i++)
        datos[i] = i * 10;
    for (i = 0; i < 5; i++)
        printf("%d ", datos[i]);
    free(datos);
    return 0;
}

```

Verificar NULL después de cada malloc es programación defensiva básica: malloc puede fallar y tu programa debe manejarlo con dignidad en lugar de tronar. Y cada malloc debe tener su free correspondiente cuando la memoria ya no se necesite; esa pareja es el contrato sagrado de la memoria dinámica de la unidad 9.

Existe una alternativa llamada calloc(5, sizeof(int)) que además inicializa toda la memoria en ceros, útil cuando necesitas el arreglo limpio desde el inicio. Y si más tarde necesitas más espacio, realloc agranda el bloque conservando los datos. malloc, calloc, realloc y free son las cuatro herramientas del heap: aprendetelas como un equipo.

RESPUESTA 49:

Una solución válida es:


```

#include <stdio.h>
int main() {
    FILE *f = fopen("notas.txt", "r");
    int n, contador = 0, suma = 0;
    if (f == NULL) {
        printf("Error: no se pudo abrir el archivo");
        return 1;
    }
    while (fscanf(f, "%d", &n) == 1) {
        contador++;
        suma += n;
    }
    fclose(f);
    printf("Numeros leidos: %d, suma: %d", contador, suma);
    return 0;
}

```

Se abre con modo "r" y se verifica que el apuntador FILE no sea NULL, que es como fopen avisa que el archivo no existe. El while usa fscanf, que regresa cuántos datos leyo: mientras sea 1 hay números por procesar, y cuando llega al fin del archivo regresa EOF y el bucle termina. Al final se cierra con fclose y se reportan el conteo y la suma. Esta es la plantilla clásica de lectura de archivos de la unidad 10: abrir, verificar, procesar en bucle, cerrar.

Una alternativa común es usar while (!feof(f)), pero tiene una trampa: feof solo se activa después de intentar leer más allá del fin, así que suele procesar un dato fantasma extra. Revisar el valor de regreso de fscanf como aquí es la forma robusta y recomendada. En archivos, como en todo C, verifica resultados en lugar de asumir.

RESPUESTA 50:

Los errores son dos: el bucle copia todos los caracteres pero nunca agrega el \0 al final de dest, así que b no queda como cadena válida; y copiar recibe char *orig cuando debería ser const char *orig porque no modifica el origen (buena práctica, aunque compila sin eso). Versión corregida:

```

#include <stdio.h>
void copiar(char *dest, const char *orig) {
    while (*orig != '\0') {
        *dest = *orig;
        dest++;
        orig++;
    }
    *dest = '\0';
}
int main() {
    char a[] = "Programar";
    char b[20];
    copiar(b, a);
    printf("%s", b);
    return 0;
}

```

La línea `*dest = '\0';` después del bucle es la que faltaba: sin el terminador, `printf("%s")` seguiría leyendo basura de memoria hasta encontrar un cero por casualidad. El bucle en si estaba bien: avanza ambos apuntadores copiando carácter por carácter. Esta pregunta cierra el curso uniendo apuntadores, cadenas y el carácter nulo, los tres pilares de las unidades 7 y 8.

En la librería estándar ya existe `strcpy` para copiar cadenas, pero hacerlo a mano con apuntadores es el mejor ejercicio para entender cómo funcionan por dentro. Fíjate que la versión corregida también declara `orig` como `const char*`: promete no modificar el origen y el compilador te avisa si lo intentas. Usar `const` donde se puede es marca de programador profesional.

Hoja de respuestas rápida 1: b - 2: b - 3: c - 4: b - 5: b - 6: b - 7: c - 8: b - 9: a - 10: b - 11: b - 12: a - 13: c - 14: a - 15: b - 16: b - 17: b - 18: a - 19: a - 20: a - 21: Verdadero - 22: Falso - 23: Verdadero - 24: Falso - 25: Falso - 26: Verdadero - 27: Verdadero - 28: Falso - 29: Verdadero - 30: Falso - 31: Verdadero - 32: Falso - 33: Verdadero - 34: Verdadero - 35: Verdadero - 36: 3 3 - 37: falta & en scanf - 38: par/impar con % - 39: 5 3 1 - 40: quitar ; del for - 41: función mayor - 42: 5 (paso por valor) - 43: promedio con arreglo - 44: 4 5 - 45: `p = &a` antes de `*p` - 46: 30 40 - 47: struct Alumno - 48: verificar NULL y free - 49: lectura de archivo - 50: agregar `\0` final

Si llegaste hasta aquí y tu puntaje fue alto, felicítate en serio: dominar C desde el primer `printf` hasta la memoria dinámica y los archivos no lo hace cualquiera, y tu lo hiciste paso a paso con cada unidad de este libro. Y si te faltó puntaje, no te aguiste: cada error que marcaste hoy es una lección que ya no se te va a olvidar, así que repasa esas unidades, vuelve a intentarlo y verás como subes. Lo importante es que no te detengas, porque programar se aprende

programando, todos los días un poco. Nos vemos en lucasegaacademy.com, donde te esperan más cursos, más retos y una comunidad que está construyendo su futuro línea por línea, igual que tu.

Curso Completo de C

Libro de programación de Lucas EGA Academy: explicaciones extensas unidad por unidad, práctica guiada y examen final para dominar el lenguaje desde cero.

Colibrí Editorial EGA

lucasegaacademy.com