

LUCAS EGA ACADEMY



C++

++



MASTERING MODERN C++ • PERFORMANCE • SYSTEMS • EMBEDDED

Curso Completo de C++

© 2026 Esteban Gutiérrez

Colibrí Editorial EGA

Hecho en México

ÍNDICE

PRÓLOGO	10
A quién va dirigido este libro	10
Por qué C++ sigue siendo rey	11
La diferencia entre C y C++	11
Cómo estudiar este libro	11
Requisitos	13
UNIDAD 1: Tus primeros pasos con C++	15
Explicación detallada	15
Ejemplos paso a paso	19
Práctica	22
Resumen de la unidad	24
Profundización: cómo piensa un compilador de C++	25
Expansión Unidad 1: estándares, ecosistema y tu programa frente al sistema	31
La evolución de los estándares: C++11, 14, 17, 20 y 23	31
C++ frente a Python, Java y JavaScript: cuándo elegir cada uno	37
Anatomía detallada de un programa en C++	39
Instalación paso a paso en Windows: MinGW-w64 y VS Code	42
Enlazado estático vs dinámico: cómo tu programa se junta con las bibliotecas	45
Tu programa y el sistema operativo: argumentos y variables de entorno	47
Catálogo de errores comunes de esta parte (y cómo evitarlos)	52
Práctica: ejercicios con solución	52
Resumen de la expansión	60
UNIDAD 2: Los fundamentos del lenguaje	62
Explicación detallada	62
Ejemplos paso a paso	65
Práctica	68
Resumen de la unidad	72
Profundización: tipos, límites y conversiones	73
Expansión Unidad 2: lógica, caracteres, bits y el detalle fino de los tipos	80
Lógica booleana a fondo: las tablas de verdad aplicadas	80

El tipo char y la tabla ASCII: cuando las letras son números	84
Operadores bit a bit: controlando cada interruptor	87
La tabla completa de precedencia de operadores	91
El operador ternario a fondo	92
sizeof aplicado a expresiones: más de lo que parece	92
Alias con using en contextos reales	93
Constantes literales: sufijos, bases y separadores	94
Catálogo de errores comunes de esta parte (y cómo evitarlos)	97
Práctica: ejercicios con solución	98
Resumen de la expansión	105
UNIDAD 3: Decisiones y repeticiones I	107
Explicación detallada	107
Ejemplos paso a paso	111
Práctica	114
Resumen de la unidad	118
Profundización: lógica, patrones de ciclos y máquinas de estado	119
Expansion: trampas, validacion y juegos completos	127
Comparaciones encadenadas: la trampa que parece obvia	128
switch con enum class: decisiones con nombre	130
El operador coma en el for	132
Ciclos infinitos a proposito	133
break y continue en ciclos anidados: salir de varios niveles	135
Validacion robusta de entrada numerica: a prueba de usuarios reales	139
Menus multinivel: programas que se sienten profesionales	142
Juego completo 1: piedra, papel o tijeras	145
Juego completo 2: batalla naval simplificada	148
Catalogo de errores comunes	151
Practica	152
Resumen de la expansion	157
UNIDAD 4: Bucles for, funciones y sobrecarga	159
Explicación detallada	159
Ejemplos paso a paso	162

Práctica	165
Resumen de la unidad	169
Profundización: recursividad, alcance y diseño de funciones	170
Expansion: parametros, la pila, lambdas y proyectos reales	177
Paso de parametros en detalle: valor, referencia y puntero	177
Devolver mas de un valor: cuatro tecnicas	181
La pila de llamadas, dibujada	185
Recursividad vs iteracion: duelo con cronometro	186
Memoizacion: Fibonacci a velocidad luz	188
Lambdas: funciones sin nombre	190
De un archivo a un proyecto real: .h y .cpp	192
Punteros a funciones: el abuelo de las lambdas	194
Recursion mutua: funciones que se llaman entre si	197
Funciones constexpr: calculos en tiempo de compilacion	199
Catalogo de errores comunes	200
Practica	200
Resumen de la expansion	206
UNIDAD 5: Arreglos, strings y punteros	208
Explicación detallada	208
Ejemplos paso a paso	211
Práctica	214
Resumen de la unidad	219
Profundización: el modelo de memoria de C++	220
std::array: el arreglo crudo con modales	227
Strings a fondo: el texto como un profesional	230
cctype: el kit completo para clasificar caracteres	237
Punteros a punteros: direcciones de direcciones	240
Punteros y const: las cuatro combinaciones	244
Argumentos de línea de comandos: el arreglo de punteros más famoso	246
qsort contra sort: el duelo del ordenamiento	248
Catálogo de errores comunes (tercera parte)	251
Práctica: ejercicios con solución	252

Resumen de la tercera parte	258
UNIDAD 6: Referencias y programación orientada a objetos	260
Explicación detallada	260
Ejemplos paso a paso	264
Práctica	267
Resumen de la unidad	272
Profundización: diseño orientado a objetos	273
La lista de inicialización a fondo	283
Delegación de constructores: no repitas tu lógica	288
Constructores explicit: freno a las conversiones silenciosas	289
El constructor de copia en detalle	290
El operador de asignación: copiar sin nacer	295
La regla de tres, la de cinco y la de cero	296
Agregación vs composición: quién posee a quién	298
Clases inmutables: objetos que no cambian	301
Namespaces: orden en la casa de clases	302
Catálogo de errores comunes (tercera parte)	306
Práctica: ejercicios con solución	307
Resumen de la tercera parte	317
UNIDAD 7: Los pilares de la programación orientada a objetos	318
Explicación detallada	318
Ejemplos paso a paso	323
Práctica	326
Resumen de la unidad	334
Profundización: polimorfismo por dentro y principios de diseño	334
Expansión Unidad 7: herencia múltiple, contratos, estrategia y jerarquías reales	342
Herencia múltiple bien usada: las interfaces	342
Clases abstractas como contratos: un sistema de pagos	345
El patrón Estrategia: cambiar comportamiento sin heredar	348
Composición vs herencia: la guía de decisión definitiva	350
Jerarquías reales: los streams de ostream explicados	353
Object slicing en vectores y su solución definitiva	356

Métodos virtuales puros con implementación por defecto	358
Catálogo de errores comunes de esta expansión	360
Práctica de la expansión	361
De herencia a composición: cómo refactorizar sin miedo	368
Lista de verificación antes de diseñar una jerarquía	370
Resumen de la expansión	370
UNIDAD 8: Técnicas avanzadas	372
Explicación detallada	372
Ejemplos paso a paso	377
Práctica	379
Resumen de la unidad	384
Profundización: plantillas, iteradores y algoritmos	385
Expansión Unidad 8: plantillas avanzadas, operadores y algoritmos	393
Plantillas con múltiples parámetros y valores por defecto	394
Plantillas vs herencia: cuándo usar cada una para código genérico	395
std::pair y std::tuple: agrupar datos sin crear clases	398
Operadores de comparación completos y cómo se relacionan	400
Operadores de conversión: cuando tu objeto quiere ser otro tipo	402
Funtores: objetos que se comportan como funciones	404
std::function: guardar "cualquier cosa llamable"	406
Más algoritmos: copy, fill, replace y unique	407
Catálogo de errores comunes de esta expansión	409
Práctica de la expansión	410
Guía de decisión: lambda, functor o std::function	421
Resumen de la expansión	422
UNIDAD 9: Herramientas profesionales	423
Explicación detallada	423
Ejemplos paso a paso	426
Práctica	430
Resumen de la unidad	436
Profundización: más STL, seguridad ante excepciones y sistema de archivos	436
Expansion Unidad 9: errores profesionales, bitacoras y datos que aguantan el mund... . .	446

Excepciones personalizadas: diseña tu propia jerarquía	447
RAII y excepciones: la garantía en la práctica	450
Relanzar excepciones y catch(...)	452
assert vs excepciones: cada uno en su lugar	455
Bitácoras (logging): tu caja negra	456
Serialización manual: objetos que viajan en texto	459
CSV robusto: comillas y comas dentro de campos	462
Catálogo de errores comunes de esta expansión	464
Práctica	465
Resumen de la expansión	475
UNIDAD 10: Nivel profesional	476
Explicación detallada	476
Ejemplos paso a paso	483
Práctica	484
Resumen de la unidad	488
Profundización: hilos, depuración y proyectos reales	488
Expansión Unidad 10: memoria avanzada, rendimiento y tu carrera profesional	494
weak_ptr: el puntero que observa sin poseer	494
Custom deleters: limpieza a tu medida	497
Move semantics: mover en vez de copiar	499
Perfilado básico: mide antes de optimizar	501
Guía de entrevistas de trabajo en C++	503
Cómo contribuir a open source (y por qué te conviene)	506
Plan de estudios de 90 días post-libro	507
Sanitizers: que el compilador encuentre tus bugs de memoria	508
Catálogo de errores comunes de esta expansión	510
Práctica	511
Cierre: lo que sigue es tuyo	519
EXAMEN FINAL	520
Preguntas de opción múltiple	520
Preguntas de verdadero o falso	523
Preguntas abiertas (escribe el código)	524

Respuestas y soluciones	525
Despedida	539

PRÓLOGO

Bienvenido a "Curso Completo de C++", el libro de Lucas EGA Academy para dominar el lenguaje que mueve los videojuegos, los sistemas operativos, los navegadores y buena parte de la infraestructura del mundo. 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.

C++ no es un lenguaje cualquiera. Es el lenguaje con el que se construyen las herramientas con las que se construyen otros lenguajes. Cuando juegas un videojuego moderno, cuando usas Google Chrome, cuando un banco procesa millones de transacciones por segundo, ahí está C++. Es veloz, es potente y exige que entiendas lo que pasa debajo del capó. Por eso es el lenguaje favorito de las universidades serias y de las empresas que necesitan rendimiento de verdad.

A quién va dirigido este libro

Este libro es para ti si ya sabes programar en C o en algún otro lenguaje y quieres dar el salto a la programación orientada a objetos con uno de los lenguajes más respetados de la industria. Es para ti si vienes de Python o JavaScript y sientes que tus programas "hacen magia" sin que entiendas qué pasa con la memoria: C++ te quita la venda de los ojos.

También es para estudiantes de ingeniería y sistemas que llevan C++ en la carrera y quieren una explicación clara, sin el lenguaje enredado de los libros de texto. Y es para autodidactas que quieren un reto serio: aprender C++ es como aprender a manejar un auto estándar después de manejar automático. Al principio cuesta, pero después entiendes el auto de verdad.

No necesitas ser experto en C para empezar, pero sí ayuda conocer lo básico: variables, condiciones y ciclos. Si no lo tienes, repasa primero el curso de C de la academia y vuelve. Aquí retomaremos desde los fundamentos, pero avanzaremos rápido.

Por qué C++ sigue siendo rey

C++ nació en los años ochenta como una extensión de C, y lleva más de cuarenta años evolucionando sin perder su esencia: control total sobre la máquina. Mientras otros lenguajes sacrifican velocidad por comodidad, C++ te deja decidir exactamente cuánta comodidad quieres y cuánta velocidad necesitas.

Mira dónde vive C++ hoy: los motores de videojuegos más potentes (Unreal Engine), los sistemas operativos y sus controladores, los navegadores web, las bases de datos de alto rendimiento, los sistemas de trading financiero donde cada microsegundo vale dinero, los robots, los drones, la inteligencia artificial de bajo nivel. Si algo necesita ser rápido y confiable, probablemente está escrito en C++ o en algo escrito en C++.

Además, aprender C++ te hace mejor programador en cualquier otro lenguaje. Cuando entiendes punteros, memoria, compilación y el costo real de las operaciones, programas con una conciencia que la mayoría no tiene. Es como entrenar con pesas: después, todo lo demás se siente ligero.

La diferencia entre C y C++

Si ya conoces C, C++ te va a resultar familiar: la sintaxis básica, los tipos, los ciclos y los punteros son casi iguales. Pero C++ agrega un mundo nuevo: programación orientada a objetos (clases, herencia, polimorfismo), plantillas que te permiten escribir código genérico, la STL (una biblioteca estándar poderosísima con contenedores y algoritmos listos para usar), manejo de excepciones moderno y herramientas para programar de forma más segura y expresiva.

La diferencia más importante es de mentalidad. C es procedural: piensas en funciones que manipulan datos. C++ te permite pensar en objetos que combinan datos y comportamiento, que se construyen solos, se destruyen solos y se relacionan entre ellos como las cosas del mundo real. Este libro te lleva de la mentalidad de C a la mentalidad de C++ paso a paso.

Cómo estudiar este libro

Este libro está organizado en 10 unidades, cada una agrupa 3 de las 30 clases originales del curso en video. Cada unidad sigue la misma estructura:

- Introducción: qué verás y por qué importa, con el panorama completo de la unidad.

- Explicación detallada: conceptos a fondo, analogías de la vida real, errores comunes que cometen los principiantes y buenas prácticas de la industria.
- Ejemplos paso a paso: código C++ completo para copiar, compilar y probar, explicado línea por línea.
- Práctica: al menos 5 ejercicios con soluciones explicadas paso a paso. C++ se aprende escribiendo código, no solo leyendo.
- Resumen: los puntos clave para repasar rápido antes de seguir.

Al final hay un EXAMEN FINAL con 50 preguntas: opción múltiple, verdadero o falso y preguntas abiertas donde escribirás tu propio código. Cada respuesta viene explicada en las soluciones. Si lo resuelves con confianza, estás listo para proyectos reales en C++.

Este es tu mapa:

Unidad 1: Tus primeros pasos con C++ (Clases 1 a 3)

Desde cero absoluto. Clase 1: qué es C++, su historia y por qué domina el software de alto rendimiento. Clase 2: las diferencias entre C y C++, qué cambió y por qué. Clase 3: instalación del compilador g++ paso a paso y tu primer programa "Hola, mundo". Terminas con el compilador funcionando en tu máquina.

Unidad 2: Los fundamentos del lenguaje (Clases 4 a 6)

La base de todo programa. Clase 4: variables y tipos de datos, con sus tamaños y límites. Clase 5: operadores aritméticos, relacionales, lógicos y sus trampas. Clase 6: entrada y salida con cin y cout, el sistema iostream. Aprenderás a pedir datos al usuario y mostrar resultados.

Unidad 3: Decisiones y repeticiones I (Clases 7 a 9)

Tu programa empieza a pensar. Clase 7: condicionales if, else if y else con lógica real. Clase 8: switch para menús y decisiones múltiples. Clase 9: ciclos while y do-while. Ya podrás escribir programas que reaccionan a lo que pasa.

Unidad 4: Bucles, funciones y sobrecarga (Clases 10 a 12)

Clase 10: el ciclo for y sus variantes. Clase 11: funciones para organizar tu código en piezas reutilizables. Clase 12: sobrecarga de funciones, la primera probada del poder de C++. Tu código empezará a verse profesional.

Unidad 5: Arreglos, strings y punteros (Clases 13 a 15)

Clase 13: arreglos para guardar colecciones de datos. Clase 14: la clase string, el manejo moderno de texto. Clase 15: punteros, el tema legendario de C++: qué son, cómo se usan y por qué no hay que tenerles miedo.

Unidad 6: Referencias y programación orientada a objetos (Clases 16 a 18)

Clase 16: referencias, la forma elegante de pasar datos. Clase 17: clases y objetos, el corazón de C++. Clase 18: constructores y destructores, cómo los objetos nacen y mueren solos.

Unidad 7: Los pilares de la POO (Clases 19 a 21)

Clase 19: encapsulamiento, proteger tus datos como un profesional. Clase 20: herencia, reutilizar código creando jerarquías. Clase 21: polimorfismo y funciones virtuales, el concepto más poderoso de la programación orientada a objetos.

Unidad 8: Técnicas avanzadas (Clases 22 a 24)

Clase 22: sobrecarga de operadores, hacer que tus objetos se sumen y se comparen como números. Clase 23: plantillas (templates), código genérico que funciona con cualquier tipo. Clase 24: la STL y los vectores, la biblioteca que te ahorra cientos de horas.

Unidad 9: Herramientas profesionales (Clases 25 a 27)

Clase 25: mapas y conjuntos de la STL. Clase 26: excepciones, manejar errores sin que tu programa se caiga. Clase 27: archivos, guardar y leer datos en disco con fstream.

Unidad 10: Nivel profesional (Clases 28 a 30)

Clase 28: punteros inteligentes, la memoria moderna y segura de C++. Clase 29: proyecto final completo, un sistema real que integra todo lo aprendido. Clase 30: buenas prácticas, qué sigue después de este libro y cómo continuar tu camino.

Requisitos

Necesitas una computadora con Windows, Linux o macOS, conexión a internet para descargar el compilador y ganas de practicar. Todos los ejemplos usan el estándar moderno de C++ (C++17 en adelante), así que asegúrate de usar un compilador actualizado: g++ versión 8 o superior, o el que venga con tu sistema.

Una advertencia honesta: C++ exige más atención que Python o JavaScript. El compilador es estricto, los errores a veces parecen jeroglíficos y la memoria no se maneja sola. Pero cada error que resuelvas te hará más fuerte, y al final del libro leerás mensajes del compilador como si fueran español. La recompensa vale el esfuerzo: pocos lenguajes te dan este nivel de poder y entendimiento.

Empecemos. Tu primer programa en C++ está a unas páginas de distancia, y tu primer sistema orientado a objetos, a unas unidades. ¡A programar!

UNIDAD 1: Tus primeros pasos con C++

Bienvenido a tu primera unidad. Aquí no escribirás grandes programas todavía: construirás los cimientos. Vas a entender qué es C++ y de dónde viene, por qué es diferente de C aunque se parezca tanto, y vas a instalar tu compilador y correr tu primer programa. Parece poco, pero estas tres clases definen todo lo demás: si entiendes la filosofía del lenguaje y tienes tu entorno funcionando, el resto del libro fluye.

Piensa en esta unidad como aprender a usar las herramientas antes de construir la casa. Un carpintero no empieza clavando tablas sin conocer su martillo. Tú no empezarás a programar objetos y plantillas sin conocer tu compilador y sin entender por qué C++ piensa como piensa.

Al terminar esta unidad sabrás: la historia y la filosofía de C++, las diferencias concretas entre C y C++, cómo instalar g++ en Windows, Linux y macOS, cómo compilar y ejecutar un programa desde la terminal, y qué hace cada parte del programa "Hola, mundo" más famoso de la programación.

Explicación detallada

Clase 1: Qué es C++ y por qué domina el software serio

C++ es un lenguaje de programación compilado, de propósito general, famoso por darte control total sobre el hardware sin renunciar a herramientas modernas de alto nivel. Nació en 1979 cuando un científico danés llamado Bjarne Stroustrup, trabajando en los laboratorios Bell, quiso agregarle a C algo que había visto en otro lenguaje llamado Simula: las clases. Su idea era simple y ambiciosa: un lenguaje tan rápido como C pero con herramientas para organizar programas grandes.

El primer nombre del proyecto fue "C con clases". En 1983 se renombró a C++, un chiste de programador: en C, el operador ++ significa "incrementa en uno", así que C++ es "C incrementado", C mejorado. El chiste se quedó para siempre.

Desde entonces, C++ ha evolucionado por estándares: C++98, C++03, C++11 (la gran revolución moderna), C++14, C++17, C++20 y C++23. Cada estándar agrega herramientas sin romper lo anterior. En este libro usamos C++17 como base, que ya trae todo lo moderno que necesitas: la palabra auto, los rangos en ciclos for, los punteros inteligentes y una STL madura.

Pero ¿por qué aprender C++ en 2026, con tantos lenguajes "más fáciles"? Por tres razones que ningún otro lenguaje combina igual.

Primera: rendimiento sin concesiones. C++ se compila directamente a código máquina, sin máquina virtual ni intérprete en medio. Cuando cada microsegundo cuenta -un motor de videojuegos renderizando 60 cuadros por segundo, un sistema de trading decidiendo compras en nanosegundos- C++ es la respuesta. Python es cómodo, pero un programa en C++ bien escrito puede ser 50 o 100 veces más rápido.

Segunda: control total de la memoria. En C++ tú decides cuándo se crea cada objeto y cuándo se destruye, dónde vive en memoria y cuánto pesa. Ese control es responsabilidad, sí, pero también es poder: puedes escribir programas que usan exactamente los recursos que necesitan, ni un byte más. Por eso los sistemas embebidos, los drones y los teléfonos usan C++ en sus partes críticas.

Tercera: es el lenguaje que te enseña cómo funcionan las computadoras de verdad. Los lenguajes modernos esconden la memoria, el compilador y el hardware. C++ te los muestra. Cuando entiendes C++, entiendes por qué un programa es lento, qué significa "reservar memoria", qué hace un compilador y cómo se organiza un programa grande. Ese conocimiento te sirve en todos los lenguajes.

La analogía clásica: aprender a programar solo en Python es como aprender a cocinar solo con microondas. Funciona, es rápido, pero no entiendes el fuego. C++ es aprender a cocinar con fuego de verdad: más peligroso al inicio, pero después puedes cocinar cualquier cosa.

¿Dónde se usa C++ hoy? En los motores de videojuegos (Unreal Engine está escrito en C++), en navegadores como Chrome, en sistemas operativos y controladores, en bases de datos como MySQL, en software de Adobe, en sistemas financieros de alta frecuencia, en robótica y en inteligencia artificial (las bibliotecas más rápidas de machine learning tienen su núcleo en C++). Si algo necesita velocidad y no puede fallar, hay C++ cerca.

Clase 2: Diferencias entre C y C++

Si vienes de C, mucho te será familiar: la sintaxis básica, los tipos fundamentales, los ciclos, los punteros. C++ nació como superconjunto de C, y casi todo programa válido en C compila en C++. Pero C++ agregó capas nuevas que cambian la forma de pensar. Veamos las diferencias una por una.

La diferencia más grande es la programación orientada a objetos. C es procedural: escribes funciones que operan sobre datos sueltos. C++ te permite crear clases: moldes que combinan datos y funciones en una sola unidad llamada objeto. Un objeto "se cuida solo": se inicializa

con un constructor, se limpia con un destructor y expone solo lo necesario. En C, una estructura de datos es pasiva; en C++, un objeto es activo. Esta diferencia la exploraremos a fondo en las unidades 6 y 7, pero es el corazón del lenguaje.

Segunda diferencia: entrada y salida modernas. C usa `printf` y `scanf`, funciones con formatos crípticos como `"%d"` y peligrosas si te equivocas. C++ usa `iostream`: `cout` para mostrar y `cin` para leer, con el operador `<<` y `>>`. No necesitas memorizar códigos de formato y el compilador verifica los tipos por ti. Es más seguro y más legible.

Tercera: el manejo de strings. En C, un texto es un arreglo de caracteres terminado en cero, y manipularlo es un campo minado de funciones como `strcpy` y `strcat`. C++ trae la clase `std::string`: un tipo de texto que se comporta como debe, se puede sumar con `+`, se compara con `==` y maneja su propia memoria. Es uno de los alivios más grandes al pasar de C a C++.

Cuarta: la STL, la Biblioteca Estándar de Plantillas. C te da unas pocas funciones de biblioteca. C++ te da contenedores listos (`vector`, `map`, `set`), algoritmos (`ordenar`, `buscar`) e iteradores, todo genérico gracias a las plantillas. Es como pasar de una caja de herramientas básica a un taller completo.

Quinta: manejo de memoria moderno. C solo tiene `malloc` y `free`, manuales y peligrosos. C++ agrega `new` y `delete`, y en el C++ moderno, los punteros inteligentes (`unique_ptr`, `shared_ptr`) que liberan la memoria automáticamente. Menos fugas de memoria, menos dolores de cabeza.

Sexta: excepciones. En C, los errores se reportan con códigos de retorno que hay que revisar a mano en cada llamada. C++ tiene `try`, `catch` y `throw`: un sistema estructurado para manejar errores que separa el código normal del código de emergencia.

Séptima: sobrecarga de funciones y operadores. En C, cada función necesita un nombre único. En C++, puedes tener varias funciones con el mismo nombre pero diferentes parámetros, y el compilador elige la correcta. Incluso puedes redefinir qué significa `+` o `==` para tus propios tipos. Esto hace el código más natural.

Octava: espacios de nombres (`namespaces`). En proyectos grandes de C, los nombres de funciones chocan. C++ organiza el código en espacios de nombres como `std`, evitando colisiones.

Y una diferencia de filosofía: C confía en el programador y le da cuerda suficiente para ahorcarse; C++ moderno intenta darte la misma potencia con redes de seguridad opcionales. Puedes seguir programando "como en C" dentro de C++, pero el lenguaje te invita a usar sus herramientas modernas.

Error común de principiantes: tratar C++ como "C con cout". Muchos programadores que vienen de C siguen usando arreglos crudos en vez de vector, `char*` en vez de string, `malloc` en vez de `new`. Compila, sí, pero pierden todo lo bueno del lenguaje. En este libro aprenderás la forma moderna desde el inicio.

Clase 3: Tu compilador y tu primer programa

C++ es un lenguaje compilado. Eso significa que no escribes y ejecutas directamente: escribes código fuente en un archivo de texto, un programa llamado compilador lo traduce a código máquina, y luego ejecutas el resultado. Este proceso tiene dos grandes ventajas: el programa final es rapidísimo y el compilador revisa tu código buscando errores antes de que se ejecute.

El compilador que usaremos es `g++`, el compilador de C++ de la colección GNU. Es gratuito, es el estándar en la industria y está disponible en todos los sistemas.

En Linux, casi siempre ya viene instalado o se instala con un comando. En Ubuntu o Debian, abre la terminal y escribe: `sudo apt install g++`. En Fedora: `sudo dnf install gcc-c++`. Para verificar que quedó bien, escribe `g++ --version` y deberías ver un número de versión.

En macOS, instala las herramientas de desarrollo de Apple con el comando `xcode-select --install` en la terminal. Eso incluye `clang++`, que funciona igual que `g++` para lo que haremos. Puedes usar `g++` como alias o directamente `clang++`.

En Windows, la forma más sencilla es instalar MinGW-w64, que te da `g++` para Windows. Descárgalo de su sitio oficial, instálalo y agrega su carpeta `bin` al `PATH` del sistema. Alternativa moderna: instala WSL (Subsistema de Windows para Linux) y trabaja en Linux dentro de Windows, que es lo que muchos profesionales hacen. Otra opción es usar un IDE como Code::Blocks o Visual Studio Community, que traen compilador incluido.

Para editar código puedes usar cualquier editor de texto plano: Visual Studio Code (el favorito, gratuito), o incluso el bloc de notas en emergencias. Lo importante es que el archivo se guarde como texto plano con extensión `.cpp`.

El ritual de compilación tiene tres pasos. Primero escribes tu programa en un archivo, por ejemplo `hola.cpp`. Luego lo compilas: `g++ hola.cpp -o hola`. Eso genera un ejecutable llamado `hola` (o `hola.exe` en Windows). Luego lo ejecutas: en Linux y macOS escribes `./hola`, en Windows escribes `hola.exe`. Si hay errores, el compilador te los muestra y no genera el ejecutable: los corriges y repites.

El programa más famoso de la historia es este:

```
#include <iostream>

int main() {
    std::cout << "Hola, mundo!" << std::endl;
    return 0;
}
```

Vamos línea por línea. `#include <iostream>` le dice al compilador: "tráeme la biblioteca de entrada y salida". Es como pedir las herramientas antes de trabajar. `int main()` es la función principal: todo programa en C++ empieza a ejecutarse aquí. Las llaves `{ }` delimitan el cuerpo de la función, el bloque de código que se ejecuta.

`std::cout` es el objeto de "salida estándar" (normalmente la pantalla). El operador `<<` le "envía" datos: `std::cout << "Hola, mundo!"` significa "manda este texto a la pantalla". `std::endl` es "fin de línea": baja el cursor al siguiente renglón. `return 0` le dice al sistema operativo que el programa terminó bien; por convención, cero significa éxito.

El prefijo `std::` indica que `cout` y `endl` viven en el espacio de nombres `std` (standard). Más adelante aprenderás que puedes escribir `using namespace std;` para no repetirlo, con sus ventajas y sus riesgos.

Error común número uno: olvidar el punto y coma. En C++, cada instrucción termina con `;`. Si lo olvidas, el compilador se queja con un mensaje que a veces apunta a la línea siguiente. Cuando veas un error raro, revisa primero si te falta un punto y coma en la línea anterior.

Error común número dos: guardar el archivo con extensión equivocada (`.txt` en vez de `.cpp`) o compilar el archivo equivocado. Si `g++` dice que no encuentra el archivo, revisa en qué carpeta estás con `pwd` y qué archivos hay con `ls` (o dir en Windows).

Buena práctica desde el día uno: compila con advertencias activadas. El comando `g++ -Wall -Wextra hola.cpp -o hola` le pide al compilador que te avise de código sospechoso. Los profesionales siempre compilan así; te ahorra horas de depuración.

Ejemplos paso a paso

Ejemplo 1: Tu primer programa, compilado a mano

Guarda esto como `hola.cpp`:

```
#include <iostream>

int main() {
    std::cout << "Hola, mundo!" << std::endl;
    std::cout << "Estoy aprendiendo C++ en Lucas EGA Academy." << std::endl;
    return 0;
}
```

Compila: `g++ hola.cpp -o hola`. Ejecuta: `./hola`. Verás las dos líneas en pantalla. Felicidades: ya eres programador de C++.

Observa que puedes encadenar varios `<<` en una sola línea: `std::cout << "Hola" << " " << "mundo" << std::endl;` funciona igual. Cada `<<` manda lo siguiente a la salida.

Ejemplo 2: Primer programa con variables

```
#include <iostream>

int main() {
    int edad = 25;
    std::cout << "Tengo " << edad << " años." << std::endl;
    edad = 26;
    std::cout << "El proximo año tendre " << edad << "." << std::endl;
    return 0;
}
```

Aquí declaramos una variable entera llamada `edad` con valor inicial 25. Nota cómo `cout` mezcla texto y variables sin formatos raros: `"Tengo " << edad << " años."`. El compilador sabe que `edad` es un número y lo convierte solo. En C habrías necesitado `printf("Tengo %d años.", edad);` con el riesgo de equivocarte en `%d`.

Ejemplo 3: Leyendo datos del usuario

```
#include <iostream>
#include <string>

int main() {
    std::string nombre;
    std::cout << "Como te llamas? ";
    std::cin >> nombre;
    std::cout << "Hola, " << nombre << ". Bienvenido a C++." << std::endl;
    return 0;
}
```

std::cin es la entrada estándar (el teclado). El operador >> "extrae" datos: std::cin >> nombre lee lo que el usuario escriba y lo guarda en nombre. Usamos std::string para el texto, así que necesitamos #include <string>. Prueba escribir tu nombre y ver el saludo personalizado.

Nota el detalle: std::cin >> nombre lee hasta el primer espacio. Si escribes "Juan Perez", solo guarda "Juan". Es una limitación que resolveremos en la unidad 5 con getline.

Ejemplo 4: Compilando con advertencias y estándar moderno

A partir de ahora, compila siempre así:

```
g++ -Wall -Wextra -std=c++17 programa.cpp -o programa
```

-Wall y -Wextra activan advertencias. -std=c++17 le dice al compilador que use el estándar moderno de C++. Si tu g++ es muy viejo y no conoce c++17, usa -std=c++11 como mínimo, pero lo ideal es actualizar el compilador.

Prueba este programa con un error intencional para ver cómo te avisa el compilador:

```
#include <iostream>

int main() {
    int numero = 10;
    std::cout << numero << std::endl;
    return 0;
}
```

Compila bien. Ahora quita el punto y coma de la línea de cout y compila de nuevo: verás un error. Acostúmbrate a leer estos mensajes desde ahora; en la unidad 10 te enseñaremos a interpretarlos como un profesional.

Práctica

Ejercicio 1

Escribe un programa que muestre en pantalla tu nombre, tu ciudad y tu lenguaje de programación favorito, cada dato en una línea. Usa tres instrucciones cout separadas.

Solución:

```
#include <iostream>

int main() {
    std::cout << "Me llamo Esteban." << std::endl;
    std::cout << "Vivo en Ciudad de Mexico." << std::endl;
    std::cout << "Mi lenguaje favorito es C++." << std::endl;
    return 0;
}
```

Explicación: cada cout con std::endl baja una línea. Podrías hacerlo en un solo cout encadenando, pero tres instrucciones separadas son más claras para empezar. Cambia los datos por los tuyos y compila.

Ejercicio 2

Modifica el programa anterior para que pida tu nombre con cin y luego lo salude. Declara la variable como std::string.

Solución:

```
#include <iostream>
#include <string>

int main() {
    std::string nombre;
    std::cout << "Escribe tu nombre: ";
    std::cin >> nombre;
    std::cout << "Hola, " << nombre << "!" << std::endl;
    std::cout << "Vivo en Ciudad de Mexico." << std::endl;
    std::cout << "Mi lenguaje favorito es C++." << std::endl;
    return 0;
}
```

Explicación: no olvides `#include <string>` para usar `std::string`. El orden es: declarar, pedir, leer, usar. Este patrón lo repetirás cientos de veces.

Ejercicio 3

Escribe un programa que declare dos variables enteras, `a = 15` y `b = 4`, y muestre su suma, su resta y su multiplicación usando `cout`.

Solución:

```
#include <iostream>

int main() {
    int a = 15;
    int b = 4;
    std::cout << "Suma: " << a + b << std::endl;
    std::cout << "Resta: " << a - b << std::endl;
    std::cout << "Multiplicacion: " << a * b << std::endl;
    return 0;
}
```

Explicación: puedes poner operaciones directamente dentro del `cout`: `a + b` se calcula primero y el resultado se envía a la salida. Es una probadita de los operadores que veremos en la unidad 2.

Ejercicio 4

Investiga en tu computadora: ejecuta `g++ --version` y anota qué versión tienes. Luego compila un programa usando la bandera `-std=c++17`. Si tu compilador no la acepta, investiga cómo actualizarlo en tu sistema operativo.

Solución: no hay código único; lo importante es el proceso. En Ubuntu: `sudo apt update && sudo apt install g++`. En Windows con MinGW-w64, descarga la versión más reciente. En macOS: `xcode-select --install`. Verifica con `g++ --version` que el número sea 8 o mayor. Este ejercicio parece administrativo, pero un entorno desactualizado causa la mitad de los problemas misteriosos de los principiantes.

Ejercicio 5

Escribe un programa que muestre un dibujo con caracteres ASCII, por ejemplo una carita o una casa, usando varios cout. Luego agrega al final una línea que pida tu edad y la muestre de vuelta.

Solución:

```
#include <iostream>

int main() {
    std::cout << " /\_/\ " << std::endl;
    std::cout << " ( o.o ) " << std::endl;
    std::cout << " > ^ < " << std::endl;
    int edad;
    std::cout << "Cuantos años tienes? ";
    std::cin >> edad;
    std::cout << "Tienes " << edad << " años. ¡Bienvenido a C++!" << std::endl;
    return 0;
}
```

Explicación: fíjate en los detalles: para mostrar una diagonal invertida \ hay que escribir \\ porque la diagonal sola es un carácter especial. La variable edad se declara sin valor inicial y se llena con cin. Este ejercicio combina todo lo de la unidad: salida, entrada y variables.

Resumen de la unidad

- C++ nació en 1979 como "C con clases" y hoy es el lenguaje del software de alto rendimiento: videojuegos, sistemas operativos, navegadores y finanzas.
- Sus ventajas: velocidad de código máquina, control total de la memoria y el entendimiento profundo de cómo funciona la computadora.
- Diferencias clave con C: programación orientada a objetos, iostream en vez de printf/scanf, std::string, la STL, excepciones, sobrecarga y espacios de nombres.
- El flujo de trabajo: escribir en .cpp, compilar con g++ archivo.cpp -o programa, ejecutar con ./programa.
- Compila siempre con g++ -Wall -Wextra -std=c++17 para que el compilador te ayude a detectar errores.
- Todo programa empieza en int main(); cout muestra datos con <<, cin lee datos con >>, y cada instrucción termina con punto y coma.

- En la siguiente unidad entrarás a los fundamentos: variables, tipos de datos, operadores y entrada/salida a fondo.

Profundización: cómo piensa un compilador de C++

En la primera parte de esta unidad instalaste g++ y compilaste tu primer programa. Ahora vamos a entender qué hace realmente el compilador, porque cuando algo falle (y fallará), saber qué pasa adentro te convertirá en detective en vez de víctima.

Compilar no es un solo paso: son cuatro. Cuando escribes g++ hola.cpp -o hola, ocurren en secreto: preprocesamiento, compilación, ensamblado y enlazado.

El preprocesamiento maneja las líneas que empiezan con #. #include <iostream> literalmente copia el contenido de ese archivo en tu código. #define crea macros de texto. Puedes ver el resultado con g++ -E hola.cpp: verás tu código expandido a miles de líneas. Por eso compilar tarda: iostream es enorme.

La compilación traduce tu C++ a lenguaje ensamblador (código casi-máquina legible). Con g++ -S hola.cpp generas hola.s y puedes espiar. No necesitas entender ensamblador, pero saber que existe explica los mensajes de error: el compilador detecta problemas aquí, antes de generar nada.

El ensamblado convierte a código objeto (.o): binario que la máquina entiende pero aún incompleto. El enlazado (linking) une tu código objeto con las bibliotecas (como la implementación de cout) y produce el ejecutable final. Los errores de "undefined reference" ocurren aquí: tu código compiló bien, pero al juntar las piezas falta algo.

Los errores del compilador, traducidos

Los mensajes de g++ asustan al inicio, pero siguen patrones. Aprende a leerlos:

"error: expected ';' before..." significa que falta un punto y coma, casi siempre en la línea anterior a la indicada. "error: 'x' was not declared in this scope" significa que usaste una variable que no existe o que olvidaste incluir una cabecera. "error: no matching function for call" significa que llamaste una función con argumentos que no coinciden con ninguna versión. "undefined reference to ..." es error de enlazado: declaraste algo que nunca definiste, típico con funciones prototipo sin cuerpo.

Regla de oro: lee solo el primer error. Los compiladores se confunden: un error real genera cascadas de errores falsos abajo. Corrige el primero y recompila.

Advertencias: tu mejor amigo

-Wall activa advertencias comunes, -Wextra agrega más. Prueba este código:

```
#include <iostream>

int main() {
    int x;
    if (x = 5) {
        std::cout << "x es 5" << std::endl;
    }
    return 0;
}
```

Sin banderas compila en silencio. Con -Wall -Wextra, g++ te avisa de dos cosas: que x se usa sin inicializar y que la asignación dentro del if es sospechosa. Esos dos avisos valen horas de depuración. Los profesionales agregan -Werror, que convierte advertencias en errores: el programa no compila hasta que el código esté limpio.

El estándar de C++: por qué importa -std=c++17

C++ evoluciona por estándares. Cada uno agrega herramientas: C++11 trajo auto y los for de rango; C++14 pulió detalles; C++17 trajo mejoras a la STL; C++20 trajo conceptos y rangos. Sin -std=c++17, tu g++ usa un estándar viejo por defecto y el código moderno no compila o se comporta raro.

Verifica tu estándar con un truco:

```
#include <iostream>

int main() {
    std::cout << __cplusplus << std::endl;
    return 0;
}
```

Si imprime 201703, estás en C++17. Si imprime 201103, estás en C++11: funciona para este libro, pero actualiza tu compilador cuando puedas.

Editores y flujo de trabajo profesional

Visual Studio Code con la extensión de C++ es el estándar actual: resalta sintaxis, autocompleta y muestra errores mientras escribes. Configura la terminal integrada para compilar con un atajo. Alternativas: CLion (potente, de pago), Code::Blocks (simple, gratuito), o Vim/Emacs si te gusta la terminal pura.

Organiza tus programas: una carpeta por proyecto, archivos .cpp con nombres descriptivos, y un archivo LEEME.txt con cómo compilar. Parece exagerado para programas de 20 líneas, pero el hábito te salva cuando llegues a proyectos de 20 archivos.

Más ejemplos

Ejemplo: preprocesador en acción

```
#include <iostream>
#define PI 3.14159
#define CUADRADO(x) ((x) * (x))

int main() {
    double radio = 5.0;
    std::cout << "Area: " << PI * CUADRADO(radio) << std::endl;
    return 0;
}
```

#define hace sustitución de texto antes de compilar. PI se reemplaza por 3.14159 en todo el código. CUADRADO(x) es una macro con parámetro: CUADRADO(radio) se convierte en ((radio) * (radio)). Los paréntesis extra evitan sorpresas con la precedencia. En C++ moderno se prefiere const y funciones inline sobre #define, pero verás macros en código heredado y debes entenderlas.

Ejemplo: varios archivos, un programa

Crea saludos.h:

```
#ifndef SALUDOS_H
#define SALUDOS_H
#include <string>
void saludar(const std::string& nombre);
#endif
```

Crea saludos.cpp:

```
#include <iostream>
#include "saludos.h"

void saludar(const std::string& nombre) {
    std::cout << "Hola, " << nombre << "!" << std::endl;
}
```

Crea main.cpp:

```
#include "saludos.h"

int main() {
    saludar("Mundo");
    return 0;
}
```

Compila todo junto: `g++ main.cpp saludos.cpp -o programa`. Las líneas `#ifndef/#define/#endif` son "guardas de inclusión": evitan que el `.h` se incluya dos veces. Nota las comillas en `#include "saludos.h"`: comillas para tus archivos, `< >` para los del sistema. Este es el esqueleto de todo proyecto C++ grande.

Ejemplo: depurando con impresiones

```
#include <iostream>

int main() {
    int a = 10, b = 0;
    std::cerr << "[DEBUG] a=" << a << ", b=" << b << std::endl;
    if (b != 0) {
        std::cout << a / b << std::endl;
    } else {
        std::cerr << "[DEBUG] division evitada" << std::endl;
    }
    return 0;
}
```

`std::cerr` es el flujo de error: como `cout` pero para diagnósticos. La técnica más antigua y efectiva de depuración es imprimir valores intermedios. Cuando algo falle, imprime lo que crees que valen tus variables: la mitad de las veces descubrirás que valen otra cosa.

Errores típicos de esta unidad y cómo evitarlos

1. Compilar el archivo equivocado: editaste main.cpp pero compilaste viejo.cpp. Verifica el nombre en el comando.
2. Guardar como .txt: el editor a veces agrega extensiones. El archivo debe terminar en .cpp.
3. Ejecutar sin compilar: si hubo errores, el ejecutable es viejo o no existe. Recompila siempre después de corregir.
4. Permiso denegado al ejecutar en Linux: el archivo necesita permiso de ejecución; g++ normalmente lo pone, pero si copias el ejecutable puede perderse. `chmod +x programa` lo arregla.
5. Mezclar compiladores: compilar parte con g++ y parte con clang++ puede dar errores de enlazado. Usa uno solo por proyecto.

Práctica adicional

Ejercicio A

Escribe un programa que muestre tu nombre usando una macro `#define NOMBRE "tu nombre"` y una constante `const std::string`. Reflexiona: ¿cuál prefieres y por qué?

Solución:

```
#include <iostream>
#include <string>
#define NOMBRE_MACRO "Esteban"

int main() {
    const std::string NOMBRE_CONST = "Esteban";
    std::cout << NOMBRE_MACRO << std::endl;
    std::cout << NOMBRE_CONST << std::endl;
    return 0;
}
```

Explicación: la constante es mejor: tiene tipo, respeta el alcance y aparece en el depurador. La macro es sustitución ciega de texto. En C++ moderno, prefiere `const`.

Ejercicio B

Divide un programa en dos archivos: uno con la función `double aFahrenheit(double c)` y otro con el `main` que la usa. Compílalos juntos.

Solución: conversion.h con el prototipo y guardas; conversion.cpp con la definición usando la fórmula $c * 9.0/5.0 + 32$; main.cpp que incluye el .h y prueba con 0, 100 y 37 grados. Compila: `g++ main.cpp conversion.cpp -o conversor`. Este ejercicio te prepara para proyectos reales.

Ejercicio C

Provoca a propósito cada uno de estos errores, anota el mensaje exacto de g++ y luego corrígelo: falta punto y coma, variable no declarada, #include con nombre mal escrito, función llamada con argumentos equivocados.

Solución: no hay código fijo; el valor está en el proceso. Guarda tus anotaciones: serán tu diccionario personal de errores. Con el tiempo reconocerás cada mensaje al instante.

Ejercicio D

Escribe un programa que pida dos números e imprima mensajes de depuración en cerr mostrando los valores leídos, la suma y el producto. Luego ejecútalo redirigiendo la salida normal a un archivo pero dejando los errores en pantalla: `./programa > salida.txt`. Observa cómo cerr sigue visible.

Solución:

```
#include <iostream>

int main() {
    double a, b;
    std::cin >> a >> b;
    std::cerr << "[DEBUG] leídos: " << a << ", " << b << std::endl;
    std::cout << "Suma: " << a + b << std::endl;
    std::cout << "Producto: " << a * b << std::endl;
    return 0;
}
```

Explicación: `>` redirige solo cout; cerr sigue yendo a la terminal. Por eso los profesionales separan salida normal de diagnósticos: puedes guardar resultados y seguir viendo qué pasa adentro.

Expansión Unidad 1: estándares, ecosistema y tu programa frente al sistema

En las dos primeras partes de esta unidad instalaste tu compilador, entendiste las cuatro fases de la compilación y corriste tu primer programa. Esta tercera parte te lleva más lejos: vas a conocer la historia viva del lenguaje a través de sus estándares, vas a entender cuándo conviene usar C++ y cuándo otro lenguaje es mejor opción, vas a desarmar un programa pieza por pieza hasta conocer su anatomía completa, vas a instalar tu entorno en Windows paso a paso como lo hacen los profesionales, vas a entender qué pasa cuando tu programa se enlaza con bibliotecas, y vas a enseñarle a tu programa a recibir órdenes desde la línea de comandos y a leer el ambiente del sistema operativo. Al final tendrás una visión completa de lo que significa programar en C++ en el mundo real, no solo en ejercicios de libro.

Esta parte es larga a propósito. Léela con calma, compila cada ejemplo y juega con ellos. Lo que aprendas aquí te va a servir durante todo el libro y durante toda tu carrera: los estándares cambian cada tres años, los sistemas operativos cambian cada década, pero la forma de pensar que vas a construir aquí no caduca.

La evolución de los estándares: C++11, 14, 17, 20 y 23

C++ no es un lenguaje congelado en el tiempo. Cada tres años, el comité internacional de estandarización (el ISO C++ Committee) publica una nueva versión del estándar: un documento que define exactamente qué debe aceptar un compilador y cómo debe comportarse cada herramienta del lenguaje. Cuando en la primera parte de esta unidad usaste la bandera `-std=c++17`, le estabas diciendo a g++: "compórtate según las reglas publicadas en 2017". Entender qué trajo cada estándar te ayuda a leer código ajeno (que puede usar estilos de distintas épocas), a elegir las herramientas correctas y a no asustarte cuando veas sintaxis que todavía no conoces.

Piensa en los estándares como las generaciones de un mismo carro: el modelo 2026 sigue siendo el mismo carro que el de 1998, pero trae frenos ABS, cámara de reversa y pantalla táctil que el viejo no tenía. El motor fundamental no cambió, pero manejar es más cómodo y más seguro.

Antes de la era moderna: C++98 y C++03

El primer estándar oficial, C++98, unificó lo que los compiladores hacían cada uno a su manera. C++03 fue solo una corrección de errores del anterior. Durante esos años, C++ ganó fama de ser un lenguaje potente pero verboso y lleno de trampas: no había una forma cómoda

de recorrer una colección, no había punteros inteligentes en la biblioteca estándar y escribir código genérico era un dolor de cabeza. Mucha de la mala fama de C++ ("es complicadísimo", "todo se rompe") viene de esa época. Lo importante es que sepas que el C++ que estás aprendiendo en este libro ya no es ese: es el C++ moderno, que empezó con la revolución de 2011.

C++11: la gran revolución moderna

C++11 fue el cambio más grande en la historia del lenguaje. Si C++98 fue "poner orden", C++11 fue "reinventar la casa". Trajo tantas herramientas nuevas que muchos programadores dicen que C++11 es en realidad un lenguaje nuevo que conservó el nombre por compatibilidad. Estas son las joyas que trajo y que usas en este libro desde el primer día:

La palabra clave `auto`: le pide al compilador que deduzca el tipo de una variable a partir de su valor inicial. `auto x = 5;` crea un `int` sin que tengas que escribirlo. Parece un detalle menor, pero cuando los tipos se vuelven largos (como los iteradores de la STL), `auto` te ahorra líneas enteras y evita errores de dedo. Eso sí: úsalo cuando el tipo sea obvio por el contexto; si `auto` esconde información importante, mejor escribe el tipo explícito.

El ciclo `for` de rango: antes de C++11, recorrer un arreglo requería manejar índices a mano, con el riesgo clásico de salirte del rango. C++11 trajo la forma moderna: `for (int n : numeros) { ... }`, que recorre cada elemento sin que toques un índice. Es más corto, más legible y elimina de un plumazo los errores de límites. Lo usarás muchísimo a partir de la unidad 4.

La palabra `nullptr`: antes se usaba `NULL` o el número 0 para representar un puntero vacío, lo cual causaba ambigüedades peligrosas (¿es el número cero o es "ningún puntero"?). `nullptr` es un valor con tipo propio que significa exactamente "puntero a nada", y el compilador lo distingue del cero. Parece sutil, pero eliminó una familia entera de bugs.

La inicialización uniforme con llaves: la sintaxis `int x{5};` que ya conoces. C++11 la introdujo como la forma preferida de inicializar porque impide los "estrechamientos" silenciosos (meter 3.99 en un `int` sin que nadie te avise). Es la forma preferida en este libro.

Las funciones lambda: pequeñas funciones anónimas que puedes escribir en el lugar donde las necesitas. Antes de C++11, pasarle una operación personalizada a un algoritmo de la STL requería escribir una función completa en otro lugar del archivo. Con lambdas, la lógica vive justo donde se usa. Las verás en acción en las unidades de algoritmos.

Los punteros inteligentes (`unique_ptr` y `shared_ptr`): antes, manejar memoria dinámica era manual y peligroso: si olvidabas liberar, tenías una fuga de memoria. C++11 metió a la biblioteca estándar estos punteros que liberan la memoria solos cuando ya no se necesitan.

Son la red de seguridad del C++ moderno y los estudiarás a fondo en la unidad de memoria dinámica.

La inferencia de tipos en el valor de retorno, el `enum class` que ya viste, la palabra `static_assert` para verificaciones en tiempo de compilación y la biblioteca de hilos `<thread>` también nacieron aquí. C++11 convirtió a C++ en un lenguaje moderno de verdad.

C++14: puliendo lo moderno

C++14 fue un estándar de "mantenimiento": no trajo revoluciones, sino que pulió lo que C++11 había introducido y facilitó la vida diaria. Sus aportes más útiles:

Los literales binarios: antes de C++14 podías escribir números en decimal, octal y hexadecimal, pero no en binario. C++14 agregó el prefijo `0b`, así que `0b1010` significa diez en binario. Parece un capricho, pero cuando trabajas con bits (como harás en la expansión de la unidad 2), escribir los bits directamente es mucho más claro que convertir mentalmente desde hexadecimal.

Los separadores de dígitos con apóstrofe: `1'000'000` se lee mucho mejor que `1000000`. El compilador ignora los apóstrofes; son solo para tus ojos. En programas financieros o científicos, donde los números grandes son pan de cada día, esto evita errores de conteo de ceros.

Las lambdas genéricas: en C++11 los parámetros de una lambda tenían tipos fijos; en C++14 puedes usar `auto` en los parámetros y la misma lambda sirve para enteros, decimales o textos. También se relajaron las reglas de `constexpr`, permitiendo más lógica dentro de funciones evaluadas en tiempo de compilación.

C++14 también trajo `std::make_unique`, el compañero que faltaba para crear punteros inteligentes únicos de forma segura. En resumen: si C++11 fue la revolución, C++14 fue la versión "ya sin bugs de diseño".

C++17: el estándar de este libro

C++17 es la base de este libro, y no es casualidad: es el estándar que la mayoría de los compiladores dominan por completo y que la industria adoptó masivamente. Trajo herramientas que usas todos los días sin darte cuenta:

Los enlaces estructurados (structured bindings): permiten "desempacar" varios valores de una sola vez. Si una función devuelve un par de valores, puedes escribir `auto [nombre, edad] = obtenerDatos();` y obtienes dos variables de un golpe. Antes había que usar `std::tie` o acceder

a `.first` y `.second`, mucho más torpe. Lo verás brillar cuando trabajemos con mapas y pares en unidades avanzadas.

El `if` con inicializador: puedes declarar una variable dentro de la condición de un `if` y esa variable solo existe dentro del `if`. Por ejemplo: `if (int resultado = calcular(); resultado > 0) { ... }`. Esto mantiene las variables cerca de donde se usan y evita que anden sueltas por el resto de la función. Es una de esas pequeñas elegancias que hacen el código más limpio.

La declaración `if constexpr`: permite elegir en tiempo de compilación qué rama de código se compila, según el tipo de dato. Es una herramienta avanzada (la exprimirás en la unidad de plantillas), pero desde ahora debes saber que existe: es la base de mucho código genérico moderno.

La clase `std::string_view`: una forma de "mirar" un texto sin copiarlo. Antes, pasar un `string` a una función que solo lo iba a leer implicaba copias costosas o referencias con sintaxis incómoda. `string_view` es una vista ligera: apunta al texto original sin duplicarlo. En programas que procesan mucho texto, la diferencia de velocidad es notable.

Los algoritmos paralelos de la STL: por primera vez, la biblioteca estándar permite decirle a un algoritmo "ejecuta esto usando varios núcleos del procesador" con una simple política de ejecución. El hardware moderno tiene muchos núcleos; C++17 te deja aprovecharlos sin escribir código de hilos a mano.

También llegaron `std::optional` (un valor que puede no existir, sin usar punteros ni valores mágicos como `-1`), `std::variant` (una variable que puede guardar uno de varios tipos) y `std::filesystem` (una forma portable de trabajar con archivos y carpetas, la misma en Windows, Linux y macOS). La biblioteca `<filesystem>` por sí sola vale el estándar: antes, listar archivos de una carpeta requería código distinto en cada sistema operativo.

C++20: conceptos, rangos y módulos

C++20 fue otra revolución, comparable a C++11. Sus cuatro grandes aportes cambian la forma de escribir C++:

Los conceptos (concepts): por fin puedes decirle al compilador "esta plantilla solo acepta tipos que se puedan sumar" y obtener un mensaje de error claro si alguien pasa algo indebido. Antes de C++20, un error en una plantilla producía páginas de mensajes incomprensibles. Los conceptos son como un contrato legible entre tu código genérico y quien lo usa.

La biblioteca de rangos (ranges): una forma nueva y más expresiva de trabajar con secuencias de datos. En vez de pasar pares de iteradores (principio y fin) a los algoritmos, trabajas con el rango completo y puedes encadenar operaciones con el operador `|`. Es un estilo más declarativo: dices qué quieres, no cómo recorrer.

Los módulos: el reemplazo moderno de `#include`. En vez de copiar textualmente el contenido de una cabecera (con el costo de compilación que ya viste en la primera parte), un módulo se compila una vez y se importa rápido. Todavía están en adopción, pero son el futuro de la organización del código en C++.

Las corrutinas: funciones que pueden pausarse y reanudarse. Son la base para escribir código asíncrono (tareas que esperan sin bloquear) de forma legible. Es tema avanzado, pero conviene saber que el lenguaje ya lo trae de serie.

C++20 también trajo el operador de comparación de tres vías (`<=>`, "nave espacial"), `std::format` (formato de texto moderno que por fin jubila a los manipuladores crípticos) y mejoras a `constexpr` que permiten aún más cómputo en tiempo de compilación.

C++23: lo más nuevo y hacia dónde vamos

C++23 es el estándar más reciente publicado (y al momento de escribir este libro, los compiladores todavía están terminando de implementarlo). Sus aportes más prácticos: `std::expected` (una forma elegante de devolver "un valor o un error" sin excepciones), mejoras a los rangos, `std::print` (imprimir en pantalla por fin sin tanto ceremonial) y más herramientas de la biblioteca. No lo usamos en este libro porque el soporte aún no es universal, pero en un par de años será lo normal.

La lección importante de esta historia: C++ evoluciona sin romper el pasado. Un programa escrito en 1998 compila hoy. Esa compatibilidad hacia atrás es una de las razones por las que la industria confía en C++ para proyectos que deben vivir décadas: tu código no caduca.

Cómo saber qué estándar estás usando y cómo cambiarlo

Ya viste el truco de `__cplusplus` en la primera parte: imprime 201703L para C++17, 202002L para C++20. La bandera `-std=` es tu palanca: `-std=c++17`, `-std=c++20`, etc. Si no la pasas, `g++` usa un estándar por defecto que depende de su versión (los `g++` modernos usan C++17 o C++20 por defecto, pero no cuentes con eso: sé explícito siempre).

Ejemplo completo que muestra qué trae cada época. Guárdalo como `estandares.cpp` y compílalo con `-std=c++17`:

```
#include <iostream>
```

```

#include <string>
#include <string_view>
#include <vector>

// C++11: auto, for de rango, nullptr, inicializacion con llaves
// C++14: literales binarios, separadores de digitos
// C++17: enlaces estructurados, if con inicializador, string_view

int* buscarMayor(std::vector<int>& datos) {
    if (datos.empty()) {
        return nullptr; // C++11: puntero vacio con tipo propio
    }
    int* mayor = &datos[0];
    for (int& n : datos) { // C++11: for de rango, sin indices
        if (n > *mayor) {
            mayor = &n;
        }
    }
    return mayor;
}

void mostrar(std::string_view texto) { // C++17: vista sin copiar
    std::cout << texto << std::endl;
}

int main() {
    auto limite{1'000'000}; // C++11: auto + C++14: separador '
    std::cout << "Limite: " << limite << std::endl;

    std::vector<int> numeros{5, 2, 9, 1, 7}; // C++11: llaves uniformes

    // C++17: if con inicializador, la variable solo vive dentro del if
    if (int* p = buscarMayor(numeros); p != nullptr) {
        std::cout << "Mayor: " << *p << std::endl;
    }

    // C++17: enlaces estructurados para desempacar un par
    auto par = std::make_pair(std::string("C++"), 17);
    auto [lenguaje, version] = par;
    std::cout << lenguaje << version << std::endl;

    int mascara{0b1111'0000}; // C++14: binario + separador
    std::cout << "Mascara: " << mascara << std::endl;

    mostrar("string_view no copia el texto"); // C++17
}

```

```
    return 0;
}
```

Vamos línea por línea por las partes nuevas. Los `#include` traen `vector` (arreglo dinámico de la STL, unidad 4) y `string_view` (vista de texto de C++17). La función `buscarMayor` recibe un vector por referencia y devuelve un puntero al elemento mayor, o `nullptr` si está vacío: `nullptr` (C++11) deja claro que es "ningún puntero", no el número cero. El `for` de rango `for (int& n : datos)` recorre cada elemento sin índices: la `&` significa que `n` es una referencia al elemento real, no una copia. En el `main`, `auto limite{1'000'000}` combina deducción de tipo (C++11) con separador de dígitos (C++14): el compilador deduce `int` e ignora los apóstrofes. El `if` con inicializador de C++17 declara `p` dentro de la condición y `p` solo existe dentro del `if`: no contamina el resto de la función. Los enlaces estructurados `auto [lenguaje, version] = par;` desempacan el par en dos variables con nombre en una sola línea. El literal `0b1111'0000` combina binario (C++14) con separador visual. Y `mostrar` recibe `string_view`: apunta al texto literal sin copiarlo.

Compíllalo así: `g++ -Wall -Wextra -std=c++17 estandares.cpp -o estandares`. Si lo compilas con `-std=c++11` verás errores en las partes de C++14 y C++17: es la prueba viva de que el estándar sí importa. Juega a cambiar la bandera y observa qué se rompe; es la forma más rápida de sentir la evolución del lenguaje en tus manos.

C++ frente a Python, Java y JavaScript: cuándo elegir cada uno

Una de las preguntas más comunes de quien empieza es: "¿por qué no aprender Python, que es más fácil?". Es una gran pregunta y merece una respuesta honesta, no fanática. Ningún lenguaje es "el mejor" en abstracto: cada uno es la mejor herramienta para ciertos trabajos. Un buen programador no se casa con un lenguaje; elige según el problema. Vamos a comparar con lupa.

La comparativa honesta

C++ es compilado a código máquina, con tipado estático fuerte y manejo manual (pero asistido) de memoria. Su punto fuerte es el rendimiento máximo y el control total del hardware. Su costo es la complejidad: tú eres responsable de la memoria, de los tipos y de muchos detalles que otros lenguajes esconden. Brilla en videojuegos, sistemas operativos, motores de bases de datos, trading de alta frecuencia, robótica y en el núcleo de las bibliotecas de inteligencia artificial.

Python es interpretado, con tipado dinámico: las variables no declaran tipo y el tipo se decide mientras el programa corre. Su punto fuerte es la velocidad de desarrollo: escribes menos

líneas y pruebas ideas rapidísimo. Tiene el ecosistema más grande para ciencia de datos, inteligencia artificial aplicada, automatización y scripting. Su costo es la velocidad de ejecución (mucho más lento que C++) y el consumo de memoria. Detalle curioso y relevante: las bibliotecas de IA más famosas (las que usas desde Python) tienen su motor interno escrito en C++: Python pone la cara amable, C++ hace el trabajo pesado.

Java es compilado a un código intermedio (bytecode) que corre sobre una máquina virtual (la JVM). Su punto fuerte es "escribe una vez, corre en todas partes" y un ecosistema empresarial gigantesco: bancos, sistemas corporativos y Android (históricamente) viven en Java. Tiene recolección de basura: no manejas la memoria a mano. Su costo es el consumo de memoria (la JVM pesa) y un arranque más lento. Es más verboso que Python pero más seguro que C++ para un principiante, porque la máquina virtual atrapa muchos errores.

JavaScript es el lenguaje de la web: corre en todos los navegadores del planeta y, con Node.js, también en servidores. Su punto fuerte es que no hay alternativa en el frontend web: si quieres que algo pase en una página, es JavaScript (o algo que compile a JavaScript). Su costo histórico es un sistema de tipos peculiar y decisiones de diseño de los años 90 que todavía se pagan. Con TypeScript encima se vuelve mucho más serio.

Tabla comparativa rápida

- Velocidad de ejecución: C++ (máxima) > Java (media-alta) > JavaScript (media) > Python (baja).
- Velocidad de desarrollo: Python (máxima) > JavaScript (alta) > Java (media) > C++ (baja al inicio).
- Control de memoria: C++ (total, manual) > Java (automático con GC) = JavaScript (automático) = Python (automático).
- Detección de errores de tipos: C++ y Java (en compilación) > JavaScript con TypeScript (en compilación) > Python y JavaScript puro (en ejecución).
- Dónde manda cada uno: C++ en rendimiento crítico; Python en datos, IA aplicada y scripts; Java en empresas y Android; JavaScript en la web.

Cuándo elegir C++ (y cuándo no)

Elige C++ cuando el rendimiento sea un requisito, no un lujo: motores de juego, audio y video en tiempo real, sistemas embebidos con poca memoria, infraestructura que atiende millones de peticiones, o cuando necesites hablar directamente con el hardware. Elige C++ también cuando quieras entender de verdad cómo funciona una computadora: lo que aprendas aquí te hará mejor programador en cualquier otro lenguaje.

No elijas C++ cuando necesites un prototipo para ayer, cuando el equipo no tenga experiencia con memoria manual, o cuando el ecosistema del problema esté en otro lado (una página web se hace en JavaScript, un análisis de datos exploratorio en Python). Usar C++ para todo sería como usar un trailer para ir por las tortillas: puede hacerlo, pero no es la herramienta adecuada.

Y una verdad que te va a dar tranquilidad: aprender C++ primero hace que los demás lenguajes se sientan fáciles después. Quien domina punteros, memoria y compilación entiende Python por dentro, no solo por fuera. El camino difícil al inicio es el camino corto a largo plazo.

Anatomía detallada de un programa en C++

Ya escribiste varios programas, pero hasta ahora los veías como un todo. Vamos a abrir el cofre y nombrar cada pieza, como un mecánico que señala cada parte del motor. Toma este programa completo y compilable, que usa todo lo que hemos visto:


```

#include <iostream>    // directiva del preprocesador
#include <string>

// Constante global: vive durante todo el programa
const double IVA{0.16};

// Declaracion anticipada (prototipo) de una funcion
double conIVA(double precio);

// Funcion principal: aqui empieza la ejecucion
int main() {
    // Comentario de una linea: pide el precio al usuario
    std::string producto;
    std::cout << "Producto: ";
    std::getline(std::cin, producto); // lee linea completa con espacios

    double precio{0.0};
    std::cout << "Precio sin IVA: ";
    std::cin >> precio;

    /* Comentario de varias lineas:
       calcula el total y lo muestra con formato */
    double total{conIVA(precio)};
    std::cout << producto << " con IVA: $" << total << std::endl;
    return 0;
}

// Definicion de la funcion declarada arriba
double conIVA(double precio) {
    return precio * (1.0 + IVA);
}

```

Las directivas del preprocesador

Las líneas que empiezan con `#` no son C++ como tal: son órdenes para el preprocesador, el primer trabajador de la cadena de compilación. `#include <iostream>` copia el contenido de la biblioteca de entrada y salida en tu archivo antes de compilar. Los `< >` significan "búscalo en las carpetas del sistema"; las comillas `" "` significan "búscalo primero en mi carpeta" (para tus propios archivos, como viste con `saludos.h`). El preprocesador no sabe de tipos ni de sintaxis: hace sustitución de texto a lo bestia. Por eso un `#include` mal escrito falla antes de que el compilador siquiera mire tu lógica.

Los comentarios y sus estilos

C++ tiene dos estilos de comentario, y cada uno tiene su momento. El comentario de una línea `//` comenta hasta el fin del renglón: úsalo para explicar el porqué de la línea siguiente o de la misma línea. El comentario de bloque `/* ... */` puede abarcar varios renglones: úsalo para explicar un algoritmo completo o para documentar una función. Los comentarios no llegan al ejecutable: el preprocesador los elimina.

Estilo profesional: comenta el porqué, no el qué. `// suma uno al contador no aporta nada (el código ya dice eso); // compensa el redondeo del sensor, que siempre mide de mas sí aporta.` Un buen comentario explica decisiones, no repite código. Y ojo con el error clásico: comentar código viejo "por si acaso" y dejarlo pudriéndose meses. Si no lo necesitas, bórralo: para eso existe el control de versiones.

Las funciones: declaración, definición y llamada

Todo programa en C++ es una colección de funciones. La función `main` es especial: es la puerta de entrada, donde el sistema operativo empieza a ejecutar. Las demás funciones son herramientas que `main` (u otras funciones) llaman.

Fíjate en el programa: arriba del `main` hay una declaración anticipada (prototipo): `double conIVA(double precio);`. Es una promesa: "existe una función llamada `conIVA` que recibe un `double` y devuelve un `double`; su cuerpo está en otro lado". Gracias a esa promesa, `main` puede llamarla aunque su definición esté hasta abajo del archivo. La definición es el cuerpo real, con las llaves y el `return`. Separar declaración y definición es lo que permite dividir programas en varios archivos: el `.h` lleva las promesas, el `.cpp` lleva los cuerpos.

Los parámetros (`double precio` dentro de los paréntesis) son las entradas; el `return` es la salida. Una función bien diseñada hace una sola cosa y su nombre lo dice: `conIVA` calcula el precio con IVA, ni más ni menos.

Las variables globales y su alcance

`const double IVA{0.16};` está fuera de cualquier función: es una variable global, visible desde cualquier parte del archivo que esté debajo de su declaración. Las globales constantes como esta son aceptables y comunes (tasas, límites, nombres del programa). Las globales que cambian (variables globales modificables) son otra historia: cualquier función puede alterarlas y rastrear quién cambió qué se vuelve una pesadilla en programas grandes. Regla práctica: constantes globales sí, variables globales que cambian casi nunca. En unidades avanzadas verás mejores alternativas.

Estilos de escritura: el código también se lee

El compilador ignora los espacios, los saltos de línea y la indentación: para él, todo el programa podría estar en una sola línea. Para los humanos, el formato lo es todo. Estos son los estilos que verás en el mundo real:

El estilo de llaves K&R (la llave de apertura en la misma línea del `if` o la función) es el que usa este libro y el más común en C++ moderno. El estilo Allman (llave de apertura en su propia línea) es popular en C# y también se ve en C++. Ninguno es "correcto" ante el compilador; lo que sí es correcto es ser consistente: no mezcles estilos en el mismo proyecto.

Los nombres siguen convenciones: variables y funciones en camelCase (`precioTotal`, `conIVA`) o snake_case (`precio_total`); constantes en MAYUSCULAS_CON_GUIONES (`IVA`, `MAXIMO`); clases en PascalCase (lo verás en la unidad 6). Elige una convención y respétala en todo tu proyecto: el código inconsistente grita "principiante" más que cualquier error de sintaxis.

La indentación es sagrada: cada nivel de llaves se indenta un nivel (4 espacios es lo estándar). Un código sin indentar es como un libro sin párrafos: técnicamente legible, prácticamente imposible. Configura tu editor para que use espacios, no tabuladores, y deja que el formateo automático trabaje por ti.

Instalación paso a paso en Windows: MinGW-w64 y VS Code

En la primera parte viste la instalación de forma resumida. Aquí va el procedimiento detallado para Windows, que es donde más se atorán los principiantes, con cada pantalla descrita para que no te pierdas. Si ya tienes tu compilador funcionando, úsalo como lista de verificación.

Primero lo primero: ¿qué es un IDE y qué es un editor?

Un editor de texto para programar (como VS Code, Notepad++ o Vim) es un bloc de notas con superpoderes: colorea la sintaxis, autocompleta, y te deja correr comandos, pero no trae compilador incluido: tú instalas el compilador por separado y el editor lo invoca. Un IDE (Entorno de Desarrollo Integrado, como Visual Studio Community, CLion o Code::Blocks) es un paquete todo en uno: editor, compilador, depurador y constructor de proyectos integrados en una sola aplicación.

La analogía: el editor es comprar el taladro, la sierra y el martillo por separado y armar tu propia caja de herramientas; el IDE es comprar el taller completo ya equipado. El taller equipado es cómodo, pero tu caja personal es más ligera, más flexible y te enseña qué hace cada herramienta. En este libro usamos la combinación editor + compilador separado (VS

Code + g++ vía MinGW-w64): es lo que usan la mayoría de los profesionales de C++ hoy, te enseña el proceso real de compilación y no te esconde nada tras botones mágicos. Cuando domines esto, cualquier IDE te parecerá trivial.

Paso 1: instalar MinGW-w64 (tu compilador g++)

MinGW-w64 es la versión de las herramientas GNU (incluido g++) para Windows. Sigue estos pasos:

1. Abre tu navegador y busca "MinGW-w64". Entra al sitio oficial (el proyecto vive en mingw-w64.org) y ve a la sección de descargas. Verás varias formas de instalarlo; la más sencilla para empezar es el instalador o un paquete precompilado.
2. Descarga la versión para tu sistema: casi todas las computadoras actuales son de 64 bits, así que elige la variante x86_64. En las opciones de "threads" elige posix y en "exception" elige seh (son las opciones estándar para 64 bits).
3. Descomprime o instala en una ruta simple y sin espacios, por ejemplo C:\mingw64. Las rutas con espacios ("Archivos de programa") causan problemas misteriosos con algunas herramientas: evita.
4. Ahora viene el paso donde más gente se atora: agregar el compilador al PATH. El PATH es la lista de carpetas donde Windows busca programas cuando escribes un comando. 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", selecciona la variable Path de tu usuario y dale a "Editar". Agrega una nueva entrada con C:\mingw64\bin (la carpeta bin dentro de donde instalaste). Acepta todo.
5. Cierra todas las terminales abiertas (las viejas no se enteran del cambio) y abre una nueva: puede ser cmd o PowerShell. Escribe g++ --version y presiona Enter. Si ves algo como "g++ (MinGW-W64 x86_64-posix-seh) 13.2.0", quedó perfecto. Si dice que no reconoce el comando, el PATH quedó mal: repite el paso 4 con calma, revisando letra por letra la ruta.

Paso 2: instalar y configurar VS Code

1. Descarga VS Code desde su sitio oficial (code.visualstudio.com) e instálalo con las opciones por defecto. Durante la instalación marca las casillas "Agregar al PATH" y "Abrir con Code" en el menú contextual: te van a ahorrar clics todos los días.

2. Abre VS Code y ve al panel de extensiones (el icono de cuadritos en la barra lateral, o Ctrl+Shift+X). Busca "C/C++" e instala la extensión de Microsoft: es la que da coloreado, autocompletado y detección de errores mientras escribes.
3. Opcional pero recomendado: instala también la extensión "Code Runner" si quieres ejecutar con un botón, aunque en este libro aprenderás a compilar desde la terminal integrada, que es la habilidad que de verdad importa.
4. Crea una carpeta para tus prácticas, por ejemplo C:\cpp o Documentos\cpp. En VS Code usa Archivo > Abrir carpeta y elige esa carpeta. Todo tu trabajo vivirá ahí, ordenado.
5. Abre la terminal integrada con Ctrl+ñ (o Ctrl+` en teclados en inglés; también está en el menú Ver > Terminal). Escribe g++ --version ahí también: debe funcionar igual que en la terminal de Windows. Si no funciona, cierra VS Code por completo y vuelve a abrirlo para que lea el PATH nuevo.

Paso 3: tu primer ciclo completo de trabajo

1. Crea un archivo nuevo (Ctrl+N), escribe tu programa hola.cpp y guárdalo (Ctrl+S) dentro de tu carpeta de prácticas. VS Code coloreará el código automáticamente gracias a la extensión.
2. En la terminal integrada escribe: g++ -Wall -Wextra -std=c++17 hola.cpp -o hola.exe y presiona Enter. Si no hay errores, no verás ningún mensaje (en el mundo Unix, el silencio es éxito).
3. Ejecuta con .hola.exe (en PowerShell) o hola.exe (en cmd). Verás tu saludo. Ese ciclo - editar, compilar, ejecutar - lo repetirás miles de veces: conviértelo en reflejo.
4. Truco profesional: la flecha hacia arriba en la terminal repite el último comando. Editas, flecha arriba, Enter, y recompilas en un segundo.

Si algo falla en estos pasos, el 90% de las veces es el PATH (paso 1.4) o haber instalado la versión de 32 bits en un sistema de 64 bits. Respira, relee el paso con calma y verifica cada letra de la ruta. Todos los programadores hemos peleado con el PATH alguna vez: no eres tú, es un rito de iniciación.

Enlazado estático vs dinámico: cómo tu programa se junta con las bibliotecas

En la primera parte aprendiste que compilar tiene cuatro fases y que la última es el enlazado (linking): unir tu código objeto con las bibliotecas para producir el ejecutable. Ahora vamos a entender qué son esas bibliotecas y las dos formas de enlazarlas, porque esta distinción explica desde por qué un .exe pesa lo que pesa hasta por qué a veces un programa "no corre en otra computadora".

Qué es una biblioteca

Una biblioteca (library) es código ya compilado (o listo para incluir) que alguien escribió para que tú lo reutilices: funciones matemáticas, manejo de imágenes, conexiones de red, interfaces gráficas. En vez de reinventar la rueda, enlazas la rueda ya inventada. La biblioteca estándar de C++ (la STL, iostream, vector, etc.) es el ejemplo que ya usas sin darte cuenta: cuando escribes `#include <iostream>`, usas su interfaz, y al enlazar, el linker junta tu programa con su implementación.

Las bibliotecas tienen dos caras: las cabeceras (.h), que declaran qué funciones existen (las promesas), y el código compilado (.lib en Windows, .a o .so en Linux), que contiene la implementación real (los cuerpos). Tú incluyes las promesas; el linker conecta los cuerpos.

Enlace estático: todo empacado en tu ejecutable

En el enlace estático, el linker copia dentro de tu ejecutable todo el código de biblioteca que tu programa usa. El resultado es un archivo más grande, pero independiente: lo puedes copiar a otra computadora y corre sin pedir nada más (mientras el sistema operativo sea compatible).

Ventajas: el programa es portable como archivo único, no depende de que la otra máquina tenga las bibliotecas instaladas y siempre usa exactamente la versión con la que lo compilaste (cero sorpresas). Desventajas: el ejecutable pesa más, y si la biblioteca tiene un bug de seguridad, tienes que recompilar y redistribuir tu programa para arreglarlo: cada programa lleva su propia copia del código.

En Linux, las bibliotecas estáticas terminan en .a (archive). Puedes crear la tuya a mano para entender el proceso. Ejemplo completo: crea estos tres archivos.

Archivo mate.h (las promesas):

```

#ifndef MATE_H
#define MATE_H
int sumar(int a, int b);
int multiplicar(int a, int b);
#endif

```

Archivo mate.cpp (los cuerpos):

```

#include "mate.h"

int sumar(int a, int b) {
    return a + b;
}

int multiplicar(int a, int b) {
    return a * b;
}

```

Archivo main.cpp (quien usa la biblioteca):

```

#include <iostream>
#include "mate.h"

int main() {
    std::cout << "3 + 4 = " << sumar(3, 4) << std::endl;
    std::cout << "3 * 4 = " << multiplicar(3, 4) << std::endl;
    return 0;
}

```

Ahora construye la biblioteca estática paso a paso en la terminal. Primero compila mate.cpp a código objeto sin enlazar (la bandera -c significa "solo compila, no enlaces"): `g++ -Wall -Wextra -std=c++17 -c mate.cpp -o mate.o`. Luego empaqueta el objeto en una biblioteca estática con la herramienta ar (archiver): `ar rcs libmate.a mate.o`. Las letras rcs significan reemplazar, crear e indexar: es el hechizo estándar para armar la biblioteca. Finalmente compila el programa enlazando la biblioteca estática: `g++ -Wall -Wextra -std=c++17 main.cpp -L. -lmate -o programa_estatico`. La bandera -L. dice "busca bibliotecas en la carpeta actual" y -lmate dice "enlaza libmate.a" (el linker agrega solo el prefijo lib y el sufijo .a).

Verifica el resultado: en Linux, el comando `ldd programa_estatico` muestra las bibliotecas dinámicas que necesita; verás que libmate no aparece ahí, porque su código ya está copiado dentro del ejecutable. Puedes borrar libmate.a y mate.o y el programa sigue corriendo: la

prueba definitiva de que el enlace fue estático. En Windows con MinGW-w64 el proceso es idéntico, solo cambian las extensiones (.a se usa igual).

Enlace dinámico: compartiendo código en tiempo de ejecución

En el enlace dinámico, tu ejecutable no copia el código de la biblioteca: solo anota "necesito estas funciones de tal biblioteca" y el sistema operativo las conecta cuando el programa arranca. Las bibliotecas dinámicas terminan en .dll en Windows, .so en Linux y .dylib en macOS.

Ventajas: los ejecutables son más pequeños, muchos programas comparten una sola copia de la biblioteca en memoria (ahorro real de RAM) y actualizar la biblioteca (por ejemplo, un parche de seguridad) arregla todos los programas a la vez sin recompilarlos. Desventajas: si la biblioteca no está en la otra computadora, tu programa no arranca (el famoso "falta el archivo .dll" de Windows), y una actualización de la biblioteca puede romper programas que esperaban la versión vieja (el legendario "infierno de las DLL").

La biblioteca estándar de C++ (libstdc++) normalmente se enlaza de forma dinámica: por eso un "hola mundo" en C++ pesa apenas unos kilobytes, aunque iostream sea enorme. g++ decide el tipo de enlace por defecto; puedes forzarlo: -static le pide enlace estático de todo (el ejecutable crece muchísimo, pruébalo una vez por curiosidad: g++ -static hola.cpp -o hola_estatico y compara los tamaños con ls -lh).

Regla práctica: para programas que vas a repartir a usuarios finales, el enlace estático (o incluir las DLL junto al .exe) evita dolores de cabeza. Para el sistema operativo y sus componentes, el enlace dinámico ahorra memoria y facilita actualizaciones. Como programador, entender ambas te permite diagnosticar esos errores de "biblioteca no encontrada" que asustan a los principiantes.

Tu programa y el sistema operativo: argumentos y variables de entorno

Hasta ahora tus programas vivían en una burbuja: pedían datos con cin y mostraban con cout. Pero los programas reales conversan con el sistema operativo de dos formas fundamentales: recibiendo argumentos en la línea de comandos y leyendo variables de entorno. Es hora de abrir la burbuja.

Argumentos de línea de comandos: argc y argv

Cuando escribes en la terminal `programa_estatico`, en realidad le estás pasando al programa una lista de palabras: el nombre del programa más lo que escribas después. C++ te entrega esa lista a través de una versión especial de `main` con dos parámetros:

```
int main(int argc, char* argv[]) { ... }
```

`argc` (argument count) es cuántas palabras recibiste, contando el nombre del programa. `argv` (argument vector) es el arreglo con esas palabras, cada una como texto (`char*`). `argv[0]` siempre es el nombre con el que se invocó al programa; `argv[1]` es el primer argumento real, y así hasta `argv[argc - 1]`.

Ejemplo completo y compilable: un programa que saluda con repeticiones configurables. Guárdalo como `saludador.cpp`:

```

#include <iostream>
#include <string>

// Convierte texto a entero de forma simple; supone entrada valida
int aEntero(const std::string& texto) {
    int valor{0};
    for (char c : texto) {
        valor = valor * 10 + (c - '0');
    }
    return valor;
}

int main(int argc, char* argv[]) {
    // argv[0] es el nombre del programa; los argumentos reales empiezan en 1
    if (argc < 2) {
        std::cout << "Uso: " << argv[0] << " <nombre> [repeticiones]" << std::endl;
        std::cout << "Ejemplo: " << argv[0] << " Ana 3" << std::endl;
        return 1; // codigo distinto de cero = algo salio mal
    }

    std::string nombre{argv[1]};
    int repeticiones{1};
    if (argc >= 3) {
        repeticiones = aEntero(argv[2]);
    }

    for (int i{0}; i < repeticiones; ++i) {
        std::cout << "Hola, " << nombre << "!" << std::endl;
    }
    return 0;
}

```

Compíllalo: `g++ -Wall -Wextra -std=c++17 saludador.cpp -o saludador`. Ahora juega en la terminal:

- `./saludador` -> muestra el uso correcto (argc es 1, solo el nombre del programa).
- `./saludador Ana` -> saluda una vez (argc es 2).
- `./saludador Ana 3` -> saluda tres veces (argc es 3).

Vamos línea por línea por lo nuevo. La función `aEntero` convierte un texto como "3" a entero recorriendo sus caracteres: `c - '0'` es el truco clásico (en ASCII los dígitos están consecutivos, así que restar el carácter '0' da el valor numérico; lo verás a fondo en la expansión de la unidad 2). En `main`, primero verificamos `argc < 2`: si el usuario no dio el nombre, mostramos cómo se

usa el programa usando `argv[0]` para que el mensaje diga el nombre real del ejecutable, y devolvemos 1: por convención, return distinto de cero significa "terminó con error", y los scripts del sistema lo usan para detectar fallas. `std::string nombre{argv[1]}` convierte el texto crudo a un string cómodo. Si hay tercer argumento, lo convertimos a número de repeticiones. El `for` imprime el saludo las veces pedidas.

Tres lecciones profesionales en este ejemplo corto: todo programa con argumentos debe validar que recibió los que necesita y explicar su uso si no (ese mensaje de "Uso:" es marca de software serio); el código de salida (`return`) es un canal de comunicación con el sistema: 0 es éxito, otro valor es error; y los argumentos siempre llegan como texto: convertirlos a números es tu responsabilidad (en programas grandes se usan bibliotecas de parseo, pero el principio es este).

Ojo con un detalle: si el usuario escribe `./saludador "Ana Maria" 2`, el nombre llega como un solo argumento porque las comillas agrupan palabras. Y si escribe `./saludador Ana tres`, nuestra conversión ingenua dará un resultado raro: validar entradas hostiles es tema de unidades avanzadas, pero ya sabes que el problema existe.

Variables de entorno: leyendo el ambiente del sistema

Las variables de entorno son pares nombre=valor que el sistema operativo mantiene y que todos los programas pueden leer: cosas como el nombre del usuario (`USER` o `USERNAME`), la carpeta personal (`HOME`), o el idioma (`LANG`). Los programas serios las usan para configurarse sin necesidad de argumentos: dónde guardar archivos, en qué modo correr, qué idioma hablar.

En C++ se leen con la función `getenv` de la cabecera `<cstdlib>`: `const char* valor = std::getenv("HOME");`. Devuelve el valor como texto, o `nullptr` si la variable no existe (siempre verifica antes de usarlo). Ejemplo completo:

```

#include <iostream>
#include <cstdlib> // para getenv

void mostrarVariable(const char* nombre) {
    const char* valor = std::getenv(nombre);
    if (valor != nullptr) {
        std::cout << nombre << " = " << valor << std::endl;
    } else {
        std::cout << nombre << " no esta definida." << std::endl;
    }
}

int main() {
    std::cout << "--- Variables de entorno ---" << std::endl;
    mostrarVariable("HOME");      // Linux/macOS: carpeta del usuario
    mostrarVariable("USER");      // Linux/macOS: nombre del usuario
    mostrarVariable("USERNAME");  // Windows: nombre del usuario
    mostrarVariable("PATH");      // donde el sistema busca programas
    mostrarVariable("LANG");      // idioma preferido

    // Ejemplo practico: saludar según el usuario del sistema
    const char* usuario = std::getenv("USER");
    if (usuario == nullptr) {
        usuario = std::getenv("USERNAME"); // intento en Windows
    }
    if (usuario != nullptr) {
        std::cout << "Hola de nuevo, " << usuario << "." << std::endl;
    }
    return 0;
}

```

Compíllalo y ejecútalo: verás los valores reales de tu sistema. Prueba también a definir tu propia variable antes de ejecutar: en Linux/macOS, `MI_NOMBRE="Esteban" ./variables`; en Windows (cmd), `set MI_NOMBRE=Esteban` y luego ejecuta. Agrega `mostrarVariable("MI_NOMBRE");` al programa y verás cómo tu programa lee lo que le inyectaste desde afuera. Esa es la base de la configuración de servidores y contenedores en la industria: el mismo programa se comporta distinto según las variables que le pongas, sin recompilar.

Combinación poderosa: un programa profesional lee primero sus argumentos (lo específico de esta ejecución), luego las variables de entorno (la configuración del ambiente) y usa valores por defecto si no hay nada. Ese orden de prioridad (argumentos > entorno > defecto) es el estándar de facto en herramientas de línea de comandos.

Catálogo de errores comunes de esta parte (y cómo evitarlos)

1. Compilar código moderno sin la bandera -std: usas enlaces estructurados o string_view, compilas sin -std=c++17 y obtienes errores crípticos. Solución: la bandera siempre, en cada compilación. Hazla parte de tu comando base.
2. Mezclar estándares entre archivos: compilas mate.cpp con C++17 y main.cpp con C++11. A veces enlaza, a veces falla con errores raros de símbolos. Solución: usa la misma bandera para todos los archivos del proyecto.
3. Olvidar que argv[0] es el programa: lees argv[0] esperando el primer argumento real y tu programa se comporta raro. Solución: los argumentos reales empiezan en argv[1]; argc incluye el nombre del programa.
4. No validar argc antes de usar argv[i]: si el usuario no pasa argumentos y accedes argv[1], lees memoria inválida y el programa truena. Solución: verifica argc primero, siempre, y muestra el mensaje de uso.
5. Asumir que getenv siempre devuelve algo: si la variable no existe, devuelve nullptr, y usarlo truena el programa. Solución: compara con nullptr antes de usar el valor.
6. El PATH mal configurado en Windows: g++ no se reconoce después de instalar MinGW-w64. Solución: revisa que agregaste la carpeta bin (no la raíz) al Path del usuario, y cierra y reabre todas las terminales.
7. Instalar MinGW-w64 en ruta con espacios: compila a veces sí y a veces no, con errores extraños. Solución: reinstala en C:\mingw64 o similar, sin espacios ni acentos.
8. Enlazar con -lmate pero olvidar -L.: el linker dice "cannot find -lmate". Solución: -L. le dice dónde buscar; sin eso solo busca en las carpetas del sistema.
9. Confundir enlace estático con dinámico al repartir el programa: compilas con enlace dinámico, le pasas el .exe a un amigo y le falta una DLL. Solución: para repartir, enlaza estático (-static) o incluye las DLL junto al ejecutable.
10. Devolver 0 aunque algo falló: tu programa detecta argumentos inválidos, imprime el error, pero devuelve 0 y los scripts creen que todo salió bien. Solución: devuelve un código distinto de cero en cada camino de error.

Práctica: ejercicios con solución

Ejercicio 1

Escribe un programa que reciba dos números como argumentos de línea de comandos y muestre su suma, resta, multiplicación y división. Si no recibe exactamente dos números, debe mostrar el uso correcto y terminar con código de error. Reutiliza la idea de conversión de texto a número del ejemplo saludador.

Solución:

```

#include <iostream>
#include <string>

double aDouble(const std::string& texto) {
    double valor{0.0};
    double divisor{1.0};
    bool decimales{false};
    bool negativo{false};
    std::size_t i{0};
    if (!texto.empty() && texto[0] == '-') {
        negativo = true;
        i = 1;
    }
    for (; i < texto.size(); ++i) {
        char c = texto[i];
        if (c == '.') {
            decimales = true;
        } else {
            valor = valor * 10.0 + (c - '0');
            if (decimales) divisor *= 10.0;
        }
    }
    valor /= divisor;
    return negativo ? -valor : valor;
}

int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cout << "Uso: " << argv[0] << " <num1> <num2>" << std::endl;
        return 1;
    }
    double a{aDouble(argv[1])};
    double b{aDouble(argv[2])};
    std::cout << "Suma: " << a + b << std::endl;
    std::cout << "Resta: " << a - b << std::endl;
    std::cout << "Multiplicacion: " << a * b << std::endl;
    if (b != 0.0) {
        std::cout << "Division: " << a / b << std::endl;
    } else {
        std::cout << "Division: no se puede dividir entre cero." << std::endl;
    }
    return 0;
}

```

Explicación: la función `aDouble` es una versión extendida de la del ejemplo: maneja signo negativo y punto decimal recorriendo carácter por carácter. En `main`, la validación es estricta: `argc != 3` significa que no llegaron exactamente dos números, así que mostramos el uso con `argv[0]` y devolvemos 1. La división verifica que el divisor no sea cero antes de operar. Pruébalo con `./calculadora 10 4` y con `./calculadora 10 0` para ver ambos caminos. Nota que aquí sí usamos el operador ternario (`negativo ? -valor : valor`), un adelanto de la expansión de la unidad 2.

Ejercicio 2

Escribe un programa que lea la variable de entorno `MI_IDIOMA` y salude en español ("Hola"), inglés ("Hello") o francés ("Bonjour") según su valor ("es", "en", "fr"). Si la variable no existe o tiene otro valor, saluda en español por defecto. Además debe aceptar un argumento opcional con el nombre de la persona.

Solución:


```

#include <iostream>
#include <cstdlib>
#include <string>

int main(int argc, char* argv[]) {
    std::string nombre{"mundo"};
    if (argc >= 2) {
        nombre = argv[1];
    }

    std::string idioma{"es"};
    const char* env = std::getenv("MI_IDIOMA");
    if (env != nullptr) {
        idioma = env;
    }

    std::string saludo{"Hola"};
    if (idioma == "en") {
        saludo = "Hello";
    } else if (idioma == "fr") {
        saludo = "Bonjour";
    }

    std::cout << saludo << ", " << nombre << "!" << std::endl;
    return 0;
}

```

Explicación: aquí se ve el orden de prioridad profesional: el argumento (específico de esta ejecución) define el nombre; la variable de entorno (configuración del ambiente) define el idioma; y hay valores por defecto ("mundo", "es") si no llega nada. Observa que `getenv` se verifica contra `nullptr` antes de usarlo, y que un idioma desconocido cae al español por defecto en vez de tronar. Pruébalo así en Linux/macOS: `MI_IDIOMA=en ./saludos Ana`. En Windows cmd: `set MI_IDIOMA=fr` y luego `saludos.exe Ana`.

Ejercicio 3

Crea una mini biblioteca estática de geometría con las funciones `double areaCirculo(double r)`, `double areaRectangulo(double b, double h)` y `double perimetroCirculo(double r)` en los archivos `geometria.h` y `geometria.cpp`. Luego escribe `main.cpp` que reciba por argumentos la figura ("circulo" o "rectangulo") y sus medidas, y muestre el área (y el perímetro si es círculo). Compila la biblioteca con `ar` y enlaza estáticamente.

Solución: geometria.h:

```
#ifndef GEOMETRIA_H
#define GEOMETRIA_H
double areaCirculo(double r);
double perimetroCirculo(double r);
double areaRectangulo(double b, double h);
#endif
```

geometria.cpp:

```
#include "geometria.h"

const double PI{3.14159265358979};

double areaCirculo(double r) {
    return PI * r * r;
}

double perimetroCirculo(double r) {
    return 2.0 * PI * r;
}

double areaRectangulo(double b, double h) {
    return b * h;
}
```

main.cpp:

```

#include <iostream>
#include <string>
#include "geometria.h"

double aDouble(const std::string& t) {
    double v{0.0}, d{1.0};
    bool dec{false}, neg{false};
    std::size_t i{0};
    if (!t.empty() && t[0] == '-') { neg = true; i = 1; }
    for (; i < t.size(); ++i) {
        if (t[i] == '.') dec = true;
        else { v = v * 10.0 + (t[i] - '0'); if (dec) d *= 10.0; }
    }
    v /= d;
    return neg ? -v : v;
}

int main(int argc, char* argv[]) {
    if (argc < 3) {
        std::cout << "Uso: " << argv[0] << " circulo <radio>" << std::endl;
        std::cout << "      o: " << argv[0] << " rectangulo <base> <altura>" <<
std::endl;
        return 1;
    }
    std::string figura{argv[1]};
    if (figura == "circulo") {
        double r{aDouble(argv[2])};
        std::cout << "Area: " << areaCirculo(r) << std::endl;
        std::cout << "Perimetro: " << perimetroCirculo(r) << std::endl;
    } else if (figura == "rectangulo") {
        if (argc < 4) {
            std::cout << "Falta la altura." << std::endl;
            return 1;
        }
        double b{aDouble(argv[2])};
        double h{aDouble(argv[3])};
        std::cout << "Area: " << areaRectangulo(b, h) << std::endl;
    } else {
        std::cout << "Figura desconocida: " << figura << std::endl;
        return 1;
    }
    return 0;
}

```

Compilación:

```
g++ -Wall -Wextra -std=c++17 -c geometria.cpp -o geometria.o
ar rcs libgeometria.a geometria.o
g++ -Wall -Wextra -std=c++17 main.cpp -L. -lgeometria -o geo
```

Explicación: este ejercicio junta tres temas de la unidad: biblioteca estática propia (promesas en .h, cuerpos en .cpp, empaquetado con ar), argumentos de línea de comandos con validación por figura, y conversión de texto a número. Observa las guardas de inclusión en el .h y la constante PI local al .cpp (no necesita ser global pública). Prueba ./geo circulo 5 y ./geo rectangulo 4 6. Si borras libgeometria.a después de compilar, geo sigue funcionando: el código ya quedó copiado dentro del ejecutable.

Ejercicio 4

Escribe un programa que imprima el valor de `__cplusplus` y, según el resultado, diga con palabras qué estándar se está usando (C++11, C++14, C++17, C++20 u otro). Compíllalo con distintas banderas `-std=` y verifica que el mensaje cambie. Incluye además una línea que use una característica de C++17 (por ejemplo, un `if` con inicializador) protegida de forma que el programa siga compilando en estándares viejos... o explica por qué eso no es posible y qué se hace en su lugar.

Solución:

```

#include <iostream>

int main() {
    std::cout << "__cplusplus = " << __cplusplus << std::endl;

    std::string nombre{"desconocido"};
    if (__cplusplus >= 202002L) {
        nombre = "C++20 o posterior";
    } else if (__cplusplus >= 201703L) {
        nombre = "C++17";
    } else if (__cplusplus >= 201402L) {
        nombre = "C++14";
    } else if (__cplusplus >= 201103L) {
        nombre = "C++11";
    }
    std::cout << "Estandar detectado: " << nombre << std::endl;

    // Característica de C++17: if con inicializador
    if (int x{42}; x > 0) {
        std::cout << "El if con inicializador compilo: x = " << x << std::endl;
    }
    return 0;
}

```

Explicación: `__cplusplus` es una macro que el compilador define con el año y mes del estándar en formato AAAAMM más la letra L (de long). La cadena de if-else la traduce a palabras. Compila con `-std=c++11`, `-std=c++14`, `-std=c++17` y `-std=c++20` y observa cómo cambia el mensaje... pero ojo: con `-std=c++11` la línea del if con inicializador FALLA, porque esa sintaxis no existía en 2011. Y aquí va la respuesta a la segunda parte: no se puede "proteger" sintaxis nueva con un if normal, porque el compilador debe entender la sintaxis antes de evaluar cualquier condición. Lo que se hace en el mundo real es la compilación condicional del preprocesador con `#if __cplusplus >= 201703L`, que elimina el código antes de que el compilador lo vea. El preprocesador sí puede decidir qué texto llega al compilador; un if en tiempo de ejecución, no.

Resumen de la expansión

- C++ evoluciona cada tres años sin romper el pasado: C++11 fue la gran revolución moderna (auto, for de rango, nullptr, lambdas, punteros inteligentes), C++14 pulió detalles (binarios, separadores de dígitos), C++17 es la base de este libro (enlaces estructurados, if con inicializador, string_view, filesystem), C++20 trajo conceptos, rangos, módulos y corrutinas, y C++23 es lo más nuevo.
- La bandera -std=c++17 elige las reglas del juego; __cplusplus te dice cuál está activa. Sé explícito siempre.
- C++ gana en velocidad y control; Python en velocidad de desarrollo y datos; Java en empresas portables; JavaScript en la web. Elige según el problema, no por fanatismo.
- Un programa se compone de directivas del preprocesador, comentarios (explica el porqué), funciones (declaración vs definición), la puerta de entrada main y variables globales (constantes sí, modificables casi nunca). El estilo consistente es marca profesional.
- En Windows: MinGW-w64 te da g++ (instálalo en ruta sin espacios y agrega su bin al PATH) y VS Code es tu editor (con la extensión C/C++). El ciclo es editar, compilar con g++ -Wall -Wextra -std=c++17, ejecutar.
- Un IDE trae todo integrado; un editor + compilador separado te enseña el proceso real. Aquí usamos lo segundo.
- Las bibliotecas son código reutilizable: cabeceras con promesas y binarios con cuerpos. El enlace estático copia el código dentro de tu ejecutable (portable, más pesado); el dinámico lo conecta al arrancar (.dll/.so, más ligero, depende del ambiente). Puedes construir tus propias bibliotecas estáticas con ar.
- main puede recibir int argc y char* argv[]: la lista de palabras de la línea de comandos. Valida argc, muestra el uso con argv[0] y devuelve códigos de error distintos de cero.
- getenv lee variables de entorno: la configuración del ambiente sin recompilar. Verifica siempre contra nullptr.
- En la siguiente unidad profundizarás en los tipos y operadores que ya conoces, hasta dominar bits, precedencias y literales como un profesional.

UNIDAD 2: Los fundamentos del lenguaje

En la unidad 1 encendiste el motor. Ahora vas a aprender a manejarlo en serio. Esta unidad cubre los fundamentos que todo programa en C++ usa: las variables donde guardas datos, los tipos que definen qué clase de datos son, los operadores con los que los transformas y el sistema de entrada y salida con el que tu programa conversa con el usuario.

Estos temas suenan básicos, y lo son, pero aquí está el secreto: la mayoría de los errores graves en C++ nacen de no entender bien los fundamentos. Un desbordamiento de enteros, una división entre enteros que "pierde" decimales, una variable sin inicializar que guarda basura: todos son errores de fundamentos. Domínalos ahora y te ahorrarás cientos de horas después.

Al terminar esta unidad sabrás: declarar e inicializar variables de todos los tipos fundamentales, las diferencias entre int, float, double, char y bool, cómo funcionan los operadores aritméticos, relacionales y lógicos (con sus trampas clásicas), y cómo usar cin y cout como un profesional, incluyendo formato de salida.

Explicación detallada

Clase 4: Variables y tipos de datos

Una variable es un espacio de memoria con nombre donde guardas un dato. Piensa en ella como una caja etiquetada: la etiqueta es el nombre, el contenido es el valor, y el tipo define qué clase de cosas puede contener la caja. En C++ el tipo se declara al crear la variable y no cambia nunca: es un lenguaje de tipado estático, y el compilador vigila que no metas un texto donde va un número.

La declaración básica es: tipo nombre = valor;. Por ejemplo: int edad = 25;. Aquí int significa "entero", edad es el nombre que elegimos y 25 el valor inicial. Los nombres de variables pueden llevar letras, números y guion bajo, pero no pueden empezar con número ni ser palabras reservadas como int o return.

Los tipos fundamentales de C++ son pocos pero importantes. Los enteros: int es el entero estándar (normalmente 32 bits, rango aproximado de -2,000 millones a 2,000 millones). Para números más grandes están long y long long. Para números pequeños o cuando quieres ahorrar memoria están short y char (que también guarda caracteres). Existen versiones sin signo: unsigned int solo guarda valores positivos pero llega al doble de rango.

Los números con decimales: float (precisión simple, unos 7 dígitos decimales) y double (precisión doble, unos 15 dígitos). Regla de oro: usa double por defecto. float solo cuando la memoria sea crítica o una biblioteca lo exija. Los cálculos con float acumulan errores de redondeo más rápido, y en 2026 la memoria no es tan escasa como para arriesgarse.

El tipo char guarda un solo carácter: char letra = 'A';. Ojo con las comillas: los caracteres usan comillas simples 'A', los textos usan comillas dobles "Hola". Confundirlas es un error clásico que el compilador detecta, por suerte.

El tipo bool guarda verdadero o falso: bool aprobado = true;. Internamente es 1 o 0, y lo usarás muchísimo en condiciones.

También existe auto, que le pide al compilador que deduzca el tipo: auto x = 5; hace a x un int automáticamente. Es cómodo y moderno, pero úsalo con moderación mientras aprendes: declarar el tipo explícitamente te obliga a pensar en qué estás guardando.

El tamaño real de cada tipo depende del sistema, pero puedes preguntarlo con el operador sizeof. Por ejemplo, sizeof(int) te dice cuántos bytes ocupa un int en tu máquina. En la práctica: char 1 byte, int 4 bytes, float 4 bytes, double 8 bytes, long long 8 bytes.

Inicializar significa dar valor inicial a la variable al declararla. En C++ moderno hay tres formas: int a = 5; (copia), int b(5); (directa) e int c{5}; (uniforme, con llaves). La forma con llaves es la recomendada en C++ moderno porque no permite "estrechamientos" peligrosos: int d{3.14}; da error en vez de truncar silenciosamente a 3. Acostúmbrate a las llaves.

La regla más importante de esta clase: inicializa SIEMPRE tus variables. Una variable declarada sin valor, como int x;, contiene basura: lo que haya quedado en esa memoria. Usarla es un error grave y difícil de detectar porque a veces "funciona" por casualidad. Con -Wall el compilador te avisa si usas variables sin inicializar. Hazle caso.

Las constantes son variables que no cambian: const double PI = 3.14159;. El compilador te impide modificarlas por accidente. Usa const para todo valor que no deba cambiar: hace tu código más seguro y más legible. Los profesionales marcan const todo lo posible.

Clase 5: Operadores

Los operadores son los símbolos que transforman datos. Los aritméticos son los conocidos: + suma, - resta, * multiplicación, / división, % módulo (el residuo de la división). Parecen inocentes, pero esconden las trampas más famosas de C++.

Trampa número uno: la división entre enteros. En C++, 7 / 2 da 3, no 3.5. Cuando ambos operandos son enteros, el resultado es entero y los decimales se truncan (no se redondean: se

cortan). Para obtener 3.5 necesitas que al menos un operando sea decimal: `7.0 / 2` da 3.5. Este error ha roto más programas de principiantes que ningún otro. Si tu cálculo "pierde" decimales misteriosamente, revisa los tipos.

Trampa número dos: el módulo `%` solo funciona con enteros. `7 % 3` da 1 (porque $7 = 2 \cdot 3 + 1$). Es utilísimo para saber si un número es par (`n % 2 == 0`), para ciclos repetitivos y para limitar valores.

Los operadores de asignación combinada ahorran escritura: `x += 5` equivale a `x = x + 5`. Existen `+=`, `-=`, `*=`, `/=` y `%=`. Úsalos: son más claros y a veces más eficientes.

Los operadores de incremento y decremento, `++` y `--`, son legendarios en C++ (el nombre del lenguaje viene de aquí). `x++` suma uno a `x`. Pero hay dos versiones: pre-incremento (`++x`) y post-incremento (`x++`). La diferencia importa cuando los usas dentro de una expresión: `++x` incrementa primero y luego usa el valor; `x++` usa el valor primero y luego incrementa. En instrucciones solas no hay diferencia, y ahí es donde deberías usarlos casi siempre para evitar confusiones.

Los operadores relacionales comparan y devuelven `bool`: `==` igual, `!=` diferente, `<` menor, `>` mayor, `<=` menor o igual, `>=` mayor o igual. El error clásico es escribir `=` (asignación) en vez de `==` (comparación) dentro de un `if`. El compilador con `-Wall` te avisa, pero mejor acostúmbrate a leer tus condiciones dos veces.

Los operadores lógicos combinan condiciones: `&&` significa Y (ambas deben ser verdaderas), `||` significa O (al menos una), `!` significa NO (invierte). Tienen "cortocircuito": en `a && b`, si `a` es falso, `b` ni se evalúa. Esto es útil y también una trampa si `b` tenía un efecto secundario que esperabas.

La precedencia define el orden: primero `*` / `%`, luego `+` / `-`, luego relacionales, luego `&&`, luego `||`. Cuando dudes, usa paréntesis: `(a + b) * c` es clarísimo y nunca falla. Los paréntesis de más no cuestan nada.

Clase 6: Entrada y salida con `iostream`

Ya usaste `cout` y `cin` en la unidad 1. Ahora vamos a fondo. `iostream` es la biblioteca de flujos de entrada y salida de C++. Un "flujo" es una secuencia de datos que fluye en una dirección: `cout` es el flujo hacia la pantalla, `cin` es el flujo desde el teclado, `cerr` es el flujo de errores.

`cout` con `<<` envía datos a la pantalla. Puedes encadenar: `cout << "Edad: " << edad << " años" << endl;`. `endl` inserta un salto de línea y vacía el búfer (fuerza que se muestre ya). Alternativa: `"\n"`, que solo baja de línea sin vaciar el búfer, y es un poco más rápida. En programas normales la diferencia no importa; usa `endl` mientras aprendes.

cin con >> lee datos del teclado. Lee "palabras" separadas por espacios: si el usuario escribe "Juan Perez", cin >> nombre solo toma "Juan". Para leer líneas completas con espacios usa getline(cin, nombre), que veremos en la unidad 5. Puedes leer varios datos seguidos: cin >> a >> b; lee dos valores.

Un detalle crucial: si pides un número y el usuario escribe texto, cin entra en estado de error y las lecturas siguientes fallan en cascada. Para programas serios hay que validar, pero por ahora basta saber que existe el problema. En la unidad 9 aprenderás a manejarlo con excepciones.

El formato de salida se controla con manipuladores de la biblioteca <iomanip>. Los más útiles: std::fixed y std::setprecision(n) para decimales. Por ejemplo, para mostrar dinero con dos decimales:

```
#include <iostream>
#include <iomanip>

int main() {
    double precio = 19.99;
    double impuesto = precio * 0.16;
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Total: $" << precio + impuesto << std::endl;
    return 0;
}
```

std::setw(n) define el ancho mínimo del siguiente dato, útil para tablas. std::left y std::right alinean. Con estas herramientas tus salidas se ven profesionales.

Buena práctica: separa la lógica de la presentación. Calcula primero, muestra después. Y dale al usuario instrucciones claras de qué escribir: cout << "Escribe tu edad: "; antes de cada cin. Un programa que se queda esperando sin decir qué quiere es un programa mal diseñado.

Ejemplos paso a paso

Ejemplo 1: Tipos y tamaños

```
#include <iostream>

int main() {
    std::cout << "int: " << sizeof(int) << " bytes" << std::endl;
    std::cout << "long long: " << sizeof(long long) << " bytes" << std::endl;
    std::cout << "float: " << sizeof(float) << " bytes" << std::endl;
    std::cout << "double: " << sizeof(double) << " bytes" << std::endl;
    std::cout << "char: " << sizeof(char) << " byte" << std::endl;
    std::cout << "bool: " << sizeof(bool) << " byte" << std::endl;

    int grande{2147483647};
    std::cout << "Maximo int: " << grande << std::endl;
    std::cout << "Maximo int + 1: " << grande + 1 << std::endl;
    return 0;
}
```

La última línea muestra el desbordamiento: sumar 1 al máximo int da un número negativo. Los enteros con signo "dan la vuelta". Es un error silencioso y peligroso: el programa no se queja, solo da un resultado absurdo. Por eso hay que elegir tipos con rango suficiente.

Ejemplo 2: La trampa de la división entera

```
#include <iostream>

int main() {
    int a{7}, b{2};
    std::cout << "7 / 2 = " << a / b << " (division entera)" << std::endl;
    std::cout << "7.0 / 2 = " << 7.0 / b << " (division real)" << std::endl;
    std::cout << "7 % 2 = " << a % b << " (residuo)" << std::endl;

    // Promedio correcto de tres calificaciones
    double c1{8.5}, c2{9.0}, c3{7.5};
    double promedio = (c1 + c2 + c3) / 3.0;
    std::cout << "Promedio: " << promedio << std::endl;
    return 0;
}
```

Fíjate en el promedio: dividimos entre 3.0 (no 3) para forzar división real. Como c1, c2, c3 ya son double, funcionaría igual, pero el hábito te salva cuando los datos son enteros.

Ejemplo 3: Calculadora de propinas con formato

```
#include <iostream>
#include <iomanip>

int main() {
    double cuenta;
    std::cout << "Total de la cuenta: $";
    std::cin >> cuenta;

    double propina = cuenta * 0.15;
    double total = cuenta + propina;

    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Propina (15%): $" << propina << std::endl;
    std::cout << "Total a pagar: $" << total << std::endl;
    return 0;
}
```

Este programa combina todo: lectura con cin, operadores aritméticos, y formato con fixed y setprecision(2) para que el dinero siempre muestre dos decimales. Pruébalo con 250 y verás \$287.50 exactos.

Ejemplo 4: Operadores lógicos en acción

```
#include <iostream>

int main() {
    int edad;
    std::cout << "Tu edad: ";
    std::cin >> edad;

    bool puedeVotar = edad >= 18;
    bool esJoven = edad >= 18 && edad <= 29;
    bool necesitaPermiso = !puedeVotar;

    std::cout << "Puede votar: " << puedeVotar << std::endl;
    std::cout << "Es joven (18-29): " << esJoven << std::endl;
    std::cout << "Necesita permiso: " << necesitaPermiso << std::endl;
    return 0;
}
```

Los bool se muestran como 0 y 1 por defecto. Guarda las condiciones en variables con nombres claros: puedeVotar dice mucho más que edad >= 18 suelto en el código. Esta costumbre hace tu lógica legible.

Práctica

Ejercicio 1

Declara variables de cada tipo fundamental (int, long long, float, double, char, bool) con valores de tu elección e imprímelas con etiquetas. Luego imprime el tamaño de cada una con sizeof.

Solución:

```
#include <iostream>

int main() {
    int entero{42};
    long long grande{9000000000LL};
    float decimalCorto{3.14f};
    double decimalLargo{3.14159265358979};
    char letra{'Z'};
    bool activo{true};

    std::cout << "int: " << entero << " (" << sizeof(int) << " bytes)" <<
std::endl;
    std::cout << "long long: " << grande << " (" << sizeof(long long) << " bytes)"
<< std::endl;
    std::cout << "float: " << decimalCorto << " (" << sizeof(float) << " bytes)" <<
std::endl;
    std::cout << "double: " << decimalLargo << " (" << sizeof(double) << " bytes)"
<< std::endl;
    std::cout << "char: " << letra << " (" << sizeof(char) << " byte)" <<
std::endl;
    std::cout << "bool: " << activo << " (" << sizeof(bool) << " byte)" <<
std::endl;
    return 0;
}
```

Explicación: nota los sufijos: LL para long long y f para float. Sin el sufijo f, 3.14 se trata como double. Son detalles que el compilador agradece que especifiques.

Ejercicio 2

Escribe un programa que convierta grados Celsius a Fahrenheit. Pide los Celsius con cin y aplica la fórmula $F = C * 9/5 + 32$. Asegúrate de que la división sea real, no entera.

Solución:

```
#include <iostream>

int main() {
    double celsius;
    std::cout << "Grados Celsius: ";
    std::cin >> celsius;
    double fahrenheit = celsius * 9.0 / 5.0 + 32;
    std::cout << celsius << " C son " << fahrenheit << " F" << std::endl;
    return 0;
}
```

Explicación: usamos 9.0 / 5.0 en vez de 9/5. Con enteros, 9/5 da 1 (¡no 1.8!) y la conversión saldría mal. Como celsius es double, la operación ya sería real, pero escribir los decimales deja clara tu intención.

Ejercicio 3

Pide al usuario un número entero de segundos y conviértelo a horas, minutos y segundos usando división y módulo. Ejemplo: 3670 segundos = 1 hora, 1 minuto, 10 segundos.

Solución:

```

#include <iostream>

int main() {
    int totalSegundos;
    std::cout << "Segundos: ";
    std::cin >> totalSegundos;

    int horas = totalSegundos / 3600;
    int resto = totalSegundos % 3600;
    int minutos = resto / 60;
    int segundos = resto % 60;

    std::cout << totalSegundos << " segundos son: ";
    std::cout << horas << "h " << minutos << "m " << segundos << "s" << std::endl;
    return 0;
}

```

Explicación: aquí la división entera es justo lo que queremos: $3670 / 3600 = 1$ hora completa. El módulo nos da lo que sobra. Este patrón de "dividir y quedarte con el resto" aparece en relojes, calendarios y paginación.

Ejercicio 4

Pide la base y la altura de un rectángulo y calcula área y perímetro. Muestra los resultados con exactamente 2 decimales usando <iomanip>.

Solución:

```

#include <iostream>
#include <iomanip>

int main() {
    double base, altura;
    std::cout << "Base: ";
    std::cin >> base;
    std::cout << "Altura: ";
    std::cin >> altura;

    double area = base * altura;
    double perimetro = 2 * (base + altura);

    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Area: " << area << std::endl;
    std::cout << "Perimetro: " << perimetro << std::endl;
    return 0;
}

```

Explicación: `fixed + setprecision(2)` se aplican a todo lo que salga después, así que se configuran una vez. Los paréntesis en `2 * (base + altura)` son obligatorios: sin ellos, la multiplicación se haría antes que la suma.

Ejercicio 5

Pide al usuario su año de nacimiento y calcula si ya puede votar (18 años o más en 2026), si tiene descuento de estudiante (menor de 25) y si ambas cosas a la vez. Usa variables `bool` con nombres claros y operadores lógicos.

Solución:


```
#include <iostream>

int main() {
    int anio;
    std::cout << "Año de nacimiento: ";
    std::cin >> anio;

    int edad = 2026 - anio;
    bool puedeVotar = edad >= 18;
    bool descuentoEstudiante = edad < 25;
    bool ambos = puedeVotar && descuentoEstudiante;

    std::cout << "Edad: " << edad << std::endl;
    std::cout << "Puede votar: " << puedeVotar << std::endl;
    std::cout << "Descuento estudiante: " << descuentoEstudiante << std::endl;
    std::cout << "Ambos beneficios: " << ambos << std::endl;
    return 0;
}
```

Explicación: guardar cada condición en un bool con buen nombre convierte lógica abstracta en frases legibles. `ambos = puedeVotar && descuentoEstudiante` se lee solo. Esta es la base de las decisiones que programarás en la unidad 3.

Resumen de la unidad

- Las variables son espacios de memoria con nombre y tipo fijo; inicialízalas siempre para evitar basura.
- Tipos fundamentales: `int`, `long long`, `float`, `double` (el predeterminado para decimales), `char`, `bool`. Usa `double` por defecto y `const` para valores fijos.
- La inicialización con llaves `int x{5}`; es la más segura porque impide conversiones peligrosas.
- Cuidado con la división entre enteros ($7/2 = 3$): usa decimales cuando necesites decimales.
- Operadores: aritméticos (`+` `-` `*` `/` `%`), asignación combinada (`+=`), incremento (`++`), relacionales (`==` `!=` `<` `>`), lógicos (`&&` `||` `!`). En la duda, paréntesis.
- `cout << muestra`, `cin >> lee`; `<iomanip>` con `fixed` y `setprecision` controla el formato.
- En la unidad 3 tu programa empezará a tomar decisiones con `if`, `switch` y ciclos `while`.

Profundización: tipos, límites y conversiones

En la primera parte aprendiste los tipos fundamentales. Ahora vamos más hondo: los límites exactos de cada tipo, qué pasa cuando los rebasas, cómo convertir entre tipos sin sorpresas y herramientas modernas como `constexpr` y las enumeraciones. Esta es la clase de conocimiento que evita bugs que cuestan dinero real.

Los límites exactos: `<limits>`

Cada tipo tiene un rango. Puedes preguntarlo en tu máquina con la cabecera `<limits>`:

```
#include <iostream>
#include <limits>

int main() {
    std::cout << "int: " << std::numeric_limits<int>::min()
               << " a " << std::numeric_limits<int>::max() << std::endl;
    std::cout << "double: " << std::numeric_limits<double>::min()
               << " a " << std::numeric_limits<double>::max() << std::endl;
    std::cout << "double digitos: " << std::numeric_limits<double>::digits10 <<
    std::endl;
    return 0;
}
```

En sistemas típicos de 64 bits: `int` va de -2,147,483,648 a 2,147,483,647; `long long` llega a 9 trillones; `float` tiene ~7 dígitos decimales de precisión; `double` ~15-16. Los `unsigned` duplican el máximo positivo a cambio de no aceptar negativos.

Desbordamiento: el error silencioso

Cuando un cálculo rebasa el rango del tipo, el comportamiento depende del tipo. Con enteros sin signo, "da la vuelta" de forma definida: `unsigned int x = 4294967295; x + 1` da 0. Con enteros con signo, el desbordamiento es comportamiento indefinido: en la práctica suele dar la vuelta y volverse negativo, pero el compilador tiene permiso de hacer cualquier cosa, incluyendo optimizar tu código de formas sorprendentes.

Con flotantes, el desbordamiento da infinito (`inf`) y las operaciones inválidas dan NaN (`not a number`). Prueba:

```
#include <iostream>
#include <cmath>

int main() {
    double grande = 1e308;
    std::cout << grande * 10 << std::endl; // inf
    double nan = std::sqrt(-1.0);
    std::cout << nan << std::endl;          // nan
    std::cout << (nan == nan) << std::endl; // 0: NaN no es igual ni a sí mismo
    return 0;
}
```

El truco de NaN != NaN se usa para detectarlo. En código financiero o científico, estos casos se validan explícitamente.

Conversiones entre tipos

Convertir (castear) es cambiar el tipo de un valor. La forma moderna y segura es `static_cast`:

```
double pi = 3.99;
int entero = static_cast<int>(pi); // 3, trunca decimales
```

`static_cast` deja clara tu intención y el compilador la verifica. La forma vieja de C, `(int)pi`, también funciona pero esconde más: puede hacer conversiones peligrosas sin avisar. En C++ moderno, evita los casts estilo C.

Las conversiones implícitas (automáticas) también ocurren: `int + double` da `double` (el `int` "promociona"). El peligro está en las que pierden información en silencio: `double` a `int` trunca, `int` grande a `short` puede desbordar. La inicialización con llaves `{}` te protege: `int x{3.99}`; no compila, te obliga a decidir.

const y constexpr

`const` declara que algo no cambia: `const double IVA = 0.16;`. El compilador impide modificarlo. `constexpr` va más allá: garantiza que el valor se calcule en tiempo de compilación:

```
constexpr int CUADRADO(int x) { return x * x; }
constexpr int area = CUADRADO(5); // se calcula al compilar, no al ejecutar
```

`constexpr` permite cálculos en compilación: más rápido en ejecución y detecta errores antes. Úsalo para constantes y funciones pequeñas que puedan evaluarse al compilar.

Enumeraciones: enum class

Cuando una variable solo puede tomar valores de un conjunto fijo (días de la semana, estados, opciones), usa enumeraciones. La forma moderna es enum class, que no contamina el alcance global:

```
#include <iostream>

enum class Dia { Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo };
enum class Estado { Pendiente, Aprobado, Rechazado };

std::string nombreDia(Dia d) {
    switch (d) {
        case Dia::Lunes: return "Lunes";
        case Dia::Martes: return "Martes";
        case Dia::Miercoles: return "Miercoles";
        case Dia::Jueves: return "Jueves";
        case Dia::Viernes: return "Viernes";
        case Dia::Sabado: return "Sabado";
        case Dia::Domingo: return "Domingo";
    }
    return "?";
}

int main() {
    Dia hoy = Dia::Viernes;
    std::cout << "Hoy es " << nombreDia(hoy) << std::endl;
    // Estado::Aprobado no se confunde con nada: cada enum es su propio tipo
    return 0;
}
```

enum class evita choques de nombres y conversiones accidentales a int. Mucho mejor que usar números mágicos (1 = lunes, 2 = martes...) regados en el código.

Alias de tipos: using

Para tipos largos o que quieras abreviar:

```
using Entero = int;
using ListaNombres = std::vector<std::string>;
```

using es la versión moderna de typedef. Útil para simplificar firmas complejas y para dar nombres con significado a tipos genéricos.

Más ejemplos

Ejemplo: detectando pérdida de precisión

```
#include <iostream>
#include <iomanip>

int main() {
    float f = 0.1f;
    double d = 0.1;
    std::cout << std::setprecision(20);
    std::cout << "float: " << f << std::endl;
    std::cout << "double: " << d << std::endl;

    // Sumar 0.1 diez veces no da exactamente 1.0
    float suma = 0.0f;
    for (int i = 0; i < 10; i++) suma += 0.1f;
    std::cout << "suma float: " << suma << std::endl;
    std::cout << "igual a 1.0? " << (suma == 1.0f) << std::endl;
    return 0;
}
```

Los decimales como 0.1 no tienen representación binaria exacta: se guardan aproximados. Por eso nunca compares flotantes con `==`: compara la diferencia contra un margen pequeño (epsilon). Y nunca uses float/double para dinero exacto en sistemas contables serios: se usan enteros de centavos.

Ejemplo: calculadora de tipos mixtos bien hecha

```

#include <iostream>
#include <iomanip>

int main() {
    int unidades;
    double precioUnitario;

    std::cout << "Unidades: ";
    std::cin >> unidades;
    std::cout << "Precio unitario: $";
    std::cin >> precioUnitario;

    double subtotal = static_cast<double>(unidades) * precioUnitario;
    constexpr double IVA = 0.16;
    double total = subtotal * (1.0 + IVA);

    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Subtotal: $" << subtotal << std::endl;
    std::cout << "IVA: $" << subtotal * IVA << std::endl;
    std::cout << "Total: $" << total << std::endl;
    return 0;
}

```

`static_cast<double>(unidades)` documenta la conversión. `constexpr` para el IVA: si la tasa cambia, se cambia en un lugar. Todo el dinero con 2 decimales.

Práctica adicional

Ejercicio A

Investiga con `<limits>` los rangos de `short`, `unsigned int` y `long long` en tu máquina e imprímelos. Luego provoca un desbordamiento controlado con `unsigned int` y observa la "vuelta".

Solución:

```

#include <iostream>
#include <limits>

int main() {
    std::cout << "short: " << std::numeric_limits<short>::min()
               << " a " << std::numeric_limits<short>::max() << std::endl;
    std::cout << "unsigned int: 0 a "
               << std::numeric_limits<unsigned int>::max() << std::endl;
    std::cout << "long long: " << std::numeric_limits<long long>::min()
               << " a " << std::numeric_limits<long long>::max() << std::endl;

    unsigned int x = std::numeric_limits<unsigned int>::max();
    std::cout << "max + 1 = " << x + 1 << " (da la vuelta a 0)" << std::endl;
    return 0;
}

```

Explicación: con unsigned el desbordamiento está definido (vuelta modular). Con signed es indefinido: nunca dependas de su comportamiento.

Ejercicio B

Escribe un programa que compare dos doubles con un margen epsilon en vez de `==`. Pide dos números y di si son "iguales" con tolerancia de 0.00001.

Solución:

```

#include <iostream>
#include <cmath>

int main() {
    double a, b;
    std::cout << "Dos numeros: ";
    std::cin >> a >> b;
    constexpr double EPSILON = 0.00001;
    if (std::fabs(a - b) < EPSILON) {
        std::cout << "Son iguales (dentro de tolerancia)." << std::endl;
    } else {
        std::cout << "Son diferentes." << std::endl;
    }
    return 0;
}

```

Explicación: fabs es el valor absoluto para doubles. Este patrón aparece en juegos, física y gráficos: comparar flotantes con == es un bug clásico.

Ejercicio C

Crea un enum class Mes con los 12 meses y una función diasDelMes(Mes m, bool bisiesto) que devuelva los días. Prueba con febrero en año bisiesto y no bisiesto.

Solución:

```
#include <iostream>

enum class Mes { Ene, Feb, Mar, Abr, May, Jun, Jul, Ago, Sep, Oct, Nov, Dic };

int diasDelMes(Mes m, bool bisiesto) {
    switch (m) {
        case Mes::Feb: return bisiesto ? 29 : 28;
        case Mes::Abr: case Mes::Jun: case Mes::Sep: case Mes::Nov:
            return 30;
        default:
            return 31;
    }
}

int main() {
    std::cout << "Feb 2024: " << diasDelMes(Mes::Feb, true) << std::endl;
    std::cout << "Feb 2025: " << diasDelMes(Mes::Feb, false) << std::endl;
    std::cout << "Abr: " << diasDelMes(Mes::Abr, false) << std::endl;
    return 0;
}
```

Explicación: el enum hace el código autodocumentado: diasDelMes(Mes::Feb, true) se entiende solo. Los casos agrupados en el switch evitan repetición.

Ejercicio D

Escribe una función double aEnteroSeguro(double valor) que trunque el decimal pero lance una advertencia en cerr si el valor está fuera del rango de int. Usa <limits> y static_cast.

Solución:


```

#include <iostream>
#include <limits>

int aEnteroSeguro(double valor) {
    if (valor > static_cast<double>(std::numeric_limits<int>::max()) ||
        valor < static_cast<double>(std::numeric_limits<int>::min())) {
        std::cerr << "[AVISO] valor fuera de rango, se trunca" << std::endl;
    }
    return static_cast<int>(valor);
}

int main() {
    std::cout << aEnteroSeguro(9.99) << std::endl;
    std::cout << aEnteroSeguro(1e20) << std::endl;
    return 0;
}

```

Explicación: convertir sin verificar es como cruzar la calle sin mirar. Esta función verifica antes de castear: el patrón de "conversión defensiva" que usan las bibliotecas serias.

Expansión Unidad 2: lógica, caracteres, bits y el detalle fino de los tipos

En las dos primeras partes de esta unidad aprendiste los tipos fundamentales, los operadores básicos, los límites de cada tipo y las conversiones seguras. Esta tercera parte te lleva al nivel de detalle que separa al programador que "más o menos entiende" del que domina: la lógica booleana hasta sus últimas consecuencias, el tipo char y su relación secreta con los números, los operadores bit a bit que te dan control sobre cada interruptor de la memoria, la tabla completa de precedencia para que nunca más dudes del orden de evaluación, el operador ternario exprimido al máximo, los secretos de sizeof, los alias con using en situaciones reales y el mundo de las constantes literales con sus sufijos y bases numéricas. Al terminar, los fundamentos dejarán de ser "lo básico" para convertirse en tu superpoder: la mayoría de los bugs sutiles de C++ viven exactamente aquí.

Lógica booleana a fondo: las tablas de verdad aplicadas

En la primera parte usaste bool con &&, || y ! para tomar decisiones. Ahora vamos a entender la lógica booleana como lo que es: un sistema matemático completo con reglas precisas. George Boole la inventó en 1854, mucho antes de las computadoras, y resulta que describe perfectamente cómo piensan los circuitos y los programas.

Las tablas de verdad, sin misterio

Una tabla de verdad lista todas las combinaciones posibles de entradas y el resultado de la operación. Con dos valores booleanos A y B hay exactamente cuatro combinaciones. Estas son las tablas de los tres operadores que ya conoces, más el "o exclusivo" que construiremos:

Operador && (Y lógico): el resultado es verdadero solo si AMBOS son verdaderos.

- $V \&\& V = V$
- $V \&\& F = F$
- $F \&\& V = F$
- $F \&\& F = F$

Lectura práctica: "puede entrar al antro si tiene INE Y tiene boleto". Si falta cualquiera de los dos, no entra.

Operador || (O lógico): el resultado es verdadero si AL MENOS UNO es verdadero.

- $V || V = V$
- $V || F = V$
- $F || V = V$
- $F || F = F$

Lectura práctica: "hay descuento si es estudiante O es adulto mayor". Basta cumplir una condición.

Operador ! (NO lógico): invierte el valor. Solo tiene una entrada.

- $!V = F$
- $!F = V$

Lectura práctica: "si NO hay sistema, muestra el error".

El O exclusivo (XOR): verdadero cuando los valores son DIFERENTES. C++ no tiene un operador lógico XOR directo para bool, pero se construye fácil: $A != B$ hace exactamente eso (si son diferentes, el resultado es verdadero).

- $V \text{ XOR } V = F$
- $V \text{ XOR } F = V$
- $F \text{ XOR } V = V$
- $F \text{ XOR } F = F$

Lectura práctica: "el interruptor de la escalera": la luz cambia cuando exactamente uno de los dos apagadores cambia. Lo usarás en validaciones del tipo "exactamente una de las dos opciones".

Las leyes que te ahorran dolores de cabeza

La lógica booleana tiene leyes algebraicas como las matemáticas, y conocerlas te permite simplificar condiciones enredadas:

Leyes de De Morgan (las más importantes de todas): $!(A \ \&\& \ B)$ es lo mismo que $!A \ || \ !B$; y $!(A \ || \ B)$ es lo mismo que $!A \ \&\& \ !B$. En palabras: "negar un Y" equivale a "negar cada parte y unir las con O", y viceversa. Ejemplo real: la condición "no es (mayor de edad y tiene licencia)" se simplifica a "es menor de edad o no tiene licencia". Cuando veas un `!` afuera de un paréntesis con `&&` o `||` adentro, aplica De Morgan y la condición se vuelve legible.

Doble negación: $!!A$ es lo mismo que A . Parece obvio, pero aparece en código real cuando alguien niega una condición ya negada.

Distributividad: $A \ \&\& \ (B \ || \ C)$ es lo mismo que $(A \ \&\& \ B) \ || \ (A \ \&\& \ C)$. Igual que en aritmética con la multiplicación sobre la suma.

Leyes de absorción: $A \ || \ (A \ \&\& \ B)$ es simplemente A . Si ya tienes A , el resto sobra. Estas simplificaciones no son adorno académico: una condición simplificada es una condición con menos lugares donde equivocarse.

El cortocircuito, explicado en serio

En la primera parte mencionamos que `&&` y `||` tienen "cortocircuito": en $A \ \&\& \ B$, si A es falso, B ni se evalúa; en $A \ || \ B$, si A es verdadero, B ni se evalúa. Esto no es un detalle de implementación: es parte del lenguaje y los programadores profesionales lo usan a propósito como herramienta.

El uso clásico es la guarda: `if (p != nullptr && p->valor > 0)`. Si p es nulo, la segunda parte jamás se ejecuta, así que nunca intentas leer `p->valor` de un puntero nulo. Sin cortocircuito, ese código tronaría. El orden importa: la condición barata y segura va primero, la cara o peligrosa va después.

El reverso peligroso: si B tiene un efecto secundario (modifica algo), ese efecto a veces no ocurre. En `if (listo && (contador++ > 5))`, si `listo` es falso, el contador no se incrementa. Si tu lógica depende de que el incremento siempre pase, tienes un bug intermitente: funciona unas veces y otras no, el peor tipo de bug. Regla de oro: nunca pongas efectos secundarios (`++`, `--`, asignaciones, llamadas que modifican estado) dentro de una condición con `&&` u `||` a menos que el cortocircuito sea exactamente lo que quieres. Haz el efecto antes, en su propia línea, y deja la condición limpia.

Ejemplo completo: tablas de verdad y lógica aplicada

Este programa imprime las tablas de verdad y luego resuelve un problema real de elegibilidad con lógica combinada:

```
#include <iostream>

// Imprime un bool como V/F para que la tabla se lea facil
char letra(bool b) {
    return b ? 'V' : 'F'; // operador ternario: lo veremos a fondo mas abajo
}

void tablas() {
    std::cout << "A B | A&&B A||B !A A!=B(XOR)" << std::endl;
    bool valores[2]{false, true};
    for (bool a : valores) {
        for (bool b : valores) {
            std::cout << letra(a) << " " << letra(b) << " | "
                << letra(a && b) << " " << letra(a || b) << " "
                << letra(!a) << " " << letra(a != b) << std::endl;
        }
    }
}

int main() {
    tablas();

    // Problema real: beca escolar
    // Requisitos: promedio >= 8.5 Y (sin materias reprobadas 0 con carta de apoyo)
    // Excepcion: si tiene reporte disciplinario, NO hay beca aunque cumpla lo demas
    double promedio{9.1};
    bool reprobadas{false};
    bool cartaApoyo{true};
    bool reporte{false};

    bool cumplePromedio{promedio >= 8.5};
```

```

    bool situacionOk{!reprobadas || cartaApoyo};
    bool elegible{cumplePromedio && situacionOk && !reporte};

    std::cout << std::endl;
    std::cout << "Cumple promedio: " << letra(cumplePromedio) << std::endl;
    std::cout << "Situacion ok: " << letra(situacionOk) << std::endl;
    std::cout << "Elegible para beca: " << letra(elegible) << std::endl;

    // Demostracion de De Morgan: estas dos condiciones son identicas
    bool forma1{!(cumplePromedio && situacionOk)};
    bool forma2{!cumplePromedio || !situacionOk};
    std::cout << "De Morgan: " << letra(forma1) << " == " << letra(forma2) <<
std::endl;
    return 0;
}

```

Vamos por partes. La función `letra` convierte un `bool` a 'V' o 'F' usando el operador ternario (condición ? valor_si_verdadero : valor_si_falso): es un adelanto, pero aquí su uso es tan simple que se entiende solo. La función `tablas` usa dos ciclos `for` de rango anidados sobre el arreglo `{false, true}`: genera las cuatro combinaciones de A y B y para cada una imprime los resultados de `&&`, `||`, `!` y `!=` (nuestro XOR). Es la tabla de verdad generada por el propio programa: confía, pero verifica.

En el problema de la beca, fíjate en la técnica: cada sub-condición se guarda en un `bool` con nombre descriptivo (`cumplePromedio`, `situacionOk`) antes de combinarlas. La condición final `elegible` se lee como una frase: "cumple el promedio Y su situación está ok Y no tiene reporte". Si hubiéramos escrito todo en una sola línea gigante, sería correcto pero ilegible; así, cada pieza se puede verificar por separado. La última parte demuestra De Morgan en vivo: `forma1` y `forma2` siempre imprimen lo mismo, sin importar los valores. Cambia los valores de las variables y compruébalo: es un experimento que vale más que mil explicaciones.

El tipo `char` y la tabla ASCII: cuando las letras son números

El tipo `char` guarda un solo carácter, pero aquí está el secreto que todo programador de C++ debe saber: un `char` ES un número entero pequeño. Cuando escribes `char letra{'A'}`, en memoria se guarda el número 65. La 'A' es solo la interpretación humana de ese número según una tabla de correspondencia llamada ASCII. Esta dualidad (es letra y es número a la vez) es una de las herramientas más útiles del lenguaje.

La tabla ASCII esencial

ASCII (American Standard Code for Information Interchange) asigna un número del 0 al 127 a cada carácter. No necesitas memorizarla completa, pero sí conocer sus zonas:

- Del 0 al 31: caracteres de control (no se ven). Los importantes: 9 es tabulación, 10 es salto de línea (`\n`), 13 es retorno de carro (`\r`).
- El 32: el espacio ' '. El carácter invisible más usado de la historia.
- Del 33 al 47: signos de puntuación y símbolos: `! " # $ % & ' ( ) * + , - . /`
- Del 48 al 57: los dígitos '0' a '9'. En ese orden, consecutivos. Este dato es oro: `'5' - '0' = 5`.
- Del 58 al 64: más símbolos: `: ; < = > ? @`
- Del 65 al 90: las mayúsculas 'A' a 'Z', consecutivas y en orden alfabético.
- Del 91 al 96: más símbolos: `[\] ^ _ ``
- Del 97 al 122: las minúsculas 'a' a 'z', consecutivas y en orden alfabético.
- El 127: el carácter de borrado (DEL).

Tres consecuencias prácticas que usarás toda la vida. Primera: los dígitos están consecutivos, así que convertir un carácter dígito a su valor numérico es restar '0': `char c{'7'}; int valor{c - '0'}; // 7`. Ya lo usaste en la expansión de la unidad 1 sin saber por qué funcionaba; ahora sabes el porqué. Segunda: las mayúsculas y minúsculas están en bloques consecutivos separados por exactamente 32 posiciones: `'a' - 'A' == 32`. Convertir mayúscula a minúscula es sumar 32 (o hacer OR con 32, truco de bits que verás abajo). Tercera: puedes comparar caracteres con `<` y `>` porque en realidad comparas sus números: `'b' > 'a'` es verdadero, y por eso ordenar texto alfabéticamente funciona.

Ojo importante: ASCII solo cubre el inglés básico. Las letras con acento (á, é, ñ) y otros alfabetos viven en tablas extendidas como Latin-1 o en Unicode (UTF-8). En este libro, cuando necesites la ñ en un programa, la escribes normal en el texto entre comillas y funciona, porque el código fuente se guarda en UTF-8; pero las operaciones carácter por carácter (como sumar 32) solo son seguras con el rango ASCII básico. Si algún día procesas texto en español letra por letra, necesitarás bibliotecas de Unicode: tema avanzado, pero ya sabes que el problema existe.

char con signo, sin signo y sus trampas

El tipo `char` ocupa 1 byte, pero hay un detalle traicionero: el estándar no dice si `char` es con signo o sin signo; depende del compilador. En la mayoría de las plataformas comunes es con signo (rango -128 a 127), pero no cuentes con eso. Si vas a usar un `char` como número

pequeño, sé explícito: signed char (de -128 a 127) o unsigned char (de 0 a 255). Si lo vas a usar como carácter, char a secas está bien. Esta ambigüedad ha causado bugs portables legendarios: código que funciona en tu máquina y falla en otra porque el char cambió de signo.

Otro detalle: char, signed char y unsigned char son tres tipos distintos para el compilador, aunque char se comporte como uno de los otros dos. Y un error clásico de principiante: char letra{"A"}; con comillas dobles. Las comillas dobles crean una cadena (arreglo de caracteres), no un carácter: el compilador se queja. Carácter solo: comillas simples. Texto: comillas dobles. Sin excepciones.

Ejemplo completo: tabla ASCII y jugando con caracteres

```
#include <iostream>

void tablaASCII() {
    std::cout << "Dec Hex Car | Dec Hex Car | Dec Hex Car" << std::endl;
    // Caracteres imprimibles: del 33 al 126
    for (int i{33}; i <= 54; ++i) {
        for (int j{0}; j < 3; ++j) {
            int codigo{i + j * 32};
            if (codigo <= 126) {
                char c{static_cast<char>(codigo)};
                std::cout << codigo << " " << std::hex << codigo << std::dec
                    << " " << c << " | ";
            }
        }
        std::cout << std::endl;
    }
}

char aMayuscula(char c) {
    // Solo convierte si es minuscúla ASCII
    if (c >= 'a' && c <= 'z') {
        return static_cast<char>(c - 32); // 'a'-'A' hay 32 de diferencia
    }
    return c;
}

int main() {
    tablaASCII();

    char digito{'7'};
    int valor{digito - '0'}; // truco ASCII: los dígitos son consecutivos
    std::cout << "\nEl carácter '" << digito << "' vale " << valor << std::endl;
```

```

std::cout << "Duplicado: " << valor * 2 << std::endl;

std::string texto{"Hola Mundo 123"};
std::cout << "Original: " << texto << std::endl;
std::cout << "Mayusculas: ";
for (char c : texto) {
    std::cout << aMayuscula(c);
}
std::cout << std::endl;

// Los char son numeros: aritmetica directa
char letra{'A'};
std::cout << "'A' + 1 = '" << static_cast<char>(letra + 1) << "'" << std::endl;
std::cout << "Comparacion: 'b' > 'a' es " << ('b' > 'a') << std::endl;
return 0;
}

```

Línea por línea lo nuevo: tablaASCII imprime los caracteres del 33 al 126 en tres columnas; el truco $i + j * 32$ reparte los 94 caracteres en 3 columnas de 32 (las zonas de 32 en ASCII no son coincidencia: el diseño de la tabla es deliberado). `std::hex` y `std::dec` son manipuladores que cambian la base de impresión: mostramos cada código en decimal y hexadecimal. `static_cast<char>(codigo)` convierte el número a su carácter: es la puerta entre los dos mundos del char. La función `aMayuscula` verifica que el carácter esté en el rango 'a'-'z' antes de restar 32: esa verificación es lo que la hace segura (si le pasas un '5' o una 'ñ', lo devuelve intacto). En main, `digito - '0'` convierte el carácter a su valor numérico real, y luego podemos multiplicarlo: con el carácter no podríamos hacer aritmética con sentido. El ciclo sobre el string convierte cada letra con nuestra función. Y al final, `letra + 1` demuestra la aritmética directa: 'A' + 1 es 'B', porque $65 + 1 = 66$.

Operadores bit a bit: controlando cada interruptor

Llegamos a los operadores más misteriosos para el principiante y más amados por el programador de sistemas: los operadores bit a bit. Trabajan sobre la representación binaria de los números, bit por bit. Si los operadores lógicos (`&&`, `||`) trabajan con valores verdadero/falso completos, los operadores bit a bit (`&`, `|`, `^`, `~`) trabajan con cada uno de los bits individuales del número. Son la herramienta para hablar directamente con el hardware: configurar dispositivos, empacar muchas banderas en un solo byte, cifrado simple, gráficos y protocolos de red.

Primero, la base: todo entero se guarda en binario. El número 13 en un byte es 00001101 ($8 + 4 + 1$). El número 11 es 00001011 ($8 + 2 + 1$). Los operadores bit a bit combinan estos patrones columna por columna.

Los cuatro operadores lógicos de bits

AND bit a bit (&): el bit resultado es 1 solo si AMBOS bits son 1. $13 \& 11$: $00001101 \& 00001011 = 00001001 = 9$. Se usa como máscara para APAGAR bits: cualquier bit donde la máscara tenga 0, se apaga en el resultado.

OR bit a bit (|): el bit resultado es 1 si AL MENOS UNO es 1. $13 | 11$: $00001101 | 00001011 = 00001111 = 15$. Se usa para ENCENDER bits: cualquier bit donde la máscara tenga 1, se enciende.

XOR bit a bit (^): el bit resultado es 1 si los bits son DIFERENTES. $13 \wedge 11$: $00001101 \wedge 00001011 = 00000110 = 6$. Se usa para ALTERNAR (toggle) bits: donde la máscara tenga 1, el bit cambia; donde tenga 0, queda igual. También es la base del cifrado más simple: aplicar XOR dos veces con la misma clave recupera el original.

NOT bit a bit (~): invierte todos los bits (los 1 se vuelven 0 y viceversa). ~ 13 en un byte: 00001101 se vuelve 11110010 . Ojo: en un int de 32 bits invierte los 32 bits, así que ~ 13 no es 242 sino -14 (por la representación de complemento a dos de los negativos). Este es el operador unario del grupo: solo necesita un operando.

No confundas & con && ni | con ||: son mundos distintos. & trabaja bit por bit sobre enteros y siempre evalúa ambos lados; && trabaja con valores lógicos y tiene cortocircuito. El error de escribir & queriendo decir && compila sin quejarse (ambos aceptan enteros) pero se comporta diferente: es uno de los bugs más difíciles de cazar porque "casi funciona".

Desplazamientos: << y >>

El operador << desplaza los bits a la izquierda, rellenando con ceros a la derecha. $13 \ll 2$: 00001101 se vuelve $00110100 = 52$. Cada desplazamiento a la izquierda multiplica por 2 ($13 * 4 = 52$). El operador >> desplaza a la derecha: $13 \gg 2$: 00001101 se vuelve $00000011 = 3$. Cada desplazamiento a la derecha divide entre 2 truncando ($13 / 4 = 3$).

Detalle crucial: << y >> son también los operadores de cout y cin. El compilador distingue por el contexto: con un flujo a la izquierda son entrada/salida; con un entero a la izquierda son desplazamientos. La sobrecarga de operadores (unidad 7) permite este doble uso.

Advertencias serias: desplazar un número con signo a la izquierda hasta que se desborde es comportamiento indefinido; desplazar a la derecha un negativo puede rellenar con unos o con ceros según la plataforma (desplazamiento aritmético vs lógico). Para trabajo con bits, usa siempre tipos sin signo (unsigned int, uint32_t): su comportamiento está perfectamente definido. Y nunca desplaces por un número de posiciones mayor o igual al tamaño del tipo: también es indefinido.

Banderas y máscaras: el uso profesional

La aplicación estrella de los bits es empacar muchas opciones booleanas en un solo número. En vez de ocho variables bool (8 bytes), usas un byte donde cada bit es una opción. Los sistemas operativos, los videojuegos y los protocolos de red hacen esto constantemente.

Ejemplo completo: sistema de permisos de un archivo, estilo Unix:

```
#include <iostream>
#include <string>

// Cada permiso es UN bit encendido en su posicion
constexpr unsigned int PERM_LEER{1u << 0};      // 0001 = 1
constexpr unsigned int PERM_ESCRIBIR{1u << 1};   // 0010 = 2
constexpr unsigned int PERM_EJECUTAR{1u << 2};   // 0100 = 4
constexpr unsigned int PERM_BORRAR{1u << 3};     // 1000 = 8

// Muestra los 4 bits relevantes como texto
std::string aBits(unsigned int valor) {
    std::string s;
    for (int i{3}; i >= 0; --i) {
        s += ((valor >> i) & 1u) ? '1' : '0';
    }
    return s;
}

void mostrar(const std::string& nombre, unsigned int permisos) {
    std::cout << nombre << " [" << aBits(permisos) << "] ";
    std::cout << ((permisos & PERM_LEER) ? "L" : "-");
    std::cout << ((permisos & PERM_ESCRIBIR) ? "E" : "-");
    std::cout << ((permisos & PERM_EJECUTAR) ? "X" : "-");
    std::cout << ((permisos & PERM_BORRAR) ? "B" : "-");
    std::cout << std::endl;
}

int main() {
    unsigned int permisos{0}; // todo apagado: 0000

    // ENCENDER bits con OR: el permiso se agrega sin tocar los demas
    permisos = permisos | PERM_LEER;      // 0001
    permisos = permisos | PERM_ESCRIBIR;   // 0011
    mostrar("Inicial", permisos);

    // Forma corta: |= enciende bits
    permisos |= PERM_EJECUTAR;             // 0111
    mostrar("Con ejecutar", permisos);
```

```

// VERIFICAR un bit con AND: si el resultado no es cero, el bit estaba encendido
if ((permisos & PERM_BORRAR) != 0) {
    std::cout << "Puede borrar" << std::endl;
} else {
    std::cout << "NO puede borrar" << std::endl;
}

// APAGAR bits con AND + NOT: la mascara invierte el bit a apagar
permisos = permisos & ~PERM_ESCRIBIR; // 0111 & 1101 = 0101
mostrar("Sin escribir", permisos);

// ALTERNAR bits con XOR: el bit cambia, los demas quedan igual
permisos = permisos ^ PERM_BORRAR; // 0101 ^ 1000 = 1101
mostrar("Toggle borrar", permisos);
permisos = permisos ^ PERM_BORRAR; // 1101 ^ 1000 = 0101 (revierte)
mostrar("Toggle otra vez", permisos);

return 0;
}

```

Este ejemplo merece atención línea por línea porque es el patrón que verás en código profesional. Las constantes se definen con `1u << n`: el número 1 desplazado a la posición n. La u significa unsigned (veremos sufijos abajo): así cada permiso es exactamente un bit. La función `aBits` convierte el número a texto de bits para visualizarlo: `(valor >> i) & 1u` desplaza el bit i hasta la posición 0 y lo aísla con AND 1; el ternario elige '1' o '0'.

Las cuatro operaciones fundamentales: encender con OR (`permisos | PERM_LEER`): los bits en 1 de la máscara se encienden, los demás no se tocan. La forma corta `|=` es el estándar en código real. Verificar con AND: `permisos & PERM_BORRAR` deja solo el bit de borrar; si el resultado es distinto de cero, el permiso está activo. Apagar con AND + NOT: `~PERM_ESCRIBIR` invierte la máscara (1111...1101) y el AND apaga justo ese bit sin tocar los demás. Alternar con XOR: el bit se invierte y aplicarlo dos veces revierte el cambio (por eso XOR sirve para cifrado reversible).

Fíjate en los paréntesis de `(permisos & PERM_LEER)`: son obligatorios porque el ternario `?:` tiene menor precedencia que `&`. Sin paréntesis, la expresión se agruparía mal. Hablando de precedencia...

La tabla completa de precedencia de operadores

En la primera parte viste la precedencia básica. Aquí está la tabla completa para los operadores que conoces hasta ahora, de mayor a menor prioridad (lo que está arriba se evalúa primero):

1. Acceso y llamadas: `()` llamadas, `[]` subíndices (los verás en arreglos).
2. Unarios y sizeof: `!` (no lógico), `~` (not de bits), `+` `-` (signos), `++` `--` (pre), `sizeof`, casts.
3. Multiplicativos: `*` `/` `%`.
4. Aditivos: `+` `-`.
5. Desplazamientos: `<<` `>>`.
6. Relacionales de orden: `<` `<=` `>` `>=`.
7. Relacionales de igualdad: `==` `!=`.
8. AND de bits: `&`.
9. XOR de bits: `^`.
10. OR de bits: `|`.
11. AND lógico: `&&`.
12. OR lógico: `||`.
13. Ternario: `?:`.
14. Asignación: `=` `+=` `-=` `*=` `/=` `%=` `<<=` `>>=` `&=` `|=` `^=`, y también `++` `--` (post).
15. Coma: `,` (el de menor prioridad de todos).

Hay dos sorpresas en esta tabla que causan bugs reales. Primera: los operadores de bits (`&`, `|`, `^`) tienen MENOR prioridad que `==` y `!=`. La expresión `a & mascara == 0` NO significa `(a & mascara) == 0`, sino `a & (mascara == 0)`, porque `==` está en el nivel 7 y `&` en el 8. El resultado es un sinsentido silencioso. Por eso en el ejemplo de permisos escribimos `(permisos & PERM_BORRAR) != 0` con paréntesis: sin ellos, el código compila pero hace otra cosa. Esta es una de las trampas más famosas de C y C++.

Segunda: el desplazamiento `<<` tiene MENOR prioridad que `+` y `-`. La expresión `1 << 2 + 3` significa `1 << (2 + 3) = 32`, no `(1 << 2) + 3 = 7`. Cuando definimos `PERM_LEER` como `1u << 0` no hay problema, pero en expresiones mixtas los paréntesis salvan.

La regla profesional: no memorices la tabla entera; memoriza las trampas (`&` vs `==`, `<<` vs `+`) y usa paréntesis generosos en todo lo demás. Los paréntesis de más cuestan cero en

rendimiento (el compilador los optimiza) y valen oro en legibilidad. El código que necesita que el lector memorice la precedencia es código mal escrito.

El operador ternario a fondo

El operador ternario `?:` es el único operador de C++ que toma tres operandos: condición ? valor_si_verdadero : valor_si_falso. Evalúa la condición; si es verdadera, la expresión completa vale lo de la izquierda de los dos puntos; si no, lo de la derecha. Ya lo usaste en los ejemplos de arriba; ahora domínalo.

Su uso ideal es elegir entre dos valores de forma compacta: `int mayor = (a > b) ? a : b;`. Es exactamente equivalente a un `if-else` que asigna, pero en una línea y, lo más importante, como EXPRESIÓN: puedes usarlo dentro de otra expresión, como argumento de función o al inicializar una constante. Un `if` no puede hacer eso: `if` es una instrucción, `?:` es una expresión. Por eso pudimos escribir `const std::string saludo = (idioma == "en") ? "Hello" : "Hola";`; con `if-else` necesitarías declarar la variable primero y asignar después, perdiendo el `const`.

Los dos valores deben ser compatibles: el compilador busca un tipo común. `(cond) ? 5 : 3.14` da un `double` (el `int` promociona). `(cond) ? "hola" : 'x'` es un problema: uno es texto y otro es carácter, no hay conversión razonable y el compilador se queja. Cuando mezcles tipos distintos a propósito, usa `static_cast` para dejar clara tu intención.

Los ternarios anidados (un `?:` dentro de otro) son legales pero peligrosos: `int nota = (p >= 90) ? 5 : (p >= 70) ? 4 : 3;`. Funciona (se agrupa a la derecha), pero más de un nivel de anidación se vuelve ilegible rapidísimo. Regla práctica: un ternario simple sí; dos niveles, solo si es trivial; tres o más, usa `if-else` o un `switch`. El código se lee muchas más veces de las que se escribe: ahorra caracteres solo cuando no cueste claridad.

Otro uso elegante: elegir qué mostrar sin repetir el `cout`. En vez de `if (x > 0) cout << "positivo"; else cout << "negativo";`, puedes escribir `cout << (x > 0 ? "positivo" : "negativo");`. Y combinado con lo anterior: `std::cout << "Tienes " << n << (n == 1 ? " mensaje" : " mensajes") << std::endl;` maneja el plural correctamente en una sola línea. Este truco aparece en todo software con textos para el usuario.

sizeof aplicado a expresiones: más de lo que parece

Ya sabes que `sizeof(tipo)` dice cuántos bytes ocupa un tipo. Pero `sizeof` también acepta expresiones: `sizeof(x + 1)` te dice el tamaño del RESULTADO de `x + 1`, sin evaluar la expresión. Ese "sin evaluar" es importante: `sizeof(i++)` NO incrementa `i`. El compilador solo

mira los tipos involucrados y calcula el tamaño en tiempo de compilación. Es un operador de tiempo de compilación, no una función: no hay llamada, no hay costo en ejecución.

Esto revela las promociones de tipos: `sizeof('a')` en C++ es 1 (un literal de carácter es `char`, a diferencia de C donde es `int`: otra diferencia sutil entre los lenguajes). `sizeof(5 + 3.14)` es 8: el `int` promociona a `double` y el resultado es `double`. `sizeof(x > 0)` es 1: una comparación devuelve `bool`.

El uso clásico es no repetirse: en vez de escribir el tamaño de un tipo dos veces, usa `sizeof` sobre la variable: `double precios[100]; ... sizeof(precios) / sizeof(precios[0])` da el número de elementos ($800 / 8 = 100$). Si mañana cambias `double` por `float`, la cuenta se ajusta sola. Este idioma aparece en todo código C++ que maneja arreglos crudos (en la unidad 4 verás por qué `vector` lo hace innecesario, pero debes reconocer el patrón en código ajeno).

Truco avanzado que verás en bibliotecas: `sizeof` se usa para detectar características en tiempo de compilación (¿qué tamaño tiene este tipo en esta plataforma?) y tomar decisiones con `constexpr`. No necesitas dominarlo ahora, pero cuando lo veas, sabrás que es el compilador midiéndose a sí mismo.

Alias con using en contextos reales

En la segunda parte viste `Entero = int;`. Parece un simple apodo, pero en contextos reales los alias resuelven problemas concretos. `using` es la versión moderna de `typedef` (que todavía verás en código viejo): hace lo mismo pero se lee mejor, sobre todo con tipos complejos, porque el nombre nuevo va a la izquierda, como en una declaración normal.

Contexto real 1: domar tipos largos de la STL. `std::map<std::string, std::vector<int>>` es un tipo válido pero escribirlo cinco veces es invitar errores de dedo. Con `using Calificaciones = std::map<std::string, std::vector<int>>;`, el código habla en tu idioma: `Calificaciones grupo;`. Si mañana cambias la estructura interna, lo cambias en un solo lugar. Los alias documentan la intención: el nombre dice para qué sirve el tipo.

Contexto real 2: independizarte de decisiones de tamaño. `using EnteroGrande = long long;` en un solo encabezado de tu proyecto. Si algún día necesitas 128 bits o quieres fijar un tipo de ancho exacto, cambias una línea. Las bibliotecas serias hacen esto: definen sus tipos una vez y el resto del código usa los alias.

Contexto real 3: alias de plantillas (esto es lo que `typedef` no puede hacer). `template<typename T> using Matriz = std::vector<std::vector<T>>;` crea un alias que funciona para cualquier tipo: `Matriz<int>` es una matriz de enteros, `Matriz<double>` de decimales. Es

una herramienta de la unidad de plantillas, pero desde ahora sabes que `using` llega donde `typedef` no llega.

Regla de estilo: los alias van en minúsculas o PascalCase según tu convención, pero lo importante es que el nombre explique el propósito, no la estructura: mejor `Puntaje` que `EnteroSinSignoDe32Bits`. Y no abuses: un alias para `int` en un programa de 20 líneas es ruido; en un proyecto de 20 archivos con tipos de la STL anidados, es salvación.

Constantes literales: sufijos, bases y separadores

Una constante literal es un valor escrito directamente en el código: `42`, `3.14`, `'A'`, `"hola"`. Cada literal tiene un tipo, y C++ te da controles finos para elegirlo. Dominar esto evita advertencias del compilador y bugs de precisión.

Sufijos: el tipo exacto que quieres

Los enteros sin sufijo son `int` por defecto: `42` es `int`. Pero a veces necesitas otro tipo. Los sufijos: `u` o `U` para unsigned (`42u` es unsigned int), `l` o `L` para long (`42L`), `ll` o `LL` para long long (`42LL`). Se pueden combinar: `42ULL` es unsigned long long. Usa siempre las mayúsculas: `42l` (de minúscula) se confunde con el número 1 a simple vista y es una invitación al error.

Los decimales sin sufijo son `double`: `3.14` es `double`. El sufijo `f` o `F` lo vuelve float: `3.14f`. Sin la `f`, guardar `3.14` en un float genera una advertencia de posible pérdida de precisión con `-Wall`: el compilador te está diciendo "ese double no cabe exacto en tu float". Para long double, el sufijo es `l` o `L`: `3.14L` (sí, se confunde con el de long; el contexto lo distingue).

Los caracteres tienen sufijos Unicode en C++ moderno (`u8'a'`, `u'a'`, `U'a'`, `L'a'`), pero para este nivel basta saber que existen: cuando trabajes con texto internacional de verdad, los estudiarás con Unicode.

Bases numéricas: decimal, hexadecimal, octal y binario

Ya conoces el decimal (base 10) y el binario `0b` de C++14. Las otras dos bases:

Hexadecimal (base 16) con prefijo `0x`: `0xFF` = 255, `0x1A3F` = 6719. Usa dígitos 0-9 y letras A-F (mayúsculas o minúsculas). Es la base favorita para trabajo con bits porque cada dígito hexadecimal es exactamente 4 bits: `0xF0` es 11110000. Cuando veas máscaras como `0xFF00`, léelas como patrones de bits, no como números.

Octal (base 8) con prefijo `0`: `017` = 15, `010` = 8. MUCHO CUIDADO: un cero a la izquierda convierte tu "decimal" en octal. Escribir `010` queriendo decir diez te da ocho, sin que el

compilador se queje. Este es un bug clásico y silencioso. Si ves un número con cero inicial en código ajeno, ya sabes que es octal a propósito (o un bug a punto de nacer).

Binario (base 2) con prefijo 0b: 0b1010 = 10. Ideal para máscaras de bits legibles, como viste en el ejemplo de permisos.

Separadores de dígitos y notación científica

El apóstrofe separa grupos de dígitos: 1'000'000, 0xFF'FF, 0b1111'0000. El compilador los ignora; existen para tus ojos. Úsalos en constantes grandes: contar ceros a ojo es una fuente de errores embarazosos.

Los flotantes aceptan notación científica: 1.5e3 = 1500.0, 2.5e-4 = 0.00025. La e significa "por diez elevado a". Es el pan de cada día en ciencia e ingeniería, y la verás en la salida de programas cuando los números son muy grandes o muy pequeños (1e+308).

Ejemplo que junta todo: configurando un color RGB como se hace en gráficos reales. Un color se empaca en 32 bits: 8 para alfa (transparencia), 8 para rojo, 8 para verde, 8 para azul. El formato hexadecimal 0xAARRGGBB es el estándar de la industria:


```

#include <iostream>
#include <cstdint> // tipos de ancho exacto: uint32_t, uint8_t

// Un color RGBA empaado en 32 bits, formato 0xAARRGGBB
using Color = std::uint32_t;

constexpr Color ROJO{0xFF'FF00'00u}; // alfa FF, rojo FF, verde 00, azul 00
constexpr Color VERDE{0xFF'00FF'00u};
constexpr Color AZUL{0xFF'0000'FFu};

std::uint8_t componente(Color c, int desplazamiento) {
    return static_cast<std::uint8_t>((c >> desplazamiento) & 0xFFu);
}

int main() {
    Color naranja{0xFF'FF80'00u}; // mucho rojo, medio verde, nada de azul

    std::cout << "Rojo:   " << static_cast<int>(componente(naranja, 16)) <<
std::endl;
    std::cout << "Verde:  " << static_cast<int>(componente(naranja, 8)) <<
std::endl;
    std::cout << "Azul:   " << static_cast<int>(componente(naranja, 0)) <<
std::endl;

    // Aclarar un color: subir cada componente con OR podria saturar mal;
    // la forma simple es sumar con tope usando el ternario
    std::uint8_t r{componente(naranja, 16)};
    std::uint8_t masRojo{static_cast<std::uint8_t>((r + 40 > 255) ? 255 : r + 40)};
    std::cout << "Rojo aclarado: " << static_cast<int>(masRojo) << std::endl;

    // Literales con sufijos: el tipo exacto importa
    auto grande{9'000'000'000LL}; // long long, no cabe en int
    auto preciso{0.1f};           // float, no double
    auto exacto{1'000'000u};       // unsigned int
    std::cout << "sizeof(grande)=" << sizeof(grande) << std::endl;
    std::cout << "sizeof(preciso)=" << sizeof(preciso) << std::endl;
    std::cout << "sizeof(exacto)=" << sizeof(exacto) << std::endl;
    return 0;
}

```

Vamos a lo nuevo: <cstdint> trae tipos de ancho exacto: uint32_t es un entero sin signo de exactamente 32 bits en cualquier plataforma. Para trabajo con bits son obligatorios: int puede medir 32 o 64 bits según el sistema, uint32_t siempre mide 32. El alias using Color = std::uint32_t; documenta la intención: no es "un número", es "un color". Las constantes usan

hexadecimal con separadores: `0xFF'FF80'00u` se lee como cuatro bytes (`FF FF 80 00`) y la `u` final lo hace unsigned para que los desplazamientos sean seguros.

La función `componente` extrae un byte: desplaza el color para llevar el byte deseado a la posición 0 y lo aísla con `& 0xFFu` (máscara que conserva solo los 8 bits bajos). Los desplazamientos son 16 (rojo), 8 (verde) y 0 (azul): el formato `0xAARRGGBB` en acción. Al imprimir un `uint8_t` hay un detalle: `cout` lo trata como carácter (porque es un char disfrazado), así que lo convertimos a `int` con `static_cast` para ver el número. Sin ese cast verías un símbolo raro en vez de 255: trampa clásica.

La aclaración del rojo usa el ternario con tope: si `r + 40` pasa de 255, se queda en 255 (saturación). Sin el tope, el `uint8_t` daría la vuelta a valores pequeños: otro desbordamiento silencioso cazado antes de nacer. Y al final, los sufijos en acción: `9'000'000'000LL` necesita `long long` (no cabe en `int`), `0.1f` es float de verdad (`sizeof` 4, no 8) y `1'000'000u` es unsigned. Los `sizeof` lo confirman: el compilador te obedece al pie de la letra.

Catálogo de errores comunes de esta parte (y cómo evitarlos)

1. Confundir `&` con `&&` (o `|` con `||`): compila, casi funciona, pero la lógica es otra. Si tu condición "a veces falla sin razón", revisa que no hayas usado el operador de bits queriendo el lógico.
2. Olvidar paréntesis en `(x & mascara) == 0`: por precedencia se evalúa como `x & (mascara == 0)`. Pon paréntesis siempre que mezcles bits con comparaciones.
3. Desplazar tipos con signo o desplazar de más: usa unsigned o `uint32_t` para bits, y verifica que el desplazamiento sea menor al tamaño del tipo.
4. Comparar chars con acentos usando aritmética ASCII: 'á' no está en el rango a-z; las operaciones de ± 32 solo valen para ASCII básico. Verifica rangos antes de transformar.
5. Comillas dobles para un carácter: `char c{"A"};` es error. Carácter: simples. Texto: dobles. Sin excepciones.
6. El cero a la izquierda accidental: `int n{010};` guarda 8, no 10. Si el número lleva cero inicial, es octal: quítalo o úsalo a propósito.
7. Comparar flotantes con `==`: `0.1 + 0.2` no es exactamente 0.3 en binario. Usa margen epsilon como viste en la segunda parte.
8. Efectos secundarios dentro de `&&` u `||`: el cortocircuito puede saltarse tu `++` o tu asignación. Haz los efectos en líneas separadas.

- 9: Ternarios anidados de tres niveles: compilan, pero nadie (incluido tú en dos semanas) los entiende. Máximo dos niveles, y solo si son triviales.
10. sizeof con efectos secundarios: sizeof(i++) no incrementa i. Si tu lógica depende de ese incremento, el bug es invisible hasta que lo buscas.
11. Olvidar el cast al imprimir uint8_t: cout muestra el carácter, no el número. Convierte a int primero.
12. Mezclar tipos en el ternario sin querer: (ok) ? 5 : 2.5 promociona a double silenciosamente; si esperabas int, el resultado te sorprende. Sé explícito con los tipos.

Práctica: ejercicios con solución

Ejercicio 1

Escribe un programa que pida al usuario un número entero del 0 al 255 y muestre su representación en binario de 8 bits (como texto de unos y ceros). Luego muestra qué bits están encendidos usando el operador & con máscaras. No uses arreglos ni strings complicados: construye la salida bit por bit.

Solución:

```

#include <iostream>

int main() {
    unsigned int n{0};
    std::cout << "Numero (0-255): ";
    std::cin >> n;
    if (n > 255) {
        std::cout << "Fuera de rango." << std::endl;
        return 1;
    }

    std::cout << "Binario: ";
    for (int i{7}; i >= 0; --i) {
        // Desplaza el bit i a la posicion 0 y aisla con AND 1
        unsigned int bit{(n >> i) & 1u};
        std::cout << bit;
    }
    std::cout << std::endl;

    std::cout << "Bits encendidos en posiciones: ";
    bool primero{true};
    for (int i{0}; i < 8; ++i) {
        unsigned int mascara{1u << i};
        if ((n & mascara) != 0) {
            if (!primero) std::cout << ", ";
            std::cout << i;
            primero = false;
        }
    }
    std::cout << std::endl;

    // Verificacion: reconstruir el numero desde los bits
    unsigned int reconstruido{0};
    for (int i{0}; i < 8; ++i) {
        if ((n & (1u << i)) != 0) {
            reconstruido |= (1u << i);
        }
    }
    std::cout << "Reconstruido: " << reconstruido
        << (reconstruido == n ? " (correcto)" : " (ERROR)") << std::endl;
    return 0;
}

```

Explicación: el primer ciclo recorre las posiciones del 7 al 0 (del bit más significativo al menos) y para cada una desplaza y aísla el bit con $(n \gg i) \& 1$: es el idioma estándar para "dame el bit i". El segundo ciclo hace lo inverso: prueba cada máscara $1 \ll i$ con AND para listar las posiciones encendidas; la variable primero maneja las comas sin dejar una colgando al inicio. La reconstrucción final demuestra que encender esos bits con OR recupera el número original: es la prueba de que entendiste el ciclo completo de bits. Prueba con 173 (10101101) y verifica a mano.

Ejercicio 2

Escribe un programa que lea una línea de texto con `getline` y cuente cuántas vocales, cuántas consonantes, cuántos dígitos y cuántos otros caracteres tiene. Usa la aritmética ASCII para clasificar (rangos 'a'-'z', 'A'-'Z', '0'-'9') y convierte todo a minúsculas para simplificar. Las vocales acentuadas cuéntalas como "otros" y menciónalo en un comentario.

Solución:

```

#include <iostream>
#include <string>

char aMinuscula(char c) {
    if (c >= 'A' && c <= 'Z') {
        return static_cast<char>(c + 32);
    }
    return c;
}

bool esVocal(char c) {
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}

int main() {
    std::string texto;
    std::cout << "Escribe una frase: ";
    std::getline(std::cin, texto);

    int vocales{0}, consonantes{0}, digitos{0}, otros{0};
    for (char c : texto) {
        char m{aMinuscula(c)};
        if (m >= 'a' && m <= 'z') {
            if (esVocal(m)) ++vocales;
            else ++consonantes;
        } else if (m >= '0' && m <= '9') {
            ++digitos;
        } else {
            // Aqui caen espacios, signos y vocales acentuadas (no son ASCII
            basico)
            ++otros;
        }
    }

    std::cout << "Vocales: " << vocales << std::endl;
    std::cout << "Consonantes: " << consonantes << std::endl;
    std::cout << "Digitos: " << digitos << std::endl;
    std::cout << "Otros: " << otros << std::endl;
    return 0;
}

```

Explicación: la estrategia es normalizar primero (todo a minúsculas con +32, válido porque verificamos el rango A-Z) y clasificar después: el código queda lineal y legible. Los rangos 'a'-'z' y '0'-'9' funcionan porque ASCII los tiene consecutivos: es la propiedad que hace posible

este programa sin tablas. Las letras acentuadas caen en "otros" porque en UTF-8 ocupan varios bytes que no están en esos rangos: el comentario lo documenta honestamente en vez de fingir que no existe el problema. Prueba con "Hola Mundo 123!" y verifica: 4 vocales, 5 consonantes, 3 dígitos, 3 otros (dos espacios y el signo).

Ejercicio 3

Implementa una función `std::string clasificarTemperatura(double t)` que devuelva "congelado" si $t < 0$, "frio" si $t < 15$, "templado" si $t < 28$, "caluroso" si $t < 38$ y "extremo" en otro caso. Escríbela DOS veces: una con if-else y otra con ternarios anidados. En main, pide varias temperaturas y muestra ambas versiones. En un comentario final, argumenta cuál versión prefieres y por qué.

Solución:

```

#include <iostream>
#include <string>

std::string conIf(double t) {
    if (t < 0) return "congelado";
    if (t < 15) return "frio";
    if (t < 28) return "templado";
    if (t < 38) return "caluroso";
    return "extremo";
}

std::string conTernario(double t) {
    return (t < 0) ? "congelado"
        : (t < 15) ? "frio"
        : (t < 28) ? "templado"
        : (t < 38) ? "caluroso"
        : "extremo";
}

int main() {
    double t{0};
    std::cout << "Temperaturas (letra para terminar):" << std::endl;
    while (std::cin >> t) {
        std::cout << t << " -> if: " << conIf(t)
            << " | ternario: " << conTernario(t) << std::endl;
    }
    // PREFIERO la version con if: cada condicion esta en su propia linea,
    // se puede depurar poniendo un punto de interrupcion en cada rama,
    // y agregar una sexta categoria no rompe el formato. El ternario
    // anidado funciona, pero con 5 niveles ya cuesta seguir la logica
    // y un error en un umbral es facil de pasar por alto.
    return 0;
}

```

Explicación: las dos versiones son equivalentes lógicamente: los returns tempranos del if hacen lo mismo que la cadena de ternarios (que se agrupa a la derecha). El while (std::cin >> t) lee temperaturas hasta que la entrada falle: es el idioma para "lee todo lo que te den". El comentario final es la parte más importante del ejercicio: no hay respuesta única válida, pero debes tener un criterio. La industria prefiere el if para cadenas largas de categorías, y reserva el ternario para elecciones simples de dos valores. Si tu argumento defiende lo contrario con razones sólidas, también vale: lo que no vale es no tener criterio.

Ejercicio 4

Escribe un programa de "caja de herramientas de bits" con un menú: el usuario elige entre (1) encender un bit, (2) apagar un bit, (3) alternar un bit, (4) consultar un bit de un número unsigned de 32 bits, mostrando siempre el valor en decimal y en binario de 8 bits (asume números de 0 a 255 para simplificar la visualización). Usa las operaciones &, |, ^, ~ con máscaras `1u << posicion`. El menú se repite hasta que el usuario elija salir.

Solución:

```
#include <iostream>
#include <stdint>

void mostrarBinario(std::uint32_t n) {
    for (int i{7}; i >= 0; --i) {
        std::cout << ((n >> i) & 1u) ? '1' : '0';
    }
}

int main() {
    std::uint32_t numero{0};
    int opcion{0};

    while (true) {
        std::cout << "\nNumero actual: " << numero << " (" <<
        mostrarBinario(numero);
        std::cout << ")" << std::endl;
        std::cout << "1) Encender bit  2) Apagar bit  3) Alternar bit\n";
        std::cout << "4) Consultar bit  0) Salir\n";
        std::cout << "Opcion: ";
        std::cin >> opcion;

        if (opcion == 0) break;

        int pos{0};
        if (opcion >= 1 && opcion <= 4) {
            std::cout << "Posicion del bit (0-7): ";
            std::cin >> pos;
            if (pos < 0 || pos > 7) {
                std::cout << "Posicion invalida." << std::endl;
                continue;
            }
        }

        std::uint32_t mascara{1u << pos};
        switch (opcion) {
```

```

        case 1:
            numero = numero | mascara;    // OR enciende
            break;
        case 2:
            numero = numero & ~mascara;    // AND con NOT apaga
            break;
        case 3:
            numero = numero ^ mascara;    // XOR alterna
            break;
        case 4: {
            bool encendido{(numero & mascara) != 0};
            std::cout << "El bit " << pos << " esta "
                << (encendido ? "ENCENDIDO" : "apagado") << std::endl;
            break;
        }
        default:
            std::cout << "Opcion invalida." << std::endl;
    }
}
std::cout << "Valor final: " << numero << std::endl;
return 0;
}

```

Explicación: este es el simulador con el que los bits "se sienten" en las manos. Cada opción aplica una de las cuatro operaciones fundamentales con la máscara $1u \ll pos$: OR para encender, AND con la máscara invertida para apagar, XOR para alternar y AND para consultar. El switch organiza el menú (lo verás a fondo en la unidad 3, pero aquí se entiende por contexto). Nota el bloque con llaves en case 4: declarar la variable encendido dentro del case requiere su propio alcance. Juega encendiendo los bits 0, 2 y 4 (obtendrás $21 = 00010101$), luego alterna el 2 y observa cómo baja a 17. Cuando puedas predecir el resultado antes de verlo, dominas los bits.

Resumen de la expansión

- La lógica booleana tiene tablas de verdad exactas: `&&` exige ambos verdaderos, `||` exige al menos uno, `!` invierte, y el XOR ($A \neq B$) exige diferencia. Las leyes de De Morgan simplifican negaciones: $\neg(A \ \&\& \ B)$ es $\neg A \ || \ \neg B$.
- El cortocircuito de `&&` y `||` es una herramienta (guardas como `p != nullptr && p->x`) y una trampa (efectos secundarios que a veces no ocurren). Nunca pongas `++`, `--` o asignaciones dentro de una condición lógica.

- Un char es un número pequeño con interpretación de texto según ASCII: dígitos consecutivos del 48 al 57 ('0'-'9'), mayúsculas del 65 al 90, minúsculas del 97 al 122, con 32 de diferencia entre cada mayúscula y su minúscula. El truco `c - '0'` convierte dígito a valor.
- `char` a secas puede ser con o sin signo según el compilador: para aritmética usa `signed char` o `unsigned char` explícitos. Comillas simples para caracteres, dobles para texto.
- Los operadores bit a bit trabajan sobre cada bit: `&` apaga (máscara), `|` enciende, `^` alterna, `~` invierte todo. `<<` multiplica por 2 por posición, `>>` divide. Para bits usa tipos sin signo (`uint32_t`) y recuerda los paréntesis con `==`.
- Precedencia: lo peligroso no es lo obvio (`*` antes que `+`), sino `&` por debajo de `==` y `<<` por debajo de `+`. Ante la duda, paréntesis: cuestan cero y salvan programas.
- El ternario `?:` es una expresión (no una instrucción): sirve para elegir valores dentro de inicializaciones y llamadas. Un nivel sí; anidados con moderación; más de dos niveles, mejor `if-else`.
- `sizeof` no evalúa su argumento: mide tipos en tiempo de compilación. `sizeof(arr)/sizeof(arr[0])` cuenta elementos sin repetir el tipo.
- `using` crea alias con nombre significativo: doma tipos largos de la STL, independiza decisiones de tamaño y (con plantillas) llega donde `typedef` no llega.
- Los literales tienen tipo controlable: sufijos `u`, `L`, `LL`, `f`; bases `0x` (hex), `0` (octal, cuidado), `0b` (binario); separadores con apóstrofe `1'000'000`; notación científica `1.5e3`. El tipo exacto evita advertencias y bugs de precisión.
- En la unidad 3 tu programa empezará a tomar decisiones y a repetir acciones: `if`, `switch`, `while` y `for`, donde toda esta lógica booleana y estos operadores se vuelven protagonistas.

UNIDAD 3: Decisiones y repeticiones I

Hasta ahora tus programas son como recetas lineales: hacen lo mismo siempre, de principio a fin. En esta unidad les darás cerebro. Aprenderás a tomar decisiones con `if` y `switch`, y a repetir acciones con `while` y `do-while`. Con estas tres herramientas, tus programas dejarán de ser calculadoras tontas y empezarán a comportarse de forma inteligente: reaccionarán a lo que el usuario haga, validarán datos y procesarán información hasta terminar.

Esta es la unidad donde la programación "hace clic" para la mayoría. Cuando escribas tu primer programa que decide solo qué camino tomar, sentirás que ya no estás siguiendo instrucciones: estás creando comportamiento. Tómate tu tiempo aquí; la lógica condicional y los ciclos son el 80% de todo programa que escribirás en tu vida.

Al terminar sabrás: escribir condiciones con `if`, `else if` y `else`, anidar decisiones sin enredarte, usar `switch` para menús, controlar ciclos `while` y `do-while`, y evitar los dos grandes peligros de los ciclos: los infinitos y los de una vuelta de más o de menos.

Explicación detallada

Clase 7: Condicionales `if`, `else if`, `else`

La instrucción `if` evalúa una condición (una expresión que da `true` o `false`) y ejecuta un bloque solo si es verdadera. La forma básica:

```
if (edad >= 18) {  
    std::cout << "Puedes votar." << std::endl;  
}
```

Si la condición es falsa, el bloque se salta. Para el caso contrario existe `else`:

```
if (edad >= 18) {  
    std::cout << "Puedes votar." << std::endl;  
} else {  
    std::cout << "Aun no puedes votar." << std::endl;  
}
```

Y para múltiples caminos, `else if` encadena condiciones que se evalúan en orden; la primera verdadera gana y las demás se saltan:

```

if (promedio >= 9) {
    std::cout << "Excelente" << std::endl;
} else if (promedio >= 8) {
    std::cout << "Muy bien" << std::endl;
} else if (promedio >= 6) {
    std::cout << "Aprobado" << std::endl;
} else {
    std::cout << "Reprobado" << std::endl;
}

```

El orden importa: las condiciones se prueban de arriba abajo. Si pusieras promedio >= 6 primero, un 9.5 caería ahí y nunca llegaría al "Excelente". Cuando las condiciones son rangos, ordénalas de la más restrictiva a la menos restrictiva, o viceversa, pero con lógica.

Las llaves { } son opcionales si el bloque tiene una sola instrucción, pero los profesionales las ponen siempre. ¿Por qué? Porque el día que agregues una segunda línea al bloque y olvides agregar las llaves, solo la primera línea quedará dentro del if y la segunda se ejecutará siempre. Es un error clásico y silencioso. Llaves siempre: no cuestan nada.

Puedes anidar if dentro de if para decisiones complejas, pero no abuses: más de dos niveles de anidación se vuelve difícil de leer. Cuando la lógica se complica, conviene extraerla a funciones (unidad 4) o replantearla con operadores lógicos.

El error más famoso: usar = en vez de ==. if (x = 5) asigna 5 a x y la condición siempre es verdadera (5 es distinto de cero). El compilador con -Wall te avisa con "suggest parentheses around assignment used as truth value". Otro truco defensivo: escribir la constante primero, if (5 == x), así si escribes = por error el compilador lo detecta de inmediato porque no puedes asignar a un literal. No todos lo usan, pero funciona.

Clase 8: switch para decisiones múltiples

Cuando tienes que comparar una misma variable contra muchos valores constantes, switch es más limpio que una escalera de else if. Su estructura:

```

int opcion;
std::cin >> opcion;
switch (opcion) {
    case 1:
        std::cout << "Elegiste uno" << std::endl;
        break;
    case 2:
        std::cout << "Elegiste dos" << std::endl;
        break;
    default:
        std::cout << "Opcion invalida" << std::endl;
}

```

Cada case es un valor posible. Cuando la variable coincide, se ejecuta ese bloque. El break es crucial: sin él, la ejecución "cae" al siguiente case y ejecuta también su código. Ese comportamiento, llamado fallthrough, casi siempre es un error accidental, aunque a veces se usa a propósito para agrupar casos:

```

switch (dia) {
    case 1: case 2: case 3: case 4: case 5:
        std::cout << "Dia laboral" << std::endl;
        break;
    case 6: case 7:
        std::cout << "Fin de semana" << std::endl;
        break;
    default:
        std::cout << "Dia invalido" << std::endl;
}

```

default atrapa todo lo que no coincidió con ningún case: es tu red de seguridad para valores inesperados. Úsalo siempre.

Limitaciones de switch: solo funciona con tipos enteros y caracteres (int, char y enumeraciones), no con strings ni con rangos. Para comparar textos o rangos, usa if/else. switch brilla en menús numéricos y máquinas de estados simples.

Clase 9: Ciclos while y do-while

Un ciclo repite un bloque mientras una condición sea verdadera. El while revisa la condición antes de cada repetición, incluyendo la primera: si es falsa desde el inicio, el bloque no se ejecuta ni una vez.

```
int n = 1;
while (n <= 5) {
    std::cout << n << " ";
    n++;
}
```

Este ciclo imprime 1 2 3 4 5. Tiene tres partes: inicialización ($n = 1$), condición ($n \leq 5$) y actualización ($n++$). Si olvidas la actualización, la condición nunca cambia y tienes un ciclo infinito: el programa se queda girando para siempre y tienes que detenerlo con Ctrl+C. Todo programador ha creado ciclos infinitos; lo importante es reconocer el síntoma (el programa "se congela") y saber dónde buscar.

El do-while es el primo del while: ejecuta el bloque primero y revisa la condición después. Garantiza al menos una ejecución. Es perfecto para menús que deben mostrarse mínimo una vez:

```
int opcion;
do {
    std::cout << "1. Saludar  2. Despedir  0. Salir: ";
    std::cin >> opcion;
    if (opcion == 1) std::cout << "Hola!" << std::endl;
    if (opcion == 2) std::cout << "Adios!" << std::endl;
} while (opcion != 0);
```

La diferencia entre while y do-while parece pequeña, pero expresa intención: while dice "quizá cero veces", do-while dice "al menos una vez". Elegir el correcto hace tu código más claro.

Dos instrucciones controlan los ciclos desde adentro: break sale del ciclo inmediatamente, continue salta a la siguiente repetición. Úsalos con moderación: un break para salir anticipadamente está bien, pero un ciclo lleno de breaks y continues es difícil de seguir.

El error de "una vuelta de más o de menos" (off-by-one) es legendario. ¿El ciclo debe ser $n < 10$ o $n \leq 10$? ¿Empieza en 0 o en 1? La forma de no equivocarte: prueba mentalmente con los extremos. Si n empieza en 1 y la condición es $n \leq 5$, las vueltas son 1,2,3,4,5: cinco vueltas. Haz esta verificación siempre.

Buena práctica: dale a tus ciclos condiciones de salida claras y evita modificar la variable de control en lugares inesperados. Un ciclo cuyo control está en un solo lugar es un ciclo que puedes entender de un vistazo.

Ejemplos paso a paso

Ejemplo 1: Clasificador de calificaciones

```
#include <iostream>

int main() {
    double calif;
    std::cout << "Calificacion (0-10): ";
    std::cin >> calif;

    if (calif < 0 || calif > 10) {
        std::cout << "Calificacion invalida." << std::endl;
    } else if (calif >= 9) {
        std::cout << "Excelente. ¡Felicidades!" << std::endl;
    } else if (calif >= 8) {
        std::cout << "Muy bien." << std::endl;
    } else if (calif >= 6) {
        std::cout << "Aprobado." << std::endl;
    } else {
        std::cout << "Reprobado. A estudiar mas." << std::endl;
    }
    return 0;
}
```

Observa el orden: primero validamos el rango, luego vamos de la condición más exigente a la menos. Cada else if solo se evalúa si los anteriores fallaron, así que no hace falta repetir límites.

Ejemplo 2: Menú con switch y do-while


```

#include <iostream>

int main() {
    int opcion;
    do {
        std::cout << "\n--- MENU ---" << std::endl;
        std::cout << "1. Saludar" << std::endl;
        std::cout << "2. Contar hasta 5" << std::endl;
        std::cout << "0. Salir" << std::endl;
        std::cout << "Elige: ";
        std::cin >> opcion;

        switch (opcion) {
            case 1:
                std::cout << "Hola, programador." << std::endl;
                break;
            case 2: {
                int i = 1;
                while (i <= 5) {
                    std::cout << i << " ";
                    i++;
                }
                std::cout << std::endl;
                break;
            }
            case 0:
                std::cout << "Adios." << std::endl;
                break;
            default:
                std::cout << "Opcion invalida." << std::endl;
        }
    } while (opcion != 0);
    return 0;
}

```

Este ejemplo combina todo: do-while para el menú, switch para las opciones, while para contar. Fíjate en las llaves extra del case 2: se ponen para declarar variables dentro de un case sin conflictos. Es un truco estándar que verás en código profesional.

Ejemplo 3: Adivina el número

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int main() {
    std::srand(std::time(0));
    int secreto = std::rand() % 100 + 1;
    int intento = 0;
    int intentos = 0;

    std::cout << "Adivina el numero (1-100)." << std::endl;
    while (intento != secreto) {
        std::cout << "Tu intento: ";
        std::cin >> intento;
        intentos++;
        if (intento < secreto) {
            std::cout << "Mas alto..." << std::endl;
        } else if (intento > secreto) {
            std::cout << "Mas bajo..." << std::endl;
        }
    }
    std::cout << "¡Correcto! Lo lograste en " << intentos << " intentos." <<
std::endl;
    return 0;
}
```

`srand(time(0))` inicializa el generador aleatorio con la hora actual para que el número cambie cada vez. `rand() % 100` da 0-99, más 1 da 1-100. El `while` sigue hasta acertar, y los `if` dan pistas. Es un juego completo con lo aprendido hasta aquí.

Ejemplo 4: Validación de entrada con `do-while`

```
#include <iostream>

int main() {
    int edad;
    do {
        std::cout << "Escribe tu edad (0-120): ";
        std::cin >> edad;
        if (edad < 0 || edad > 120) {
            std::cout << "Edad invalida, intenta de nuevo." << std::endl;
        }
    } while (edad < 0 || edad > 120);
    std::cout << "Edad registrada: " << edad << std::endl;
    return 0;
}
```

Este patrón de "pedir hasta que sea válido" aparece en todos los programas serios. El do-while garantiza que se pregunte al menos una vez, y la condición repite mientras el dato sea inválido. Memorízalo: lo usarás toda la vida.

Práctica

Ejercicio 1

Pide tres números y muestra cuál es el mayor usando if/else if/else.

Solución:

```
#include <iostream>

int main() {
    double a, b, c;
    std::cout << "Tres numeros: ";
    std::cin >> a >> b >> c;

    double mayor = a;
    if (b > mayor) mayor = b;
    if (c > mayor) mayor = c;

    std::cout << "El mayor es: " << mayor << std::endl;
    return 0;
}
```

Explicación: en vez de una escalera complicada de condiciones, suponemos que a es el mayor y lo reemplazamos si encontramos algo más grande. Es más simple y funciona con cualquier cantidad de números. Este patrón de "supón y mejora" es fundamental.

Ejercicio 2

Escribe un programa que pida un año y diga si es bisiesto. Un año es bisiesto si es divisible entre 4, excepto los divisibles entre 100, salvo que también sean divisibles entre 400.

Solución:

```
#include <iostream>

int main() {
    int anio;
    std::cout << "Año: ";
    std::cin >> anio;

    bool bisiesto = (anio % 4 == 0 && anio % 100 != 0) || (anio % 400 == 0);

    if (bisiesto) {
        std::cout << anio << " es bisiesto." << std::endl;
    } else {
        std::cout << anio << " no es bisiesto." << std::endl;
    }
    return 0;
}
```

Explicación: la regla cabe en una sola expresión booleana. Los paréntesis dejan clara la lógica: (divisible entre 4 y no entre 100) o (divisible entre 400). Guardarla en bisiesto hace el if legible. Prueba con 2024 (sí), 1900 (no) y 2000 (sí).

Ejercicio 3

Con un while, muestra la tabla de multiplicar del número que pida el usuario, del 1 al 10.

Solución:

```
#include <iostream>

int main() {
    int n;
    std::cout << "Numero: ";
    std::cin >> n;

    int i = 1;
    while (i <= 10) {
        std::cout << n << " x " << i << " = " << n * i << std::endl;
        i++;
    }
    return 0;
}
```

Explicación: el ciclo va de 1 a 10 con la condición $i \leq 10$. No olvides `i++`: sin él tendrías un ciclo infinito mostrando "`n x 1`" para siempre.

Ejercicio 4

Pide números al usuario hasta que escriba 0, y al final muestra cuántos números positivos ingresó y la suma de todos (sin contar el 0). Usa `while`.

Solución:

```

#include <iostream>

int main() {
    int numero;
    int positivos = 0;
    int suma = 0;

    std::cout << "Escribe numeros (0 para terminar):" << std::endl;
    std::cin >> numero;
    while (numero != 0) {
        suma += numero;
        if (numero > 0) positivos++;
        std::cin >> numero;
    }

    std::cout << "Numeros positivos: " << positivos << std::endl;
    std::cout << "Suma total: " << suma << std::endl;
    return 0;
}

```

Explicación: este es el patrón de "acumulador": inicializas contadores en cero y los vas actualizando dentro del ciclo. Nota que leemos antes del ciclo y al final de cada vuelta: así el 0 que termina no se procesa. Es el patrón estándar para procesar datos hasta una señal de fin.

Ejercicio 5

Crea una calculadora con menú (do-while + switch): opciones 1=sumar, 2=restar, 3=multiplicar, 4=dividir, 0=salir. En la división, valida que el divisor no sea cero.

Solución:

```

#include <iostream>

int main() {
    int opcion;
    do {
        std::cout << "\n1.Sumar 2.Restar 3.Multiplicar 4.Dividir 0.Salir: ";
        std::cin >> opcion;

        if (opcion >= 1 && opcion <= 4) {
            double a, b;
            std::cout << "Dos numeros: ";
            std::cin >> a >> b;
            switch (opcion) {
                case 1: std::cout << "Resultado: " << a + b << std::endl; break;
                case 2: std::cout << "Resultado: " << a - b << std::endl; break;
                case 3: std::cout << "Resultado: " << a * b << std::endl; break;
                case 4:
                    if (b != 0) {
                        std::cout << "Resultado: " << a / b << std::endl;
                    } else {
                        std::cout << "Error: no se puede dividir entre cero." <<
std::endl;
                    }
                    break;
            }
        } else if (opcion != 0) {
            std::cout << "Opcion invalida." << std::endl;
        }
    } while (opcion != 0);
    std::cout << "Adios." << std::endl;
    return 0;
}

```

Explicación: combina menú (do-while), selección (switch), validación (if del divisor) y lectura condicional de datos (solo pide números si la opción es válida). Es un programa completo y útil: guárdalo como referencia.

Resumen de la unidad

- if/else if/else elige caminos según condiciones; pon llaves siempre y ordena los rangos con lógica.
- == compara, = asigna: confundirlos es el error más clásico de la programación.

- switch compara una variable entera contra valores constantes; no olvides break ni default.
- while revisa antes de cada vuelta (puede ejecutarse cero veces); do-while ejecuta al menos una vez.
- Todo ciclo necesita inicialización, condición de salida y actualización: si falta una, hay ciclo infinito.
- Patrones clave: acumulador, pedir-hasta-válido, menú con do-while + switch.
- En la unidad 4 dominarás el ciclo for y aprenderás a organizar código en funciones reutilizables.

Profundización: lógica, patrones de ciclos y máquinas de estado

Ya sabes decidir y repetir. Ahora vamos a pensar como ingeniero: leyes lógicas para simplificar condiciones, patrones de ciclos que resuelven familias enteras de problemas, y el switch usado como máquina de estados, una técnica profesional para programas que "recuerdan" en qué paso van.

Leyes de De Morgan: simplifica tus negaciones

Dos leyes que todo programador debe saber de memoria:

- $\neg(A \ \&\& \ B)$ equivale a $(\neg A \ || \ \neg B)$
- $\neg(A \ || \ B)$ equivale a $(\neg A \ \&\& \ \neg B)$

En palabras: "no (esto y aquello)" es lo mismo que "no esto o no aquello". Sirven para simplificar condiciones negadas, que son las más difíciles de leer:

```
// Difícil de leer:
if (!(edad >= 18 && tieneINE)) { ... }

// Equivalente y más claro con De Morgan:
if (edad < 18 || !tieneINE) { ... }
```

La segunda versión dice directamente cuándo se rechaza: menor de edad o sin INE. Acostúmbrate a transformar negaciones complejas: tu yo del futuro te lo agradecerá.

Cortocircuito: úsalo a tu favor

En $A \ \&\& \ B$, si A es falso, B no se evalúa. En $A \ || \ B$, si A es verdadero, B no se evalúa. Esto no es solo optimización: es una herramienta de seguridad:


```
if (indice >= 0 && indice < tamano && datos[indice] > 0) { ... }
```

Si índice es inválido, las comparaciones siguientes no se ejecutan y nunca accedes fuera del arreglo. El orden importa: pon primero las condiciones baratas y las que protegen a las demás. Es un patrón defensivo estándar.

Pero cuidado con el reverso: no pongas efectos importantes en la parte que podría no evaluarse. `if (listo && guardar())` es peligroso si guardar debe ejecutarse siempre.

Condiciones anidadas vs. retornos tempranos

Los if anidados profundos son difíciles de leer:

```
if (usuarioValido) {  
    if (tienePermiso) {  
        if (saldoSuficiente) {  
            procesar();  
        }  
    }  
}
```

La técnica de "retorno temprano" o "cláusulas guarda" aplanan la lógica:

```
if (!usuarioValido) return;  
if (!tienePermiso) return;  
if (!saldoSuficiente) return;  
procesar();
```

Cada guarda elimina un caso inválido y el código principal queda al final, limpio. Esta técnica es estándar en código profesional.

Patrones de ciclos

Casi todo ciclo que escribirás pertenece a una de estas familias:

Contador: repite N veces. `for (int i = 0; i < n; i++)`.

Acumulador: suma o combina valores. Inicializa en neutro (0 para suma, 1 para producto) y actualiza en cada vuelta.

Búsqueda: recorre hasta encontrar. Usa un bool encontrado y break, o una función que retorne al encontrar.

Centinela: procesa hasta un valor especial (como el 0 del ejercicio 4). Lee antes del ciclo y al final de cada vuelta.

Validación: repite hasta dato válido (do-while). Ya lo viste en la unidad 3.

Filtros: recorre todo y procesa solo lo que cumple una condición (if dentro del for).

Reconocer el patrón antes de escribir acelera todo: en vez de inventar la lógica, adaptas el molde.

Switch como máquina de estados

Un programa con "modos" (menú principal, juego corriendo, pausa, fin) se modela como máquina de estados: una variable guarda el estado actual y un switch decide qué hacer:

```

#include <iostream>

enum class Estado { Menu, Jugando, Pausa, Fin };

int main() {
    Estado estado = Estado::Menu;
    int opcion;
    int puntos = 0;

    while (estado != Estado::Fin) {
        switch (estado) {
            case Estado::Menu:
                std::cout << "1.Jugar 0.Salir: ";
                std::cin >> opcion;
                estado = (opcion == 1) ? Estado::Jugando : Estado::Fin;
                break;

            case Estado::Jugando:
                puntos += 10;
                std::cout << "Puntos: " << puntos << " (1.seguir 2.pausa): ";
                std::cin >> opcion;
                estado = (opcion == 2) ? Estado::Pausa : Estado::Jugando;
                if (puntos >= 30) estado = Estado::Fin;
                break;

            case Estado::Pausa:
                std::cout << "Pausa. 1.continuar: ";
                std::cin >> opcion;
                estado = Estado::Jugando;
                break;

            case Estado::Fin:
                break;
        }
    }
    std::cout << "Fin del juego. Puntos: " << puntos << std::endl;
    return 0;
}

```

Cada estado es un case, las transiciones cambian la variable. Los videojuegos, los protocolos de red y los asistentes telefónicos funcionan así. Es una de las arquitecturas más útiles que aprenderás.

Depurando ciclos

Técnica 1: imprime la variable de control. Si el ciclo se porta raro, un cerr dentro mostrando i revela si avanza como esperas.

Técnica 2: verifica los extremos. Para un for, calcula mentalmente la primera y última vuelta. Para un while, pregúntate: "¿qué pasa si la condición es falsa desde el inicio?" y "¿qué la hace volverse falsa?".

Técnica 3: el ciclo infinito controlado. Si sospechas un infinito, agrega un contador de seguridad temporal:

```
int seguridad = 0;
while (condicion && seguridad < 10000) {
    // ...
    seguridad++;
}
```

Si termina por seguridad, confirmaste el infinito sin colgar tu máquina.

Más ejemplos

Ejemplo: tabla de verdad de operadores lógicos

```
#include <iostream>

int main() {
    bool valores[] = {false, true};
    std::cout << "A B | A&&B A||B !A" << std::endl;
    for (bool a : valores) {
        for (bool b : valores) {
            std::cout << a << " " << b << " | "
                << (a && b) << "    "
                << (a || b) << "    "
                << (!a) << std::endl;
        }
    }
    return 0;
}
```

Generar la tabla de verdad con código es la mejor forma de interiorizar la lógica. Guárdala como referencia.

Ejemplo: número primo con optimizaciones

```
#include <iostream>
#include <cmath>

bool esPrimo(int n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;
    int limite = static_cast<int>(std::sqrt(n));
    for (int i = 3; i <= limite; i += 2) {
        if (n % i == 0) return false;
    }
    return true;
}

int main() {
    int contador = 0;
    for (int n = 2; n <= 1000; n++) {
        if (esPrimo(n)) contador++;
    }
    std::cout << "Primos hasta 1000: " << contador << std::endl;
    return 0;
}
```

Optimizaciones: descartar pares de un golpe, probar solo impares, llegar solo hasta la raíz cuadrada. De probar n divisores a probar $\sqrt{n}/2$: para $n=1,000,000$ pasas de un millón de pruebas a 500. Así piensan los algoritmos eficientes.

Práctica adicional

Ejercicio A

Simplifica con De Morgan: $!(x > 0 \ \&\& \ y > 0)$. Escribe un programa que verifique con varios valores que ambas formas dan lo mismo.

Solución:

```
#include <iostream>

int main() {
    int pruebas[][2] = {{5, 3}, {-1, 3}, {5, -2}, {0, 0}};
    for (auto& p : pruebas) {
        int x = p[0], y = p[1];
        bool original = !(x > 0 && y > 0);
        bool simplificada = (x <= 0 || y <= 0);
        std::cout << x << ", " << y << ": " << original
                    << " vs " << simplificada
                    << (original == simplificada ? " OK" : " FALLO") << std::endl;
    }
    return 0;
}
```

Explicación: probar equivalencias lógicas con código es la forma científica de validarlas. Las cuatro combinaciones cubren todos los casos.

Ejercicio B

Reescribe el clasificador de calificaciones de la unidad 3 usando cláusulas guarda en una función `string clasificar(double c)`.

Solución:

```
#include <iostream>
#include <string>

std::string clasificar(double c) {
    if (c < 0 || c > 10) return "Invalida";
    if (c >= 9) return "Excelente";
    if (c >= 8) return "Muy bien";
    if (c >= 6) return "Aprobado";
    return "Reprobado";
}

int main() {
    for (double c : {4.5, 6.0, 8.3, 9.7, 11.0}) {
        std::cout << c << " -> " << clasificar(c) << std::endl;
    }
    return 0;
}
```

Explicación: cada return temprano elimina un caso; no hay anidación. La función se lee de arriba abajo como una lista de reglas. Mucho más limpio que la escalera original.

Ejercicio C

Escribe un programa que lea números hasta encontrar el primer múltiplo de 7, usando el patrón de búsqueda con break. Muestra cuántos números se leyeron.

Solución:

```
#include <iostream>

int main() {
    int numero, leidos = 0;
    bool encontrado = false;
    std::cout << "Escribe numeros:" << std::endl;
    while (!encontrado) {
        std::cin >> numero;
        leidos++;
        if (numero % 7 == 0) {
            encontrado = true;
        }
    }
    std::cout << "Multiplo de 7 encontrado: " << numero
        << " despues de leer " << leidos << " numeros." << std::endl;
    return 0;
}
```

Explicación: el bool encontrado es la bandera del patrón de búsqueda. El ciclo termina en cuanto se cumple la condición, sin procesar de más.

Ejercicio D

Implementa una máquina de estados para un semáforo: estados Verde, Amarillo, Rojo que cambian cada vez que el usuario presiona Enter (línea vacía). Termina cuando escriba "fin".

Solución:

```

#include <iostream>
#include <string>

enum class Semaforo { Verde, Amarillo, Rojo };

int main() {
    Semaforo estado = Semaforo::Rojo;
    std::string linea;
    std::cout << "Enter para cambiar, 'fin' para terminar." << std::endl;
    while (true) {
        std::getline(std::cin, linea);
        if (linea == "fin") break;
        switch (estado) {
            case Semaforo::Rojo:
                estado = Semaforo::Verde;
                std::cout << "VERDE: avance" << std::endl;
                break;
            case Semaforo::Verde:
                estado = Semaforo::Amarillo;
                std::cout << "AMARILLO: precaucion" << std::endl;
                break;
            case Semaforo::Amarillo:
                estado = Semaforo::Rojo;
                std::cout << "ROJO: alto" << std::endl;
                break;
        }
    }
    return 0;
}

```

Explicación: el semáforo real funciona exactamente así: un estado, transiciones fijas, eventos que las disparan. Este modelo escala a sistemas enormes.

Expansion: trampas, validacion y juegos completos

Ya dominas if, switch, while y do-while. Ahora viene la parte donde dejas de ser estudiante y empiezas a programar como alguien que ha visto programas romperse en produccion. Esta expansion te ensena las trampas que el lenguaje te pone (y como esquivarlas), como validar la entrada del usuario de forma que ningun dato raro tumbe tu programa, como armar menus con varios niveles, y cierra con dos juegos completos que juntan todo lo aprendido. Al terminar, tus programas aguantaran usuarios reales: esos que escriben "hola" donde va un numero y presionan Enter como si no hubiera manana.

Comparaciones encadenadas: la trampa que parece obvia

En matematicas escribes $0 < x < 10$ y significa "x esta entre 0 y 10". En C++ esa expresion también compila sin quejarse... y casi siempre esta mal. Es una de las trampas mas traicioneras del lenguaje porque el codigo se ve perfecto y el compilador no te avisa.

Por que `if (0 < x < 10)` no hace lo que crees

C++ no evalua `0 < x < 10` como una comparacion triple. La evalua de izquierda a derecha, en dos pasos separados:

1. Primero calcula `0 < x`, que da `true` o `false`.
2. Después compara ese resultado booleano contra 10.

Y aqui esta el truco sucio: cuando un `bool` participa en una comparacion numerica, `true` se convierte en 1 y `false` en 0. Asi que la expresion se transforma en `1 < 10` (verdadero) o `0 < 10` (verdadero). Da igual cuanto valga x: el resultado siempre es verdadero. Tu programa miente con cara seria.

Mira este programa de demostracion. Leelo con calma antes de compilarlo, trata de predecir que imprimira, y luego compruebalo:

```
#include <iostream>

int main() {
    int x = 500;

    if (0 < x && x < 10) {
        std::cout << "Version correcta: x esta entre 0 y 10" << std::endl;
    } else {
        std::cout << "Version correcta: x NO esta entre 0 y 10" << std::endl;
    }

    if (0 < x < 10) {
        std::cout << "Version encadenada: dice que SI esta (mentira)" << std::endl;
    } else {
        std::cout << "Version encadenada: dice que no esta" << std::endl;
    }
    return 0;
}
```

Con x valiendo 500, la version correcta dice que NO esta en el rango. La encadenada dice que si. El error es silencioso: ningun aviso, ningun error de compilacion, solo un programa que toma decisiones equivocadas.

La regla de oro: en C++ los rangos siempre se escriben con dos comparaciones unidas por `&&`. Para "x entre 0 y 10, sin incluir extremos": `x > 0 && x < 10`. Para incluirlos: `x >= 0 && x <= 10`. Memoriza este patron; lo usaras en validaciones toda tu vida.

La misma trampa aparece con el igual encadenado. `if (a == b == c)` no pregunta "los tres son iguales". Evalua `(a == b) == c`: primero da true o false, y luego compara ese booleano contra c. Si c vale 1, la condicion es verdadera siempre que `a == b`. Para comparar tres valores necesitas `a == b && b == c`.

Comparar numeros con punto flotante: otra trampa silenciosa

Cambia de tema pero no de familia: nunca compares `double` con `==`. Los numeros decimales se guardan en binario con precision limitada, y `0.1 + 0.2` no es exactamente 0.3 en la memoria de la computadora. Comprueballo:

```
#include <iostream>
#include <cmath>

int main() {
    double a = 0.1 + 0.2;
    double b = 0.3;

    std::cout << "a == b: " << (a == b ? "true" : "false") << std::endl;

    const double EPS = 1e-9; // tolerancia: una milmillonesima
    bool casiIguales = std::fabs(a - b) < EPS;
    std::cout << "Casi iguales: " << (casiIguales ? "true" : "false") << std::endl;
    return 0;
}
```

La primera comparacion da false aunque en tu calculadora $0.1 + 0.2 = 0.3$. La tecnica profesional es comparar la diferencia absoluta contra una tolerancia pequena (epsilon): si estan "suficientemente cerca", se consideran iguales. Para dinero, de todos modos, prefiere enteros (centavos) en vez de double: es la practica estandar en sistemas financieros.

switch con enum class: decisiones con nombre

En la primera parte usaste enum class para los estados de una maquina. Ahora veras su otro gran uso con switch: reemplazar los "numeros magicos" por nombres con significado, y de paso poner al compilador a trabajar para ti.

Del numero magico al nombre con significado

Un numero magico es un valor numerico regado por el codigo cuyo significado solo vive en tu cabeza: ¿que era 3? ¿Error de red? ¿Nivel dificil? ¿La opcion de salir? Con enum class cada valor tiene nombre:

```
enum class Dificultad { Facil, Media, Dificil, Experto };  
enum class Operacion { Sumar, Restar, Multiplicar, Dividir };
```

Un switch sobre enum class se lee solo, y si alguien lee `case Operacion::Dividir` no necesita adivinar nada. Pero la ventaja real es otra: el compilador puede verificar que no olvidaste ningun caso.

El compilador como tu guardaespaldas

Compila con `-Wall -Wextra` (cosa que ya deberias hacer siempre) y observa: si tu switch sobre un enum class no cubre todos los valores, el compilador te avisa con algo como "enumeration value 'Experto' not handled in switch". Con un switch sobre int eso es imposible: el compilador no sabe cuantos valores "validos" hay. Esta es una red de seguridad gratis: si manana agregas un valor al enum, el compilador te senala cada switch que necesita actualizarse.

Ejemplo completo: una calculadora que despacha por operacion nombrada. Fijate en tres detalles: cada case termina en return (asi no necesitas break), no hay default a proposito (queremos que el compilador avise si falta un caso), y al final hay un return de emergencia para calmar al compilador:

```

#include <iostream>

enum class Operacion { Sumar, Restar, Multiplicar, Dividir };

double calcular(double a, double b, Operacion op) {
    switch (op) {
        case Operacion::Sumar:
            return a + b;
        case Operacion::Restar:
            return a - b;
        case Operacion::Multiplicar:
            return a * b;
        case Operacion::Dividir:
            if (b == 0) {
                std::cout << "Error: division entre cero." << std::endl;
                return 0;
            }
            return a / b;
    }
    return 0; // nunca deberia llegar aqui; solo para el compilador
}

std::string nombreOperacion(Operacion op) {
    switch (op) {
        case Operacion::Sumar:      return "suma";
        case Operacion::Restar:     return "resta";
        case Operacion::Multiplicar: return "multiplicacion";
        case Operacion::Dividir:    return "division";
    }
    return "?";
}

int main() {
    double a = 10, b = 4;
    Operacion ops[] = {Operacion::Sumar, Operacion::Restar,
                       Operacion::Multiplicar, Operacion::Dividir};
    for (Operacion op : ops) {
        std::cout << nombreOperacion(op) << ": "
                  << calcular(a, b, op) << std::endl;
    }
    return 0;
}

```

Si agregas ``Operacion::Potencia`` al enum y recompilas con `-Wall`, el compilador te marcara los dos switch. Eso, en un proyecto de miles de lineas, vale oro: convierte olvidos silenciosos en errores visibles.

Cuando si usar default con enum class: cuando el enum representa datos que vienen de fuera (un archivo, la red) y podrian traer valores inesperados, o cuando deliberadamente quieres ignorar algunos casos. En logica interna donde controlas todos los valores, omitir default y dejar que el compilador vigile es la jugada profesional.

El operador coma en el for

El encabezado del for tiene tres huecos: inicializacion, condicion y actualizacion. ¿Y si necesitas dos variables de control que avanzan juntas? Para eso existe el operador coma: evalua varias expresiones de izquierda a derecha, separadas por comas. Su valor final es el de la ultima expresion, pero lo importante aqui es el efecto: puedes inicializar y actualizar dos variables en un solo for.

```
#include <iostream>

int main() {
    // i sube de 2 en 2, j baja de 3 en 3, se encuentran en el medio
    for (int i = 0, j = 30; i <= j; i += 2, j -= 3) {
        std::cout << "i = " << i << ", j = " << j << std::endl;
    }
    return 0;
}
```

Lee el encabezado por partes: se crean `i = 0` y `j = 30`; mientras `i <= j` se ejecuta el cuerpo; después de cada vuelta, `i += 2` y `j -= 3`. Las dos variables viven solo dentro del ciclo, como cualquier variable declarada en el for.

Otro uso clasico: recorrer una secuencia desde ambos extremos a la vez, o llevar un contador de vueltas junto con la variable principal:

```
#include <iostream>

int main() {
    int vuelta = 0;
    for (int n = 100, v = 0; n > 0; n /= 2, v++) {
        std::cout << "Vuelta " << v << ": n = " << n << std::endl;
    }
    return 0;
}
```

Aquí `n` se divide entre 2 cada vuelta (100, 50, 25, 12, 6, 3, 1) y `v` cuenta las vueltas. Sin el operador coma necesitarías declarar `v` fuera o incrementarla dentro del cuerpo.

Advertencia honesta de profesional: el operador coma es una herramienta de precisión, no un juguete. Si las dos variables no están intimamente relacionadas, es más legible declarar una fuera del `for` o actualizarla dentro del cuerpo. Un `for` cuyo encabezado parece acertijo es un `for` que alguien (probablemente tú, en tres meses) maldecirá al leer. Úsalo cuando las variables bailan juntas, como los dos extremos que se acercan; evítalo cuando solo quieres ahorrarte una línea.

Ciclos infinitos a propósito

Toda la unidad te advirtió contra los ciclos infinitos accidentales. Ahora la vuelta de tuerca: los profesionales escriben ciclos infinitos a propósito. Un servidor web es, en esencia, un ``while (true)`` que atiende peticiones hasta que lo apagan. El bucle principal de un videojuego es un ciclo infinito que dibuja, lee controles y actualiza el mundo sesenta veces por segundo hasta que sales. La diferencia entre un bug y una arquitectura es una sola cosa: la salida diseñada.

Anatomía de un infinito bien hecho

Un ciclo infinito intencional tiene tres ingredientes:

1. La condición eterna: ``while (true)`` o ``for (;)`` (ambos significan "para siempre"; elige el que te parezca más legible).
2. El trabajo repetido: lo que el programa hace en cada vuelta.
3. Al menos una salida diseñada: un `break`, un `return` o una llamada a `exit()` ante una condición clara.

Sin el ingrediente 3 tienes un bug. Con él, tienes un servidor.

Mira el esqueleto de un servidor simplificado. No es código para compilar tal cual (la red real necesita bibliotecas especiales), pero la estructura es exactamente la de un servidor de verdad:

```
while (true) {
    Cliente c = esperarCliente(); // se queda aquí hasta que alguien se conecta
    if (servidorDebeApagarse()) break;
    atender(c); // procesa la petición y responde
}
```

El programa nunca "termina" por sí solo: vive para servir. La única salida es la condición de apagado. Así funcionan los programas que deben estar siempre disponibles.

Ejemplo completo: bucle principal de un minijuego

Este sí compila y juega. Es el patrón "game loop" en miniatura: un `while(true)` que muestra estado, lee un comando, actualiza el mundo y revisa condiciones de salida:

```
#include <iostream>
#include <string>

int main() {
    int salud = 100;
    int pociones = 2;
    std::string comando;

    std::cout << "=== MAZMORRA MINIMA ===" << std::endl;
    std::cout << "Comandos: atacar, pocion, huir" << std::endl;

    while (true) { // bucle principal: corre hasta una salida diseñada
        std::cout << "\nSalud: " << salud << " | Pociones: " << pociones
            << " | Tu comando: ";
        std::cin >> comando;

        if (comando == "huir") {
            std::cout << "Escapas de la mazmorra. Fin del juego." << std::endl;
            break; // salida diseñada 1: el jugador decide irse
        } else if (comando == "atacar") {
            salud -= 15;
            std::cout << "Golpeas al troll, pero te alcanza con su garrote." <<
std::endl;
        } else if (comando == "pocion") {
            if (pociones > 0) {
                pociones--;
            }
        }
    }
}
```

```

        salud += 25;
        if (salud > 100) salud = 100;
        std::cout << "Bebes una pocion. Salud: " << salud << std::endl;
    } else {
        std::cout << "No te quedan pociones." << std::endl;
    }
    continue; // el troll no ataca si solo bebiste
} else {
    std::cout << "Comando desconocido." << std::endl;
    continue;
}

if (salud <= 0) {
    std::cout << "El troll te derroto. Fin del juego." << std::endl;
    break; // salida disenada 2: condicion del mundo
}
std::cout << "El troll ruge y se prepara..." << std::endl;
}
return 0;
}

```

Observa el continue: cuando el comando fue "pocion" o invalido, salta directo a la siguiente vuelta sin pasar por el "rugido" del troll ni revisar la derrota. Es un uso legitimo y legible de continue: evita anidar todo el resto del turno dentro de un else gigante.

Disciplina obligatoria para infinitos intencionales:

- Cada while(true) debe tener su salida visible y comentada. Si tu no la ves a simple vista, no existe.
- En programas reales de consola, Ctrl+C siempre puede interrumpir. No dependas de que el usuario "salga bien": si tu ciclo escribe archivos, guarda el progreso cada cierto numero de vueltas, no solo al final.
- Un infinito que hace trabajo pesado sin pausas puede comerse el CPU. Los juegos reales duermen unos milisegundos por vuelta; los servidores se bloquean esperando clientes. Un while(true) vacio o sin espera es casi siempre un error.

break y continue en ciclos anidados: salir de varios niveles

Aqui hay un detalle que tarde o temprano te mordera: `break` solo sale del ciclo mas interno, el que lo contiene directamente. Si tienes un for dentro de otro for y llamas a break en el interno, el externo sigue girando como si nada. Lo mismo con continue: solo afecta a su propio nivel.

El problema, en código

Imagina que buscas el primer par (i, j) con i y j entre 1 y 10 cuyo producto supere un límite. El intento ingenuo falla de forma sutil:

```
#include <iostream>

int main() {
    int limite = 50;
    int pi = 0, pj = 0;
    for (int i = 1; i <= 10; i++) {
        for (int j = 1; j <= 10; j++) {
            if (i * j > limite) {
                pi = i;
                pj = j;
                break; // sale solo del for de j...
            }
        }
        // ...pero el for de i sigue con la siguiente i. El resultado
        // final sera el ultimo par encontrado, no el primero.
    }
    std::cout << "Par: (" << pi << ", " << pj << ")" << std::endl;
    return 0;
}
```

El break rompe el ciclo de j, pero el de i continua y el par se sobrescribe en cada vuelta externa. El programa "funciona" sin errores y da un resultado equivocado: el peor tipo de bug.

Alternativa 1: bandera booleana

La tecnica clasica: una variable que senala "ya termine" y que ambos ciclos revisan:

```

#include <iostream>

int main() {
    int limite = 50;
    int pi = 0, pj = 0;
    bool encontrado = false;

    for (int i = 1; i <= 10 && !encontrado; i++) {
        for (int j = 1; j <= 10 && !encontrado; j++) {
            if (i * j > limite) {
                pi = i;
                pj = j;
                encontrado = true; // el break sale del interno;
            } // la condicion frena al externo
        }
    }
    std::cout << "Primer par: (" << pi << ", " << pj << ")" << std::endl;
    return 0;
}

```

Funciona y es explicita, pero ensucia las condiciones de los ciclos con una preocupacion que no es suya. Con dos niveles se tolera; con tres se vuelve un enredo.

Alternativa 2: extraer a una funcion y usar return (la mas limpia)

Esta es la tecnica que usan los profesionales. Mete los ciclos anidados en una funcion: un return sale de todos los niveles de un golpe, y de paso la busqueda queda con nombre propio y reutilizable:

```

#include <iostream>

// Busca el primer par (i, j) con i*j > limite.
// Devuelve true y deja el par en pi, pj si lo encuentra.
bool buscarPar(int limite, int& pi, int& pj) {
    for (int i = 1; i <= 10; i++) {
        for (int j = 1; j <= 10; j++) {
            if (i * j > limite) {
                pi = i;
                pj = j;
                return true; // sale de AMBOS ciclos de un golpe
            }
        }
    }
    return false; // recorrio todo sin encontrar
}

int main() {
    int pi, pj;
    if (buscarPar(50, pi, pj)) {
        std::cout << "Primer par: (" << pi << ", " << pj << ")" << std::endl;
    } else {
        std::cout << "No hay par que supere el limite." << std::endl;
    }
    return 0;
}

```

El return temprano hace el trabajo de "break multinivel" sin trucos. Como beneficio extra, el main queda limpio y la funcion se puede probar por separado. Cuando sientas la tentacion de un break que salte dos niveles, casi siempre la respuesta correcta es extraer una funcion.

Alternativa 3: reestructurar con un while

A veces el problema se disuelve cambiando la estructura: un solo while cuya condicion combine ambos niveles. Es menos comun, pero cuando el patron encaja el codigo queda elegante. La regla general: si el anidamiento existe solo para buscar algo, la funcion con return gana; si los niveles representan dimensiones reales del problema (filas y columnas de una tabla), la bandera o la funcion son validas según el caso.

El tabu del goto, explicado con honestidad

C++ tiene ``goto``, que salta a una etiqueta en cualquier parte de la función, incluyendo fuera de ciclos anidados. Los libros viejos lo usaban justo para este problema. Los profesionales modernos casi nunca lo usan, y no por capricho: un goto rompe el flujo estructurado y convierte el código en una madeja donde cada salto es una sorpresa. La función con `return` temprano logra exactamente lo mismo (salir de varios niveles) sin sacrificar la legibilidad. Conoce que goto existe para que no te asuste si lo ves en código ajeno; no lo uses en código propio.

Validación robusta de entrada numérica: a prueba de usuarios reales

Hasta ahora tus programas suponen un usuario educado que escribe números donde se piden números. Los usuarios reales escriben "hola", dejan la línea vacía, escriben "25abc" o presionan Ctrl+D. Un programa serio no se cae ni se cicla ante eso: lo detecta, avisa y vuelve a pedir. Esta sección te da el protocolo completo.

Anatomía del desastre

Que pasa exactamente cuando haces ``std::cin >> edad;`` y el usuario escribe "veinte":

1. La extracción falla porque "veinte" no es un entero. La variable `edad` conserva su valor anterior (basura si no la inicializaste).
2. `cin` enciende su bandera de error (failbit): "algo salió mal".
3. El texto "veinte" se queda atorado en el buffer de entrada, sin consumir.
4. Mientras la bandera siga encendida, TODAS las lecturas futuras fallan de inmediato sin siquiera intentarlo.

Si ese `cin` está dentro de un ciclo de validación, tienes un ciclo infinito que imprime "edad inválida" a toda velocidad sin dejarte escribir nada. Es el bug más común de los programas de consola para principiantes, y se ve así cuando lo provocas a propósito:

```
#include <iostream>

int main() {
    int edad = 0;
    std::cout << "Escribe tu edad: ";
    std::cin >> edad; // escribe "hola" y observa
    std::cout << "Leído: " << edad << std::endl;

    int otro = 0;
    std::cout << "Escribe otro numero: ";
    std::cin >> otro; // ni siquiera te deja escribir: falla al instante
    std::cout << "Leído: " << otro << std::endl;
    return 0;
}
```

Ejecuta este programa, escribe "hola" en la primera pregunta y mira como la segunda lectura ni siquiera espera tu teclado. El buffer atorado y la bandera encendida son los culpables.

El protocolo de rescate: `clear()` e `ignore()`

Para recuperarte necesitas dos pasos, siempre juntos y en este orden:

1. `std::cin.clear();` apaga la bandera de error. Sin esto, cin seguira fallando.
2. `std::cin.ignore(n, '\n');` tira a la basura lo que quedo atorado en el buffer, hasta el fin de linea. Sin esto, la proxima lectura encontrara la misma basura y fallara otra vez.

Para n se usa `std::numeric_limits<std::streamsize>::max()`, que significa "tira todo lo que haya hasta el delimitador", con `#include <limits>`. Es el hechizo estandar; memorízalo tal cual.

Funcion reutilizable: `leerEnteroEnRango`

Con el protocolo claro, lo encapsulamos en una funcion que pide un entero dentro de un rango y no regresa hasta tenerlo. A partir de ahora, ningun programa tuyo volvera a romperse por una entrada mala:

```

#include <iostream>
#include <limits>
#include <string>

// Pide un entero entre minimo y maximo. No regresa hasta obtenerlo.
int leerEnteroEnRango(const std::string& mensaje, int minimo, int maximo) {
    int valor;
    while (true) {
        std::cout << mensaje << " (" << minimo << "-" << maximo << "): ";
        if (std::cin >> valor) {
            // La lectura funciona. Limpia el resto de la linea
            // (por si escribio "25abc": el 25 sirve, lo demas se tira).
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            if (valor >= minimo && valor <= maximo) {
                return valor; // dato bueno: terminamos
            }
            std::cout << "Fuera de rango. Intenta de nuevo." << std::endl;
        } else {
            std::cout << "Eso no es un numero entero. Intenta de nuevo." <<
std::endl;
            std::cin.clear(); // apaga la bandera de error
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); //
tira la linea mala
        }
    }
}

int main() {
    int edad = leerEnteroEnRango("Escribe tu edad", 0, 120);
    int calif = leerEnteroEnRango("Escribe tu calificacion", 0, 10);

    std::cout << "\nDatos registrados: edad " << edad
        << ", calificacion " << calif << std::endl;

    if (edad >= 18) {
        std::cout << "Eres mayor de edad." << std::endl;
    }
    if (calif >= 6) {
        std::cout << "Aprobado." << std::endl;
    } else {
        std::cout << "Reprobado." << std::endl;
    }
    return 0;
}

```

Prueba este programa escribiendo letras, numeros fuera de rango, "30 tacos", lineas vacias: siempre se recupera y sigue pidiendo. Esa es la diferencia entre un ejercicio y una herramienta.

Tres notas finas que te ahorraran horas de depuracion:

- Después de una lectura exitosa con `>>`, el Enter que el usuario presiono queda en el buffer. Si mas adelante usas `getline` para leer texto, ese `getline` leera una linea vacia. La llamada a `ignore()` después de cada lectura numerica previene este problema clasico.
- ``while (true)`` con `return` adentro es el patron ideal para "repetir hasta lograrlo": la salida esta en el dato valido, no en una condicion del encabezado.
- Valida el tipo primero y el rango después. Son dos errores distintos ("no es numero" vs "esta fuera de rango") y merecen mensajes distintos: el usuario corrige mas rapido cuando sabe exactamente que hizo mal.

Con esta funcion en tu caja de herramientas, los juegos de las siguientes secciones pueden pedir datos sin miedo.

Menus multinivel: programas que se sienten profesionales

Un menu de un solo nivel ya lo dominas. Los programas reales tienen jerarquias: menu principal -> submenu -> accion, con la opcion de regresar. La tecnica es simple: cada submenu es una funcion con su propio `do-while`, y la opcion 0 significa "regresar al nivel anterior" en vez de "salir del programa". Solo el menu principal usa 0 para salir.

Reglas de navegacion

1. Cada nivel es una funcion: ``menuPrincipal()``, ``submenuReportes()``, etc. Asi cada menu se prueba y se lee por separado.
2. La opcion 0 siempre es "regresar" (o "salir" solo en el principal). El usuario nunca debe sentirse atrapado.
3. Muestra donde esta: un encabezado como `"--- VENTAS > REPORTES ---"` orienta mas de lo que crees.
4. No anides `switch` dentro de `switch` en una sola funcion gigante: con tres niveles se vuelve ilegible. Funciones, siempre funciones.

Ejemplo completo: sistema de la tiendita

Un mini sistema con menu principal (Ventas, Inventario, Reportes, Salir) y un submenu de Reportes. Las ventas e inventario son sencillas pero funcionales: llevan totales reales. Fijate como cada submenu es una funcion independiente y como el flujo de navegacion queda cristalino:

```
#include <iostream>
#include <limits>

// Limpia el buffer tras leer un numero con >>
void limpiarLinea() {
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}

int leerOpcion(int min, int max) {
    int op;
    while (true) {
        std::cout << "Elige (" << min << "-" << max << "): ";
        if (std::cin >> op && op >= min && op <= max) {
            limpiarLinea();
            return op;
        }
        std::cout << "Opcion invalida." << std::endl;
        std::cin.clear();
        limpiarLinea();
    }
}

void submenuReportes(double totalVentas, int totalPiezas) {
    int op;
    do {
        std::cout << "\n--- REPORTES ---" << std::endl;
        std::cout << "1. Total vendido" << std::endl;
        std::cout << "2. Piezas vendidas" << std::endl;
        std::cout << "3. Ticket promedio" << std::endl;
        std::cout << "0. Regresar" << std::endl;
        op = leerOpcion(0, 3);
        switch (op) {
            case 1:
                std::cout << "Total vendido: $" << totalVentas << std::endl;
                break;
            case 2:
                std::cout << "Piezas vendidas: " << totalPiezas << std::endl;
                break;
        }
    } while (op != 0);
}
```



```

        case 3:
            if (totalPiezas > 0) {
                std::cout << "Ticket promedio: $"
                    << totalVentas / totalPiezas << std::endl;
            } else {
                std::cout << "Aun no hay ventas." << std::endl;
            }
            break;
    }
} while (op != 0); // 0 = regresar, no salir
}

int main() {
    double totalVentas = 0;
    int totalPiezas = 0;
    int op;

    do {
        std::cout << "\n===== LA TIENDITA =====" << std::endl;
        std::cout << "1. Registrar venta" << std::endl;
        std::cout << "2. Ver inventario vendido" << std::endl;
        std::cout << "3. Reportes" << std::endl;
        std::cout << "0. Salir" << std::endl;
        op = leerOpcion(0, 3);

        switch (op) {
            case 1: {
                double monto;
                std::cout << "Monto de la venta: $";
                while (!(std::cin >> monto) || monto < 0) {
                    std::cout << "Monto invalido. Otra vez: $";
                    std::cin.clear();
                    limpiarLinea();
                }
                limpiarLinea();
                totalVentas += monto;
                totalPiezas++;
                std::cout << "Venta registrada." << std::endl;
                break;
            }
            case 2:
                std::cout << "Piezas vendidas: " << totalPiezas
                    << " | Total: $" << totalVentas << std::endl;
                break;
            case 3:

```

```

        submenuReportes(totalVentas, totalPiezas); // entrar al submenu
        break; // al regresar, el menu principal se muestra solo
    case 0:
        std::cout << "Cerrando caja. ¡Hasta luego!" << std::endl;
        break;
    }
} while (op != 0);
return 0;
}

```

El truco arquitectonico: `submenuReportes` recibe los datos que necesita como parametros y tiene su propio ciclo. Cuando el usuario elige 0, la funcion termina con return implicito y el menu principal continua su do-while como si nada. Agregar un tercer nivel (por ejemplo, "Reportes > Por fecha") seria otra funcion llamada desde el submenu: la estructura escala sin enredarse.

Nota las llaves en `case 1:`: declaran monto dentro del case sin contaminar los demas. Ya viste este truco en la primera parte; aqui es indispensable porque el menu real declara variables en varios casos.

Juego completo 1: piedra, papel o tijeras

Hora de juntar todo: enum class con switch, aleatoriedad, validacion robusta, ciclos y funciones. Este juego es pequeno pero completo: la computadora elige al azar, se juega al mejor de 5 rondas, lleva marcador y permite revancha. Es el tipico programa que puedes presumirle a alguien no programador.

Diseno antes del codigo

Todo juego se disena en tres preguntas: ¿cual es el estado? (marcadores, ronda actual), ¿cuales son las reglas? (piedra vence a tijeras, etc.), ¿cuando termina? (alguien llega a 3 victorias, y luego opcionalmente la revancha). Con eso claro, el codigo se escribe solo:

- `enum class Jugada { Piedra, Papel, Tijeras };` para jugadas con nombre.
- `Jugada jugadaComputadora()` genera 1-3 al azar.
- `int definirGanador(Jugada j, Jugada c)` devuelve 0 (empate), 1 (gana jugador) o 2 (gana computadora) usando switch sobre enum class.
- El main lleva el marcador con un for de rondas y un do-while para la revancha.

```

#include <iostream>
#include <cstdlib>

```

```

#include <ctime>
#include <string>

enum class Jugada { Piedra, Papel, Tijeras };

// Convierte 1/2/3 a Jugada. Supone n valido (se valida antes de llamar).
Jugada jugadaDe(int n) {
    switch (n) {
        case 1: return Jugada::Piedra;
        case 2: return Jugada::Papel;
        default: return Jugada::Tijeras;
    }
}

std::string nombre(Jugada j) {
    switch (j) {
        case Jugada::Piedra: return "Piedra";
        case Jugada::Papel: return "Papel";
        case Jugada::Tijeras: return "Tijeras";
    }
    return "?";
}

Jugada jugadaComputadora() {
    return jugadaDe(std::rand() % 3 + 1); // 1, 2 o 3 al azar
}

// 0 = empate, 1 = gana el jugador, 2 = gana la computadora
int definirGanador(Jugada jugador, Jugada compu) {
    if (jugador == compu) return 0;
    switch (jugador) {
        case Jugada::Piedra:
            return (compu == Jugada::Tijeras) ? 1 : 2;
        case Jugada::Papel:
            return (compu == Jugada::Piedra) ? 1 : 2;
        case Jugada::Tijeras:
            return (compu == Jugada::Papel) ? 1 : 2;
    }
    return 0;
}

// Pide 1/2/3 con validacion robusta (no acepta letras ni fuera de rango)
int pedirJugada() {
    int n;
    while (true) {

```

```

        std::cout << "Tu jugada (1=Piedra 2=Papel 3=Tijeras): ";
        if (std::cin >> n && n >= 1 && n <= 3) {
            return n;
        }
        std::cout << "Jugada invalida. " << std::endl;
        std::cin.clear();
        std::cin.ignore(1000, '\n');
    }
}

int main() {
    std::srand(std::time(0));
    char revancha;

    do {
        int ganadas = 0, perdidas = 0;
        int ronda = 1;

        std::cout << "\n=== PIEDRA, PAPEL O TIJERAS (mejor de 5) ===" << std::endl;
        while (ganadas < 3 && perdidas < 3) {
            std::cout << "\n--- Ronda " << ronda << " ---" << std::endl;
            Jugada jugador = jugadaDe(pedirJugada());
            Jugada compu = jugadaComputadora();

            std::cout << "Tu: " << nombre(jugador)
                << " vs Compu: " << nombre(compu) << std::endl;

            int g = definirGanador(jugador, compu);
            if (g == 0) {
                std::cout << "Empate." << std::endl;
            } else if (g == 1) {
                std::cout << "¡Ganaste la ronda!" << std::endl;
                ganadas++;
            } else {
                std::cout << "La computadora gana la ronda." << std::endl;
                perdidas++;
            }
            std::cout << "Marcador: tu " << ganadas << " - " << perdidas << " compu"
<< std::endl;
            ronda++;
        }

        if (ganadas == 3) {
            std::cout << "\n¡CAMPEON! Ganaste el duelo." << std::endl;
        } else {

```

```

        std::cout << "\nLa computadora gana el duelo. Practica mas." << std::endl;
    }

    std::cout << "¿Revancha? (s/n): ";
    std::cin >> revancha;
} while (revancha == 's' || revancha == 'S');

std::cout << "Gracias por jugar." << std::endl;
return 0;
}

```

Lecciones del diseno:

- El `while (ganadas < 3 && perdidas < 3)` termina en cuanto alguien llega a 3, aunque no se hayan jugado las 5 rondas: si vas 3-0, el duelo ya se decidió. Es el patrón "jugar hasta que haya ganador".
- Los empates no cuentan para nadie pero si consumen ronda: el marcador solo avanza con victorias, así el duelo siempre tiene un ganador.
- `definirGanador` concentra TODA la regla del juego en un solo lugar. Si mañana agregas "lagarto" y "Spock", solo tocas esa función.
- La validación en `pedirJugada` usa el protocolo de rescate: el juego nunca se cicla aunque el usuario escriba "piedra" con letras.

Juego completo 2: batalla naval simplificada

El plato fuerte: un juego con tablero visible, estado que evoluciona con cada jugada y condiciones de victoria y derrota. La versión completa de batalla naval usa una cuadrícula de dos dimensiones; aquí la simplificamos a una sola fila de 10 casillas con un barco de 3 casillas contiguas escondido. Toda la diversión táctica (buscar, acorralar, hundir) cabe en una dimensión, y el código se mantiene legible.

Diseno

- Tablero: arreglo de 10 caracteres. '.' = sin disparar, 'O' = agua (fallo), 'X' = impacto.
- Barco: 3 casillas contiguas; su posición inicial se elige al azar entre 0 y 7 (para que quepa: $7 + 3 = 10$).
- El jugador tiene 8 disparos. Gana si logra 3 impactos; pierde si se le acaban.
- Cada turno: mostrar el tablero numerado, pedir casilla (validada: 0-9 y no repetida), actualizar, revisar fin.

El arreglo de caracteres es un adelanto de la unidad 5, pero aquí lo usamos de forma simple: una lista de 10 cajitas que vamos marcando. Piensa en él como 10 variables seguidas.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

const int TAM = 10;
const int LARGO_BARCO = 3;
const int MAX_DISPAROS = 8;

void mostrarTablero(const char tablero[]) {
    std::cout << "\n ";
    for (int i = 0; i < TAM; i++) std::cout << i << " ";
    std::cout << "\n ";
    for (int i = 0; i < TAM; i++) std::cout << tablero[i] << " ";
    std::cout << "\n . = sin disparar    0 = agua    X = ¡impacto!" << std::endl;
}

// ¿La casilla pos pertenece al barco que empieza en inicio?
bool esBarco(int pos, int inicio) {
    return pos >= inicio && pos < inicio + LARGO_BARCO;
}

int pedirDisparo(const char tablero[]) {
    int pos;
    while (true) {
        std::cout << "Dispara a la casilla (0-9): ";
        if (std::cin >> pos && pos >= 0 && pos < TAM) {
            if (tablero[pos] == '.') return pos;
            std::cout << "¡Ahí ya disparaste. Elige otra." << std::endl;
        } else {
            std::cout << "Casilla inválida." << std::endl;
            std::cin.clear();
            std::cin.ignore(1000, '\n');
        }
    }
}

int main() {
    std::srand(std::time(0));
    int inicioBarco = std::rand() % (TAM - LARGO_BARCO + 1); // 0 a 7

    char tablero[TAM];
    for (int i = 0; i < TAM; i++) tablero[i] = '.';
}
```

```

int impactos = 0;
int disparos = 0;

std::cout << "=== BATALLA NAVAL ===" << std::endl;
std::cout << "Hay un barco de " << LARGO_BARCO
    << " casillas escondido. Tienes " << MAX_DISPAROS
    << " disparos." << std::endl;

while (disparos < MAX_DISPAROS && impactos < LARGO_BARCO) {
    mostrarTablero(tablero);
    std::cout << "Disparos restantes: " << (MAX_DISPAROS - disparos)
        << " | Impactos: " << impactos << "/" << LARGO_BARCO << std::endl;

    int pos = pedirDisparo(tablero);
    disparos++;

    if (esBarco(pos, inicioBarco)) {
        tablero[pos] = 'X';
        impactos++;
        std::cout << "¡IMPACTO!" << std::endl;
        if (impactos == LARGO_BARCO) {
            std::cout << "¡BARCO HUNDIDO!" << std::endl;
        }
    } else {
        tablero[pos] = '0';
        std::cout << "Agua..." << std::endl;
    }
}

mostrarTablero(tablero);
if (impactos == LARGO_BARCO) {
    std::cout << "\n¡VICTORIA! Hundiste el barco en " << disparos
        << " disparos." << std::endl;
} else {
    std::cout << "\n¡DERROTA. El barco estaba en las casillas "
        << inicioBarco << ", " << (inicioBarco + 1) << ", "
        << (inicioBarco + 2) << "." << std::endl;
}
return 0;
}

```

Detalles de diseño que vale la pena notar:

- `inicioBarco = rand() % (TAM - LARGO_BARCO + 1)` da 0 a 7: el barco siempre cabe completo. Este "+ 1" es el ajuste off-by-one clasico; sin el, el barco podria salirse del tablero.
- La condicion del while combina las dos formas de terminar: sin disparos o barco hundido. Es el patron "jugar hasta que haya desenlace".
- `pedirDisparo` valida dos cosas: rango y no repeticion. Rechazar disparos repetidos no es amabilidad: es correccion, porque un disparo repetido gastaria un turno sin informacion nueva.
- Al perder, el programa revela la posicion del barco. Todo juego de azar/deducion debe mostrar la respuesta al final; si no, el jugador no aprende ni confia.

Prueba jugarlo varias veces: notarás que la estrategia importa. Disparar al centro primero divide el tablero, y tras un impacto conviene probar las casillas vecinas. Sin darte cuenta, estas ejecutando un algoritmo de busqueda.

Catalogo de errores comunes

1. Comparacion encadenada: `if (0 < x < 10)` siempre es verdadera. Escribe `x > 0 && x < 10`.
2. Comparar doubles con `==`: `if (a == b)` con decimales casi siempre falla. Compara `std::fabs(a - b) < 1e-9`.
3. Igual encadenado: `if (a == b == c)` no compara los tres. Escribe `a == b && b == c`.
4. switch sobre enum class sin cubrir todos los casos: compila pero un valor cae al vacio. Compila con -Wall y atiende los avisos, o agrega default consciente.
5. Creer que break sale de todos los ciclos anidados: solo sale del mas interno. Para salir de varios, extrae una funcion y usa return.
6. while(true) sin salida disenada: revisa que cada infinito tenga su break o return visible. Si no lo ves, es un bug.
7. Validacion sin protocolo completo: si pones `cin.clear()` pero olvidas `ignore()`, la basura sigue en el buffer y la siguiente lectura falla igual. Siempre los dos, en ese orden.
8. Pedir un numero y luego un texto con getline sin limpiar el Enter: el getline lee la linea vacia. Un `ignore()` tras cada lectura numerica lo evita.
9. `rand() % n` sin `srand(time(0))`: el juego usa la misma secuencia cada vez que lo corres. Inicializa una sola vez al inicio del main.

10. Off-by-one en posiciones aleatorias: ``rand() % 10`` para un barco de 3 que empieza en 8 se sale del tablero. El limite correcto es ``rand() % (TAM - LARGO + 1)``.
11. Variable de control del for usada fuera del for: ``for (int i = ...)`` muere al terminar el ciclo. Si necesitas el valor después, declara la variable antes.
12. continue dentro de do-while que salta la actualizacion: si actualizas la variable de control al final del cuerpo y un continue la salta, el ciclo no avanza. Revisa donde queda la actualizacion.

Practica

Ejercicio 1: el detector de mentiras

Este programa siempre dice "edad laboral", incluso con edad 5 o 90. Explica por que y corrígelo.

```
#include <iostream>
int main() {
    int edad = 5;
    if (18 <= edad <= 65) {
        std::cout << "Edad laboral" << std::endl;
    } else {
        std::cout << "No laboral" << std::endl;
    }
    return 0;
}
```

Solucion:

```
#include <iostream>
int main() {
    int edad = 5;
    if (edad >= 18 && edad <= 65) {
        std::cout << "Edad laboral" << std::endl;
    } else {
        std::cout << "No laboral" << std::endl;
    }
    return 0;
}
```

Explicacion: `18 <= edad <= 65` se evalua como `(18 <= edad) <= 65`. Con edad 5, `18 <= 5` es false, que vale 0, y `0 <= 65` es verdadero. Con edad 90, `18 <= 90` es true, que vale 1, y `1 <= 65` también es verdadero. Siempre verdadero, sin importar la edad. La correccion usa dos comparaciones unidas con `&&`, el patron de rangos de esta unidad.

Ejercicio 2: semaforo con enum class

Define `enum class Luz { Rojo, Amarillo, Verde };` y escribe la funcion `std::string accion(Luz l)` que devuelva "Alto", "Precaucion" o "Avance" usando switch. Prueba las tres luces en main. Compila con -Wall.

Solucion:

```
#include <iostream>
#include <string>

enum class Luz { Rojo, Amarillo, Verde };

std::string accion(Luz l) {
    switch (l) {
        case Luz::Rojo:    return "Alto";
        case Luz::Amarillo: return "Precaucion";
        case Luz::Verde:   return "Avance";
    }
    return "?";
}

int main() {
    Luz luces[] = {Luz::Rojo, Luz::Amarillo, Luz::Verde};
    for (Luz l : luces) {
        std::cout << accion(l) << std::endl;
    }
    return 0;
}
```

Explicacion: cada case devuelve directo, sin breaks. Si agregaras `Luz::Intermitente` al enum, el compilador con -Wall te avisaria que el switch quedo incompleto: esa es la red de seguridad que los numeros magicos no te dan.

Ejercicio 3: promedio blindado

Pide las calificaciones de 5 materias (0 a 10) usando una funcion de lectura validada como la de esta unidad, y muestra el promedio. El programa no debe romperse con letras ni con numeros fuera de rango.

Solucion:

```
#include <iostream>
#include <limits>
#include <string>

int leerEnRango(const std::string& msg, int min, int max) {
    int v;
    while (true) {
        std::cout << msg << " (" << min << "-" << max << "): ";
        if (std::cin >> v && v >= min && v <= max) {
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            return v;
        }
        std::cout << "Dato invalido, intenta de nuevo." << std::endl;
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    }
}

int main() {
    int suma = 0;
    for (int i = 1; i <= 5; i++) {
        suma += leerEnRango("Calificacion " + std::to_string(i), 0, 10);
    }
    std::cout << "Promedio: " << suma / 5.0 << std::endl;
    return 0;
}
```

Explicacion: la funcion encapsula el protocolo de rescate completo (clear + ignore) y la validacion de rango. El main queda limpio: un for que acumula. Observa `suma / 5.0`: dividir entre 5.0 (double) evita la division entera. Esta funcion de lectura te servira en todos tus programas de consola de ahora en adelante.

Ejercicio 4: búsqueda limpia sin arreglos

Encuentra los primeros enteros i, j entre 1 y 20 tales que $i*i + j*j > 200$. Hazlo con una función que use return temprano para salir de los dos ciclos, e imprime el par y su suma de cuadrados.

Solucion:

```
#include <iostream>

bool buscar(int& pi, int& pj) {
    for (int i = 1; i <= 20; i++) {
        for (int j = 1; j <= 20; j++) {
            if (i * i + j * j > 200) {
                pi = i;
                pj = j;
                return true;
            }
        }
    }
    return false;
}

int main() {
    int i, j;
    if (buscar(i, j)) {
        std::cout << "Par: (" << i << ", " << j << ")"
                  << " suma de cuadrados: " << i * i + j * j << std::endl;
    }
    return 0;
}
```

Explicación: el return dentro del ciclo interno abandona la función completa, así no necesitas banderas ni breaks dobles. Los parámetros por referencia devuelven el par encontrado (técnica que veras a fondo en la unidad 4). El primer par que cumple es (1, 15): $1 + 225 = 226 > 200$.

Ejercicio 5: batalla naval con dos barcos

Modifica el juego de batalla naval para esconder DOS barcos: uno de 3 casillas y uno de 2, sin traslaparse, en un tablero de 12 casillas. El jugador tiene 10 disparos y gana al hundir ambos.

Solución (cambios clave sobre la versión original):

```
#include <iostream>
```

```

#include <cstdlib>
#include <ctime>

const int TAM = 12;
const int MAX_DISPAROS = 10;

// Barcos: inicio y largo de cada uno
const int INICIOS_MAX = 2;

bool esBarco(int pos, const int inicios[], const int largos[], int n) {
    for (int b = 0; b < n; b++) {
        if (pos >= inicios[b] && pos < inicios[b] + largos[b]) return true;
    }
    return false;
}

bool seTraslapa(int ini1, int len1, int ini2, int len2) {
    return ini1 < ini2 + len2 && ini2 < ini1 + len1;
}

int main() {
    std::srand(std::time(0));
    const int largos[INICIOS_MAX] = {3, 2};
    int inicios[INICIOS_MAX];

    // Coloca el primero al azar
    inicios[0] = std::rand() % (TAM - largos[0] + 1);
    // El segundo: reintenta hasta no traslaparse
    do {
        inicios[1] = std::rand() % (TAM - largos[1] + 1);
    } while (seTraslapa(inicios[0], largos[0], inicios[1], largos[1]));

    char tablero[TAM];
    for (int i = 0; i < TAM; i++) tablero[i] = '.';

    int impactos = 0, disparos = 0;
    const int TOTAL_CASILLAS = largos[0] + largos[1];

    while (disparos < MAX_DISPAROS && impactos < TOTAL_CASILLAS) {
        std::cout << "\n ";
        for (int i = 0; i < TAM; i++) std::cout << i % 10 << " ";
        std::cout << "\n ";
        for (int i = 0; i < TAM; i++) std::cout << tablero[i] << " ";
        std::cout << "\nDisparos restantes: " << (MAX_DISPAROS - disparos) <<
std::endl;
    }
}

```

```

        int pos;
        while (true) {
            std::cout << "Dispara (0-11): ";
            if (std::cin >> pos && pos >= 0 && pos < TAM && tablero[pos] == '.')
                break;

            std::cout << "Casilla invalida o repetida." << std::endl;
            if (std::cin.fail()) {
                std::cin.clear();
                std::cin.ignore(1000, '\n');
            }
        }
        disparos++;

        if (esBarco(pos, inicios, largos, INICIOS_MAX)) {
            tablero[pos] = 'X';
            impactos++;
            std::cout << "¡IMPACTO!" << std::endl;
        } else {
            tablero[pos] = '0';
            std::cout << "Agua..." << std::endl;
        }
    }

    if (impactos == TOTAL_CASILLAS) {
        std::cout << "\n¡VICTORIA! Hundiste los dos barcos." << std::endl;
    } else {
        std::cout << "\nDERROTA. Los barcos empezaban en "
            << inicios[0] << " y " << inicios[1] << "." << std::endl;
    }
    return 0;
}

```

Explicacion: la idea nueva es el do-while de colocacion: genera una posicion para el segundo barco y repite mientras se traslape con el primero. `seTraslapa` usa la prueba estandar de interseccion de intervalos. El resto del juego no cambia porque `esBarco` ahora revisa una lista de barcos en vez de uno solo: disenar con funciones pequeñas hace que extender el juego sea barato.

Resumen de la expansion

- Los rangos se escriben con &&: `x > 0 && x < 10`. Las comparaciones encadenadas siempre mienten y los doubles nunca se comparan con == (usa epsilon).

- switch sobre enum class te da nombres legibles y avisos del compilador si olvidas un caso: compila siempre con -Wall.
- El operador coma permite dos variables de control en un for; usalo solo cuando bailan juntas.
- while(true) es arquitectura legitima (servidores, game loops) siempre que tenga salidas disenadas y visibles.
- break solo rompe el ciclo interno: para salir de varios niveles, extrae una funcion y usa return temprano.
- La validacion robusta sigue el protocolo cin.clear() + ignore(): encapsulalo en una funcion reutilizable y olvidate del problema para siempre.
- Cada nivel de menu es una funcion con su propio ciclo; 0 significa regresar, solo el principal sale.
- Ya armaste dos juegos completos: piedra-papel-tijeras y batalla naval. Todo programa interactivo es un ciclo, un estado y reglas: ya sabes construir los tres.

UNIDAD 4: Bucles for, funciones y sobrecarga

En la unidad 3 aprendiste a repetir con while. Ahora conocerás el ciclo for, el favorito de los programadores para repeticiones contadas, y darás un salto más grande: las funciones. Las funciones te permiten partir un programa grande en piezas pequeñas, cada una con una tarea clara. Es la diferencia entre un cuarto lleno de herramientas tiradas y una caja de herramientas organizada.

Y como cereza del pastel: la sobrecarga de funciones, tu primera probada del polimorfismo de C++. Varias funciones con el mismo nombre, cada una haciendo lo correcto según lo que reciba. A partir de aquí tu código empezará a verse y sentirse profesional.

Al terminar sabrás: usar el ciclo for en todas sus formas, declarar y definir funciones, pasar parámetros por valor, devolver resultados con return, entender el alcance de las variables, y crear funciones sobrecargadas.

Explicación detallada

Clase 10: El ciclo for

El for es un while con las tres partes del control reunidas en una sola línea: inicialización, condición y actualización.

```
for (int i = 1; i <= 5; i++) {  
    std::cout << i << " ";  
}
```

Lee así: "crea i valiendo 1; mientras i sea menor o igual a 5, ejecuta el bloque; después de cada vuelta, incrementa i". Todo el control del ciclo está visible en el encabezado, por eso el for es ideal cuando sabes cuántas vueltas darás.

La variable declarada en el for (int i) solo existe dentro del ciclo: al terminar, desaparece. Esto se llama alcance y es una ventaja: no contaminas el resto del programa con variables temporales. Si necesitas el valor después, declárala antes del for.

El for moderno de C++ tiene una joya: el for de rango, que recorre colecciones sin índices:


```
int numeros[] = {10, 20, 30, 40};
for (int n : numeros) {
    std::cout << n << " ";
}
```

Se lee "para cada n en numeros". No hay índices, no hay riesgo de pasarte del final, no hay errores off-by-one. Cuando veas arreglos y vectores en las unidades 5 y 8, este for será tu mejor amigo.

Puedes controlar el paso: for (int i = 10; i >= 1; i--) cuenta hacia atrás; for (int i = 0; i < 100; i += 10) va de 10 en 10. Y puedes anidar ciclos: un for dentro de otro sirve para tablas, matrices y patrones. El ciclo interno se completa entero por cada vuelta del externo.

¿Cuándo usar for y cuándo while? Regla práctica: si sabes cuántas vueltas darás (recorrer 10 elementos, repetir 5 veces), usa for. Si depende de una condición que cambia (leer hasta encontrar un 0, esperar una respuesta válida), usa while. No es ley, pero hace el código más expresivo.

Clase 11: Funciones

Una función es un bloque de código con nombre que hace una tarea y puede devolver un resultado. Su anatomía:

```
tipo_de_retorno nombre(tipo1 param1, tipo2 param2) {
    // cuerpo
    return valor;
}
```

Ejemplo:

```
double promedio(double a, double b) {
    return (a + b) / 2.0;
}
```

Para usarla, la "llamas": double p = promedio(8.5, 9.0);. Los valores que envías se llaman argumentos; las variables que los reciben se llaman parámetros. En C++ los parámetros se pasan por valor por defecto: la función recibe una copia. Si la función modifica el parámetro, la variable original no se entera. Esto es seguro pero tiene un costo con datos grandes, que resolveremos con referencias en la unidad 6.

Toda función debe declararse antes de usarse. Tienes dos opciones: definirla completa antes de main, o poner un prototipo arriba y la definición abajo:

```
double promedio(double a, double b); // prototipo

int main() {
    std::cout << promedio(8.5, 9.0) << std::endl;
    return 0;
}

double promedio(double a, double b) { // definición
    return (a + b) / 2.0;
}
```

El prototipo le dice al compilador "existe una función así, confía en mí". Los profesionales separan prototipos (en archivos .h) de definiciones (en archivos .cpp): es la base de los proyectos grandes.

return termina la función y entrega el resultado. Una función puede tener varios return, pero el primero que se ejecute gana. Las funciones que no devuelven nada usan void:

```
void saludar(std::string nombre) {
    std::cout << "Hola, " << nombre << std::endl;
}
```

El alcance (scope) define dónde existe cada variable. Las variables declaradas dentro de una función son locales: nacen al entrar y mueren al salir. Las globales (fuera de toda función) existen siempre, pero son mala práctica: cualquier función puede cambiarlas y los errores se vuelven imposibles de rastrear. Prefiere pasar datos como parámetros.

Principio de diseño: cada función debe hacer una sola cosa y hacerla bien. Una función de 200 líneas que hace cinco cosas es una mala función; cinco funciones de 40 líneas con nombres claros son un buen programa. Cuando no sepas cómo partir un problema, pregúntate: "¿qué pasos daría para explicárselo a una persona?". Cada paso es una función candidata.

Clase 12: Sobrecarga de funciones

En C, cada función necesita un nombre único. En C++, puedes definir varias funciones con el mismo nombre siempre que sus parámetros sean diferentes (en cantidad o en tipo). El compilador elige automáticamente la versión correcta según los argumentos. Esto se llama sobrecarga.

```
int sumar(int a, int b) {
    return a + b;
}

double sumar(double a, double b) {
    return a + b;
}

int sumar(int a, int b, int c) {
    return a + b + c;
}
```

sumar(3, 4) usa la primera; sumar(3.5, 4.2) la segunda; sumar(1, 2, 3) la tercera. Tú escribes sumar y el compilador se encarga del resto. El código se lee de forma natural, sin nombres artificiales como sumar_int o sumar_double.

Ojo: la sobrecarga se distingue por los parámetros, no por el tipo de retorno. No puedes tener int procesar(int x) y double procesar(int x): el compilador no sabría cuál elegir solo por lo que esperas recibir.

También existen los parámetros con valor predeterminado:

```
double potencia(double base, int exponente = 2) {
    double resultado = 1;
    for (int i = 0; i < exponente; i++) resultado *= base;
    return resultado;
}
```

potencia(3.0) calcula 3 al cuadrado; potencia(2.0, 10) calcula 2 a la 10. Los parámetros con default van al final. Es otra forma de tener "varias versiones" sin escribir varias funciones, y combina bien con la sobrecarga.

La sobrecarga es tu primer encuentro con una idea central de C++: el mismo nombre, comportamiento adaptado al contexto. Más adelante verás sobrecarga de operadores (unidad 8) y polimorfismo (unidad 7), que son la misma idea llevada más lejos.

Ejemplos paso a paso

Ejemplo 1: for en todas sus formas

```
#include <iostream>

int main() {
    // Contar de 1 a 10
    for (int i = 1; i <= 10; i++) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    // Contar hacia atrás de 10 en 10
    for (int i = 100; i >= 10; i -= 10) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    // Tabla de multiplicar con for anidados
    for (int i = 1; i <= 3; i++) {
        for (int j = 1; j <= 3; j++) {
            std::cout << i * j << "\t";
        }
        std::cout << std::endl;
    }

    // For de rango
    double precios[] = {19.99, 5.50, 120.00};
    double total = 0;
    for (double p : precios) {
        total += p;
    }
    std::cout << "Total: $" << total << std::endl;
    return 0;
}
```

El "\t" es un tabulador que alinea columnas. Observa cómo el for anidado genera una tabla: el interno (j) se completa para cada valor del externo (i).

Ejemplo 2: Funciones bien diseñadas

```

#include <iostream>

// Prototipos
bool esPar(int n);
int factorial(int n);
void mostrarMenu();

int main() {
    mostrarMenu();
    int num;
    std::cout << "Numero: ";
    std::cin >> num;
    if (esPar(num)) {
        std::cout << num << " es par." << std::endl;
    } else {
        std::cout << num << " es impar." << std::endl;
    }
    std::cout << "Factorial: " << factorial(num) << std::endl;
    return 0;
}

bool esPar(int n) {
    return n % 2 == 0;
}

int factorial(int n) {
    int resultado = 1;
    for (int i = 2; i <= n; i++) {
        resultado *= i;
    }
    return resultado;
}

void mostrarMenu() {
    std::cout << "=== ANALIZADOR DE NUMEROS ===" << std::endl;
}

```

Cada función tiene un nombre que dice lo que hace, hace una sola cosa y es corta. main se lee como una historia: muestra menú, pide número, analiza, muestra factorial. Esa legibilidad es el premio de usar funciones.

Ejemplo 3: Sobrecarga en acción

```
#include <iostream>
#include <string>

void mostrar(int x) {
    std::cout << "Entero: " << x << std::endl;
}

void mostrar(double x) {
    std::cout << "Decimal: " << x << std::endl;
}

void mostrar(std::string x) {
    std::cout << "Texto: " << x << std::endl;
}

void mostrar(int x, int y) {
    std::cout << "Par: (" << x << ", " << y << ")" << std::endl;
}

int main() {
    mostrar(42);
    mostrar(3.14159);
    mostrar("Hola C++");
    mostrar(10, 20);
    return 0;
}
```

Cuatro funciones llamadas `mostrar`, cada una adaptada a lo que recibe. Imagina tener que llamar `mostrar_int`, `mostrar_double`, `mostrar_string`: el código sería más verboso y menos elegante. La sobrecarga deja que el nombre exprese la intención y los parámetros el detalle.

Práctica

Ejercicio 1

Usa un `for` para mostrar los primeros 20 números pares (2, 4, 6, ..., 40) y al final su suma.

Solución:

```
#include <iostream>

int main() {
    int suma = 0;
    for (int i = 1; i <= 20; i++) {
        int par = i * 2;
        std::cout << par << " ";
        suma += par;
    }
    std::cout << "\nSuma: " << suma << std::endl;
    return 0;
}
```

Explicación: en vez de incrementar de 2 en 2, generamos el par como $i * 2$. Ambas formas funcionan; esta deja claro que son "los primeros 20 pares". El acumulador suma sigue el patrón de la unidad 3.

Ejercicio 2

Escribe la función `esPrimo(int n)` que devuelva true si `n` es primo. Pruébala en main con varios números.

Solución:

```
#include <iostream>

bool esPrimo(int n) {
    if (n < 2) return false;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return false;
    }
    return true;
}

int main() {
    for (int n = 1; n <= 20; n++) {
        if (esPrimo(n)) std::cout << n << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Explicación: basta probar divisores hasta la raíz cuadrada de n ($i * i \leq n$): si n tuviera un divisor mayor, su complemento sería menor y ya lo habríamos encontrado. Los return anticipados hacen la función corta y clara: en cuanto encontramos un divisor, terminamos.

Ejercicio 3

Sobrecarga la función `area` para calcular: el área de un cuadrado (un lado), de un rectángulo (base y altura) y de un círculo (radio, usando un tercer parámetro `char` para distinguirla o un nombre de función con tipo distinto). Prueba las tres.

Solución:

```
#include <iostream>

double area(double lado) {
    return lado * lado;
}

double area(double base, double altura) {
    return base * altura;
}

double area(double radio, bool esCirculo) {
    if (esCirculo) return 3.14159 * radio * radio;
    return 0;
}

int main() {
    std::cout << "Cuadrado: " << area(5.0) << std::endl;
    std::cout << "Rectangulo: " << area(4.0, 6.0) << std::endl;
    std::cout << "Circulo: " << area(3.0, true) << std::endl;
    return 0;
}
```

Explicación: las tres versiones se distinguen por sus parámetros. El truco del `bool` es un ejemplo didáctico; en código real, para el círculo se usaría una función con nombre distinto o un tipo específico. Lo importante es ver cómo el compilador elige la versión correcta.

Ejercicio 4

Escribe una función `dibujarTriangulo(int altura)` que imprima un triángulo de asteriscos con la altura dada, usando for anidados. Ejemplo para altura 4: `* *** *****`

Solución:

```
#include <iostream>

void dibujarTriangulo(int altura) {
    for (int fila = 1; fila <= altura; fila++) {
        for (int col = 1; col <= fila; col++) {
            std::cout << "*";
        }
        std::cout << std::endl;
    }
}

int main() {
    dibujarTriangulo(4);
    std::cout << std::endl;
    dibujarTriangulo(7);
    return 0;
}
```

Explicación: el for externo controla las filas; el interno imprime tantos asteriscos como el número de fila. Los for anidados para patrones son el ejercicio clásico para dominar el control de ciclos: si entiendes este, entiendes los ciclos.

Ejercicio 5

Crea funciones sobrecargadas máximo para: dos enteros, dos doubles y tres enteros. Úsalas en un programa que pida datos al usuario.

Solución:

```

#include <iostream>

int maximo(int a, int b) {
    return (a > b) ? a : b;
}

double maximo(double a, double b) {
    return (a > b) ? a : b;
}

int maximo(int a, int b, int c) {
    return maximo(maximo(a, b), c);
}

int main() {
    std::cout << maximo(10, 25) << std::endl;
    std::cout << maximo(3.7, 9.2) << std::endl;
    std::cout << maximo(5, 12, 8) << std::endl;
    return 0;
}

```

Explicación: la versión de tres enteros reutiliza la de dos: `maximo(maximo(a, b), c)`. Reutilizar tus propias funciones es la esencia del buen diseño. El operador `?:` es el "if compacto": `(condición) ? valor_si_true : valor_si_false`.

Resumen de la unidad

- El `for` reúne inicialización, condición y actualización en el encabezado; ideal para repeticiones contadas. El `for` de rango recorre colecciones sin índices.
- Las funciones dividen el programa en piezas con nombre, parámetros y retorno. Declara prototipos antes de `main`.
- Los parámetros se pasan por valor (copias); `return` entrega el resultado; `void` no devuelve nada.
- Cada función, una sola tarea. Nombres claros. Evita variables globales.
- La sobrecarga permite varias funciones con el mismo nombre y distintos parámetros; el compilador elige la correcta.
- En la unidad 5 entrarás a colecciones de datos: arreglos, strings modernos y el legendario mundo de los punteros.

Profundización: recursividad, alcance y diseño de funciones

En la primera parte aprendiste funciones básicas y sobrecarga. Ahora subirás de nivel: recursividad (funciones que se llaman a sí mismas), el alcance y tiempo de vida de las variables con detalle, variables estáticas, y principios de diseño que separan código amateur de profesional.

Recursividad: funciones que se llaman a sí mismas

Una función recursiva resuelve un problema llamándose con una versión más pequeña del mismo problema. Todo algoritmo recursivo necesita dos cosas: un caso base (que se resuelve directo) y un paso recursivo (que reduce el problema).

```
#include <iostream>

long long factorial(int n) {
    if (n <= 1) return 1;          // caso base
    return n * factorial(n - 1);  // paso recursivo
}

int main() {
    std::cout << factorial(5) << std::endl; // 120
    return 0;
}
```

$\text{factorial}(5) = 5 * \text{factorial}(4) = 5 * 4 * \text{factorial}(3) \dots$ hasta $\text{factorial}(1) = 1$. La pila de llamadas guarda cada nivel hasta llegar al base, luego se "desenrolla" multiplicando. Visualízalo como muñecas rusas: cada llamada contiene una más pequeña.

Sin caso base (o si nunca se alcanza), tienes recursión infinita: la pila se llena y el programa muere con "stack overflow". Siempre verifica que el problema se achique hacia el base.

La recursividad brilla en problemas naturalmente recursivos: recorrer árboles, laberintos, la serie de Fibonacci, la torre de Hanoi. Para ciclos simples, el for suele ser más eficiente y claro. Regla: usa recursividad cuando el problema "se parece a sí mismo en pequeño".

Ejemplo clásico, Fibonacci:

```
long long fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Funciona, pero es lentísimo para n grande: calcula los mismos valores miles de veces. La lección: la recursividad ingenua puede ser elegante pero ineficiente; existe la "memoización" (guardar resultados) para arreglarla, tema de cursos avanzados.

Alcance y tiempo de vida, a fondo

Ya sabes que las variables locales mueren al salir de su bloque. Profundicemos:

- Alcance de bloque: todo lo declarado entre { } vive solo ahí. Los for crean su propio alcance: la i del for no existe fuera.
- Sombreado (shadowing): puedes declarar una variable con el mismo nombre que una externa; la interna "esconde" a la externa temporalmente. Compila, pero confunde: evítalo.
- Variables globales: existen en todo el archivo. Úsalas solo para constantes. Una global modificable es una trampa: cualquier función puede cambiarla y los bugs se vuelven cazas de fantasmas.

Variables estáticas locales

Una variable static dentro de una función conserva su valor entre llamadas:

```
#include <iostream>

int contadorLlamadas() {
    static int contador = 0; // se inicializa UNA vez
    contador++;
    return contador;
}

int main() {
    std::cout << contadorLlamadas() << std::endl; // 1
    std::cout << contadorLlamadas() << std::endl; // 2
    std::cout << contadorLlamadas() << std::endl; // 3
    return 0;
}
```

La inicialización ocurre solo la primera vez. Útil para contadores, cachés simples o IDs autoincrementales. Úsala con moderación: esconde estado y complica las pruebas.

Parámetros por defecto y resolución de sobrecarga

Los parámetros con default deben ir al final, y la sobrecarga se resuelve en este orden: coincidencia exacta, promociones (int a double), conversiones estándar. Si hay ambigüedad (dos versiones igual de buenas), el compilador se queja: es tu señal de rediseñar.

```
void f(int x) { std::cout << "int"; }  
void f(double x) { std::cout << "double"; }  
// f(5) -> int (exacta). f(5.0) -> double. f('a') -> int (promoción de char).
```

Funciones inline

Para funciones pequeñísimas llamadas miles de veces, inline sugiere al compilador insertar el código directo en vez de llamar:

```
inline int cuadrado(int x) { return x * x; }
```

En compiladores modernos es solo una sugerencia (ellos deciden mejor que tú), pero documenta tu intención. No abuses: inlinear funciones grandes infla el ejecutable.

Diseño: cómo partir un problema

El arte de las funciones está en decidir dónde cortar. Señales de que necesitas una función nueva:

- Repites el mismo código dos veces: extráelo.
- Una función supera ~30-40 líneas: probablemente hace dos cosas.
- No puedes nombrar la función con claridad: no sabes qué hace; rediseñala.
- Necesitas comentarios para explicar secciones: cada sección es una función candidata.

Nombres: verbos para acciones (calcularTotal, validarEntrada), sustantivos para valores (promedio, esValido para bool). Un buen nombre elimina la necesidad de leer el cuerpo.

Más ejemplos

Ejemplo: torre de Hanoi

```
#include <iostream>

void hanoi(int discos, char origen, char destino, char auxiliar) {
    if (discos == 1) {
        std::cout << "Mueve disco 1 de " << origen << " a " << destino <<
std::endl;
        return;
    }
    hanoi(discos - 1, origen, auxiliar, destino);
    std::cout << "Mueve disco " << discos << " de " << origen << " a " << destino
<< std::endl;
    hanoi(discos - 1, auxiliar, destino, origen);
}

int main() {
    hanoi(3, 'A', 'C', 'B');
    return 0;
}
```

El problema milenario resuelto en 10 líneas: mueve n-1 discos al auxiliar, mueve el grande, mueve n-1 encima. Intentar esto con ciclos es una pesadilla; con recursividad es natural. Este ejemplo justifica aprender recursividad.

Ejemplo: generador de IDs con static

```
#include <iostream>
#include <string>

std::string nuevoFolio() {
    static int consecutivo = 1000;
    return "FOLIO-" + std::to_string(consecutivo++);
}

int main() {
    std::cout << nuevoFolio() << std::endl;
    std::cout << nuevoFolio() << std::endl;
    std::cout << nuevoFolio() << std::endl;
    return 0;
}
```

Cada llamada genera un folio único creciente. El static guarda el estado entre llamadas sin variables globales. Útil para folios, tickets y secuencias.

Práctica adicional

Ejercicio A

Escribe la función recursiva potencia(int base, int exp) para exponentes no negativos. Caso base: $\text{exp} == 0$ devuelve 1.

Solución:

```
long long potencia(int base, int exp) {
    if (exp == 0) return 1;
    return base * potencia(base, exp - 1);
}
```

Explicación: $3^4 = 3 * 3^3 = 3 * 3 * 3^2...$ hasta $3^0 = 1$. Cada llamada reduce el exponente en 1: el problema se achica garantizado.

Ejercicio B

Escribe una función recursiva que cuente los dígitos de un entero positivo: contarDigitos(12345) = 5. Pista: $12345 / 10 = 1234$.

Solución:

```
int contarDigitos(int n) {
    if (n < 10) return 1;
    return 1 + contarDigitos(n / 10);
}
```

Explicación: quitar el último dígito (división entera entre 10) achica el problema. El caso base es un solo dígito. La recursividad "pela" el número dígito por dígito.

Ejercicio C

Crea tres versiones sobrecargadas de describir: para int (muestra "Entero: X"), para double con 2 decimales (usa <iomanip>), y para string (muestra "Texto: X"). Prueba que el compilador elija bien.

Solución:

```
#include <iostream>
#include <iomanip>
#include <string>

void describir(int x) { std::cout << "Entero: " << x << std::endl; }
void describir(double x) {
    std::cout << "Decimal: " << std::fixed << std::setprecision(2) << x <<
    std::endl;
}
void describir(const std::string& x) { std::cout << "Texto: " << x << std::endl; }

int main() {
    describir(42);
    describir(3.14159);
    describir("Hola");
    return 0;
}
```

Explicación: el compilador elige por coincidencia exacta de tipos. Nota el const& en el string: eficiente y seguro, como aprendiste en la unidad 6.

Ejercicio D

Refactoriza el juego de adivinar el número de la unidad 3 en funciones: int generarSecreto(), int pedirIntento(), void darPista(int intento, int secreto). El main solo orquesta.

Solución:


```

#include <iostream>
#include <cstdlib>
#include <ctime>

int generarSecreto() {
    std::srand(std::time(0));
    return std::rand() % 100 + 1;
}

int pedirIntento() {
    int intento;
    std::cout << "Tu intento: ";
    std::cin >> intento;
    return intento;
}

void darPista(int intento, int secreto) {
    if (intento < secreto) std::cout << "Mas alto..." << std::endl;
    else if (intento > secreto) std::cout << "Mas bajo..." << std::endl;
}

int main() {
    int secreto = generarSecreto();
    int intento = 0, intentos = 0;
    std::cout << "Adivina el numero (1-100)." << std::endl;
    while (intento != secreto) {
        intento = pedirIntento();
        intentos++;
        darPista(intento, secreto);
    }
    std::cout << "¡Correcto en " << intentos << " intentos!" << std::endl;
    return 0;
}

```

Explicación: el main se lee como la descripción del juego. Cada función tiene un nombre-verbo claro y una sola tarea. Si quieres cambiar cómo se genera el secreto (por ejemplo, pedirlo a otro jugador), solo tocas generarSecreto.

Expansion: parametros, la pila, lambdas y proyectos reales

Ya sabes crear funciones, sobrecargarlas y hasta hacerlas recursivas. Esta expansion te lleva al taller interno: como viajan los datos al entrar a una funcion (y por que a veces conviene que la funcion toque tus variables), como devolver mas de un resultado, que pasa en la memoria cada vez que llamas a una funcion, como medir de verdad si la recursion le gana a la iteracion, el truco de la memoizacion que convierte un Fibonacci eterno en instantaneo, tus primeras funciones sin nombre (lambdas) y como organizar funciones en proyectos de varios archivos como los profesionales.

Paso de parametros en detalle: valor, referencia y puntero

En la primera parte aprendiste que los parametros se pasan por valor: la funcion recibe una copia. Eso es solo una de tres formas de pasar datos, y elegir la correcta es una de las decisiones mas frecuentes (y mas importantes) al disenar funciones.

Por valor: la copia segura

```
#include <iostream>

void intentarCambiar(int x) {
    x = 999; // solo cambia la copia local
}

int main() {
    int n = 5;
    intentarCambiar(n);
    std::cout << n << std::endl; // sigue valiendo 5
    return 0;
}
```

Por valor es el modo por defecto y el mas seguro: la funcion no puede danar tu variable ni aunque quiera. El precio es la copia: para un int es gratis, para un texto de un millon de caracteres es un desperdicio. Usalo para tipos pequenos (int, double, char, bool) y cuando quieras garantizar que el original no se toca.

Por referencia: la funcion toca el original

Agrega & al parametro y la funcion recibe un alias de tu variable, no una copia. Lo que haga con el parametro le pasa a tu variable:

```

#include <iostream>

void duplicar(int& x) {
    x *= 2; // modifica la variable original
}

void intercambiar(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int n = 21;
    duplicar(n);
    std::cout << n << std::endl; // 42: cambio de verdad

    int x = 10, y = 20;
    intercambiar(x, y);
    std::cout << x << " " << y << std::endl; // 20 10
    return 0;
}

```

Sin referencias, `intercambiar` seria imposible: con copias, el cambio se perderia al salir. Las referencias son la herramienta para funciones que deben modificar sus argumentos. Pero ojo: al leer `duplicar(n)` en el main, nada avisa que n va a cambiar. Por eso los profesionales reservan los parametros por referencia modificable para casos donde modificar es el proposito evidente de la funcion (como intercambiar), y documentan con el nombre.

Por referencia constante: lo mejor de los dos mundos

¿Y si quieres evitar la copia de un dato grande pero sin permitir que la funcion lo modifique? `const &`: referencia para no copiar, const para no tocar.

```

#include <iostream>
#include <string>

// Recibe el texto por referencia constante: sin copia, sin riesgo.
std::size_t contarVocales(const std::string& texto) {
    std::size_t total = 0;
    for (char c : texto) {
        char m = static_cast<char>(std::tolower(static_cast<unsigned char>(c)));
        if (m == 'a' || m == 'e' || m == 'i' || m == 'o' || m == 'u') total++;
    }
    return total;
}

int main() {
    std::string frase = "Programar en C++ es un superpoder";
    std::cout << "Vocales: " << contarVocales(frase) << std::endl;
    std::cout << "La frase sigue intacta: " << frase << std::endl;
    return 0;
}

```

Si dentro de la función intentas modificar texto, el compilador lo impide. Regla de oro profesional: todo parámetro de entrada que sea "grande" (string, struct, contenedor) se pasa por const&. Es el modo más usado en código C++ real.

Por puntero: el primo explícito

Un puntero también permite modificar el original, pero de forma explícita: en la llamada ves &n y dentro de la función ves *p. Esa explicitud es su virtud y su costo.

```
#include <iostream>

void ponerEnCero(int* p) {
    if (p != nullptr) { // el puntero puede no apuntar a nada: hay que revisar
        *p = 0;
    }
}

int main() {
    int n = 42;
    ponerEnCero(&n);      // &n grita "voy a pasar la direccion"
    std::cout << n << std::endl; // 0

    ponerEnCero(nullptr); // legal: la funcion lo maneja
    return 0;
}
```

¿Cual es la diferencia practica con la referencia? Tres:

1. Un puntero puede ser nullptr ("no hay dato"); una referencia siempre refiere a algo valido. Si "ausencia de valor" es un caso legitimo, usa puntero.
2. Un puntero se puede reasignar para apuntar a otra cosa; una referencia queda atada de por vida a su variable.
3. La sintaxis del puntero (& en la llamada, * dentro) hace visible que habra modificacion. Algunos equipos prefieren punteros justo por esa honestidad visual.

En C++ moderno, para modificar un argumento se prefiere la referencia; el puntero se reserva para cuando la nulidad es significativa, para interfaces con C y para estructuras de datos dinamicas (unidades 7 y 8).

Tabla de decision

- Parametro pequeno de entrada (int, double, char, bool) que no se modifica: por valor.
- Parametro grande de entrada (string, struct) que no se modifica: por const&.
- Parametro que la funcion debe modificar: por referencia (int&).
- Parametro opcional que puede no existir: por puntero, revisando nullptr.
- ¿Duda entre referencia y puntero para modificar? Referencia, salvo que necesites representar "sin valor".

Peligro mortal: devolver referencia a una variable local

```
int& peligro() {  
    int local = 10;  
    return local; // ¡ERROR! local muere al salir de la funcion  
}
```

La referencia devuelta apunta a una variable que ya no existe: usarla es comportamiento indefinido (puede funcionar hoy y explotar mañana). Regla: nunca devuelvas referencias ni punteros a locales. Devuelve por valor (la copia te salva) o recibe el destino como parametro por referencia.

Devolver mas de un valor: cuatro tecnicas

`return` solo puede devolver una cosa. Pero los problemas reales suelen pedir varias: cociente y residuo, minimo y maximo, exito y mensaje de error. C++ te da cuatro tecnicas; cada una tiene su momento.

Tecnica 1: parametros de salida por referencia

La funcion devuelve el resultado "principal" con return y los secundarios escribiendolos en parametros por referencia:

```

#include <iostream>

// Divide a entre b. Devuelve true si se pudo; cociente y residuo salen por
referencia.
bool dividir(int a, int b, int& cociente, int& residuo) {
    if (b == 0) return false;
    cociente = a / b;
    residuo = a % b;
    return true;
}

int main() {
    int c, r;
    if (dividir(17, 5, c, r)) {
        std::cout << "Cociente: " << c << ", residuo: " << r << std::endl;
    } else {
        std::cout << "No se puede dividir entre cero." << std::endl;
    }
    return 0;
}

```

Es la tecnica mas antigua y sigue viva en APIs classicas. Su debilidad: en la llamada `dividir(17, 5, c, r)` no se distingue que parametros son de entrada y cuales de salida; hay que leer la firma.

Tecnica 2: devolver un struct con nombre

Agrupar los resultados en una estructura con campos nombrados. Es la mas legible cuando los valores forman una unidad logica:

```

#include <iostream>

struct Estadisticas {
    double minimo;
    double maximo;
    double promedio;
};

Estadisticas analizar(double a, double b, double c) {
    Estadisticas e;
    e.minimo = a;
    if (b < e.minimo) e.minimo = b;
    if (c < e.minimo) e.minimo = c;
    e.maximo = a;
    if (b > e.maximo) e.maximo = b;
    if (c > e.maximo) e.maximo = c;
    e.promedio = (a + b + c) / 3.0;
    return e; // se devuelve por valor; el compilador evita la copia (RVO)
}

int main() {
    Estadisticas e = analizar(7.5, 9.0, 6.0);
    std::cout << "Min: " << e.minimo << " Max: " << e.maximo
                << " Prom: " << e.promedio << std::endl;
    return 0;
}

```

Los campos tienen nombre (e.minimo se explica solo), y devolver el struct por valor es barato gracias a una optimización del compilador. Cuando los valores devueltos "pertenecen juntos", el struct es la respuesta.

Técnica 3: `std::pair` para dos valores

Para exactamente dos valores relacionados, `<utility>` ofrece `std::pair`:


```

#include <iostream>
#include <utility>
#include <string>

std::pair<std::string, int> buscar(const std::string& nombre) {
    // Simula una busqueda: devuelve nombre encontrado y su puntaje
    if (nombre == "Ana") return {"Ana", 95};
    return {"no encontrado", 0};
}

int main() {
    auto resultado = buscar("Ana");
    std::cout << resultado.first << ": " << resultado.second << std::endl;
    return 0;
}

```

`first` y `second` son nombres genericos; funciona, pero si los dos valores necesitan nombres con significado, el struct gana.

Tecnica 4: `std::tuple` y `structured bindings` (C++17)

Para tres o mas valores sin querer definir un struct, `<tuple>` mas los `structured bindings` de C++17 te dejan "desempacar" el resultado en variables con nombre:

```

#include <iostream>
#include <tuple>

// Devuelve cociente, residuo y un mensaje, todo junto
std::tuple<int, int, std::string> dividirCompleto(int a, int b) {
    if (b == 0) return {0, 0, "error: division entre cero"};
    return {a / b, a % b, "ok"};
}

int main() {
    auto [cociente, residuo, estado] = dividirCompleto(17, 5);
    std::cout << estado << ": " << cociente << " residuo " << residuo << std::endl;

    auto [c2, r2, e2] = dividirCompleto(10, 0);
    std::cout << e2 << std::endl;
    return 0;
}

```

``auto [cociente, residuo, estado] = ...`` crea tres variables con los nombres que tu elijas, cada una con su tipo deducido. Es la tecnica moderna favorita para resultados multiples ad-hoc.

Guia rapida: ¿los valores forman un concepto con nombre? struct. ¿Son dos valores rapidos y temporales? pair. ¿Tres o mas sin concepto propio? tuple + structured bindings. ¿Vienes de codigo estilo C o la funcion ya devuelve un bool de exito? parametros de salida.

La pila de llamadas, dibujada

Cada vez que llamas a una funcion, C++ aparta un pedazo de memoria llamado marco de pila (stack frame) donde guarda: los parametros, las variables locales y la direccion de regreso (a donde volver cuando la funcion termine). Los marcos se apilan uno sobre otro; cuando la funcion termina, su marco se destruye y el control vuelve al marco de abajo. Por eso se llama pila: el ultimo en entrar es el primero en salir.

Recorrido visual

Sigue este programa marco por marco:

```
#include <iostream>

int doble(int x) {           // marco 3
    int r = x * 2;
    return r;
}

int sumaDobles(int a, int b) { // marco 2
    int da = doble(a);
    int db = doble(b);
    return da + db;
}

int main() {                 // marco 1 (fondo de la pila)
    int resultado = sumaDobles(3, 4);
    std::cout << resultado << std::endl; // 14
    return 0;
}
```

Lo que pasa en la pila, paso a paso:

```

[Pila] main: { resultado = ? }
[Pila] main: { resultado = ? }
      sumaDobles: { a = 3, b = 4, da = ?, db = ? }
[Pila] main: { resultado = ? }
      sumaDobles: { a = 3, b = 4, da = ?, db = ? }
      doble: { x = 3, r = ? }
[Pila] main: { resultado = ? }
      sumaDobles: { a = 3, b = 4, da = 6, db = ? }    <- doble(3) termino, su
marco murio
[Pila] main: { resultado = ? }
      sumaDobles: { a = 3, b = 4, da = 6, db = ? }
      doble: { x = 4, r = ? }
[Pila] main: { resultado = ? }
      sumaDobles: { a = 3, b = 4, da = 6, db = 8 }    <- doble(4) termino
[Pila] main: { resultado = 14 }                      <- sumaDobles termino

```

Cada llamada a `doble` crea un marco nuevo aunque sea la misma funcion: por eso la recursion funciona, cada nivel tiene sus propias variables. Y por eso las variables locales "mueren" al salir: su marco se desmonta.

Stack overflow: cuando la pila se desborda

La pila tiene tamaño limitado (unos pocos megabytes). Si encadenas demasiadas llamadas sin retornar - una recursion infinita o una recursion muy profunda - los marcos agotan la memoria y el programa muere con "stack overflow" (desbordamiento de pila). Ya viste la advertencia en la primera parte; ahora entiendes el mecanismo: no es magia, es una pila que se lleno.

Implicacion practica: una funcion recursiva con 100,000 niveles de profundidad probablemente reventara la pila, mientras que el ciclo equivalente usa un solo marco. La profundidad de recursion es un recurso finito: disenala con eso en mente.

Recursividad vs iteracion: duelo con cronometro

La primera parte te enseno que la recursion es elegante para problemas que se parecen a si mismos en pequeno. Pero ¿es rapida? Vamos a medirlo de verdad con `<chrono>`, el cronometro de la biblioteca estandar, en vez de opinar.

El experimento

El mismo problema - sumar los numeros del 1 al n - resuelto de dos formas. Medimos milisegundos con `high_resolution_clock`:

```

#include <iostream>
#include <chrono>

long long sumaIterativa(long long n) {
    long long s = 0;
    for (long long i = 1; i <= n; i++) s += i;
    return s;
}

long long sumaRecursiva(long long n) {
    if (n <= 0) return 0;
    return n + sumaRecursiva(n - 1);
}

double medirIterativa(long long n) {
    auto ini = std::chrono::high_resolution_clock::now();
    volatile long long r = sumaIterativa(n); // volatile: prohíbe al compilador
    hacer trampa
    auto fin = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms = fin - ini;
    (void)r;
    return ms.count();
}

double medirRecursiva(long long n) {
    auto ini = std::chrono::high_resolution_clock::now();
    volatile long long r = sumaRecursiva(n);
    auto fin = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms = fin - ini;
    (void)r;
    return ms.count();
}

int main() {
    const long long N = 50000;
    std::cout << "Iterativa: " << medirIterativa(N) << " ms" << std::endl;
    std::cout << "Recursiva: " << medirRecursiva(N) << " ms" << std::endl;
    return 0;
}

```

Compila sin optimización (g++ -std=c++17 programa.cpp) para que la medición sea honesta: con -O2 el compilador convierte la versión iterativa en la fórmula directa y la comparación pierde sentido.

Resultados típicos en una máquina normal: la iterativa termina en una fracción de milisegundo; la recursiva tarda varias veces más. ¿Por qué? Cada llamada recursiva paga el costo de crear y destruir un marco de pila: guardar parámetros, dirección de regreso, saltar y volver. El ciclo no paga nada de eso.

Y hay un segundo costo, peor: prueba $N = 10,000,000$ en la versión recursiva y el programa morirá con stack overflow. Diez millones de marcos no caben en la pila. La iterativa lo calcula sin despeinarse.

Lectura honesta: cuando gana cada una

- Gana la iteración cuando: el problema es un recorrido o una acumulación simple, necesitas máximo rendimiento, o la profundidad sería enorme. Es el caballo de batalla.
- Gana la recursión cuando: el problema es naturalmente recursivo (árboles, laberintos, Hanoi, búsquedas con retroceso) y la profundidad es moderada. Ahí la recursión no solo es más clara: intentar escribirla con ciclos es más propenso a errores.

La moraleja no es "la recursión es lenta" sino "cada herramienta tiene su terreno". Un profesional mide antes de afirmar, y ahora tú también sabes medir.

Memoización: Fibonacci a velocidad luz

En la primera parte viste el Fibonacci recursivo ingenuo y la advertencia de que es lentísimo. Ahora veremos por qué, con números, y cómo arreglarlo con una técnica simple y poderosa: memoización (guardar resultados ya calculados para no repetir trabajo).

El problema, contado con llamadas

$\text{fib}(5)$ llama a $\text{fib}(4)$ y $\text{fib}(3)$. $\text{fib}(4)$ llama a $\text{fib}(3)$ y $\text{fib}(2)$... El mismo $\text{fib}(3)$ se calcula dos veces, $\text{fib}(2)$ tres veces. El árbol de llamadas crece exponencialmente: $\text{fib}(40)$ ingenuo hace más de 200 millones de llamadas. Vamos a contarlos:

```

#include <iostream>
#include <vector>
#include <chrono>

long long llamadasIngenuas = 0;

long long fibIngenuo(int n) {
    llamadasIngenuas++;
    if (n <= 1) return n;
    return fibIngenuo(n - 1) + fibIngenuo(n - 2);
}

// Version con memoizacion: memo[i] guarda fib(i) ya calculado, -1 = aun no
long long fibMemo(int n, std::vector<long long>& memo) {
    if (memo[n] != -1) return memo[n];          // ya lo calcule: lo reuso
    if (n <= 1) return memo[n] = n;            // caso base, y lo guardo
    return memo[n] = fibMemo(n - 1, memo) + fibMemo(n - 2, memo); // calculo y
guardado
}

int main() {
    const int N = 40;

    auto ini = std::chrono::high_resolution_clock::now();
    long long r1 = fibIngenuo(N);
    auto fin = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms1 = fin - ini;
    std::cout << "Ingenuo: fib(" << N << ") = " << r1
        << " en " << ms1.count() << " ms, "
        << llamadasIngenuas << " llamadas." << std::endl;

    std::vector<long long> memo(N + 1, -1);
    ini = std::chrono::high_resolution_clock::now();
    long long r2 = fibMemo(N, memo);
    fin = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms2 = fin - ini;
    std::cout << "Memoizado: fib(" << N << ") = " << r2
        << " en " << ms2.count() << " ms." << std::endl;
    return 0;
}

```

El resultado es dramático: el ingenuo tarda segundos (o decenas de segundos) con cientos de millones de llamadas; el memoizado responde en una fracción de milisegundo con apenas unas 80 llamadas. De exponencial a lineal, cambiando una idea.

Como funciona: el vector memo es una cache. Antes de calcular fib(n), la funcion pregunta "¿ya lo tengo?". Si si, lo devuelve sin llamar a nadie. Si no, lo calcula una sola vez y lo guarda antes de devolverlo. Cada valor del 0 al N se calcula exactamente una vez.

La leccion general va mas alla de Fibonacci: memoizacion es cambiar memoria por tiempo. Sirve para cualquier funcion recursiva que repita subproblemas (y son muchisimas: rutas, conteos, optimizaciones). El patron es siempre el mismo: cache, preguntar antes de calcular, guardar después de calcular. Cuando en el futuro escuches "programacion dinamica", sabras que nacio de esta idea.

Nota el detalle de diseno: memo se pasa por referencia (¡la tecnica de la primera seccion en accion!) para que todas las llamadas compartan la misma cache. Por valor, cada llamada tendria su propia copia vacia y la memoizacion no serviria de nada.

Lambdas: funciones sin nombre

A veces necesitas una funcion tan pequena y tan especifica que ponerle nombre y declararla aparte es mas estorbo que ayuda: un criterio de ordenamiento, una condicion para contar, una operacion que aplicaras una sola vez. Para eso estan las lambdas: funciones anonimas que defines justo donde las usas.

Sintaxis minima

```
[capturas](parametros) { cuerpo }
```

Los corchetes son obligatorios y son la seña de identidad de la lambda. Ejemplo:

```
#include <iostream>

int main() {
    auto doble = [](int x) { return x * 2; }; // lambda guardada en una variable
    std::cout << doble(21) << std::endl;      // 42

    // 0 usada directamente, sin nombre ni variable:
    std::cout << [](int a, int b) { return a > b ? a : b; }(10, 20) << std::endl;
    return 0;
}
```

`auto doble = [](int x) {...}` crea una funcion sin nombre y la guarda en doble, que se usa como cualquier funcion. El tipo de retorno se deduce solo cuando el cuerpo es un simple return.

Capturas: la superpotencia

Una funcion normal solo ve sus parametros y las globales. Una lambda puede "capturar" variables locales del lugar donde se define, y usarlas adentro:

```
#include <iostream>

int main() {
    int umbral = 18;
    int bono = 100;

    // [=] captura por copia: la lambda ve umbral y bono como estaban al definirla
    auto esAdultoConBono = [=](int edad) {
        return edad >= umbral ? bono : 0;
    };

    // [&] captura por referencia: ve los cambios posteriores de las variables
    auto subirUmbral = [&]() { umbral += 1; };

    std::cout << esAdultoConBono(20) << std::endl; // 100
    subirUmbral();
    std::cout << "Nuevo umbral: " << umbral << std::endl; // 19
    return 0;
}
```

Formas de captura:

- `[]`` nada: la lambda solo usa sus parametros.
- `[=]` todo por copia: foto de las variables al momento de definirla.
- `[&]` todo por referencia: ve los cambios en vivo.
- `[umbral]` solo esa variable, por copia; `[&umbral]` por referencia. Capturas mixtas: `[=, &contador]` todo por copia excepto contador.

Cuidado con `[&]`: si la lambda sobrevive a las variables que capturo (por ejemplo, la devuelves de la funcion), las referencias quedan colgando igual que el peligro de la seccion de parametros. Para lambdas de uso inmediato, `[&]` es comodo y seguro.

Donde brillan: algoritmos con criterio propio

La biblioteca `<algorithm>` trae operaciones genericas (ordenar, contar, buscar) y las lambdas les ponen el criterio. Aqui con un arreglo simple, sin necesidad de la unidad 5:


```

#include <iostream>
#include <algorithm>

int main() {
    int edades[] = {15, 22, 17, 30, 12, 25, 19};
    const int N = 7;
    const int mayoria = 18;

    // Contar cuantos cumplen una condicion inventada aqui mismo
    int adultos = std::count_if(edades, edades + N,
                                [mayoria](int e) { return e >= mayoria; });
    std::cout << "Adultos: " << adultos << std::endl;

    // Ordenar de mayor a menor con criterio propio
    std::sort(edades, edades + N, [](int a, int b) { return a > b; });
    std::cout << "Ordenadas: ";
    for (int e : edades) std::cout << e << " ";
    std::cout << std::endl;
    return 0;
}

```

Sin lambdas, cada criterio necesitaria una funcion separada con nombre, lejos del lugar donde se usa. Con lambdas, el criterio vive pegado a la llamada: se lee la intencion completa en dos lineas.

Limites honestos: si la lambda supera unas 5-10 lineas, o si se usa en mas de un lugar, merece ser una funcion normal con nombre. La lambda es para logica pequena y local; convertirla en el cuerpo principal de tu programa es abusar de ella.

De un archivo a un proyecto real: .h y .cpp

Hasta ahora todos tus programas viven en un solo archivo. Los proyectos reales separan el codigo en modulos: cada modulo promete servicios en un encabezado (.h) y los implementa en un archivo fuente (.cpp). Esta separacion es la base del trabajo en equipo y de los programas de miles de lineas.

Declaracion vs definicion, en serio

- Declaracion (prototipo): "existe una funcion con esta firma". No genera codigo, solo informa. Vive en el .h.
- Definicion: el cuerpo completo. Genera codigo. Vive en el .cpp.

El .h es el contrato ("que hace"), el .cpp es el taller ("como lo hace"). Quien usa tu modulo solo necesita leer el .h.

Ejemplo: mini calculadora en tres archivos

Archivo mate.h (el contrato):

```
#ifndef MATE_H
#define MATE_H

// Prototipos: que existe, no como funciona
int sumar(int a, int b);
int restar(int a, int b);
bool esPar(int n);

#endif
```

Las tres lineas `#ifndef` / `#define` / `#endif` son las guardas de inclusion: evitan que el encabezado se procese dos veces si varios archivos lo incluyen (sin ellas, el compilador se quejaria de definiciones duplicadas). Todo .h profesional las lleva.

Archivo mate.cpp (la implementacion):

```
#include "mate.h" // comillas: busca primero en mi proyecto, no en el sistema

int sumar(int a, int b) {
    return a + b;
}

int restar(int a, int b) {
    return a - b;
}

bool esPar(int n) {
    return n % 2 == 0;
}
```

Nota las comillas en `#include "mate.h"`: para encabezados del sistema (`<iostream>`) se usan `<>`, para los tuyos se usan comillas.

Archivo main.cpp (el cliente):

```
#include <iostream>
#include "mate.h"

int main() {
    int a = 20, b = 7;
    std::cout << "Suma: " << sumar(a, b) << std::endl;
    std::cout << "Resta: " << restar(a, b) << std::endl;
    std::cout << (esPar(a) ? "a es par" : "a es impar") << std::endl;
    return 0;
}
```

Como se compila

```
g++ -std=c++17 -Wall -o programa main.cpp mate.cpp
```

El compilador procesa cada .cpp por separado y luego el enlazador (linker) los une en un ejecutable. Si olvidas mate.cpp en el comando, obtendras "undefined reference to sumar": el main promete (via el .h) pero nadie entrega la implementacion. Si defines una funcion en el .h y dos .cpp lo incluyen, el enlazador se queja de "multiple definition": las definiciones van en UN solo .cpp (salvo funciones inline o plantillas, temas avanzados).

Ventajas que se sienten rapido:

- Compilacion incremental: si cambias mate.cpp, solo se recompila ese archivo.
- Trabajo en equipo: tu companero usa tu .h sin esperar a que termines el .cpp.
- Encapsulacion: puedes reescribir mate.cpp por dentro sin tocar a nadie, mientras el .h no cambie.

Este es el esqueleto de todo proyecto C++ serio: main.cpp mas modulos .h/.cpp. Cuando llegues a clases (unidad 7), veras que cada clase vive exactamente asi.

Punteros a funciones: el abuelo de las lambdas

Antes de que existieran las lambdas, C++ ya permitia guardar funciones en variables: los punteros a funciones. Una lambda sin capturas, de hecho, puede convertirse en uno. Entenderlos te explica de donde vienen las lambdas y te prepara para APIs classicas (incluidas muchas de C) que los usan para callbacks.

Sintaxis sin miedo

La declaracion parece un acertijo, pero se lee por partes:

```
int (*operacion)(int, int); // puntero a funcion que recibe dos int y devuelve int
```

Se lee: "operacion es un puntero (*) a una funcion que toma (int, int) y devuelve int". Los parentesis alrededor de *operacion son obligatorios: sin ellos seria una funcion que devuelve un puntero. Para no escribir ese trabalenguas cada vez, los profesionales usan un alias:

```
using OperacionBinaria = int (*)(int, int);
```

Ahora `OperacionBinaria` es un tipo como cualquier otro.

Ejemplo completo: calculadora con operaciones intercambiables

La gracia es que la operacion se elige en tiempo de ejecucion: la funcion que calcula no sabe (ni le importa) cual operacion aplicara:

```

#include <iostream>

int sumar(int a, int b) { return a + b; }
int restar(int a, int b) { return a - b; }
int multiplicar(int a, int b) { return a * b; }

using OperacionBinaria = int (*)(int, int);

// Aplica la operacion recibida. El criterio viene de fuera.
int aplicar(int a, int b, OperacionBinaria op) {
    return op(a, b);
}

int main() {
    OperacionBinaria ops[] = {sumar, restar, multiplicar};
    const char* nombres[] = {"suma", "resta", "multiplicacion"};

    int a = 20, b = 6;
    for (int i = 0; i < 3; i++) {
        std::cout << nombres[i] << ": " << aplicar(a, b, ops[i]) << std::endl;
    }

    // Una lambda sin capturas también cabe en un puntero a funcion:
    OperacionBinaria maximo = [](int x, int y) { return x > y ? x : y; };
    std::cout << "maximo: " << aplicar(a, b, maximo) << std::endl;
    return 0;
}

```

Fijate en la ultima parte: una lambda sin capturas se convierte implicitamente a puntero a funcion. En cuanto la lambda captura algo ([umbral]), esa conversion ya no es posible: necesita guardar las capturas y deja de ser un simple puntero. Esa es exactamente la razon por la que las lambdas son mas poderosas: un puntero a funcion es una lambda sin memoria.

¿Donde veras punteros a funciones en la vida real? En callbacks de bibliotecas C (como qsort o las rutinas de interrupcion), en maquinas de estado donde cada estado es una funcion, y en codigo que debe elegir comportamiento en tiempo de ejecucion sin el costo de mecanismos mas pesados. Cuando el criterio necesita memoria (capturas), usa lambdas; cuando es una funcion fija conocida, el puntero es simple y rapido.

Recursion mutua: funciones que se llaman entre si

La recursion no exige que la funcion se llame a si misma directamente: dos (o mas) funciones pueden llamarse entre ellas formando un ciclo. Se llama recursion mutua o indirecta, y aparece en lugares sorprendentes: un parser donde "expresion" llama a "termino" y "termino" llama a "expresion", o un juego por turnos donde "turnoJugador" llama a "turnoEnemigo" y viceversa.

El ejemplo clasico: par e impar definidos el uno con el otro. Un numero es par si el anterior es impar; es impar si el anterior es par; el cero es par:

```
#include <iostream>

bool esParRec(int n); // declaracion adelantada: esImparRec la necesita antes

bool esImparRec(int n) {
    if (n == 0) return false; // caso base
    return esParRec(n - 1);    // paso: pregunto por el anterior
}

bool esParRec(int n) {
    if (n == 0) return true;   // caso base
    return esImparRec(n - 1);  // paso: pregunto por el anterior
}

int main() {
    for (int i = 0; i <= 10; i++) {
        std::cout << i << (esParRec(i) ? " es par" : " es impar") << std::endl;
    }
    return 0;
}
```

Sin la declaracion adelantada de esParRec, el codigo no compila: esImparRec la usa antes de su definicion. Este es uno de los casos donde los prototipos no son opcionales sino obligatorios.

Cada funcion necesita su propio caso base, y el problema debe achicarse en cada salto entre funciones: aqui n baja en 1 cada llamada, asi que la cadena siempre llega a cero. Si una de las dos olvidara reducir n, tendrías recursion infinita mutua y el mismo stack overflow de siempre.

En la pila, la recursion mutua se ve como marcos alternados: esParRec, esImparRec, esParRec... cada uno con su propia n. El mecanismo es identico al de la recursion directa; solo cambia quien llama a quien.

Ejemplo menos juguete, mas real: turnos de un duelo. Cada funcion representa el turno de un combatiente y llama a la del otro hasta que alguien cae:

```
#include <iostream>
#include <string>

struct Combatiente {
    std::string nombre;
    int salud;
};

void turnoDe(Combatiente& atacante, Combatiente& defensor); // adelantada

void revisarFin(Combatiente& a, Combatiente& b) {
    if (b.salud <= 0) {
        std::cout << a.nombre << " gana el duelo." << std::endl;
        return;
    }
    turnoDe(b, a); // el defensor ahora ataca: recursion mutua
}

void turnoDe(Combatiente& atacante, Combatiente& defensor) {
    defensor.salud -= 10;
    std::cout << atacante.nombre << " golpea. Salud de "
        << defensor.nombre << ": " << defensor.salud << std::endl;
    revisarFin(atacante, defensor);
}

int main() {
    Combatiente heroe{"Heroe", 30};
    Combatiente troll{"Troll", 30};
    turnoDe(heroe, troll);
    return 0;
}
```

El duelo avanza alternando atacantes sin ningun ciclo: la recursion mutua ES el bucle. Para duelos cortos es elegante y expresivo; para batallas de miles de turnos, un while seria mas seguro para la pila. La misma regla de siempre: recursion para profundidad moderada y estructura clara, iteracion para lo demas.

Funciones constexpr: calculos en tiempo de compilacion

Un ultimo superpoder de las funciones modernas: `constexpr` le dice al compilador "esta funcion puede evaluarse mientras compilas, no hace falta esperar a que el programa corra". Si la llamas con valores conocidos al compilar, el resultado ya viene calculado dentro del ejecutable: cero costo en tiempo de ejecucion.

```
#include <iostream>

constexpr long long factorialConst(int n) {
    long long r = 1;
    for (int i = 2; i <= n; i++) r *= i; // desde C++14, constexpr permite ciclos
    return r;
}

int main() {
    constexpr long long f10 = factorialConst(10); // se calcula AL COMPILAR
    static_assert(factorialConst(5) == 120, "factorial(5) debe ser 120");

    std::cout << "10! = " << f10 << std::endl;

    int n = 6;
    std::cout << n << "! = " << factorialConst(n) << std::endl; // aqui si corre
    normal
    return 0;
}
```

Tres ideas clave:

1. `static\_assert` es la verificacion en tiempo de compilacion: si la condicion es falsa, el programa ni siquiera compila. Es como un examen que tu codigo debe pasar antes de nacer. Usalo para validar supuestos ("esta tabla mide 256", "el factorial de 5 es 120").
2. Una funcion constexpr sigue siendo una funcion normal: si la llamas con una variable (n = 6), se ejecuta en tiempo de ejecucion como siempre. constexpr no obliga, permite.
3. Las reglas: una funcion constexpr solo puede usar operaciones "computables al compilar" (sin entrada/salida, sin memoria dinamica, sin llamadas a funciones no-constexpr). El compilador te avisara si te pasas.

¿Para que sirve en la practica? Tablas precalculadas (senos, codigos), validacion de constantes, y documentar intencion: un constexpr grita "este valor se conoce desde el codigo fuente". No lo necesitas todos los dias, pero cuando lo necesitas no hay sustituto.

Catalogo de errores comunes

1. Modificar un parametro por valor esperando que el cambio salga: ``void f(int x) { x = 5; }`` nunca afecta al argumento. Si la funcion debe modificar, el parametro lleva `&`.
2. Pasar structs o strings grandes por valor en cada llamada: funciona, pero copias megabytes en cada llamada. Usa `const&`.
3. Devolver referencia o puntero a una variable local: la variable muere al salir y la referencia queda colgando. Devuelve por valor.
4. Pasar `nullptr` a una funcion que no lo revisa: el programa truena al desreferenciar. Si el parametro es puntero, revisa `nullptr` o documenta que no se acepta.
5. Olvidar `<tuple>` o `<utility>` al usar `tuple/pair`: el error del compilador habla de plantillas y asusta; casi siempre es un `include` faltante.
6. Structured bindings con el numero equivocado de variables: ``auto [a, b] =`` sobre un tuple de 3 no compila. El numero debe coincidir exacto.
7. Lambda con `[&]` que sobrevive a sus variables: si guardas la lambda y la usas después de que las locales murieron, es comportamiento indefinido. Para lambdas que viajan, captura por copia.
8. Olvidar las guardas de inclusion en un `.h`: incluirlo desde dos archivos da errores de redefinicion. Todo `.h` lleva `#ifndef`.
9. Definir funciones `no-inline` en el `.h`: cada `.cpp` que lo incluya genera su copia y el enlazador grita "multiple definition". Las definiciones van en el `.cpp`.
10. Compilar solo `main.cpp` olvidando el otro `.cpp`: "undefined reference". El comando `g++` debe listar todos los `.cpp`.
11. Medir rendimiento con `-O2` sin saberlo: el compilador optimiza tu experimento y mides su inteligencia, no tu algoritmo. Para comparar algoritmos, compila sin optimizacion o disena la prueba a prueba de optimizaciones.
12. Recursion profunda sin pensar en la pila: 100,000 niveles de recursion revientan la pila; el ciclo equivalente ni se inmuta. La profundidad es un recurso finito.

Practica

Ejercicio 1: intercambiar y min-max

Escribe `void intercambiar(int& a, int& b)` y `void minMax(double x, double y, double& menor, double& mayor)` (los resultados salen por referencia). Prueba ambas en main con valores leídos del usuario.

Solucion:

```
#include <iostream>

void intercambiar(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void minMax(double x, double y, double& menor, double& mayor) {
    if (x < y) {
        menor = x;
        mayor = y;
    } else {
        menor = y;
        mayor = x;
    }
}

int main() {
    int a = 10, b = 25;
    intercambiar(a, b);
    std::cout << a << " " << b << std::endl; // 25 10

    double x, y, menor, mayor;
    std::cout << "Dos numeros: ";
    std::cin >> x >> y;
    minMax(x, y, menor, mayor);
    std::cout << "Menor: " << menor << " Mayor: " << mayor << std::endl;
    return 0;
}
```

Explicacion: intercambiar es imposible por valor: necesita los alias &. minMax muestra la tecnica de parametros de salida: la funcion "devuelve" dos valores escribiendolos en referencias. En la llamada se nota quien es entrada (x, y) y quien es salida (menor, mayor) solo si lees la firma: documenta con nombres claros.

Ejercicio 2: fracciones con struct

Define `struct Fraccion { int num, den; }` y la funcion `Fraccion sumar(Fraccion a, Fraccion b)` que devuelva la suma simplificada (usa el maximo comun divisor). Pruébala con $1/2 + 1/3 = 5/6$.

Solucion:

```
#include <iostream>

struct Fraccion {
    int num;
    int den;
};

int mcd(int a, int b) {
    while (b != 0) {
        int t = b;
        b = a % b;
        a = t;
    }
    return a;
}

Fraccion simplificar(Fraccion f) {
    int d = mcd(f.num < 0 ? -f.num : f.num, f.den);
    f.num /= d;
    f.den /= d;
    return f;
}

Fraccion sumar(Fraccion a, Fraccion b) {
    Fraccion r;
    r.num = a.num * b.den + b.num * a.den;
    r.den = a.den * b.den;
    return simplificar(r);
}

int main() {
    Fraccion r = sumar({1, 2}, {1, 3});
    std::cout << r.num << "/" << r.den << std::endl; // 5/6
    return 0;
}
```

Explicacion: devolver el struct por valor es limpio y barato (el compilador evita la copia).
`sumar({1, 2}, {1, 3})` construye las fracciones al vuelo. El mcd usa el algoritmo de Euclides con un while. Cuando los datos devueltos forman un concepto (una fraccion), el struct con campos nombrados es la tecnica correcta.

Ejercicio 3: mide tu maquina

Compara dos formas de calcular la suma del 1 al n (numero triangular): con ciclo y con la formula $n*(n+1)/2$. Midelas con <chrono> para n = 100,000,000 y reporta los tiempos.

Solucion:

```
#include <iostream>
#include <chrono>

long long triangularCiclo(long long n) {
    long long s = 0;
    for (long long i = 1; i <= n; i++) s += i;
    return s;
}

long long triangularFormula(long long n) {
    return n * (n + 1) / 2;
}

int main() {
    const long long N = 100000000LL;

    auto ini = std::chrono::high_resolution_clock::now();
    volatile long long r1 = triangularCiclo(N);
    auto fin = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms1 = fin - ini;

    ini = std::chrono::high_resolution_clock::now();
    volatile long long r2 = triangularFormula(N);
    fin = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms2 = fin - ini;

    std::cout << "Ciclo: " << ms1.count() << " ms, resultado " << r1 << std::endl;
    std::cout << "Formula: " << ms2.count() << " ms, resultado " << r2 <<
std::endl;
    return 0;
}
```

Explicacion: el ciclo tarda decenas o cientos de milisegundos; la formula, practicamente cero. Ambos dan el mismo resultado. La leccion es de algoritmos, no de C++: un cambio de estrategia ($O(n)$ a $O(1)$) le gana a cualquier micro-optimizacion. Por eso medir importa: convierte opiniones en datos. Compila sin `-O2` para que la prueba sea honesta.

Ejercicio 4: escalones con memoizacion

Hay n escalones y puedes subirlos de 1 en 1 o de 2 en 2. ¿De cuantas formas puedes llegar arriba? Implementa la version recursiva ingenua `formas(n) = formas(n-1) + formas(n-2)` (con `formas(0)=1`, `formas(1)=1`), midela con $n=35$, y luego escribe la version memoizada y comparalas.

Solucion:

```

#include <iostream>
#include <vector>
#include <chrono>

long long formasIngenua(int n) {
    if (n <= 1) return 1;
    return formasIngenua(n - 1) + formasIngenua(n - 2);
}

long long formasMemo(int n, std::vector<long long>& memo) {
    if (memo[n] != -1) return memo[n];
    if (n <= 1) return memo[n] = 1;
    return memo[n] = formasMemo(n - 1, memo) + formasMemo(n - 2, memo);
}

int main() {
    const int N = 35;

    auto ini = std::chrono::high_resolution_clock::now();
    long long r1 = formasIngenua(N);
    auto fin = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms1 = fin - ini;
    std::cout << "Ingenua: " << r1 << " en " << ms1.count() << " ms" << std::endl;

    std::vector<long long> memo(N + 1, -1);
    ini = std::chrono::high_resolution_clock::now();
    long long r2 = formasMemo(N, memo);
    fin = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms2 = fin - ini;
    std::cout << "Memoizada: " << r2 << " en " << ms2.count() << " ms" <<
std::endl;
    return 0;
}

```

Explicacion: es el mismo patron que Fibonacci porque es el mismo problema disfrazado: cada escalon se alcanza desde los dos anteriores. La ingenua tarda segundos con $n=35$; la memoizada es instantanea. Si reconoces la forma $f(n) = f(n-1) + f(n-2)$, ya sabes que la memoizacion la vuelve lineal. Este es el primer paso hacia la programacion dinamica.

Ejercicio 5: lambda con criterio propio

Dado el arreglo `int edades[] = {15, 22, 17, 30, 12, 25, 19, 41, 16, 28};`, usa `std::count_if` con una lambda que capture la mayoría de edad para contar adultos, y luego ordena el arreglo de mayor a menor con `std::sort` y una lambda. Imprime ambos resultados.

Solucion:

```
#include <iostream>
#include <algorithm>

int main() {
    int edades[] = {15, 22, 17, 30, 12, 25, 19, 41, 16, 28};
    const int N = 10;
    const int mayoria = 18;

    int adultos = std::count_if(edades, edades + N,
                               [mayoria](int e) { return e >= mayoria; });
    std::cout << "Adultos: " << adultos << std::endl;

    std::sort(edades, edades + N, [](int a, int b) { return a > b; });

    std::cout << "De mayor a menor: ";
    for (int e : edades) std::cout << e << " ";
    std::cout << std::endl;
    return 0;
}
```

Explicacion: la primera lambda captura mayoria por copia (`[mayoria]`) porque solo la lee; la de ordenamiento no captura nada (`[]`) porque solo compara sus parametros. El criterio vive pegado a la llamada: cualquiera que lea el codigo entiende la intencion sin buscar funciones lejanas. Si el criterio creciera o se reusara, seria momento de convertirlo en funcion con nombre.

Resumen de la expansion

- Por valor copia (seguro, para tipos pequenos); por referencia modifica el original (para salidas); por `const&` evita copias sin riesgo (para entradas grandes); por puntero cuando la ausencia de valor es legitima.
- Nunca devuelvas referencias ni punteros a variables locales.

- Para devolver varios valores: struct con nombre si forman un concepto, pair para dos valores rapidos, tuple + structured bindings para varios sin concepto propio, parametros de salida al estilo clasico.
- Cada llamada crea un marco en la pila con parametros y locales; al terminar, el marco muere. La recursion profunda agota la pila: la profundidad es un recurso finito.
- Medir con <chrono> convierte opiniones en datos: la iteracion suele ganar en velocidad y la recursion en claridad para problemas recursivos.
- Memoizacion: cache + preguntar antes de calcular + guardar después. Convierte recursion exponencial en lineal cambiando memoria por tiempo.
- Las lambdas son funciones anonimas con capturas ([], [=], [&]): perfectas para criterios pequenos junto a algoritmos como sort y count_if.
- Los proyectos reales separan contrato (.h con prototipos y guardas) de implementacion (.cpp); se compilan listando todos los .cpp y el enlazador los une.
- Los punteros a funciones guardan comportamiento en variables y permiten elegir operaciones en tiempo de ejecucion; una lambda sin capturas puede convertirse en uno.
- La recursion mutua (funciones que se llaman entre si) necesita declaraciones adelantadas y un caso base en cada funcion; sirve para parsers, dialogos y turnos alternados.
- constexpr mueve calculos al tiempo de compilacion y static_assert los verifica antes de que el programa nazca.

Con esto cierras el taller de funciones: ya no solo las usas, entiendes como viajan los datos, cuanto cuestan, como se miden y como se organizan en proyectos de verdad. En la unidad 5 ese conocimiento se vuelve poder sobre colecciones de datos: arreglos, strings modernos y punteros.

UNIDAD 5: Arreglos, strings y punteros

Llegaste a la unidad que separa a los turistas de los residentes en C++. Aquí manejarás colecciones de datos con arreglos, texto moderno con `std::string` y, sobre todo, punteros: el tema más famoso (y más temido) del lenguaje. Te prometo algo: al terminar esta unidad, los punteros no te darán miedo. Los entenderás, los usarás y sabrás exactamente cuándo preferir alternativas modernas.

La idea central es la memoria. Hasta ahora tus variables eran cajas aisladas. Ahora verás que la memoria es un gran almacén numerado, que los arreglos son filas de cajas contiguas y que un puntero es simplemente la dirección de una caja. Cuando lo ves así, todo encaja.

Al terminar sabrás: declarar y recorrer arreglos, usar `std::string` para texto sin sufrimiento, declarar punteros, obtener direcciones con `&` y valores con `*`, la relación entre arreglos y punteros, y reservar memoria dinámica con `new` y `delete`.

Explicación detallada

Clase 13: Arreglos

Un arreglo guarda varios valores del mismo tipo bajo un solo nombre, ordenados por índice. Piensa en una fila de casilleros numerados: el nombre es el edificio, el índice es el número de casillero.

```
int calificaciones[5] = {8, 9, 7, 10, 6};
```

Esto crea 5 enteros contiguos en memoria. Los índices empiezan en 0: `calificaciones[0]` es 8, `calificaciones[4]` es 6. El primer índice siempre es 0 y el último es tamaño - 1. Graba esto a fuego: el error más común es acceder al índice 5 en un arreglo de 5, que no existe.

Puedes inicializar con menos valores: `int datos[10] = {1, 2};` pone 1 y 2, y el resto queda en cero. Puedes omitir el tamaño si inicializas: `int datos[] = {1, 2, 3, 4, 5, 6};` el compilador cuenta por ti.

Para recorrer un arreglo usa un `for`:

```
int suma = 0;
for (int i = 0; i < 5; i++) {
    suma += calificaciones[i];
}
```

O mejor, el for de rango: `for (int c : calificaciones) suma += c;`. Sin índices, sin riesgo de pasarte.

La regla de oro de los arreglos en C++: el lenguaje NO verifica que tus índices sean válidos. Si escribes `calificaciones[100]`, el programa lee o escribe memoria que no le pertenece. A veces "funciona", a veces se cae, a veces corrompe otros datos en silencio. Es uno de los errores más peligrosos y por eso en C++ moderno se prefiere vector (unidad 8), que sí puede verificar. Mientras uses arreglos crudos, cuida tus índices como oro.

Los arreglos se pueden pasar a funciones, pero con un detalle importante: al pasar un arreglo a una función, en realidad pasas un puntero al primer elemento, y la función no sabe el tamaño. Por eso siempre se pasa el tamaño aparte:

```
void mostrar(int arr[], int tamano) {
    for (int i = 0; i < tamano; i++) {
        std::cout << arr[i] << " ";
    }
}
```

Los arreglos multidimensionales son arreglos de arreglos. Una matriz de 3x4: `int matriz[3][4];`. Se recorre con for anidados. Útiles para tableros, imágenes y tablas.

Clase 14: Strings modernos con `std::string`

En C, el texto es un arreglo de char terminado en cero: frágil y tedioso. C++ ofrece `std::string`, una clase que maneja texto como debe ser. Incluye `<string>` y olvídate del sufrimiento.

```
#include <string>
std::string nombre = "Esteban";
std::string saludo = "Hola, " + nombre + "!";
```

Puedes sumar strings con `+`, compararlos con `==` (compara contenido, no direcciones), saber su longitud con `.length()` o `.size()`, y acceder a caracteres con `[i]` como un arreglo. Métodos útiles: `.substr(pos, len)` extrae un pedazo, `.find(texto)` busca y devuelve la posición (o `std::string::npos` si no encuentra), `.empty()` dice si está vacío.

Para leer texto con espacios usa getline:

```
std::string linea;  
std::getline(std::cin, linea);
```

Recuerda el detalle de la unidad 1: `cin >>` deja el salto de línea pendiente, así que si mezclas `cin >>` número con `getline`, el `getline` lee una línea vacía. La solución estándar es `std::cin.ignore()` entre ambos para descartar el salto pendiente. Es un truco que todo programador de C++ conoce:

```
int edad;  
std::string nombre;  
std::cin >> edad;  
std::cin.ignore(); // descarta el Enter pendiente  
std::getline(std::cin, nombre);
```

Los strings se pasan a funciones por referencia constante (lo verás en la unidad 6): `void saludar(const std::string& nombre)`. Por ahora basta saber que existe esa forma eficiente.

Clase 15: Punteros

Un puntero es una variable que guarda una dirección de memoria en vez de un dato normal. Si las variables son cajas, un puntero es un papelito con el número de pasillo y estante donde está la caja.

```
int x = 42;  
int* p = &x;
```

El operador `&` obtiene la dirección: `&x` es "la dirección de x". El `*` en la declaración dice "p es un puntero a int". Después, `*p` (el operador de desreferencia) significa "el valor guardado en la dirección a la que apunta p". Así que `*p` vale 42, y `*p = 100` cambia x a 100. Dos símbolos, dos significados según el contexto: en declaración, `*` crea un puntero; en uso, `*` sigue la dirección.

¿Para qué sirven? Tres usos principales. Primero: funciones que modifican sus argumentos. Como los parámetros son copias, una función no puede cambiar la variable original... a menos que le pases su dirección:

```
void duplicar(int* n) {
    *n = *n * 2;
}
int valor = 21;
duplicar(&valor); // valor ahora es 42
```

Segundo: memoria dinámica. Con new pides memoria en tiempo de ejecución y con delete la devuelves:

```
int* p = new int(42); // reserva un entero con valor 42
std::cout << *p << std::endl;
delete p; // libera la memoria
```

Si olvidas delete, tienes una fuga de memoria: memoria reservada que nadie usa ni libera. En programas largos, las fugas acumulan hasta agotar la memoria. Por eso el C++ moderno prefiere punteros inteligentes (unidad 10), que hacen delete solos.

Tercero: estructuras dinámicas y polimorfismo, que verás más adelante. Los punteros son la base de las listas, árboles y del polimorfismo en tiempo de ejecución.

La relación entre arreglos y punteros: el nombre de un arreglo es esencialmente un puntero a su primer elemento. calificaciones equivale a &calificaciones[0], y calificaciones[i] equivale a *(calificaciones + i). La aritmética de punteros (sumar a un puntero) se mueve por elementos, no por bytes: p + 1 apunta al siguiente entero. Es elegante y peligroso a partes iguales.

Puntero nulo: un puntero que no apunta a nada se representa con nullptr. Inicializa siempre tus punteros: int* p = nullptr;. Usar un puntero sin inicializar es como seguir una dirección inventada: el programa se cae o corrompe memoria. Antes de desreferenciar un puntero que podría ser nulo, verifícalo: if (p != nullptr).

Errores clásicos: olvidar inicializar, usar memoria ya liberada (puntero colgante), liberar dos veces la misma memoria, y confundir * en declaración con * en uso. Con disciplina (inicializar, verificar, liberar una sola vez) los punteros son manejables. Y recuerda: en C++ moderno, muchas tareas que antes exigían punteros crudos hoy se hacen con referencias, string, vector y punteros inteligentes.

Ejemplos paso a paso

Ejemplo 1: Arreglos en acción

```
#include <iostream>

int main() {
    double ventas[7];
    double total = 0;

    std::cout << "Ventas de la semana:" << std::endl;
    for (int i = 0; i < 7; i++) {
        std::cout << "Dia " << i + 1 << ": $";
        std::cin >> ventas[i];
        total += ventas[i];
    }

    std::cout << "Total: $" << total << std::endl;
    std::cout << "Promedio: $" << total / 7 << std::endl;

    // Encontrar el mejor día
    int mejor = 0;
    for (int i = 1; i < 7; i++) {
        if (ventas[i] > ventas[mejor]) mejor = i;
    }
    std::cout << "Mejor dia: " << mejor + 1 << " ($" << ventas[mejor] << ")" <<
std::endl;
    return 0;
}
```

Un arreglo para 7 datos relacionados, un for para llenarlo, otro patrón "supón y mejora" para el máximo. Este es el uso típico de arreglos: colecciones homogéneas que se procesan en ciclo.

Ejemplo 2: Análisis de texto con string

```

#include <iostream>
#include <string>

int main() {
    std::string frase;
    std::cout << "Escribe una frase: ";
    std::getline(std::cin, frase);

    int vocales = 0;
    for (char c : frase) {
        char min = std::tolower(c);
        if (min == 'a' || min == 'e' || min == 'i' || min == 'o' || min == 'u') {
            vocales++;
        }
    }

    std::cout << "La frase tiene " << frase.length() << " caracteres." <<
std::endl;
    std::cout << "Vocales: " << vocales << std::endl;

    if (frase.find("C++") != std::string::npos) {
        std::cout << "¡Mencionaste C++!" << std::endl;
    }
    return 0;
}

```

getline lee la frase completa con espacios. El for de rango recorre cada carácter. find busca "C++" y lo comparamos con npos (el valor que significa "no encontrado"). Todo con std::string, sin un solo char* manual.

Ejemplo 3: Punteros paso a paso

```

#include <iostream>

void intercambiar(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;
    std::cout << "Antes: x=" << x << ", y=" << y << std::endl;
    std::cout << "Direccion de x: " << &x << std::endl;

    int* p = &x;
    std::cout << "p apunta a: " << p << ", valor: " << *p << std::endl;

    *p = 99;
    std::cout << "Despues de *p = 99: x=" << x << std::endl;

    intercambiar(&x, &y);
    std::cout << "Despues de intercambiar: x=" << x << ", y=" << y << std::endl;

    // Memoria dinámica
    int* dinamico = new int(42);
    std::cout << "Dinamico: " << *dinamico << std::endl;
    delete dinamico;
    dinamico = nullptr;
    return 0;
}

```

Este ejemplo muestra todo: & para direcciones, * para desreferenciar, punteros como parámetros para modificar originales, y new/delete. Después de delete, asignamos nullptr para no dejar un puntero colgante. Ejecútalo y observa las direcciones: son números hexadecimales largos, las "coordenadas" de la memoria.

Práctica

Ejercicio 1

Pide 10 números, guárdalos en un arreglo y muestra: el promedio, cuántos están por encima del promedio y el mayor.

Solución:

```
#include <iostream>

int main() {
    const int N = 10;
    double nums[N];
    double suma = 0;

    for (int i = 0; i < N; i++) {
        std::cout << "Numero " << i + 1 << ": ";
        std::cin >> nums[i];
        suma += nums[i];
    }

    double promedio = suma / N;
    int encima = 0;
    double mayor = nums[0];
    for (int i = 0; i < N; i++) {
        if (nums[i] > promedio) encima++;
        if (nums[i] > mayor) mayor = nums[i];
    }

    std::cout << "Promedio: " << promedio << std::endl;
    std::cout << "Encima del promedio: " << encima << std::endl;
    std::cout << "Mayor: " << mayor << std::endl;
    return 0;
}
```

Explicación: const int N evita "números mágicos" regados en el código. Dos pasadas sobre el arreglo: una para leer y sumar, otra para analizar. Separar lectura de análisis hace el código más claro.

Ejercicio 2

Pide una palabra al usuario y determina si es un palíndromo (se lee igual al derecho y al revés, como "oso" o "reconocer"). Usa std::string.

Solución:


```

#include <iostream>
#include <string>

int main() {
    std::string palabra;
    std::cout << "Palabra: ";
    std::cin >> palabra;

    bool esPalindromo = true;
    int n = palabra.length();
    for (int i = 0; i < n / 2; i++) {
        if (palabra[i] != palabra[n - 1 - i]) {
            esPalindromo = false;
            break;
        }
    }

    if (esPalindromo) {
        std::cout << palabra << " es palindromo." << std::endl;
    } else {
        std::cout << palabra << " no es palindromo." << std::endl;
    }
    return 0;
}

```

Explicación: comparamos el primer carácter con el último, el segundo con el penúltimo, etc. Basta llegar a la mitad. break sale en cuanto encontramos una diferencia: no tiene sentido seguir revisando.

Ejercicio 3

Escribe una función void invertir(int arr[], int tamano) que invierta el arreglo en su lugar (el primero queda último, etc.), usando punteros o índices. Pruébala.

Solución:

```

#include <iostream>

void invertir(int* arr, int tamano) {
    for (int i = 0; i < tamano / 2; i++) {
        int temp = arr[i];
        arr[i] = arr[tamano - 1 - i];
        arr[tamano - 1 - i] = temp;
    }
}

void mostrar(int* arr, int tamano) {
    for (int i = 0; i < tamano; i++) std::cout << arr[i] << " ";
    std::cout << std::endl;
}

int main() {
    int datos[] = {1, 2, 3, 4, 5};
    mostrar(datos, 5);
    invertir(datos, 5);
    mostrar(datos, 5);
    return 0;
}

```

Explicación: intercambiamos pares simétricos hasta la mitad. Como el arreglo se pasa como puntero, la función modifica el original directamente. Nota la firma `int* arr`: es equivalente a `int arr[]` en parámetros.

Ejercicio 4

Crea un programa que pida al usuario cuántos números quiere ingresar, reserve un arreglo dinámico de ese tamaño con `new`, lea los números, muestre el promedio y libere la memoria con `delete[]`.

Solución:

```

#include <iostream>

int main() {
    int n;
    std::cout << "Cuantos numeros? ";
    std::cin >> n;

    double* datos = new double[n];
    double suma = 0;
    for (int i = 0; i < n; i++) {
        std::cout << "Numero " << i + 1 << ": ";
        std::cin >> datos[i];
        suma += datos[i];
    }
    std::cout << "Promedio: " << suma / n << std::endl;

    delete[] datos;
    datos = nullptr;
    return 0;
}

```

Explicación: con `new[]` reservas memoria para `n` elementos donde `n` se conoce hasta la ejecución: imposible con arreglos estáticos. Los corchetes en `delete[]` son obligatorios para arreglos; `delete` sin corchetes en un arreglo es un error grave. Esta es la razón histórica de los punteros, aunque hoy `vector` lo hace mejor.

Ejercicio 5

Escribe una función `contarCaracter(const std::string& texto, char letra)` que cuente cuántas veces aparece la letra (sin importar mayúsculas/minúsculas). Pruébala con una frase del usuario.

Solución:

```

#include <iostream>
#include <string>
#include <cctype>

int contarCaracter(const std::string& texto, char letra) {
    int contador = 0;
    char buscada = std::tolower(letra);
    for (char c : texto) {
        if (std::tolower(c) == buscada) contador++;
    }
    return contador;
}

int main() {
    std::string frase;
    std::cout << "Frase: ";
    std::getline(std::cin, frase);
    char letra;
    std::cout << "Letra a contar: ";
    std::cin >> letra;
    std::cout << "Aparece " << contarCaracter(frase, letra) << " veces." <<
std::endl;
    return 0;
}

```

Explicación: convertimos todo a minúsculas con tolower para comparar sin importar el caso. El parámetro es const std::string&: referencia constante, eficiente y segura (adelanto de la unidad 6). cctype trae las funciones de caracteres.

Resumen de la unidad

- Los arreglos guardan colecciones del mismo tipo; los índices van de 0 a tamaño-1 y C++ no verifica límites: cuídalos.
- std::string es el texto moderno: se suma, se compara, mide su longitud y se lee con getline. Usa cin.ignore() al mezclar con cin >>.
- Un puntero guarda una dirección: & obtiene direcciones, * las sigue. Inicializa con nullptr.
- Los punteros permiten modificar argumentos en funciones y reservar memoria dinámica con new/delete (y delete[] para arreglos).
- Fugas de memoria, punteros colgantes y doble liberación son los pecados capitales: disciplina o punteros inteligentes.

- En la unidad 6 conocerás las referencias y entrarás al mundo de las clases y objetos.

Profundización: el modelo de memoria de C++

Los punteros dejan de dar miedo cuando entiendes dónde viven las cosas. Esta profundización te muestra el mapa completo de la memoria de tu programa: la pila (stack), el montón (heap), el segmento de datos y el código. Con este mapa, new, delete, los punteros colgantes y las fugas se vuelven obvios.

Las cuatro regiones de la memoria

Cuando tu programa corre, el sistema operativo le da cuatro zonas:

1. Código: tus instrucciones compiladas. Solo lectura.
2. Datos globales/estáticos: variables globales y static. Existen durante todo el programa.
3. Pila (stack): variables locales y parámetros. Crece y se encoge automáticamente con cada llamada a función. Rápida, pero limitada (típicamente 1-8 MB).
4. Montón (heap): memoria pedida con new/malloc. Tú la administras: vive hasta que haces delete. Grande, pero manual.

Las variables locales viven en la pila: nacen al entrar al bloque y mueren al salir, automáticamente. Por eso no necesitas liberarlas. El heap es para lo que debe sobrevivir a la función que lo creó, o cuyo tamaño solo se conoce en ejecución.

Analogía: la pila es como la bandeja de entrada de tu escritorio (automática, limitada, se limpia sola); el heap es como rentar una bodega (tú pides el espacio, tú lo devuelves, cabe mucho más).

La pila de llamadas, visualizada

Cada llamada a función apila un "marco" con sus variables locales y la dirección de retorno:

```
void c() { int z = 3; }      // marco de c encima del de b
void b() { int y = 2; c(); } // marco de b encima del de a
void a() { int x = 1; b(); } // marco de a
```

Cuando c termina, su marco se destruye y b continúa. Por eso las variables locales "desaparecen": su marco se desmontó. Y por eso devolver punteros a variables locales es un error mortal: apuntas a un marco que ya no existe.

El "stack overflow" ocurre cuando la pila se llena: recursión infinita o arreglos locales gigantes. Un `int datos[1000000]`; local puede tumbar tu programa: un millón de enteros son 4 MB, más que la pila típica. Para arreglos grandes usa el heap (`new` o `vector`).

Aritmética de punteros, a fondo

Sumar a un puntero avanza por elementos, no por bytes:

```
int arr[] = {10, 20, 30, 40};
int* p = arr;      // apunta a arr[0]
p++;               // ahora apunta a arr[1] (avanza 4 bytes: un int)
std::cout << *p;   // 20
```

`p + i` apunta al elemento `i`. Por eso `arr[i]` es exactamente `*(arr + i)`: el índice es un desplazamiento. Incluso `i[arr]` funciona (es `*(i + arr)`), aunque nadie lo escribe así.

La resta de punteros da la distancia en elementos: si `q` apunta a `arr[3]` y `p` a `arr[0]`, `q - p = 3`. Útil para saber posiciones relativas.

Punteros y arreglos dinámicos de objetos

`new[]` también funciona con clases: llama al constructor de cada elemento.

```
class Punto { /* ... */ };
Punto* puntos = new Punto[10]; // 10 constructores por defecto
// ...
delete[] puntos; // 10 destructores
```

`delete[]` es obligatorio: llama a cada destructor. Con `delete` simple solo llamaría al primero y fugaría el resto. Otra razón para preferir `vector`, que hace todo esto solo.

Catálogo de pecados con punteros

Pecado 1: puntero sin inicializar. `int* p; *p = 5;` -> escribe en dirección basura. Penitencia: inicializa siempre, aunque sea a `nullptr`.

Pecado 2: puntero colgante (dangling). Usar memoria después de `delete`. Penitencia: `p = nullptr`; justo después de `delete`.

Pecado 3: doble liberación. `delete p; delete p;` -> comportamiento indefinido. Penitencia: `nullptr` después de `delete` (borrar `nullptr` es seguro y no hace nada).

Pecado 4: fugas. new sin delete en caminos con return o excepciones. Penitencia: punteros inteligentes o RAII.

Pecado 5: mezclar new[] con delete (sin corchetes). Corrompe el heap. Penitencia: empareja siempre new/delete y new[]/delete[].

Pecado 6: aritmética fuera de límites. p + 1000 en un arreglo de 10. Penitencia: verifica límites o usa contenedores.

Detectives de memoria: sanitizers

El compilador puede vigilar tu memoria en ejecución con sanitizers:

```
g++ -fsanitize=address -g programa.cpp -o programa
./programa
```

AddressSanitizer detecta: accesos fuera de límites, uso después de liberar, doble delete y fugas (con LeakSanitizer incluido). Ralentiza el programa, así que úsalo en desarrollo, no en producción. Es como tener un inspector de memoria mirando por encima de tu hombro: te dice exactamente dónde pecaste.

Más ejemplos

Ejemplo: mapa de la memoria en vivo

```

#include <iostream>

int global = 1; // segmento de datos

void funcion() {
    int local = 2; // pila
    int* heap = new int(3); // montón
    std::cout << "global: " << &global << std::endl;
    std::cout << "local:  " << &local << std::endl;
    std::cout << "heap:   " << heap << std::endl;
    delete heap;
}

int main() {
    funcion();
    return 0;
}

```

Ejecútalo: verás tres direcciones en rangos muy distintos. La global y la local están en zonas fijas del programa; la del heap viene de otra región. Repetir este experimento graba el modelo en tu cabeza.

Ejemplo: arreglo dinámico bidimensional


```

#include <iostream>

int main() {
    int filas = 3, columnas = 4;
    // Matriz dinámica: arreglo de punteros a filas
    int** matriz = new int*[filas];
    for (int i = 0; i < filas; i++) {
        matriz[i] = new int[columnas];
    }

    int valor = 1;
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            matriz[i][j] = valor++;
        }
    }

    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            std::cout << matriz[i][j] << "\t";
        }
        std::cout << std::endl;
    }

    for (int i = 0; i < filas; i++) {
        delete[] matriz[i]; // libera cada fila
    }
    delete[] matriz; // libera el arreglo de punteros
    return 0;
}

```

matriz[i][j] funciona con doble desreferencia. La liberación es en orden inverso: primero las filas, luego el contenedor. Es tedioso: en código moderno usarías `vector<vector<int>>`. Pero entenderlo te hace respetar lo que vector hace por ti.

Práctica adicional

Ejercicio A

Escribe un programa que reserve con `new` un arreglo de `N` doubles (`N` pedido al usuario), lo llene con la serie 1.5, 3.0, 4.5..., calcule el promedio y libere correctamente. Luego compílalo con `-fsanitize=address` y verifica que no reporte errores.

Solución:

```
#include <iostream>

int main() {
    int n;
    std::cout << "N: ";
    std::cin >> n;
    double* arr = new double[n];
    double suma = 0;
    for (int i = 0; i < n; i++) {
        arr[i] = 1.5 * (i + 1);
        suma += arr[i];
    }
    std::cout << "Promedio: " << suma / n << std::endl;
    delete[] arr;
    arr = nullptr;
    return 0;
}
```

Explicación: new[] con delete[] emparejados, nullptr después. El sanitizer confirmará que no hay fugas ni accesos inválidos. Acostúmbrate a verificar así tu código con memoria manual.

Ejercicio B

Provoca a propósito (en un programa de prueba) un uso-después-de-delete y observa lo que reporta AddressSanitizer. Luego corrígelo.

Solución:

```
#include <iostream>

int main() {
    int* p = new int(42);
    delete p;
    std::cout << *p << std::endl; // ERROR: uso después de liberar
    return 0;
}
```

Explicación: compila con -fsanitize=address -g y ejecuta: el sanitizer te muestra el archivo, la línea y el tipo exacto de error ("heap-use-after-free"). Esta es la forma profesional de cazar estos bugs: no a ojo, con herramientas.

Ejercicio C

Implementa una función `char* duplicarTexto(const char* original)` que reserve memoria suficiente, copie el texto y devuelva el puntero. El llamador debe liberarla. Documenta quién es responsable de cada `new/delete`.

Solución:

```
#include <iostream>
#include <cstring>

char* duplicarTexto(const char* original) {
    size_t len = std::strlen(original);
    char* copia = new char[len + 1]; // +1 para el terminador
    std::strcpy(copia, original);
    return copia; // el LLAMADOR debe hacer delete[]
}

int main() {
    char* texto = duplicarTexto("Hola, heap");
    std::cout << texto << std::endl;
    delete[] texto; // responsabilidad del llamador
    return 0;
}
```

Explicación: el +1 es clásico: olvidar el terminador nulo corrompe memoria. La documentación de "quién libera" es vital en interfaces con memoria manual: la ambigüedad aquí causa fugas o dobles deletes. En C++ moderno, esta función devolvería `std::string` y nadie liberaría nada.

Ejercicio D

Explica por qué este código es peligroso y corrígelo de dos formas (una con vector, otra manteniendo `new` pero segura):

```
int* crearArreglo(int n) {
    int datos[n]; // ¡arreglo local!
    return datos; // ¡devuelve dirección de la pila!
}
```

Solución: `datos` vive en la pila de la función; al retornar, el marco se destruye y el puntero queda colgando. El llamador leerá basura. Corrección 1 (moderna): devolver `std::vector<int>(n)`. Corrección 2 (manual): `int* datos = new int[n];` con la documentación de

que el llamador debe hacer delete[]. La lección: nunca devuelvas direcciones de variables locales. Esta tercera parte de la Unidad 5 sube el nivel de las herramientas que ya conoces. En las partes anteriores aprendiste los arreglos crudos, lo esencial de std::string y los punteros con la memoria dinámica. Aquí vas a modernizar todo eso: std::array como el reemplazo civilizado del arreglo crudo, strings usados como un profesional (concatenación eficiente, búsqueda avanzada, cirugía de texto), el kit completo de cctype, punteros a punteros sin miedo, las cuatro combinaciones de punteros con const, los argumentos de línea de comandos y el duelo entre qsort y sort. Todo lo que sigue es contenido nuevo: nada de lo que ya leíste se repite, solo se usa como base.

std::array: el arreglo crudo con modales

El arreglo crudo de C (int datos[10];) tiene tres defectos de fábrica que ya conoces: no sabe su propio tamaño, se convierte en puntero al pasarlo a funciones y no verifica límites. std::array, disponible en la cabecera <array> desde C++11, es un arreglo de tamaño fijo que conserva la velocidad del arreglo crudo pero se comporta como un objeto de verdad: sabe su tamaño, se puede copiar, se puede comparar y ofrece acceso con verificación de límites. Piensa en std::array como el arreglo crudo después de ir a la escuela de buenos modales.

La sintaxis incluye el tamaño como parte del tipo:

```
#include <array>
std::array<int, 5> numeros = {10, 20, 30, 40, 50};
```

Fíjate en los dos parámetros de la plantilla: el tipo de los elementos y la cantidad. Ese 5 no es un valor cualquiera: es parte del tipo. Un std::array<int, 5> y un std::array<int, 6> son tipos distintos, y el compilador lo sabe. Esto significa que el tamaño siempre se conoce en tiempo de compilación, lo cual permite optimizaciones y verificación estática.

Lo que std::array hace mejor que el arreglo crudo

Primero, conoce su tamaño con el método `size()`. Ya no necesitas pasar el tamaño por separado a las funciones ni definir constantes `N` regadas por el código. Segundo, se puede copiar con una simple asignación: `copia = original`; copia todos los elementos, algo imposible con arreglos crudos (donde el nombre del arreglo es un puntero y asignarlo solo copiaría la dirección). Tercero, se puede comparar con `==` y `!=`: dos arreglos son iguales si tienen los mismos elementos en el mismo orden. Cuarto, ofrece el método `at(i)`, que verifica el índice y lanza una excepción `std::out_of_range` si te sales, en vez de corromper memoria en silencio. Y quinto, al pasarlo a una función por referencia, no decae a puntero: la función recibe el arreglo completo con su tamaño incluido en el tipo.

Los métodos más útiles en el día a día son: `size()` para el tamaño, `at(i)` para acceso verificado, `operator[]` para acceso rápido sin verificación (igual que el arreglo crudo), `front()` para el primer elemento, `back()` para el último, `fill(valor)` para poner el mismo valor en todo el arreglo, `empty()` que en un array siempre devuelve `false` (el tamaño es fijo y mayor que cero en la práctica, aunque un `std::array<int, 0>` es legal), y `data()` que devuelve un puntero crudo al primer elemento para interoperar con código viejo que pide punteros.

Pasar `std::array` a funciones: por referencia, siempre

La forma profesional de recibir un `std::array` en una función es por referencia constante cuando solo lees, y por referencia normal cuando modificas:

```
double promedio(const std::array<double, 8>& notas) {
    double suma = 0.0;
    for (double n : notas) suma += n;
    return suma / notas.size();
}
```

Observa que el tamaño (8) forma parte del parámetro. Esto es una ventaja y una limitación a la vez: la función solo acepta arreglos de exactamente 8 elementos. Si necesitas una función que sirva para cualquier tamaño, tienes dos caminos: usar plantillas (lo verás en unidades avanzadas) o aceptar que `std::array` es para tamaños fijos conocidos. Para colecciones de tamaño variable en ejecución, el contenedor correcto es `std::vector`, que conocerás en la unidad 8. La regla práctica queda así: tamaño fijo y conocido al compilar -> `std::array`; tamaño variable o desconocido -> `std::vector`; interoperar con código C viejo -> arreglo crudo o puntero.

Ejemplo completo 1: std::array en un programa real

```
#include <iostream>
#include <array>
#include <stdexcept>

// Por referencia constante: no copia el arreglo y no decae a puntero.
double promedio(const std::array<double, 8>& notas) {
    double suma = 0.0;
    for (double n : notas) {
        suma += n;
    }
    return suma / notas.size();
}

int main() {
    std::array<double, 8> notas = {7.5, 8.0, 9.2, 6.8, 10.0, 8.7, 7.9, 9.5};

    std::cout << "Numero de notas: " << notas.size() << std::endl;
    std::cout << "Primera: " << notas.front() << ", ultima: " << notas.back() <<
std::endl;

    // at() verifica el indice y lanza una excepcion si te sales del rango.
    try {
        std::cout << "Nota 3: " << notas.at(2) << std::endl;
        std::cout << "Nota 20: " << notas.at(20) << std::endl; // fuera de rango
    } catch (const std::out_of_range& e) {
        std::cout << "Error atrapado: indice fuera de rango." << std::endl;
    }

    std::cout << "Promedio: " << promedio(notas) << std::endl;

    // Los array se copian y se comparan: los arreglos crudos no pueden.
    std::array<double, 8> copia = notas;
    if (copia == notas) {
        std::cout << "La copia es identica al original." << std::endl;
    }
    copia[0] = 0.0;
    if (copia != notas) {
        std::cout << "Despues de modificarla, ya son distintos." << std::endl;
    }

    // fill() pone el mismo valor en todo el arreglo de una vez.
    std::array<int, 5> ceros;
    ceros.fill(0);
    std::cout << "Arreglo de " << ceros.size() << " elementos llenado con ceros." <<
```

```
std::endl;

    return 0;
}
```

Recorre el ejemplo con calma. La función promedio recibe el arreglo por referencia constante: no hay copia de 8 doubles ni decaimiento a puntero, y el tamaño viaja con el tipo. El bloque try/catch muestra la diferencia entre `at()` y `operator[]`: con `at(20)` el programa no corrompe memoria ni se cae de forma misteriosa; lanza una excepción limpia que puedes atrapar y manejar. La comparación `copia == notas` es algo que con arreglos crudos requeriría un ciclo manual. Y `fill()` reemplaza el clásico for de inicialización. Si vienes de los arreglos crudos de la primera parte de la unidad, este ejemplo debería sentirse como cambiar un coche sin cinturón por uno con bolsas de aire: la misma velocidad, mucha más seguridad.

Limitaciones honestas de `std::array`

Ninguna herramienta es perfecta y un libro honesto dice dónde aprieta el zapato. Primera limitación: el tamaño debe ser una constante conocida al compilar. No puedes hacer `std::array<int, n>` con `n` leído del teclado; para eso está `std::vector`. Segunda: las funciones que reciben `std::array<int, 5>` no aceptan `std::array<int, 6>`; el tamaño es parte del tipo y no hay flexibilidad. Tercera: sigue siendo un arreglo contiguo de tamaño fijo, así que no puedes agregarle elementos en ejecución. Con estas tres reglas claras, sabrás exactamente cuándo `std::array` es la herramienta correcta y cuándo pedirle a `std::vector` que tome el relevo.

Strings a fondo: el texto como un profesional

En la primera parte de la unidad aprendiste lo esencial de `std::string`: crear, sumar con `+`, comparar con `==`, medir con `length()`, buscar con `find()` y extraer con `substr()`. Era lo necesario para empezar. Ahora viene el resto del arsenal: cómo concatenar sin desperdiciar memoria, cómo comparar con criterio, búsqueda avanzada con la familia `find`, cirugía de texto con `replace`, `insert` y `erase`, y la conversión entre números y strings. Si la primera parte te enseñó a hablar con strings, esta te enseña a discutir, negociar y operar con ellos.

Concatenación eficiente: el costo oculto del `+`

Cada vez que escribes `a + b` con strings, C++ crea un string nuevo, reserva memoria para el resultado y copia ambos operandos. Para dos o tres sumas no importa. Pero dentro de un ciclo con miles de iteraciones, ese costo se multiplica: cada `+` reserva memoria nueva y copia todo lo acumulado hasta el momento. El resultado es un programa que se vuelve lento de forma cuadrática sin que el código "se vea" lento.

La solución tiene tres piezas. Primera: el operador `+=`, que agrega al string existente en vez de crear uno nuevo. Segunda: el método `append()`, que es el hermano con más opciones de `+=` (puede agregar substrings, caracteres repetidos o rangos). Tercera: el método `reserve(n)`, que pide memoria por adelantado para `n` caracteres, de modo que las concatenaciones siguientes no tengan que reservar memoria una y otra vez.

```
std::string texto;
texto.reserve(1000); // pido memoria una sola vez
for (int i = 0; i < 100; i++) {
    texto += "linea "; // agrega sin crear strings temporales
    texto += std::to_string(i);
    texto += "\n";
}
```

La capacidad (`capacity()`) es la memoria reservada; el tamaño (`size()`) es lo que realmente usas. Cuando el tamaño supera la capacidad, el string reserva un bloque más grande (típicamente el doble) y copia todo: esa es la operación cara que `reserve()` evita. Regla práctica: si vas a concatenar en un ciclo y tienes una idea aproximada del tamaño final, llama a `reserve()` antes del ciclo. Es una sola línea que puede hacer tu programa varias veces más rápido.

Otra trampa clásica: concatenar muchos literales con `+` en una sola expresión, como `"a" + "b" + variable`. Los dos primeros operandos son `const char*`, no `std::string`, y el compilador se queja porque no sabe sumar dos punteros. La solución es que al menos uno de los primeros operandos sea `std::string`: `std::string("a") + "b" + variable`, o empezar la cadena con un string vacío. Es un error de compilación común que confunde a los principiantes porque "son solo textos".

Comparación de strings: más allá del `==`

Ya sabes que `==` compara contenido carácter por carácter. Pero los strings también soportan `<`, `>`, `<=` y `>=`, que comparan en orden lexicográfico (como en un diccionario, usando los códigos de los caracteres). Esto sirve para ordenar palabras alfabéticamente:


```
std::string a = "casa";
std::string b = "casco";
if (a < b) {
    std::cout << a << " va antes que " << b << std::endl; // "casa" < "casco"
}
```

"casa" va antes que "casco" porque en la cuarta letra, 'a' (97) es menor que 'o' (111). Ojo con un detalle: las mayúsculas van antes que las minúsculas en este orden ('Z' es 90, 'a' es 97), así que "Zapato" < "casa". Si necesitas comparar sin importar mayúsculas, convierte ambos a minúsculas primero (con el truco de `cctype` que verás más abajo) o escribe tu propia comparación.

El método `compare()` es la versión con esteroides de la comparación: devuelve un número negativo si el string es menor que el argumento, cero si son iguales y positivo si es mayor. Además permite comparar solo un fragmento: `s1.compare(pos, len, s2)` compara un pedazo de `s1` contra `s2` completo, y hay variantes para comparar pedazo contra pedazo. Es el método que usan las bibliotecas cuando necesitan comparaciones finas, por ejemplo al implementar diccionarios o índices.

Búsqueda avanzada: toda la familia `find`

Ya conoces `find(texto)`, que devuelve la posición de la primera aparición o `std::string::npos` si no hay. La familia completa te da control total sobre la búsqueda:

- `find(sub, pos)`: busca `sub` empezando en la posición `pos`. Ideal para encontrar "la siguiente aparición" en un ciclo.
- `rfind(sub)`: busca desde el final hacia atrás. Perfecto para encontrar la última aparición, como la última diagonal en una ruta de archivo.
- `find_first_of(conjunto)`: encuentra el primer carácter que pertenezca al conjunto. Por ejemplo, `texto.find_first_of("aeiou")` encuentra la primera vocal.
- `find_last_of(conjunto)`: lo mismo pero desde el final.
- `find_first_not_of(conjunto)`: encuentra el primer carácter que NO esté en el conjunto. Útil para saltar espacios en blanco al inicio: `texto.find_first_not_of(" \t")`.
- `find_last_not_of(conjunto)`: el último carácter que no está en el conjunto. Combinado con el anterior, recorta espacios de ambos extremos.

Todas devuelven `std::string::npos` cuando no encuentran nada, así que la comparación `if (pos != std::string::npos)` es tu mejor amiga. Un detalle importante: `npos` no es -1 aunque a veces se

vea así; es un valor especial del tipo `size_type`. Compáralo siempre contra `std::string::npos`, nunca contra `-1` ni contra un número mágico.

Un patrón clásico con `find` en ciclo es procesar todas las apariciones de algo:

```
std::string texto = "uno dos uno tres uno";
std::string::size_type pos = 0;
while ((pos = texto.find("uno", pos)) != std::string::npos) {
    std::cout << "Encontrado en: " << pos << std::endl;
    pos += 3; // avanzo más allá de lo encontrado para no repetirme
}
```

El segundo argumento de `find` (la posición inicial) es lo que permite avanzar por el texto. Sin él, el ciclo encontraría la misma aparición infinitamente. Este patrón aparece en censores de palabras, resaltadores de sintaxis y cualquier programa que procese texto por partes.

Cirugía de strings: `substr`, `replace`, `insert` y `erase`

Ya viste `substr(pos, len)`. Sus compañeros de quirófano son:

- `replace(pos, len, nuevo)`: sustituye `len` caracteres desde `pos` por el string `nuevo`. Es la base de los "buscar y reemplazar".
- `insert(pos, texto)`: inserta `texto` en la posición `pos` sin borrar nada.
- `erase(pos, len)`: borra `len` caracteres desde `pos`. Con `erase(pos)` solo, borra hasta el final.
- `clear()`: vacía el string completo.

Combinados con `find`, resuelven problemas reales. El ejemplo clásico es reemplazar todas las apariciones de una palabra:

```
std::string mensaje = "El gato persigue al gato.";
std::string::size_type pos = 0;
while ((pos = mensaje.find("gato", pos)) != std::string::npos) {
    mensaje.replace(pos, 4, "perro");
    pos += 5; // avanzo lo que mide "perro" para seguir buscando después
}
```

Cuidado con el detalle del avance: después de reemplazar "gato" (4 letras) por "perro" (5 letras), hay que avanzar 5 posiciones, no 4. Si el reemplazo contiene a la palabra buscada (reemplazar "a" por "aa"), avanzar mal produce un ciclo infinito. Es uno de esos detalles que solo se aprenden escribiendo el ciclo una vez y viéndolo colgarse.

Para recortar espacios en blanco de ambos extremos de un string (el famoso "trim" que C++ no trae de fábrica), la combinación es:

```
std::string s = "  hola mundo  ";
std::string::size_type inicio = s.find_first_not_of(" \t\n");
std::string::size_type fin = s.find_last_not_of(" \t\n");
if (inicio != std::string::npos) {
    s = s.substr(inicio, fin - inicio + 1);
} else {
    s.clear(); // el string era puro espacio
}
```

Memoriza este patrón: aparece en validación de formularios, lectura de archivos de configuración y limpieza de entrada del usuario.

Números y strings: to_string, stoi y compañía

Convertir números a string es fácil con std::to_string(valor), que funciona con int, long, double y compañía:

```
int edad = 30;
std::string texto = "Tienes " + std::to_string(edad) + " años.";
```

El camino inverso lo cubre la familia sto* (string to...): stoi (a int), stol (a long), stoul (a unsigned long), stoll (a long long), stof (a float) y stod (a double). Todas lanzan excepciones si el texto no es un número válido: std::invalid_argument si no hay nada convertible y std::out_of_range si el número no cabe en el tipo. En programas que leen entrada del usuario, vale la pena atraparlas:

```
try {
    int n = std::stoi(entrada);
    std::cout << "Doble: " << n * 2 << std::endl;
} catch (const std::invalid_argument&) {
    std::cout << "Eso no era un numero." << std::endl;
} catch (const std::out_of_range&) {
    std::cout << "Numero demasiado grande." << std::endl;
}
```

Las funciones sto* tienen un segundo parámetro opcional donde devuelven cuántos caracteres consumieron, útil para parsear "123abc" y saber que el número terminaba en la posición 3. Y si necesitas formato fino de decimales (por ejemplo, siempre dos decimales), to_string no te deja

controlar: para eso se usa `std::ostringstream` o la biblioteca `<format>` de C++20, temas de unidades posteriores.

Ejemplo completo 2: analizador de bitácora

Este programa procesa líneas de un registro (log), cuenta los errores, arma un reporte con concatenación eficiente y censura una palabra sensible. Usa `find` con posición inicial, `substr`, `replace`, `+=` y `reserve` en un solo programa coherente.

```
#include <iostream>
#include <string>
#include <array>

struct Entrada {
    std::string fecha;
    std::string nivel;
    std::string mensaje;
};

// Separa "2026-09-19 ERROR disco lleno" en fecha, nivel y mensaje.
// Devuelve false si la linea no tiene el formato esperado.
bool parsear(const std::string& linea, Entrada& e) {
    std::string::size_type p1 = linea.find(' ');
    if (p1 == std::string::npos) return false;
    std::string::size_type p2 = linea.find(' ', p1 + 1);
    if (p2 == std::string::npos) return false;

    e.fecha = linea.substr(0, p1);
    e.nivel = linea.substr(p1 + 1, p2 - p1 - 1);
    e.mensaje = linea.substr(p2 + 1);
    return true;
}

// Reemplaza todas las apariciones de "buscada" por "reemplazo".
void reemplazarTodo(std::string& texto, const std::string& buscada,
                    const std::string& reemplazo) {
    std::string::size_type pos = 0;
    while ((pos = texto.find(buscada, pos)) != std::string::npos) {
        texto.replace(pos, buscada.length(), reemplazo);
        pos += reemplazo.length();
    }
}

int main() {
    std::array<std::string, 5> bitacora = {
```

```

        "2026-09-19 INFO sistema iniciado",
        "2026-09-19 ERROR disco lleno",
        "2026-09-19 INFO respaldo completo",
        "2026-09-19 ERROR red caída",
        "2026-09-19 INFO sistema apagado"
    };

    std::string reporte;
    reporte.reserve(400); // evito reservas repetidas en el ciclo
    int errores = 0;

    for (const std::string& linea : bitacora) {
        Entrada e;
        if (!parsear(linea, e)) continue; // linea malformada: se ignora
        if (e.nivel == "ERROR") {
            errores++;
            reporte += "[ALERTA] ";
            reporte += e.fecha;
            reporte += ": ";
            reporte += e.mensaje;
            reporte += "\n";
        }
    }

    reemplazarTodo(reporte, "disco", "*****"); // censura dato sensible

    std::cout << "Errores encontrados: " << errores << "\n\n";
    std::cout << reporte;
    return 0;
}

```

Observa las decisiones: `parsear()` usa `find` con posición inicial para localizar los dos espacios separadores y `substr` para extraer cada campo; si falta algún espacio, devuelve `false` y el `main` ignora la línea (programación defensiva con datos reales). El reporte se arma con `+=` dentro del ciclo y un `reserve()` previo, el patrón eficiente que viste arriba. La censura usa el ciclo `find/replace` con el avance correcto (`reemplazo.length()`) para no caer en ciclos infinitos. Este programa es pequeño pero contiene los patrones que verás en procesadores de logs, filtros de chat y herramientas de análisis de texto de verdad.

cctype: el kit completo para clasificar caracteres

En la primera parte usaste `std::tolower` de pasada. La cabecera `<cctype>` trae una familia completa de funciones que responden preguntas sobre caracteres individuales: ¿es letra?, ¿es dígito?, ¿es espacio?, ¿es mayúscula? Son las herramientas con las que se construyen validadores, analizadores y limpiadores de texto. Todas reciben un `int` (el carácter) y devuelven distinto de cero si la respuesta es sí.

La tabla completa de clasificación

- `isalpha(c)`: ¿es letra? (a-z, A-Z).
- `isdigit(c)`: ¿es dígito decimal? (0-9).
- `isalnum(c)`: ¿es letra o dígito?
- `isspace(c)`: ¿es espacio en blanco? (espacio, tab, salto de línea, etc.).
- `isupper(c)`: ¿es mayúscula?
- `islower(c)`: ¿es minúscula?
- `ispunct(c)`: ¿es signo de puntuación? (!, ., ,, ;, etc.).
- `isxdigit(c)`: ¿es dígito hexadecimal? (0-9, a-f, A-F).
- `iscntrl(c)`: ¿es carácter de control? (como `\n` o `\t`).
- `isprint(c)`: ¿es imprimible? (incluye el espacio).
- `isgraph(c)`: ¿es imprimible pero no espacio?

Y las dos de conversión: `toupper(c)` devuelve la mayúscula equivalente (o el mismo carácter si no tiene), y `tolower(c)` la minúscula equivalente.

Un detalle técnico que evita bugs raros: estas funciones esperan un valor representable como `unsigned char` o el valor especial EOF. Si tu `char` es negativo (pasa con letras acentuadas en algunas configuraciones), el comportamiento es indefinido. La práctica segura es convertir: `std::isalpha(static_cast<unsigned char>(c))`. Parece ceremonioso, pero es lo que hace el código profesional y te salva de fallos que solo aparecen en la computadora del cliente.

Trucos prácticos con cctype

Con estas funciones, tareas comunes se vuelven ciclos simples. Validar que un string sea un número entero positivo: recorrer y pedir que todo sea `isdigit`. Contar cuántas mayúsculas tiene una contraseña: `isupper` en ciclo. Convertir todo un string a mayúsculas:

```
std::string texto = "hola Mundo 123";
for (char& c : texto) {
    c = static_cast<char>(std::toupper(static_cast<unsigned char>(c)));
}
```

El `char&` con referencia es esencial: sin él modificarías copias y el string quedaría igual (este truco ya lo viste en la unidad 6 con `aMayusculas`, aquí lo ves aplicado con el cast seguro). Otro clásico: quitar todos los espacios de un string construyendo uno nuevo solo con los caracteres que no son espacio:

```
std::string original = "h o l a";
std::string limpio;
limpio.reserve(original.size());
for (char c : original) {
    if (!std::isspace(static_cast<unsigned char>(c))) {
        limpio += c;
    }
}
```

Ejemplo: validador de contraseñas

Las reglas de contraseñas ("mínimo 8 caracteres, una mayúscula, una minúscula, un dígito y un símbolo") son exactamente un ejercicio de `cctype`. Este programa las verifica todas y dice qué falta, en vez de un simple sí/no.

```

#include <iostream>
#include <string>
#include <cctype>

struct Reporte {
    bool largo = false;
    bool mayus = false;
    bool minus = false;
    bool digito = false;
    bool simbolo = false;
    bool valida() const { return largo && mayus && minus && digito && simbolo; }
};

Reporte validar(const std::string& clave) {
    Reporte r;
    r.largo = clave.length() >= 8;
    for (char c : clave) {
        unsigned char u = static_cast<unsigned char>(c);
        if (std::isupper(u)) r.mayus = true;
        else if (std::islower(u)) r.minus = true;
        else if (std::isdigit(u)) r.digito = true;
        else if (std::ispunct(u)) r.simbolo = true;
    }
    return r;
}

int main() {
    std::string clave;
    std::cout << "Nueva clave: ";
    std::getline(std::cin, clave);

    Reporte r = validar(clave);
    if (r.valida()) {
        std::cout << "Clave aceptada." << std::endl;
    } else {
        std::cout << "Clave debil, te falta:" << std::endl;
        if (!r.largo) std::cout << " - al menos 8 caracteres" << std::endl;
        if (!r.mayus) std::cout << " - una mayuscula" << std::endl;
        if (!r.minus) std::cout << " - una minuscula" << std::endl;
        if (!r.digito) std::cout << " - un digito" << std::endl;
        if (!r.simbolo) std::cout << " - un simbolo (!, #, etc.)" << std::endl;
    }
    return 0;
}

```


El struct Reporte junta los cinco criterios y el método valida() los resume. La función validar() hace una sola pasada por la clave: eficiente y clara. Nota que el largo se verifica fuera del ciclo, porque es una propiedad del string completo, no de cada carácter. Este patrón (una pasada, varias banderas) aparece en validadores de formularios, filtros de contenido y análisis de texto en general.

Punteros a punteros: direcciones de direcciones

Un puntero guarda la dirección de una variable. Un puntero a puntero guarda la dirección de un puntero. Se declara con dos asteriscos: `int pp;`. Si `int* p = &x`, entonces `int pp = &p`. Y para llegar al valor hay que desreferenciar dos veces: **pp es x. La idea se extiende (int*** y más), pero en la práctica dos niveles cubren casi todo; tres niveles suelen ser señal de que el diseño necesita simplificarse.**

Dibuja el mapa mental: x es una caja con el valor 10. p es un papelito con la dirección de la caja de x. pp es un papelito con la dirección del papelito p. Seguir pp una vez (*pp) te lleva al papelito p; seguirlo dos veces (pp) te lleva a la caja de x. **Cada nivel de * quita una capa de indirección.**

El uso estrella: modificar un puntero dentro de una función

Ya sabes que para modificar un int dentro de una función le pasas int*. La misma lógica sube un nivel: para modificar un puntero (int*) dentro de una función, le pasas su dirección, que es un int. **El caso típico es una función que reserva memoria y debe entregar el puntero resultante al llamador:**

```

bool crearArreglo(int** pp, int n) {
    if (pp == nullptr || n <= 0) return false;
    *pp = new int[n];          // modifico el puntero del llamador
    for (int i = 0; i < n; i++) {
        (*pp)[i] = (i + 1) * 10;
    }
    return true;
}

int main() {
    int* datos = nullptr;
    if (crearArreglo(&datos, 5)) { // paso la dirección del puntero
        for (int i = 0; i < 5; i++) std::cout << datos[i] << " ";
        std::cout << std::endl;
        delete[] datos;
        datos = nullptr;
    }
    return 0;
}

```

Sin el doble puntero, crearArreglo recibiría una copia del puntero y el new se perdería al salir de la función (una fuga instantánea). Con `int pp`, la asignación `*pp = new int[n]` escribe directamente en la variable `datos` del `main`. Nota los paréntesis en `(*pp)[i]`: son obligatorios porque el operador `[]` tiene mayor precedencia que `*`: `*pp[i]` significaría `*(pp[i])`, algo muy distinto y erróneo aquí. Este es uno de esos detalles de precedencia que muerden una sola vez, pero muerden fuerte.

En C++ moderno, este patrón se escribe más limpio devolviendo el puntero o usando una referencia a puntero (`int*&`), que verás en la unidad 6. Pero el puntero a puntero sigue apareciendo en interfaces estilo C, en matrices dinámicas y en `argv`, así que debes leerlo con fluidez aunque no sea tu primera opción para escribirlo.

Arreglos de punteros: una colección de direcciones

Un arreglo de punteros es exactamente lo que dice: cada casilla guarda una dirección. La declaración `int* lista[4]`; crea 4 punteros a `int`. Sirve para colecciones donde los elementos viven en lugares distintos o tienen tamaños distintos, como un arreglo de cadenas de texto de diferente longitud:

```
const char* dias[] = {"lunes", "martes", "miercoles", "jueves", "viernes"};
for (int i = 0; i < 5; i++) {
    std::cout << dias[i] << std::endl;
}
```

Cada `dias[i]` apunta al primer carácter de un literal distinto. Los literales tienen tamaños diferentes, pero como solo guardamos punteros, el arreglo es uniforme: 5 direcciones. Esta es la estructura exacta de `argv`, que verás en la siguiente sección. También sirve para "vistas" sobre datos existentes: un arreglo de punteros puede apuntar a elementos de otro arreglo en un orden distinto, creando un reordenamiento sin mover los datos originales.

Ejemplo completo 3: redimensionar un arreglo con puntero doble

Este programa muestra el patrón completo: una función que agranda un arreglo dinámico. Necesita el puntero doble porque debe reemplazar el puntero del llamador por uno nuevo, copiar los datos viejos y liberar la memoria anterior.

```

#include <iostream>

// Agrandar el arreglo apuntado por *pp de 'actual' a 'nuevo' elementos.
// Los elementos existentes se conservan; los nuevos quedan en cero.
bool redimensionar(int** pp, int actual, int nuevo) {
    if (pp == nullptr || *pp == nullptr || nuevo <= actual) return false;

    int* ampliado = new int[nuevo];
    for (int i = 0; i < actual; i++) {
        ampliado[i] = (*pp)[i]; // copio los datos viejos
    }
    for (int i = actual; i < nuevo; i++) {
        ampliado[i] = 0; // los nuevos en cero
    }

    delete[] *pp; // libero la memoria vieja
    *pp = ampliado; // el llamador ahora apunta a la memoria nueva
    return true;
}

void mostrar(int* arr, int n) {
    for (int i = 0; i < n; i++) std::cout << arr[i] << " ";
    std::cout << std::endl;
}

int main() {
    int* datos = new int[3]{10, 20, 30};
    std::cout << "Original: ";
    mostrar(datos, 3);

    if (redimensionar(&datos, 3, 6)) {
        std::cout << "Ampliado: ";
        mostrar(datos, 6);
    }

    delete[] datos;
    datos = nullptr;
    return 0;
}

```

Sigue el flujo: redimensionar recibe &datos (un int), **reserva el arreglo nuevo, copia los 3 valores viejos, pone ceros en los 3 nuevos, libera el arreglo viejo y escribe el puntero nuevo en *pp, que es la variable datos del main. Si la función recibiera solo int*, el *pp = ampliado final modificaría una copia y el main seguiría apuntando a memoria liberada:**

un desastre. Este es exactamente el trabajo que `std::vector` hace por ti internamente (por eso existe), pero verlo manual te hace entender qué paga vector con su comodidad: nada, porque lo hace igual de bien y sin tus bugs.

Punteros y const: las cuatro combinaciones

El const con punteros confunde porque hay dos cosas que pueden ser constantes: el puntero (la dirección guardada) y el dato apuntado (el valor en esa dirección). Dos opciones por cada lado dan cuatro combinaciones, y cada una dice algo distinto. Dominarlas es obligatorio para leer código profesional, donde const aparece por todas partes.

La tabla de las cuatro combinaciones

Declaración	¿Cambia el dato?	¿Cambia el puntero?	Se lee como
<code>int* p</code>	Sí	No	puntero a entero
<code>const int* p</code>	No	No	puntero a entero constante
<code>int* const p</code>	Sí	No	puntero constante a entero
<code>const int* const p</code>	No	No	puntero constante a entero constante

La clave para leerlas es que const modifica lo que tiene inmediatamente a su izquierda; si no hay nada a la izquierda, modifica lo que tiene a la derecha. Así, en `const int* p`, el const está antes de int: el entero es constante. En `int* const p`, el const está después del *: el puntero es constante. En `const int* const p` hay dos const: ambos lados constantes. (La forma `int const* p` es equivalente a `const int* p`; verás ambas en código ajeno.)

Otro truco de lectura: lee la declaración de derecha a izquierda. "`int* const p`" se lee "p es un const puntero a int": puntero constante a entero. "`const int* p`" se lee "p es un puntero a int const": puntero a entero constante. Con práctica, esto se vuelve automático.

Por qué importa cada combinación

- `int* p`: el puntero de trabajo diario, lectura y escritura total.
- `const int* p` (puntero a const): la forma de decir "te presto mis datos para leer, no para modificar". Es lo que deberías usar en parámetros de función que solo leen: `void mostrar(const int* datos, int n)`. El llamador tiene la garantía de que sus datos no cambiarán.
- `int* const p` (puntero const): el puntero siempre apunta al mismo lugar, pero el dato se puede modificar. Aparece menos, pero es útil cuando la dirección es fija por diseño, como un puntero a un registro de hardware o a un buffer que nunca se reubica.

- `const int* const p`: ni la dirección ni el dato cambian. Típico en tablas de configuración de solo lectura.

Un detalle que atrapa a muchos: puedes asignar un `int*` a un `const int*` (agregar `const` es seguro: estás prometiendo modificar menos), pero no al revés. Quitar `const` requiere un cast explícito (`const_cast`) y modificar un dato que era originalmente `const` por esa vía es comportamiento indefinido. El compilador te protege; no luches contra él.

Ejemplo: las cuatro combinaciones en acción

```
#include <iostream>

void mostrar(const int* datos, int n) {
    // *datos = 0; // ERROR: la promesa const lo impide
    for (int i = 0; i < n; i++) {
        std::cout << datos[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    int a = 10, b = 20;

    int* p1 = &a;           // 1) todo modificable
    const int* p2 = &a;      // 2) dato protegido
    int* const p3 = &a;      // 3) puntero fijo
    const int* const p4 = &a; // 4) todo protegido

    *p1 = 11; p1 = &b;      // OK, OK
    // *p2 = 12;           // ERROR: dato constante
    p2 = &b;                // OK: el puntero si se mueve
    *p3 = 13;               // OK: el dato si cambia
    // p3 = &b;             // ERROR: puntero constante
    // *p4 = 14; p4 = &b;   // ERROR y ERROR

    int valores[] = {1, 2, 3, 4};
    mostrar(valores, 4); // int* se convierte solo a const int*

    std::cout << "a vale: " << a << std::endl;
    return 0;
}
```

Descomenta mentalmente cada línea marcada ERROR y verifica contra la tabla: cada error corresponde exactamente a intentar cambiar algo declarado const. La función mostrar() demuestra el uso profesional: recibe const int* porque solo lee, y el llamador le pasa un int* normal sin problemas (agregar const es una conversión segura e implícita). Si mostrar() intentara modificar, el compilador lo atraparía antes de que el bug llegara a ejecución. Ese es el poder del const: convierte errores de ejecución en errores de compilación, que son infinitamente más baratos.

Argumentos de línea de comandos: el arreglo de punteros más famoso

Todo programa de C++ puede recibir argumentos desde la terminal. El main tiene una firma especial para eso:

```
int main(int argc, char* argv[]) {
```

- argc (argument count): cuántos argumentos llegaron, incluyendo el nombre del programa. Si ejecutas ./suma 10 20, argc vale 3.
- argv (argument vector): un arreglo de punteros a char, uno por argumento. argv[0] es el nombre del programa ("./suma"), argv[1] es "10", argv[2] es "20". El arreglo termina con un nullptr en argv[argc], por si prefieres recorrerlo sin contador.

Fíjate en el tipo: char* argv[] es un arreglo de punteros, exactamente la estructura que estudiaste en la sección de punteros a punteros. Cada argv[i] apunta al primer carácter de un texto con el argumento. Los argumentos siempre llegan como texto, aunque representen números: "10" son los caracteres '1' y '0', no el entero 10. Para convertirlos usas las funciones stoi* de la sección de strings: std::stoi(argv[1]) o std::stod(argv[2]) (el char* se convierte solo a std::string temporal).

Ejemplo: calculadora de línea de comandos

```

#include <iostream>
#include <string>

int main(int argc, char* argv[]) {
    if (argc < 4) {
        std::cout << "Uso: " << argv[0] << " <num1> <operacion> <num2>" <<
std::endl;
        std::cout << "Ejemplo: " << argv[0] << " 10 + 20" << std::endl;
        return 1; // código de error: argumentos insuficientes
    }

    double a, b;
    try {
        a = std::stod(argv[1]);
        b = std::stod(argv[3]);
    } catch (...) {
        std::cout << "Error: los numeros no son validos." << std::endl;
        return 1;
    }

    std::string op = argv[2];
    double resultado = 0;
    bool ok = true;

    if (op == "+") resultado = a + b;
    else if (op == "-") resultado = a - b;
    else if (op == "x") resultado = a * b;
    else if (op == "/") {
        if (b == 0) { std::cout << "Error: division entre cero." << std::endl;
return 1; }
        resultado = a / b;
    } else ok = false;

    if (!ok) {
        std::cout << "Operacion desconocida. Usa + - x /" << std::endl;
        return 1;
    }

    std::cout << a << " " << op << " " << b << " = " << resultado << std::endl;
    return 0;
}

```

Compíllalo y pruébalo: ./calculadora 10 + 20, ./calculadora 7 x 6, ./calculadora sin argumentos. Observa los hábitos profesionales: el programa valida argc antes de tocar argv (acceder a

argv[3] con argc < 4 leería basura), muestra el uso correcto incluyendo argv[0] (así el mensaje siempre dice el nombre real del programa), convierte con stod dentro de try/catch y devuelve códigos de salida distintos de cero cuando algo falla (los scripts y otros programas leen ese código). Nota el detalle del *: en la terminal el asterisco es un comodín del shell, por eso el programa usa "x" para multiplicar. Son los roces con el mundo real que solo aparecen cuando sales del IDE.

Un patrón frecuente es recorrer todos los argumentos con un for de i = 1 a argc - 1, como una lista de archivos a procesar o números a sumar. Y una advertencia: argv solo existe durante el main; guardar esos punteros para usarlos después de que main termine no tiene sentido, y modificar los textos de argv es legal pero de mala educación (otros códigos asumen que no cambian).

qsort contra sort: el duelo del ordenamiento

C++ heredó de C la función qsort (cabecera <cstdlib>) y ofrece su propia std::sort (cabecera <algorithm>). Ambas ordenan, pero pertenecen a mundos distintos. Entender las dos te enseña por qué el C++ moderno prefiere tipos seguros sobre trucos genéricos.

qsort: el veterano de C

```
#include <cstdlib>

int compararInts(const void* a, const void* b) {
    int x = *static_cast<const int*>(a);
    int y = *static_cast<const int*>(b);
    return (x > y) - (x < y); // 1 si x>y, -1 si x<y, 0 si iguales
}

int datos[] = {40, 10, 30, 20, 50};
std::qsort(datos, 5, sizeof(int), compararInts);
```

qsort recibe: el puntero al inicio, cuántos elementos, el tamaño de cada uno y un comparador. Como C no tiene plantillas, el comparador recibe void*: punteros sin tipo que debes convertir con static_cast antes de usar. El comparador devuelve negativo si el primero va antes, cero si empatan y positivo si va después. El truco (x > y) - (x < y) produce esos tres valores sin ramas. Funciona, pero es frágil: si te equivocas en el sizeof o en el cast, el compilador no te avisa y el ordenamiento produce basura silenciosa. Además, la llamada al comparador es indirecta (por puntero a función), lo que impide ciertas optimizaciones.

std::sort: el campeón de C++

```
#include <algorithm>

int datos[] = {40, 10, 30, 20, 50};
std::sort(std::begin(datos), std::end(datos)); // ascendente

std::sort(std::begin(datos), std::end(datos),
          [](int a, int b) { return a > b; }); // descendente con lambda
```

std::sort recibe dos iteradores (inicio y fin) y un comparador opcional. Sin comparador ordena ascendente con <. Con una lambda defines el criterio en el lugar, con tipos correctos verificados por el compilador: si comparas peras con manzanas, no compila. La lambda [](int a, int b) { return a > b; } dice "a va antes que b si a es mayor", produciendo orden descendente. Como sort es una plantilla, el compilador genera código específico para tu tipo y tu comparador, y suele ser más rápido que qsort.

Veredicto comparativo

| Aspecto | qsort | std::sort | |---|---|---| | Tipos | void*, sin verificación | Plantilla, verificación total | | Comparador | Función separada con casts | Lambda en el lugar, tipos seguros | | Velocidad | Buena | Típicamente mejor (optimiza el comparador) | | Errores típicos | sizeof o cast mal -> basura silenciosa | El compilador te avisa | | Cuándo usarlo | Código C puro o APIs que lo exigen | Todo lo demás |

La moraleja va más allá del ordenamiento: qsort representa la era donde la genericidad se lograba borrando los tipos (void*), y std::sort la era donde se logra con plantillas que conservan los tipos. Cada vez que dudes entre una solución "a la C" y una "a la C++", pregunta cuál te da más ayuda del compilador. Casi siempre gana la segunda.

Ejemplo: el duelo en código

```

#include <iostream>
#include <cstdlib>    // qsort
#include <algorithm> // sort
#include <string>

int compararInts(const void* a, const void* b) {
    int x = *static_cast<const int*>(a);
    int y = *static_cast<const int*>(b);
    return (x > y) - (x < y);
}

void mostrar(int* arr, int n, const std::string& titulo) {
    std::cout << titulo;
    for (int i = 0; i < n; i++) std::cout << arr[i] << " ";
    std::cout << std::endl;
}

int main() {
    int datos1[] = {40, 10, 30, 20, 50};
    int datos2[] = {40, 10, 30, 20, 50};

    std::qsort(datos1, 5, sizeof(int), compararInts);
    mostrar(datos1, 5, "qsort ascendente: ");

    std::sort(std::begin(datos2), std::end(datos2));
    mostrar(datos2, 5, "sort ascendente: ");

    std::sort(std::begin(datos2), std::end(datos2),
              [](int a, int b) { return a > b; });
    mostrar(datos2, 5, "sort descendente: ");

    // Ordenar strings: con sort es trivial, con qsort un dolor de casts
    std::string palabras[] = {"pera", "manzana", "uva", "platano"};
    std::sort(std::begin(palabras), std::end(palabras));
    std::cout << "Palabras ordenadas: ";
    for (const auto& p : palabras) std::cout << p << " ";
    std::cout << std::endl;

    return 0;
}

```

El ejemplo cierra el argumento: ordenar strings con `std::sort` es una línea que funciona con el operador `<` de `std::string`; hacerlo con `qsort` exigiría un comparador que convierta `void*` a `std::string*` con casts manuales, frágil y verboso. Cuando la herramienta moderna hace en una línea lo que la vieja hace en veinte líneas peligrosas, la elección está clara.

Catálogo de errores comunes (tercera parte)

Error 1: usar `std::array` con tamaño variable. `std::array<int, n>` donde `n` se lee del teclado no compila: el tamaño debe ser constante en compilación. Si el tamaño se conoce hasta la ejecución, usa `std::vector` (unidad 8) o memoria dinámica.

Error 2: pasar `std::array` por valor a una función. Compila y funciona, pero copia todo el arreglo en cada llamada. Para arreglos grandes en funciones llamadas miles de veces, es un desperdicio silencioso. Usa `const std::array<T, N>&` para lectura.

Error 3: confundir `size()` con `capacity()` en strings. Después de `reserve(1000)`, `size()` sigue siendo 0 hasta que agregues caracteres. Iterar hasta `capacity()` lee basura. El tamaño real siempre es `size()` o `length()`.

Error 4: concatenar con `+` dentro de un ciclo sin `reserve()`. Funciona, pero cada iteración puede reservar memoria y copiar todo lo acumulado: costo cuadrático. Con pocos cientos de iteraciones no lo notas; con cientos de miles, el programa se arrastra. El patrón es: `reserve()` antes del ciclo, `+=` dentro.

Error 5: sumar literales con `"a" + "b"`. Dos `const char*` no se pueden sumar con `+`; el compilador protesta con un error críptico. Al menos un operando debe ser `std::string`: `std::string("a") + "b"`.

Error 6: comparar el resultado de `find()` con `-1` o con `0` en vez de `std::string::npos`. `find` devuelve `size_type` (sin signo); comparar con `-1` puede comportarse raro y comparar con `0` confunde "encontrado al inicio" con "no encontrado". Siempre: `if (pos != std::string::npos)`.

Error 7: olvidar avanzar la posición en un ciclo `find/replace`. Si después de `replace` no sumas el largo del reemplazo a `pos`, el ciclo puede encontrar lo mismo eternamente o procesar el texto recién insertado. Avanza siempre `pos += reemplazo.length()`.

Error 8: pasar `char` con signo directo a funciones de `<cctype>`. Con caracteres extendidos (acentos en ciertas configuraciones) el valor puede ser negativo y el comportamiento es indefinido. Acostúmbrate al cast: `std::isalpha(static_cast<unsigned char>(c))`.

Error 9: confundir los niveles de indirección en punteros dobles. `*pp` es el puntero, **pp es el dato**. Escribir `*pp = 5` cuando querías `pp = 5` cambia la dirección en vez del valor: el puntero

queda apuntando a la dirección 5 y el siguiente uso se cae. Y recuerda los paréntesis: `(*pp)[i]` no es lo mismo que `*pp[i]`.

Error 10: leer mal las declaraciones con `const`. `const int* p` protege el dato, `int* const p` fija el puntero. Si los confundes, el compilador te dará errores en líneas que "se ven bien". Lee de derecha a izquierda hasta que salga natural.

Error 11: acceder a `argv[i]` sin verificar `argc`. Si el usuario ejecuta el programa sin argumentos y tu código lee `argv[1]`, lee memoria inválida. Verifica siempre `argc` antes de tocar `argv`, y muestra el uso correcto con `argv[0]`.

Error 12: usar el `sizeof` equivocado en `qsort`. `qsort(datos, 5, sizeof(double), compararInts)` con un arreglo de `int` ordena basura sin avisar: `qsort` confía ciegamente en el tamaño que le das. Otra razón para preferir `std::sort`, donde este error no existe.

Error 13: asumir que los argumentos de `argv` son números. `argv` siempre entrega texto; "10" no es el entero 10 hasta que lo conviertes con `stoi/stod`. Y esas conversiones lanzan excepciones con entrada inválida: no las llares a ciegas con lo que escribió el usuario.

Práctica: ejercicios con solución

Ejercicio 1: estadísticas con `std::array`

Pide al usuario las temperaturas de 7 días, guárdalas en un `std::array<double, 7>` y muestra: el promedio, la máxima y la mínima. Usa `size()`, `front()`, `back()` y `at()` donde tenga sentido, y pasa el arreglo a una función por referencia constante.

Solución:

```

#include <iostream>
#include <array>

void estadisticas(const std::array<double, 7>& temps) {
    double suma = 0.0;
    double maximo = temps.front();
    double minimo = temps.front();

    for (std::size_t i = 0; i < temps.size(); i++) {
        double t = temps.at(i); // acceso verificado
        suma += t;
        if (t > maximo) maximo = t;
        if (t < minimo) minimo = t;
    }

    std::cout << "Promedio: " << suma / temps.size() << std::endl;
    std::cout << "Maxima: " << maximo << ", minima: " << minimo << std::endl;
    std::cout << "Ultima registrada: " << temps.back() << std::endl;
}

int main() {
    std::array<double, 7> temps;
    for (std::size_t i = 0; i < temps.size(); i++) {
        std::cout << "Temperatura del dia " << i + 1 << ": ";
        std::cin >> temps[i];
    }
    estadisticas(temps);
    return 0;
}

```

Explicación: el arreglo viaja a la función por referencia constante: sin copia y con el tamaño incluido en el tipo (el 7 es parte de la firma). `at(i)` verifica cada acceso; `front()` inicializa máximo y mínimo con un valor real del arreglo en vez de un número mágico; `back()` muestra el último dato sin calcular índices. Compara con la versión de arreglos crudos de la primera parte: aquí la función no necesita parámetro de tamaño porque el tipo ya lo trae.

Ejercicio 2: censor de palabras con `find` y `replace`

Escribe una función `void censurar(std::string& texto, const std::string& prohibida)` que reemplace todas las apariciones de la palabra prohibida por asteriscos (uno por cada letra). Pruébala con una frase del usuario. Cuida el avance de la posición.

Solución:

```

#include <iostream>
#include <string>

void censurar(std::string& texto, const std::string& prohibida) {
    if (prohibida.empty()) return; // sin esto, el ciclo seria infinito
    std::string asteriscos(prohibida.length(), '*');
    std::string::size_type pos = 0;
    while ((pos = texto.find(prohibida, pos)) != std::string::npos) {
        texto.replace(pos, prohibida.length(), asteriscos);
        pos += asteriscos.length();
    }
}

int main() {
    std::string frase, mala;
    std::cout << "Frase: ";
    std::getline(std::cin, frase);
    std::cout << "Palabra a censurar: ";
    std::getline(std::cin, mala);
    censurar(frase, mala);
    std::cout << "Resultado: " << frase << std::endl;
    return 0;
}

```

Explicación: `std::string asteriscos(n, '*')` construye un string de `n` asteriscos, del mismo largo que la palabra prohibida, así el texto no cambia de tamaño y el avance es simple. La guarda contra palabra vacía es vital: `find("")` siempre "encuentra" en la posición actual y el ciclo jamás terminaría. El avance `pos += asteriscos.length()` salta lo recién insertado; como los asteriscos no contienen la palabra prohibida (salvo que la palabra prohibida sea ""), no hay riesgo de reprocesar.

Ejercicio 3: clasificador de caracteres con `cctype`

Pide una línea al usuario y reporta cuántas letras, cuántos dígitos, cuántos espacios en blanco y cuántos otros caracteres contiene. Usa las funciones de `<cctype>` con el cast seguro a `unsigned char`.

Solución:

```

#include <iostream>
#include <string>
#include <cctype>

int main() {
    std::string linea;
    std::cout << "Escribe una línea: ";
    std::getline(std::cin, linea);

    int letras = 0, digitos = 0, espacios = 0, otros = 0;
    for (char c : linea) {
        unsigned char u = static_cast<unsigned char>(c);
        if (std::isalpha(u))      letras++;
        else if (std::isdigit(u)) digitos++;
        else if (std::isspace(u)) espacios++;
        else                     otros++;
    }

    std::cout << "Letras: " << letras << std::endl;
    std::cout << "Digitos: " << digitos << std::endl;
    std::cout << "Espacios: " << espacios << std::endl;
    std::cout << "Otros: " << otros << std::endl;
    return 0;
}

```

Explicación: una sola pasada clasifica cada carácter en cuatro cubetas excluyentes (el else if garantiza que cada carácter cuente una sola vez). isspace cubre espacio, tab y salto de línea, no solo el espacio normal. El cast a unsigned char es el hábito seguro que evita comportamiento indefinido con caracteres extendidos. Este esqueleto (un ciclo, varias categorías) sirve para analizadores de contraseñas, contadores de palabras y validadores de formato.

Ejercicio 4: reservar memoria con puntero doble

Escribe una función `bool reservarEnCeros(int pp, int n)` que reserve un arreglo de `n` enteros, lo llene con ceros y lo entregue por el puntero doble. Devuelve false si los argumentos no son válidos. El main debe pedir `n` al usuario, usar la función, comprobar que todo quedó en cero y liberar correctamente.

Solución:


```

#include <iostream>

bool reservarEnCeros(int** pp, int n) {
    if (pp == nullptr || n <= 0) return false;
    *pp = new int[n];
    for (int i = 0; i < n; i++) {
        (*pp)[i] = 0;
    }
    return true;
}

int main() {
    int n;
    std::cout << "Cuantos enteros? ";
    std::cin >> n;

    int* datos = nullptr;
    if (!reservarEnCeros(&datos, n)) {
        std::cout << "No se pudo reservar." << std::endl;
        return 1;
    }

    bool todoCero = true;
    for (int i = 0; i < n; i++) {
        if (datos[i] != 0) todoCero = false;
    }
    std::cout << (todoCero ? "Todo en cero, correcto." : "Error: hay basura.") <<
std::endl;

    delete[] datos;
    datos = nullptr;
    return 0;
}

```

Explicación: el `&datos` del `main` convierte el `int*` en `int`, **permitiendo que la función escriba el resultado del `new` directamente en la variable del llamador**. Sin el nivel extra de indirección, el puntero reservado se perdería al salir de la función. Las validaciones (`pp` nulo, `n` no positivo) hacen la función robusta: una función que reserva memoria debe decir claramente cuándo falla, porque el llamador no puede adivinarlo. El `main` verifica el contenido, libera con `delete[]` y pone `nullptr`: el protocolo completo de memoria manual.

Ejercicio 5: identifica las cuatro combinaciones

Sin compilar, indica para cada declaración qué operaciones son válidas y cuáles darían error de compilación. Luego verifica compilando.

```
int valor = 100;
const int* a = &valor;
int* const b = &valor;
const int* const c = &valor;

// Operaciones:
// 1) *a = 50;
// 2) a = nullptr;
// 3) *b = 50;
// 4) b = nullptr;
// 5) *c = 50;
// 6) c = nullptr;
```

Solución: la 1) es ERROR: a es puntero a entero constante, el dato no se puede modificar por a. La 2) es válida: el puntero a sí puede cambiar (incluso a nullptr). La 3) es válida: b es puntero constante, pero el dato apuntado sí se puede modificar. La 4) es ERROR: b es el puntero constante, no se puede reasignar después de inicializarlo. La 5) es ERROR y la 6) es ERROR: c tiene ambos const, nada se puede cambiar por c. Verifica con el compilador: cada error confirma tu lectura de la declaración. Si fallaste alguna, repasa la regla de "const modifica lo que tiene a la izquierda" y la lectura de derecha a izquierda hasta que las seis salgan sin dudar.

Ejercicio 6: suma desde la línea de comandos

Escribe un programa que reciba números como argumentos de línea de comandos y muestre su suma y su promedio. Si no recibe argumentos, debe mostrar cómo se usa (usando argv[0]) y terminar con código de error. Usa std::stoi con manejo de excepciones para entradas no numéricas.

Solución:

```

#include <iostream>
#include <string>

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cout << "Uso: " << argv[0] << " <num1> <num2> ..." << std::endl;
        return 1;
    }

    double suma = 0.0;
    for (int i = 1; i < argc; i++) {
        try {
            suma += std::stod(argv[i]);
        } catch (const std::exception& e) {
            std::cout << "Error: '" << argv[i] << "' no es un numero." <<
std::endl;
            return 1;
        }
    }

    int cantidad = argc - 1; // argv[0] es el programa, no se cuenta
    std::cout << "Suma: " << suma << std::endl;
    std::cout << "Promedio: " << suma / cantidad << std::endl;
    return 0;
}

```

Explicación: el ciclo empieza en 1 porque argv[0] es el nombre del programa, no un dato. argc - 1 es la cantidad real de números, y se usa como divisor del promedio. Cada argv[i] llega como texto y stod lo convierte; el try/catch atrapa tanto texto no numérico (invalid_argument) como números fuera de rango (out_of_range), porque ambas heredan de std::exception. Los return 1 comunican el fallo al sistema operativo: un script que llame a este programa puede detectar si algo salió mal. Pruébalo con ./suma 10 20 30, con ./suma solo, y con ./suma 10 hola 30 para ver cada camino.

Resumen de la tercera parte

- std::array es el arreglo de tamaño fijo moderno: sabe su tamaño, se copia, se compara y ofrece at() con verificación. Para tamaños variables, std::vector (unidad 8).
- Concatenar en ciclos: reserve() antes, += dentro. La familia find (find, rfind, find_first_of, find_last_of, find_first_not_of, find_last_not_of) cubre toda la búsqueda; replace, insert y erase hacen la cirugía.

- `<cctype>` clasifica caracteres (`isalpha`, `isdigit`, `isspace`, ...) y convierte (`toupper`, `tolower`); usa el cast a `unsigned char`.
- `int` **guarda la dirección de un puntero: sirve para modificar punteros dentro de funciones y para arreglos de punteros como `argv`.**
- Las cuatro combinaciones de `const` con punteros se leen de derecha a izquierda; `const int*` protege el dato, `int* const` fija el puntero.
- `main(int argc, char* argv[])` recibe argumentos como texto; conviértelos con `stoi/stod` y valida `argc` antes de usar `argv`.
- `std::sort` con lambdas supera a `qsort` en seguridad de tipos, claridad y velocidad.

UNIDAD 6: Referencias y programación orientada a objetos

Esta es la unidad donde C++ se convierte en C++ de verdad. Hasta ahora programaste como en C mejorado: funciones sueltas y datos sueltos. Ahora aprenderás las referencias (la forma elegante de compartir datos) y entrarás a la programación orientada a objetos: clases y objetos, constructores y destructores. Es un cambio de mentalidad, no solo de sintaxis, así que tómate tu tiempo.

La idea central de la POO es modelar el mundo: en vez de datos pasivos y funciones separadas, creas objetos que combinan ambos. Un objeto CuentaBancaria no es solo un saldo: es un saldo que sabe depositarse, retirarse y mostrarse, y que no deja que lo manipulen mal. Los objetos se cuidan solos, y eso cambia todo.

Al terminar sabrás: usar referencias para parámetros eficientes, definir clases con atributos y métodos, crear objetos, escribir constructores y destructores, y entender el ciclo de vida de un objeto.

Explicación detallada

Clase 16: Referencias

Una referencia es un alias: otro nombre para una variable que ya existe. Se declara con &:

```
int x = 10;
int& ref = x;
ref = 20; // x ahora vale 20
```

ref no es una copia de x: ES x con otro nombre. Todo lo que hagas con ref le pasa a x. A diferencia de los punteros, la referencia debe inicializarse al declararse y no puede cambiar para apuntar a otra variable después. Es más segura y más legible.

Su uso estrella: parámetros de función que se pueden modificar o que son grandes:

```
void incrementar(int& n) {  
    n++;  
}  
int valor = 5;  
incrementar(valor); // valor ahora es 6
```

Sin el &, la función recibiría una copia y el incremento se perdería. Con punteros lograrías lo mismo, pero la sintaxis de referencias es limpia: no hay * ni & al llamar.

Para parámetros grandes que no quieres modificar, usa referencia constante:

```
void mostrar(const std::string& texto) {  
    std::cout << texto << std::endl;  
}
```

const std::string& evita copiar el string completo (eficiente) e impide modificarlo (seguro). Es el estándar profesional para pasar objetos a funciones. Memorízalo: verás const Tipo& en todo código C++ serio.

También puedes devolver referencias desde funciones, pero con una regla de hierro: nunca devuelvas referencia a una variable local, porque muere al salir de la función. Es un error clásico que produce basura.

¿Cuándo usar referencia y cuándo puntero? Regla práctica: usa referencias cuando el dato siempre existe y no puede ser nulo (parámetros, alias simples). Usa punteros cuando el dato puede no existir (nullptr) o necesitas cambiar a qué apunta. En C++ moderno, además, los punteros inteligentes cubren la memoria dinámica.

Clase 17: Clases y objetos

Una clase es un molde para crear objetos. Define qué datos tiene el objeto (atributos) y qué puede hacer (métodos). Un objeto es una instancia concreta de la clase: si la clase es el plano de una casa, los objetos son las casas construidas con ese plano.

```

class Rectangulo {
public:
    double base;
    double altura;

    double area() {
        return base * altura;
    }

    void mostrar() {
        std::cout << "Base: " << base << ", Altura: " << altura << std::endl;
    }
};

int main() {
    Rectangulo r;
    r.base = 5.0;
    r.altura = 3.0;
    r.mostrar();
    std::cout << "Area: " << r.area() << std::endl;
    return 0;
}

```

La palabra `public:` indica que lo siguiente es accesible desde fuera. (En la unidad 7 verás `private` y el encapsulamiento.) Los métodos se llaman con el operador punto: `r.area()`. Dentro de un método, puedes usar los atributos directamente, sin prefijo: el método "sabe" a qué objeto pertenece.

Nota el punto y coma después de la llave de cierre de la clase: es obligatorio y olvidarlo es un error clásico con mensajes confusos del compilador.

La diferencia con `struct`: en C++, `struct` y `class` son casi idénticos; la única diferencia es que en `struct` todo es público por defecto y en `class` todo es privado por defecto. Por convención, `struct` se usa para agrupar datos simples y `class` para objetos con comportamiento.

Los métodos pueden definirse fuera de la clase usando el operador de resolución de ámbito `::`:

```
class Rectangulo {
public:
    double base, altura;
    double area();
};

double Rectangulo::area() {
    return base * altura;
}
```

Esta separación (declaración dentro, definición fuera) es la base de los archivos de cabecera .h y de implementación .cpp en proyectos reales. En este libro definiremos dentro por brevedad, pero debes conocer la otra forma.

Clase 18: Constructores y destructores

Cuando creas un objeto con Rectangulo r;, sus atributos quedan sin inicializar (basura), igual que las variables sueltas. El constructor resuelve esto: es un método especial que se ejecuta automáticamente al crear el objeto. Se llama igual que la clase y no tiene tipo de retorno:

```
class Rectangulo {
public:
    double base, altura;

    Rectangulo(double b, double h) {
        base = b;
        altura = h;
    }

    double area() {
        return base * altura;
    }
};

int main() {
    Rectangulo r(5.0, 3.0); // el constructor se ejecuta solo
    std::cout << "Area: " << r.area() << std::endl;
    return 0;
}
```

La forma moderna de inicializar en el constructor es la lista de inicialización:


```
Rectangulo(double b, double h) : base(b), altura(h) {}
```

Es más eficiente (inicializa directamente en vez de asignar después) y obligatoria para ciertos miembros. Acostúmbrate a ella desde ahora.

Puedes tener varios constructores (sobrecarga): uno sin parámetros (por defecto), uno con parámetros, etc. Si no escribes ningún constructor, el compilador genera uno por defecto que no hace nada útil. Si escribes uno con parámetros, el compilador ya no genera el de por defecto: tenlo en cuenta.

El destructor es el gemelo del constructor: se ejecuta automáticamente cuando el objeto muere (sale de alcance). Se llama ~NombreClase y sirve para liberar recursos:

```
~Rectangulo() {  
    // limpieza si fuera necesaria  
}
```

Este par constructor/destructor es una de las ideas más poderosas de C++: los objetos nacen inicializados y mueren limpiando lo suyo. Se llama RAII (adquisición de recursos es inicialización) y es la base de la gestión segura de recursos en C++ moderno, incluyendo los punteros inteligentes de la unidad 10.

El ciclo de vida: un objeto local nace al declararse y muere al cerrarse su bloque. Un objeto creado con new nace con new y muere con delete (ahí se llama su destructor). Entender este ciclo es entender la mitad de C++.

Ejemplos paso a paso

Ejemplo 1: Referencias en la práctica

```

#include <iostream>
#include <string>

void aMayusculas(std::string& texto) {
    for (char& c : texto) {
        c = std::toupper(c);
    }
}

void imprimir(const std::string& texto) {
    std::cout << texto << std::endl;
}

int main() {
    std::string mensaje = "hola mundo";
    imprimir(mensaje);      // referencia constante: solo lee
    aMayusculas(mensaje);   // referencia normal: modifica
    imprimir(mensaje);      // ahora en mayúsculas

    int& ref = *(new int(0)); // NO hagas esto: ejemplo de lo prohibido
    delete &ref;
    return 0;
}

```

Fíjate en el for con char& c: la referencia permite modificar cada carácter del string en su lugar. Sin el &, modificarías copias y nada cambiaría. (El fragmento con new es solo para mostrar sintaxis: en código real no crearías referencias así.)

Ejemplo 2: Clase CuentaBancaria completa

```

#include <iostream>
#include <string>

class CuentaBancaria {
private:
    std::string titular;
    double saldo;

public:
    CuentaBancaria(std::string t, double sInicial) : titular(t), saldo(sInicial) {
        std::cout << "Cuenta creada para " << titular << std::endl;
    }

    ~CuentaBancaria() {

```

```

        std::cout << "Cuenta de " << titular << " cerrada." << std::endl;
    }

    void depositar(double monto) {
        if (monto > 0) {
            saldo += monto;
            std::cout << "Deposito: $" << monto << std::endl;
        }
    }

    bool retirar(double monto) {
        if (monto > 0 && monto <= saldo) {
            saldo -= monto;
            std::cout << "Retiro: $" << monto << std::endl;
            return true;
        }
        std::cout << "Fondos insuficientes." << std::endl;
        return false;
    }

    void mostrar() const {
        std::cout << "Titular: " << titular << ", Saldo: $" << saldo << std::endl;
    }
};

int main() {
    CuentaBancaria cuenta("Esteban", 1000.0);
    cuenta.depositar(500.0);
    cuenta.retirar(200.0);
    cuenta.retirar(5000.0); // falla: sin fondos
    cuenta.mostrar();
    return 0; // aquí se llama el destructor
}

```

Observa: saldo es privado (nadie puede ponerle un valor absurdo desde fuera), pero los métodos públicos ofrecen operaciones seguras con validación. El constructor inicializa, el destructor avisa al cerrar. El const en mostrar() promete no modificar el objeto: buena práctica en métodos de solo lectura.

Ejemplo 3: Clase Fecha con varios constructores

```

#include <iostream>

class Fecha {
private:
    int dia, mes, anio;

public:
    Fecha() : dia(1), mes(1), anio(2000) {
        std::cout << "Fecha por defecto." << std::endl;
    }

    Fecha(int d, int m, int a) : dia(d), mes(m), anio(a) {
        std::cout << "Fecha personalizada." << std::endl;
    }

    void mostrar() const {
        std::cout << dia << "/" << mes << "/" << anio << std::endl;
    }
};

int main() {
    Fecha f1;
    Fecha f2(19, 9, 2026);
    f1.mostrar();
    f2.mostrar();
    return 0;
}

```

Sobrecarga de constructores: el compilador elige según los argumentos. f1 usa el de por defecto, f2 el personalizado. Así los objetos siempre nacen en un estado válido.

Práctica

Ejercicio 1

Escribe una función void ordenar(double& a, double& b) que deje el menor en a y el mayor en b usando referencias. Pruébala.

Solución:

```

#include <iostream>

void ordenar(double& a, double& b) {
    if (a > b) {
        double temp = a;
        a = b;
        b = temp;
    }
}

int main() {
    double x = 9.5, y = 3.2;
    ordenar(x, y);
    std::cout << "x=" << x << ", y=" << y << std::endl;
    return 0;
}

```

Explicación: sin referencias, la función trabajaría con copias y x, y no cambiarían. Las referencias permiten "devolver" varios resultados modificando los parámetros. Es el reemplazo moderno del truco de punteros de la unidad 5.

Ejercicio 2

Crea la clase Libro con atributos privados titulo, autor y paginas; constructor que los inicialice; métodos mostrar() const y esLargo() que devuelva true si tiene más de 300 páginas.

Solución:

```

#include <iostream>
#include <string>

class Libro {
private:
    std::string titulo;
    std::string autor;
    int paginas;

public:
    Libro(std::string t, std::string a, int p) : titulo(t), autor(a), paginas(p) {}

    void mostrar() const {
        std::cout << "\"" << titulo << "\" de " << autor << " (" << paginas << "
pags.)" << std::endl;
    }

    bool esLargo() const {
        return paginas > 300;
    }
};

int main() {
    Libro l1("Cien años de soledad", "Garcia Marquez", 432);
    Libro l2("El principito", "Saint-Exupery", 96);
    l1.mostrar();
    std::cout << "Es largo: " << l1.esLargo() << std::endl;
    l2.mostrar();
    std::cout << "Es largo: " << l2.esLargo() << std::endl;
    return 0;
}

```

Explicación: los datos están protegidos (private) y solo se accede por métodos. esLargo encapsula la regla "más de 300 páginas": si mañana cambia a 400, solo se toca un lugar.

Ejercicio 3

Crea la clase Punto con x, y; constructor por defecto (0,0) y constructor con parámetros; método distancia(const Punto& otro) const que calcule la distancia euclidiana (necesitas <cmath> y sqrt).

Solución:

```

#include <iostream>
#include <cmath>

class Punto {
private:
    double x, y;

public:
    Punto() : x(0), y(0) {}
    Punto(double px, double py) : x(px), y(py) {}

    double distancia(const Punto& otro) const {
        double dx = x - otro.x;
        double dy = y - otro.y;
        return std::sqrt(dx * dx + dy * dy);
    }

    void mostrar() const {
        std::cout << "(" << x << ", " << y << ")" << std::endl;
    }
};

int main() {
    Punto a(0, 0), b(3, 4);
    a.mostrar(); b.mostrar();
    std::cout << "Distancia: " << a.distancia(b) << std::endl;
    return 0;
}

```

Explicación: dentro de distancia, podemos acceder a otro.x aunque sea privado: los métodos de una clase ven los privados de otros objetos de la misma clase. La distancia entre (0,0) y (3,4) es 5: el triángulo 3-4-5 clásico.

Ejercicio 4

Agrega a la clase Punto un destructor que imprima "Punto destruido" y un método mover(double dx, double dy). Crea puntos dentro de un bloque { } para ver cuándo se llama el destructor.

Solución:

```

#include <iostream>

class Punto {
private:
    double x, y;

public:
    Punto(double px, double py) : x(px), y(py) {
        std::cout << "Punto creado." << std::endl;
    }
    ~Punto() {
        std::cout << "Punto destruido." << std::endl;
    }
    void mover(double dx, double dy) {
        x += dx;
        y += dy;
    }
    void mostrar() const {
        std::cout << "(" << x << ", " << y << ")" << std::endl;
    }
};

int main() {
    std::cout << "Inicio" << std::endl;
    {
        Punto p(1, 2);
        p.mover(5, 5);
        p.mostrar();
        std::cout << "Fin del bloque interno" << std::endl;
    }
    std::cout << "Fin del main" << std::endl;
    return 0;
}

```

Explicación: el destructor se llama justo al cerrar el bloque interno, antes de "Fin del main". Ver el ciclo de vida en acción es la mejor forma de interiorizar RAI: los recursos se liberan solos al salir del alcance.

Ejercicio 5

Crea la clase Termometro con un atributo privado celsius; constructor; métodos setCelsius (valida que no sea menor a -273.15), getCelsius, y aFahrenheit(). Pruébala incluyendo un intento de valor inválido.

Solución:

```
#include <iostream>

class Termometro {
private:
    double celsius;

public:
    Termometro(double c) : celsius(c) {
        if (celsius < -273.15) celsius = -273.15;
    }

    void setCelsius(double c) {
        if (c >= -273.15) {
            celsius = c;
        } else {
            std::cout << "Temperatura imposible. No se cambio." << std::endl;
        }
    }

    double getCelsius() const { return celsius; }
    double aFahrenheit() const { return celsius * 9.0 / 5.0 + 32; }
};

int main() {
    Termometro t(25);
    std::cout << t.getCelsius() << "C = " << t.aFahrenheit() << "F" << std::endl;
    t.setCelsius(-300); // inválido
    t.setCelsius(-40);
    std::cout << t.getCelsius() << "C = " << t.aFahrenheit() << "F" << std::endl;
    return 0;
}
```

Explicación: el cero absoluto (-273.15) es un invariante físico: ninguna temperatura puede ser menor. Al validar en el constructor y en el setter, garantizamos que ningún objeto Termometro exista en estado imposible. Eso es encapsulamiento: el objeto protege sus propias reglas.

Resumen de la unidad

- Una referencia (Tipo&) es un alias: otro nombre para una variable existente. Ideal para parámetros modificables y para evitar copias con const Tipo&.

- Una clase es un molde; un objeto es una instancia. Atributos = datos, métodos = comportamiento.
- El constructor (mismo nombre que la clase) inicializa el objeto al crearse; usa lista de inicialización. Puede haber varios (sobrecarga).
- El destructor (~Nombre) limpia al morir el objeto. Juntos implementan RAI: recursos que se gestionan solos.
- private protege los datos; public expone la interfaz segura. Los métodos const prometen no modificar.
- En la unidad 7 profundizarás en los tres pilares de la POO: encapsulamiento, herencia y polimorfismo.

Profundización: diseño orientado a objetos

Ya sabes crear clases. Ahora aprenderás a diseñarlas bien: el puntero this, miembros estáticos, const-correctness (el arte de prometer no modificar), funciones amigas, y cómo pensar en objetos antes de escribir código. Esto es lo que separa a quien "usa clases" de quien "diseña con objetos".

El puntero this

Dentro de cualquier método, this es un puntero al objeto actual. Normalmente es invisible, pero a veces lo necesitas:

```

class Contador {
private:
    int valor;
public:
    Contador(int valor) : valor(valor) {}
    // Sin this, el parámetro ocultaría al atributo:
    void setValor(int valor) {
        this->valor = valor; // this->valor es el atributo, valor el parámetro
    }
    Contador& incrementar() {
        valor++;
        return *this; // devuelve el objeto para encadenar
    }
    void mostrar() const {
        std::cout << valor << std::endl;
    }
};

int main() {
    Contador c(10);
    c.incrementar().incrementar().mostrar(); // encadenado: 12
    return 0;
}

```

Devolver `*this` por referencia permite encadenar llamadas: `c.incrementar().incrementar()`. Es el truco detrás de `cout << a << b` (cada `<<` devuelve el flujo). Úsalo en métodos que "configuran" el objeto.

Miembros estáticos: lo que pertenece a la clase

Un miembro static pertenece a la clase, no a cada objeto: hay una sola copia compartida.

```

#include <iostream>

class Usuario {
private:
    std::string nombre;
    static int totalUsuarios; // compartido

public:
    Usuario(const std::string& n) : nombre(n) {
        totalUsuarios++;
    }
    ~Usuario() {
        totalUsuarios--;
    }
    static int getTotal() {
        return totalUsuarios;
    }
};

int Usuario::totalUsuarios = 0; // definición obligatoria fuera de la clase

int main() {
    Usuario a("Ana"), b("Luis");
    std::cout << "Total: " << Usuario::getTotal() << std::endl; // 2
    return 0;
}

```

Los static sirven para contadores, constantes compartidas, fábricas y configuración global de la clase. Se acceden con Clase::miembro sin necesidad de un objeto. Los métodos static no tienen this (no pertenecen a ningún objeto) y solo pueden tocar miembros static.

Const-correctness: promete y cumple

Marcar todo lo posible como const es una disciplina con beneficios reales:

- Parámetros const&: no copiar, no modificar.
- Métodos const: garantizan no modificar el objeto; además, pueden llamarse en objetos const.
- Variables const: el compilador impide cambios accidentales.

```

class Punto {
    double x, y;
public:
    double getX() const { return x; }          // promete no modificar
    void mover(double dx, double dy) { x += dx; y += dy; } // sí modifica
};

void imprimir(const Punto& p) {
    std::cout << p.getX() << std::endl; // OK: getX es const
    // p.mover(1,1); // ERROR: mover no es const
}

```

Si olvidas el const en getX, no podrás llamarlo desde una referencia const. El const es contagioso: úsalo desde el inicio o te perseguirán errores.

Funciones y clases amigas (friend)

friend otorga a una función o clase externa acceso a los privados. Úsalo con moderación: rompe el encapsulamiento, pero a veces es lo correcto (como operator<<).

```

class Caja {
private:
    int secreto;
    friend void espiar(const Caja& c); // esta función sí puede ver
public:
    Caja(int s) : secreto(s) {}
};

void espiar(const Caja& c) {
    std::cout << "Secreto: " << c.secreto << std::endl; // permitido por friend
}

```

La amistad no es transitiva ni heredada: hay que declararla explícitamente en cada clase.

Copias: el constructor de copia (introducción)

Cuando copias un objeto (Punto b = a;), C++ usa el constructor de copia. El que genera el compilador copia miembro por miembro (copia superficial). Para clases con punteros y memoria propia, eso es peligroso: dos objetos apuntando a la misma memoria, doble delete al morir.

```

class Simple {
    int* dato;
public:
    Simple(int v) : dato(new int(v)) {}
    ~Simple() { delete dato; }
    // ¡Falta constructor de copia y operador=: copiar esto es un bug!
};

```

La regla de tres (clásica): si defines destructor, constructor de copia u operador de asignación, probablemente necesitas los tres. En C++ moderno, la solución elegante es no manejar memoria cruda: usa vector, string o unique_ptr y el compilador genera copias correctas solo. Profundizarás en cursos avanzados; por ahora, reconoce el peligro.

Pensar en objetos: del problema al diseño

Ante un problema, haz esto:

1. Nombra los sustantivos: son tus clases candidatas (Cliente, Pedido, Producto).
2. Nombra los verbos: son tus métodos (registrar, calcular, enviar).
3. Define responsabilidades: cada clase hace una cosa bien (principio de responsabilidad única).
4. Dibuja relaciones: ¿quién usa a quién? ¿quién contiene a quién?
5. Empieza por la interfaz pública: ¿qué necesita pedirle el resto del programa a esta clase?

Ejemplo: sistema de estacionamiento. Clases: Auto (placa, horaEntrada), Cajon (número, ocupado, auto), Estacionamiento (vector de cajones, métodos estacionar, retirar, cobrar). Cada una con su responsabilidad. El main solo orquesta.

Composición: "tiene un"

La mayoría de las relaciones no son herencia sino composición: una clase contiene objetos de otra.

```

class Motor {
public:
    void encender() { std::cout << "Motor encendido." << std::endl; }
};

class Auto {
private:
    Motor motor; // Auto TIENE UN motor (no ES UN motor)
    std::string modelo;
public:
    Auto(const std::string& m) : modelo(m) {}
    void arrancar() {
        motor.encender();
        std::cout << modelo << " en marcha." << std::endl;
    }
};

```

El motor se construye con el auto y muere con él. Prefiere composición sobre herencia cuando dades: es más flexible y genera menos acoplamiento.

Más ejemplos

Ejemplo: clase con todo (static, const, this, friend)

```

#include <iostream>
#include <string>

class Producto {
private:
    std::string nombre;
    double precio;
    static int totalProductos;

public:
    Producto(const std::string& n, double p) : nombre(n), precio(p) {
        totalProductos++;
    }
    ~Producto() { totalProductos--; }

    Producto& aplicarDescuento(double porcentaje) {
        precio *= (1.0 - porcentaje / 100.0);
        return *this; // encadenable
    }

    void mostrar() const {
        std::cout << nombre << ": $" << precio << std::endl;
    }

    static int getTotal() { return totalProductos; }

    friend bool masCaro(const Producto& a, const Producto& b);
};

int Producto::totalProductos = 0;

bool masCaro(const Producto& a, const Producto& b) {
    return a.precio > b.precio; // friend: acceso directo
}

int main() {
    Producto p1("Laptop", 15000), p2("Mouse", 500);
    p1.aplicarDescuento(10).mostrar(); // encadenado
    p2.mostrar();
    std::cout << "Total productos: " << Producto::getTotal() << std::endl;
    std::cout << "p1 mas caro que p2? " << masCaro(p1, p2) << std::endl;
    return 0;
}

```


Todo junto: static compartido, encadenamiento con *this, método const, función friend. Este es el nivel de diseño de una clase profesional.

Práctica adicional

Ejercicio A

Crea una clase ContadorVisitas con un static que cuente cuántas instancias se han creado en total (histórico, que no baje al destruirse) y un método static para consultarlo.

Solución:

```
#include <iostream>

class ContadorVisitas {
private:
    static int historico;
public:
    ContadorVisitas() { historico++; }
    static int getHistorico() { return historico; }
};

int ContadorVisitas::historico = 0;

int main() {
    {
        ContadorVisitas a, b;
        std::cout << ContadorVisitas::getHistorico() << std::endl; // 2
    }
    ContadorVisitas c;
    std::cout << ContadorVisitas::getHistorico() << std::endl; // 3
    return 0;
}
```

Explicación: a diferencia del ejemplo de Usuario (que resta al destruir), aquí el histórico solo sube: cuenta creaciones totales. Dos usos distintos del mismo mecanismo.

Ejercicio B

Diseña las clases para una pizzería: Ingrediente (nombre, costo), Pizza (vector de Ingredientes, tamaño, métodos agregarIngrediente y costoTotal). Usa composición.

Solución:

```

#include <iostream>
#include <string>
#include <vector>

class Ingrediente {
private:
    std::string nombre;
    double costo;
public:
    Ingrediente(const std::string& n, double c) : nombre(n), costo(c) {}
    double getCosto() const { return costo; }
    std::string getNombre() const { return nombre; }
};

class Pizza {
private:
    std::string tamano;
    std::vector<Ingrediente> ingredientes; // composición: TIENE ingredientes
    static constexpr double BASE = 80.0;
public:
    Pizza(const std::string& t) : tamano(t) {}
    void agregarIngrediente(const Ingrediente& i) {
        ingredientes.push_back(i);
    }
    double costoTotal() const {
        double total = BASE;
        for (const auto& ing : ingredientes) total += ing.getCosto();
        return total;
    }
    void mostrar() const {
        std::cout << "Pizza " << tamano << " con " << ingredientes.size()
            << " ingredientes: $" << costoTotal() << std::endl;
    }
};

int main() {
    Pizza p("grande");
    p.agregarIngrediente(Ingrediente("Queso extra", 25));
    p.agregarIngrediente(Ingrediente("Peperoni", 30));
    p.mostrar();
    return 0;
}

```

Explicación: Pizza contiene Ingredientes (composición), no hereda de ellos. static constexpr BASE es una constante de clase evaluada en compilación. El diseño refleja el mundo real: la pizza tiene ingredientes.

Ejercicio C

Explica qué problema habría si la clase Simple del ejemplo de copias se copiara, y propón la solución moderna.

Solución: el constructor de copia por defecto copiaría el puntero dato, no el entero apuntado. Dos objetos compartirían la misma memoria; al morir el primero haría delete, y el segundo quedaría con puntero colgante; al morir el segundo, doble delete. Soluciones modernas: usar `std::unique_ptr<int>` (no copiable, deja claro que no se debe copiar) o `std::shared_ptr<int>` (copia compartida segura), o simplemente `int` sin puntero si no se necesita memoria dinámica. La moraleja: evita la memoria cruda en clases y el problema desaparece.

Ejercicio D

Crea una clase Calculadora con métodos encadenables sumar, restar y mostrar, usando `return *this`. Prueba: `calc.sumar(5).restar(2).sumar(10).mostrar()` debe mostrar 13.

Solución:

```
#include <iostream>

class Calculadora {
private:
    double valor;
public:
    Calculadora(double v = 0) : valor(v) {}
    Calculadora& sumar(double x) { valor += x; return *this; }
    Calculadora& restar(double x) { valor -= x; return *this; }
    void mostrar() const { std::cout << "Resultado: " << valor << std::endl; }
};

int main() {
    Calculadora calc;
    calc.sumar(5).restar(2).sumar(10).mostrar(); // 13
    return 0;
}
```

Explicación: cada método devuelve `*this` por referencia, permitiendo encadenar. Es el patrón "interfaz fluida" que usan bibliotecas famosas. El constructor con default permite Calculadora calc; empezando en 0. Esta tercera parte de la Unidad 6 convierte tus clases de "funcionan" a "están bien diseñadas". En las partes anteriores aprendiste referencias, clases, objetos, constructores, destructores, `this`, `static`, `const` y `friend`. Aquí profundizarás en la lista de inicialización (su orden secreto y cuándo es obligatoria), la delegación de constructores, el freno de `explicit`, el constructor de copia con un bug real y su corrección, el operador de asignación, la famosa regla de tres/cinco/cero, la diferencia entre agregación y composición, las clases inmutables y los namespaces para organizar tu código. Todo es contenido nuevo que se apoya en lo anterior sin repetirlo.

La lista de inicialización a fondo

En la primera parte viste la forma moderna de escribir un constructor:

```
Rectangulo(double b, double h) : base(b), altura(h) {}
```

Esa parte después de los dos puntos es la lista de inicialización. Hasta ahora la usabas por costumbre; ahora entenderás por qué los profesionales la consideran obligatoria y qué trampas esconde.

Inicializar no es asignar

Hay una diferencia real entre inicializar un miembro en la lista y asignarlo en el cuerpo del constructor. Cuando escribes:

```
class Persona {
private:
    std::string nombre;
public:
    Persona(const std::string& n) {
        nombre = n; // ASIGNACION
    }
};
```

ocurren dos pasos: primero se construye nombre con su constructor por defecto (string vacío), y después se le asigna n (se destruye el vacío y se copia el nuevo). Es trabajo doble. Con la lista de inicialización:

```
Persona(const std::string& n) : nombre(n) {} // INICIALIZACION directa
```

nombre se construye directamente con n: un solo paso. Para un int la diferencia es invisible, pero para un std::string, un vector o cualquier objeto con constructor costoso, la lista evita una construcción y una asignación inútiles por cada objeto creado. En programas que crean miles de objetos, se nota. La regla es simple: inicializa en la lista siempre que puedas; deja el cuerpo del constructor para la lógica que no sea inicialización (validaciones, mensajes, cálculos derivados).

El orden lo manda la declaración, no la lista

Aquí está la trampa más famosa de la lista de inicialización: los miembros se inicializan en el orden en que fueron DECLARADOS en la clase, no en el orden en que los escribas en la lista. El compilador ignora tu orden y usa el suyo. Mira este bug clásico:

```
class Medicion {
private:
    int base;
    int doble;
public:
    Medicion(int b) : doble(base * 2), base(b) {} // ¡ORDEN ENGANOSO!
};
```

Lees la lista y parece que doble se calcula después de base. Pero base se declaró antes que doble, así que base se inicializa primero (con b, correcto) y luego doble con base * 2 (correcto por accidente). El peligro aparece al revés:

```
class Medicion {
private:
    int doble; // declarado PRIMERO
    int base;  // declarado DESPUÉS
public:
    Medicion(int b) : base(b), doble(base * 2) {} // ¡BUG!
};
```

Aquí doble se inicializa antes que base (porque se declaró antes), así que doble = base * 2 usa una base todavía sin inicializar: basura multiplicada por dos. El compilador ni siquiera se queja en todos los casos (algunos avisan con -Wall, otros no). La defensa tiene dos partes: primera, declara los miembros en un orden lógico y escríbelos en la lista en ese mismo orden; segunda, evita que un miembro dependa de otro en la inicialización si puedes calcular el valor antes.

Cuando veas un warning del compilador sobre orden de inicialización, tómallo en serio: es el compilador intentando salvarte de un bug silencioso.

Miembros const y referencias: la lista es obligatoria

Hay dos casos donde la lista de inicialización no es una recomendación sino una obligación: los miembros const y las referencias miembro. Un miembro const no se puede asignar después de construido (asignar es modificar, y es const), así que la única oportunidad de darle valor es la inicialización. Lo mismo pasa con una referencia miembro: debe quedar ligada a su objetivo en el momento de nacer, no después.

```
class Sensor {
private:
    const int id;           // const: solo la lista puede darle valor
    const std::string& zona; // referencia miembro: debe ligarse al nacer
    double lectura;
public:
    Sensor(int i, const std::string& z, double l)
        : id(i), zona(z), lectura(l) {}
    // Sin la lista, este constructor NO compila.

    void mostrar() const {
        std::cout << "Sensor " << id << " en " << zona
                    << ": " << lectura << std::endl;
    }
};
```

Intenta escribir ese constructor con asignaciones en el cuerpo y el compilador te lo impedirá: `id = i;` es un error porque `id` es const, y no hay forma de "reasignar" una referencia. Esta es la razón profunda por la que la lista existe: hay miembros que solo pueden nacer con su valor final. Un consejo de diseño: las referencias miembro son poderosas pero peligrosas, porque el objeto queda atado a la vida del original (si la zona muere antes que el sensor, la referencia cuelga). Úsalas cuando la relación sea claramente "este objeto no tiene sentido sin ese otro", y documenta quién vive más.

Miembros objeto: la lista construye en cadena

Cuando un miembro es un objeto de otra clase con constructor con parámetros, la lista es la única forma de pasarle esos parámetros:

```
class Motor {
public:
    Motor(int cilindros) { /* ... */ }
};

class Auto {
private:
    Motor motor;
public:
    Auto(int c) : motor(c) {} // sin esto, no compila: Motor no tiene constructor
    por defecto
};
```

Sin la lista, el compilador intentaría construir motor con un constructor por defecto que no existe, y fallaría. Este es el pegamento de la composición (que ya viste en la segunda parte): el objeto contenedor inicializa a sus partes en su propia lista. Observa cómo todo encaja: la lista de inicialización es el mecanismo que hace posible construir objetos dentro de objetos de forma controlada.

Ejemplo completo 1: lista de inicialización bien usada

```

#include <iostream>
#include <string>

class CuentaAhorro {
private:
    const std::string numero; // const: obligatoria la lista
    const double tasa;        // const: obligatoria la lista
    double saldo;
    std::string titular;      // objeto: mejor inicializar que asignar
public:
    // Orden de la lista = orden de declaracion: numero, tasa, saldo, titular
    CuentaAhorro(const std::string& num, const std::string& tit,
                 double t, double inicial)
        : numero(num), tasa(t), saldo(inicial), titular(tit) {
        // El cuerpo queda para logica, no para inicializar
        if (saldo < 0) saldo = 0;
        std::cout << "Cuenta " << numero << " creada." << std::endl;
    }

    void aplicarInteres() {
        saldo += saldo * tasa;
    }

    void mostrar() const {
        std::cout << titular << " [" << numero << "]: $"
                  << saldo << " (tasa " << tasa * 100 << "%)" << std::endl;
    }
};

int main() {
    CuentaAhorro c("AH-001", "Esteban", 0.05, 1000.0);
    c.mostrar();
    c.aplicarInteres();
    c.mostrar();
    // c.tasa = 0.1; // ERROR: tasa es const, ni el propio objeto la cambia
    return 0;
}

```

Analiza las decisiones: numero y tasa son const porque una cuenta no cambia de número ni de tasa por capricho; eso los obliga a la lista, lo cual es bueno porque documenta la intención. titular es std::string y se inicializa en la lista para evitar la construcción doble. El orden de la lista coincide con el de declaración, eliminando la trampa del orden. El cuerpo solo valida (saldo negativo a cero) e informa. Y la línea comentada muestra el beneficio del const: ni

siquiera el propio código puede corromper la tasa por accidente. Esta clase es pequeña pero está blindada por diseño, no por suerte.

Delegación de constructores: no repitas tu lógica

Cuando una clase tiene varios constructores, es tentador copiar la lógica de validación e inicialización en cada uno. El resultado es código duplicado que se desincroniza: arreglas un bug en un constructor y se te olvida en los otros dos. Desde C++11, un constructor puede llamar a otro constructor de la misma clase en su lista de inicialización. Eso es delegar: un constructor principal concentra toda la lógica y los demás solo le pasan valores por defecto.

```
class Empleado {
private:
    std::string nombre;
    std::string puesto;
    double salario;
public:
    // Constructor principal: TODA la logica vive aqui
    Empleado(const std::string& n, const std::string& p, double s)
        : nombre(n), puesto(p), salario(s) {
        if (salario < 0) salario = 0; // validacion en un solo lugar
        std::cout << "Empleado creado: " << nombre << std::endl;
    }

    // Delegan: no repiten ni una linea de logica
    Empleado(const std::string& n, const std::string& p)
        : Empleado(n, p, 0.0) {}

    Empleado(const std::string& n)
        : Empleado(n, "Por asignar", 0.0) {}

    Empleado()
        : Empleado("Sin nombre") {}

    void mostrar() const {
        std::cout << nombre << " | " << puesto << " | $" << salario << std::endl;
    }
};
```

La sintaxis es : Empleado(argumentos) en la lista de inicialización del constructor que delega. El constructor delegado se ejecuta completo (incluyendo su cuerpo) antes de que el que delega haga nada más. Hay reglas: un constructor que delega no puede inicializar otros

miembros en su propia lista (toda la inicialización la hace el delegado); no puede haber ciclos (A delega en B y B delega en A es un error); y la delegación es la única forma correcta de "llamar" a otro constructor (escribir `Empleado(n, p, 0.0)`; dentro del cuerpo crearía un objeto temporal desechable, no inicializaría el actual: un bug clásico de quienes no conocen la delegación).

El patrón recomendado es tener un constructor "maestro" con todos los parámetros y la validación, y que los demás sean atajos que delegan con valores por defecto sensatos. Así, cuando cambie una regla de negocio (por ejemplo, el salario mínimo), la cambias una vez. La delegación es DRY (Don't Repeat Yourself) aplicado a constructores.

Constructores explicit: freno a las conversiones silenciosas

Un constructor con un solo parámetro (o con varios pero todos con valores por defecto) le permite al compilador convertir implícitamente: si una función espera un objeto de tu clase y le pasas un valor del tipo del parámetro, el compilador construye el objeto solo, sin que nadie se lo pida. A veces es cómodo; a menudo es un bug disfrazado.

```
class Metros {
private:
    double valor;
public:
    Metros(double v) : valor(v) {} // permite conversion implicita
    double get() const { return valor; }
};

void imprimirDistancia(Metros m) {
    std::cout << "Distancia: " << m.get() << " m" << std::endl;
}

int main() {
    imprimirDistancia(10.0); // ¡compila! el 10.0 se convierte solo a Metros
    return 0;
}
```

Ese `imprimirDistancia(10.0)` compila porque el compilador ve que puede fabricar un `Metros` a partir de un `double` y lo hace en silencio. En este ejemplo es inofensivo, pero imagina una clase `Dinero` con constructor `Dinero(double)`: una función `pagar(Dinero)` aceptaría `pagar(100)` y nadie notaría la conversión. Peor aún: con dos constructores de un parámetro (`Metros(double)` y `Pulgadas(double)`), una llamada ambigua puede resolverse de formas sorprendentes. Las conversiones implícitas son el compilador adivinando tu intención, y adivinar es peligroso.

La palabra `explicit` prohíbe esas conversiones silenciosas: el constructor sigue existiendo, pero solo se usa cuando lo invocas directamente.

```
class Metros {
private:
    double valor;
public:
    explicit Metros(double v) : valor(v) {}
    double get() const { return valor; }
};

int main() {
    // imprimirDistancia(10.0); // ERROR: conversion implicita bloqueada
    imprimirDistancia(Metros(10.0)); // OK: conversion visible e intencional
    Metros m = 5.0; // ERROR también: la inicializacion con = es implicita
    Metros n(5.0); // OK: directa y explicita
    Metros o{5.0}; // OK: con llaves también es directa
    return 0;
}
```

La regla profesional: marca `explicit` todo constructor de un solo parámetro, salvo que quieras deliberadamente la conversión implícita. Los casos donde la conversión implícita sí se quiere son pocos y famosos: `std::string` se construye implícitamente desde `const char*` (por eso puedes pasar "hola" donde esperan string), y las clases que modelan valores numéricos intercambiables a veces lo permiten. Para todo lo demás, `explicit`. Es una de esas decisiones que no cuestan nada al escribir y ahorran horas de depuración: cada conversión visible en el código es una intención documentada.

Nota que `explicit` también afecta a la inicialización con `=`: `Metros m = 5.0;` usa conversión implícita y queda bloqueada, mientras que `Metros m(5.0);` y `Metros m{5.0};` son directas y permitidas. Si vienes de código viejo sin `explicit`, agregar la palabra puede romper compilaciones que dependían de conversiones silenciosas: esa es precisamente la señal de que había intenciones ocultas en el código.

El constructor de copia en detalle

Ya sabes que al copiar un objeto (`Texto b = a;`) se invoca el constructor de copia. En la segunda parte viste la advertencia: el que genera el compilador hace copia superficial (miembro por miembro), peligrosa con punteros. Ahora lo verás con un bug real, ejecutable, para que sientas el problema antes de aprender la solución.

El bug: copia superficial con memoria propia

```
#include <iostream>
#include <cstring>

class Texto {
private:
    char* datos;
public:
    Texto(const char* t) {
        datos = new char[std::strlen(t) + 1];
        std::strcpy(datos, t);
    }
    ~Texto() { delete[] datos; }

    void mostrar() const { std::cout << datos << std::endl; }
    void cambiarPrimera(char c) { datos[0] = c; }
};

void procesar(Texto t) { // ¡aquí se copia! (parametro por valor)
    t.cambiarPrimera('X');
    t.mostrar(); // muestra "Xola"
} // ¡aquí muere la copia y su destructor libera la memoria!

int main() {
    Texto original("Hola");
    procesar(original); // la copia superficial comparte el buffer
    original.mostrar(); // memoria ya liberada: BASURA o caída
    return 0;           // doble delete al morir 'original': BUG
}
```

Sigue la tragedia paso a paso. Al llamar a `procesar(original)`, el constructor de copia generado por el compilador copia el puntero `datos` miembro por miembro: ahora dos objetos apuntan al mismo buffer. Dentro de `procesar`, `cambiarPrimera` modifica el buffer compartido (el original también lo "siente", aunque no debería). Al salir de `procesar`, el destructor de la copia hace `delete[]` del buffer. De vuelta en `main`, `original.datos` es un puntero colgante: `mostrar()` lee memoria liberada. Y al terminar `main`, el destructor de `original` hace `delete[]` otra vez sobre la misma memoria: doble liberación, comportamiento indefinido. Compila este programa con `-fsanitize=address` (lo aprendiste en la unidad 5) y verás el reporte exacto: `heap-use-after-free` seguido de intento de doble liberación. Es el bug de copias más famoso de C++ y ha causado vulnerabilidades de seguridad reales en software de verdad.

La corrección: copia profunda

La solución es escribir tu propio constructor de copia que reserve memoria nueva y copie el contenido, no el puntero. Eso se llama copia profunda: el nuevo objeto es independiente del original.

```
#include <iostream>
#include <cstring>

class Texto {
private:
    char* datos;
public:
    Texto(const char* t) {
        datos = new char[std::strlen(t) + 1];
        std::strcpy(datos, t);
    }

    // CONSTRUCTOR DE COPIA: copia profunda
    Texto(const Texto& otro) {
        datos = new char[std::strlen(otro.datos) + 1];
        std::strcpy(datos, otro.datos);
        std::cout << "Copia profunda realizada." << std::endl;
    }

    ~Texto() { delete[] datos; }

    void mostrar() const { std::cout << datos << std::endl; }
    void cambiarPrimera(char c) { datos[0] = c; }
};

void procesar(Texto t) {
    t.cambiarPrimera('X');
    t.mostrar(); // "Xola": solo la copia cambia
} // la copia muere liberando SU propia memoria

int main() {
    Texto original("Hola");
    procesar(original);
    original.mostrar(); // "Hola": intacto, sin doble delete
    Texto copia = original; // constructor de copia explicito
    copia.mostrar();       // "Hola", en memoria independiente
    return 0;
}
```

La firma es siempre `const Texto&` otro: referencia constante porque copiar no debe modificar el original, y referencia (no valor) porque pasarlo por valor invocaría al constructor de copia recursivamente hasta desbordar la pila. Con la copia profunda, cada objeto tiene su propio buffer: `procesar()` modifica solo su copia, cada destructor libera su propia memoria y el programa termina limpio. Verifica con el sanitizer: cero reportes.

Cuándo se invoca el constructor de copia

No solo con `Texto b = a;`. El constructor de copia se invoca en cuatro situaciones: al inicializar un objeto con otro (`Texto b = a;` o `Texto b(a);`), al pasar un objeto por valor a una función, al devolver un objeto por valor desde una función, y al lanzar y atrapar excepciones por valor. Por eso el bug de arriba aparecía "solo" al llamar a `procesar`: el paso por valor era la copia silenciosa. Regla de oro: si tu clase maneja recursos (memoria, archivos, conexiones), o escribes el constructor de copia correctamente, o evitas las copias pasando por referencia constante. Lo segundo es más barato; lo primero es obligatorio cuando las copias deben existir.

Ejemplo completo 2: el bug y su corrección, frente a frente

Para que la lección quede grabada, aquí está la versión mínima que demuestra el problema y la solución en un solo programa conceptual. La clase `Mala` usa la copia del compilador; la clase `Buena` define la suya. (En código real jamás tendrías ambas: es un laboratorio, no un diseño.)

```

#include <iostream>
#include <cstring>

class Buena {
private:
    char* datos;
public:
    Buena(const char* t) {
        datos = new char[std::strlen(t) + 1];
        std::strcpy(datos, t);
    }
    Buena(const Buena& otro) { // copia profunda
        datos = new char[std::strlen(otro.datos) + 1];
        std::strcpy(datos, otro.datos);
    }
    Buena& operator=(const Buena& otro) { // asignacion profunda (ver siguiente
seccion)
        if (this != &otro) {
            delete[] datos;
            datos = new char[std::strlen(otro.datos) + 1];
            std::strcpy(datos, otro.datos);
        }
        return *this;
    }
    ~Buena() { delete[] datos; }
    void mostrar() const { std::cout << datos << std::endl; }
};

int main() {
    Buena original("Funciona");
    Buena copia = original; // constructor de copia
    Buena otra("Temporal");
    otra = original; // operador de asignacion
    original.mostrar();
    copia.mostrar();
    otra.mostrar(); // las tres muestran "Funciona", cada una con su
memoria
    return 0; // tres destructores, tres liberaciones, cero dramas
}

```

Tres objetos, tres buffers independientes, tres liberaciones limpias. Este es el comportamiento correcto que el compilador no puede adivinar por ti cuando hay recursos de por medio. Y ya asoma el operador= en el ejemplo: copiar al construir y copiar al asignar son dos operaciones distintas que necesitan sus propias versiones profundas. Esa es la siguiente sección.

El operador de asignación: copiar sin nacer

El constructor de copia crea un objeto nuevo a partir de otro. El operador de asignación (operator=) copia un objeto sobre otro que YA EXISTE. Son operaciones distintas y confundirlas es fuente de bugs: Texto b = a; usa el constructor de copia (b nace aquí), mientras que b = a; con b ya existente usa el operador de asignación.

```
class Texto {
private:
    char* datos;
public:
    // ... constructor y constructor de copia (copia profunda) ...

    Texto& operator=(const Texto& otro) {
        if (this != &otro) {           // 1) proteccion contra autoasignacion
            delete[] datos;           // 2) libero mi memoria vieja
            datos = new char[std::strlen(otro.datos) + 1]; // 3) reservo nueva
            std::strcpy(datos, otro.datos); // 4) copio el contenido
        }
        return *this;                  // 5) devuelvo *this para encadenar
    }
};
```

La anatomía tiene cinco piezas con su porqué. Primera: la guarda de autoasignación (if (this != &otro)). Sin ella, a = a; liberaría la memoria de a y luego intentaría copiar desde la memoria recién liberada: un desastre. La autoasignación parece absurda hasta que aparece disfrazada (a = b donde b resulta ser un alias de a, o a[i] = a[j] con i == j). Segunda: liberar la memoria vieja antes de reservar la nueva; si lo haces al revés y el new falla (excepción), habrás perdido el puntero original con su memoria: fuga. Tercera y cuarta: reserva y copia profunda, igual que en el constructor de copia. Quinta: devolver *this por referencia permite encadenar: a = b = c; funciona porque (b = c) devuelve b y luego a = (resultado).

El orden "liberar primero, reservar después" tiene una debilidad teórica: si el new lanza una excepción, el objeto queda con datos en un puntero liberado (aunque en la práctica, con la guarda y el delete previo, el objeto queda en estado válido pero vacío si asignas nullptr después del delete). La técnica profesional para blindarlo es el idioma copy-and-swap: crear una copia temporal del otro objeto y luego intercambiar los punteros. Si algo falla durante la copia, el objeto original ni se entera. Es un patrón avanzado que vale conocer de nombre: cuando veas operator= implementado con una copia local y un swap, sabrás que está hecho a prueba de excepciones.

Un error frecuente es olvidar el `return *this` o devolver por valor (`Texto` en vez de `Texto&`). Devolver por valor compila pero hace una copia innecesaria en cada asignación encadenada y rompe la semántica esperada. La firma canónica es siempre `T& operator=(const T&)`: referencia de entrada, referencia de salida.

La regla de tres, la de cinco y la de cero

Con lo que ya sabes, emerge un patrón famoso. Si tu clase necesita un destructor personalizado (porque libera memoria, cierra archivos, etc.), casi seguro necesita también un constructor de copia y un operador de asignación personalizados. Esa es la regla de tres: destructor, constructor de copia y operador de asignación van juntos o no van. La lógica es simple: si el destructor hace algo especial, es porque la clase posee un recurso; y si posee un recurso, copiar el objeto miembro por miembro (lo que hace el compilador por defecto) compartirá ese recurso incorrectamente. La clase `Texto` es el ejemplo perfecto: sin los tres, hay doble delete.

C++11 agregó la semántica de movimiento (mover recursos en vez de copiarlos, tema de unidades avanzadas), y la regla creció a la regla de cinco: destructor, constructor de copia, operador de asignación por copia, constructor de movimiento y operador de asignación por movimiento. Si defines alguno de los cinco, piensa en los cinco. El movimiento permite, por ejemplo, devolver un `Texto` desde una función sin copiar el buffer: se transfiere la propiedad del puntero y el original queda vacío. Es una optimización con reglas estrictas que merece su propia unidad.

Pero la regla más importante es la moderna: la regla de cero. Dice que tus clases no deberían manejar recursos crudos directamente: delega en clases que ya lo hacen bien (`std::string`, `std::vector`, `std::unique_ptr`). Si `Texto` guardara `std::string` en vez de `char*`, no necesitaría destructor, ni constructor de copia, ni `operator=`: el compilador generaría los tres correctos automáticamente, porque `std::string` ya sabe copiarse profundamente. La regla de cero no significa "no pienses en copias"; significa "diseña para no necesitar código personalizado".

| Situación | Qué hacer | |---|---| | Miembros solo con tipos que se copian solos (`int`, `string`, `vector`) | Regla de cero: no escribas nada, el compilador lo hace bien | | La clase posee un recurso crudo (`new`, `archivo`, `conexión`) | Regla de tres como mínimo: destructor + copia + asignación | | Además quieres mover eficientemente (C++11 en adelante) | Regla de cinco: agrega las dos operaciones de movimiento | | Recurso exclusivo no copiable | Prohíbe la copia (ver ejercicios) y permite solo movimiento |

Memoriza la tabla al revés también: si te encuentras escribiendo un destructor que hace delete, detente y escribe inmediatamente el constructor de copia y el operador=. Los tres nacen juntos. Y antes de escribirlos, pregúntate si un std::string, un std::vector o un std::unique_ptr podrían poseer ese recurso por ti: nueve de cada diez veces, la respuesta es sí y la regla de cero te ahorra los tres.

Ejemplo: clase que cumple la regla de tres

```
#include <iostream>

class ArregloSeguro {
private:
    int* datos;
    int tam;
public:
    explicit ArregloSeguro(int n) : tam(n) {
        datos = new int[tam];
        for (int i = 0; i < tam; i++) datos[i] = 0;
    }

    // 1 de 3: constructor de copia (profunda)
    ArregloSeguro(const ArregloSeguro& otro) : tam(otro.tam) {
        datos = new int[tam];
        for (int i = 0; i < tam; i++) datos[i] = otro.datos[i];
    }

    // 2 de 3: operador de asignacion (profundo)
    ArregloSeguro& operator=(const ArregloSeguro& otro) {
        if (this != &otro) {
            delete[] datos;
            tam = otro.tam;
            datos = new int[tam];
            for (int i = 0; i < tam; i++) datos[i] = otro.datos[i];
        }
        return *this;
    }

    // 3 de 3: destructor
    ~ArregloSeguro() {
        delete[] datos;
    }

    void set(int i, int v) {
        if (i >= 0 && i < tam) datos[i] = v;
    }
};
```

```

    }
    int get(int i) const {
        return (i >= 0 && i < tam) ? datos[i] : 0;
    }
    int size() const { return tam; }
};

int main() {
    ArregloSeguro a(5);
    a.set(0, 42);
    ArregloSeguro b = a;    // copia profunda: b tiene su propia memoria
    b.set(0, 99);
    ArregloSeguro c(3);
    c = a;                  // asignacion profunda
    std::cout << a.get(0) << " " << b.get(0) << " " << c.get(0) << std::endl;    // 42
99 42
    return 0;    // tres destructores, cada uno libera lo suyo
}

```

Los tres métodos especiales están juntos y son consistentes: todos hacen copia profunda, todos emparejan new[] con delete[]. El main demuestra la independencia: modificar b no afecta a a, y asignar a sobre c reemplaza su contenido sin fugas (la memoria vieja de c se libera primero). Observa también el explicit en el constructor: ArregloSeguro(int) con un solo parámetro podría permitir conversiones implícitas desde int, así que lo blindamos (conexión directa con la sección de explicit). Esta clase es el molde mental para cualquier recurso: si algún día escribes una clase que maneje un archivo o una conexión, su esqueleto será este mismo.

Agregación vs composición: quién posee a quién

En la segunda parte viste la composición: un Auto TIENE UN Motor que nace y muere con él. Pero no todas las relaciones de "tiene un" son iguales. La agregación es la prima de la composición: el todo también tiene partes, pero las partes existen por su cuenta y sobreviven al todo. La diferencia es de propiedad y ciclo de vida, y elegir mal genera diseños rígidos o fugas conceptuales.

Composición (el todo posee las partes): el Motor se construye dentro del Auto y muere con él. Se implementa con miembros objeto directos (Motor motor;) o con punteros que el todo crea y destruye. Pregunta de diagnóstico: "si destruyo el todo, ¿tiene sentido que la parte siga existiendo?" Si la respuesta es no, es composición. Una Casa y sus Cuartos: al demoler la casa, los cuartos dejan de existir como tales.

Agregación (el todo usa partes prestadas): un Departamento TIENE Profesores, pero los profesores existen antes de entrar al departamento y siguen existiendo si el departamento desaparece. Se implementa con punteros o referencias a objetos creados fuera: el todo no los construye ni los destruye. Pregunta de diagnóstico: "si destruyo el todo, ¿la parte sigue viva?" Si sí, es agregación. Un Equipo y sus Jugadores: si el equipo se disuelve, los jugadores siguen existiendo.

La diferencia en código es quién hace new y delete (o, mejor, quién es dueño del objeto). En composición, el dueño es el todo. En agregación, el dueño es alguien externo y el todo solo guarda referencias. Con punteros inteligentes modernos la distinción se vuelve explícita: composición con `unique_ptr` (propiedad exclusiva) y agregación con punteros crudos observadores o `shared_ptr` según el caso.

Ejemplo completo 3: agregación y composición lado a lado

```
#include <iostream>
#include <string>
#include <vector>
#include <memory>

// ----- COMPOSICION: la Pagina nace y muere con el Libro -----
class Pagina {
private:
    int numero;
public:
    explicit Pagina(int n) : numero(n) {}
    int getNumero() const { return numero; }
};

class Libro {
private:
    std::string titulo;
    std::vector<Pagina> paginas; // composicion: el libro POSEE sus paginas
public:
    Libro(const std::string& t, int n) : titulo(t) {
        for (int i = 1; i <= n; i++) paginas.push_back(Pagina(i));
    }
    void mostrar() const {
        std::cout << "\"" << titulo << "\" con " << paginas.size()
            << " paginas." << std::endl;
    }
}; // al morir el libro, mueren sus paginas: nadie mas las tenia
```

```
// ----- AGREGACION: el Autor existe sin la Editorial -----
class Autor {
private:
    std::string nombre;
public:
    explicit Autor(const std::string& n) : nombre(n) {}
    std::string getNombre() const { return nombre; }
};

class Editorial {
private:
    std::string nombre;
    std::vector<const Autor*> autores; // agregacion: solo los referencia
public:
    explicit Editorial(const std::string& n) : nombre(n) {}
    void contratar(const Autor* a) { autores.push_back(a); }
    void mostrar() const {
        std::cout << "Editorial " << nombre << " publica a:" << std::endl;
        for (const auto* a : autores) {
            std::cout << "    - " << a->getNombre() << std::endl;
        }
    }
}; // al morir la editorial, los autores siguen vivos

int main() {
    Libro libro("C++ desde cero", 300);
    libro.mostrar();

    Autor a1("Elena"), a2("Marcos"); // los autores viven en el main
    Editorial ed("Colibri Press");
    ed.contratar(&a1);
    ed.contratar(&a2);
    ed.mostrar();
    // ed muere aqui; a1 y a2 siguen vivos: eso es agregacion
    std::cout << "Los autores siguen vivos: "
        << a1.getNombre() << ", " << a2.getNombre() << std::endl;
    return 0;
}
```

Compara las dos relaciones. Libro crea sus Pagina en el constructor y las guarda por valor en un vector: nadie fuera del libro las conoce ni puede usarlas; cuando el libro muere, el vector destruye las páginas con él. Editorial, en cambio, recibe punteros a Autor creados en el main: no los construye, no los destruye, solo los usa. El vector<const Autor*> documenta la intención: punteros a objetos constantes que no me pertenecen (el const promete no modificarlos, y el puntero crudo avisa "no soy el dueño, no me liberes"). En C++ moderno,

cuando veas un puntero crudo como miembro, casi siempre es agregación u observación: la propiedad real vive en otro lado, idealmente en un `unique_ptr`, un vector o una variable local con el ciclo de vida adecuado.

Clases inmutables: objetos que no cambian

Una clase inmutable es aquella cuyos objetos no pueden modificarse después de crearse. Todos sus miembros son `const` (o efectivamente de solo lectura: sin setters y sin métodos que muten), y cualquier "cambio" se expresa creando un objeto nuevo. Suena restrictivo, pero la inmutabilidad compra tres cosas valiosas: seguridad (nadie puede corromper el objeto por accidente), facilidad de razonamiento (si lo creaste con ciertos valores, esos valores valen para siempre) y compatibilidad con concurrencia (los objetos inmutables se comparten entre hilos sin candados).

El diseño es directo: miembros `const` inicializados en la lista, solo getters y métodos `const`, ningún setter. Las operaciones que conceptualmente "modifican" devuelven un objeto nuevo:

```
class PuntoImmutable {
private:
    const double x;
    const double y;
public:
    PuntoImmutable(double px, double py) : x(px), y(py) {}

    double getX() const { return x; }
    double getY() const { return y; }

    // "Mover" el punto devuelve un punto NUEVO; el original no cambia
    PuntoImmutable trasladado(double dx, double dy) const {
        return PuntoImmutable(x + dx, y + dy);
    }

    double distanciaAlOrigen() const {
        return std::sqrt(x * x + y * y);
    }

    void mostrar() const {
        std::cout << "(" << x << ", " << y << ")" << std::endl;
    }
};
```

Observa que `trasladado()` es `const` y devuelve por valor un objeto nuevo: el punto original queda intacto. Este estilo (métodos que devuelven versiones transformadas) es el corazón de la programación funcional y aparece en bibliotecas modernas. Un detalle técnico: como los miembros son `const`, la clase inmutable no puede usar el operador de asignación por defecto (no se puede reasignar un `const`), así que los objetos inmutables se copian al construir pero no se reasignan. En la práctica esto rara vez molesta: si necesitas "otro valor", construyes otro objeto.

La inmutabilidad también puede ser parcial: una clase con algunos miembros `const` (un `id`, una fecha de creación) y otros mutables (un contador, un estado). Es el punto medio más común en código real: lo que es identidad del objeto es `const`, lo que es estado cambiante no lo es. Por ejemplo, una `CuentaBancaria` con `numero` `const` y `saldo` mutable. Diseñar así comunica qué partes del objeto son su identidad (nunca cambian) y cuáles son su estado (pueden cambiar), y el compilador hace cumplir la distinción.

Cuándo usar inmutabilidad total: valores matemáticos (puntos, fracciones, fechas, cantidades con unidad), configuraciones que se leen una vez, claves y tokens, y cualquier objeto que vayas a compartir sin querer pensar en quién lo modifica. Cuándo no: entidades con ciclo de vida cambiante (una cuenta bancaria cuyo saldo se mueve, un jugador con posición actualizada cada frame). La inmutabilidad es una herramienta de diseño, no una religión: aplícala donde la estabilidad valga más que la comodidad de mutar.

Namespaces: orden en la casa de clases

Conforme tus programas crecen, los nombres empiezan a chocar: tu clase `Logger` choca con la `Logger` de una biblioteca, tu función `abrir` choca con otra `abrir`. Los namespaces son apellidos para tu código: agrupan clases, funciones y constantes bajo un nombre, evitando colisiones sin obligarte a inventar nombres larguísimos como `MiProyectoLogger`.

```

namespace Geometria {
    class Circulo {
    private:
        double radio;
    public:
        explicit Circulo(double r) : radio(r) {}
        double area() const { return 3.14159 * radio * radio; }
    };

    double diametro(double radio) { return radio * 2; }
}

namespace Finanzas {
    class Cuenta {
    private:
        double saldo;
    public:
        explicit Cuenta(double s) : saldo(s) {}
        double getSaldo() const { return saldo; }
    };
}

int main() {
    Geometria::Circulo c(5.0);
    Finanzas::Cuenta cuenta(1000.0);
    std::cout << "Area: " << c.area() << std::endl;
    std::cout << "Diametro: " << Geometria::diametro(5.0) << std::endl;
    return 0;
}

```

El operador `::` califica el nombre: `Geometria::Circulo` es "el Circulo de Geometria". Dos namespaces pueden tener clases con el mismo nombre sin conflicto (`Geometria::Figura` y `Fisica::Figura` conviven en paz). Los namespaces se pueden anidar (`namespace Empresa::Contabilidad`) y reabrir: puedes agregar más cosas a un namespace en otro archivo con otro bloque `namespace Geometria { ... }`, lo cual es ideal para organizar proyectos grandes en varios archivos.

Tres herramientas relacionadas que debes conocer. Primera: el alias de namespace, para no repetir nombres largos: `namespace Geo = Geometria;` y luego `Geo::Circulo`. Segunda: `using namespace`, que importa todo un namespace al alcance actual: `using namespace Geometria;` permite escribir `Circulo` directamente. Úsalo con moderación: en archivos `.cpp` pequeños es cómodo, pero en cabeceras (`.h`) es una mala práctica grave, porque contamina con nombres a todos los archivos que incluyan tu cabecera y reintroduce las colisiones que el namespace

evitaba. Tercera: el using específico, `using Geometria::Circulo;`, que importa un solo nombre: el punto medio sensato entre escribir el prefijo siempre y abrir todo el namespace.

El namespace anónimo (`namespace { ... }` sin nombre) merece mención: lo que declares ahí es visible solo en ese archivo de código. Es la forma moderna de hacer funciones y variables "privadas del archivo", reemplazando al `static global` de C. Úsalo para ayudantes internos que nadie más necesita ver.

Y ya conoces el namespace más famoso: `std`. Todo lo de la biblioteca estándar (`std::string`, `std::cout`, `std::vector`) vive ahí, por eso siempre escribes el prefijo o un `using`. Cuando veas `using namespace std;` en ejemplos del libro, es una conveniencia didáctica; en tu código profesional, prefiere el prefijo `std::` o usings específicos. Escribir `std::string` cuesta cinco caracteres y evita colisiones misteriosas.

Ejemplo completo 4: dos bibliotecas que chocan, resueltas con namespaces

```

#include <iostream>
#include <string>

// Dos equipos escribieron su propia clase "Sensor" sin coordinarse.
namespace Clima {
    class Sensor {
    private:
        double temperatura;
    public:
        explicit Sensor(double t) : temperatura(t) {}
        double leer() const { return temperatura; }
        std::string unidad() const { return "Celsius"; }
    };
}

namespace Trafico {
    class Sensor {
    private:
        int autosPorMinuto;
    public:
        explicit Sensor(int a) : autosPorMinuto(a) {}
        int leer() const { return autosPorMinuto; }
        std::string unidad() const { return "autos/min"; }
    };
}

int main() {
    Clima::Sensor s1(28.5);    // sin namespaces, estos dos
    Trafico::Sensor s2(42);    // nombres chocarían

    std::cout << "Clima: " << s1.leer() << " " << s1.unidad() << std::endl;
    std::cout << "Trafico: " << s2.leer() << " " << s2.unidad() << std::endl;

    namespace T = Trafico;    // alias para el código que usa mucho un namespace
    T::Sensor s3(60);
    std::cout << "Trafico 2: " << s3.leer() << " " << s3.unidad() << std::endl;
    return 0;
}

```

Sin namespaces, tener dos clases Sensor sería un error de redefinición y alguien tendría que renombrar su clase (SensorClima, SensorTrafico: los "apellidos manuales" que los namespaces hacen innecesarios). Con namespaces, cada equipo trabaja en su propio apellido y el programa principal califica cuál quiere. El alias namespace T = Trafico muestra cómo mantener el código legible cuando un namespace se usa intensivamente. Este es el

mecanismo que permite que bibliotecas de distintos autores convivan en un mismo programa, y por eso todo código C++ serio vive dentro de algún namespace.

Catálogo de errores comunes (tercera parte)

Error 1: inicializar miembros en la lista en distinto orden que su declaración. Compila, pero los miembros se inicializan en orden de declaración. Si uno depende de otro declarado después, usa basura. Escribe la lista en el mismo orden en que declaras los miembros y activa -Wall para que el compilador te avise.

Error 2: intentar asignar miembros const o referencias en el cuerpo del constructor. No compila, y es correcto que no compile: esos miembros solo pueden recibir valor en la lista de inicialización. Si te topas con este error, no es un obstáculo: es el lenguaje empujándote al diseño correcto.

Error 3: "llamar" a otro constructor dentro del cuerpo (`Empleado(n, p, 0.0);` como sentencia). Eso crea un objeto temporal que se destruye inmediatamente; el objeto actual queda sin inicializar. La delegación real solo ocurre en la lista: `: Empleado(n, p, 0.0)`.

Error 4: ciclo de delegación (A delega en B, B delega en A). No compila, pero el mensaje de error puede ser confuso. Diseña una jerarquía clara: un constructor maestro y atajos que delegan hacia él, nunca entre atajos.

Error 5: olvidar `explicit` en constructores de un parámetro. El código compila y "funciona", pero permite conversiones silenciosas que algún día pasarán un `double` donde esperabas un objeto con invariantes. Marca `explicit` por defecto y quítalo solo con razón.

Error 6: escribir el constructor de copia pero olvidar el operador de asignación (o al revés). La regla de tres es un paquete: si el destructor libera un recurso, las dos operaciones de copia deben ser profundas. Una clase con copia profunda y asignación superficial sigue teniendo doble delete por la puerta de atrás.

Error 7: olvidar la guarda de autoasignación en `operator=`. `a = a;` parece imposible hasta que aparece como `arr[i] = arr[j]` con índices iguales o a través de alias. Sin la guarda, liberas la memoria y luego copias desde ella: corrupción garantizada.

Error 8: devolver `void` o por valor en `operator=`. La firma canónica es `T& operator=(const T&)`. Devolver `void` rompe el encadenamiento (`a = b = c`); devolver por valor hace copias inútiles y puede comportarse raro en código genérico.

Error 9: pasar el parámetro del constructor de copia por valor (`Texto(Texto otro)`). Cada llamada al constructor de copia intentaría copiar el parámetro... invocando al constructor de copia recursivamente hasta desbordar la pila. Siempre por referencia constante: `const Texto&`.

Error 10: confundir agregación con composición en la propiedad. Si el todo guarda un puntero a un objeto que no creó y además lo libera en su destructor, habrá doble liberación cuando el dueño real también lo libere. Decide quién es dueño una sola vez y documenta con el tipo: `unique_ptr` para propiedad, puntero crudo observador para agregación.

Error 11: poner `using namespace` en archivos de cabecera. Contamina el espacio de nombres de todo el que incluya tu cabecera y reintroduce colisiones. En `.h` usa nombres calificados (`MiLib::Clase`) o usings específicos; reserva `using namespace` para `.cpp` pequeños y ejemplos didácticos.

Error 12: intentar reasignar un objeto de clase inmutable. Con miembros `const`, el operador de asignación por defecto no existe y `a = b` no compila. No es un bug: es el diseño. Si necesitas "cambiarlo", construye uno nuevo (ese es el estilo inmutable).

Error 13: declarar el parámetro del constructor de copia sin `const` (`Texto(Texto& otro)`). Compila, pero impide copiar objetos constantes y temporales: no podrás hacer `procesar(Texto("hola"))` ni copiar desde una referencia `const`. El `const` en la firma no es decoración: es lo que hace la copia universal.

Práctica: ejercicios con solución

Ejercicio 1: rescata una clase con miembros `const`

La siguiente clase no compila. Explica por qué y corrígela usando la lista de inicialización, cuidando el orden.

```

class Registro {
private:
    const int folio;
    const std::string& responsable;
    double monto;
public:
    Registro(int f, const std::string& r, double m) {
        folio = f;          // ERROR
        responsable = r;    // ERROR
        monto = m;
    }
};

```

Solución:

```

#include <iostream>
#include <string>

class Registro {
private:
    const int folio;
    const std::string& responsable;
    double monto;
public:
    // Orden de la lista = orden de declaracion: folio, responsable, monto
    Registro(int f, const std::string& r, double m)
        : folio(f), responsable(r), monto(m) {}

    void mostrar() const {
        std::cout << "Folio " << folio << " (" << responsable
            << "): $" << monto << std::endl;
    }
};

int main() {
    std::string jefe = "Elena";
    Registro reg(101, jefe, 5000.0);
    reg.mostrar();
    return 0;
}

```

Explicación: folio es const y responsable es referencia: ninguno admite asignación posterior a la construcción, así que el cuerpo con asignaciones no puede compilar. La lista los inicializa al

nacer, que es la única oportunidad. El orden de la lista coincide con el de declaración para evitar la trampa del orden. Nota de diseño: responsable es referencia al string jefe del main, así que jefe debe vivir más que reg; si reg sobreviviera a jefe, la referencia colgaría. Por eso las referencias miembro exigen pensar en ciclos de vida.

Ejercicio 2: blindaje con explicit

Dada esta clase, indica qué líneas compilan y cuáles no después de agregar explicit al constructor. Explica cada caso.

```
class Kilos {
private:
    double valor;
public:
    explicit Kilos(double v) : valor(v) {}
    double get() const { return valor; }
};

void despachar(Kilos k);

int main() {
    despachar(25.0);           // A
    despachar(Kilos(25.0));    // B
    Kilos a = 10.0;           // C
    Kilos b(10.0);             // D
    Kilos c{10.0};             // E
}
```

Solución: la línea A NO compila: pasar 25.0 donde se espera Kilos requeriría una conversión implícita, bloqueada por explicit. La B SÍ compila: la conversión es explícita y visible, exactamente lo que explicit permite. La C NO compila: la inicialización con = es una forma de conversión implícita. La D SÍ compila: construcción directa con paréntesis. La E SÍ compila: construcción directa con llaves. La lección: explicit no prohíbe construir el objeto, prohíbe que el compilador lo construya a escondidas. Cada construcción permitida deja una huella visible en el código (Kilos(...)), documentando la intención.

Ejercicio 3: regla de tres para un Buffer

Implementa la clase Buffer que maneja un arreglo dinámico de char de tamaño fijo: constructor con tamaño, constructor de copia profunda, operador de asignación profundo y destructor. Incluye métodos set/get con verificación de límites. Demuestra en el main que copiar no comparte memoria.

Solución:

```
#include <iostream>

class Buffer {
private:
    char* datos;
    int tam;
public:
    explicit Buffer(int n) : tam(n) {
        datos = new char[tam];
        for (int i = 0; i < tam; i++) datos[i] = 0;
    }

    Buffer(const Buffer& otro) : tam(otro.tam) { // 1: copia profunda
        datos = new char[tam];
        for (int i = 0; i < tam; i++) datos[i] = otro.datos[i];
    }

    Buffer& operator=(const Buffer& otro) { // 2: asignacion profunda
        if (this != &otro) {
            delete[] datos;
            tam = otro.tam;
            datos = new char[tam];
            for (int i = 0; i < tam; i++) datos[i] = otro.datos[i];
        }
        return *this;
    }

    ~Buffer() { delete[] datos; } // 3: destructor

    void set(int i, char c) {
        if (i >= 0 && i < tam) datos[i] = c;
    }
    char get(int i) const {
        return (i >= 0 && i < tam) ? datos[i] : 0;
    }
    int size() const { return tam; }
};
```

```

int main() {
    Buffer b1(5);
    b1.set(0, 'H');
    Buffer b2 = b1;          // constructor de copia
    b2.set(0, 'X');
    Buffer b3(5);
    b3 = b1;                // operador de asignacion
    b3.set(0, 'Y');
    std::cout << b1.get(0) << b2.get(0) << b3.get(0) << std::endl; // HXY
    b1 = b1;                // autoasignacion: no debe romper nada
    std::cout << "Autoasignacion OK: " << b1.get(0) << std::endl;
    return 0;
}

```

Explicación: los tres métodos especiales van juntos (regla de tres) y los tres hacen copia profunda con `new[]/delete[]` emparejados. El main prueba los tres caminos de copia: construcción-copia (b2), asignación (b3) y autoasignación (b1 = b1), que la guarda this != &otro maneja sin corromper. La salida HXY demuestra la independencia: cada objeto tiene su propio buffer. Los set/get verifican límites, un hábito que evita los desbordamientos de la unidad 5. Compíllalo con -fsanitize=address para confirmar cero fugas y cero dobles liberaciones.

Ejercicio 4: agregación o composición, y en código

Modela una Universidad con Facultades y Estudiantes cumpliendo: (a) cada Facultad pertenece a una sola Universidad y muere con ella (composición); (b) cada Estudiante puede estar inscrito en Facultades y sigue existiendo si la Facultad desaparece (agregación). Implementa ambas relaciones y demuéstralas en el main.

Solución:

```

#include <iostream>
#include <string>
#include <vector>

class Estudiante {
private:
    std::string nombre;
public:
    explicit Estudiante(const std::string& n) : nombre(n) {}
    std::string getNombre() const { return nombre; }
};

class Facultad {
private:

```



```

    std::string nombre;
    std::vector<const Estudiante*> inscritos; // AGREGACION: no los posee
public:
    explicit Facultad(const std::string& n) : nombre(n) {}
    void inscribir(const Estudiante* e) { inscritos.push_back(e); }
    void mostrar() const {
        std::cout << "Facultad " << nombre << ":" << std::endl;
        for (const auto* e : inscritos)
            std::cout << "  - " << e->getNombre() << std::endl;
    }
};

class Universidad {
private:
    std::string nombre;
    std::vector<Facultad> facultades; // COMPOSICION: las posee
public:
    explicit Universidad(const std::string& n) : nombre(n) {}
    void agregarFacultad(const std::string& f) {
        facultades.push_back(Facultad(f));
    }
    Facultad& getFacultad(int i) { return facultades.at(i); }
    void mostrar() const {
        std::cout << "Universidad " << nombre << " con "
            << facultades.size() << " facultades." << std::endl;
    }
};

int main() {
    Estudiante e1("Ana"), e2("Luis"); // viven en el main, dueños reales
    {
        Universidad u("UNAM");
        u.agregarFacultad("Ingenieria");
        u.getFacultad(0).inscribir(&e1);
        u.getFacultad(0).inscribir(&e2);
        u.getFacultad(0).mostrar();
        u.mostrar();
    } // u muere: sus facultades mueren con ella (composicion)
    // pero los estudiantes siguen vivos (agregacion):
    std::cout << "Sobreviven: " << e1.getNombre()
        << ", " << e2.getNombre() << std::endl;
    return 0;
}

```

Explicación: la Universidad guarda Facultades por valor en un vector: las crea (agregarFacultad), nadie más las conoce y mueren con ella al cerrar el bloque. Eso es composición. La Facultad guarda punteros const a Estudiante creados en el main: los usa pero no los destruye. Al morir la universidad, e1 y e2 siguen vivos y se pueden nombrar: eso es agregación. El tipo `vector<const Estudiante*>` documenta "observo, no poseo". Si la Facultad hiciera delete de esos punteros, habría doble liberación cuando el main terminara: el error 10 del catálogo.

Ejercicio 5: fracción inmutable

Diseña la clase `FraccionInmutable` con `numerador` y `denominador` const (el denominador nunca cero, validado en el constructor). Sin setters. Incluye un método `sumar` que devuelva una `FraccionInmutable` nueva y un método `mostrar()`. Demuestra que el original no cambia al sumar.

Solución:

```

#include <iostream>

class FraccionInmutable {
private:
    const int num;
    const int den;
public:
    FraccionInmutable(int n, int d) : num(n), den(d == 0 ? 1 : d) {
        if (d == 0) std::cout << "Denominador cero corregido a 1." << std::endl;
    }

    int getNum() const { return num; }
    int getDen() const { return den; }

    FraccionInmutable sumar(const FraccionInmutable& otra) const {
        int n = num * otra.den + otra.num * den;
        int d = den * otra.den;
        return FraccionInmutable(n, d); // objeto NUEVO, el original intacto
    }

    void mostrar() const {
        std::cout << num << "/" << den << std::endl;
    }
};

int main() {
    const FraccionInmutable a(1, 2);
    const FraccionInmutable b(1, 3);
    FraccionInmutable c = a.sumar(b); // 5/6
    std::cout << "a = "; a.mostrar(); // 1/2: sin cambios
    std::cout << "b = "; b.mostrar(); // 1/3: sin cambios
    std::cout << "a + b = "; c.mostrar();
    // a = b; // ERROR: con miembros const no hay asignacion. Es el diseño.
    return 0;
}

```

Explicación: num y den son const e inicializados en la lista; el constructor corrige el denominador cero (un invariante protegido desde el nacimiento). sumar() es const y devuelve por valor una fracción nueva: el estilo inmutable. Los objetos se declaran const FraccionInmutable porque ni siquiera necesitan la posibilidad de mutar. La línea comentada muestra la consecuencia del diseño: sin operador de asignación disponible, "reasignar" no existe; si quieres otro valor, construyes otro objeto. Para valores matemáticos como fracciones, fechas o cantidades, esta rigidez es una virtud: elimina categorías enteras de bugs.

Ejercicio 6: dos bibliotecas, un programa

Dos bibliotecas definen cada una su clase Reloj (una para hora local, otra para cronómetro).
Escribe ambas dentro de namespaces distintos, úsalas en el mismo main sin colisiones, y usa un alias de namespace para la que más se repita.

Solución:

```

#include <iostream>

namespace HoraLocal {
    class Reloj {
    private:
        int hora, minuto;
    public:
        Reloj(int h, int m) : hora(h), minuto(m) {}
        void mostrar() const {
            std::cout << hora << ":" << (minuto < 10 ? "0" : "")
                << minuto << std::endl;
        }
    };
}

namespace Cronometro {
    class Reloj {
    private:
        int segundos;
    public:
        explicit Reloj(int s) : segundos(s) {}
        void mostrar() const {
            std::cout << segundos << " segundos" << std::endl;
        }
    };
}

int main() {
    HoraLocal::Reloj actual(10, 5);
    Cronometro::Reloj carrera(125);

    namespace HL = HoraLocal; // alias: menos escritura, misma claridad
    HL::Reloj cena(20, 30);

    std::cout << "Ahora: "; actual.mostrar();
    std::cout << "Carrera: "; carrera.mostrar();
    std::cout << "Cena: "; cena.mostrar();
    return 0;
}

```

Explicación: sin namespaces, dos clases Reloj serían una redefinición y no compilaría. Cada namespace es un apellido que desambigua: HoraLocal::Reloj y Cronometro::Reloj son tipos distintos que conviven. El alias HL = HoraLocal acorta el código que usa mucho un namespace sin recurrir a using namespace (que importaría todo y podría reintroducir colisiones). Observa

el explicit en el cronómetro: Reloj(int) de un parámetro quedaría expuesto a conversiones implícitas si no lo blindamos, conectando con la sección de explicit. Los namespaces no son decoración: son la herramienta que permite componer software de múltiples autores sin que los nombres choquen.

Resumen de la tercera parte

- La lista de inicialización construye miembros directamente (más eficiente que asignar), es obligatoria para miembros const y referencias, y sigue el orden de declaración, no el de la lista: escríbela en ese orden.
- Un constructor puede delegar en otro con : Clase(args) en su lista; concentra la lógica en un constructor maestro y evita duplicarla.
- Marca explicit los constructores de un parámetro salvo que quieras la conversión implícita: convierte bugs silenciosos en errores de compilación visibles.
- El constructor de copia por defecto es superficial: con recursos propios necesitas copia profunda. Se invoca al inicializar con otro objeto, pasar por valor y devolver por valor.
- El operador de asignación copia sobre un objeto existente: protégelo de la autoasignación, libera antes de reservar y devuelve *this.
- Regla de tres (destructor + copia + asignación), de cinco (agrega movimientos) y de cero (delega recursos en string/vector/unique_ptr y no escribas nada).
- Composición: el todo posee y destruye las partes. Agregación: el todo usa partes que viven por su cuenta. El tipo del miembro documenta quién es dueño.
- Las clases inmutables (todo const, sin setters, transformaciones que devuelven objetos nuevos) eliminan bugs de estado compartido.
- Los namespaces evitan colisiones de nombres; califica con ::, usa alias para abreviar y evita using namespace en cabeceras.

UNIDAD 7: Los pilares de la programación orientada a objetos

En la unidad 6 aprendiste a crear clases. Ahora aprenderás los tres pilares que hacen poderosa a la programación orientada a objetos: encapsulamiento (proteger los datos), herencia (reutilizar creando jerarquías) y polimorfismo (tratar objetos diferentes de forma uniforme). Estos tres conceptos son el corazón de C++ y de la mayoría del software profesional del mundo.

No son ideas abstractas de libro de texto: son soluciones a problemas reales. El encapsulamiento evita que un cambio en un módulo rompa otros diez. La herencia evita copiar y pegar código. El polimorfismo permite escribir código que funciona con objetos que aún no existen. Domínalos y pensarás como un arquitecto de software.

Al terminar sabrás: diseñar clases encapsuladas con getters y setters, crear jerarquías con herencia, usar funciones virtuales para polimorfismo, y entender cuándo aplicar cada pilar.

Explicación detallada

Clase 19: Encapsulamiento

Encapsulamiento significa esconder los detalles internos de un objeto y exponer solo una interfaz controlada. Los atributos van en `private`; el acceso se hace por métodos públicos. Ya lo hiciste en la unidad 6 con `CuentaBancaria`; ahora lo formalizamos con getters y setters.

```

class Persona {
private:
    std::string nombre;
    int edad;

public:
    Persona(std::string n, int e) : nombre(n) {
        setEdad(e);
    }

    std::string getNombre() const { return nombre; }
    int getEdad() const { return edad; }

    void setEdad(int e) {
        if (e >= 0 && e <= 150) {
            edad = e;
        }
    }

    void setNombre(const std::string& n) {
        if (!n.empty()) nombre = n;
    }
};

```

¿Por qué tanto trabajo para algo que podría ser público? Por cuatro razones. Una: validación. El setter impide edades imposibles; con el atributo público, cualquiera podría poner edad = -50. Dos: invariantes. Garantizas que el objeto siempre esté en un estado válido, como el Termómetro de la unidad 6. Tres: flexibilidad futura. Si mañana la edad se calcula desde la fecha de nacimiento, solo cambias getEdad; el código que lo usa no se entera. Cuatro: depuración. Si algo sale mal con la edad, sabes que solo pudo cambiar por setEdad: un solo lugar que revisar.

Buenas prácticas: no crees getters y setters "automáticos" para todo sin pensar; si un atributo solo se lee, dale solo getter. Los métodos que no modifican llevan const. Y valida en el constructor llamando a los setters, como en el ejemplo: así la validación vive en un solo lugar.

El encapsulamiento también aplica a métodos: los métodos auxiliares internos van en private. La clase expone lo necesario y esconde cómo lo hace. Piensa en un control remoto: tiene pocos botones (interfaz pública) y esconde kilómetros de circuitos (implementación privada).

Clase 20: Herencia

La herencia permite crear una clase nueva (derivada) a partir de una existente (base), heredando sus atributos y métodos y agregando los propios. Es la herramienta de reutilización de la POO.

```
class Animal {
protected:
    std::string nombre;

public:
    Animal(std::string n) : nombre(n) {}
    void comer() {
        std::cout << nombre << " esta comiendo." << std::endl;
    }
    void dormir() {
        std::cout << nombre << " esta durmiendo." << std::endl;
    }
};

class Perro : public Animal {
public:
    Perro(std::string n) : Animal(n) {}

    void ladrar() {
        std::cout << nombre << " dice: ¡Guau!" << std::endl;
    }
};
```

Perro hereda `comer()` y `dormir()` de `Animal` y agrega `ladrar()`. La sintaxis `class Perro : public Animal` se lee "Perro es un Animal". El constructor de Perro llama al de Animal con `: Animal(n)`: siempre se construye primero la base.

Aparece un nuevo nivel de acceso: `protected`. Es como `private`, pero las clases derivadas sí pueden verlo. `nombre` es `protected` para que `Perro::ladrar` pueda usarlo, pero el código externo no. Los tres niveles: `public` (todos ven), `protected` (solo la clase y sus derivadas), `private` (solo la clase).

La herencia modela relaciones "es un": un Perro es un Animal, un Gerente es un Empleado, un `AutoElectrico` es un `Auto`. Si la relación es "tiene un" (un Auto tiene un Motor), no uses herencia: usa composición (un atributo de tipo Motor). Confundir "es un" con "tiene un" es el error de diseño más común.

La herencia también permite redefinir métodos de la base en la derivada. Si Perro define su propio `dormir()`, al llamar `perrito.dormir()` se usa la versión de Perro. Esto se llama ocultamiento, y con funciones virtuales (siguiente clase) se convierte en polimorfismo de verdad.

Cuidado con la herencia excesiva: jerarquías de 5 niveles son difíciles de entender. Prefiere jerarquías planas (2-3 niveles) y composición cuando dudes. Un buen diseño usa herencia para lo que realmente es jerárquico.

Clase 21: Polimorfismo y funciones virtuales

Polimorfismo significa "muchas formas": tratar objetos de distintas clases derivadas a través de una interfaz común, y que cada uno se comporte según su tipo real. Es el pilar más poderoso y el que más cuesta entender al inicio.

El ingrediente clave es la función virtual. Cuando un método de la base se declara virtual, las clases derivadas pueden redefinirlo con `override`, y la llamada se resuelve según el tipo real del objeto, no según el tipo de la variable:

```

class Animal {
public:
    virtual void hacerSonido() const {
        std::cout << "..." << std::endl;
    }
    virtual ~Animal() {}
};

class Perro : public Animal {
public:
    void hacerSonido() const override {
        std::cout << "¡Guau!" << std::endl;
    }
};

class Gato : public Animal {
public:
    void hacerSonido() const override {
        std::cout << "¡Miau!" << std::endl;
    }
};

void concierto(const Animal& a) {
    a.hacerSonido(); // suena según el animal real
}

int main() {
    Perro p;
    Gato g;
    concierto(p); // ¡Guau!
    concierto(g); // ¡Miau!
    return 0;
}

```

La función concierto recibe "un Animal" y funciona con cualquier animal presente o futuro: si mañana creas la clase Pato, concierto la acepta sin cambios. Ese es el poder: código que no necesita modificarse para manejar tipos nuevos. Sin virtual, concierto siempre sonaría "...", el de la base.

Nota el destructor virtual: `virtual ~Animal() {}`. Si destruyes un objeto derivado a través de un puntero a la base, sin destructor virtual solo se llamaría el destructor de la base y habría fugas. Regla de oro: si una clase tiene alguna función virtual, su destructor debe ser virtual.

Las clases abstractas son clases con al menos una función virtual pura (virtual void dibujar() = 0;). No se pueden instanciar: sirven solo como interfaz. Obligan a las derivadas a implementar esos métodos. Son el contrato: "todo Animal debe saber hacerSonido, pero cada uno a su manera".

El polimorfismo suele usarse con punteros o referencias a la base (como concierto). Con objetos por valor hay un problema llamado "rebanado" (slicing): al copiar un Perro en una variable Animal, se pierde la parte de Perro. Por eso se usan referencias, punteros o punteros inteligentes.

Analogía final: el control remoto de la TV tiene un botón de encendido (interfaz común). Cada marca lo implementa a su manera (polimorfismo). Tú no necesitas saber la marca para encenderla: aprietas el botón y cada TV responde según su tipo. Las funciones virtuales son ese botón.

Ejemplos paso a paso

Ejemplo 1: Sistema de empleados con herencia

```
#include <iostream>
#include <string>

class Empleado {
protected:
    std::string nombre;
    double salarioBase;

public:
    Empleado(std::string n, double s) : nombre(n), salarioBase(s) {}

    virtual double calcularPago() const {
        return salarioBase;
    }

    virtual void mostrar() const {
        std::cout << "Empleado: " << nombre << std::endl;
    }

    virtual ~Empleado() {}
};

class Vendedor : public Empleado {
```

```

private:
    double comisiones;

public:
    Vendedor(std::string n, double s, double c)
        : Empleado(n, s), comisiones(c) {}

    double calcularPago() const override {
        return salarioBase + comisiones;
    }

    void mostrar() const override {
        std::cout << "Vendedor: " << nombre
                    << ", Pago: $" << calcularPago() << std::endl;
    }
};

class Gerente : public Empleado {
private:
    double bono;

public:
    Gerente(std::string n, double s, double b)
        : Empleado(n, s), bono(b) {}

    double calcularPago() const override {
        return salarioBase + bono;
    }

    void mostrar() const override {
        std::cout << "Gerente: " << nombre
                    << ", Pago: $" << calcularPago() << std::endl;
    }
};

void procesarNomina(const Empleado& e) {
    e.mostrar();
    std::cout << "    Pago total: $" << e.calcularPago() << std::endl;
}

int main() {
    Vendedor v("Ana", 10000, 3500);
    Gerente g("Luis", 20000, 8000);
    procesarNomina(v);
    procesarNomina(g);
}

```

```
    return 0;
}
```

procesarNomina no sabe si recibe un Vendedor o un Gerente, y no le importa: cada uno calcula su pago a su manera. Si la empresa crea la clase Becario mañana, la nómina la acepta sin tocar una línea.

Ejemplo 2: Figuras geométricas polimórficas

```
#include <iostream>
#include <vector>
#include <memory>

class Figura {
public:
    virtual double area() const = 0; // virtual pura: clase abstracta
    virtual void describir() const = 0;
    virtual ~Figura() {}
};

class Circulo : public Figura {
private:
    double radio;
public:
    Circulo(double r) : radio(r) {}
    double area() const override {
        return 3.14159 * radio * radio;
    }
    void describir() const override {
        std::cout << "Circulo de radio " << radio << std::endl;
    }
};

class RectanguloFig : public Figura {
private:
    double base, altura;
public:
    RectanguloFig(double b, double h) : base(b), altura(h) {}
    double area() const override {
        return base * altura;
    }
    void describir() const override {
        std::cout << "Rectangulo de " << base << "x" << altura << std::endl;
    }
};
```

```

int main() {
    std::vector<std::unique_ptr<Figura>> figuras;
    figuras.push_back(std::make_unique<Circulo>(5.0));
    figuras.push_back(std::make_unique<RectanguloFig>(4.0, 6.0));
    figuras.push_back(std::make_unique<Circulo>(2.5));

    double total = 0;
    for (const auto& f : figuras) {
        f->describir();
        total += f->area();
    }
    std::cout << "Area total: " << total << std::endl;
    return 0;
}

```

Un vector de punteros a Figura guarda círculos y rectángulos mezclados. Cada `f->area()` llama la versión correcta. Figura es abstracta: no puedes crear una "Figura" sola, solo sus hijas. (Usamos `unique_ptr` y `vector`: adelantos de las unidades 8 y 10 que aquí brillan juntos.)

Práctica

Ejercicio 1

Crea la clase Vehiculo con atributos protegidos `marca` y `anio`, constructor, y método virtual `describir()`. Crea las derivadas `Auto` (agrega `numeroPuertas`) y `Moto` (agrega `cilindrada`), cada una redefiniendo `describir()`. Prueba el polimorfismo con una función `mostrarVehiculo(const Vehiculo&)`.

Solución:

```

#include <iostream>
#include <string>

class Vehiculo {
protected:
    std::string marca;
    int anio;

public:
    Vehiculo(std::string m, int a) : marca(m), anio(a) {}
    virtual void describir() const {
        std::cout << marca << " (" << anio << ")" << std::endl;
    }
};

```

```

    }
    virtual ~Vehiculo() {}
};

class Auto : public Vehiculo {
private:
    int numeroPuertas;
public:
    Auto(std::string m, int a, int p) : Vehiculo(m, a), numeroPuertas(p) {}
    void describir() const override {
        std::cout << "Auto " << marca << " (" << anio << "), "
            << numeroPuertas << " puertas." << std::endl;
    }
};

class Moto : public Vehiculo {
private:
    int cilindrada;
public:
    Moto(std::string m, int a, int c) : Vehiculo(m, a), cilindrada(c) {}
    void describir() const override {
        std::cout << "Moto " << marca << " (" << anio << "), "
            << cilindrada << "cc." << std::endl;
    }
};

void mostrarVehiculo(const Vehiculo& v) {
    v.describir();
}

int main() {
    Auto a("Toyota", 2022, 4);
    Moto m("Honda", 2023, 250);
    mostrarVehiculo(a);
    mostrarVehiculo(m);
    return 0;
}

```

Explicación: marca y anio son protected para que las derivadas los usen en describir().
mostrarVehiculo demuestra el polimorfismo: una sola función, dos comportamientos.

Ejercicio 2

Diseña una jerarquía de instrumentos musicales: clase base Instrumento con método virtual puro tocar(). Deriva Guitarra, Piano y Bateria, cada una imprimiendo su sonido. Crea un vector de punteros a Instrumento y hazlos "tocar" a todos en un ciclo.

Solución:

```

#include <iostream>
#include <vector>
#include <memory>

class Instrumento {
public:
    virtual void tocar() const = 0;
    virtual ~Instrumento() {}
};

class Guitarra : public Instrumento {
public:
    void tocar() const override { std::cout << "Rasgueo de guitarra." << std::endl; }
};

class Piano : public Instrumento {
public:
    void tocar() const override { std::cout << "Melodia de piano." << std::endl; }
};

class Bateria : public Instrumento {
public:
    void tocar() const override { std::cout << "Ritmo de bateria." << std::endl; }
};

int main() {
    std::vector<std::unique_ptr<Instrumento>> banda;
    banda.push_back(std::make_unique<Guitarra>());
    banda.push_back(std::make_unique<Piano>());
    banda.push_back(std::make_unique<Bateria>());

    for (const auto& inst : banda) {
        inst->tocar();
    }
    return 0;
}

```

Explicación: Instrumento es abstracta (no se puede instanciar). Cada instrumento implementa tocar() a su manera. El ciclo trata a todos igual: polimorfismo puro. Agregar un Violín mañana no toca el ciclo.

Ejercicio 3

Crea la clase CuentaBancaria con saldo privado y métodos depositar/retirar (validados). Deriva CuentaAhorros que agrega interes mensual con el método aplicarInteres(). Demuestra que el encapsulamiento se hereda: la derivada no puede tocar saldo directamente.

Solución:

```
#include <iostream>

class CuentaBancaria {
private:
    double saldo;

protected:
    void agregar(double monto) { saldo += monto; }

public:
    CuentaBancaria(double inicial) : saldo(inicial >= 0 ? inicial : 0) {}

    void depositar(double monto) {
        if (monto > 0) agregar(monto);
    }

    bool retirar(double monto) {
        if (monto > 0 && monto <= saldo) {
            saldo -= monto;
            return true;
        }
        return false;
    }

    double getSaldo() const { return saldo; }
};

class CuentaAhorros : public CuentaBancaria {
private:
    double tasaInteres;

public:
    CuentaAhorros(double inicial, double tasa)
        : CuentaBancaria(inicial), tasaInteres(tasa) {}

    void aplicarInteres() {
        agregar(getSaldo() * tasaInteres); // usa el helper protegido
    }
}
```

```
};

int main() {
    CuentaAhorros c(1000, 0.05);
    c.depositar(500);
    c.aplicarInteres();
    std::cout << "Saldo: $" << c.getSaldo() << std::endl;
    return 0;
}
```

Explicación: saldo es privado, así que CuentaAhorros no puede hacer saldo += ... directamente (pruébalo: no compila). En su lugar usa el método protegido agregar(). Así el encapsulamiento se mantiene incluso en la herencia: la derivada usa la interfaz, no los datos crudos.

Ejercicio 4

Explica con un ejemplo qué pasa si olvidas virtual en el destructor de una clase base y destruyes un objeto derivado con delete a través de un puntero a la base. Escribe el código correcto.

Solución:

```

#include <iostream>

class Base {
public:
    Base() { std::cout << "Constructor Base" << std::endl; }
    virtual ~Base() { std::cout << "Destructor Base" << std::endl; }
};

class Derivada : public Base {
private:
    int* datos;
public:
    Derivada() : datos(new int[100]) {
        std::cout << "Constructor Derivada" << std::endl;
    }
    ~Derivada() override {
        delete[] datos;
        std::cout << "Destructor Derivada" << std::endl;
    }
};

int main() {
    Base* p = new Derivada();
    delete p; // llama AMBOS destructores gracias a virtual
    return 0;
}

```

Explicación: con destructor virtual, delete p llama primero ~Derivada (libera el arreglo) y luego ~Base. Sin virtual, solo se llamaría ~Base y los 100 enteros quedarían fugados. Regla de oro: clase con funciones virtuales = destructor virtual. Siempre.

Ejercicio 5

Diseña una jerarquía para una tienda: Producto base (nombre, precio, método virtual descripcion() y virtual precioFinal() const), derivadas LibroTienda (descuento 10%) y Electronico (impuesto 16% adicional). Calcula el total de un carrito polimórfico.

Solución:

```

#include <iostream>
#include <string>
#include <vector>
#include <memory>

```

```

class Producto {
protected:
    std::string nombre;
    double precio;
public:
    Producto(std::string n, double p) : nombre(n), precio(p) {}
    virtual std::string descripcion() const { return nombre; }
    virtual double precioFinal() const { return precio; }
    virtual ~Producto() {}
};

class LibroTienda : public Producto {
public:
    LibroTienda(std::string n, double p) : Producto(n, p) {}
    double precioFinal() const override { return precio * 0.90; }
    std::string descripcion() const override { return nombre + " (libro, -10%)"; }
};

class Electronico : public Producto {
public:
    Electronico(std::string n, double p) : Producto(n, p) {}
    double precioFinal() const override { return precio * 1.16; }
    std::string descripcion() const override { return nombre + " (electronico, +16%
IVA)"; }
};

int main() {
    std::vector<std::unique_ptr<Producto>> carrito;
    carrito.push_back(std::make_unique<LibroTienda>("Curso C++", 500));
    carrito.push_back(std::make_unique<Electronico>("Teclado", 1200));
    carrito.push_back(std::make_unique<Producto>("Taza", 150));

    double total = 0;
    for (const auto& p : carrito) {
        std::cout << p->descripcion() << ": $" << p->precioFinal() << std::endl;
        total += p->precioFinal();
    }
    std::cout << "TOTAL: $" << total << std::endl;
    return 0;
}

```

Explicación: cada producto calcula su precio final con sus propias reglas (descuento, impuesto, nada). El carrito suma sin conocer los detalles. Este es el patrón que usan los sistemas reales de comercio electrónico.

Resumen de la unidad

- Encapsulamiento: datos en private, acceso por métodos públicos con validación. Getters/setters, métodos const, invariantes protegidos.
- Herencia: class Derivada : public Base. Reutiliza código modelando "es un". protected es visible para derivadas. Prefiere composición para "tiene un".
- Polimorfismo: funciones virtual + override resuelven la llamada según el tipo real. Permite código extensible sin modificaciones.
- Destructor virtual siempre que haya funciones virtuales. Clases abstractas (= 0) como contratos.
- Evita el rebanado usando referencias o punteros a la base.
- En la unidad 8 subirás de nivel: sobrecarga de operadores, plantillas genéricas y la poderosa STL.

Profundización: polimorfismo por dentro y principios de diseño

Ya usas herencia y polimorfismo. Ahora mira debajo del capó: cómo implementa C++ las funciones virtuales (la famosa tabla virtual), cuánto cuestan, y los principios de diseño que los profesionales usan para decidir cuándo heredar y cuándo no. Este conocimiento te da criterio de arquitecto.

Cómo funciona virtual por dentro: la vtable

Cuando una clase tiene funciones virtuales, el compilador le agrega un puntero oculto (vptr) que apunta a una tabla de funciones virtuales (vtable): una tabla con las direcciones de las implementaciones correctas. Cada objeto sabe así a qué versión llamar en tiempo de ejecución.

```
Animal* a = new Perro();  
a->hacerSonido(); // sigue el vptr -> vtable de Perro -> Perro::hacerSonido
```

El costo: un puntero extra por objeto y una indirección por llamada. En la práctica es insignificante salvo en código ultra-crítico (motores de videojuegos evitan virtual en el bucle más interno). Para el 99% del software, la flexibilidad vale mucho más que ese costo.

Por eso el polimorfismo necesita punteros o referencias: con objetos por valor ocurre el "rebanado" (slicing), donde la parte derivada se corta al copiar en una variable de la base. Demuéstralo:

```
Perro p("Firulais");  
Animal a = p; // ¡rebanado! a es solo un Animal  
a.hacerSonido(); // suena "...", no "¡Guau!"
```

La copia solo copió la parte Animal. Con `Animal& a = p;` no hay copia y el polimorfismo funciona.

Principios SOLID (versión práctica)

Cinco principios de diseño orientado a objetos, resumidos para usarlos ya:

1. Responsabilidad única: una clase, una razón para cambiar. Si tu clase hace validación, cálculos y archivos, pártela en tres.
2. Abierto/cerrado: abierto a extensión, cerrado a modificación. Agrega comportamientos con clases nuevas, no editando las viejas (como agregar Revista sin tocar Material).
3. Sustitución de Liskov: una derivada debe poder usarse donde se espera la base, sin sorpresas. Si tu PatoDeHule no puede volar pero hereda de Pato que sí, el diseño está mal.
4. Segregación de interfaces: mejor varias interfaces pequeñas que una gigante. Nadie debe depender de métodos que no usa.
5. Inversión de dependencias: depende de abstracciones (clases base), no de concreciones. `procesarNomina` recibe `const Empleado&`, no `Vendedor`.

No los memorices como dogma: úsalos como preguntas cuando diseñes. "¿Esta clase tiene más de una responsabilidad?" es una pregunta poderosa.

Herencia múltiple: existe, úsala poco

C++ permite heredar de varias bases: `class C : public A, public B`. Tiene usos legítimos (mezclar interfaces), pero crea el "problema del diamante": si A y B heredan de Base, C tiene dos copias de Base. Existe `virtual inheritance` para eso, pero la complejidad rara vez vale la pena. En la práctica profesional, la herencia múltiple se usa casi solo para interfaces (clases con solo métodos virtuales puros).

Errores de diseño comunes

Error 1: jerarquías por capricho. No todo "es un" necesita herencia. Un Círculo no debería heredar de Punto aunque matemáticamente se relacione: el círculo tiene un centro (composición), no es un punto.

Error 2: romper Liskov. Si la derivada necesita deshabilitar métodos de la base (lanzar "no soportado"), la jerarquía miente. Rediseña.

Error 3: usar herencia para reutilizar código sin relación conceptual. Si solo quieres "copiar" funcionalidad, la composición es más honesta.

Error 4: exponer protected en exceso. protected es una promesa a futuros derivados: cambiarlo rompe a todos. Mantén protected mínimo.

Downcasting: de base a derivada

A veces tienes un `Animal*` y necesitas tratarlo como `Perro`. `dynamic_cast` lo hace con verificación:

```
Animal* a = new Perro("Firulais");
Perro* p = dynamic_cast<Perro*>(a);
if (p) {
    p->ladrar(); // seguro: sí era un Perro
}
```

Si el objeto no es del tipo pedido, `dynamic_cast` devuelve `nullptr` (o lanza con referencias). Si te encuentras haciendo `dynamic_cast` a cada rato, suele ser señal de mal diseño: el polimorfismo debería evitarlo.

Más ejemplos

Ejemplo: plugins con polimorfismo

```

#include <iostream>
#include <string>
#include <vector>
#include <memory>

// Interfaz: todo plugin sabe su nombre y sabe ejecutarse
class Plugin {
public:
    virtual std::string nombre() const = 0;
    virtual void ejecutar() = 0;
    virtual ~Plugin() {}
};

class PluginSaludo : public Plugin {
public:
    std::string nombre() const override { return "Saludo"; }
    void ejecutar() override { std::cout << "¡Hola desde el plugin!" << std::endl; }
};

class PluginSuma : public Plugin {
public:
    std::string nombre() const override { return "Suma"; }
    void ejecutar() override { std::cout << "2 + 2 = " << 2 + 2 << std::endl; }
};

int main() {
    std::vector<std::unique_ptr<Plugin>> plugins;
    plugins.push_back(std::make_unique<PluginSaludo>());
    plugins.push_back(std::make_unique<PluginSuma>());

    // El programa principal no conoce los plugins concretos:
    // solo la interfaz. Agregar uno nuevo no toca este código.
    for (const auto& p : plugins) {
        std::cout << "[" << p->nombre() << "] ";
        p->ejecutar();
    }
    return 0;
}

```

Esta es la arquitectura de plugins de aplicaciones reales (editores, navegadores): el núcleo define la interfaz y los plugins la implementan. El núcleo nunca cambia al agregar funcionalidad.

Ejemplo: el costo de virtual, medido

```
#include <iostream>
#include <chrono>

class Base {
public:
    virtual int calcular(int x) { return x * 2; }
    virtual ~Base() {}
};

class Derivada : public Base {
public:
    int calcular(int x) override { return x * 3; }
};

int main() {
    Derivada d;
    Base* b = &d;
    const long N = 100000000;
    auto inicio = std::chrono::high_resolution_clock::now();
    long total = 0;
    for (long i = 0; i < N; i++) total += b->calcular(5);
    auto fin = std::chrono::high_resolution_clock::now();
    auto ms = std::chrono::duration_cast<std::chrono::milliseconds>(fin -
    inicio).count();
    std::cout << "100M llamadas virtuales: " << ms << " ms (total=" << total << ")"
    << std::endl;
    return 0;
}
```

Mide en tu máquina: verás que 100 millones de llamadas virtuales toman una fracción de segundo. El "costo" de virtual es real pero diminuto; no lo uses como excusa para evitar el polimorfismo.

Práctica adicional

Ejercicio A

Demuestra el rebanado (slicing): crea Perro y Gato derivados de Animal con hacerSonido virtual, asigna un Perro a una variable Animal por valor y observa qué suena. Luego hazlo con referencia y compara.

Solución:

```
#include <iostream>

class Animal {
public:
    virtual void hacerSonido() const { std::cout << "... " << std::endl; }
    virtual ~Animal() {}
};

class Perro : public Animal {
public:
    void hacerSonido() const override { std::cout << "¡Guau!" << std::endl; }
};

int main() {
    Perro p;
    Animal a = p;      // rebanado: se pierde la parte Perro
    a.hacerSonido();    // "... "
    Animal& r = p;      // referencia: sin copia
    r.hacerSonido();    // "¡Guau!"
    return 0;
}
```

Explicación: la asignación por valor copia solo la subparte Animal. La referencia no copia nada: apunta al Perro completo. Por eso el polimorfismo vive en punteros y referencias.

Ejercicio B

Diseña una jerarquía de notificaciones: clase abstracta Notificacion con enviar(string mensaje) virtual puro; derivadas Email, SMS y Push con sus detalles. Una función notificarATodos recibe un vector de punteros y un mensaje.

Solución:

```

#include <iostream>
#include <string>
#include <vector>
#include <memory>

class Notificacion {
public:
    virtual void enviar(const std::string& mensaje) = 0;
    virtual ~Notificacion() {}
};

class Email : public Notificacion {
    std::string direccion;
public:
    Email(const std::string& d) : direccion(d) {}
    void enviar(const std::string& m) override {
        std::cout << "Email a " << direccion << ": " << m << std::endl;
    }
};

class SMS : public Notificacion {
    std::string numero;
public:
    SMS(const std::string& n) : numero(n) {}
    void enviar(const std::string& m) override {
        std::cout << "SMS al " << numero << ": " << m << std::endl;
    }
};

void notificarATodos(const std::vector<std::unique_ptr<Notificacion>>& canales,
                    const std::string& mensaje) {
    for (const auto& c : canales) c->enviar(mensaje);
}

int main() {
    std::vector<std::unique_ptr<Notificacion>> canales;
    canales.push_back(std::make_unique<Email>("ana@correo.com"));
    canales.push_back(std::make_unique<SMS>("555-1234"));
    notificarATodos(canales, "Tu pedido llegó");
    return 0;
}

```

Explicación: agregar WhatsApp mañana es crear una clase nueva; notificarATodos no cambia (abierto/cerrado). Cada canal implementa el envío a su manera (polimorfismo).

Ejercicio C

Encuentra el error de diseño: class Cuadrado : public Rectangulo donde Rectangulo tiene setBase y setAltura independientes. Explica por qué rompe Liskov y propón una solución.

Solución: un Cuadrado no puede cumplir el contrato de Rectangulo: si alguien hace setBase(5); setAltura(3); esperando un rectángulo de 5x3, el "cuadrado" quedaría en estado inconsistente (¿5x5 o 3x3?). Cualquier código que use Rectangulo se rompería con un Cuadrado. Esto viola Liskov: la derivada no se comporta como la base promete. Soluciones: no heredar (usar composición o clases independientes), o heredar ambas de una Figura abstracta con area() virtual, sin prometer setters independientes.

Ejercicio D

Usa dynamic_cast para contar cuántos Perros hay en un vector<unique_ptr<Animal>> mezclado con Gatos.

Solución:

```
#include <iostream>
#include <vector>
#include <memory>

class Animal {
public:
    virtual ~Animal() {}
};

class Perro : public Animal {};
class Gato : public Animal {};

int main() {
    std::vector<std::unique_ptr<Animal>> animales;
    animales.push_back(std::make_unique<Perro>());
    animales.push_back(std::make_unique<Gato>());
    animales.push_back(std::make_unique<Perro>());

    int perros = 0;
    for (const auto& a : animales) {
        if (dynamic_cast<Perro*>(a.get())) perros++;
    }
    std::cout << "Perros: " << perros << std::endl;
    return 0;
}
```

Explicación: `dynamic_cast` verifica en tiempo de ejecución; devuelve `nullptr` si no es Perro. `a.get()` extrae el puntero crudo del `unique_ptr` sin transferir propiedad. Recuerda: si necesitas esto muy seguido, reconsidera el diseño.

Expansión Unidad 7: herencia múltiple, contratos, estrategia y jerarquías reales

En las partes anteriores de esta unidad aprendiste los tres pilares: encapsulamiento, herencia simple y polimorfismo con funciones virtuales. También viste por dentro la `vtable`, los principios SOLID, el problema del diamante y el `dynamic_cast`. Ahora toca subir otro escalón: usar la herencia múltiple de forma legítima y segura, diseñar contratos con clases abstractas, aplicar el patrón Estrategia, decidir entre composición y herencia con criterio profesional, entender una jerarquía real que usas todos los días (los streams de `iostream`), evitar el object slicing en colecciones y aprovechar un truco poco conocido: los métodos virtuales puros con implementación por defecto.

Esta expansión es práctica de principio a fin. Cada tema trae código completo que puedes compilar, probar y romper a propósito para aprender. Al terminar tendrás criterio de diseñador: no solo sabrás qué hace cada mecanismo, sino cuándo conviene usarlo y cuándo es mejor no tocarlo.

Herencia múltiple bien usada: las interfaces

Ya sabes que C++ permite heredar de varias clases base y que eso puede crear el problema del diamante cuando dos bases comparten un ancestro común. Por eso la recomendación general fue "úsala poco". Pero hay un caso donde la herencia múltiple no solo es aceptable, sino que es la herramienta correcta: cuando las clases base son interfaces puras, es decir, clases que solo contienen métodos virtuales puros (y quizá un destructor virtual) y ningún dato miembro.

Una interfaz describe un "puede hacer" en lugar de un "es un". Un Pato puede volar y puede nadar, pero no "es un" Volador en el sentido de heredar datos de vuelo: simplemente cumple el contrato de saber volar. Cuando una clase hereda de varias interfaces, no hay diamante posible porque no hay datos que duplicar: solo hay promesas de comportamiento que la clase derivada debe implementar.

La regla práctica es esta: herencia múltiple de interfaces, sí; herencia múltiple de clases con datos, casi nunca. Los lenguajes como Java y C# ni siquiera permiten herencia múltiple de

clases, pero sí permiten implementar varias interfaces. En C++ tienes ambas opciones, así que la disciplina la pones tú.

Mira el siguiente ejemplo completo. Define dos interfaces, Imprimible y Guardable, y una clase Reporte que cumple ambas. Fíjate en que las interfaces no tienen ni un solo atributo: son contratos puros.

```
#include <iostream>
#include <string>
#include <vector>
#include <memory>

// Interfaz 1: todo lo imprimible sabe mostrarse en consola
class Imprimible {
public:
    virtual void imprimir() const = 0;
    virtual ~Imprimible() {}
};

// Interfaz 2: todo lo guardable sabe convertirse a texto plano
class Guardable {
public:
    virtual std::string serializar() const = 0;
    virtual ~Guardable() {}
};

// Reporte cumple AMBOS contratos: herencia múltiple de interfaces
class Reporte : public Imprimible, public Guardable {
private:
    std::string titulo;
    int paginas;

public:
    Reporte(const std::string& t, int p) : titulo(t), paginas(p) {}

    void imprimir() const override {
        std::cout << "=== " << titulo << " ===" << std::endl;
        std::cout << "Paginas: " << paginas << std::endl;
    }

    std::string serializar() const override {
        return titulo + "|" + std::to_string(paginas);
    }
};
```



```
// Funciones que dependen de interfaces, no de clases concretas
void mostrarEnPantalla(const Imprimible& obj) {
    obj.imprimir();
}

void guardarEnArchivo(const Guardable& obj) {
    std::cout << "Guardando: " << obj.serializar() << std::endl;
}

int main() {
    Reporte r("Ventas 2026", 12);
    mostrarEnPantalla(r); // lo ve como Imprimible
    guardarEnArchivo(r);  // lo ve como Guardable

    // Una colección puede mezclar objetos por cualquiera de sus interfaces
    std::vector<std::unique_ptr<Imprimible>> cola;
    // (aquí podrías agregar Factura, Grafica, etc., si cumplen Imprimible)
    return 0;
}
```

Observa lo que ganaste: `mostrarEnPantalla` acepta cualquier cosa imprimible (reportes, facturas, gráficas) y `guardarEnArchivo` acepta cualquier cosa guardable. Un mismo objeto puede pertenecer a varias "familias" de comportamiento sin que esas familias compartan datos ni se estorben. Esa es la herencia múltiple legítima: composición de capacidades, no de estado.

Dos detalles finos que debes notar. Primero: cada interfaz tiene su propio destructor virtual. Si una interfaz no tuviera destructor virtual y alguien destruyera un `Reporte` a través de un puntero `Imprimible*`, el comportamiento sería indefinido. Segundo: si dos interfaces declararan un método con el mismo nombre y firma, la clase derivada tendría que resolver la ambigüedad calificando (por ejemplo, `obj.Imprimible::imprimir()`). Por eso conviene que los nombres de métodos de interfaces distintas no choquen, o resolverlo explícitamente.

Cuándo NO usar herencia múltiple

No la uses para "reutilizar" dos clases con datos. Si necesitas la funcionalidad de dos clases concretas, casi siempre es mejor composición: tu clase tiene un miembro de cada tipo y expone lo necesario. La herencia múltiple de clases con estado crea ambigüedades (¿de cuál base viene este atributo?), constructores complicados y el diamante. La comunidad de C++ aprendió esto a golpes durante los años noventa; no tienes que repetir la lección.

Clases abstractas como contratos: un sistema de pagos

Ya viste clases abstractas con *Figura* y su método *area()* virtual puro. Ahora veamos el uso profesional más importante de las clases abstractas: definir contratos entre partes de un sistema que evolucionan por separado. El ejemplo clásico es un sistema de pagos: la tienda necesita cobrar, y existen muchas formas de pago (tarjeta, efectivo, transferencia, billetera digital). La tienda no quiere saber los detalles de cada una; solo necesita un contrato: "todo método de pago sabe cobrar un monto y sabe describirse".

Ese contrato es una clase abstracta. Fíjate en el diseño: la clase *Tienda* depende solo de la abstracción *MetodoPago*. Agregar un método nuevo (por ejemplo, pago con criptomonedas) significa crear una clase nueva, sin tocar la tienda. Esto es el principio abierto/cerrado y la inversión de dependencias de SOLID trabajando juntos en código real.

```
#include <iostream>
#include <string>
#include <vector>
#include <memory>

// CONTRATO: todo método de pago debe poder cobrar y describirse
class MetodoPago {
public:
    virtual bool cobrar(double monto) = 0;
    virtual std::string descripcion() const = 0;
    virtual ~MetodoPago() {}
};

class Tarjeta : public MetodoPago {
private:
    std::string numero;
    double limite;

public:
    Tarjeta(const std::string& n, double l) : numero(n), limite(l) {}

    bool cobrar(double monto) override {
        if (monto <= 0 || monto > limite) return false;
        limite -= monto;
        std::string ultimos = numero.substr(numero.length() - 4);
        std::cout << "Cobrando $" << monto << " a tarjeta *****" << ultimos <<
std::endl;
        return true;
    }
}
```

```

        std::string descripcion() const override {
            return "Tarjeta de credito";
        }
};

class Efectivo : public MetodoPago {
private:
    double enCaja;

public:
    Efectivo(double inicial) : enCaja(inicial) {}

    bool cobrar(double monto) override {
        if (monto <= 0 || monto > enCaja) return false;
        enCaja -= monto;
        std::cout << "Cobrando $" << monto << " en efectivo" << std::endl;
        return true;
    }

    std::string descripcion() const override {
        return "Efectivo";
    }
};

class Transferencia : public MetodoPago {
private:
    std::string clabe;

public:
    Transferencia(const std::string& c) : clabe(c) {}

    bool cobrar(double monto) override {
        if (monto <= 0) return false;
        std::cout << "Transfiriendo $" << monto << " a CLABE " << clabe << std::endl;
        return true;
    }

    std::string descripcion() const override {
        return "Transferencia bancaria";
    }
};

// La tienda solo conoce el CONTRATO, no las clases concretas
class Tienda {
private:

```

```

    std::string nombre;

public:
    Tienda(const std::string& n) : nombre(n) {}

    bool vender(double monto, MetodoPago& pago) {
        std::cout << nombre << ": venta por $" << monto
            << " con " << pago.descripcion() << std::endl;
        return pago.cobrar(monto);
    }
};

int main() {
    Tienda tienda("La Esquina");
    Tarjeta t("1234567812345678", 5000);
    Efectivo e(2000);
    Transferencia tr("012345678901234567");

    tienda.vender(350, t);
    tienda.vender(120, e);
    tienda.vender(900, tr);

    // El carrito de métodos acepta cualquier implementación futura
    std::vector<std::unique_ptr<MetodoPago>> metodos;
    metodos.push_back(std::make_unique<Tarjeta>("9999888877776666", 10000));
    metodos.push_back(std::make_unique<Efectivo>(500));
    for (const auto& m : metodos) {
        std::cout << "- Disponible: " << m->descripcion() << std::endl;
    }
    return 0;
}

```

Hay varias decisiones de diseño que vale la pena desmenuzar. Una: cobrar devuelve bool para reportar éxito o fracaso, en lugar de lanzar excepciones o imprimir errores silenciosos. El contrato no solo dice qué métodos existen, también dice cómo se comunican los resultados. Dos: cada implementación valida a su manera (la tarjeta revisa el límite, el efectivo revisa la caja, la transferencia revisa que el monto sea positivo). La validación vive donde está el conocimiento, no en la tienda. Tres: descripcion() es const porque describir no modifica el objeto; cobrar no es const porque sí modifica el estado (baja el límite, saca dinero de la caja).

Este patrón aparece en todas partes: sistemas de notificaciones (el contrato "enviar mensaje" con Email, SMS, Push), videojuegos (el contrato "enemigo" con distintos comportamientos), drivers de dispositivos. Cuando veas un sistema que debe aceptar "cosas nuevas" sin modificarse, piensa en una clase abstracta como contrato.

Un matiz importante: un contrato bien diseñado es pequeño y estable. Si cada mes le agregas métodos virtuales puros nuevos a MetodoPago, todas las clases existentes se rompen (dejan de compilar hasta implementar el método nuevo). Por eso la segregación de interfaces de SOLID pide interfaces pequeñas: es más fácil mantener estable un contrato de dos métodos que uno de veinte. Si un método de pago necesita algo extra (por ejemplo, reembolsar), quizá convenga una segunda interfaz Reembolsable en lugar de engordar el contrato original. Y adivina qué: esa segunda interfaz se combinaría con herencia múltiple de interfaces, como viste en la sección anterior. Todo se conecta.

El patrón Estrategia: cambiar comportamiento sin heredar

Hay un problema que la herencia simple resuelve mal: cuando un objeto necesita cambiar su comportamiento en tiempo de ejecución. Imagina una tienda que aplica descuentos: a veces hay descuento de temporada (10%), a veces cupón fijo (\$100), a veces "compra 2 y el tercero gratis", y las promociones cambian cada semana. Si modelas cada combinación con herencia (TiendaConDescuentoTemporada, TiendaConCupon...), terminas con una explosión de clases y no puedes cambiar la promoción sin reconstruir el objeto.

El patrón Estrategia resuelve esto con polimorfismo y composición combinados: defines una familia de algoritmos (las estrategias) detrás de una interfaz común, y el objeto "cliente" guarda un puntero a la estrategia activa. Cambiar de comportamiento es cambiar el puntero, incluso mientras el programa corre.

```
#include <iostream>
#include <string>
#include <memory>

// Interfaz de la estrategia: todo descuento sabe calcular el precio final
class EstrategiaDescuento {
public:
    virtual double aplicar(double precio) const = 0;
    virtual std::string nombre() const = 0;
    virtual ~EstrategiaDescuento() {}
};

class SinDescuento : public EstrategiaDescuento {
public:
    double aplicar(double precio) const override { return precio; }
    std::string nombre() const override { return "Sin descuento"; }
};

class DescuentoPorcentaje : public EstrategiaDescuento {
```

```

private:
    double porcentaje;
public:
    DescuentoPorcentaje(double p) : porcentaje(p) {}
    double aplicar(double precio) const override {
        return precio * (1.0 - porcentaje / 100.0);
    }
    std::string nombre() const override { return "Descuento de temporada"; }
};

class DescuentoFijo : public EstrategiaDescuento {
private:
    double monto;
public:
    DescuentoFijo(double m) : monto(m) {}
    double aplicar(double precio) const override {
        double resultado = precio - monto;
        return resultado < 0 ? 0 : resultado;
    }
    std::string nombre() const override { return "Cupon de descuento"; }
};

// El cliente USA la estrategia (composición), no la hereda
class CajaRegistradora {
private:
    std::unique_ptr<EstrategiaDescuento> estrategia;

public:
    CajaRegistradora()
        : estrategia(std::make_unique<SinDescuento>()) {}

    // Cambiar el comportamiento en tiempo de ejecución
    void setEstrategia(std::unique_ptr<EstrategiaDescuento> e) {
        estrategia = std::move(e);
    }

    double cobrar(double precio) const {
        double final = estrategia->aplicar(precio);
        std::cout << "[" << estrategia->nombre() << "]" "
            << "$" << precio << " -> $" << final << std::endl;
        return final;
    }
};

int main() {

```

```

CajaRegistradora caja;
caja.cobrar(1000); // sin descuento

// Llega la temporada de ofertas: se cambia la estrategia al vuelo
caja.setEstrategia(std::make_unique<DescuentoPorcentaje>(15));
caja.cobrar(1000);

// Y luego un cupón especial
caja.setEstrategia(std::make_unique<DescuentoFijo>(200));
caja.cobrar(1000);
return 0;
}

```

Fíjate en la arquitectura: CajaRegistradora no hereda de nadie y no tiene ni una línea de lógica de descuentos. Su único trabajo es delegar en la estrategia actual. Cada estrategia es una clase pequeña, fácil de probar por separado: puedes verificar que DescuentoFijo nunca devuelva un precio negativo sin montar una tienda completa. Y agregar una estrategia nueva (por ejemplo, "2x1") es crear una clase; la caja no se toca.

Este patrón es pariente cercano del sistema de pagos de la sección anterior, pero con una diferencia clave de intención: en el sistema de pagos, el polimorfismo modela "tipos de cosas" (cada pago es una cosa distinta); en la estrategia, modela "formas de hacer algo" intercambiables dentro del mismo objeto. Cuando dudes entre heredar un comportamiento o parametrizarlo con una estrategia, pregúntate: "¿este comportamiento necesita cambiar mientras el programa corre, o combinarse de formas que no prevé?". Si la respuesta es sí, la estrategia casi siempre gana.

La estrategia también aparece disfrazada en la vida real del programador: los comparadores personalizados de `std::sort`, las políticas de un asignador de memoria, los "manejadores" de un servidor web. Es uno de esos patrones que, una vez que lo ves, empiezas a ver en todas partes.

Composición vs herencia: la guía de decisión definitiva

En la parte principal de la unidad aprendiste la regla básica: "es un" sugiere herencia, "tiene un" sugiere composición. Es una buena primera aproximación, pero los diseños reales necesitan más criterio. Aquí tienes la guía completa que usan los profesionales, con ejemplos de cada caso.

Usa herencia cuando se cumplan estas condiciones

1. Existe una relación "es un" genuina y estable. Un Gerente es un Empleado hoy y lo seguirá siendo mañana. Si la relación puede dejar de ser cierta, no heredes.
2. Quieres polimorfismo: tratar objetos distintos de forma uniforme a través de la base. Si nunca vas a usar punteros o referencias a la base, la herencia te da poco.
3. La base es una abstracción diseñada para ser heredada (tiene funciones virtuales y destructor virtual). Heredar de una clase concreta que no fue diseñada para eso es frágil.

Usa composición cuando se cumpla cualquiera de estas

1. La relación es "tiene un" o "usa un". Un Auto tiene un Motor; un Pedido tiene una lista de Productos; una Ventana tiene un Botón. Ninguno de estos "es" el otro.
2. Quieres reutilizar funcionalidad sin acoplarte a una jerarquía. Una clase Registro puede tener un miembro ArchivoLog y usarlo, sin ser un ArchivoLog.
3. El comportamiento debe cambiar en tiempo de ejecución (ahí entra la estrategia de la sección anterior, que es composición + polimorfismo).
4. Solo quieres "copiar" código de otra clase sin relación conceptual. La herencia para reutilización perezosa crea jerarquías mentirosas que luego nadie entiende.

Mira el contraste con dos diseños del mismo problema: una Pila (estructura LIFO). Versión con herencia (mala idea) y versión con composición (correcta):


```

#include <iostream>
#include <vector>
#include <string>

// MAL: Pila "es un" vector... ¿de verdad? Una pila no debería
// permitir insertar en medio, pero al heredar de vector sí puede.
class PilaMala : public std::vector<int> {
public:
    void push(int v) { push_back(v); }
    void pop() { pop_back(); }
    int cima() const { return back(); }
};

// BIEN: Pila "tiene un" vector y expone solo lo necesario
class PilaBuena {
private:
    std::vector<int> datos; // composición: el vector es un detalle interno
public:
    void push(int v) { datos.push_back(v); }
    void pop() { if (!datos.empty()) datos.pop_back(); }
    int cima() const { return datos.back(); }
    bool vacia() const { return datos.empty(); }
};

int main() {
    PilaMala pm;
    pm.push(1); pm.push(2);
    pm.insert(pm.begin(), 999); // ¡se puede! rompe la idea de pila
    std::cout << "PilaMala permite insertar en medio: " << pm[0] << std::endl;

    PilaBuena pb;
    pb.push(1); pb.push(2);
    // pb.insert(...); // NO compila: la interfaz es solo de pila
    std::cout << "Cima de PilaBuena: " << pb.cima() << std::endl;
    return 0;
}

```

PilaMala hereda públicamente de vector y con ello hereda también insert, erase, operator[] y todo lo demás: la "pila" deja de comportarse como pila. Además viola Liskov: cualquier código que espera un vector (por ejemplo, una función que inserta en la posición 0) aceptaría una PilaMala y la rompería conceptualmente. PilaBuena, en cambio, esconde el vector en private y solo expone push, pop, cima y vacia: la interfaz dice exactamente lo que la pila promete, ni más ni menos.

Hay un caso intermedio que debes conocer: la herencia privada o protegida (`class Pila : private std::vector<int>`). Eso significa "está implementada en términos de", y no crea relación "es un" hacia afuera. Es una forma de reutilización, pero en la práctica la composición es más clara y flexible, así que la verás poco en código moderno.

La prueba del enunciado

Cuando dudes, formula el diseño como enunciado y escucha cómo suena: "Un Gerente es un Empleado" (suena bien: herencia). "Una Pila es un Vector" (suena raro: composición). "Un Pato es un Volador y es un Nadador" (suena a capacidades: interfaces). "Una CajaRegistradora tiene una EstrategiaDescuento" (suena bien: composición). Si el enunciado te obliga a hacer malabares ("bueno, es un... más o menos"), es señal de que el diseño está forzado. Los buenos diseños se enuncian en una frase simple.

Otro criterio útil es la dirección del cambio: la herencia acopla la derivada a la base para siempre (si cambias la base, revisas todas las derivadas). La composición acopla solo a la interfaz que usas del componente. En sistemas grandes que viven años, ese acoplamiento menor se paga solo. Por eso el consejo clásico de diseño dice "prefiere composición sobre herencia": no porque la herencia sea mala, sino porque la composición es el valor predeterminado más seguro, y la herencia se reserva para los casos donde el polimorfismo la justifica.

Jerarquías reales: los streams de iostream explicados

Hasta ahora las jerarquías de los ejemplos fueron inventadas para enseñar. Pero hay una jerarquía de herencia que usas desde tu primer programa en C++ y que es un ejemplo magistral de diseño orientado a objetos: la de los streams de la biblioteca estándar, la que te da `std::cout` y `std::cin`.

La idea central es separar dos dimensiones independientes: la dirección del flujo (entrada o salida) y el destino (consola, archivo, cadena de texto). Con herencia, cada combinación reutiliza lo ya construido en lugar de duplicarlo. La jerarquía real (simplificada) es así:

- `ios_base` / `ios`: la raíz. Guarda el estado del stream (¿falló la última lectura? ¿llegamos al final?) y el formato (decimales, ancho de campo, etc.). Todo stream "es un" `ios`.
- `istream` : public `ios`: agrega la capacidad de extraer datos con `operator>>`. `std::cin` es un `istream`.
- `ostream` : public `ios`: agrega la capacidad de insertar datos con `operator<<`. `std::cout` es un `ostream`.

- `iostream` : `public istream`, `public ostream`: ¡herencia múltiple legítima! Un stream de entrada y salida necesita ambas capacidades. Aquí sí hay un diamante potencial (`istream` y `ostream` comparten la base `ios`), y la biblioteca lo resuelve con herencia virtual: `class istream : virtual public ios`. La herencia virtual garantiza una sola copia compartida de la base común.
- `ifstream` : `public istream`: entrada desde archivo. Reutiliza todo `operator>>` y solo agrega `open/close` y el manejo del archivo.
- `ofstream` : `public ostream`: salida a archivo.
- `fstream` : `public iostream`: archivos de entrada y salida.
- `istringstream` : `public istream`: entrada desde una cadena en memoria. `ostringstream` : `public ostream`: salida hacia una cadena.

¿Por qué este diseño es brillante? Primero, porque cualquier función que reciba un `std::istream&` puede leer tanto del teclado (`cin`) como de un archivo (`ifstream`) o de una cadena (`istringstream`), sin cambiar una línea. Eso es polimorfismo aplicado a la entrada de datos, y es la razón por la que puedes probar tu lógica de lectura con cadenas en memoria en lugar de teclear datos a mano. Segundo, porque el estado y el formato viven una sola vez en la base `ios`: no hay duplicación. Tercero, porque la herencia múltiple aparece exactamente donde debe: `iostream` combina dos interfaces/capacidades (entrada y salida) sin duplicar estado, gracias a la herencia virtual.

Mira cómo este diseño te beneficia directamente con un ejemplo: una función que procesa datos da igual de dónde vengan.

```

#include <iostream>
#include <sstream>
#include <fstream>
#include <string>

// Esta función solo exige "algo de lo que se puede leer":
// acepta cin, archivos y cadenas gracias a la jerarquía de streams
double promediar(std::istream& entrada) {
    double suma = 0;
    int n = 0;
    double x;
    while (entrada >> x) {
        suma += x;
        n++;
    }
    return n == 0 ? 0 : suma / n;
}

int main() {
    // 1) Desde una cadena en memoria (ideal para pruebas)
    std::istringstream datos("10 20 30 40");
    std::cout << "Promedio cadena: " << promediar(datos) << std::endl;

    // 2) Desde el teclado funcionaría igual: promediar(std::cin)
    // 3) Desde un archivo funcionaría igual con std::ifstream

    // Y la salida también es polimórfica: ostringstream construye texto
    std::ostringstream reporte;
    reporte << "El promedio fue " << promediar(std::istringstream("5 15")) << ".";
    std::cout << reporte.str() << std::endl;
    return 0;
}

```

promediar recibe `std::istream&` y no sabe ni le importa si los datos vienen del teclado, de un archivo o de una cadena: el polimorfismo de la jerarquía lo resuelve. Esta es la lección de diseño más valiosa de los streams: cuando modelas una capacidad abstracta ("fuente de datos legible", "destino escribible") como clase base, todo tu código se vuelve reutilizable a través de ella. Es el mismo principio del contrato de pagos, aplicado por los diseñadores de la biblioteca estándar hace décadas.

Un detalle que ahora puedes apreciar con ojos nuevos: `operator<<` y `operator>>` no son virtuales en general, porque la lectura y escritura de cada tipo se resuelve con sobrecarga (tema de la unidad 8), no con polimorfismo de streams. El polimorfismo de la jerarquía está en

el comportamiento del flujo (dónde lee, cómo reporta errores), no en cada tipo de dato. Diseño fino: cada mecanismo donde rinde más.

Object slicing en vectores y su solución definitiva

En las partes anteriores viste el rebanado (slicing) básico: copiar un Perro en una variable Animal pierde la parte de Perro. Ahora veamos dónde este problema muerde de verdad en programas reales: en los contenedores. Es un error tan común que merece su sección propia, porque el código compila sin quejarse y el programa "funciona", pero se comporta mal.

El problema: `std::vector<Animal>` guarda objetos Animal por valor, cada uno del mismo tamaño. Si intentas meter un Perro, el vector solo puede guardar la parte Animal: la parte Perro se rebana al entrar. Y como el tipo estático de cada elemento es Animal, las llamadas virtuales... bueno, mira el ejemplo y compruébalo tú mismo.

```

#include <iostream>
#include <string>
#include <vector>
#include <memory>

class Animal {
public:
    virtual void hacerSonido() const { std::cout << "... " << std::endl; }
    virtual ~Animal() {}
};

class Perro : public Animal {
public:
    void hacerSonido() const override { std::cout << "¡Guau!" << std::endl; }
};

class Gato : public Animal {
public:
    void hacerSonido() const override { std::cout << "¡Miau!" << std::endl; }
};

int main() {
    Perro p;
    Gato g;

    // PROBLEMA: vector de objetos por valor -> rebanado silencioso
    std::vector<Animal> corral;
    corral.push_back(p); // solo entra la parte Animal del Perro
    corral.push_back(g); // solo entra la parte Animal del Gato
    std::cout << "Con vector<Animal> (rebanado):" << std::endl;
    for (const auto& a : corral) a.hacerSonido(); // "... " y "... "

    // SOLUCIÓN: vector de punteros inteligentes a la base
    std::vector<std::unique_ptr<Animal>> refugio;
    refugio.push_back(std::make_unique<Perro>());
    refugio.push_back(std::make_unique<Gato>());
    std::cout << "Con vector<unique_ptr<Animal>> (correcto):" << std::endl;
    for (const auto& a : refugio) a->hacerSonido(); // "¡Guau!" y "¡Miau!"
    return 0;
}

```

La solución profesional es `vector<std::unique_ptr<Animal>>`: el vector guarda punteros (todos del mismo tamaño), y cada puntero apunta al objeto completo en memoria dinámica, sin rebanar nada. Además `unique_ptr` se encarga de liberar la memoria automáticamente: no hay

delete manual ni fugas. Ya viste este patrón en los ejemplos de figuras y notificaciones; ahora entiendes por qué es la forma estándar y no un capricho.

Hay una segunda solución válida para casos especiales: guardar los objetos derivados en sus propios vectores (`vector<Perro>`, `vector<Gato>`) y procesarlos por separado. Pero en cuanto necesitas mezclarlos y tratarlos uniformemente, el vector de punteros a la base es el camino.

Tres reglas de oro para no rebanar nunca: uno, las colecciones polimórficas siempre guardan punteros (inteligentes de preferencia) o referencias, jamás objetos por valor de la base. Dos, las funciones que reciben objetos polimórficos los reciben por referencia o puntero a la base, nunca por valor. Tres, si una función recibe la base por valor, el compilador no te avisa del rebanado: es tu responsabilidad. Un truco defensivo que usan algunos equipos es declarar el constructor de copia de la base como protegido o eliminado cuando la clase está pensada solo para usarse polimórficamente; así, el intento de copiar (y rebanar) falla en compilación en lugar de fallar en silencio en ejecución.

Métodos virtuales puros con implementación por defecto

Aquí hay un truco que sorprende a casi todos: en C++ un método virtual puro (= 0) puede tener implementación. Parece contradictorio: si es puro, la clase es abstracta y no se puede instanciar, ¿para qué darle cuerpo? La respuesta es elegante: para ofrecer un comportamiento por defecto que las derivadas pueden aprovechar, sin dejar de obligarlas a tomar una decisión consciente.

La sintaxis es peculiar: declaras `virtual void metodo() = 0;` dentro de la clase, y defines `void Clase::metodo() { ... }` fuera de ella. La clase sigue siendo abstracta (no instanciable), cada derivada debe declarar su propia versión, pero dentro de su versión puede llamar a la implementación base con `Clase::metodo()` para reutilizar el comportamiento común.

¿Para qué sirve en la práctica? Para el patrón "plantilla de comportamiento": la base define los pasos comunes que casi todas las derivadas querrán, pero exige que cada derivada escriba su propia versión (quizá para agregar pasos antes o después). Es un punto medio entre "virtual normal con implementación" (la derivada puede ignorarlo y heredar en silencio) y "virtual puro sin cuerpo" (cada derivada repite el código común).

```

#include <iostream>
#include <string>

// Clase abstracta: nadie puede crear un "Documento" a secas
class Documento {
public:
    // Virtual puro CON implementación por defecto
    virtual void exportar() const = 0;
    virtual ~Documento() {}
};

// La implementación por defecto vive fuera de la clase
void Documento::exportar() const {
    std::cout << "[Encabezado estandar del documento]" << std::endl;
}

class ReporteDoc : public Documento {
public:
    void exportar() const override {
        Documento::exportar(); // reutiliza el comportamiento común
        std::cout << "Contenido del reporte de ventas." << std::endl;
    }
};

class CartaDoc : public Documento {
public:
    void exportar() const override {
        // Esta derivada decide NO usar el encabezado estándar
        std::cout << "Querido lector: esta carta es personal." << std::endl;
    }
};

int main() {
    // Documento d; // NO compila: sigue siendo abstracta
    ReporteDoc r;
    CartaDoc c;
    r.exportar();
    std::cout << "---" << std::endl;
    c.exportar();
    return 0;
}

```

Nota lo que conseguiste: Documento es abstracta (el = 0 la mantiene así aunque tenga cuerpo), cada derivada está obligada a definir exportar (no hay herencia silenciosa de un

comportamiento quizá inadecuado), pero el código común del encabezado existe una sola vez y quien lo quiera lo invoca explícitamente con `Documento::exportar()`. Es obligación con ayuda: el contrato exige, la base apoya.

Úsalo cuando haya un comportamiento "casi siempre igual" pero quieras forzar a cada derivada a pensarlo. No lo uses si el comportamiento por defecto siempre es correcto: ahí basta un virtual normal. Y no lo uses si no hay nada común: ahí basta el virtual puro sin cuerpo. Como todo en diseño, es una herramienta de precisión, no un martillo.

Catálogo de errores comunes de esta expansión

Error 1: olvidar el destructor virtual en una interfaz. Las interfaces se usan siempre a través de punteros o referencias a la base, así que destruir sin destructor virtual es comportamiento indefinido. Toda interfaz lleva virtual `~Nombre() {}` aunque esté vacío. Si heredas de dos interfaces, ambas deben tenerlo.

Error 2: diamante accidental con herencia múltiple de clases con datos. Si A y B heredan de Base y C hereda de A y B, C tiene dos subobjetos Base: ¿a cuál te refieres con `obj.miembro`? El compilador marca ambigüedad. La salida profesional es no llegar ahí: hereda datos de una sola clase y capacidades de interfaces. La herencia virtual (`class A : virtual public Base`) existe como remedio, pero si la necesitas es casi seguro que el diseño debe simplificarse.

Error 3: ambigüedad de métodos con el mismo nombre en dos bases. Si Imprimible y Guardable declararan ambas un método `info()`, la derivada debe calificar: `obj.Imprimible::info()`. Para evitarlo, nombra los métodos de cada interfaz de forma distintiva (imprimir vs serializar, no "procesar" en ambas).

Error 4: slicing en contenedores y parámetros por valor. `vector<Base>` rebana, función que recibe Base por valor rebana, retorno por valor de un objeto derivado asignado a variable Base rebana. El compilador no avisa. Acostúmbrate a la sospecha sana: cada vez que veas un objeto polimórfico copiado por valor, algo huele mal.

Error 5: no marcar override en las implementaciones. Si escribes `void cobrar(double monto)` en la derivada pero la base decía `virtual bool cobrar(double)`, sin override el compilador cree que es un método nuevo y el polimorfismo no ocurre: la base sigue siendo "pura" y tu clase sigue siendo abstracta, con un error confuso. Con override, el compilador te avisa al instante del desacuerdo de firmas. Úsalo siempre.

Error 6: constructores enredados en herencia múltiple. Cada base se construye en el orden de declaración de la herencia, no en el orden de la lista de inicialización. Con varias bases,

escribe la lista de inicialización en el mismo orden en que declaraste las bases para no confundirte (y para que el compilador no te regañe con warnings).

Error 7: confundir "estrategia" con "herencia de comportamiento". Si agregas una clase derivada por cada variante de un algoritmo (TiendaOferta, TiendaNormal, TiendaCupon...), multiplicas clases y no puedes cambiar en caliente. Cuando el comportamiento varía independientemente del tipo de objeto, la estrategia (composición + interfaz) es el diseño correcto.

Error 8: llamar a la implementación por defecto del virtual puro sin calificar. Dentro de ReporteDoc::exportar, escribir exportar(); a secas sería recursión infinita (se llama a sí misma). Debe ser Documento::exportar(); con el nombre de la base calificando. Es el mismo cuidado que con cualquier llamada a versión base.

Práctica de la expansión

Ejercicio 1

Diseña un sistema de exportación de documentos con herencia múltiple de interfaces: crea las interfaces ExportablePDF (método exportarPDF() const) y ExportableHTML (método exportarHTML() const). Luego crea la clase Informe que implemente ambas, y una función publicar que reciba un objeto y lo exporte en ambos formatos usando las dos interfaces. Demuestra que una misma clase puede cumplir varios contratos.

Solución:

```

#include <iostream>
#include <string>

class ExportablePDF {
public:
    virtual std::string exportarPDF() const = 0;
    virtual ~ExportablePDF() {}
};

class ExportableHTML {
public:
    virtual std::string exportarHTML() const = 0;
    virtual ~ExportableHTML() {}
};

class Informe : public ExportablePDF, public ExportableHTML {
private:
    std::string titulo;
public:
    Informe(const std::string& t) : titulo(t) {}
    std::string exportarPDF() const override {
        return "[PDF] " + titulo;
    }
    std::string exportarHTML() const override {
        return "<h1>" + titulo + "</h1>";
    }
};

// Recibe el objeto concreto y lo ve por cada interfaz
void publicar(const Informe& inf) {
    const ExportablePDF& comoPDF = inf;
    const ExportableHTML& comoHTML = inf;
    std::cout << comoPDF.exportarPDF() << std::endl;
    std::cout << comoHTML.exportarHTML() << std::endl;
}

int main() {
    Informe inf("Resultados 2026");
    publicar(inf);
    return 0;
}

```

Explicación: Informe hereda de dos interfaces puras, sin datos compartidos, así que no hay diamante ni ambigüedad. publicar demuestra que un mismo objeto puede tratarse por

cualquiera de sus contratos. Si mañana agregas la interfaz ExportableCSV, Informe la implementa sin que publicar se rompa.

Ejercicio 2

Implementa el patrón Estrategia para un videojuego: la interfaz EstrategiaAtaque con atacar() const y nombre() const; tres estrategias (AtaqueEspada, AtaqueArco, AtaqueMagia) con distinto daño; y la clase Personaje que guarda la estrategia actual y permite cambiarla con setEstrategia. En main, crea un personaje, ataca con espada, cambia a magia y vuelve a atacar.

Solución:

```
#include <iostream>
#include <string>
#include <memory>

class EstrategiaAtaque {
public:
    virtual int atacar() const = 0;
    virtual std::string nombre() const = 0;
    virtual ~EstrategiaAtaque() {}
};

class AtaqueEspada : public EstrategiaAtaque {
public:
    int atacar() const override { return 15; }
    std::string nombre() const override { return "Espada"; }
};

class AtaqueArco : public EstrategiaAtaque {
public:
    int atacar() const override { return 10; }
    std::string nombre() const override { return "Arco"; }
};

class AtaqueMagia : public EstrategiaAtaque {
public:
    int atacar() const override { return 25; }
    std::string nombre() const override { return "Magia"; }
};

class Personaje {
private:
```

```

    std::string nombre;
    std::unique_ptr<EstrategiaAtaque> arma;
public:
    Personaje(const std::string& n)
        : nombre(n), arma(std::make_unique<AtaqueEspada>()) {}

    void setEstrategia(std::unique_ptr<EstrategiaAtaque> e) {
        arma = std::move(e);
    }

    void atacar() const {
        std::cout << nombre << " ataca con " << arma->nombre()
            << " causando " << arma->atacar() << " de daño." << std::endl;
    }
};

int main() {
    Personaje heroe("Aria");
    heroe.atacar(); // espada por defecto
    heroe.setEstrategia(std::make_unique<AtaqueMagia>());
    heroe.atacar(); // ahora magia, sin reconstruir el personaje
    heroe.setEstrategia(std::make_unique<AtaqueArco>());
    heroe.atacar();
    return 0;
}

```

Explicación: el personaje no hereda formas de atacar; las contiene como estrategia intercambiable. Cambiar de arma es cambiar un puntero, incluso a mitad del juego. Agregar "AtaqueDaga" mañana es una clase nueva de diez líneas. Con herencia habrías necesitado PersonajeConEspada, PersonajeConArco... y combinaciones imposibles.

Ejercicio 3

Demuestra y corrige el object slicing en un vector: crea la jerarquía Empleado2 (método virtual rol()) con derivadas Programador y Diseñador. Primero mete objetos en un vector<Empleado2> y muestra el rebanado; luego usa vector<unique_ptr<Empleado2>> y muestra el comportamiento correcto. Explica en un comentario qué se perdió en el primer caso.

Solución:

```

#include <iostream>
#include <string>
#include <vector>
#include <memory>

class Empleado2 {
public:
    virtual std::string rol() const { return "Empleado general"; }
    virtual ~Empleado2() {}
};

class Programador : public Empleado2 {
public:
    std::string rol() const override { return "Programador"; }
};

class Diseñador : public Empleado2 {
public:
    std::string rol() const override { return "Diseñador"; }
};

int main() {
    // CASO 1: rebanado. El vector solo guarda "la parte Empleado2".
    std::vector<Empleado2> plantilla;
    plantilla.push_back(Programador()); // se pierde lo de Programador
    plantilla.push_back(Diseñador());   // se pierde lo de Diseñador
    std::cout << "Por valor (rebanado):" << std::endl;
    for (const auto& e : plantilla) std::cout << "- " << e.rol() << std::endl;

    // CASO 2: correcto. El vector guarda punteros a objetos completos.
    std::vector<std::unique_ptr<Empleado2>> equipo;
    equipo.push_back(std::make_unique<Programador>());
    equipo.push_back(std::make_unique<Diseñador>());
    std::cout << "Por puntero (correcto):" << std::endl;
    for (const auto& e : equipo) std::cout << "- " << e->rol() << std::endl;
    return 0;
}

```

Explicación: en el caso 1, `push_back` copia solo la subparte `Empleado2` de cada objeto (el resto se "rebana") y además el tipo estático de cada elemento es `Empleado2`, así que `rol()` usa la versión base. En el caso 2 no hay copia del objeto: el puntero apunta al `Programador` completo y la llamada virtual se resuelve a su versión. Regla: colecciones polimórficas guardan punteros, no valores.

Ejercicio 4

Crea una clase abstracta `Sensor` con un método virtual puro `calibrar() = 0` que TENGA implementación por defecto (imprime "Calibración base aplicada"). Deriva `SensorTemperatura`, que en su `calibrar()` llama primero a la implementación base y luego imprime su ajuste propio; y deriva `SensorPresion`, que implementa `calibrar()` sin usar la base. Demuestra que `Sensor` sigue siendo abstracta (no instanciable) y que cada derivada decide si reutiliza el comportamiento común.

Solución:

```

#include <iostream>

class Sensor {
public:
    virtual void calibrar() = 0; // puro, pero con cuerpo abajo
    virtual ~Sensor() {}
};

// Implementación por defecto del método puro
void Sensor::calibrar() {
    std::cout << "Calibración base aplicada." << std::endl;
}

class SensorTemperatura : public Sensor {
public:
    void calibrar() override {
        Sensor::calibrar(); // reutiliza lo común (calificado, no recursivo)
        std::cout << "Ajuste fino de temperatura." << std::endl;
    }
};

class SensorPresion : public Sensor {
public:
    void calibrar() override {
        // Decide no usar la base: su calibración es totalmente propia
        std::cout << "Calibración especial de presión." << std::endl;
    }
};

int main() {
    // Sensor s; // NO compila: la clase sigue siendo abstracta
    SensorTemperatura st;
    SensorPresion sp;
    st.calibrar();
    std::cout << "---" << std::endl;
    sp.calibrar();
    return 0;
}

```

Explicación: el = 0 mantiene a Sensor abstracta aunque calibrar tenga cuerpo; cada derivada está obligada a definir su versión. SensorTemperatura reutiliza el comportamiento común llamándolo calificado (Sensor::calibrar()); sin el calificador sería recursión infinita. SensorPresion demuestra que usar la base es opcional: el contrato obliga a implementar, no a reutilizar.

Ejercicio 5

Decide con criterio: para cada caso indica si usarías herencia o composición y justifica en una frase. a) Un Rectangulo y un Punto para su esquina. b) Una clase ArchivoLog reutilizada por muchas clases. c) Distintos tipos de Enemigo en un juego que el motor trata uniformemente. d) Una clase ListaOrdenada que necesita la funcionalidad de vector pero sin exponer insert en medio.

Solución: a) Composición: el rectángulo tiene un punto (y otro para la esquina opuesta); un rectángulo no es un punto. b) Composición: cada clase tiene un miembro ArchivoLog y lo usa; heredar de él crearía una jerarquía falsa solo por reutilizar código. c) Herencia: hay relación "es un" genuina (un Orco es un Enemigo) y se necesita polimorfismo para que el motor los trate igual. d) Composición: heredar públicamente de vector expondría insert y rompería el orden (viola Liskov); con un vector privado expones solo push ordenado y acceso de lectura.

De herencia a composición: cómo refactorizar sin miedo

En código real te toparás con jerarquías heredadas (nunca mejor dicho) que abusan de la herencia: clases que heredan para reutilizar dos métodos, niveles de seis de profundidad, derivadas que deshabilitan métodos de la base lanzando errores. Reescribir todo de golpe es peligroso; la forma profesional es refactorizar por pasos, y el destino más común es pasar de herencia a composición manteniendo el polimorfismo donde sí se necesita.

El procedimiento es mecánico. Paso uno: identifica qué usa realmente la derivada de la base. Muchas veces son uno o dos métodos, no toda la interfaz. Paso dos: extrae esos métodos a una clase o interfaz pequeña. Paso tres: cambia la herencia por un miembro de ese nuevo tipo y delega. Paso cuatro: si el polimorfismo era necesario, conserva una interfaz abstracta mínima y haz que la clase la implemente por composición interna.

Mira un caso típico: una clase EmpleadoTemporal que heredaba de Empleado solo para reutilizar el cálculo de impuestos, aunque conceptualmente el temporal no compartía el resto del contrato (rompía Liskov cada vez que alguien lo trataba como empleado de planta).

```

#include <iostream>
#include <string>

// Paso 1 y 2: extraer lo reutilizable a una pieza pequeña y honesta
class CalculadoraImpuestos {
public:
    double impuestoSobre(double monto) const {
        return monto * 0.16;
    }
};

// Paso 3: composición en lugar de herencia falsa
class EmpleadoTemporal {
private:
    std::string nombre;
    double pagoPorHora;
    int horas;
    CalculadoraImpuestos calc; // tiene una, no es una
public:
    EmpleadoTemporal(const std::string& n, double ph, int h)
        : nombre(n), pagoPorHora(ph), horas(h) {}

    double pagoBruto() const { return pagoPorHora * horas; }
    double pagoNeto() const {
        double bruto = pagoBruto();
        return bruto - calc.impuestoSobre(bruto);
    }
    void mostrar() const {
        std::cout << nombre << ": neto $" << pagoNeto() << std::endl;
    }
};

int main() {
    EmpleadoTemporal t("Paco", 120, 40);
    t.mostrar();
    return 0;
}

```

El resultado es una clase que dice la verdad: un empleado temporal tiene una forma de calcular impuestos, no es un empleado de planta disfrazado. La calculadora, además, quedó reutilizable para cualquier otra clase (proveedores, freelancers) sin arrastrar jerarquías. Cuando refactorizas así, cada paso compila y cada paso se puede probar: nunca hay un momento en que "todo está roto a la mitad". Esa es la diferencia entre refactorizar y reescribir.

Lista de verificación antes de diseñar una jerarquía

Guarda estas siete preguntas y repásalas cada vez que vayas a crear una relación entre clases. Responderlas te toma dos minutos y te ahorra semanas de refactorización.

1. ¿La relación "es un" suena natural en una frase simple? Si tienes que explicarla con "bueno, más o menos", usa composición.
2. ¿Voy a tratar estos objetos de forma uniforme a través de la base? Si nunca usarás punteros o referencias a la base, el polimorfismo no te da nada y la herencia solo acopla.
3. ¿La base fue diseñada para heredarse? Busca destructor virtual y documentación de qué métodos se pueden redefinir. Heredar de una clase concreta no preparada es frágil.
4. ¿Estoy heredando datos de más de una base? Si es así, detente: convierte las bases en interfaces puras o cambia a composición.
5. ¿Alguna derivada necesitará deshabilitar o contradecir un método de la base? Eso es Liskov roto desde el diseño: rediseña ahora, no después.
6. ¿El comportamiento varía en tiempo de ejecución o en combinaciones impredecibles? Entonces es un caso para el patrón Estrategia, no para más niveles de herencia.
7. ¿La jerarquía tiene más de tres niveles? Cada nivel multiplica la dificultad de entender el código. Aplana o compone.

Si alguna respuesta te incomoda, no la ignores: la incomodidad es tu criterio de diseñador despertando. Los mejores programadores de C++ no son los que usan más herencia, sino los que saben exactamente cuándo no usarla.

Resumen de la expansión

- Herencia múltiple legítima: solo de interfaces puras (sin datos). Combina capacidades ("puede hacer") sin diamantes ni ambigüedades de estado.
- Clases abstractas como contratos: la base define qué se promete (métodos virtuales puros); las derivadas, cómo se cumple. El código cliente depende de la abstracción y no cambia al agregar implementaciones.
- Patrón Estrategia: familia de algoritmos intercambiables tras una interfaz, guardada por composición en el cliente. Cambia comportamiento en tiempo de ejecución sin explosión de clases.

- Composición vs herencia: "es un" + polimorfismo = herencia; "tiene un", reutilización sin jerarquía o comportamiento cambiante = composición. La prueba del enunciado revela diseños forzados.
- Los streams de iostream son herencia real y bien diseñada: ios como raíz del estado, istream/ostream por dirección, herencia múltiple con base virtual en iostream, y derivadas por destino (archivos, cadenas).
- Object slicing: las colecciones polimórficas guardan punteros inteligentes a la base, nunca objetos por valor. El compilador no avisa del rebanado: la disciplina es tuya.
- Virtual puro con implementación: la clase sigue abstracta y obliga a cada derivada a definir el método, pero ofrece un comportamiento común reutilizable con llamada calificada.
- Con esto cierras la unidad 7 con criterio de arquitecto: ya no solo aplicas los pilares, decides cómo combinarlos.

UNIDAD 8: Técnicas avanzadas

Esta unidad te lleva al C++ que usan los profesionales todos los días. Tres herramientas que multiplican tu poder: la sobrecarga de operadores (hacer que tus objetos se sumen, se comparen y se impriman como tipos nativos), las plantillas (escribir código una vez que funcione con cualquier tipo) y la STL con su contenedor estrella, el vector.

Estas tres ideas comparten una filosofía: escribir menos código que haga más, sin perder velocidad. La sobrecarga hace tu código expresivo. Las plantillas lo hacen genérico. La STL te da componentes probados por décadas. Juntas, son la diferencia entre programar en C++ y programar C con clases.

Al terminar sabrás: sobrecargar operadores para tus clases, escribir funciones y clases plantilla, y usar `std::vector` como tu contenedor predeterminado.

Explicación detallada

Clase 22: Sobrecarga de operadores

Ya sobrecargaste funciones (mismo nombre, distintos parámetros). Ahora sobrecargarás operadores: definir qué significa `+`, `==`, `<<` y otros para tus propios tipos. El objetivo es que tus objetos se usen con la misma naturalidad que los números.

El ejemplo clásico: una clase `Fraccion`.

```

class Fraccion {
private:
    int num, den;

public:
    Fraccion(int n, int d) : num(n), den(d) {}

    Fraccion operator+(const Fraccion& otra) const {
        return Fraccion(num * otra.den + otra.num * den, den * otra.den);
    }

    bool operator==(const Fraccion& otra) const {
        return num * otra.den == otra.num * den;
    }

    void mostrar() const {
        std::cout << num << "/" << den << std::endl;
    }
};

int main() {
    Fraccion a(1, 2), b(1, 3);
    Fraccion c = a + b;    // suma de fracciones con +!
    c.mostrar();           // 5/6
    std::cout << (a == b) << std::endl; // 0 (falso)
    return 0;
}

```

La sintaxis Tipo operator+(...) define el comportamiento. Dentro, tienes acceso a los privados de ambas fracciones. El resultado: $a + b$ se lee como matemática, no como `a.sumar(b)`.

El operador `<<` para imprimir merece atención especial: debe ser una función externa (no método), porque el operando izquierdo es `cout`, no tu objeto:

```

std::ostream& operator<<(std::ostream& os, const Fraccion& f) {
    os << f.getNum() << "/" << f.getDen();
    return os;
}
// Ahora: std::cout << a << std::endl; funciona

```

Devuelve el flujo por referencia para permitir encadenar: `cout << a << b`. Necesitarás getters públicos o declararla friend (una función amiga con acceso a privados). Este operador es casi obligatorio en toda clase que diseñes: facilita depurar e imprimir.

Reglas y buenas prácticas: no puedes inventar operadores nuevos ni cambiar la precedencia. Sobrecarga solo cuando el significado sea obvio: + para sumar/ combinar, == para igualdad real. Si el operador confundiría, mejor un método con nombre. Y mantén la semántica esperada: == debe ser simétrico, + no debe modificar los operandos.

Operadores comunes de sobrecargar: + - * / (aritmética), == != < > (comparación), << >> (flujos), [] (acceso, útil en contenedores), = (asignación, con cuidado). Los verás en acción en bibliotecas reales: los strings se suman, los vectores se comparan, todo gracias a esto.

Clase 23: Plantillas (templates)

¿Te has cansado de escribir la misma lógica para int, double y string? Las plantillas resuelven eso: escribes el código una vez con un tipo genérico T, y el compilador genera las versiones concretas que uses.

```
template <typename T>
T mayor(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    std::cout << mayor(3, 7) << std::endl;           // T = int
    std::cout << mayor(3.5, 2.1) << std::endl;        // T = double
    std::cout << mayor(std::string("a"), std::string("b")) << std::endl;
    return 0;
}
```

template <typename T> declara "lo siguiente funciona con cualquier tipo T". El compilador deduce T de los argumentos y genera el código. No hay costo en ejecución: las plantillas se resuelven en compilación, así que son tan rápidas como el código escrito a mano.

También hay clases plantilla:

```

template <typename T>
class Caja {
private:
    T contenido;
public:
    Caja(T c) : contenido(c) {}
    T ver() const { return contenido; }
};

int main() {
    Caja<int> ci(42);
    Caja<std::string> cs("hola");
    std::cout << ci.ver() << " " << cs.ver() << std::endl;
    return 0;
}

```

Caja<int> y Caja<std::string> son clases distintas generadas del mismo molde. Así funciona std::vector: es una clase plantilla, y vector<int> es "vector de enteros".

Las plantillas son la base de la programación genérica en C++ y de toda la STL. Su única desventaja didáctica: los errores del compilador con plantillas pueden ser largos y crípticos. Con la práctica aprendes a leer la primera línea del error, que suele decir lo importante.

Clase 24: La STL y std::vector

La STL (Standard Template Library) es la biblioteca estándar de C++: contenedores, algoritmos e iteradores genéricos, probados durante décadas. Sus tres pilares: contenedores (guardan datos: vector, map, set), algoritmos (operan sobre datos: sort, find, count) e iteradores (el "pegamento" que conecta ambos).

El contenedor que usarás el 80% del tiempo es std::vector: un arreglo dinámico que crece solo.


```
#include <vector>

std::vector<int> v;           // vacío
v.push_back(10);             // agrega al final
v.push_back(20);
v.push_back(30);

std::cout << v[0] << std::endl;    // 10
std::cout << v.size() << std::endl; // 3

for (int x : v) std::cout << x << " "; // for de rango

v.pop_back(); // quita el último
```

Ventajas sobre arreglos crudos: crece dinámicamente (adiós a `new[]`), sabe su tamaño (`size()`), se libera solo (adiós a `delete[]`), y con `.at(i)` verifica límites lanzando una excepción si te pasas (`v[i]` no verifica, por velocidad; `.at(i)` sí, por seguridad).

Operaciones esenciales: `push_back` agrega, `pop_back` quita el último, `size` da el tamaño, `empty` dice si está vacío, `clear` lo vacía, `front/back` dan el primero/último, `insert/erase` en posiciones (más costosos). Puedes inicializar con valores: `std::vector<int> v = {1, 2, 3};`.

Y los algoritmos de la STL trabajan con vectores (y otros contenedores) directamente:

```
#include <algorithm>

std::vector<int> v = {5, 2, 8, 1, 9};
std::sort(v.begin(), v.end()); // ordena: 1 2 5 8 9

if (std::find(v.begin(), v.end(), 8) != v.end()) {
    std::cout << "Encontrado" << std::endl;
}

int pares = std::count_if(v.begin(), v.end(), [](int x){ return x % 2 == 0; });
```

`begin()` y `end()` son iteradores: el inicio y "uno después del final". `sort`, `find`, `count_if` son algoritmos genéricos que funcionan con cualquier contenedor. La expresión `[](int x){ return x % 2 == 0; }` es una lambda: una función anónima pequeña. Las lambdas merecen su propio estudio, pero esta forma básica la usarás mucho.

Regla práctica: ante la duda, usa vector. Reemplaza el 95% de los usos de arreglos crudos y de `new[]`. Es más seguro, más expresivo y igual de rápido para uso normal.

Ejemplos paso a paso

Ejemplo 1: Clase Punto con operadores

```
#include <iostream>

class Punto {
private:
    double x, y;

public:
    Punto(double px = 0, double py = 0) : x(px), y(py) {}

    Punto operator+(const Punto& otro) const {
        return Punto(x + otro.x, y + otro.y);
    }

    bool operator==(const Punto& otro) const {
        return x == otro.x && y == otro.y;
    }

    friend std::ostream& operator<<(std::ostream& os, const Punto& p) {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
};

int main() {
    Punto a(1, 2), b(3, 4);
    Punto c = a + b;
    std::cout << "a = " << a << std::endl;
    std::cout << "b = " << b << std::endl;
    std::cout << "a + b = " << c << std::endl;
    std::cout << "a == b? " << (a == b) << std::endl;
    return 0;
}
```

friend permite que `operator<<` acceda a los privados. Los puntos se suman, se comparan y se imprimen con naturalidad total. Compara esto con una versión con métodos `sumar()` y `mostrar()`: la sobrecarga gana en legibilidad.

Ejemplo 2: Plantilla de intercambio y búsqueda

```
#include <iostream>
#include <string>

template <typename T>
void intercambiar(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

template <typename T>
int buscar(const T arr[], int tamano, const T& valor) {
    for (int i = 0; i < tamano; i++) {
        if (arr[i] == valor) return i;
    }
    return -1;
}

int main() {
    int x = 1, y = 2;
    intercambiar(x, y);
    std::cout << x << " " << y << std::endl;

    std::string a = "hola", b = "mundo";
    intercambiar(a, b);
    std::cout << a << " " << b << std::endl;

    double nums[] = {1.5, 2.5, 3.5};
    std::cout << "Posicion: " << buscar(nums, 3, 2.5) << std::endl;
    return 0;
}
```

Una sola función intercambiar sirve para int, string y cualquier tipo copiable. El compilador genera cada versión automáticamente. Imagina escribir esto a mano para cada tipo: las plantillas te ahorran ese trabajo para siempre.

Ejemplo 3: Vector como agenda

```

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

int main() {
    std::vector<std::string> agenda;

    agenda.push_back("Ana");
    agenda.push_back("Luis");
    agenda.push_back("Maria");
    agenda.push_back("Luis");

    std::cout << "Contactos (" << agenda.size() << "):" << std::endl;
    for (const auto& nombre : agenda) {
        std::cout << "- " << nombre << std::endl;
    }

    // Ordenar
    std::sort(agenda.begin(), agenda.end());
    std::cout << "\nOrdenados:" << std::endl;
    for (const auto& nombre : agenda) {
        std::cout << "- " << nombre << std::endl;
    }

    // Buscar
    auto it = std::find(agenda.begin(), agenda.end(), "Maria");
    if (it != agenda.end()) {
        std::cout << "\nMaria encontrada." << std::endl;
        agenda.erase(it); // eliminar
    }

    std::cout << "Restan " << agenda.size() << " contactos." << std::endl;
    return 0;
}

```

Agenda completa con vector y algoritmos: agregar, ordenar, buscar, eliminar. Sin new, sin delete, sin tamaños manuales. auto deduce el tipo del iterador. Este es el C++ moderno en su forma más productiva.

Práctica

Ejercicio 1

Completa la clase Fraccion: agrega operator-, operator* y operator!=. Prueba con 1/2 y 1/3.

Solución:

```
#include <iostream>

class Fraccion {
private:
    int num, den;

public:
    Fraccion(int n, int d) : num(n), den(d) {}

    Fraccion operator+(const Fraccion& o) const {
        return Fraccion(num * o.den + o.num * den, den * o.den);
    }
    Fraccion operator-(const Fraccion& o) const {
        return Fraccion(num * o.den - o.num * den, den * o.den);
    }
    Fraccion operator*(const Fraccion& o) const {
        return Fraccion(num * o.num, den * o.den);
    }
    bool operator==(const Fraccion& o) const {
        return num * o.den == o.num * den;
    }
    bool operator!=(const Fraccion& o) const {
        return !(*this == o);
    }
    friend std::ostream& operator<<(std::ostream& os, const Fraccion& f) {
        return os << f.num << "/" << f.den;
    }
};

int main() {
    Fraccion a(1, 2), b(1, 3);
    std::cout << "a + b = " << a + b << std::endl; // 5/6
    std::cout << "a - b = " << a - b << std::endl; // 1/6
    std::cout << "a * b = " << a * b << std::endl; // 1/6
    std::cout << "a != b: " << (a != b) << std::endl;
    return 0;
}
```

Explicación: != se define en términos de == (reutilización). *this es el objeto actual: !(*this == o) significa "no soy igual a o". Las fracciones no se simplifican aquí; agregar un método simplificar con el máximo común divisor sería un gran ejercicio extra.

Ejercicio 2

Escribe una función plantilla contarMayores que reciba un vector<T>, un valor T y devuelva cuántos elementos son mayores que el valor. Pruébala con vector<int> y vector<double>.

Solución:

```
#include <iostream>
#include <vector>

template <typename T>
int contarMayores(const std::vector<T>& v, const T& valor) {
    int contador = 0;
    for (const T& elemento : v) {
        if (elemento > valor) contador++;
    }
    return contador;
}

int main() {
    std::vector<int> enteros = {3, 8, 1, 9, 4};
    std::vector<double> decimales = {1.5, 9.9, 3.3};

    std::cout << contarMayores(enteros, 4) << std::endl;    // 2
    std::cout << contarMayores(decimales, 2.0) << std::endl; // 2
    return 0;
}
```

Explicación: funciona con cualquier T que soporte >. Los parámetros son const& para no copiar el vector. Esta es la esencia de la programación genérica: un algoritmo, infinitos tipos.

Ejercicio 3

Crea una clase plantilla Pila<T> con métodos push, pop, cima y vacia, usando un vector interno. Pruébala con int y con string.

Solución:

```

#include <iostream>
#include <vector>
#include <string>

template <typename T>
class Pila {
private:
    std::vector<T> datos;

public:
    void push(const T& valor) { datos.push_back(valor); }

    void pop() {
        if (!datos.empty()) datos.pop_back();
    }

    T cima() const { return datos.back(); }
    bool vacia() const { return datos.empty(); }
    int tamano() const { return datos.size(); }
};

int main() {
    Pila<int> p;
    p.push(10); p.push(20); p.push(30);
    while (!p.vacia()) {
        std::cout << p.cima() << " ";
        p.pop();
    }
    std::cout << std::endl;

    Pila<std::string> s;
    s.push("hola"); s.push("mundo");
    std::cout << s.cima() << std::endl;
    return 0;
}

```

Explicación: una pila (LIFO: lo último que entra es lo primero que sale) implementada en minutos gracias a vector. `push_back/pop_back/back()` son exactamente las operaciones de pila. La plantilla la hace universal.

Ejercicio 4

Usa vector y los algoritmos de la STL para: leer 8 calificaciones, ordenarlas, mostrar la más alta y la más baja, y contar cuántas son aprobatorias (≥ 6).

Solución:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<double> califs;
    for (int i = 0; i < 8; i++) {
        double c;
        std::cout << "Calificacion " << i + 1 << ": ";
        std::cin >> c;
        califs.push_back(c);
    }

    std::sort(califs.begin(), califs.end());

    std::cout << "Ordenadas: ";
    for (double c : califs) std::cout << c << " ";
    std::cout << std::endl;

    std::cout << "Mas baja: " << califs.front() << std::endl;
    std::cout << "Mas alta: " << califs.back() << std::endl;

    int aprobadas = std::count_if(califs.begin(), califs.end(),
                                  [](double c){ return c >= 6; });
    std::cout << "Aprobatorias: " << aprobadas << std::endl;
    return 0;
}
```

Explicación: después de ordenar, el menor está al frente y el mayor atrás. `count_if` con lambda cuenta las que cumplen la condición. Tres algoritmos de la STL en un programa corto: así se programa en C++ moderno.

Ejercicio 5

Sobrecarga los operadores `<` y `>` para la clase `Punto` (comparar por distancia al origen) y luego ordena un `vector<Punto>` con `std::sort`. Necesitarás `<cmath>`.

Solución:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

class Punto {
private:
    double x, y;
public:
    Punto(double px, double py) : x(px), y(py) {}

    double distanciaOrigen() const {
        return std::sqrt(x * x + y * y);
    }

    bool operator<(const Punto& o) const {
        return distanciaOrigen() < o.distanciaOrigen();
    }

    friend std::ostream& operator<<(std::ostream& os, const Punto& p) {
        return os << "(" << p.x << ", " << p.y << ")";
    }
};

int main() {
    std::vector<Punto> puntos = {Punto(3, 4), Punto(1, 1), Punto(0, 5)};
    std::sort(puntos.begin(), puntos.end());
    for (const auto& p : puntos) std::cout << p << " ";
    std::cout << std::endl;
    return 0;
}
```

Explicación: `std::sort` necesita `operator<` para ordenar: al definirlo, tus objetos se vuelven "ordenables" automáticamente. El `sort` usa tu `<` internamente. Así se integran tus clases con la STL: defines los operadores y los algoritmos hacen el resto.

Resumen de la unidad

- Sobrecarga de operadores: define `+`, `==`, `<<`, etc. para tus clases. Haz que el significado sea obvio y respeta la semántica esperada.

- `operator<<` como función externa (o friend) permite imprimir tus objetos con `cout`.
- Plantillas: `template <typename T>` escribe código genérico una vez; el compilador genera versiones por tipo, sin costo en ejecución.
- La STL = contenedores + algoritmos + iteradores. `std::vector` es tu contenedor predeterminado: dinámico, seguro y rápido.
- Algoritmos clave: `sort`, `find`, `count_if` con `begin()/end()`. Las lambdas `[](...)` permiten criterios personalizados.
- En la unidad 9 verás más STL (mapas y conjuntos), manejo de errores con excepciones y archivos.

Profundización: plantillas, iteradores y algoritmos

Ya usas plantillas básicas y vectores. Ahora profundiza: especialización de plantillas, más operadores sobrecargados (asignación, subíndice, incremento), iteradores como concepto y una gira por los algoritmos más útiles de la STL. Esta es la caja de herramientas del C++ genérico profesional.

Especialización de plantillas

A veces el código genérico no sirve para un tipo concreto. La especialización permite una versión a medida:

```

#include <iostream>
#include <string>

template <typename T>
std::string describir(T valor) {
    return "Valor generico";
}

// Especialización para std::string
template <>
std::string describir<std::string>(std::string valor) {
    return "Texto de " + std::to_string(valor.length()) + " caracteres";
}

int main() {
    std::cout << describir(42) << std::endl;
    std::cout << describir(std::string("hola")) << std::endl;
    return 0;
}

```

La versión genérica atiende a todos; la especializada, solo a string. El compilador elige la más específica. Útil cuando un tipo necesita lógica distinta (comparaciones especiales, formatos propios).

Más operadores: =, [] y ++

El operador de asignación = merece cuidado: debe devolver *this por referencia para permitir a = b = c, y debe manejar la auto-asignación:

```

class Contador {
    int valor;
public:
    Contador& operator=(const Contador& otro) {
        if (this != &otro) { // protege contra a = a
            valor = otro.valor;
        }
        return *this;
    }
};

```

El operador [] da acceso tipo arreglo a tus clases:

```
class ArregloSeguro {
    std::vector<int> datos;
public:
    ArregloSeguro(int n) : datos(n, 0) {}
    int& operator[](int i) { return datos[i]; } // lectura/escritura
    int operator[](int i) const { return datos[i]; } // solo lectura
};
```

Dos versiones: la no-const devuelve referencia modificable; la const, valor de solo lectura. Así `arr[i] = 5` funciona y en objetos const solo se puede leer.

Los operadores ++ tienen dos formas (prefijo y postfijo); el postfijo lleva un int ficticio para distinguirse:

```
Contador& operator++() { valor++; return *this; } // ++c
Contador operator++(int) { Contador t = *this; valor++; return t; } // c++
```

El postfijo devuelve una copia del valor anterior: es más costoso. Prefiere ++i sobre i++ en ciclos con objetos.

Iteradores: el pegamento de la STL

Un iterador es un "puntero generalizado" para recorrer contenedores. Todos los contenedores ofrecen `begin()` (inicio) y `end()` (uno después del final):

```
std::vector<int> v = {10, 20, 30};
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << " "; // desreferencia como puntero
}
```

`auto` evita escribir el tipo largo del iterador. Los iteradores permiten escribir algoritmos genéricos: `sort` no sabe qué contenedor recorre, solo usa iteradores. Hay categorías (de avance simple, bidireccionales, aleatorios); `vector` tiene los más potentes.

Gira de algoritmos esenciales

Además de `sort`, `find` y `count_if`, estos te cambiarán la vida:

- `std::reverse`: invierte el orden.
- `std::max_element` / `std::min_element`: devuelven iterador al mayor/menor.

- `std::accumulate` (en `<numeric>`): suma (o combina) todo.
- `std::transform`: aplica una función a cada elemento y guarda el resultado.
- `std::remove_if` + `erase`: el famoso "erase-remove" para borrar los que cumplan condición.
- `std::all_of` / `std::any_of` / `std::none_of`: preguntas lógicas sobre el contenedor.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

int main() {
    std::vector<int> v = {5, 2, 8, 1, 9, 2};

    std::cout << "Suma: " << std::accumulate(v.begin(), v.end(), 0) << std::endl;

    auto maxIt = std::max_element(v.begin(), v.end());
    std::cout << "Maximo: " << *maxIt << std::endl;

    // Duplicar todo con transform
    std::vector<int> dobles(v.size());
    std::transform(v.begin(), v.end(), dobles.begin(), [](int x){ return x * 2; });

    // Borrar los menores a 5 (erase-remove)
    v.erase(std::remove_if(v.begin(), v.end(), [](int x){ return x < 5; }),
v.end());

    std::cout << "Mayores o iguales a 5: ";
    for (int x : v) std::cout << x << " ";
    std::cout << std::endl;

    std::cout << "Alguno > 8? " << std::any_of(v.begin(), v.end(), [](int x){
return x > 8; }) << std::endl;
    return 0;
}
```

El erase-remove merece explicación: `remove_if` no borra, reordena los "a conservar" al frente y devuelve el iterador donde empiezan los descartados; `erase` los elimina de verdad. Es un modismo que verás en todo código C++ serio.

Lambdas, un poco más

Las lambdas pueden capturar variables del entorno con `&` (por referencia) o `=` (por copia):

```
int limite = 5;
int contador = std::count_if(v.begin(), v.end(), [&](int x){ return x > limite; });
```

[&] captura todo por referencia: la lambda ve limite. Es la forma de parametrizar algoritmos sin escribir funciones nombradas.

Más ejemplos

Ejemplo: clase Matriz con operator()

Para matrices, el operador () con dos índices es el estándar (ya que [] solo acepta un índice):

```
#include <iostream>
#include <vector>

class Matriz {
private:
    int filas, columnas;
    std::vector<double> datos;
public:
    Matriz(int f, int c) : filas(f), columnas(c), datos(f * c, 0.0) {}

    double& operator()(int f, int c) { return datos[f * columnas + c]; }
    double operator()(int f, int c) const { return datos[f * columnas + c]; }

    void mostrar() const {
        for (int i = 0; i < filas; i++) {
            for (int j = 0; j < columnas; j++) {
                std::cout << (*this)(i, j) << "t";
            }
            std::cout << std::endl;
        }
    }
};

int main() {
    Matriz m(2, 3);
    m(0, 0) = 1; m(0, 1) = 2; m(0, 2) = 3;
    m(1, 0) = 4; m(1, 1) = 5; m(1, 2) = 6;
    m.mostrar();
    return 0;
}
```

Los datos viven en un solo vector (eficiente y contiguo); `operator()` traduce (fila, columna) a posición. `m(0,1) = 2` funciona porque devuelve referencia. Así se diseñan las clases numéricas profesionales.

Práctica adicional

Ejercicio A

Escribe una función plantilla `intercambiarSiMayor(T& a, T& b)` que intercambie solo si `a > b`. Pruébala con `int`, `double` y `string`.

Solución:

```
#include <iostream>
#include <string>

template <typename T>
void intercambiarSiMayor(T& a, T& b) {
    if (a > b) {
        T temp = a;
        a = b;
        b = temp;
    }
}

int main() {
    int x = 9, y = 3;
    intercambiarSiMayor(x, y);
    std::cout << x << " " << y << std::endl; // 3 9

    std::string a = "zebra", b = "araña";
    intercambiarSiMayor(a, b);
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

Explicación: funciona con cualquier `T` que soporte `>` y copia. Es el corazón del ordenamiento burbuja, generalizado.

Ejercicio B

Usa `accumulate`, `max_element` y `any_of` para analizar un vector de temperaturas: promedio, máxima y si alguna supera 40 grados.

Solución:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

int main() {
    std::vector<double> temps = {36.5, 38.2, 41.0, 37.1, 39.8};
    double suma = std::accumulate(temps.begin(), temps.end(), 0.0);
    std::cout << "Promedio: " << suma / temps.size() << std::endl;
    std::cout << "Maxima: " << *std::max_element(temps.begin(), temps.end()) <<
std::endl;
    bool fiebreAlta = std::any_of(temps.begin(), temps.end(),
                                  [](double t){ return t > 40.0; });
    std::cout << "Alguna > 40? " << fiebreAlta << std::endl;
    return 0;
}
```

Explicación: tres algoritmos, cero ciclos manuales. El 0.0 inicial en `accumulate` fija el tipo `double`. Este estilo declarativo ("qué quiero", no "cómo iterar") es la marca del C++ moderno.

Ejercicio C

Sobrecarga `operator==` y `operator<` para una clase `Alumno2` (nombre, promedio) comparando por promedio. Ordena un `vector<Alumno2>` por promedio con `sort`.

Solución:


```

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

class Alumno2 {
    std::string nombre;
    double promedio;
public:
    Alumno2(const std::string& n, double p) : nombre(n), promedio(p) {}
    bool operator==(const Alumno2& o) const { return promedio == o.promedio; }
    bool operator<(const Alumno2& o) const { return promedio < o.promedio; }
    friend std::ostream& operator<<(std::ostream& os, const Alumno2& a) {
        return os << a.nombre << " (" << a.promedio << ")";
    }
};

int main() {
    std::vector<Alumno2> grupo = {"Ana", 9.2}, {"Luis", 7.5}, {"Maria", 8.8};
    std::sort(grupo.begin(), grupo.end());
    for (const auto& a : grupo) std::cout << a << std::endl;
    return 0;
}

```

Explicación: sort usa tu operator< sin saber nada de alumnos. Integrar tus clases con la STL mediante operadores es una habilidad profesional clave.

Ejercicio D

Implementa con transform y una lambda un programa que convierta un vector<string> a mayúsculas (usa toupper de <cctype>).

Solución:

```

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cctype>

int main() {
    std::vector<std::string> palabras = {"hola", "mundo", "c++"};
    std::transform(palabras.begin(), palabras.end(), palabras.begin(),
        [](std::string s) {
            for (char& c : s) c = std::toupper(c);
            return s;
        });
    for (const auto& p : palabras) std::cout << p << " ";
    std::cout << std::endl;
    return 0;
}

```

Explicación: transform escribe el resultado en el mismo vector (begin como destino). La lambda recibe por valor, modifica su copia y la devuelve. Combinar algoritmos con lambdas es el estilo funcional moderno de C++.

Expansión Unidad 8: plantillas avanzadas, operadores y algoritmos

En las partes anteriores de esta unidad aprendiste la sobrecarga de operadores con Fraccion y Punto, las plantillas básicas de funciones y clases, y la STL con std::vector, algoritmos esenciales e iteradores. Viste especialización de plantillas, los operadores =, [] y ++, y una gira de algoritmos como sort, find, accumulate y transform.

Esta expansión completa tu caja de herramientas genérica: plantillas con varios parámetros y valores por defecto, el criterio para elegir entre plantillas y herencia cuando quieres código genérico, los utilitarios std::pair y std::tuple, la familia completa de operadores de comparación y cómo se relacionan, los operadores de conversión, los funtores como alternativa seria a las lambdas, std::function para guardar "cualquier cosa llamable", y más algoritmos indispensables: copy, fill, replace y unique. Todo con código completo y compilable, catálogo de errores y ejercicios con solución.

Plantillas con múltiples parámetros y valores por defecto

Hasta ahora tus plantillas tenían un solo parámetro (typename T). Pero las plantillas aceptan varios parámetros de tipo, parámetros que no son tipos (como un entero constante) y valores por defecto, igual que los parámetros de una función. Esto permite diseñar componentes genéricos mucho más expresivos.

El caso más común: dos tipos independientes. Piensa en una función que combina dos valores de tipos distintos, o en una clase Pareja que guarda dos datos que no tienen por qué ser del mismo tipo.

```
#include <iostream>
#include <string>

// Dos parámetros de tipo independientes
template <typename T1, typename T2>
void mostrarPareja(const T1& primero, const T2& segundo) {
    std::cout << primero << " - " << segundo << std::endl;
}

// Parámetro no-tipo: un entero constante conocido en compilación
template <typename T, int N>
class ArregloFijo {
private:
    T datos[N];
public:
    ArregloFijo() {
        for (int i = 0; i < N; i++) datos[i] = T{};
    }
    T& operator[](int i) { return datos[i]; }
    int tamano() const { return N; }
};

// Valores por defecto: si no dices nada, T2 es igual a T1
template <typename T1, typename T2 = T1>
class Pareja {
private:
    T1 primero;
    T2 segundo;
public:
    Pareja(const T1& p, const T2& s) : primero(p), segundo(s) {}
    void mostrar() const {
        std::cout << "(" << primero << ", " << segundo << ")" << std::endl;
    }
};
```

```

int main() {
    mostrarPareja(42, std::string("respuesta")); // T1=int, T2=string
    mostrarPareja(3.14, 2);                      // T1=double, T2=int

    ArregloFijo<int, 5> arr; // arreglo de 5 enteros, tamaño fijo en compilación
    arr[0] = 10;
    std::cout << "Tamano: " << arr.tamano() << ", [0]=" << arr[0] << std::endl;

    Pareja<int> p1(1, 2);                      // T2 toma el valor por defecto: int
    Pareja<int, std::string> p2(1, "uno"); // tipos distintos, explícitos
    p1.mostrar();
    p2.mostrar();
    return 0;
}

```

Tres ideas nuevas aquí. Una: con varios parámetros de tipo, el compilador deduce cada uno por separado de los argumentos. Dos: los parámetros no-tipo (como `int N`) deben ser constantes en tiempo de compilación; por eso `ArregloFijo<int, 5>` reserva exactamente 5 espacios sin memoria dinámica, como `std::array` (que de hecho está implementado así: `template <typename T, size_t N>`). Tres: los valores por defecto funcionan igual que en funciones: los parámetros con defecto van al final, y puedes omitirlos al usar la plantilla.

Un detalle importante sobre la deducción: en `mostrarPareja(3.14, 2)`, `T1` se deduce como `double` y `T2` como `int` sin que escribas nada. Pero en las clases plantilla la deducción de C++17 también funciona en muchos casos, aunque escribir los tipos explícitamente (`Pareja<int, std::string>`) siempre es válido y a veces más claro. Cuando la deducción falle o elijas tipos ambiguos, el compilador te lo dirá; especificar los tipos entre `< >` resuelve casi todo.

¿Para qué sirven los parámetros no-tipo en la vida real? Para tamaños fijos (matrices de 3x3 en gráficos), para políticas de compilación (un `Buffer<T, 1024>` con capacidad fija sin `new`), y para valores constantes que cambian el comportamiento sin costo en ejecución. Es programación genérica de alto nivel: el mismo código, configurado en compilación, sin pagar nada en ejecución.

Plantillas vs herencia: cuándo usar cada una para código genérico

Ya tienes dos herramientas para "código que sirve para varios tipos": las plantillas (una función o clase que el compilador adapta a cada tipo) y la herencia con polimorfismo (una interfaz común que cada tipo implementa). Hacen cosas parecidas desde lejos, pero por dentro son opuestas, y elegir mal te cuesta rendimiento o flexibilidad. Esta es una de las decisiones de diseño más importantes en C++.

La diferencia fundamental es cuándo se decide todo. Con plantillas, todo se resuelve en compilación: el compilador genera una versión del código por cada tipo usado. Es rapidísimo en ejecución (no hay llamadas virtuales ni indirecciones) pero cada tipo genera su propio código (el binario crece) y los tipos deben conocerse en compilación. Con herencia y polimorfismo, la decisión ocurre en ejecución: un `vector<unique_ptr<Figura>>` puede mezclar tipos decididos mientras el programa corre (por ejemplo, figuras leídas de un archivo). Es flexible pero paga el costo de la llamada virtual y exige una jerarquía diseñada.

La guía práctica:

- Usa plantillas cuando los tipos se conocen en compilación, quieres máximo rendimiento y las operaciones son las mismas para todos los tipos. Ejemplos: contenedores (`vector<T>`), algoritmos (`sort`), funciones matemáticas genéricas. La STL entera está construida así.
- Usa herencia y polimorfismo cuando los tipos se deciden en ejecución, necesitas colecciones heterogéneas (mezclar tipos distintos en un contenedor) o el conjunto de tipos crece sin que quieras recompilar el código cliente. Ejemplos: plugins, enemigos de un juego cargados de un archivo, métodos de pago.
- Se pueden combinar: una clase plantilla puede heredar de una interfaz (por ejemplo, un repositorio genérico que además cumple un contrato), y una jerarquía puede usar plantillas internamente.

Mira el contraste con un ejemplo concreto: contar elementos mayores que un valor. Versión plantilla (resuelta en compilación) y versión polimórfica (resuelta en ejecución).

```
#include <iostream>
#include <vector>
#include <memory>

// VERSIÓN PLANTILLA: el tipo se conoce en compilación, cero costo extra
template <typename T>
int contarMayoresPlantilla(const std::vector<T>& v, const T& limite) {
    int n = 0;
    for (const auto& x : v) if (x > limite) n++;
    return n;
}
```

```

// VERSIÓN HERENCIA: interfaz común, tipos decididos en ejecución
class Medible {
public:
    virtual double valor() const = 0;
    virtual ~Medible() {}
};

class Temperatura : public Medible {
    double grados;
public:
    Temperatura(double g) : grados(g) {}
    double valor() const override { return grados; }
};

class Presion : public Medible {
    double pascales;
public:
    Presion(double p) : pascales(p) {}
    double valor() const override { return pascales; }
};

int contarMayoresPolimorfico(const std::vector<std::unique_ptr<Medible>>& v,
                             double limite) {
    int n = 0;
    for (const auto& m : v) if (m->valor() > limite) n++;
    return n;
}

int main() {
    // Plantilla: rápida, tipos fijos en compilación
    std::vector<int> enteros = {3, 8, 1, 9, 4};
    std::cout << "Plantilla: " << contarMayoresPlantilla(enteros, 4) << std::endl;

    // Polimorfismo: mezcla tipos distintos decididos al ejecutar
    std::vector<std::unique_ptr<Medible>> mediciones;
    mediciones.push_back(std::make_unique<Temperatura>(38.5));
    mediciones.push_back(std::make_unique<Presion>(101325));
    mediciones.push_back(std::make_unique<Temperatura>(41.2));
    std::cout << "Polimorfico: " << contarMayoresPolimorfico(mediciones, 40) <<
std::endl;
    return 0;
}

```

Nota que la versión polimórfica puede mezclar Temperaturas y Presiones en el mismo vector porque ambas "son" Medible, algo imposible con la plantilla (un vector<T> tiene un solo T). A

cambio, cada llamada a `valor()` pasa por la `vtable`. Ninguna es "mejor" en abstracto: la plantilla gana cuando el problema es homogéneo y conocido en compilación; la herencia gana cuando el problema es heterogéneo o abierto a tipos futuros.

Un error común es usar herencia donde bastaba una plantilla ("jerarquía de una sola derivada por tipo de dato") y usar plantillas donde se necesitaba polimorfismo en ejecución ("¿cómo meto tipos distintos en un `vector<T>?`"). Si te haces esas preguntas a tiempo, la decisión se toma sola.

`std::pair` y `std::tuple`: agrupar datos sin crear clases

No todo grupo de datos merece una clase con nombre. Cuando una función necesita devolver dos valores, o quieres guardar pares asociados en un contenedor, `std::pair` y `std::tuple` te evitan crear clases de un solo uso. Son plantillas de la biblioteca estándar (en `<utility>` y `<tuple>`) diseñadas exactamente para eso.

`std::pair` guarda dos valores (`first` y `second`), que pueden ser de tipos distintos. `std::tuple` generaliza la idea a cualquier cantidad de elementos, accedidos con `std::get<índice>`.

```

#include <iostream>
#include <string>
#include <utility> // pair, make_pair
#include <tuple>    // tuple, make_tuple, get

// Devolver dos valores sin crear una clase: el caso clásico de pair
std::pair<int, int> dividirConResiduo(int a, int b) {
    return {a / b, a % b}; // {cociente, residuo}
}

int main() {
    // pair: dos valores juntos
    auto division = dividirConResiduo(17, 5);
    std::cout << "Cociente: " << division.first
                << ", Residuo: " << division.second << std::endl;

    std::pair<std::string, double> producto = {"Teclado", 1200.0};
    std::cout << producto.first << ": $" << producto.second << std::endl;

    // tuple: tres o más valores juntos
    auto alumno = std::make_tuple("Ana", 20, 9.2);
    std::cout << "Nombre: " << std::get<0>(alumno)
                << ", Edad: " << std::get<1>(alumno)
                << ", Promedio: " << std::get<2>(alumno) << std::endl;

    // Desempaquetar con structured bindings (C++17): nombres directos
    auto [nombre, edad, promedio] = alumno;
    std::cout << nombre << " tiene " << edad << " años." << std::endl;

    // Modificar un elemento del tuple
    std::get<1>(alumno) = 21;
    std::cout << "Nueva edad: " << std::get<1>(alumno) << std::endl;
    return 0;
}

```

Los structured bindings (`auto [nombre, edad, promedio] = alumno;`) son la forma moderna de desempaquetar: le das nombres significativos a cada posición y el código se lee solo. Funcionan con pair, tuple y arreglos. Es una de las características de C++17 que más mejora la legibilidad del código cotidiano.

¿Cuándo usar cada uno? pair para exactamente dos valores relacionados (clave-valor, coordenadas, resultado + código de error). tuple para tres o más valores temporales. Y la regla de honestidad: si el grupo de datos tiene significado propio en tu dominio, aparece en varias

funciones y merece métodos, entonces sí crea una clase con nombre (como Alumno o Producto). pair y tuple son para datos de paso, no para modelar tu dominio. Un programa lleno de tuple<string, int, double, bool> es tan ilegible como uno lleno de clases de un solo uso es verboso: el equilibrio está en usar cada uno donde corresponde.

Operadores de comparación completos y cómo se relacionan

En las partes anteriores sobrecargaste ==, != y < para tus clases. Pero una clase bien diseñada suele necesitar la familia completa: ==, !=, <, >, <= y >=. La buena noticia es que no tienes que inventar seis lógicas distintas: con dos operadores bien elegidos puedes definir los otros cuatro en términos de ellos, igual que en el ejercicio de Fraccion definiste != como !(*this == o).

Las relaciones lógicas son estas. Si defines == y <, todo lo demás se deriva:

- a != b es !(a == b)
- a > b es b < a
- a <= b es !(b < a), es decir, !(a > b)
- a >= b es !(a < b)

Esto no es un truco de flojera: es una garantía de consistencia. Si cada operador tuviera su propia lógica, podrían contradecirse (que a < b y a > b sean ciertos a la vez, por ejemplo), y std::sort se comportaría de forma errática. Definiendo todo en términos de un par base, la familia completa es coherente por construcción.

Nota para el futuro: C++20 introdujo el operador "nave espacial" <=> que genera automáticamente todas las comparaciones, pero como este libro trabaja con C++17, aquí aprendes la forma manual, que además te enseña las relaciones lógicas que el operador automático esconde.

```
#include <iostream>
#include <vector>
#include <algorithm>

// Hora del día: ejemplo donde el orden total es natural y útil
class Hora {
private:
    int h, m, s;
    int aSegundos() const { return h * 3600 + m * 60 + s; }

public:
```

```

Hora(int hh, int mm, int ss) : h(hh), m(mm), s(ss) {}

// Los dos operadores base: toda la lógica vive aquí
bool operator==(const Hora& o) const {
    return aSegundos() == o.aSegundos();
}
bool operator<(const Hora& o) const {
    return aSegundos() < o.aSegundos();
}

// Los cuatro derivados: sin lógica nueva, solo combinaciones
bool operator!=(const Hora& o) const { return !(*this == o); }
bool operator>(const Hora& o) const { return o < *this; }
bool operator<=(const Hora& o) const { return !(o < *this); }
bool operator>=(const Hora& o) const { return !(*this < o); }

friend std::ostream& operator<<(std::ostream& os, const Hora& hh) {
    os << hh.h << ":";
    if (hh.m < 10) os << "0";
    os << hh.m << ":";
    if (hh.s < 10) os << "0";
    return os << hh.s;
}
};

int main() {
    Hora a(10, 30, 0), b(14, 15, 30), c(10, 30, 0);

    std::cout << "a == c? " << (a == c) << std::endl; // 1
    std::cout << "a != b? " << (a != b) << std::endl; // 1
    std::cout << "a < b? " << (a < b) << std::endl;    // 1
    std::cout << "b > a? " << (b > a) << std::endl;    // 1
    std::cout << "a <= c? " << (a <= c) << std::endl; // 1
    std::cout << "b >= a? " << (b >= a) << std::endl; // 1

    // Y como < existe y es consistente, sort funciona de inmediato
    std::vector<Hora> agenda = {b, a, c};
    std::sort(agenda.begin(), agenda.end());
    std::cout << "Ordenadas: ";
    for (const auto& hh : agenda) std::cout << hh << " ";
    std::cout << std::endl;
    return 0;
}

```

Fíjate en el diseño: toda la inteligencia está en `aSegundos()` y en los dos operadores base. Los otros cuatro son de una línea cada uno y no pueden contradecir a los base porque están definidos con ellos. Este es el patrón profesional para C++17: implementa `==` y `<` con cuidado, deriva el resto mecánicamente.

Una advertencia importante: este esquema supone un orden total coherente (como números o tiempo). Si tu clase no tiene un orden natural (¿qué significa que un Punto sea "menor" que otro?), no inventes uno: define solo `==` y `!=`, y deja fuera `<`, `>`, `<=`, `>=`. Un orden artificial confunde a quien use tu clase y puede romper algoritmos que asumen propiedades matemáticas del orden. En el ejercicio de Punto de la parte principal se usó la distancia al origen como criterio, lo cual es válido si lo documentas, pero no es un orden "natural": dos puntos distintos pueden empatar en distancia. Para ordenar con criterios alternativos existen los comparadores personalizados (lambdas o funtores pasados a `sort`), que verás más adelante.

Operadores de conversión: cuando tu objeto quiere ser otro tipo

Hay un operador poco conocido pero muy útil: el operador de conversión. Permite que tu objeto se convierta automáticamente (o explícitamente) a otro tipo. La sintaxis es peculiar: `operator Tipo() const`, sin tipo de retorno declarado (el tipo de retorno es el propio Tipo).

El ejemplo clásico es una clase que representa una cantidad convertible a número, o un tipo "booleano" para usarse en condiciones. Pero cuidado: las conversiones implícitas son un arma de doble filo, porque el compilador las aplica en silencio y pueden producir sorpresas. La regla moderna es marcarlas como `explicit` salvo que la conversión sea totalmente natural e inocua.

```
#include <iostream>
#include <string>
#include <cmath>

// Dinero: se puede ver como double (su cantidad), pero con cuidado
class Dinero {
private:
    double cantidad; // en pesos
    std::string moneda;
public:
    Dinero(double c, const std::string& m) : cantidad(c), moneda(m) {}

    // Conversión a double: explícita para evitar sorpresas silenciosas
    explicit operator double() const { return cantidad; }

    // Conversión a string: representación legible
```

```

explicit operator std::string() const {
    return std::to_string(cantidad) + " " + moneda;
}

// Conversión a bool: ¿tiene fondos? Esta sí suele ser implícita
// en tipos "manejadores", pero aquí la dejamos explícita por claridad
explicit operator bool() const { return cantidad > 0; }

friend std::ostream& operator<<(std::ostream& os, const Dinero& d) {
    return os << d.cantidad << " " << d.moneda;
}
};

// Fracción exacta que sí quiere comportarse como número
class Razon {
private:
    int num, den;
public:
    Razon(int n, int d) : num(n), den(d) {}
    // Implícita: una razón ES un número en casi todos los contextos
    operator double() const { return static_cast<double>(num) / den; }
};

int main() {
    Dinero pago(1500.50, "MXN");

    // Conversiones explícitas: claras y sin sorpresas
    double c = static_cast<double>(pago);
    std::string texto = static_cast<std::string>(pago);
    std::cout << "Cantidad: " << c << ", Texto: " << texto << std::endl;

    if (static_cast<bool>(pago)) {
        std::cout << "Hay fondos suficientes." << std::endl;
    }

    // Conversión implícita: la razón se usa directo como double
    Razon r(1, 2);
    double mitad = r; // funciona sin cast: operator double() implícito
    std::cout << "La mitad es " << mitad << std::endl;
    std::cout << "Doble de la razon: " << r * 2 << std::endl; // r se convierte solo
    return 0;
}

```

Compara las dos decisiones. Dinero usa conversiones explicit: nadie quiere que 1500.50 pesos se conviertan silenciosamente en un double y se sumen con otra cosa perdiendo la

moneda. El programador debe escribir `static_cast<double>(pago)` para dejar claro "sé lo que hago". Razon usa conversión implícita porque una razón es conceptualmente un número: que `r * 2` funcione directo es el comportamiento esperado, no una sorpresa.

El peligro clásico de lo implícito: si `Dinero` tuviera `operator double()` implícito y también `operator string()` implícito, una expresión como `pago + 1` sería ambigua (¿suma numérica o concatenación?) y el compilador se quejaría con un error difícil de leer. Con `explicit`, esas ambigüedades no existen porque la conversión solo ocurre donde tú la pides. Regla de oro: `explicit` por defecto; implícito solo cuando la conversión sea la naturaleza del tipo (como los tipos numéricos o los punteros inteligentes, cuyo `operator bool` implícito en contextos booleanos es intencional).

Funtores: objetos que se comportan como funciones

Una lambda es cómoda para criterios pequeños, pero tiene límites: no puede guardar estado complejo con nombre, no se puede reutilizar con un nombre claro en varios lugares y no puede tener múltiples sobrecargas. Para esos casos existe el functor (objeto función): una clase que sobrecarga `operator()`, de modo que sus objetos "se pueden llamar" como funciones.

```
#include <iostream>
#include <vector>
#include <algorithm>

// Functor con estado: cuenta cuántos elementos cumplen, con umbral configurable
class ContadorMayoresQue {
private:
    int umbral;
    int cuenta;
public:
    ContadorMayoresQue(int u) : umbral(u), cuenta(0) {}

    // operator(): el objeto "se llama" como función
    void operator()(int x) {
        if (x > umbral) cuenta++;
    }

    int resultado() const { return cuenta; }
};

// Functor reutilizable con nombre claro: criterio de aprobación
class EsAprobatorio {
private:
```

```

    double minimo;
public:
    EsAprobatorio(double m) : minimo(m) {}
    bool operator()(double calif) const {
        return calif >= minimo;
    }
};

int main() {
    std::vector<int> datos = {3, 8, 1, 9, 4, 12, 2};

    // El functor guarda estado entre llamadas: cuenta al recorrer
    ContadorMayoresQue contador(5);
    contador = std::for_each(datos.begin(), datos.end(), contador);
    std::cout << "Mayores que 5: " << contador.resultado() << std::endl;

    // El functor como predicado con nombre: se lee solo
    std::vector<double> califs = {7.5, 5.9, 8.2, 6.0, 4.8};
    int aprobados = std::count_if(califs.begin(), califs.end(), EsAprobatorio(6.0));
    std::cout << "Aprobados: " << aprobados << std::endl;

    // El mismo functor, otro umbral, sin reescribir nada
    int destacados = std::count_if(califs.begin(), califs.end(), EsAprobatorio(8.0));
    std::cout << "Destacados: " << destacados << std::endl;
    return 0;
}

```

Tres ventajas sobre la lambda. Una: estado con nombre y métodos. ContadorMayoresQue acumula la cuenta dentro del objeto y la expone con resultado(); una lambda puede capturar por referencia, pero el functor lo hace explícito y reutilizable. (Detalle técnico: std::for_each devuelve el functor, posiblemente copiado, por eso reasignamos contador con el valor de retorno.) Dos: reutilización. EsAprobatorio(6.0) se usa en dos conteos con parámetros distintos; con lambdas habrías duplicado el cuerpo o capturado variables sueltas. Tres: el functor puede tener varias sobrecargas de operator() para distintos tipos, constructores con validación y hasta herencia: es una clase completa, con todo lo que eso implica.

¿Cuándo elegir cada uno? Lambda para criterios de una línea usados una vez (el 80% de los casos). Functor cuando el criterio tiene estado significativo, se reutiliza en varios lugares, necesita un nombre que documente la intención o requiere lógica compleja que merece su propia clase testeable. Y hay un dato histórico curioso: antes de C++11 no había lambdas, así que todo se hacía con funtores; las lambdas modernas son, por dentro, funtores generados por el compilador. Cuando escribes un functor a mano, estás haciendo explícito lo que la lambda hace en secreto.

std::function: guardar "cualquier cosa llamable"

Las lambdas y los funtores son geniales, pero cada uno tiene un tipo distinto y desconocido: no puedes guardar una lambda en una variable de tipo fijo ni mezclar lambdas y funtores en un mismo vector. std::function (en <functional>) resuelve eso: es un contenedor polimórfico para cualquier cosa que se pueda llamar con una firma dada. Piensa en él como "el vector<unique_ptr<Interfaz>> de las funciones": borra el tipo concreto y conserva solo la firma.

La sintaxis es std::function<TipoRetorno(TipoParam1, TipoParam2, ...)>. Dentro puedes guardar funciones normales, lambdas, funtores, e incluso métodos asociados a objetos. Esto permite patrones muy potentes: menús de comandos, sistemas de eventos (callbacks), y estrategias seleccionadas en ejecución sin jerarquías de clases.

```
#include <iostream>
#include <string>
#include <vector>
#include <functional>

// Una función normal: también cabe en std::function
int sumar(int a, int b) { return a + b; }

class Calculadora {
public:
    // Guarda la operación actual: puede ser función, lambda o functor
    std::function<int(int, int)> operacion;

    void setOperacion(std::function<int(int, int)> op) {
        operacion = op;
    }

    int calcular(int a, int b) const {
        return operacion(a, b);
    }
};

// Functor que también puede guardarse
struct Multiplicar {
    int operator()(int a, int b) const { return a * b; }
};

int main() {
    Calculadora calc;

    calc.setOperacion(sumar); // función normal
```

```

std::cout << "Suma: " << calc.calcular(6, 7) << std::endl;

calc.setOperacion([](int a, int b) { return a - b; }); // lambda
std::cout << "Resta: " << calc.calcular(6, 7) << std::endl;

calc.setOperacion(Multiplicar{}); // functor
std::cout << "Multiplicación: " << calc.calcular(6, 7) << std::endl;

// Un menú de comandos: nombre + acción, todo mezclado
std::vector<std::pair<std::string, std::function<void()>>> menu;
menu.push_back({"saludar", []() { std::cout << "¡Hola!" << std::endl; }});
menu.push_back({"despedir", []() { std::cout << "¡Adiós!" << std::endl; }});

for (const auto& [nombre, accion] : menu) {
    std::cout << "[" << nombre << "]" ";
    accion();
}
return 0;
}

```

Observa el ejemplo del menú: es el patrón Estrategia de la unidad 7, pero sin clases ni herencia. Cada comando es "algo que se puede llamar sin parámetros", y `std::function` los unifica. Para comportamientos simples e intercambiables en ejecución, `std::function` suele ser más ligero que diseñar una jerarquía de estrategias. Para comportamientos complejos con estado rico y varias operaciones relacionadas, la jerarquía de la unidad 7 sigue siendo superior. Otra vez: la herramienta depende del problema.

Dos advertencias honestas. Una: `std::function` tiene un pequeño costo (borrado de tipo con indirección, parecido a una llamada virtual). No lo uses dentro del bucle más caliente de un motor de videojuegos; para callbacks, menús y eventos, el costo es irrelevante. Dos: una `std::function` vacía (sin nada asignado) lanza una excepción si la llamas. Si puede estar vacía, verifica con `if (operacion)` antes de llamarla: `std::function` se puede evaluar como booleano, igual que un puntero.

Más algoritmos: copy, fill, replace y unique

En la gira de algoritmos ya conoces `sort`, `find`, `count_if`, `accumulate`, `transform`, `reverse` y el modismo `erase-remove`. Ahora agregamos cuatro que completan tu vocabulario diario: `std::copy`, `std::fill`, `std::replace` y `std::unique`. Todos viven en `<algorithm>` y todos trabajan con iteradores, así que sirven para vectores, arreglos y demás contenedores.

`std::copy(v.begin(), v.end(), destino)` copia un rango a otro lugar. El destino debe tener espacio suficiente: es tu responsabilidad. Combinado con `std::back_inserter` (en `<iterator>`), el destino crece solo con `push_back`, lo que elimina el problema del tamaño.

`std::fill(inicio, fin, valor)` pone el mismo valor en todo un rango: inicializar, reiniciar tableros, poner ceros. Su primo `std::fill_n(inicio, n, valor)` llena `n` elementos desde un punto.

`std::replace(inicio, fin, viejo, nuevo)` sustituye todas las apariciones de un valor por otro, en el lugar, sin cambiar el tamaño. Con `std::replace_if` puedes usar un predicado (lambda o functor) en vez de un valor fijo.

`std::unique(inicio, fin)` elimina duplicados consecutivos: reordena el rango dejando un representante de cada grupo de iguales al frente y devuelve el iterador donde empiezan los "sobrantes", que luego borras con `erase`. Ojo: solo elimina duplicados consecutivos, así que normalmente se combina con `sort` previo si quieres duplicados globales eliminados.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator> // back_inserter
#include <string>

void mostrar(const std::vector<int>& v, const std::string& etiqueta) {
    std::cout << etiqueta << ": ";
    for (int x : v) std::cout << x << " ";
    std::cout << std::endl;
}

int main() {
    std::vector<int> v = {5, 2, 8, 2, 1, 5, 9, 2};
    mostrar(v, "Original");

    // copy con back_inserter: el destino crece solo
    std::vector<int> copia;
    std::copy(v.begin(), v.end(), std::back_inserter(copia));
    mostrar(copia, "Copia");

    // fill: reiniciar un rango a un valor
    std::fill(copia.begin(), copia.begin() + 3, 0);
    mostrar(copia, "Primeros 3 a cero");

    // replace: cambiar todas las apariciones de un valor
    std::replace(v.begin(), v.end(), 2, 20);
    mostrar(v, "2 -> 20");
}
```

```

// replace_if con lambda: cambiar los que cumplan condición
std::replace_if(v.begin(), v.end(), [](int x) { return x > 10; }, 99);
mostrar(v, "Mayores a 10 -> 99");

// unique: quitar duplicados consecutivos (tras ordenar, quita todos)
std::vector<int> d = {5, 2, 8, 2, 1, 5, 9, 2};
std::sort(d.begin(), d.end());
mostrar(d, "Ordenado");
auto nuevoFin = std::unique(d.begin(), d.end());
d.erase(nuevoFin, d.end()); // el erase del modismo erase-remove/unique
mostrar(d, "Sin duplicados");

// fill_n: llenar n elementos desde una posición
std::vector<int> tablero(10);
std::fill_n(tablero.begin(), 10, -1);
mostrar(tablero, "Tablero inicializado");
return 0;
}

```

Fíjate en los patrones que se repiten. Uno: varios algoritmos devuelven un iterador que marca "hasta dónde llegó lo válido" (`unique`, `remove_if`), y el `erase` posterior hace la limpieza real. Es el mismo modismo `erase-remove` de la parte anterior, aplicado a duplicados. Dos: las versiones `_if` aceptan predicados, lo que conecta estos algoritmos con lambdas y funtores: el algoritmo pone la mecánica (recorrer, comparar, mover) y tú pones el criterio. Tres: `back_inserter` convierte un destino de tamaño fijo en uno que crece, eliminando una fuente clásica de errores (escribir fuera del vector destino).

Estos cuatro, sumados a los de la gira anterior, cubren la inmensa mayoría de las manipulaciones de colecciones en programas reales. La próxima vez que vayas a escribir un ciclo `for` para copiar, rellenar, sustituir o deduplicar, detente y pregunta si ya existe el algoritmo: casi siempre sí, y el algoritmo comunica la intención mejor que el ciclo manual ("quita duplicados" se entiende al instante; un ciclo con índices hay que leerlo con cuidado).

Catálogo de errores comunes de esta expansión

Error 1: orden de parámetros con defecto en plantillas. Los parámetros con valor por defecto van al final: `template <typename T1, typename T2 = T1>` es válido, pero `template <typename T1 = int, typename T2>` no compila. Igual que en funciones.

Error 2: parámetro no-tipo que no es constante en compilación. `ArregloFijo<int, n>` donde `n` es una variable normal no compila: `N` debe conocerse al compilar (literal, `const` o `constexpr`). Si el tamaño se decide en ejecución, usa `vector`.

Error 3: definir solo algunos operadores de comparación con lógica independiente. Si tu `<` dice una cosa y tu `>` dice otra, `sort` puede entrar en comportamiento indefinido (ciclos infinitos o resultados absurdos). Deriva todos de `==` y `<` como se mostró, y la coherencia es automática.

Error 4: conversión implícita que crea ambigüedad. Dos operadores de conversión implícitos (a `double` y a `string`) hacen que expresiones como `obj + 1` no compilen por ambigüedad, con errores largos. Marca las conversiones como `explicit` salvo que sean la naturaleza del tipo.

Error 5: olvidar que `std::function` vacía lanza al llamarse. Una `std::function` declarada pero sin asignar no es nula en silencio: llamarla lanza `std::bad_function_call`. Verifica con `if (f)` cuando pueda estar vacía.

Error 6: `unique` sin ordenar cuando quieres eliminar todos los duplicados. `std::unique` solo compacta duplicados consecutivos: en `{1, 2, 1}` no elimina nada porque los unos no están juntos. Para deduplicación global, `sort` antes de `unique`.

Error 7: `copy` a un destino sin espacio. `std::copy` no agranda el destino: si copias 8 elementos a un vector vacío sin `back_inserter`, escribes fuera de memoria (comportamiento indefinido, quizá un fallo mucho después). Usa `back_inserter` o asegura el tamaño con `resize` primero.

Error 8: usar `tuple` para modelar el dominio. Un `tuple<string, int, double>` que viaja por diez funciones es ilegible: nadie recuerda qué significa cada posición. Si el grupo tiene significado propio y vida larga, crea una clase con nombres de verdad. `pair` y `tuple` son para datos de paso.

Error 9: elegir plantilla cuando necesitabas polimorfismo en ejecución (o al revés). Si los tipos llegan de un archivo o de plugins cargados dinámicamente, la plantilla no sirve: necesitas herencia. Si todo se conoce en compilación y quieres velocidad, la jerarquía con virtuales es sobreingeniería.

Error 10: functor que modifica estado usado con algoritmos que copian el functor. Algunos algoritmos copian el predicado internamente; si tu functor cuenta cosas en su estado, la copia puede perderse. `std::for_each` devuelve el functor (posiblemente el usado), así que reasigna el resultado como en el ejemplo. Para conteos simples, además, `count_if` suele ser más directo que un functor con estado.

Práctica de la expansión

Ejercicio 1

Escribe una clase plantilla `Tabla<T, int FILAS, int COLUMNAS>` (dos parámetros no-tipo) que guarde una matriz de tamaño fijo, con métodos `set(f, c, valor)`, `get(f, c)` y un valor por defecto para el tipo de un segundo parámetro de etiqueta... no: manténlo simple. Requisitos: constructor que inicializa todo a `T{}`, `set/get` con verificación de rangos (usa `assert` o ignora índices inválidos), y pruébala con `Tabla<int, 3, 3>` y `Tabla<double, 2, 4>`.

Solución:

```

#include <iostream>

template <typename T, int FILAS, int COLUMNAS>
class Tabla {
private:
    T datos[FILAS][COLUMNAS];
public:
    Tabla() {
        for (int f = 0; f < FILAS; f++)
            for (int c = 0; c < COLUMNAS; c++)
                datos[f][c] = T{};
    }
    void set(int f, int c, const T& valor) {
        if (f >= 0 && f < FILAS && c >= 0 && c < COLUMNAS)
            datos[f][c] = valor;
    }
    T get(int f, int c) const {
        if (f >= 0 && f < FILAS && c >= 0 && c < COLUMNAS)
            return datos[f][c];
        return T{};
    }
    void mostrar() const {
        for (int f = 0; f < FILAS; f++) {
            for (int c = 0; c < COLUMNAS; c++)
                std::cout << datos[f][c] << "\t";
            std::cout << std::endl;
        }
    }
};

int main() {
    Tabla<int, 3, 3> t;
    t.set(0, 0, 1); t.set(1, 1, 5); t.set(2, 2, 9);
    t.mostrar();
    std::cout << "---" << std::endl;
    Tabla<double, 2, 4> d;
    d.set(0, 3, 3.14);
    d.mostrar();
    return 0;
}

```

Explicación: FILAS y COLUMNAS son parámetros no-tipo: el tamaño se fija en compilación y no hay memoria dinámica. T{} es el "valor cero" del tipo (0 para int, 0.0 para double, cadena vacía para string). La verificación de rangos en set/get evita accesos inválidos. Así están implementadas las matrices pequeñas de bibliotecas gráficas y numéricas.

Ejercicio 2

Completa la familia de comparaciones para una clase Producto2 (nombre, precio): implementa == y < por precio (a igual precio, desempatan por nombre para que el orden sea total), deriva !=, >, <=, >=, y ordena un vector<Producto2> con sort. Luego usa std::max_element para encontrar el más caro.

Solución:

```

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

class Producto2 {
private:
    std::string nombre;
    double precio;
public:
    Producto2(const std::string& n, double p) : nombre(n), precio(p) {}

    bool operator==(const Producto2& o) const {
        return precio == o.precio && nombre == o.nombre;
    }
    bool operator<(const Producto2& o) const {
        if (precio != o.precio) return precio < o.precio;
        return nombre < o.nombre; // desempate: orden total
    }
    bool operator!=(const Producto2& o) const { return !(*this == o); }
    bool operator>(const Producto2& o) const { return o < *this; }
    bool operator<=(const Producto2& o) const { return !(o < *this); }
    bool operator>=(const Producto2& o) const { return !(*this < o); }

    friend std::ostream& operator<<(std::ostream& os, const Producto2& p) {
        return os << p.nombre << " ($" << p.precio << ")";
    }
};

int main() {
    std::vector<Producto2> tienda = {
        {"Teclado", 1200}, {"Mouse", 350}, {"Monitor", 1200}, {"USB", 150}
    };
    std::sort(tienda.begin(), tienda.end());
    for (const auto& p : tienda) std::cout << p << std::endl;

    auto caro = std::max_element(tienda.begin(), tienda.end());
    std::cout << "Mas caro: " << *caro << std::endl;
    return 0;
}

```

Explicación: el desempate por nombre garantiza un orden total: dos productos distintos nunca son "equivalentes" para el orden, lo que mantiene a sort feliz y predecible. Los cuatro operadores derivados no agregan lógica nueva, solo combinan `==` y `<`. `max_element` usa tu `<` internamente: otra integración gratis con la STL.

Ejercicio 3

Crea un functor `RangoValido(min, max)` con `operator()(double) const` que diga si un valor está en el rango. Úsalo con `count_if` para contar lecturas de sensores dentro de rango, y con `replace_if` para marcar las fuera de rango con `-1` en el vector. Compara en un comentario cuándo preferirías una lambda.

Solución:


```

#include <iostream>
#include <vector>
#include <algorithm>

class RangoValido {
private:
    double minimo, maximo;
public:
    RangoValido(double a, double b) : minimo(a), maximo(b) {}
    bool operator()(double x) const {
        return x >= minimo && x <= maximo;
    }
};

// Functor inverso reutilizando el primero (composición de funtores)
class FueraDeRango {
private:
    RangoValido rango;
public:
    FueraDeRango(double a, double b) : rango(a, b) {}
    bool operator()(double x) const { return !rango(x); }
};

int main() {
    std::vector<double> lecturas = {36.5, 41.2, 37.0, 35.8, 42.0, 36.9};

    int validas = std::count_if(lecturas.begin(), lecturas.end(),
                                RangoValido(36.0, 37.5));
    std::cout << "Lecturas validas: " << validas << std::endl;

    std::replace_if(lecturas.begin(), lecturas.end(),
                    FueraDeRango(36.0, 37.5), -1.0);
    std::cout << "Marcadas: ";
    for (double x : lecturas) std::cout << x << " ";
    std::cout << std::endl;
    return 0;
}

// Comentario: preferiría una lambda si el criterio se usa una sola vez y es
// de una línea, como [](double x){ return x >= 36.0 && x <= 37.5; }.
// El functor gana cuando el criterio se reutiliza (aquí en count_if y en
// replace_if con su inverso), necesita nombre que documente la intención,
// o su lógica merece probarse por separado.

```

Explicación: el functor encapsula el concepto "rango válido" con sus límites como estado; FueraDeRango lo reutiliza por composición en lugar de duplicar la condición. Usar el mismo criterio en dos algoritmos sin repetir código es exactamente el caso donde el functor supera a la lambda.

Ejercicio 4

Usa `std::function` para un sistema de notificaciones multicanal: un vector de pares (nombre del canal, `std::function<void(const std::string&>>`) con tres canales implementados de formas distintas (una función normal para consola, una lambda para "email", un functor para "SMS"). Una función `difundir` recorre el vector y envía el mensaje por todos los canales.

Solución:

```

#include <iostream>
#include <string>
#include <vector>
#include <utility>
#include <functional>

void canalConsola(const std::string& msg) {
    std::cout << "[Consola] " << msg << std::endl;
}

struct CanalsMS {
    std::string numero;
    CanalsMS(const std::string& n) : numero(n) {}
    void operator()(const std::string& msg) const {
        std::cout << "[SMS al " << numero << "]" << msg << std::endl;
    }
};

void difundir(
    const std::vector<std::pair<std::string, std::function<void(const
std::string&)>>>& canales,
    const std::string& mensaje) {
    for (const auto& [nombre, enviar] : canales) {
        std::cout << "Canal " << nombre << ": ";
        enviar(mensaje);
    }
}

int main() {
    std::vector<std::pair<std::string, std::function<void(const std::string&)>>>
canales;
    canales.push_back({"consola", canalConsola}); // función normal
    canales.push_back({"email", [](const std::string& m) { // lambda con estado
        std::cout << "[Email a admin@tienda.com] " << m << std::endl;
    }});
    canales.push_back({"sms", CanalsMS("555-1234")}); // functor con estado

    difundir(canales, "Tu pedido llegó");
    return 0;
}

```

Explicación: `std::function` unifica los tres tipos de "llamables" bajo una sola firma, así el vector los mezcla sin jerarquías de clases. `difundir` no sabe ni le importa cómo está implementado cada canal: es el patrón Estrategia sin herencia, ideal cuando cada comportamiento cabe en

una función. Si un canal necesitara varias operaciones relacionadas (enviar, reintentar, registrar), ahí sí convendría la jerarquía de interfaces de la unidad 7.

Ejercicio 5

Procesa una lista de calificaciones con los algoritmos nuevos: dado `vector<int> califs` con valores que incluyen duplicados y un -1 inválido, 1) reemplaza los -1 por 0 con `replace`, 2) ordena, 3) elimina duplicados con `unique + erase`, 4) haz una copia de seguridad con `copy + back_inserter` antes de modificar, y 5) crea un `vector<string>` del mismo tamaño lleno de "pendiente" con `fill`. Muestra cada paso.

Solución:

```

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>

void mostrar(const std::vector<int>& v, const std::string& e) {
    std::cout << e << ": ";
    for (int x : v) std::cout << x << " ";
    std::cout << std::endl;
}

int main() {
    std::vector<int> califs = {8, -1, 6, 8, 10, 6, -1, 7};

    // 4) copia de seguridad ANTES de modificar
    std::vector<int> respaldo;
    std::copy(califs.begin(), califs.end(), std::back_inserter(respaldo));

    // 1) -1 inválido -> 0
    std::replace(califs.begin(), califs.end(), -1, 0);
    mostrar(califs, "Sin invalidos");

    // 2) y 3) ordenar + quitar duplicados
    std::sort(califs.begin(), califs.end());
    califs.erase(std::unique(califs.begin(), califs.end()), califs.end());
    mostrar(califs, "Ordenadas sin duplicar");

    // 5) vector de estados con fill
    std::vector<std::string> estados(califs.size());
    std::fill(estados.begin(), estados.end(), "pendiente");
    std::cout << "Estados: ";
    for (const auto& s : estados) std::cout << s << " ";
    std::cout << std::endl;

    mostrar(respaldo, "Respaldo intacto");
    return 0;
}

```

Explicación: el orden de los pasos importa: la copia de seguridad se hace antes de modificar (si la hicieras después, respaldarías datos ya cambiados). unique necesita el sort previo porque solo compacta duplicados consecutivos. fill inicializa el vector de estados en una línea, sin ciclo. Cinco algoritmos, cero ciclos manuales: este es el estilo declarativo que la STL premia.

Guía de decisión: lambda, functor o std::function

Ya tienes tres formas de representar "algo llamable": lambdas, funtores y std::function. Se parecen, pero cada una brilla en un escenario distinto. Aquí va el criterio para no dudar.

Usa una lambda cuando el criterio es corto, se usa en un solo lugar y no necesita nombre. Es el caso más común: predicados de count_if, transformaciones de transform, comparadores de sort escritos al vuelo. La lambda vive junto al algoritmo que la usa y se lee en contexto. Si la lambda crece más de cinco líneas o necesita capturas complicadas, es señal de que merece ser un functor con nombre.

Usa un functor cuando el comportamiento tiene identidad propia: se reutiliza en varios lugares, guarda estado significativo con invariantes, merece pruebas unitarias independientes o necesita varias sobrecargas de operator(). El functor es una clase de verdad: puedes documentarlo, validarlo en el constructor y componerlo con otros funtores, como hiciste con FueraDeRango reutilizando RangoValido. Piensa en el functor como "una lambda que creció y se independizó".

Usa std::function cuando necesitas almacenar o pasar llamables de tipos distintos bajo una misma firma: un menú de comandos, una lista de callbacks de eventos, una estrategia elegida en ejecución entre función, lambda y functor. std::function es el pegamento heterogéneo: borra el tipo concreto y conserva solo la firma. Su costo (una indirección parecida a la llamada virtual) es irrelevante para callbacks y eventos, pero no lo pongas en el bucle más caliente de tu programa.

Y hay una cuarta opción que no debes olvidar: la función normal con nombre. Para lógica reutilizable sin estado, una función libre sigue siendo lo más simple y lo más rápido (el compilador puede incluso integrarla). No todo necesita ser un objeto llamable: la sofisticación debe ganarse su lugar.

Un ejemplo que combina las cuatro en su lugar correcto: un programa que procesa ventas usa una función normal para calcular el IVA (lógica fija sin estado), una lambda para filtrar ventas mayores a mil (criterio de un solo uso), un functor ValidadorDeCupones para verificar cupones (estado: lista de cupones válidos; reutilizado en varios puntos y con pruebas propias), y un vector de std::function para los canales de notificación (mezcla de función, lambda y functor elegidos en ejecución). Cada herramienta donde rinde más: esa es la marca del programador que domina la programación genérica en lugar de aplicar una sola receta a todo.

Resumen de la expansión

- Plantillas multiparámetro: varios typename, parámetros no-tipo (constantes en compilación) y valores por defecto. Más expresividad sin costo en ejecución.
- Plantillas vs herencia: plantillas cuando los tipos se conocen en compilación y quieres velocidad; herencia cuando los tipos se deciden en ejecución o necesitas colecciones heterogéneas. Se pueden combinar.
- `std::pair` y `std::tuple` agrupan datos de paso sin crear clases; los structured bindings de C++17 los desempaquetan con nombres claros. Para modelar el dominio, crea clases con nombre.
- Comparaciones completas: implementa `==` y `<` con cuidado y deriva `!=`, `>`, `<=`, `>=` mecánicamente. Coherencia garantizada y `sort` funciona gratis. Sin orden natural, no inventes uno.
- Operadores de conversión: `operator Tipo()` permite usar tu objeto como otro tipo. `explicit` por defecto; implícito solo si la conversión es la naturaleza del tipo.
- Functores: clases con `operator()` que se llaman como funciones, con estado, nombre y reutilización. La alternativa sería a la lambda para criterios complejos o repetidos.
- `std::function`: borra el tipo de cualquier llamable y conserva la firma. Menús de comandos, callbacks y estrategias ligeras sin jerarquías. Verifica que no esté vacía antes de llamar.
- Nuevos algoritmos: `copy` (+ `back_inserter`), `fill` / `fill_n`, `replace` / `replace_if`, `unique` (+ erase tras `sort`). Cubren copiar, inicializar, sustituir y deduplicar sin ciclos manuales.
- Con esto cierras la unidad 8: ya dominas la programación genérica de C++, desde la sobrecarga de operadores hasta la STL avanzada. En la unidad 9 aplicarás todo con mapas, excepciones y archivos.

Un consejo final para llevar: la programación genérica premia al que piensa en términos de conceptos ("¿qué operaciones necesita este código?") en lugar de tipos concretos ("¿es `int` o `double`?"). Cuando escribas una plantilla, pregúntate qué exige tu código del tipo: ¿compararse con `<`? ¿copiarse? ¿convertirse a texto? Esas exigencias son tu contrato implícito, y documentarlas en un comentario ("requiere `T` copiable y comparable con `<`") convierte tu plantilla en un componente profesional que otros pueden usar con confianza. Esa mentalidad, más que cualquier algoritmo memorizado, es lo que distingue al programador de C++ moderno.

UNIDAD 9: Herramientas profesionales

Ya dominas el lenguaje y la orientación a objetos. Esta unidad te da las herramientas que separan un programa de juguete de un programa profesional: más contenedores de la STL para organizar datos (mapas y conjuntos), excepciones para manejar errores con elegancia y archivos para que tus datos sobrevivan cuando el programa termina.

Piensa en un sistema real: una tienda guarda productos (mapa por código), evita duplicados (conjunto), maneja errores sin caerse (excepciones) y guarda todo en disco (archivos). Al terminar esta unidad podrás construir exactamente eso.

Al terminar sabrás: usar `std::map` y `std::set`, lanzar y atrapar excepciones, leer y escribir archivos de texto con `fstream`, y combinar todo en programas robustos.

Explicación detallada

Clase 25: Mapas y conjuntos de la STL

El vector guarda elementos por posición. Pero muchas veces quieres buscar por clave: el teléfono de "Ana", el precio del producto "P123". Para eso está `std::map`: un diccionario que asocia claves únicas con valores.

```
#include <map>
#include <string>

std::map<std::string, std::string> telefonos;
telefonos["Ana"] = "555-1234";
telefonos["Luis"] = "555-5678";

std::cout << telefonos["Ana"] << std::endl; // 555-1234
```

El operador `[]` busca o crea: si la clave no existe, la crea con un valor por defecto. Para solo buscar sin crear, usa `find`:

```
auto it = telefonos.find("Pedro");
if (it != telefonos.end()) {
    std::cout << it->second << std::endl;
}
```


it->first es la clave, it->second el valor. Recorrer un mapa con for de rango te da pares ordenados por clave (los mapas se auto-ordenan).

std::set guarda elementos únicos ordenados, sin valores asociados: perfecto para evitar duplicados o para membresía rápida.

```
#include <set>
std::set<std::string> visitados;
visitados.insert("inicio");
visitados.insert("inicio"); // duplicado: se ignora
std::cout << visitados.size() << std::endl; // 1
if (visitados.count("inicio")) { /* ya visitado */ }
```

¿Cuándo usar cada contenedor? vector: orden importa y acceso por posición. map: búsqueda por clave. set: unicidad y pertenencia. Esta tríada cubre el 95% de las necesidades de estructuración de datos.

Clase 26: Excepciones

Hasta ahora, los errores se manejaban con códigos de retorno o validaciones manuales. Las excepciones ofrecen algo mejor: separar el código normal del manejo de errores. Cuando algo sale mal, "lanzas" una excepción con throw; el código que sabe qué hacer la "atrapa" con try/catch.

```

#include <iostream>
#include <stdexcept>

double dividir(double a, double b) {
    if (b == 0) {
        throw std::runtime_error("Division entre cero");
    }
    return a / b;
}

int main() {
    try {
        std::cout << dividir(10, 2) << std::endl;
        std::cout << dividir(10, 0) << std::endl; // lanza
        std::cout << "Esto no se ejecuta" << std::endl;
    } catch (const std::runtime_error& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
    std::cout << "El programa sigue vivo." << std::endl;
    return 0;
}

```

El flujo: dentro del try, al lanzar la excepción se abandona todo y se salta al catch correspondiente. El programa no se cae: se recupera y sigue. e.what() trae el mensaje.

La biblioteca <stdexcept> trae excepciones estándar: std::runtime_error (errores en ejecución), std::invalid_argument (argumento inválido), std::out_of_range (índice fuera de rango, la que lanza vector::at). Puedes crear las tuyas heredando de std::exception.

Buenas prácticas: lanza excepciones para errores excepcionales (archivo no encontrado, sin memoria), no para control de flujo normal. Atrapa por referencia constante (const std::exception&). No tragues excepciones en silencio: al menos registra el error. Y recuerda RAI: cuando una excepción atraviesa una función, los destructores de los objetos locales se ejecutan (los vectores y strings se limpian solos). Por eso el C++ moderno y las excepciones se llevan tan bien.

Clase 27: Archivos con fstream

Los datos en memoria mueren cuando el programa termina. Los archivos les dan persistencia. La biblioteca <fstream> trae ofstream (escritura), ifstream (lectura) y fstream (ambas).

Escribir:

```
#include <fstream>

std::ofstream archivo("datos.txt");
if (archivo.is_open()) {
    archivo << "Hola, archivo." << std::endl;
    archivo << 42 << " " << 3.14 << std::endl;
    archivo.close();
}
```

El operador << funciona igual que con cout: ¡los archivos son flujos! Leer:

```
std::ifstream entrada("datos.txt");
if (entrada.is_open()) {
    std::string linea;
    while (std::getline(entrada, linea)) {
        std::cout << linea << std::endl;
    }
    entrada.close();
}
```

El patrón while (getline(...)) lee hasta el fin del archivo: getline devuelve el flujo, que se evalúa como falso al llegar al final. Es el patrón estándar de lectura.

Verifica siempre is_open(): si el archivo no existe o no tienes permiso, el flujo falla silenciosamente y tus lecturas darán basura. Para errores más estructurados, combina con excepciones: lanza si no se pudo abrir.

Modos de apertura: por defecto ofstream trunca (borra) el archivo. Para agregar al final usa std::ios::app: std::ofstream log("app.log", std::ios::app);.

Un formato simple y poderoso para guardar datos es el CSV (valores separados por comas) o líneas con formato fijo. Para proyectos serios existen bibliotecas JSON/XML, pero con texto plano y string llegas lejos.

Ejemplos paso a paso

Ejemplo 1: Agenda telefónica con map

```
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<std::string, std::string> agenda;
    agenda["Ana"] = "555-1234";
    agenda["Luis"] = "555-5678";
    agenda["Maria"] = "555-9012";

    std::string nombre;
    std::cout << "Buscar: ";
    std::getline(std::cin, nombre);

    auto it = agenda.find(nombre);
    if (it != agenda.end()) {
        std::cout << it->first << ": " << it->second << std::endl;
    } else {
        std::cout << "No encontrado. Agregando..." << std::endl;
        std::string tel;
        std::cout << "Telefono: ";
        std::getline(std::cin, tel);
        agenda[nombre] = tel;
    }

    std::cout << "\nAgenda completa:" << std::endl;
    for (const auto& par : agenda) {
        std::cout << par.first << " -> " << par.second << std::endl;
    }
    return 0;
}
```

El mapa se muestra ordenado alfabéticamente sin esfuerzo. `find` evita crear entradas accidentales. Este patrón de "buscar o agregar" es omnipresente en software real.

Ejemplo 2: Calculadora robusta con excepciones

```

#include <iostream>
#include <stdexcept>

double dividir(double a, double b) {
    if (b == 0) throw std::runtime_error("No se puede dividir entre cero");
    return a / b;
}

int leerNumero() {
    int n;
    std::cin >> n;
    if (std::cin.fail()) {
        throw std::runtime_error("Entrada invalida: se esperaba un numero");
    }
    return n;
}

int main() {
    try {
        std::cout << "Dividendo: ";
        int a = leerNumero();
        std::cout << "Divisor: ";
        int b = leerNumero();
        std::cout << "Resultado: " << dividir(a, b) << std::endl;
    } catch (const std::runtime_error& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
    std::cout << "Programa terminado correctamente." << std::endl;
    return 0;
}

```

Dos funciones que lanzan, un solo try/catch que las cubre a ambas. El código normal queda limpio, sin if de error entre cada línea. Si el usuario escribe letras donde va un número, cin.fail() lo detecta y lanzamos una excepción clara.

Ejemplo 3: Registro de alumnos en archivo

```

#include <iostream>
#include <fstream>
#include <string>
#include <vector>

struct Alumno {
    std::string nombre;

```

```

    double promedio;
};

void guardar(const std::vector<Alumno>& alumnos, const std::string& archivo) {
    std::ofstream out(archivo);
    if (!out.is_open()) throw std::runtime_error("No se pudo abrir el archivo");
    for (const auto& a : alumnos) {
        out << a.nombre << "," << a.promedio << "\n";
    }
}

std::vector<Alumno> cargar(const std::string& archivo) {
    std::vector<Alumno> alumnos;
    std::ifstream in(archivo);
    if (!in.is_open()) return alumnos; // archivo inexistente: lista vacía
    std::string linea;
    while (std::getline(in, linea)) {
        size_t pos = linea.find(',');
        if (pos != std::string::npos) {
            Alumno a;
            a.nombre = linea.substr(0, pos);
            a.promedio = std::stod(linea.substr(pos + 1));
            alumnos.push_back(a);
        }
    }
    return alumnos;
}

int main() {
    try {
        std::vector<Alumno> alumnos = cargar("alumnos.txt");
        std::cout << "Alumnos registrados: " << alumnos.size() << std::endl;

        Alumno nuevo;
        std::cout << "Nombre: ";
        std::getline(std::cin, nuevo.nombre);
        std::cout << "Promedio: ";
        std::cin >> nuevo.promedio;
        alumnos.push_back(nuevo);

        guardar(alumnos, "alumnos.txt");
        std::cout << "Guardado." << std::endl;
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
}

```

```
    return 0;
}
```

Sistema completo de persistencia: carga al inicio, guarda al final, formato CSV simple. stod convierte string a double. Cada ejecución acumula alumnos. Con esto ya puedes hacer programas que "recuerdan".

Práctica

Ejercicio 1

Crea un inventario con `map<string, int>` (producto -> cantidad). Permite agregar stock, vender (restar, sin dejar negativo) y consultar. Usa `find` para búsquedas seguras.

Solución:

```
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<std::string, int> inventario;
    int opcion;

    do {
        std::cout << "\n1.Agregar 2.Vender 3.Consultar 4.Ver todo 0.Salir: ";
        std::cin >> opcion;
        std::cin.ignore();
        std::string prod;
        int cant;

        switch (opcion) {
            case 1:
                std::cout << "Producto: "; std::getline(std::cin, prod);
                std::cout << "Cantidad: "; std::cin >> cant;
                inventario[prod] += cant;
                break;
            case 2: {
                std::cout << "Producto: "; std::getline(std::cin, prod);
                auto it = inventario.find(prod);
                if (it == inventario.end()) {
                    std::cout << "No existe." << std::endl;
                    break;
                }
            }
        }
    }
```

```

        std::cout << "Cantidad: "; std::cin >> cant;
        if (cant <= it->second) {
            it->second -= cant;
            std::cout << "Vendido." << std::endl;
        } else {
            std::cout << "Stock insuficiente." << std::endl;
        }
        break;
    }
    case 3: {
        std::cout << "Producto: "; std::getline(std::cin, prod);
        auto it = inventario.find(prod);
        if (it != inventario.end())
            std::cout << "Stock: " << it->second << std::endl;
        else
            std::cout << "No existe." << std::endl;
        break;
    }
    case 4:
        for (const auto& p : inventario)
            std::cout << p.first << ": " << p.second << std::endl;
        break;
    }
} while (opcion != 0);
return 0;
}

```

Explicación: inventario[prod] += cant crea el producto si no existe (aquí sí queremos ese comportamiento). Para vender y consultar usamos find para no crear entradas fantasma. cin.ignore() limpia el Enter antes de cada getline.

Ejercicio 2

Usa un set<string> para registrar palabras únicas de un texto que el usuario escriba línea por línea (línea vacía termina). Muestra cuántas palabras únicas hay y la lista ordenada.

Solución:


```

#include <iostream>
#include <set>
#include <string>
#include <sstream>

int main() {
    std::set<std::string> unicas;
    std::string linea;
    std::cout << "Escribe texto (linea vacia termina):" << std::endl;

    while (std::getline(std::cin, linea) && !linea.empty()) {
        std::istringstream flujo(linea);
        std::string palabra;
        while (flujo >> palabra) {
            unicas.insert(palabra);
        }
    }

    std::cout << "Palabras unicas: " << unicas.size() << std::endl;
    for (const auto& p : unicas) std::cout << p << std::endl;
    return 0;
}

```

Explicación: istringstream trata un string como flujo de entrada, así partimos cada línea en palabras con >>. El set elimina duplicados y ordena automáticamente. Dos herramientas de la STL combinadas en un programa útil.

Ejercicio 3

Escribe una función `raizCuadrada(double x)` que lance `std::invalid_argument` si `x` es negativo. Pruébala con try/catch y varios valores.

Solución:

```

#include <iostream>
#include <cmath>
#include <stdexcept>

double raizCuadrada(double x) {
    if (x < 0) {
        throw std::invalid_argument("No existe raiz real de numero negativo");
    }
    return std::sqrt(x);
}

int main() {
    double valores[] = {16, 2, -9, 0};
    for (double v : valores) {
        try {
            std::cout << "raiz(" << v << ") = " << raizCuadrada(v) << std::endl;
        } catch (const std::invalid_argument& e) {
            std::cout << "raiz(" << v << "): Error: " << e.what() << std::endl;
        }
    }
    return 0;
}

```

Explicación: `invalid_argument` es más específico que `runtime_error`: comunica que el problema fue el argumento. Atrapar la excepción específica permite manejos diferenciados. El ciclo sigue con los demás valores aunque uno falle: el programa es resiliente.

Ejercicio 4

Modifica el programa de alumnos del ejemplo 3 para que también guarde la matrícula (int) y al cargar valide que cada línea tenga las 3 partes; si una línea está corrupta, lánzala como excepción con el número de línea.

Solución:

```

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <stdexcept>

struct Alumno {
    int matricula;
    std::string nombre;
}

```

```

    double promedio;
};

std::vector<Alumno> cargar(const std::string& archivo) {
    std::vector<Alumno> alumnos;
    std::ifstream in(archivo);
    if (!in.is_open()) return alumnos;
    std::string linea;
    int numLinea = 0;
    while (std::getline(in, linea)) {
        numLinea++;
        size_t p1 = linea.find(',');
        size_t p2 = linea.find(',', p1 == std::string::npos ? 0 : p1 + 1);
        if (p1 == std::string::npos || p2 == std::string::npos) {
            throw std::runtime_error("Linea " + std::to_string(numLinea) + "
corrupta");
        }
        Alumno a;
        a.matricula = std::stoi(linea.substr(0, p1));
        a.nombre = linea.substr(p1 + 1, p2 - p1 - 1);
        a.promedio = std::stod(linea.substr(p2 + 1));
        alumnos.push_back(a);
    }
    return alumnos;
}

int main() {
    try {
        auto alumnos = cargar("alumnos.txt");
        std::cout << "Cargados: " << alumnos.size() << std::endl;
        for (const auto& a : alumnos) {
            std::cout << a.matricula << " | " << a.nombre << " | " << a.promedio <<
std::endl;
        }
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
    return 0;
}

```

Explicación: validamos el formato antes de convertir: si faltan comas, la línea está corrupta y lo reportamos con su número. stoi convierte a int. Este es el nivel de robustez que distingue software profesional: no confía en que los datos vengan bien.

Ejercicio 5

Crea un diario personal: cada ejecución pide una entrada de texto, la agrega con fecha al archivo "diario.txt" (modo append) y luego muestra las últimas 3 entradas. Usa <ctime> para la fecha.

Solución:

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <ctime>

std::string fechaHoy() {
    std::time_t t = std::time(nullptr);
    char buf[20];
    std::strftime(buf, sizeof(buf), "%Y-%m-%d", std::localtime(&t));
    return buf;
}

int main() {
    std::string entrada;
    std::cout << "Escribe tu entrada: ";
    std::getline(std::cin, entrada);

    std::ofstream out("diario.txt", std::ios::app);
    if (out.is_open()) {
        out << "[" << fechaHoy() << "]" << entrada << "\n";
    }

    std::ifstream in("diario.txt");
    std::vector<std::string> lineas;
    std::string linea;
    while (std::getline(in, linea)) lineas.push_back(linea);

    std::cout << "\nÚltimas entradas:" << std::endl;
    int inicio = lineas.size() > 3 ? lineas.size() - 3 : 0;
    for (size_t i = inicio; i < lineas.size(); i++) {
        std::cout << lineas[i] << std::endl;
    }
    return 0;
}
```

Explicación: modo append no borra lo anterior: el diario crece con cada ejecución. strftime formatea la fecha. Leer todo a un vector y mostrar el final es simple y funciona bien para archivos razonables. Tu primer programa con "memoria" entre ejecuciones.

Resumen de la unidad

- std::map asocia claves únicas con valores (diccionario ordenado); [] crea si no existe, find busca sin crear. std::set guarda elementos únicos.
- Excepciones: throw lanza, try/catch atrapa. Separan el camino feliz del manejo de errores. Usa tipos de <stdexcept> y atrapa por const&.
- Archivos: ofstream escribe, ifstream lee, << y >> funcionan igual que con consola. Verifica is_open() y usa ios::app para agregar.
- Patrón while (getline(...)) para leer archivos línea por línea.
- En la unidad 10 cerrarás con punteros inteligentes, un proyecto final que integra todo y las buenas prácticas del profesional.

Profundización: más STL, seguridad ante excepciones y sistema de archivos

Ya manejas mapas, conjuntos, excepciones y archivos de texto. Ahora completa tu caja profesional: más contenedores (unordered_map, queue, stack, deque), los niveles de seguridad ante excepciones que todo diseñador debe conocer, archivos binarios, stringstream para procesar texto en memoria y la biblioteca <filesystem> para manejar archivos y carpetas como un profesional.

Más contenedores: elige el correcto

unordered_map: como map, pero sin orden y con búsqueda promedio más rápida (usa tabla hash). Úsalo cuando el orden no importe y quieras velocidad:

```
#include <unordered_map>
std::unordered_map<std::string, int> conteo;
conteo["manzana"]++; // rápido, sin orden garantizado
```

Regla: map si necesitas orden o rangos; unordered_map si solo buscas por clave y quieres velocidad.

queue: cola FIFO (el primero en entrar es el primero en salir). Para turnos, impresoras, tareas pendientes:

```
#include <queue>
std::queue<std::string> fila;
fila.push("Ana"); fila.push("Luis");
std::cout << fila.front() << std::endl; // Ana
fila.pop();
```

stack: pila LIFO (calculadoras, deshacer, paréntesis balanceados). deque: cola de doble extremo, eficiente por ambos lados. array: `std::array<int, 5>` es un arreglo de tamaño fijo con la interfaz segura de la STL (`size()`, `at()`).

Tabla de decisión rápida:

- Acceso por posición y crecimiento: vector.
- Búsqueda por clave ordenada: map. Sin orden: `unordered_map`.
- Unicidad: `set` / `unordered_set`.
- Turnos: queue. Historial/deshacer: stack.
- Tamaño fijo conocido: array.

Seguridad ante excepciones: los tres niveles

Cuando diseñas funciones que pueden lanzar, hay tres garantías posibles:

1. Garantía básica: si lanza, el programa queda en estado válido (sin fugas ni corrupción), aunque los datos hayan cambiado.
2. Garantía fuerte: si lanza, todo queda como si la función nunca se hubiera llamado (transaccional: todo o nada).
3. Garantía de no-lanzar (`noexcept`): la función jamás lanza. Para destructores, movimientos y operaciones críticas.

La garantía fuerte se logra con "copia y cambia": trabaja sobre una copia y solo al final intercambia con el original. Si algo falla a la mitad, el original sigue intacto. Los contenedores de la STL ofrecen estas garantías; tus clases deberían aspirar a lo mismo.

Marca con `noexcept` lo que no lanza: `void limpiar() noexcept`. Permite optimizaciones y es parte del contrato. Los destructores son `noexcept` por defecto: nunca dejes que un destructor lance (si se lanza durante el desenrollado de otra excepción, el programa termina).

Archivos binarios

Hasta ahora guardaste texto. Para datos crudos (imágenes, estructuras, registros compactos), usa modo binario:

```
#include <fstream>

struct Registro {
    int id;
    double valor;
};

int main() {
    Registro r{42, 3.14};

    // Escribir binario
    std::ofstream out("datos.bin", std::ios::binary);
    out.write(reinterpret_cast<char*>(&r), sizeof(r));
    out.close();

    // Leer binario
    Registro leido{};
    std::ifstream in("datos.bin", std::ios::binary);
    in.read(reinterpret_cast<char*>(&leido), sizeof(leido));
    std::cout << leido.id << " " << leido.valor << std::endl;
    return 0;
}
```

write vuelca los bytes tal cual: compacto y rápido. Advertencias: no sirve para clases con punteros o strings (guardarías direcciones, no contenido), y los archivos no son portables entre arquitecturas distintas. Para datos simples y velocidad, es imbatible.

stringstream: flujos en memoria

<sstream> permite tratar strings como archivos: útil para convertir y parsear.

```

#include <sstream>
#include <iostream>

int main() {
    // Convertir número a string
    std::ostringstream oss;
    oss << "ID-" << 42 << "-" << 3.14;
    std::string s = oss.str();
    std::cout << s << std::endl;

    // Parsear string
    std::istringstream iss("Ana 25 1.70");
    std::string nombre;
    int edad;
    double estatura;
    iss >> nombre >> edad >> estatura;
    std::cout << nombre << ", " << edad << " años, " << estatura << "m" <<
std::endl;
    return 0;
}

```

ostringstream construye texto complejo sin concatenaciones tediosas; istringstream parte texto con >> como si fuera cin. Son las herramientas estándar para formateo y parsing.

<filesystem>: archivos y carpetas

Desde C++17, manipular el sistema de archivos es estándar:


```

#include <iostream>
#include <filesystem>

namespace fs = std::filesystem;

int main() {
    fs::path dir = "mi_proyecto";
    fs::create_directory(dir); // crea la carpeta

    std::cout << "Existe: " << fs::exists(dir) << std::endl;

    // Listar contenido actual
    for (const auto& entrada : fs::directory_iterator(".")) {
        std::cout << entrada.path().filename() << std::endl;
    }

    std::cout << "Directorio actual: " << fs::current_path() << std::endl;
    return 0;
}

```

Crear carpetas, verificar existencia, listar directorios, copiar/mover archivos: todo portable entre Windows, Linux y macOS. Antes de C++17 cada sistema necesitaba código distinto.

Más ejemplos

Ejemplo: contador de palabras con `unordered_map`

```

#include <iostream>
#include <string>
#include <unordered_map>
#include <sstream>
#include <algorithm>
#include <vector>

int main() {
    std::string texto = "el gato come y el perro corre y el gato duerme";
    std::unordered_map<std::string, int> conteo;

    std::istringstream iss(texto);
    std::string palabra;
    while (iss >> palabra) {
        conteo[palabra]++;
    }

    // Pasar a vector para ordenar por frecuencia
    std::vector<std::pair<std::string, int>> ranking(conteo.begin(), conteo.end());
    std::sort(ranking.begin(), ranking.end(),
              [](const auto& a, const auto& b){ return a.second > b.second; });

    for (const auto& par : ranking) {
        std::cout << par.first << ": " << par.second << std::endl;
    }
    return 0;
}

```

`unordered_map` cuenta rápido; luego convertimos a vector para ordenar por frecuencia (los mapas no se ordenan por valor). Combinación típica en análisis de texto real.

Ejemplo: paréntesis balanceados con stack

```

#include <iostream>
#include <string>
#include <stack>

bool balanceados(const std::string& expr) {
    std::stack<char> pila;
    for (char c : expr) {
        if (c == '(' || c == '[' || c == '{') {
            pila.push(c);
        } else if (c == ')' || c == ']' || c == '}') {
            if (pila.empty()) return false;
            char abierto = pila.top();
            pila.pop();
            if ((c == ')' && abierto != '(') ||
                (c == ']' && abierto != '[') ||
                (c == '}' && abierto != '{')) {
                return false;
            }
        }
    }
    return pila.empty();
}

int main() {
    for (const std::string& e : {"(a+b)*[c-{d}]", "((a+b)", "(a+b])", ""}) {
        std::cout << e << " -> " << (balanceados(e) ? "OK" : "MAL") << std::endl;
    }
    return 0;
}

```

Problema clásico de entrevistas resuelto con stack: cada apertura se apila, cada cierre debe corresponder al tope. Si al final la pila no está vacía, faltaron cierres. Los compiladores hacen algo parecido con tus llaves.

Práctica adicional

Ejercicio A

Compara map vs unordered_map: inserta 100,000 pares y mide el tiempo de inserción y de 10,000 búsquedas con <chrono>. ¿Cuál gana y en qué?

Solución:

```

#include <iostream>
#include <map>
#include <unordered_map>
#include <chrono>
#include <string>

template <typename Mapa>
long medir(Mapa& m) {
    auto inicio = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < 100000; i++) m["clave" + std::to_string(i)] = i;
    long encontrados = 0;
    for (int i = 0; i < 10000; i++) {
        auto it = m.find("clave" + std::to_string(i * 7));
        if (it != m.end()) encontrados++;
    }
    auto fin = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::milliseconds>(fin -
inicio).count();
}

int main() {
    std::map<std::string, int> m1;
    std::unordered_map<std::string, int> m2;
    std::cout << "map: " << medir(m1) << " ms" << std::endl;
    std::cout << "unordered_map: " << medir(m2) << " ms" << std::endl;
    return 0;
}

```

Explicación: unordered_map suele ganar en búsquedas puras por su hash O(1). map gana si necesitas recorrer ordenado o buscar rangos. Medir en tu máquina vale más que cualquier teoría.

Ejercicio B

Escribe una función que valide un CSV simple: cada línea debe tener exactamente 3 campos separados por comas. Usa stringstream o find y lanza excepción con el número de línea si alguna falla.

Solución:

```

#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <stdexcept>

void validarCSV(const std::vector<std::string>& lineas) {
    int num = 0;
    for (const auto& linea : lineas) {
        num++;
        std::istringstream ss(linea);
        std::string campo;
        int campos = 0;
        while (std::getline(ss, campo, ',')) campos++;
        if (campos != 3) {
            throw std::runtime_error("Linea " + std::to_string(num) +
                                     ": se esperaban 3 campos, hay " +
std::to_string(campos));
        }
    }
}

int main() {
    try {
        validarCSV({"a,b,c", "1,2", "x,y,z"});
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
    return 0;
}

```

Explicación: getline con delimitador ',' parte cada línea en campos. Validar formatos de entrada es trabajo diario en software real: los datos del mundo vienen sucios.

Ejercicio C

Crea un programa que liste todos los archivos .txt del directorio actual usando <filesystem> y muestre su tamaño en bytes.

Solución:

```

#include <iostream>
#include <filesystem>

namespace fs = std::filesystem;

int main() {
    for (const auto& entrada : fs::directory_iterator(".")) {
        if (entrada.is_regular_file() && entrada.path().extension() == ".txt") {
            std::cout << entrada.path().filename()
                      << " (" << fs::file_size(entrada.path()) << " bytes)" <<
std::endl;
        }
    }
    return 0;
}

```

Explicación: `extension()` filtra por tipo, `file_size` da el tamaño. Con `<filesystem>` este utilitario es portable: compila igual en Windows, Linux y macOS.

Ejercicio D

Implementa una cola de impresión con `queue<string>`: agrega 5 documentos, "imprime" (muestra y saca) los 2 primeros, agrega uno más y muestra cuántos quedan y cuál sigue.

Solución:

```

#include <iostream>
#include <string>
#include <queue>

int main() {
    std::queue<std::string> cola;
    for (int i = 1; i <= 5; i++) {
        cola.push("Documento " + std::to_string(i));
    }

    for (int i = 0; i < 2; i++) {
        std::cout << "Imprimiendo: " << cola.front() << std::endl;
        cola.pop();
    }

    cola.push("Documento urgente");
    std::cout << "En cola: " << cola.size() << std::endl;
    std::cout << "Siguiente: " << cola.front() << std::endl;
    return 0;
}

```

Explicación: FIFO puro: el primero que llegó es el primero que sale. `front()` mira sin sacar, `pop()` saca sin devolver (por seguridad ante excepciones: así diseñó la STL la interfaz).

Expansion Unidad 9: errores profesionales, bitacoras y datos que aguantan el mundo real

Ya sabes lanzar y atrapar excepciones, guardar archivos de texto y organizar datos con mapas y conjuntos. Con eso puedes escribir programas que funcionan cuando todo sale bien. Esta expansion te enseña la otra mitad del oficio: diseñar tus propios tipos de error como lo hacen las bibliotecas serias, garantizar que los recursos se liberen aunque una excepcion parta tu programa a la mitad, decidir entre `assert` y excepciones sin dudar, dejar bitacoras que te cuenten que paso cuando no estabas viendo, y leer archivos de datos sucios sin que tu programa se rompa. Esa es la diferencia entre un programa de tarea y un sistema que alguien usaria de verdad.

Nada de lo que sigue repite las clases anteriores: aqui construyes sobre ellas. Si algo te suena familiar, es porque lo estas usando, no porque lo estes releendo.

Excepciones personalizadas: disena tu propia jerarquia

Por que crear tus propias excepciones

En la clase 26 lanzaste `std::runtime_error` con mensajes de texto. Funciona, pero tiene un limite: el codigo que atrapa solo puede leer el mensaje. No puede distinguir con el tipo si fue un error de red, de permisos o de datos corruptos, a menos que compare strings, lo cual es fragil y feo.

Las bibliotecas profesionales resuelven esto con jerarquias: una clase base para todos sus errores y derivadas para cada caso. Asi el que usa tu codigo puede atrapar lo especifico (`FondosInsuficientes`) o lo general (`ErrorBanco`) según le convenga. Es el mismo principio del polimorfismo que ya conoces, aplicado a los errores.

Anatomia de una excepcion bien hecha

Toda excepcion personalizada hereda de `std::exception` (o de una derivada como `std::runtime_error`) y sobrescribe `what()`. Tres reglas de oro:

1. Hereda de `std::exception`, directa o indirectamente. Asi cualquier `catch (const std::exception&)` la atrapa.
2. Guarda el mensaje en un `std::string` miembro. Nunca devuelvas el `c_str()` de un temporal: el puntero quedaria colgando.
3. Marca `what()` como `const noexcept override`. Si `what()` lanzara, el programa terminaria: es un contrato sagrado.

El constructor suele recibir los datos del error y armar el mensaje ahi mismo, para que el codigo que lanza quede limpio: una sola linea con los datos, sin concatenar strings en cada `throw`.

Ejemplo completo 1: jerarquia de errores de un banco

```
#include <iostream>
#include <string>
#include <exception>

// Clase base de todos los errores del banco
class ErrorBanco : public std::exception {
private:
    std::string mensaje;
public:
```



```

    explicit ErrorBanco(const std::string& msg) : mensaje("[Banco] " + msg) {}
    const char* what() const noexcept override {
        return mensaje.c_str();
    }
};

// Errores especificos: cada uno sabe armar su propio mensaje
class FondosInsuficientes : public ErrorBanco {
public:
    FondosInsuficientes(double saldo, double retiro)
        : ErrorBanco("Fondos insuficientes: saldo " +
            std::to_string(saldo) + ", retiro " +
            std::to_string(retiro)) {}
};

class CuentaBloqueada : public ErrorBanco {
public:
    explicit CuentaBloqueada(const std::string& cuenta)
        : ErrorBanco("La cuenta " + cuenta + " esta bloqueada") {}
};

class LimiteDiarioExcedido : public ErrorBanco {
public:
    explicit LimiteDiarioExcedido(double monto)
        : ErrorBanco("Retiro de " + std::to_string(monto) +
            " excede el limite diario") {}
};

class Cuenta {
private:
    std::string numero;
    double saldo;
    bool bloqueada;
    double retiradoHoy;
    static constexpr double LIMITE_DIARIO = 5000.0;
public:
    Cuenta(const std::string& num, double inicial)
        : numero(num), saldo(inicial), bloqueada(false), retiradoHoy(0.0) {}

    void bloquear() { bloqueada = true; }
    void desbloquear() { bloqueada = false; }

    void retirar(double monto) {
        if (bloqueada) throw CuentaBloqueada(numero);
        if (monto < 0) throw ErrorBanco("Monto negativo no permitido");
    }
};

```

```

        if (monto > saldo) throw FondosInsuficientes(saldo, monto);
        if (retiradoHoy + monto > LIMITE_DIARIO)
            throw LimiteDiarioExcedido(monto);
        saldo -= monto;
        retiradoHoy += monto;
        std::cout << "Retiro exitoso: " << monto
                    << ". Nuevo saldo: " << saldo << std::endl;
    }

    double getSaldo() const { return saldo; }
};

int main() {
    Cuenta c("123-456", 10000.0);

    try { c.retirar(2000.0); }
    catch (const ErrorBanco& e) { std::cout << "Error: " << e.what() << std::endl; }

    try { c.retirar(20000.0); }
    catch (const FondosInsuficientes& e) {
        std::cout << "Sin fondos: " << e.what() << std::endl;
    }

    c.bloquear();
    try { c.retirar(100.0); }
    catch (const ErrorBanco& e) {
        std::cout << "Error de banco: " << e.what() << std::endl;
    }
    c.desbloquear();

    try {
        c.retirar(4000.0); // ya retiro 2000 hoy: 2000 + 4000 > 5000
    } catch (const LimiteDiarioExcedido& e) {
        std::cout << "Limite: " << e.what() << std::endl;
    }

    std::cout << "Saldo final: " << c.getSaldo() << std::endl;
    return 0;
}

```

Fijate en el patron: la clase Cuenta no sabe nada de pantallas ni de mensajes al usuario; solo describe el problema con un tipo. Quien llama decide: el cajero automatico podria atrapar FondosInsuficientes y ofrecer un monto menor, mientras que un reporte nocturno atraparía ErrorBanco y lo registraría todo junto. Esa flexibilidad es exactamente para lo que sirven las jerarquias.

Atrapar por jerarquia: del especifico al general

Cuando pongas varios catch, el orden importa: primero las derivadas, al final la base. Si pones catch (const ErrorBanco&) antes que catch (const FondosInsuficientes&), el primero se come todo y el especifico jamas se ejecuta. El compilador no siempre te avisa; es tu responsabilidad. Piensalo como un embudo: lo mas especifico arriba, lo mas general abajo, y al fondo del todo, si quieres, un catch (...) como red de seguridad, que veremos en la siguiente seccion.

Otra buena practica: no abuses de la profundidad. Dos o tres niveles bastan (ErrorRed -> ErrorConexion / ErrorTiempoAgotado). Una jerarquia de diez niveles es peor que no tener jerarquia, porque nadie se acuerda de que existe cada tipo.

RAII y excepciones: la garantia en la practica

Una clase, dos recursos: donde RAII brilla de verdad

Ya sabes que RAII significa que el recurso vive atado a la vida de un objeto: se adquiere en el constructor y se libera en el destructor. Con un solo recurso es facil de crear. El caso interesante es cuando una clase adquiere DOS recursos y el segundo falla: el primero debe liberarse solo, sin que escribas ni una linea de limpieza manual. Eso es exactamente lo que pasa en el siguiente ejemplo, y es la razon por la que el C++ moderno puede usar excepciones sin vivir con miedo.

La regla clave: los miembros se construyen en el orden en que estan declarados, y si el constructor lanza a la mitad, los miembros ya contruidos se destruyen en orden inverso. Tu no haces nada: el compilador genera esa limpieza.

Ejemplo completo 2: respaldo que adquiere base de datos y bitacora

```
#include <iostream>
#include <fstream>
#include <string>
#include <stdexcept>

// Recurso 1: simula una conexion a base de datos
class ConexionBD {
private:
    std::string nombre;
    bool abierta;
public:
    explicit ConexionBD(const std::string& bd) : nombre(bd), abierta(false) {
```

```

        if (bd.empty()) throw std::runtime_error("Nombre de BD vacio");
        abierta = true;
        std::cout << "[BD] Conexion abierta a '" << nombre << "'" << std::endl;
    }
    ~ConexionBD() {
        if (abierta)
            std::cout << "[BD] Conexion cerrada a '" << nombre << "'" << std::endl;
    }
    void ejecutar(const std::string& sql) {
        if (!abierta) throw std::runtime_error("BD cerrada");
        std::cout << "[BD] Ejecutando: " << sql << std::endl;
    }
};

// Adquiere DOS recursos: la conexion y el archivo de bitacora
class GestorRespaldo {
private:
    ConexionBD bd;           // se construye primero...
    std::ofstream bitacora; // ...y este después
public:
    GestorRespaldo(const std::string& nombreBD, const std::string& archivoLog)
        : bd(nombreBD), bitacora(archivoLog, std::ios::app) {
        if (!bitacora.is_open())
            throw std::runtime_error("No se pudo abrir la bitacora");
        bitacora << "[respaldo] inicio\n";
        std::cout << "[Gestor] Recursos adquiridos." << std::endl;
    }
    ~GestorRespaldo() {
        // Orden inverso al constructor: primero la bitacora, luego la BD
        bitacora << "[respaldo] fin\n";
        std::cout << "[Gestor] Recursos liberados." << std::endl;
    }
    void respaldar(const std::string& tabla) {
        bd.ejecutar("SELECT * FROM " + tabla);
        bitacora << "[respaldo] tabla " << tabla << " OK\n";
        if (tabla == "usuarios") {
            throw std::runtime_error("Fallo simulado al respaldar 'usuarios'");
        }
    }
};

int main() {
    try {
        GestorRespaldo g("mi_base", "respaldo.log");
        g.respaldar("productos");
    }
}

```

```

        g.respalдар("usuarios");    // lanza a la mitad
        g.respalдар("ventas");      // nunca se ejecuta
    } catch (const std::exception& e) {
        std::cout << "Atrapado: " << e.what() << std::endl;
    }
    std::cout << "Programa estable: ningun recurso quedo abierto." << std::endl;

    // Caso 2: si la bitacora falla, la BD ya abierta se cierra sola
    try {
        GestorRespaldo g2("mi_base", "/ruta/imposible/x.log");
    } catch (const std::exception& e) {
        std::cout << "Atrapado: " << e.what() << std::endl;
    }
    return 0;
}

```

Lee la salida con atencion y veras la coreografia completa. En el primer caso, la excepcion interrumpe respaldar, pero antes de llegar al catch se ejecutan los destructores: primero el de GestorRespaldo (que escribe el "fin" en la bitacora), luego sus miembros en orden inverso (bitacora, después bd). En el segundo caso, el constructor lanza porque el archivo no se pudo abrir, y la ConexionBD que ya se habia construido se destruye sola. Cero fugas, cero codigo de limpieza manual.

Este es el patron que usan las clases serias: cada miembro es un RAII por si mismo (un fstream, un unique_ptr, un lock_guard), asi la clase contenedora queda automaticamente segura ante excepciones. Si en cambio guardaras punteros crudos o identificadores sueltos (un FILE*, un int de socket), tendrías que escribir la limpieza a mano en cada camino de error, y tarde o temprano olvidarias uno.

Orden de destruccion: tu aliado silencioso

Repite la regla hasta que salga sola: los miembros se destruyen en orden inverso al de construccion, y el orden de construccion lo dictan las declaraciones en la clase, no la lista de inicializacion. Si declaras primero la bitacora y después la BD, pero en la lista de inicializacion pones la BD primero, el compilador te hara caso en las declaraciones (y con -Wall te avisara del desorden). Declara los miembros en el orden en que deben nacer, y la muerte ocurrira al reves, como debe ser.

Relanzar excepciones y catch(...)

throw; conserva el tipo original

A veces una funcion de nivel bajo detecta el error pero no tiene contexto para manejarlo; una funcion de nivel alto si lo tiene. La tecnica es atrapar, agregar informacion (por ejemplo, escribir en la bitacora o reintentar) y relanzar la misma excepcion con throw; solito, sin nada después. Ese throw vacio relanza el objeto original intacto, con su tipo dinamico completo.

El error clasico es escribir throw e; donde e es la referencia atrapada. Eso crea una COPIA estatica del tipo declarado en el catch, y si atrapaste por la base, el objeto se "rebana" (slicing): pierde su tipo derivado y la informacion extra. Regla simple: para relanzar lo mismo, throw; siempre.

catch(...) : la red de seguridad final

catch (...) atrapa absolutamente todo, incluso tipos que no heredan de std::exception (alguien podria hacer throw 42; aunque no deberia). Usalo solo en dos lugares: en main como ultima red para que el programa termine con un mensaje decente en vez de un aborto crudo, y cuando necesites hacer limpieza antes de relanzar sin importar que tipo de excepcion venga en camino. Jamas lo uses para "tragar" errores en silencio: una excepcion tragada es un bug invisible.

Ejemplo completo 3: reintento con contexto y red final

```
#include <iostream>
#include <string>
#include <stdexcept>

class ErrorRed : public std::runtime_error {
public:
    explicit ErrorRed(const std::string& m) : std::runtime_error(m) {}
};

void descargarArchivo(const std::string& url) {
    if (url.empty()) throw ErrorRed("URL vacia");
    throw ErrorRed("No se pudo conectar al servidor");
}

void sincronizar(const std::string& url) {
    try {
        descargarArchivo(url);
    } catch (const ErrorRed& e) {
        std::cout << "[sincronizar] fallo la descarga, reintentando..." << std::endl;
        try {
```

```

        descargarArchivo(url); // segundo intento
    } catch (...) {
        std::cout << "[sincronizar] segundo intento fallo, relanzando" <<
std::endl;
        throw; // relanza la excepcion ORIGINAL, sin slicing
    }
}

int main() {
    try {
        sincronizar("http://ejemplo.com/datos");
    } catch (const ErrorRed& e) {
        std::cout << "Error de red atrapado: " << e.what() << std::endl;
    } catch (...) {
        std::cout << "Error desconocido: terminando con dignidad." << std::endl;
        return 1;
    }

    // Demostracion del slicing: throw e vs throw;
    try {
        try {
            throw ErrorRed("original");
        } catch (const std::runtime_error& e) {
            throw e; // MAL: rebana el objeto a runtime_error
        }
    } catch (const ErrorRed&) {
        std::cout << "Llego como ErrorRed" << std::endl;
    } catch (const std::runtime_error& e) {
        std::cout << "Slicing: llego como runtime_error: " << e.what() << std::endl;
    }
    return 0;
}

```

El ejemplo muestra el patron completo de capas: sincronizar sabe que un reintento vale la pena, pero no sabe que hacer si el reintento también falla, así que relanza; main si sabe que hacer: informar y terminar. Cada capa maneja lo que puede y deja pasar lo demas. Y la segunda parte te demuestra con tus propios ojos por que throw e; es un bug: el mensaje dice "Slicing" porque el objeto llego convertido a su clase base.

Un detalle fino: dentro del catch (...) no puedes saber que atra paste, pero si puedes relanzarlo con throw; y dejar que alguien mas arriba lo examine. Esa combinacion (catch (...) + limpieza + throw;) es el idioma estandar para "hago mi parte y paso la estafeta".

assert vs excepciones: cada uno en su lugar

Dos herramientas, dos trabajos

assert (de <cassert>) y las excepciones no compiten: detectan clases de error distintas.

- assert verifica invariantes internas: cosas que, si tu código está bien escrito, JAMÁS deberían fallar. "Este puntero no es nulo aquí", "el índice está en rango", "la lista no está vacía antes de sacar el primero". Es una red para atrapar TUS bugs durante el desarrollo.
- Las excepciones manejan errores del mundo exterior: cosas que pueden fallar aunque tu código sea perfecto. El archivo no existe, la red se cayó, el usuario escribió texto donde iba un número.

La diferencia práctica más importante: assert desaparece en compilaciones de producción (con NDEBUG definido), mientras que las excepciones siempre están ahí. Por eso nunca pongas en un assert algo con efectos secundarios ni una validación de la que dependa la seguridad del programa: en producción esa línea no existe.

```
#include <cassert>
#include <stdexcept>
#include <vector>

int elementoSeguro(const std::vector<int>& v, size_t i) {
    // Error del MUNDO: el que llama puede pedir un índice inválido.
    // Debe manejarse siempre, incluso en producción -> excepción.
    if (i >= v.size())
        throw std::out_of_range("Índice fuera de rango");
    return v[i];
}

double promedio(double suma, int n) {
    // Invariante INTERNA: si n es 0 aquí, es un bug de MI código,
    // no un error del usuario. En desarrollo quiero que truene fuerte.
    assert(n > 0 && "promedio llamado con n = 0: bug interno");
    return suma / n;
}
```

Tabla de decisión rápida:

Situación	Herramienta	Por qué	--- --- ---	Archivo no encontrado	Excepción	El mundo exterior falla
División entre cero por entrada del usuario	Excepción	Dato externo inválido				
Puntero nulo donde tu lógica garantiza que no es nulo	assert	Bug tuyo, detectalo en				

desarrollo | | Indice calculado por tu algoritmo fuera de rango | assert | Tu algoritmo esta mal | |
Indice pedido por el usuario fuera de rango | Excepcion | Dato externo | | Validacion que debe
existir en produccion | Excepcion | assert no existe con NDEBUG |

Un consejo de veterano: cuando dudes, pregunta "si esto falla en produccion, quiero que el programa se recupere o que me avise que mi logica esta mal?". Recuperarse = excepcion. Mi logica esta mal = assert. Y si el assert truena durante tus pruebas, no lo silencies: es el mejor regalo que te puede dar tu codigo, un bug encontrado antes de que lo encuentre un usuario.

Bitacoras (logging): tu caja negra

Por que los cout no bastan

Cuando un programa falla en tu maquina, le pones cout por todos lados y lo ves. Cuando falla en la maquina del cliente a las 3 de la manana, no hay nadie viendo la consola. La bitacora (log) es el archivo donde tu programa escribe que hizo, que decidio y que salio mal, para que puedas reconstruir la historia después. Todo sistema profesional tiene una; los programas sin bitacora se depuran a ciegas.

Una bitacora util tiene tres ingredientes: niveles de severidad para filtrar el ruido, destino en archivo (modo append, para que no se borre entre ejecuciones) y mensajes con contexto (que paso, con que datos, cuando).

Niveles de severidad

- DEBUG: detalle fino para el desarrollador ("entre a la funcion X con n=42"). En produccion se apaga.
- INFO: eventos normales importantes ("usuario 'ana' inicio sesion", "respaldo completado").
- AVISO (warning): algo raro pero no fatal ("disco al 85%", "reintento 2 de 3").
- ERROR: algo fallo y alguien debe enterarse ("no se pudo conectar a la BD").

La gracia de los niveles es el filtro: en desarrollo pones el minimo en DEBUG y ves todo; en produccion lo subes a AVISO y el archivo no crece sin control. Cada mensaje se compara con el nivel minimo configurado y los de nivel inferior se descartan barato, sin tocar el disco.

Ejemplo completo 4: clase Logger con niveles y archivo

```
#include <iostream>
#include <fstream>
#include <string>
#include <mutex>
#include <stdexcept>

enum class Nivel { DEBUG, INFO, AVISO, ERROR };

class Logger {
private:
    std::ofstream archivo;
    Nivel nivelMinimo;
    std::mutex mtx; // por si algun dia hay hilos (unidad 10)

    static std::string aTexto(Nivel n) {
        switch (n) {
            case Nivel::DEBUG: return "DEBUG";
            case Nivel::INFO:  return "INFO ";
            case Nivel::AVISO: return "AVISO";
            case Nivel::ERROR: return "ERROR";
        }
        return "?????";
    }
public:
    Logger(const std::string& ruta, Nivel minimo = Nivel::INFO)
        : nivelMinimo(minimo) {
        archivo.open(ruta, std::ios::app);
        if (!archivo.is_open())
            throw std::runtime_error("No se pudo abrir el log: " + ruta);
    }

    void registrar(Nivel n, const std::string& mensaje) {
        if (n < nivelMinimo) return; // filtro barato
        std::lock_guard<std::mutex> candado(mtx);
        archivo << "[" << aTexto(n) << "]" << mensaje << "\n";
        archivo.flush(); // que no se pierda si el programa se cae
        if (n == Nivel::ERROR) {
            std::cerr << "[ERROR] " << mensaje << std::endl;
        }
    }

    void debug(const std::string& m) { registrar(Nivel::DEBUG, m); }
    void info(const std::string& m) { registrar(Nivel::INFO, m); }
    void aviso(const std::string& m) { registrar(Nivel::AVISO, m); }
```

```

    void error(const std::string& m) { registrar(Nivel::ERROR, m); }
};

int main() {
    try {
        Logger log("app.log", Nivel::DEBUG);
        log.debug("Iniciando aplicacion, version 1.0");
        log.info("Usuario 'ana' inicio sesion");
        log.aviso("Disco al 85% de capacidad");
        log.error("No se pudo conectar a la base de datos");

        Logger produccion("prod.log", Nivel::AVISO);
        produccion.debug("esto no se guarda"); // filtrado
        produccion.info("esto tampoco");        // filtrado
        produccion.aviso("esto si se guarda");
        produccion.error("esto tambien");

        std::cout << "Revisa app.log y prod.log" << std::endl;
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
    return 0;
}

```

Detalles que lo hacen profesional: el archivo se abre en modo append (cada ejecucion agrega, no borra el historial); flush() después de cada mensaje para que un crash no se coma las ultimas lineas (las mas importantes); los ERROR además van a cerr para que alguien los vea en vivo; el mutex deja la puerta abierta al multihilo sin costo visible hoy; y si el log no se puede abrir, se lanza excepcion en vez de seguir ciego (un programa sin bitacora que cree tener bitacora es peor que uno sin bitacora).

Que registrar y que no

Registra: inicios y fines de operaciones importantes, decisiones ("reintentando conexion, intento 2"), errores con su contexto (que archivo, que usuario, que codigo de error), y eventos de seguridad (inicios de sesion, cambios de permisos). No registres: contraseñas ni tokens (un log con contraseñas es una brecha de seguridad esperando a pasar), datos personales sensibles sin necesidad, ni cada iteracion de un ciclo de un millon de vueltas (el disco y tu paciencia tienen limites). La regla: si mañana lees esa linea a las 3 am, te ayuda a entender que paso? Si la respuesta es si, registrala.

Serializacion manual: objetos que viajan en texto

El formato es un contrato

Serializar es convertir un objeto en una secuencia (texto o bytes) que puedas guardar o transmitir, y deserializar es reconstruirlo. Ya lo hiciste de forma simple con los alumnos en CSV, pero el mundo real tiene nombres con espacios, comas y comillas, y ahi el formato ingenuo se rompe. La leccion: el formato de serializacion es un contrato entre quien escribe y quien lee, y debe definir como se escapan los caracteres especiales.

El formato que usaremos: cada objeto en una linea, campos separados por |, y los textos van entre comillas dobles, duplicando las comillas internas ("""). Asi un nombre puede contener espacios, el separador | e incluso comillas sin romper nada.

Ejemplo completo 5: serializar y leer objetos Persona

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <stdexcept>

struct Persona {
    std::string nombre;
    int edad;
    double estatura;

    // Convierte el objeto a una linea de texto: "nombre|edad|estatura"
    std::string serializar() const {
        std::string seguro = nombre;
        std::string::size_type pos = 0;
        while ((pos = seguro.find('"', pos)) != std::string::npos) {
            seguro.insert(pos, 1, '"'); // "" escapa una comilla
            pos += 2;
        }
        std::ostringstream oss;
        oss << '"' << seguro << '"' << '|' << edad << '|' << estatura;
        return oss.str();
    }

    // Reconstruye el objeto desde una linea; lanza si el formato es invalido
    static Persona deserializar(const std::string& linea, int numLinea) {
```

```

        auto error = [&](const std::string& detalle) {
            throw std::runtime_error("Linea " + std::to_string(numLinea) + ": " +
detalle);
        };
        Persona p;
        if (linea.empty() || linea[0] != '"')
            error("se esperaba el nombre entre comillas");

        std::string::size_type i = 1;
        std::string nombre;
        bool cerrada = false;
        while (i < linea.size()) {
            if (linea[i] == '"') {
                if (i + 1 < linea.size() && linea[i + 1] == '"') {
                    nombre += '"';
                    i += 2;
                } else {
                    i++;
                    cerrada = true;
                    break;
                }
            } else {
                nombre += linea[i];
                i++;
            }
        }
        if (!cerrada) error("comilla sin cerrar en el nombre");
        if (i >= linea.size() || linea[i] != '|')
            error("se esperaba '|' despues del nombre");

        std::istringstream iss(linea.substr(i + 1));
        char sep;
        if (!(iss >> p.edad >> sep >> p.estatura) || sep != '|')
            error("edad o estatura invalidos");
        p.nombre = nombre;
        return p;
    }

    void mostrar() const {
        std::cout << nombre << ", " << edad << " años, "
            << estatura << " m" << std::endl;
    }
};

void guardarPersonas(const std::vector<Persona>& v, const std::string& ruta) {

```

```

    std::ofstream out(ruta);
    if (!out.is_open()) throw std::runtime_error("No se pudo escribir " + ruta);
    for (const auto& p : v) out << p.serialize() << "\n";
}

std::vector<Persona> cargarPersonas(const std::string& ruta) {
    std::vector<Persona> v;
    std::ifstream in(ruta);
    if (!in.is_open()) throw std::runtime_error("No se pudo leer " + ruta);
    std::string linea;
    int n = 0;
    while (std::getline(in, linea)) {
        n++;
        if (linea.empty()) continue;
        v.push_back(Persona::deserialize(linea, n));
    }
    return v;
}

int main() {
    std::vector<Persona> grupo = {
        {"Ana Maria Lopez", 28, 1.65},
        {"Luis \"El Profe\" Garcia", 45, 1.78},
        {"Marta", 33, 1.70}
    };
    guardarPersonas(grupo, "personas.txt");

    std::vector<Persona> leidas = cargarPersonas("personas.txt");
    std::cout << "Leidas " << leidas.size() << " personas:" << std::endl;
    for (const auto& p : leidas) p.mostrar();
    return 0;
}

```

El archivo personas.txt queda asi:

```

"Ana Maria Lopez"|28|1.65
"Luis ""El Profe"" Garcia"|45|1.78
"Marta"|33|1.7

```

Tres decisiones de diseño que valen oro. Una: serializar y deserializar viven JUNTO al struct, como métodos, para que el formato tenga un solo dueño (si mañana agregas un campo, lo cambias en un solo lugar). Dos: deserializar valida todo y reporta el número de línea: cuando un archivo de 10,000 líneas trae una corrupta, quieres saber cuál. Tres: el escape de comillas duplicándolas es el estándar de facto (lo usa el CSV, como verás enseguida), así que tu formato ya habla un dialecto que el mundo entiende.

Nota: 1.70 se guarda como "1.7" porque ostringstream usa 6 dígitos significativos por defecto. Al leer, 1.7 vuelve a ser el mismo double. Para dinero o ciencia usarías precisión fija (ver <iomanip>), pero para este formato el ida y vuelta es exacto.

CSV robusto: comillas y comas dentro de campos

Las reglas del CSV real

El ejercicio B de la profundización validaba un CSV simple partiendo por comas. Eso se rompe en cuanto un campo contiene una coma: "García, Luis",45 se parte en tres campos en vez de dos. El CSV de verdad (RFC 4180, el que exporta Excel) tiene reglas claras:

1. Los campos se separan con comas.
2. Un campo entre comillas dobles puede contener comas, espacios y hasta saltos de línea.
3. Una comilla dentro de un campo entrecomillado se escribe duplicada ("").
4. Una comilla que aparece en un campo SIN entrecomillar es un error de formato.

Con esas cuatro reglas puedes leer casi cualquier CSV del mundo real. El parser es una maquina de estados: voy "entre comillas" o "fuera de comillas", y cada carácter se interpreta según el estado.

Ejemplo completo 6: parser CSV que aguanta datos sucios

```
#include <iostream>
#include <string>
#include <vector>
#include <stdexcept>

// Parte una línea CSV según RFC 4180 simplificado.
// Lanza excepción con el número de línea si el formato es inválido.
std::vector<std::string> parsearCSV(const std::string& línea, int numLínea) {
    std::vector<std::string> campos;
    std::string actual;
```

```

bool entreComillas = false;

for (size_t i = 0; i < linea.size(); i++) {
    char c = linea[i];
    if (entreComillas) {
        if (c == '"') {
            if (i + 1 < linea.size() && linea[i + 1] == '"') {
                actual += '"'; // "" -> comilla literal
                i++;
            } else {
                entreComillas = false; // fin del campo entrecomillado
            }
        } else {
            actual += c; // comas y todo lo demas van tal cual
        }
    } else {
        if (c == '"') {
            if (!actual.empty())
                throw std::runtime_error("Linea " + std::to_string(numLinea) +
                    ": comilla en medio de campo sin entrecomillar");
            entreComillas = true;
        } else if (c == ',') {
            campos.push_back(actual);
            actual.clear();
        } else {
            actual += c;
        }
    }
}

if (entreComillas)
    throw std::runtime_error("Linea " + std::to_string(numLinea) +
        ": comilla sin cerrar");

campos.push_back(actual);
return campos;
}

int main() {
    std::vector<std::string> lineas = {
        "nombre,edad,ciudad",
        "Ana,28,\"Ciudad de Mexico\"",
        "\"Garcia, Luis\",45,Monterrey",
        "\"Dice \"\"hola\"\" siempre\",33,Puebla"
    };

    int n = 0;

```



```

for (const auto& l : lineas) {
    n++;
    try {
        auto campos = parsearCSV(l, n);
        std::cout << "Linea " << n << " (" << campos.size() << " campos):";
        for (const auto& c : campos) std::cout << " [" << c << "];";
        std::cout << std::endl;
        if (n > 1 && campos.size() != 3)
            std::cout << " -> AVISO: se esperaban 3 campos" << std::endl;
    } catch (const std::exception& e) {
        std::cout << "Linea " << n << " ERROR: " << e.what() << std::endl;
    }
}
return 0;
}

```

La salida muestra que "Garcia, Luis" queda como un solo campo y que las comillas duplicadas se convierten en una sola. Este parser de 40 lineas ya supera a la mayoría del código "rápido" que parte por comas a mano, que es la causa número uno de archivos mal leídos en sistemas reales.

Dos extensiones naturales cuando las necesites: campos con saltos de línea (requiere leer el archivo teniendo en cuenta el estado entrecomillado en vez de línea por línea) y la función inversa, `escaparCampo`, que envuelve un campo en comillas si contiene coma, comilla o salto. Esa función inversa es justamente uno de los ejercicios de práctica: escribir y leer son las dos caras del mismo contrato.

Catalogo de errores comunes de esta expansion

1. Atrapar por valor en vez de por `const&`: `catch (ErrorBanco e)` copia el objeto y puede rebanarlo si era de un tipo derivado. Atrapa siempre por referencia constante.
2. `throw e;` en vez de `throw;` para relanzar. El primero copia con el tipo estático del `catch` y pierde el tipo derivado; el segundo conserva el objeto original intacto.
3. Poner el `catch` de la clase base antes que el de la derivada. El base se come todas las excepciones y los `catch` específicos de abajo quedan muertos. Orden: de lo más específico a lo más general.
4. `catch (...)` que traga errores en silencio. Usalo solo como red final en `main` o para limpiar antes de relanzar. Un `catch (...)` vacío convierte bugs en misterios.

5. Lanzar excepciones desde destructores. Si el destructor se ejecuta durante el desenrollado de otra excepcion, dos excepciones activas terminan el programa (`std::terminate`). Los destructores limpian, no reportan.
6. `what()` sin `noexcept` o devolviendo el `c_str()` de un string temporal. Guarda el mensaje en un miembro `std::string` y declara `const char* what() const noexcept override`. Un `what()` que lanza o cuelga es un bug casi imposible de depurar.
7. Meter efectos secundarios en un `assert`: `assert(guardar());`; desaparece en produccion con `NDEBUG` y tu programa dejaria de guardar sin avisar. El `assert` solo observa, nunca actua.
8. Usar `assert` para validar datos del usuario o del mundo exterior. En produccion el `assert` no existe y la validacion se evapora. Datos externos = excepciones, siempre.
9. Logger sin `flush()` que pierde las ultimas lineas justo cuando el programa se cae. Esas lineas eran las mas importantes. Haz `flush` después de cada mensaje o al menos en los de nivel `ERROR`.
10. Abrir la bitacora en modo truncado (sin `ios::app`). Cada ejecucion borra el historial anterior, y el historial anterior es justo lo que necesitabas para comparar.
11. Registrar contraseñas, tokens o datos sensibles en el log. Los logs se copian, se respaldan y se leen por mucha gente: un secreto en el log es un secreto publicado.
12. Partir CSV por comas a mano sin respetar las comillas. "Garcia, Luis" se rompe en dos campos y todo lo que sigue se recorre. Usa el parser con estados o una biblioteca.
13. Serializar sin versionar el formato. Cuando agregues un campo al struct, los archivos viejos dejaran de leerse. Incluye desde el dia uno algo como "FORMATO=1" en la primera linea del archivo.
14. Confiar en que deserializar siempre recibe datos buenos. Valida cada campo y reporta el numero de linea: los archivos del mundo real siempre traen sorpresas.

Practica

Ejercicio 1

Disena una jerarquia de excepciones para una tienda en linea: clase base ErrorTienda y derivadas ProductoAgotado (con nombre del producto y stock disponible), PagoRechazado (con motivo) y DireccionInvalida. Escribe una funcion comprar(producto, cantidad, tarjetaValida) que lance la excepcion adecuada en cada caso y un main que las atrape de forma diferenciada.

Solucion:

```
#include <iostream>
#include <string>
#include <exception>
#include <map>

class ErrorTienda : public std::exception {
private:
    std::string mensaje;
public:
    explicit ErrorTienda(const std::string& m) : mensaje("[Tienda] " + m) {}
    const char* what() const noexcept override { return mensaje.c_str(); }
};

class ProductoAgotado : public ErrorTienda {
public:
    ProductoAgotado(const std::string& p, int stock)
        : ErrorTienda("'" + p + "' agotado: solo quedan " +
                      std::to_string(stock)) {}
};

class PagoRechazado : public ErrorTienda {
public:
    explicit PagoRechazado(const std::string& motivo)
        : ErrorTienda("Pago rechazado: " + motivo) {}
};

class DireccionInvalida : public ErrorTienda {
public:
    DireccionInvalida() : ErrorTienda("La direccion de envio es invalida") {}
};

std::map<std::string, int> stock = {"teclado", 5}, {"mouse", 0}};

void comprar(const std::string& producto, int cantidad, bool tarjetaValida,
             const std::string& direccion) {
    if (direccion.empty()) throw DireccionInvalida();
    auto it = stock.find(producto);
```

```

    int disponible = (it == stock.end()) ? 0 : it->second;
    if (disponible < cantidad) throw ProductoAgotado(producto, disponible);
    if (!tarjetaValida) throw PagoRechazado("fondos insuficientes");
    stock[producto] -= cantidad;
    std::cout << "Compra exitosa: " << cantidad << " x " << producto << std::endl;
}

int main() {
    try { comprar("mouse", 1, true, "Calle 123"); }
    catch (const ProductoAgotado& e) {
        std::cout << "Ofrecer alternativa. " << e.what() << std::endl;
    } catch (const ErrorTienda& e) {
        std::cout << "Error general: " << e.what() << std::endl;
    }

    try { comprar("teclado", 2, false, "Calle 123"); }
    catch (const PagoRechazado& e) {
        std::cout << "Pedir otra tarjeta. " << e.what() << std::endl;
    } catch (const ErrorTienda& e) {
        std::cout << "Error general: " << e.what() << std::endl;
    }

    try { comprar("teclado", 1, true, ""); }
    catch (const DireccionInvalida& e) {
        std::cout << "Pedir direccion. " << e.what() << std::endl;
    } catch (const ErrorTienda& e) {
        std::cout << "Error general: " << e.what() << std::endl;
    }

    try { comprar("teclado", 1, true, "Calle 123"); }
    catch (const ErrorTienda& e) {
        std::cout << "No deberia fallar: " << e.what() << std::endl;
    }
    return 0;
}

```

Explicacion: cada tipo de error permite una reaccion distinta en la interfaz: producto agotado sugiere alternativas, pago rechazado pide otra tarjeta, direccion invalida pide corregirla. Si todo fuera `runtime_error` con texto, el codigo tendria que comparar strings para decidir, lo cual es fragil. Los catch van de lo especifico a lo general, y el ultimo caso demuestra el camino feliz sin excepciones.

Ejercicio 2

Crea una clase CopiaSegura que en su constructor abre DOS archivos: origen (lectura) y destino (escritura). Si el destino no se puede abrir, el origen debe cerrarse solo (demuéstralo con mensajes). Agrega un metodo copiar() que pase el contenido linea por linea y lance si alguna linea supera 1000 caracteres.

Solucion:

```
#include <iostream>
#include <fstream>
#include <string>
#include <stdexcept>

class ArchivoLectura {
private:
    std::ifstream f;
    std::string nombre;
public:
    explicit ArchivoLectura(const std::string& ruta) : nombre(ruta) {
        f.open(ruta);
        if (!f.is_open()) throw std::runtime_error("No se pudo abrir " + ruta);
        std::cout << "[abierto lectura] " << nombre << std::endl;
    }
    ~ArchivoLectura() {
        if (f.is_open()) {
            f.close();
            std::cout << "[cerrado lectura] " << nombre << std::endl;
        }
    }
    std::ifstream& flujo() { return f; }
};

class CopiaSegura {
private:
    ArchivoLectura origen;    // recurso 1
    std::ofstream destino;    // recurso 2
public:
    CopiaSegura(const std::string& o, const std::string& d) : origen(o) {
        destino.open(d);
        if (!destino.is_open())
            throw std::runtime_error("No se pudo crear " + d);
        std::cout << "[abierto escritura] " << d << std::endl;
    }
    ~CopiaSegura() {
```

```

        if (destino.is_open()) destino.close();
        std::cout << "[cerrado escritura]" << std::endl;
    }
    void copiar() {
        std::string linea;
        int n = 0;
        while (std::getline(origen.flujo(), linea)) {
            n++;
            if (linea.size() > 1000)
                throw std::runtime_error("Linea " + std::to_string(n) +
                                           " demasiado larga");

            destino << linea << "\n";
        }
        std::cout << "Copiadas " << n << " lineas." << std::endl;
    }
};

int main() {
    // Caso que falla: destino imposible -> origen se cierra solo
    try {
        CopiaSegura c("parte-09c-u9.txt", "/ruta/imposible/salida.txt");
    } catch (const std::exception& e) {
        std::cout << "Atrapado: " << e.what() << std::endl;
    }
    return 0;
}

```

Explicacion: ArchivoLectura es un RAII que anuncia su apertura y cierre, asi ves la coreografia. Cuando el constructor de CopiaSegura lanza al fallar el destino, el miembro origen ya estaba construido y su destructor se ejecuta automaticamente: veras "[cerrado lectura]" aunque el objeto CopiaSegura jamas termino de construirse. Esa es la garantia basica en accion sin escribir ni una linea de limpieza.

Ejercicio 3

Escribe una funcion leerEnteroPositivo() que pida un numero, lance invalid_argument si no es numero y lance out_of_range si es negativo o mayor a 1,000,000. El main debe reintentar hasta 3 veces: cada reintento agrega contexto al mensaje usando relanzamiento (atrapa, envuelve el mensaje con el numero de intento y relanza con throw; ... o mejor: relanza una nueva excepcion con contexto). Implementa la version con nueva excepcion con contexto.

Solucion:

```
#include <iostream>
```

```

#include <string>
#include <stdexcept>

int leerEnteroPositivo() {
    std::string texto;
    std::getline(std::cin, texto);
    size_t pos = 0;
    int valor = 0;
    try {
        valor = std::stoi(texto, &pos);
    } catch (...) {
        throw std::invalid_argument("'" + texto + "' no es un numero");
    }
    if (pos != texto.size())
        throw std::invalid_argument("'" + texto + "' no es un numero entero limpio");
    if (valor < 0 || valor > 1000000)
        throw std::out_of_range("valor " + std::to_string(valor) +
                                " fuera de [0, 1000000]");
    return valor;
}

int pedirConReintentos(int maxIntentos) {
    for (int intento = 1; intento <= maxIntentos; intento++) {
        try {
            std::cout << "Dame un entero positivo (intento " << intento
                        << "/" << maxIntentos << "): ";
            return leerEnteroPositivo();
        } catch (const std::exception& e) {
            std::cout << " Fallo: " << e.what() << std::endl;
            if (intento == maxIntentos) {
                // Agrego contexto y relanzo como nueva excepcion
                throw std::runtime_error(
                    std::string("Se agotaron los intentos. Ultimo error: ") +
e.what());
            }
        }
    }
    throw std::runtime_error("Inalcanzable");
}

int main() {
    try {
        int v = pedirConReintentos(3);
        std::cout << "Valor aceptado: " << v << std::endl;
    } catch (const std::exception& e) {

```

```

        std::cout << "Error final: " << e.what() << std::endl;
    }
    return 0;
}

```

Explicacion: dos tecnicas combinadas. leerEnteroPositivo traduce errores de bajo nivel (stoi) a excepciones con significado; pedirConReintentos implementa la politica de reintentos y, al agotarlos, relanza una NUEVA excepcion que conserva el mensaje original como contexto ("Ultimo error: ..."). Esta es la alternativa a throw; cuando quieres cambiar el tipo o agregar informacion: envuelves, no rebanas. El main solo ve el resultado final con toda la historia.

Ejercicio 4

Extiende la clase Logger del ejemplo 4 para que cada linea incluya la fecha y hora actual (usa <ctime> y strftime con formato "%Y-%m-%d %H:%M:%S"). Además agrega un metodo estativo o funcion que devuelva un Logger global compartido, para no pasarlo como parametro por todo el programa.

Solucion:

```

#include <iostream>
#include <fstream>
#include <string>
#include <mutex>
#include <stdexcept>
#include <ctime>

enum class Nivel { DEBUG, INFO, AVISO, ERROR };

class Logger {
private:
    std::ofstream archivo;
    Nivel nivelMinimo;
    std::mutex mtx;

    static std::string ahora() {
        std::time_t t = std::time(nullptr);
        char buf[20];
        std::strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S",
                      std::localtime(&t));
        return buf;
    }
}

static std::string aTexto(Nivel n) {
    switch (n) {

```



```

        case Nivel::DEBUG: return "DEBUG";
        case Nivel::INFO:  return "INFO ";
        case Nivel::AVISO: return "AVISO";
        case Nivel::ERROR: return "ERROR";
    }
    return "?????";
}

Logger(const std::string& ruta, Nivel minimo) : nivelMinimo(minimo) {
    archivo.open(ruta, std::ios::app);
    if (!archivo.is_open())
        throw std::runtime_error("No se pudo abrir el log");
}

public:
    // Singleton simple: un solo logger global, creado al primer uso
    static Logger& instancia() {
        static Logger unico("app.log", Nivel::DEBUG);
        return unico;
    }

    void registrar(Nivel n, const std::string& mensaje) {
        if (n < nivelMinimo) return;
        std::lock_guard<std::mutex> candado(mtx);
        archivo << "[" << ahora() << "]"[" << aTexto(n) << "]" "
                << mensaje << "\n";
        archivo.flush();
    }

    void debug(const std::string& m) { registrar(Nivel::DEBUG, m); }
    void info(const std::string& m)  { registrar(Nivel::INFO, m); }
    void aviso(const std::string& m) { registrar(Nivel::AVISO, m); }
    void error(const std::string& m) { registrar(Nivel::ERROR, m); }
};

void procesoA() {
    Logger::instancia().info("procesoA iniciado");
    Logger::instancia().debug("procesoA detalle fino");
}

int main() {
    Logger::instancia().info("aplicacion iniciada");
    procesoA();
    Logger::instancia().error("algo salio mal");
    std::cout << "Revisa app.log: cada linea trae fecha y hora." << std::endl;
    return 0;
}

```

Explicacion: ahora() genera la estampa de tiempo con strftime; cada linea del log queda como "[2026-09-19 10:30:00][INFO] aplicacion iniciada", que es el formato estandar de la industria. El metodo instancia() implementa el patron singleton con variable estatica local: se crea una sola vez, al primer uso, y es seguro ante hilos desde C++11. Asi cualquier funcion registra con Logger::instancia().info(...) sin recibir el logger como parametro. Ojo: el singleton es practico pero dificulta las pruebas; en proyectos grandes se prefiere inyectar el logger como dependencia.

Ejercicio 5

Escribe la funcion inversa del parser CSV: std::string escaparCampoCSV(const std::string& campo), que devuelve el campo tal cual si no contiene coma, comilla ni salto de linea, y entrecomillado (duplicando comillas internas) si los contiene. Pruebala haciendo ida y vuelta: escapa varios campos, unelos con comas, parsealos con parsearCSV y verifica que recuperas los originales.

Solucion:

```
#include <iostream>
#include <string>
#include <vector>
#include <cassert>

// (Se asume parsearCSV del ejemplo 6 disponible arriba)

std::string escaparCampoCSV(const std::string& campo) {
    bool necesita = campo.find_first_of(",\"\\n") != std::string::npos;
    if (!necesita) return campo;
    std::string r = "\"";
    for (char c : campo) {
        if (c == '\"') r += "\\\"";
        else r += c;
    }
    r += "\"";
    return r;
}

// Copia minima del parser del ejemplo 6 para que el ejercicio sea autocontenido
std::vector<std::string> parsearCSV(const std::string& linea) {
    std::vector<std::string> campos;
    std::string actual;
    bool entreComillas = false;
    for (size_t i = 0; i < linea.size(); i++) {
```

```

        char c = linea[i];
        if (entreComillas) {
            if (c == '"') {
                if (i + 1 < linea.size() && linea[i + 1] == '"') { actual += '"'; i++;
            }

                else entreComillas = false;
            } else actual += c;
        } else if (c == '"') entreComillas = true;
        else if (c == ',') { campos.push_back(actual); actual.clear(); }
        else actual += c;
    }
    campos.push_back(actual);
    return campos;
}

int main() {
    std::vector<std::string> originales = {
        "Ana", "Garcia, Luis", "Dice \"hola\"", "Ciudad de Mexico", "simple"
    };

    std::string linea;
    for (size_t i = 0; i < originales.size(); i++) {
        if (i > 0) linea += ",";
        linea += escaparCampoCSV(originales[i]);
    }
    std::cout << "CSV generado: " << linea << std::endl;

    auto recuperados = parsearCSV(linea);
    assert(recuperados.size() == originales.size());
    for (size_t i = 0; i < originales.size(); i++) {
        assert(recuperados[i] == originales[i]);
        std::cout << "OK [" << recuperados[i] << "]" << std::endl;
    }
    std::cout << "Ida y vuelta perfecta." << std::endl;
    return 0;
}

```

Explicación: `escaparCampoCSV` y `parsearCSV` son inversas: una escribe siguiendo las reglas y la otra las interpreta. La prueba de ida y vuelta (round-trip) con `assert` es la forma profesional de verificar un formato: si escapar y parsear se cancelan mutuamente con datos difíciles (comas, comillas), el contrato está cumplido. Nota el uso legítimo de `assert` aquí: verifica una invariante interna de TU código durante el desarrollo, no datos del usuario.

Resumen de la expansion

- Diseña jerarquias de excepciones propias: base comun con mensaje guardado en miembro, `what()` `const noexcept override`, derivadas que arman su mensaje en el constructor. Atrapa de lo especifico a lo general.
- RAII con varios recursos funciona solo: los miembros ya contruidos se destruyen en orden inverso si el constructor lanza. Haz que cada miembro sea un RAII y la clase contenedora queda segura gratis.
- Relanza con `throw`; para conservar el tipo original; `throw e`; rebana. Usa `catch (...)` solo como red final o para limpiar antes de relanzar.
- `assert` para tus invariantes internas (desaparece en produccion); excepciones para errores del mundo exterior (siempre activas). Nunca efectos secundarios dentro de un `assert`.
- Una bitacora profesional tiene niveles (DEBUG, INFO, AVISO, ERROR), escribe en archivo en modo `append` con `flush`, y registra contexto sin secretos.
- Serializar es un contrato: define el formato, escapa los caracteres especiales, valida al leer y reporta el numero de linea.
- El CSV real respeta comillas: un campo entrecomillado puede contener comas y comillas duplicadas. El parser es una maquina de dos estados.
- Con esto cierras la Unidad 9 con herramientas de ingenieria real: tus programas ya no solo funcionan, aguantan.

UNIDAD 10: Nivel profesional

Llegaste a la última unidad, y es la que te convierte en programador de C++ moderno de verdad. Verás los punteros inteligentes (la forma segura de manejar memoria dinámica), construirás un proyecto final completo que integra todo lo aprendido en el libro y cerrarás con las buenas prácticas que usan los profesionales y una guía de qué estudiar después.

Si las unidades anteriores te dieron las piezas, esta te enseña a armar la máquina completa. El proyecto final no es un ejercicio más: es un sistema de gestión con clases, herencia, polimorfismo, STL, archivos y excepciones. Cuando lo termines, tendrás algo que mostrar con orgullo.

Al terminar sabrás: usar `unique_ptr` y `shared_ptr`, diseñar un sistema completo orientado a objetos con persistencia, aplicar buenas prácticas profesionales y planear tu siguiente paso en C++.

Explicación detallada

Clase 28: Punteros inteligentes

En la unidad 5 aprendiste `new` y `delete`, y el peligro de olvidar liberar la memoria. Los punteros inteligentes resuelven esto aplicando RAI: son objetos que se comportan como punteros pero liberan la memoria automáticamente cuando mueren. Viven en `<memory>` y hay dos principales.

`unique_ptr`: propiedad exclusiva. Un solo dueño, la memoria se libera cuando el `unique_ptr` muere. No se puede copiar (eso rompería la exclusividad), pero sí mover.

```
#include <memory>

std::unique_ptr<int> p = std::make_unique<int>(42);
std::cout << *p << std::endl;
// al salir del alcance, la memoria se libera sola. Sin delete.
```

`make_unique` es la forma moderna y segura de crearlos (evita fugas incluso si hay excepciones en la misma expresión). Úsalo siempre en vez de `new` directo.

`shared_ptr`: propiedad compartida. Varios dueños, la memoria se libera cuando muere el último. Lleva un contador interno de referencias.

```

auto a = std::make_shared<int>(100);
{
    auto b = a; // ahora hay 2 dueños
    std::cout << *b << std::endl;
} // b muere, queda 1 dueño
// a muere: se libera la memoria

```

El contador tiene un pequeño costo, así que usa `unique_ptr` por defecto y `shared_ptr` solo cuando realmente necesites compartir propiedad. Esa es la recomendación oficial de la comunidad C++.

¿Por qué son mejores que `new/delete` manual? Tres razones. Una: imposible olvidar el `delete`. Dos: seguros ante excepciones (si una excepción atraviesa el código, los destructores liberan todo). Tres: expresan intención (`unique` = "esto es mío", `shared` = "esto lo compartimos").

Regla moderna de C++: no uses `new` y `delete` en código nuevo salvo que tengas una razón muy específica. `unique_ptr`, `shared_ptr`, `vector` y `string` cubren casi todo. Los punteros crudos quedan para casos especiales (código de bajo nivel, interoperabilidad con C).

Un detalle importante con herencia y polimorfismo: los punteros inteligentes funcionan perfecto con clases base:

```

std::vector<std::unique_ptr<Animal>> zoo;
zoo.push_back(std::make_unique<Perro>("Firulais"));
zoo.push_back(std::make_unique<Gato>("Misu"));
// al morir el vector, todo se libera solo, con los destructores correctos

```

Ya lo viste en la unidad 7 sin explicarlo a fondo; ahora sabes por qué es la forma correcta.

Clase 29: Proyecto final - Sistema de biblioteca

Vamos a integrar todo: clases, herencia, polimorfismo, encapsulamiento, `vector`, `map`, archivos, excepciones y punteros inteligentes. Construiremos un sistema de gestión de biblioteca con libros físicos y digitales, préstamos y persistencia en archivo.

El diseño: clase abstracta `Material` (título, disponible, prestar/devolver virtuales), derivadas `LibroFisico` (ubicación en estantería) y `LibroDigital` (tamaño en MB, descargas). Clase `Biblioteca` que administra un `vector` de `unique_ptr<Material>`, con métodos para agregar, prestar, devolver, listar y guardar/cargar en archivo.

```

#include <iostream>
#include <string>

```

```

#include <vector>
#include <memory>
#include <fstream>
#include <stdexcept>

class Material {
protected:
    std::string titulo;
    std::string autor;
    bool prestado;

public:
    Material(std::string t, std::string a)
        : titulo(t), autor(a), prestado(false) {}
    virtual ~Material() {}

    virtual std::string tipo() const = 0;
    virtual std::string detalle() const = 0;

    void prestar() {
        if (prestado) throw std::runtime_error("'" + titulo + "' ya esta prestado");
        prestado = true;
    }

    void devolver() {
        if (!prestado) throw std::runtime_error("'" + titulo + "' no estaba prestado");
        prestado = false;
    }

    bool estaPrestado() const { return prestado; }
    std::string getTitulo() const { return titulo; }
    std::string getAutor() const { return autor; }

    void mostrar() const {
        std::cout << "[" << tipo() << "]" " << titulo << " - " << autor
            << " | " << detalle()
            << (prestado ? " | PRESTADO" : " | disponible") << std::endl;
    }
};

class LibroFisico : public Material {
private:
    std::string estanteria;

```

```

public:
    LibroFisico(std::string t, std::string a, std::string e)
        : Material(t, a), estanteria(e) {}

    std::string tipo() const override { return "Fisico"; }
    std::string detalle() const override { return "Estante: " + estanteria; }
    std::string getEstanteria() const { return estanteria; }
};

class LibroDigital : public Material {
private:
    double tamanoMB;
    int descargas;

public:
    LibroDigital(std::string t, std::string a, double mb)
        : Material(t, a), tamanoMB(mb), descargas(0) {}

    std::string tipo() const override { return "Digital"; }
    std::string detalle() const override {
        return std::to_string(tamanoMB) + " MB, " + std::to_string(descargas) + "
descargas";
    }

    void descargar() {
        descargas++;
        std::cout << "Descargando '" << titulo << "' (" << tamanoMB << " MB)..." <<
std::endl;
    }
};

class Biblioteca {
private:
    std::vector<std::unique_ptr<Material>> catalogo;
    std::string archivoDatos;

    int buscar(const std::string& titulo) const {
        for (size_t i = 0; i < catalogo.size(); i++) {
            if (catalogo[i]->getTitulo() == titulo) return i;
        }
        return -1;
    }

public:
    Biblioteca(const std::string& archivo) : archivoDatos(archivo) {

```



```

        cargar();
    }

    ~Biblioteca() {
        guardar();
    }

    void agregarFisico(const std::string& t, const std::string& a, const std::string&
e) {
        catalogo.push_back(std::make_unique<LibroFisico>(t, a, e));
    }

    void agregarDigital(const std::string& t, const std::string& a, double mb) {
        catalogo.push_back(std::make_unique<LibroDigital>(t, a, mb));
    }

    void prestar(const std::string& titulo) {
        int i = buscar(titulo);
        if (i < 0) throw std::runtime_error("Material no encontrado: " + titulo);
        catalogo[i]->prestar();
        std::cout << "Prestado: " << titulo << std::endl;
    }

    void devolver(const std::string& titulo) {
        int i = buscar(titulo);
        if (i < 0) throw std::runtime_error("Material no encontrado: " + titulo);
        catalogo[i]->devolver();
        std::cout << "Devuelto: " << titulo << std::endl;
    }

    void listar() const {
        if (catalogo.empty()) {
            std::cout << "Catalogo vacio." << std::endl;
            return;
        }
        for (const auto& m : catalogo) m->mostrar();
    }

    void guardar() const {
        std::ofstream out(archivoDatos);
        if (!out.is_open()) throw std::runtime_error("No se pudo guardar");
        for (const auto& m : catalogo) {
            out << m->tipo() << "|" << m->getTitulo() << "|"
                << m->getAutor() << "|" << m->estaPrestado() << "|"
                << m->detalle() << "\n";
        }
    }

```

```

    }
}

void cargar() {
    std::ifstream in(archivoDatos);
    if (!in.is_open()) return;
    std::string linea;
    while (std::getline(in, linea)) {
        // Formato: tipo|titulo|autor|prestado|detalle
        // (parsing simplificado para el ejemplo)
        std::cout << "(registro cargado: " << linea.substr(0, 30) << "...)" <<
std::endl;
    }
}

};

int main() {
    try {
        Biblioteca biblio("biblioteca.txt");

        biblio.agregarFisico("Cien años de soledad", "Garcia Marquez", "A-12");
        biblio.agregarDigital("Curso Completo de C++", "Lucas EGA Academy", 8.5);
        biblio.agregarFisico("El principito", "Saint-Exupery", "B-03");

        std::cout << "=== CATALOGO ===" << std::endl;
        biblio.listar();

        std::cout << "\n=== PRESTAMOS ===" << std::endl;
        biblio.prestar("El principito");
        biblio.prestar("El principito"); // lanza excepción
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
    // Al salir, el destructor de Biblioteca guarda automáticamente
    return 0;
}

```

Analiza lo que integra: clase abstracta Material con métodos virtuales puros (polimorfismo), dos derivadas con sus detalles, encapsulamiento (prestado solo cambia por prestar/devolver validados), vector de unique_ptr (memoria segura), excepciones para errores de negocio, persistencia en archivo con guardado automático en el destructor (RAII aplicado a la aplicación completa).

Este es el nivel de sistema que se espera de un programador junior de C++: no código suelto, sino objetos que colaboran, errores manejados y datos persistentes. Si entiendes este proyecto línea por línea, estás listo.

Clase 30: Buenas prácticas y qué sigue

Cierra el curso con los hábitos que distinguen a un profesional.

Compila con advertencias siempre: `g++ -Wall -Wextra -std=c++17`. Trata las advertencias como errores: configúralo con `-Werror` cuando puedas. La mitad de los bugs se detectan en compilación si dejas que el compilador hable.

Nombres claros: variables y funciones en camelCase o snake_case (elige uno y sé consistente), clases en PascalCase, constantes en MAYUSCULAS. Un nombre como `calcularPromedioFinal` vale más que mil comentarios.

const todo lo posible: parámetros que no cambian (`const&`), métodos que no modifican (`const`), variables fijas (`const`). El compilador se vuelve tu aliado contra errores tontos.

Inicializa siempre: usa llaves `{}` y listas de inicialización. Nunca dejes variables ni atributos sin valor inicial.

Prefiere la biblioteca estándar: `vector` antes que `new[]`, `string` antes que `char*`, algoritmos de `<algorithm>` antes que ciclos manuales. Está probada, es rápida y todo programador de C++ la entiende.

RAII para recursos: cada recurso (memoria, archivo, conexión) debe pertenecer a un objeto que lo libere en su destructor. Punteros inteligentes, `fstream` que se cierra solo, `lock_guard` en hilos. Si escribes `delete` a mano, pregúntate si un `unique_ptr` lo haría mejor.

Funciones cortas y con un solo propósito. Clases cohesivas: cada clase, una responsabilidad. Evita la herencia profunda; prefiere composición. Comenta el porqué, no el qué: el código ya dice qué hace.

Prueba tu código: escribe programas pequeños que verifiquen cada clase (pruebas manuales al inicio, frameworks como Catch2 o GoogleTest después). Depura con herramientas: `gdb` para depuración, `valgrind` o sanitizers (`-fsanitize=address`) para detectar errores de memoria. Aprende a leer los mensajes del compilador: el primer error suele ser el real, los demás son consecuencia.

¿Y después de este libro? El camino natural: profundiza en la STL (más contenedores y algoritmos), aprende plantillas avanzadas, estudia estructuras de datos y algoritmos clásicos implementándolos en C++, explora programación multihilo (`std::thread`), y construye proyectos:

un juego con SFML, un servidor simple, un sistema completo como el de la biblioteca pero más grande. Los cursos de la academia y la documentación en cppreference.com serán tus compañeros.

Ejemplos paso a paso

Ejemplo: unique_ptr vs shared_ptr

```
#include <iostream>
#include <memory>

class Recurso {
public:
    Recurso() { std::cout << "Recurso adquirido." << std::endl; }
    ~Recurso() { std::cout << "Recurso liberado." << std::endl; }
    void usar() { std::cout << "Usando recurso." << std::endl; }
};

void demoUnique() {
    std::cout << "--- unique_ptr ---" << std::endl;
    auto p = std::make_unique<Recurso>();
    p->usar();
    // se libera solo al salir
}

void demoShared() {
    std::cout << "--- shared_ptr ---" << std::endl;
    auto a = std::make_shared<Recurso>();
    {
        auto b = a;
        std::cout << "Dueños: " << a.use_count() << std::endl;
        b->usar();
    }
    std::cout << "Dueños: " << a.use_count() << std::endl;
    // se libera cuando a muera
}

int main() {
    demoUnique();
    demoShared();
    return 0;
}
```

Observa el orden de los mensajes: "Recurso liberado" aparece automáticamente al salir de cada función, sin un solo delete. use_count() muestra cuántos dueños hay. Ejecútalo y sigue el ciclo de vida con tus ojos.

Práctica

Ejercicio 1

Convierte el ejemplo de memoria dinámica de la unidad 5 (ejercicio 4) a usar std::vector en vez de new[]/delete[]. Compara ambos códigos.

Solución:

```
#include <iostream>
#include <vector>

int main() {
    int n;
    std::cout << "Cuantos numeros? ";
    std::cin >> n;

    std::vector<double> datos(n);
    double suma = 0;
    for (int i = 0; i < n; i++) {
        std::cout << "Numero " << i + 1 << ": ";
        std::cin >> datos[i];
        suma += datos[i];
    }
    std::cout << "Promedio: " << suma / n << std::endl;
    return 0; // sin delete: el vector se limpia solo
}
```

Explicación: el vector acepta el tamaño en ejecución igual que new[], pero se libera solo. Menos código, cero riesgo de fugas. Esta es la forma moderna; reserva new[] para cuando una API te lo exija.

Ejercicio 2

Crea una clase Tarea (descripcion, completada) y un gestor con vector<unique_ptr<Tarea>> que permita agregar, completar por índice y listar. Usa make_unique.

Solución:

```

#include <iostream>
#include <string>
#include <vector>
#include <memory>

class Tarea {
private:
    std::string descripcion;
    bool completada;
public:
    Tarea(const std::string& d) : descripcion(d), completada(false) {}
    void completar() { completada = true; }
    void mostrar(int i) const {
        std::cout << i << ". [" << (completada ? "x" : " ") << "] "
            << descripcion << std::endl;
    }
};

int main() {
    std::vector<std::unique_ptr<Tarea>> tareas;
    tareas.push_back(std::make_unique<Tarea>("Estudiar C++"));
    tareas.push_back(std::make_unique<Tarea>("Hacer ejercicio"));
    tareas.push_back(std::make_unique<Tarea>("Leer 30 min"));

    tareas[1]->completar();

    for (size_t i = 0; i < tareas.size(); i++) {
        tareas[i]->mostrar(i + 1);
    }
    return 0;
}

```

Explicación: `unique_ptr` en vector es el patrón estándar para colecciones polimórficas o de objetos no copiables. -> accede al objeto a través del puntero inteligente con sintaxis natural.

Ejercicio 3

Extiende el sistema de biblioteca: agrega la clase `Revista` (hereda de `Material`, con número de edición) y un método `buscarPorAutor` en `Biblioteca` que devuelva los títulos coincidentes.

Solución:

```

#include <iostream>
#include <string>
#include <vector>
#include <memory>

// (Se asume la clase Material del ejemplo de la clase 29)

class Revista : public Material {
private:
    int edicion;
public:
    Revista(std::string t, std::string a, int e)
        : Material(t, a), edicion(e) {}
    std::string tipo() const override { return "Revista"; }
    std::string detalle() const override {
        return "Edicion " + std::to_string(edicion);
    }
};

// En Biblioteca, agregar:
std::vector<std::string> buscarPorAutor(const std::string& autor) const {
    std::vector<std::string> resultados;
    for (const auto& m : catalogo) {
        if (m->getAutor() == autor) resultados.push_back(m->getTitulo());
    }
    return resultados;
}

```

Explicación: agregar un tipo nuevo no tocó Material ni el resto del sistema: herencia y polimorfismo en acción. buscarPorAutor devuelve valores (strings), no punteros: seguro y simple. Para usarlo, agrega el método dentro de la clase Biblioteca del ejemplo anterior.

Ejercicio 4

Escribe un programa que demuestre RAII con archivos: crea una clase Registro que abre un archivo en el constructor, tiene un método escribir(linea) y lo cierra en el destructor. Provoca una excepción a la mitad y verifica que el archivo quede bien cerrado.

Solución:

```

#include <iostream>
#include <fstream>
#include <string>
#include <stdexcept>

class Registro {
private:
    std::ofstream archivo;
public:
    Registro(const std::string& nombre) {
        archivo.open(nombre, std::ios::app);
        if (!archivo.is_open()) throw std::runtime_error("No se pudo abrir");
        std::cout << "Registro abierto." << std::endl;
    }
    ~Registro() {
        if (archivo.is_open()) archivo.close();
        std::cout << "Registro cerrado." << std::endl;
    }
    void escribir(const std::string& linea) {
        archivo << linea << "\n";
    }
};

void procesoRiesgoso() {
    Registro r("log.txt");
    r.escribir("Inicio del proceso");
    throw std::runtime_error("Algo fallo a la mitad");
    r.escribir("Esto nunca se escribe");
}

int main() {
    try {
        procesoRiesgoso();
    } catch (const std::exception& e) {
        std::cout << "Atrapado: " << e.what() << std::endl;
    }
    std::cout << "Fin." << std::endl;
    return 0;
}

```

Explicación: aunque la excepción interrumpa procesoRiesgoso, el destructor de r se ejecuta durante el "desenrollado" y cierra el archivo. Verás "Registro cerrado" antes de "Atrapado". Sin RAII, el archivo quedaría abierto. Esta garantía es la razón por la que C++ moderno ama las excepciones.

Ejercicio 5

Refactoriza cualquier programa de las unidades 1-5 que use arreglos crudos o new/delete, aplicándole: vector en vez de arreglos, string donde aplique, const donde aplique y nombres claros. Explica cada cambio.

Solución: ejemplo con el ejercicio 4 de la unidad 5 (promedio dinámico), ya refactorizado en el ejercicio 1 de esta unidad. Los cambios aplicados: new double[n]/delete[] -> std::vector<double> datos(n) (memoria segura y automática); se eliminó el puntero crudo; el código se acortó y ya no puede fugar memoria. La lección general: cada vez que veas new/delete en tu código, pregúntate qué contenedor o puntero inteligente lo reemplaza. Con el tiempo, este reflejo se vuelve automático y tu código se vuelve profesional solo.

Resumen de la unidad

- unique_ptr: un solo dueño, se libera solo. Úsalo por defecto con make_unique. shared_ptr: varios dueños con conteo; solo cuando necesites compartir.
- Evita new/delete manual en código nuevo: vector, string y punteros inteligentes cubren casi todo.
- Un sistema real integra: clases abstractas, herencia, polimorfismo, STL, archivos, excepciones y RAII.
- Buenas prácticas: advertencias al compilar, nombres claros, const, inicialización siempre, prefiere la STL, funciones cortas, comenta el porqué.
- Herramientas: gdb, sanitizers (-fsanitize=address), pruebas constantes.
- Siguiente paso: STL avanzada, multihilo, estructuras de datos clásicas y proyectos grandes. cppreference.com es tu referencia oficial.
- ¡Felicidades! Completaste el curso completo de C++. Ahora demuestra lo aprendido en el examen final.

Profundización: hilos, depuración y proyectos reales

Para cerrar el libro, una mirada a lo que sigue: programación multihilo básica, depuración con gdb, automatización de compilación con Make y una lista de proyectos para seguir creciendo. Esto no te hace experto en estos temas, pero te deja la puerta abierta y el mapa en la mano.

Multihilo básico: std::thread

Los procesadores modernos tienen varios núcleos. std::thread (cabecera <thread>) permite usarlos:

```
#include <iostream>
#include <thread>
#include <vector>

void trabajar(int id) {
    std::cout << "Hilo " << id << " trabajando." << std::endl;
}

int main() {
    std::vector<std::thread> hilos;
    for (int i = 0; i < 4; i++) {
        hilos.emplace_back(trabajar, i);
    }
    for (auto& h : hilos) {
        h.join(); // espera a que termine
    }
    std::cout << "Todos terminaron." << std::endl;
    return 0;
}
```

Cada thread ejecuta una función en paralelo. join() espera su fin (olvidarlo termina el programa abruptamente). El peligro: dos hilos tocando los mismos datos causan "carreras" (race conditions). La solución es std::mutex: un candado que solo un hilo puede tomar a la vez.

```
#include <mutex>
std::mutex mtx;
int contador = 0;

void incrementar() {
    for (int i = 0; i < 100000; i++) {
        std::lock_guard<std::mutex> candado(mtx); // RAII: se libera solo
        contador++;
    }
}
```

lock_guard toma el mutex al crearse y lo suelta al destruirse: RAII aplicado a hilos. Sin él, dos hilos incrementando a la vez pierden cuentas. El multihilo es un mundo: esto es solo la entrada.

Depuración con gdb

Cuando las impresiones no bastan, gdb te deja ejecutar paso a paso:

```
g++ -g programa.cpp -o programa # -g incluye información de depuración
gdb ./programa
```

Comandos esenciales dentro de gdb: break main (pausa en main), run (ejecuta), next (siguiente línea sin entrar a funciones), step (entra a la función), print x (muestra variable), continue (sigue hasta el próximo break), backtrace (muestra la pila de llamadas al caerse), quit (salir).

El flujo típico: el programa se cae, corres en gdb, backtrace te dice exactamente en qué línea murió y con qué valores. Es diez veces más rápido que adivinar.

Automatiza con Make

Compilar a mano funciona con 2 archivos; con 20 es un infierno. Un Makefile automatiza:

```
CXX = g++
FLAGS = -Wall -Wextra -std=c++17

programa: main.o util.o
$(CXX) $(FLAGS) main.o util.o -o programa

main.o: main.cpp util.h
$(CXX) $(FLAGS) -c main.cpp

util.o: util.cpp util.h
$(CXX) $(FLAGS) -c util.cpp

clean:
rm -f *.o programa
```

make compila solo lo que cambió; make clean limpia. La indentación con tabulador es obligatoria. Para proyectos grandes existe CMake, que genera Makefiles portables: es el estándar de la industria y vale la pena aprenderlo después.

Lista de verificación de código profesional

Antes de decir "terminé", revisa:

- ¿Compila sin advertencias con -Wall -Wextra?

- ¿Todas las variables están inicializadas?
- ¿Los recursos se liberan solos (RAII) o hay delete manuales?
- ¿Las funciones son cortas y con nombres claros?
- ¿Hay números mágicos que deberían ser constantes con nombre?
- ¿Los errores posibles están manejados (archivos, entradas, divisiones)?
- ¿Probé los casos extremo: vacío, cero, negativo, muy grande?
- ¿El código se entiende sin explicármelo en voz alta?

Los equipos profesionales hacen "code review": otro programador lee tu código buscando justo esto. Acostúmbrate a revisar el tuyo con ojos de extraño.

Proyectos para seguir creciendo

1. Juego de Snake en consola: ciclos, entrada, vectores, tiempo real básico.
2. Sistema de inventario con archivos: CRUD completo, búsquedas, reportes.
3. Analizador de texto: cuenta palabras, frecuencias, estadísticas con map.
4. Agenda con recordatorios: fechas, persistencia, búsquedas.
5. Juego de Blackjack: clases Carta, Baraja, Jugador; aleatoriedad y lógica.
6. Servidor de chat simple con hilos: un hilo por cliente (avanzado).
7. Tu propia versión del sistema de biblioteca, con préstamos por fecha y multas.

Cada proyecto te enseñará más que diez capítulos. Elige uno, térmalo, y luego otro.

Dónde seguir aprendiendo

- cppreference.com: la referencia oficial y completa (en inglés, precisa).
- learncpp.com: tutorial gratuito excelente, muy didáctico.
- Los cursos avanzados de Lucas EGA Academy en lucasegaacademy.com.
- Libros: "Programming: Principles and Practice Using C++" (Stroustrup) para fundamentos; "Effective C++" (Meyers) cuando ya programes con soltura.
- Práctica: HackerRank, LeetCode y Advent of Code tienen problemas en C++.

Práctica adicional

Ejercicio A

Escribe un programa con 2 hilos que incrementen un contador compartido 100,000 veces cada uno, primero SIN mutex y luego CON mutex. Compara los resultados finales. ¿Por qué difieren?

Solución:

```
#include <iostream>
#include <thread>
#include <mutex>

int contadorSin = 0;
int contadorCon = 0;
std::mutex mtx;

void sinMutex() {
    for (int i = 0; i < 100000; i++) contadorSin++;
}

void conMutex() {
    for (int i = 0; i < 100000; i++) {
        std::lock_guard<std::mutex> l(mtx);
        contadorCon++;
    }
}

int main() {
    std::thread t1(sinMutex), t2(sinMutex);
    t1.join(); t2.join();
    std::cout << "Sin mutex: " << contadorSin << " (esperado 200000)" << std::endl;

    std::thread t3(conMutex), t4(conMutex);
    t3.join(); t4.join();
    std::cout << "Con mutex: " << contadorCon << " (esperado 200000)" << std::endl;
    return 0;
}
```

Explicación: sin mutex, los incrementos se pisan (leer-modificar-escribir no es atómico) y el total queda menor a 200,000. Es una "carrera de datos": el bug más traicionero del multihilo porque a veces "funciona". El mutex serializa el acceso y el resultado es exacto.

Ejercicio B

Crea un Makefile para un proyecto de 2 archivos (main.cpp + mate.cpp con mate.h que declara `int factorial(int)`). Incluye objetivo `clean`.

Solución:

```
CXX = g++
FLAGS = -Wall -Wextra -std=c++17

programa: main.o mate.o
    $(CXX) $(FLAGS) main.o mate.o -o programa

main.o: main.cpp mate.h
    $(CXX) $(FLAGS) -c main.cpp

mate.o: mate.cpp mate.h
    $(CXX) $(FLAGS) -c mate.cpp

clean:
    rm -f *.o programa
```

Explicación: cada `.o` depende de sus fuentes; si cambias solo `mate.cpp`, solo se recompila `mate.o`. El tabulador al inicio de cada receta es obligatorio (no espacios). Prueba: `make`, toca un archivo, `make` de nuevo y observa que solo recompila lo necesario.

Ejercicio C

Aplica la lista de verificación a tu programa del sistema de biblioteca (o a cualquier programa largo que hayas escrito) y anota al menos 5 mejoras concretas.

Solución: ejemplo de hallazgos típicos: agregar `-Wall -Wextra` al compilar; validar entradas del usuario antes de convertir; usar `const` en métodos de solo lectura; reemplazar números mágicos por constantes con nombre; verificar `is_open()` en archivos; agregar mensajes de error claros. El valor está en el hábito: revisar tu propio código con esta lista, siempre.

Ejercicio D

Diseña (en papel o comentarios, sin programar todo) las clases para el juego de Blackjack: qué atributos y métodos tendría Carta, Baraja, Jugador y Juego. Define quién es responsable de qué.

Solución: Carta: palo, valor, métodos valorBlackjack() y mostrar(). Baraja: vector<Carta>, métodos barajar() y repartir(). Jugador: nombre, vector<Carta> mano, métodos pedirCarta(), puntaje() y mostrarMano(). Juego: Baraja, Jugador jugador, Jugador crupier, método jugar() con el ciclo del juego. Responsabilidades: Carta solo se describe; Baraja administra el mazo; Jugador administra su mano; Juego orquesta las reglas. Diseñar antes de programar evita reescrituras: 30 minutos de diseño ahorran 3 horas de código.

Expansion Unidad 10: memoria avanzada, rendimiento y tu carrera profesional

Ya manejas unique_ptr y shared_ptr, construiste un sistema completo y conoces las buenas practicas. Esta expansion te lleva a los temas que separan a quien "sabe C++" de quien lo domina en serio: el tercer puntero inteligente (weak_ptr), limpieza personalizada de recursos, mover objetos en vez de copiarlos, medir el rendimiento antes de optimizar, y luego la parte que ningun compilador te ensena: como prepararte para entrevistas, como entrar al mundo open source y que estudiar los proximos 90 dias.

weak_ptr: el puntero que observa sin poseer

El problema que shared_ptr no puede resolver solo

shared_ptr cuenta duenos y libera cuando el ultimo muere. Pero hay un caso donde ese conteo se vuelve una trampa: dos objetos que se apuntan mutuamente con shared_ptr. A apunta a B y B apunta a A; cuando el resto del programa los suelta, cada uno todavia tiene un dueno (el otro), el conteo nunca llega a cero y la memoria jamas se libera. Es una fuga silenciosa hecha de punteros "seguros".

weak_ptr es la salida: un puntero que observa un objeto manejado por shared_ptr sin contarse como dueno. No mantiene vivo al objeto, no impide su destruccion, pero te permite preguntar "sigues vivo?" y, si la respuesta es si, obtener un shared_ptr temporal para usarlo con seguridad.

Ejemplo completo 1: cache que no secuestra memoria

Una cache guarda resultados costosos para reutilizarlos. El dilema: si la cache guarda `shared_ptr`, los objetos viven para siempre aunque nadie los use (la cache los secuestra). Si guarda punteros crudos, pueden quedar colgando. `weak_ptr` es la respuesta exacta: la cache recuerda los objetos sin adueñarse de ellos.

```
#include <iostream>
#include <memory>
#include <string>
#include <unordered_map>

// Recurso costoso de crear (una textura, un documento, una conexion)
class Recurso {
private:
    std::string nombre;
public:
    explicit Recurso(const std::string& n) : nombre(n) {
        std::cout << " [creando recurso '" << nombre << "'" << std::endl;
    }
    ~Recurso() {
        std::cout << " [destruyendo recurso '" << nombre << "'" << std::endl;
    }
    void usar() const {
        std::cout << " usando '" << nombre << "'" << std::endl;
    }
};

class CacheRecurso {
private:
    // weak_ptr: la cache observa, no posee
    std::unordered_map<std::string, std::weak_ptr<Recurso>> cache;
public:
    std::shared_ptr<Recurso> obtener(const std::string& nombre) {
        auto it = cache.find(nombre);
        if (it != cache.end()) {
            std::shared_ptr<Recurso> vivo = it->second.lock();
            if (vivo) {
                std::cout << "[cache] hit: '" << nombre << "'" << std::endl;
                return vivo;
            }
            std::cout << "[cache] expirado, recreando '" << nombre << "'" <<
std::endl;
        } else {
            std::cout << "[cache] miss: '" << nombre << "'" << std::endl;
        }
        auto nuevo = std::make_shared<Recurso>(nombre);
```



```

        cache[nombre] = nuevo; // se guarda debil: no aumenta el conteo
        return nuevo;
    }

    void limpiarExpirados() {
        for (auto it = cache.begin(); it != cache.end(); ) {
            if (it->second.expired()) {
                std::cout << "[cache] limpiando entrada expirada '"
                    << it->first << "'" << std::endl;
                it = cache.erase(it);
            } else {
                ++it;
            }
        }
    }
};

int main() {
    CacheRecursos cache;

    {
        auto r1 = cache.obtener("logo.png"); // miss: se crea
        r1->usar();
        auto r2 = cache.obtener("logo.png"); // hit: el mismo objeto
        std::cout << "Mismo objeto? " << (r1 == r2 ? "si" : "no") << std::endl;
    } // r1 y r2 mueren: el recurso se destruye, la cache no lo retiene

    auto r3 = cache.obtener("logo.png"); // expirado: se recrea
    r3->usar();

    cache.limpiarExpirados();
    return 0;
}

```

Sigue el ciclo de vida en la salida: el primer obtener crea el recurso, el segundo lo reutiliza (hit), al cerrar el bloque los dos `shared_ptr` mueren y el recurso se destruye aunque la cache lo "recuerde", y el tercer obtener detecta el `weak_ptr` expirado y lo recrea. La cache acelera sin secuestrar memoria: exactamente lo que quieres.

lock() y expired(): el protocolo seguro

No puedes usar un `weak_ptr` directamente (no tiene operador `*` ni `->`): primero lo conviertes con `lock()`, que devuelve un `shared_ptr` valido si el objeto sigue vivo o uno vacio si ya murio. El patron es siempre el mismo:

```
std::weak_ptr<Recurso> observador = algunShared;
if (auto ptr = observador.lock()) { // ptr es shared_ptr temporal
    ptr->usar(); // seguro: mientras ptr viva, el objeto vive
} else {
    std::cout << "El objeto ya no existe" << std::endl;
}
```

Esto elimina la carrera clasica del puntero crudo observado ("lo revise y existia, pero se destruyo antes de usarlo"): entre el lock() y el uso, el shared_ptr temporal mantiene vivo al objeto. expired() sirve para limpieza periodica, como en limpiarExpirados, pero para usar el objeto siempre prefiere lock().

Usos tipicos de weak_ptr en el mundo real: caches (como la del ejemplo), el puntero "hijo -> padre" en arboles (el padre posee a los hijos con shared_ptr o unique_ptr, los hijos miran al padre con weak_ptr), y observadores o callbacks que no deben prolongar la vida de lo observado.

Custom deleters: limpieza a tu medida

Cuando delete no es la respuesta

unique_ptr sabe llamar a delete cuando muere. Pero no todos los recursos se liberan con delete: un FILE* de C se cierra con fclose, una conexion de una biblioteca C tiene su propia funcion de cierre, un arreglo creado con new[] necesita delete[]. El deleter personalizado te deja decirle al unique_ptr exactamente como limpiar.

Hay dos formas: pasar una funcion (o puntero a funcion) como segundo parametro de plantilla, o usar una lambda. La lambda sin capturas es la mas comoda y no cuesta nada en tiempo de ejecucion.

Ejemplo completo 2: unique_ptr que cierra archivos C y rastrea liberaciones

```

#include <iostream>
#include <memory>
#include <cstdio>
#include <string>

int main() {
    // FILE* de C administrado por unique_ptr: fclose se llama solo,
    // incluso si una excepcion atraviesa este codigo
    auto archivo = std::unique_ptr<FILE, decltype(&std::fclose)>(
        std::fopen("salida.txt", "w"), &std::fclose);

    if (!archivo) {
        std::cerr << "No se pudo abrir el archivo" << std::endl;
        return 1;
    }
    std::fprintf(archivo.get(), "Hola desde C con limpieza de C++\n");
    std::cout << "Escrito. Al salir del main, fclose se llama solo." << std::endl;

    // Deleter con lambda: registra que se libero y con que valor
    auto rastreador = std::unique_ptr<int, void(*)>(int*)>(
        new int(42),
        [](int* p) {
            std::cout << "Liberando entero con valor " << *p << std::endl;
            delete p;
        });
    std::cout << "Valor: " << *rastreador << std::endl;

    // Arreglos: unique_ptr<T[]> usa delete[] automaticamente
    std::unique_ptr<int[]> arreglo(new int[5]);
    for (int i = 0; i < 5; i++) arreglo[i] = i * 10;
    std::cout << "arreglo[3] = " << arreglo[3]
        << " (delete[] automatico, sin deleter manual)" << std::endl;

    return 0;
}

```

Tres ideas en un ejemplo. Una: puedes envolver APIs de C (FILE*, sockets, handles del sistema) en unique_ptr y obtener seguridad ante excepciones gratis: el recurso de C ahora vive con reglas de C++. Dos: el deleter puede hacer mas que liberar (registrar, medir, notificar); es un punto de extension limpio. Tres: para arreglos usa unique_ptr<T[]> o, mejor aun, vector; nunca mezcles new[] con delete a secas, que es comportamiento indefinido.

Nota de diseño: si el deleter es una lambda SIN capturas, no aumenta el tamaño del `unique_ptr`. Si captura variables, el `unique_ptr` crece para guardarlas. Para la mayoría de los casos, la función o la lambda sin capturas es lo correcto.

Move semantics: mover en vez de copiar

Copiar vs mover: la mudanza

Copiar un vector de un millón de strings significa duplicar cada string: lento y costoso. Pero muchas veces no necesitas dos copias: necesitas trasladar el contenido de un objeto a otro, como una mudanza donde te llevas los muebles en vez de comprarlos de nuevo. Eso es la semántica de movimiento: transferir los recursos internos (punteros, manejadores) del objeto origen al destino, dejando al origen en un estado válido pero vacío.

`std::move` no mueve nada por sí mismo: es una autorización. Le dice al compilador "este objeto ya no lo necesito; puedes canibalizarlo". La función o constructor que lo recibe decide si mueve (barato) o copia (caro). Los tipos de la STL (`vector`, `string`, `unique_ptr`) saben moverse; los tipos simples como `int` se copian igual porque moverlos no ahorra nada.

Ejemplo completo 3: medir copiar vs mover con tus propios ojos

```
#include <iostream>
#include <vector>
#include <string>
#include <chrono>

std::vector<std::string> crearDatos(int n) {
    std::vector<std::string> datos;
    datos.reserve(n);
    for (int i = 0; i < n; i++) {
        datos.push_back("registro_numero_" + std::to_string(i) +
                        "_con_texto_de_relleno_para_que_pese");
    }
    return datos; // NO se copia al retornar: se mueve (o se elide)
}

// Recibe POR VALOR: el argumento se copia... o se mueve, según como lo llames
long medirMs(std::vector<std::string> copia) {
    auto inicio = std::chrono::high_resolution_clock::now();
    volatile size_t total = 0;
    for (const auto& s : copia) total += s.size();
    auto fin = std::chrono::high_resolution_clock::now();
```

```

        (void)total;
        return std::chrono::duration_cast<std::chrono::milliseconds>(fin -
inicio).count());
    }

int main() {
    const int N = 200000;
    std::vector<std::string> datos = crearDatos(N);
    std::cout << "Creados " << datos.size() << " registros." << std::endl;

    { // Caso A: pasar por valor COPIANDO
        std::vector<std::string> copia = datos;
        auto inicio = std::chrono::high_resolution_clock::now();
        medirMs(copia); // duplica el vector completo al entrar
        auto fin = std::chrono::high_resolution_clock::now();
        long ms = std::chrono::duration_cast<std::chrono::milliseconds>(
            fin - inicio).count();
        std::cout << "Pasar por valor (copia): " << ms << " ms" << std::endl;
    }

    { // Caso B: pasar por valor MOVIENDO
        std::vector<std::string> mio = datos; // copia para no vaciar 'datos'
        auto inicio = std::chrono::high_resolution_clock::now();
        medirMs(std::move(mio)); // transfiere punteros: casi gratis
        auto fin = std::chrono::high_resolution_clock::now();
        long ms = std::chrono::duration_cast<std::chrono::milliseconds>(
            fin - inicio).count();
        std::cout << "Pasar con std::move: " << ms << " ms" << std::endl;
        std::cout << "'mio' quedo con " << mio.size()
            << " elementos (valido pero vacio)" << std::endl;
    }

    std::cout << "'datos' intacto: " << datos.size() << " elementos." << std::endl;
    return 0;
}

```

Ejecuta y compara: el caso A puede tardar cientos de milisegundos duplicando 200,000 strings, mientras el caso B es casi instantaneo porque solo transfiere tres punteros internos (inicio, fin, capacidad). Esa diferencia es la razon por la que devolver vectores por valor desde funciones es barato en C++ moderno: el return mueve, no copia.

Reglas de oro del movimiento

1. Después de `std::move(x)`, no uses `x` excepto para reasignarlo o destruirlo. Queda en estado valido pero indeterminado (en la practica, vacio). Leerlo es legal pero su contenido es sorpresa.
2. No pongas `std::move` en un return de variable local: `return datos;` ya mueve (o elide la copia) solo. Escribir `return std::move(datos);` puede impedir la elision y salir peor.
3. `unique_ptr` SOLO se puede mover, nunca copiar: esa es su gracia (propiedad exclusiva). Si un codigo necesita copiar un `unique_ptr`, el diseno esta mal.
4. Para parametros de entrada que solo se leen: `const&`. Para parametros que la funcion debe quedarse (hundir): por valor + `std::move` adentro. Es el idioma moderno.
5. Mover no es gratis para todo: un `array<int, 1000>` se mueve copiando (no tiene punteros internos que transferir). El movimiento brilla con tipos que manejan recursos dinamicos.

Perfilado basico: mide antes de optimizar

El programador es mal adivino del rendimiento

Casi todos los programadores, incluidos los experimentados, fallan al adivinar donde esta lento su programa. Optimizan el ciclo que "se ve pesado" mientras el verdadero cuello de botella esta en una concatenacion de strings tres funciones mas alla. La regla de oro del rendimiento: mide primero, optimiza después, y vuelve a medir para comprobar que la optimizacion sirvio.

La herramienta mas simple es `<chrono>`: marcas el tiempo antes y después, repites varias veces y comparas. No es un perfilador profesional (esos son `perf`, `gprof` o los de tu IDE), pero resuelve el 80% de los casos y siempre esta disponible.

Ejemplo completo 4: cazar un cuello de botella real

El sospechoso habitual numero uno en C++: concatenar strings en un ciclo sin reservar memoria. Cada `+=` puede realocar y copiar todo lo acumulado: un costo cuadratico disfrazado de codigo inocente.

```
#include <iostream>
#include <string>
#include <chrono>
#include <vector>

// Version ingenua: puede realocar en cada iteracion
```

```

std::string unirIngenuo(const std::vector<std::string>& partes) {
    std::string resultado;
    for (const auto& p : partes) {
        resultado += p;
    }
    return resultado;
}

// Version con reserva: una sola asignacion de memoria
std::string unirReservado(const std::vector<std::string>& partes) {
    size_t total = 0;
    for (const auto& p : partes) total += p.size();
    std::string resultado;
    resultado.reserve(total);
    for (const auto& p : partes) {
        resultado += p; // sin realocaciones
    }
    return resultado;
}

template <typename F>
long medir(F funcion, const std::vector<std::string>& partes, int repeticiones) {
    auto inicio = std::chrono::high_resolution_clock::now();
    volatile size_t n = 0; // volatile: evita que el optimizador borre el trabajo
    for (int i = 0; i < repeticiones; i++) {
        n += funcion(partes).size();
    }
    auto fin = std::chrono::high_resolution_clock::now();
    (void)n;
    return std::chrono::duration_cast<std::chrono::milliseconds>(
        fin - inicio).count();
}

int main() {
    std::vector<std::string> partes;
    for (int i = 0; i < 20000; i++) {
        partes.push_back("linea_" + std::to_string(i) + "_de_texto;");
    }

    const int REPS = 20;
    long t1 = medir(unirIngenuo, partes, REPS);
    long t2 = medir(unirReservado, partes, REPS);

    std::cout << "Sin reserve: " << t1 << " ms" << std::endl;
    std::cout << "Con reserve: " << t2 << " ms" << std::endl;
}

```

```

    if (t1 > 0)
        std::cout << "Mejora: ~" << (100 * (t1 - t2) / t1) << "% mas rapido" <<
std::endl;

    std::cout << "\nPrueba compilar con -O0 y luego con -O2:" << std::endl;
    std::cout << "los numeros cambian, pero la version con reserve sigue ganando." <<
std::endl;
    return 0;
}

```

Dos trucos de medicion honesta que el ejemplo ya aplica: repetir muchas veces (una sola medicion la contamina el ruido del sistema) y usar volatile o consumir el resultado (si el compilador detecta que el resultado no se usa, borra todo el calculo y mides cero milisegundos de nada). El benchmark que no se protege del optimizador mide fantasia.

Que cambian -O2 y -O3 (y cuando importan)

-O0 (sin optimizar) compila rapido y es lo mejor para depurar: cada linea de tu codigo corresponde a instrucciones predecibles. -O2 activa optimizaciones serias (funciones en linea, eliminacion de codigo muerto, desenrollado de ciclos): el programa corre mucho mas rapido y compila un poco mas lento. -O3 agrega optimizaciones agresivas (vectorizacion, mas inlining): a veces gana un poco mas, a veces el ejecutable crece sin beneficio.

Consecuencias practicas: mide el rendimiento SIEMPRE con las optimizaciones que usaras en produccion (normalmente -O2). Un benchmark en -O0 te dice como corre tu codigo en depuracion, no como correra en manos del usuario. Y ojo: -O2 no arregla un algoritmo malo; convierte un $O(n^2)$ en un $O(n^2)$ mas rapido. La optimizacion algoritmica (elegir mejor estructura de datos) casi siempre le gana a la optimizacion del compilador. Primero el algoritmo, después el -O2, y solo al final micro-optimizaciones.

Guia de entrevistas de trabajo en C++

Como piensan los entrevistadores

Una entrevista de C++ rara vez pregunta sintaxis memorizada; pregunta criterio. Quieren saber si entiendes RAII, si sabes cuando algo es costoso, si tus decisiones tienen fundamento. Las respuestas modelo de abajo son cortas a proposito: en la entrevista real debes poder explicar cada una con tus palabras y, mejor aun, con un ejemplo de codigo. Si alguna respuesta te cuesta, es senal de que ese tema necesita repaso: la entrevista empieza mucho antes de la entrevista.

Preguntas típicas con respuestas modelo

1. Que es RAII y por que es central en C++?

Respuesta: atar la vida de un recurso a la vida de un objeto: se adquiere en el constructor y se libera en el destructor. Es central porque da seguridad ante excepciones automaticamente (los destructores corren durante el desenrollado) y elimina fugas por caminos olvidados. vector, string, unique_ptr, fstream y lock_guard son todos RAII.

2. unique_ptr vs shared_ptr vs weak_ptr: cuando usas cada uno?

Respuesta: unique_ptr por defecto (propiedad exclusiva, costo cero). shared_ptr solo cuando la propiedad es genuinamente compartida y no se sabe quien muere al ultimo (costo: conteo atómico). weak_ptr para observar sin poseer: caches y referencias hijo -> padre, rompiendo ciclos que shared_ptr solo no puede romper.

3. Por que el destructor de una clase base debe ser virtual?

Respuesta: si destruyes un objeto derivado a través de un puntero a la base sin destructor virtual, solo corre el destructor de la base: los recursos del derivado fugan (comportamiento indefinido en realidad). Con destructor virtual, el desenlace es correcto aunque apuntes con la base.

4. Que es std::move y que hace realmente?

Respuesta: no mueve nada; convierte su argumento en una referencia a rvalue, autorizando que el receptor "canibalice" sus recursos internos en vez de copiarlos. El movimiento real lo hace el constructor o asignacion de movimiento del tipo. Después de mover, el origen queda valido pero indeterminado: no lo uses salvo para reasignar.

5. vector vs list: cuando conviene cada uno?

Respuesta: vector casi siempre: datos contiguos, cache-friendly, acceso $O(1)$. list solo cuando necesitas inserciones y borrados $O(1)$ en medio CON iteradores estables que nunca se invalidan, y aun asi mide primero: en la practica vector le gana a list incluso insertando, por la localidad de cache.

6. Que es comportamiento indefinido? Da un ejemplo.

Respuesta: codigo para el que el estandar no define que pasa: el programa puede hacer cualquier cosa, incluido "funcionar" hoy y explotar mañana. Ejemplos: desreferenciar nullptr, usar variable sin inicializar, mezclar new[] con delete, data race. Por eso -Wall, sanitizers y valgrind son obligatorios.

7. Que significa const correctness y por que importa?

Respuesta: marcar const todo lo que no cambia: parametros de solo lectura (const&), metodos que no mutan, variables fijas. El compilador atrapa mutaciones accidentales, documenta intenciones y permite optimizaciones. Un metodo que no es const no se puede llamar sobre un objeto const: disena la constancia desde el inicio.

8. Que pasa si una excepcion sale de un destructor?

Respuesta: si el destructor corria durante el desenrollado de otra excepcion, hay dos excepciones activas y el programa termina con `std::terminate`. Por eso los destructores son noexcept por defecto y jamas deben lanzar: limpian, no reportan.

9. map vs unordered_map?

Respuesta: map es arbol ordenado $O(\log n)$ con recorrido ordenado y rangos; unordered_map es tabla hash $O(1)$ promedio sin orden. Si necesitas orden o rangos, map; si solo buscas por clave y quieres velocidad, unordered_map. Y mide en tu caso: la teoria orienta, el benchmark decide.

10. Como depurarias un crash en produccion sin poder reproducirlo facil?

Respuesta: compilar con -g y -fsanitize=address en un entorno de pruebas para cazar memoria; correr bajo gdb y usar backtrace en el core dump; agregar bitacora con niveles para reconstruir que hacia el programa antes de morir; reducir a un caso minimo reproducible. Y nunca adivinar: el primer error del compilador o del sanitizer suele ser el real.

11. Que es una data race y como la evitas?

Respuesta: dos hilos acceden al mismo dato, al menos uno escribe, sin sincronizacion: comportamiento indefinido. Se evita con mutex (lock_guard para RAII), o disenando sin estado compartido: datos por hilo, paso de mensajes, o `std::atomic` para contadores simples.

12. Diferencia entre new/delete y malloc/free?

Respuesta: new/delete son operadores de C++: llaman constructores y destructores, son type-safe y lanzan excepcion si falla new. malloc/free son funciones de C: solo reservan bytes crudos, sin construir nada, devuelven nullptr si fallan. En C++ moderno casi nunca usas ninguno directamente: vector, string y punteros inteligentes los envuelven.

Preguntas trampa: las que no son de sintaxis

Además de las tecnicas, hay preguntas disenadas para ver como piensas bajo presion. Tres classicas:

"Cuéntame de un bug difícil que hayas resuelto." No quieren el bug más impresionante, quieren tu método. Estructura tu respuesta: qué síntoma viste, qué hipótesis probaste, qué herramienta te dio la pista (gdb, sanitizer, bitacora), cuál fue la causa raíz y qué cambiaste para que no vuelva. Si puedes contar uno donde un sanitizer o una bitacora te salvo, mejor: demuestra que usas herramientas profesionales.

"¿Qué característica de C++ evitarías y por qué?" Respuesta con criterio: los casts estilo C (prefiere `static_cast` y evita reinterpretar), `new/delete` manuales en código nuevo (`unique_ptr`, `vector` y `string` los reemplazan), o las macros del preprocesador para constantes (`constexpr` es `type-safe`). Lo importante no es la lista, es el "por qué": cada respuesta debe mencionar seguridad, legibilidad o mantenimiento.

"¿Cómo harías X sin la STL?" (por ejemplo, un vector dinámico). No es para que reinventes la STL en producción: quieren ver si entiendes lo que hay debajo. Describe la idea en dos minutos: arreglo dinámico con capacidad, crecimiento por duplicado amortizado $O(1)$, RAII para liberar, regla de tres/cinco si hay copia. Y cierra con: "en código real usaría `std::vector`, esto es solo para demostrar que se que hace por dentro".

La trampa común a las tres: hablar sin estructura. Usa siempre el formato situación -> acción -> resultado. Practica estas respuestas en voz alta antes de la entrevista; la diferencia entre "saberlo" y "explicarlo bien" es exactamente lo que el entrevistador está midiendo.

Consejos para el día de la entrevista

Escribe código en papel o pizarra aunque te den computadora: demuestra que piensas antes de teclear. Habla mientras resuelves: el entrevistador califica tu razonamiento, no solo tu respuesta. Si no sabes algo, dilo y razona en voz alta ("no recuerdo el nombre exacto, pero el problema es X y lo atacaría por Y"): la honestidad razonada vale más que un invento. Y lleva preguntas para ellos: ¿qué versión de C++ usan, cómo hacen code review, qué herramientas de build usan. Una entrevista es en dos direcciones.

Como contribuir a open source (y por que te conviene)

El camino de cero a tu primer PR

Contribuir a open source es la forma más rápida de que tu código lo lean programadores mejores que tú, que es como se aprende de verdad. Y los reclutadores miran GitHub: un historial de contribuciones reales pesa más que un certificado.

1. Elige un proyecto que uses o te guste, escrito en C++ y con actividad reciente. Buenos puntos de entrada: bibliotecas pequenas y populares (fmt, nlohmann/json, Catch2), o proyectos grandes con etiquetas "good first issue" (SFML, Godot tiene areas en C++).
2. Lee el CONTRIBUTING.md y el codigo de conducta antes de tocar nada. Cada proyecto tiene sus reglas de estilo, de commits y de como proponer cambios.
3. Compila el proyecto y corre sus pruebas en tu maquina primero. Si no compila a la primera, ya encontraste tu primera contribucion potencial: mejorar la documentacion de compilacion.
4. Empieza pequeno: un typo en la documentacion, un warning del compilador, una prueba faltante. Los mantenedores confian mas en quien entrega cinco PR pequenos correctos que en quien propone reescribir medio proyecto.
5. Haz fork, crea una rama con nombre descriptivo, un commit por cambio logico, y abre el pull request describiendo QUE cambiaste y POR QUE. Responde a la revision sin tomartelo personal: el review es el regalo, no el obstaculo.

Lo que aprendes en el proceso no esta en ningun libro: leer codigo ajeno grande, seguir convenciones que no inventaste, discutir diseno con desconocidos y aceptar que tu codigo puede mejorar. Esas son exactamente las habilidades de un trabajo real.

Plan de estudios de 90 dias post-libro

Terminaste el libro: tienes el mapa completo del lenguaje. Estos 90 dias convierten ese mapa en territorio recorrido. La regla: cada dia, codigo. Aunque sean 30 minutos.

Dias 1-10: Repaso activo de la STL. Un contenedor o algoritmo por dia, con un programita que lo use de verdad: hoy <algorithm> (sort, find_if, transform), manana <map>, pasado <unordered_map>. Si no lo tecleas, no cuenta.

Dias 11-20: Estructuras de datos classicas implementadas por ti: lista enlazada, pila y cola con tus clases, tabla hash simple, arbol binario de busqueda con recorridos. No para "reinventar la STL", sino para entender que hay debajo.

Dias 21-30: Plantillas en serio. Escribe una funcion y una clase plantilla, luego una con especializacion. Lee como la STL las usa. Las plantillas dejan de dar miedo cuando escribes diez.

Dias 31-45: Proyecto 1 (elige de la lista de la unidad 10, por ejemplo el sistema de inventario con archivos). Aplicalo TODO: clases, excepciones propias, bitacora, persistencia, Makefile, -Wall -Wextra -Werror. Un proyecto terminado vale por veinte a medias.

Dias 46-60: Multihilo y herramientas. El ejemplo de hilos de la unidad 10, luego un programa con mutex de verdad; aprende CMake basico, corre tus programas con `-fsanitize=address` y perfila algo con `<chrono>` como en esta expansion.

Dias 61-75: Open source y portafolio. Sigue la guia de arriba: compila un proyecto, encuentra un "good first issue", abre tu primer PR. En paralelo, sube tus proyectos a GitHub con README decente (que hace, como se compila, ejemplo de uso).

Dias 76-90: Modo entrevista. Responde en voz alta las 12 preguntas de esta expansion, resuelve un problema de logica al dia (HackerRank o LeetCode en C++), y construye un proyecto final que junte todo: por ejemplo, el sistema de biblioteca con hilos, bitacora, persistencia y pruebas. Ese proyecto es tu carta de presentacion.

Al dia 90 no seras senior (nadie lo es en 90 dias), pero seras un junior solido con codigo publico que lo demuestra, que es exactamente lo que contratan.

Sanitizers: que el compilador encuentre tus bugs de memoria

Tu red de seguridad dinamica

En la clase 30 se mencionaron los sanitizers de pasada; aqui los usamos en serio porque son, junto con las pruebas, la herramienta con mejor relacion esfuerzo-beneficio que existe en C++. Un sanitizer es codigo que el compilador inyecta en tu programa para vigilarlo mientras corre: AddressSanitizer detecta lecturas/escrituras fuera de rango, uso después de liberar, dobles liberaciones y fugas; UndefinedBehaviorSanitizer atrapa desbordamientos de enteros con signo, shifts invalidos y demas comportamiento indefinido.

Se activan con una flag, sin cambiar tu codigo:

```
g++ -g -fsanitize=address,undefined -std=c++17 programa.cpp -o programa
./programa
```

Si tu programa tiene un bug de memoria, en vez de "funcionar" misteriosamente o caerse lejos del culpable, obtienes un reporte que dice exactamente que paso, en que linea y con que pila de llamadas. Es como tener un gdb que ya sabe donde mirar.

Ejemplo completo 5: cazando un desbordamiento con AddressSanitizer

Este programa tiene un bug típico y silencioso: escribe un elemento mas alla del vector usando un indice calculado. Sin sanitizer, a veces funciona, a veces corrompe memoria, a veces se cae en otro lado. Con sanitizer, el culpable queda expuesto al instante.

```
#include <iostream>
#include <vector>
#include <string>

int main() {
    std::vector<int> calificaciones = {90, 85, 92, 78, 88};
    int n = calificaciones.size();

    // Bug: el ciclo llega hasta n inclusive (deberia ser < n)
    for (int i = 0; i <= n; i++) {
        // operator[] NO verifica rangos: la ultima iteracion escribe fuera
        calificaciones[i] = calificaciones[i] + 5;
    }

    for (int c : calificaciones) std::cout << c << " ";
    std::cout << std::endl;
    return 0;
}
```

Compilalo normal y ejecútalo: probablemente imprima numeros y termine "bien", aunque corrompio memoria del heap. Ahora compilalo con `-fsanitize=address` y ejecutalo: obtendras un reporte como este (resumido):

```
ERROR: AddressSanitizer: heap-buffer-overflow on address ...
WRITE of size 4 at ...
#0 main programa.cpp:11
```

Línea 11, escritura de 4 bytes fuera del buffer del heap: el diagnostico exacto. El arreglo es cambiar `<=` por `<`, o mejor aun, usar el for de rango o `.at()` cuando el indice viene de logica compleja. La lección profesional: integra los sanitizers a tu rutina. Compila tu proyecto con `-fsanitize=address,undefined` al menos una vez por semana y corre tus pruebas; los bugs de memoria que encuentran son los que en produccion se convierten en vulnerabilidades de seguridad o corrupcion de datos silenciosa.

Tres notas practicas. Una: los sanitizers hacen el programa mas lento y usan mas memoria; son para desarrollo y pruebas, no para el ejecutable final (ahi usas `-O2` sin sanitizers). Dos:

AddressSanitizer y valgrind hacen trabajo parecido; el sanitizer es mas rapido y sus reportes mas precisos, valgrind no necesita recompilar. Tres: si un reporte te confunde, compila con -g para que las lineas del reporte correspondan a tu codigo fuente.

Catalogo de errores comunes de esta expansion

1. Usar `shared_ptr` para todo "por si acaso". Cada `shared_ptr` paga un conteo atomico y oscurece quien es el dueno real. Empieza con `unique_ptr` y sube a `shared_ptr` solo cuando la propiedad compartida sea genuina.
2. Crear ciclos de `shared_ptr` (A posee a B y B posee a A). El conteo nunca llega a cero y la memoria fuga en silencio. Rompe el ciclo con `weak_ptr` en una de las direcciones, tipicamente hijo -> padre.
3. Intentar usar un `weak_ptr` directamente con `*` o `->`. No compila a proposito: primero `lock()` para obtener un `shared_ptr` temporal, y trabaja con ese. El protocolo existe para que no haya carreras entre "verificar" y "usar".
4. Guardar el resultado de `lock()` en un `shared_ptr` miembro "para no estar convirtiendo". Acabas de convertir tu observador en dueno y reintrodujiste el ciclo o el secuestro de memoria. El `shared_ptr` de `lock()` debe ser temporal y local.
5. Poner `std::move` en el return de una variable local: `return std::move(datos);` puede impedir la elision de copia (NRVO) y salir peor que `return datos;` a secas. Deja que el compilador haga su trabajo.
6. Usar un objeto después de `std::move(x)` como si nada hubiera pasado. Queda valido pero con contenido indeterminado (normalmente vacio). Solo reasignalo o destruyelo.
7. Escribir un deleter que no libera (olvida el `delete/fclose` dentro de la lambda). El `unique_ptr` confia ciegamente en tu deleter: si no libera, fugaste con estilo moderno.
8. Medir rendimiento en -O0 y tomar decisiones con esos numeros. El codigo de produccion corre con -O2; un benchmark en -O0 mide otro programa. Mide siempre con las flags reales.
9. Benchmarks que el optimizador borra: calculas algo y no usas el resultado, el compilador elimina el calculo y mides cero. Consume el resultado (`volatile`, suma acumulada, imprimirlo).
10. Optimizar sin medir primero. El 97% del tiempo se va en el 3% del codigo (o algo asi): si optimizas a ciegas, casi seguro optimizas lo que no importa. Mide, ataca el cuello de botella, vuelve a medir.

11. -O3 como varita magica sobre codigo con comportamiento indefinido. El optimizador agresivo y el undefined behavior son enemigos mortales: lo que "funcionaba" en -O0 puede explotar en -O3. Primero sanitizers, después optimizacion.
12. En entrevistas: decir "no se" y quedarse callado. Lo que se califica es el razonamiento: "no recuerdo el detalle, pero el problema es X y lo atacaria asi" es una respuesta valida y valiente.
13. En open source: abrir un PR gigante que cambia diez cosas a la vez. Los mantenedores no pueden revisarlo y lo ignoran. Un cambio logico por PR, pequeno y bien descrito.
14. Estudiar 90 dias viendo videos sin teclear. El plan de arriba solo funciona si cada dia hay codigo compilado. La comprension vive en los dedos.

Practica

Ejercicio 1

Crea dos clases, Empleado y Departamento, donde el departamento posee a sus empleados con shared_ptr y cada empleado referencia a su departamento. Implementa primero la version con el ciclo (shared_ptr en ambas direcciones) y demuestra la fuga con mensajes en destructores. Luego arreglala con weak_ptr y demuestra que todo se destruye.

Solucion:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>

class Departamento; // declaracion anticipada

class Empleado {
private:
    std::string nombre;
    std::weak_ptr<Departamento> depto; // observa, no posee: rompe el ciclo
public:
    explicit Empleado(const std::string& n) : nombre(n) {
        std::cout << " Empleado '" << nombre << "' creado" << std::endl;
    }
    ~Empleado() { std::cout << " Empleado '" << nombre << "' destruido" << std::endl;
    }

    void asignar(std::shared_ptr<Departamento> d) { depto = d; }
    bool tieneDepto() const { return !depto.expired(); }
```



```

};

class Departamento : public std::enable_shared_from_this<Departamento> {
private:
    std::string nombre;
    std::vector<std::shared_ptr<Empleado>> personal; // el depto SI posee
public:
    explicit Departamento(const std::string& n) : nombre(n) {
        std::cout << "Departamento '" << nombre << "' creado" << std::endl;
    }
    ~Departamento() {
        std::cout << "Departamento '" << nombre << "' destruido" << std::endl;
    }
    void contratar(const std::string& nombreEmpleado) {
        auto e = std::make_shared<Empleado>(nombreEmpleado);
        e->asignar(shared_from_this());
        personal.push_back(e);
    }
};

int main() {
    {
        auto ventas = std::make_shared<Departamento>("Ventas");
        ventas->contratar("Ana");
        ventas->contratar("Luis");
        std::cout << "Fin del bloque: todo debe destruirse..." << std::endl;
    }
    std::cout << "Si ves los 3 'destruido', no hay ciclo." << std::endl;

    // Demostracion del bug original (comentada para no fugar de verdad):
    // si Empleado guardara shared_ptr<Departamento> en vez de weak_ptr,
    // al cerrar el bloque el conteo del departamento nunca llegaria a cero
    // porque cada empleado lo mantendria vivo, y verias cero mensajes
    // de destruccion. Pruebalo cambiando weak_ptr por shared_ptr.
    return 0;
}

```

Explicacion: la direccion de propiedad es departamento -> empleados (shared_ptr, posee) y empleado -> departamento (weak_ptr, observa). Al morir el ultimo shared_ptr del departamento, todo se destruye en cascada: primero el departamento, luego sus empleados, aunque cada empleado "conoce" a su departamento. enable_shared_from_this permite que el departamento se pase a si mismo como shared_ptr de forma segura dentro de contratar(). Si inviertes la direccion y pones shared_ptr en ambos lados, obtienes la fuga silenciosa: ningun destructor se ejecuta.

Ejercicio 2

Envuelve un buffer reservado con malloc en un unique_ptr con deleter personalizado que llame a free y registre la liberacion. Escribe y lee 5 enteros en el buffer. Luego explica en un comentario por que esto es mejor que manejar malloc/free a mano.

Solucion:

```
#include <iostream>
#include <memory>
#include <cstdlib>
#include <stdexcept>

int main() {
    // unique_ptr con deleter que libera con free (para memoria de malloc)
    auto buffer = std::unique_ptr<int, void(*)(void*)>(
        static_cast<int*>(std::malloc(5 * sizeof(int))),
        [](void* p) {
            std::cout << "Liberando buffer con free()" << std::endl;
            std::free(p);
        });

    if (!buffer) throw std::runtime_error("malloc fallo");

    for (int i = 0; i < 5; i++) buffer.get()[i] = (i + 1) * 100;
    for (int i = 0; i < 5; i++) std::cout << buffer.get()[i] << " ";
    std::cout << std::endl;

    // Comentario pedido:
    // Mejor que malloc/free a mano porque:
    // 1. Si una excepcion atraviesa este codigo, el deleter corre igual (free
    // garantizado).
    // 2. Es imposible olvidar el free o ponerlo en un solo camino de salida.
    // 3. El tipo del unique_ptr documenta como se libera (free, no delete).
    // 4. No se puede copiar accidentalmente el puntero crudo y hacer doble free.
    return 0;
}
```

Explicacion: este es el patron para interoperar con bibliotecas C que devuelven memoria con malloc: la envuelves una vez en un unique_ptr con el deleter correcto y a partir de ahi vive con reglas de C++. El static_cast es necesario porque malloc devuelve void*. Observa que el deleter se declara como void(*)(void*): recibe el puntero almacenado (int*) convertido a void*, y free lo acepta.

Ejercicio 3

Escribe una función `procesarRegistros` que reciba un `vector<string>` por valor, lo ordene y devuelva cuantos registros contienen la palabra "ERROR". Luego escribe un main que la llame dos veces: una pasando una copia (cuando necesitas conservar el original) y otra con `std::move` (cuando ya no lo necesitas). Mide e imprime el tiempo de cada llamada con `<chrono>` para ver la diferencia.

Solucion:

```

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <chrono>

int procesarRegistros(std::vector<std::string> datos) { // por valor: hunde
    std::sort(datos.begin(), datos.end());
    int errores = 0;
    for (const auto& r : datos)
        if (r.find("ERROR") != std::string::npos) errores++;
    return errores;
}

long ahoraMs() {
    return std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now().time_since_epoch()).count();
}

int main() {
    std::vector<std::string> base;
    for (int i = 0; i < 150000; i++)
        base.push_back("registro_" + std::to_string(i) +
            (i % 7 == 0 ? "_ERROR_critico" : "_ok"));

    // Caso 1: conservo el original -> paso copia explicita
    long t1 = ahoraMs();
    int e1 = procesarRegistros(base); // copia dentro del parametro
    long dt1 = ahoraMs() - t1;
    std::cout << "Con copia: " << dt1 << " ms, errores=" << e1
        << ", base intacta: " << base.size() << std::endl;

    // Caso 2: ya no necesito el original -> nuevo
    std::vector<std::string> paraTirar = base; // copia de trabajo
    long t2 = ahoraMs();
    int e2 = procesarRegistros(std::move(paraTirar)); // mueve: sin duplicar
    long dt2 = ahoraMs() - t2;
    std::cout << "Con move: " << dt2 << " ms, errores=" << e2
        << ", paraTirar quedo con " << paraTirar.size() << std::endl;
    return 0;
}

```

Explicacion: recibir por valor es el idioma moderno para "la funcion se queda con los datos": el que llama decide si copiar (pasa el vector tal cual) o mover (std::move). La medicion muestra

que el caso con move evita una duplicacion completa de 150,000 strings. Fijate que no hay ninguna copia visible dentro de procesarRegistros: la unica copia ocurre en la llamada del caso 1, exactamente donde el programador la autorizo al no mover.

Ejercicio 4

Demuestra con codigo y medicion por que "el destructor virtual" importa: crea una clase Base con un recurso (un vector grande como atributo) y destructor NO virtual, y una Derivada que agrega otro recurso. Destruye un objeto Derivada a través de un puntero Base* (con new/delete a proposito para el experimento) y observa que destructor corre. Luego repite con destructor virtual y compara. Explica que paso en cada caso.

Solucion:

```
#include <iostream>
#include <vector>
#include <string>

class BaseSinVirtual {
public:
    std::vector<int> datos{std::vector<int>(1000000, 1)};
    ~BaseSinVirtual() { std::cout << " ~BaseSinVirtual" << std::endl; }
};

class DerivadaSinVirtual : public BaseSinVirtual {
public:
    std::string etiqueta = std::string(1000000, 'x');
    ~DerivadaSinVirtual() { std::cout << " ~DerivadaSinVirtual" << std::endl; }
};

class BaseConVirtual {
public:
    std::vector<int> datos{std::vector<int>(1000000, 1)};
    virtual ~BaseConVirtual() { std::cout << " ~BaseConVirtual" << std::endl; }
};

class DerivadaConVirtual : public BaseConVirtual {
public:
    std::string etiqueta = std::string(1000000, 'x');
    virtual ~DerivadaConVirtual() { std::cout << " ~DerivadaConVirtual" << std::endl; }
};

int main() {
```

```

std::cout << "Sin virtual:" << std::endl;
BaseSinVirtual* p1 = new DerivadaSinVirtual();
delete p1; // comportamiento indefinido: solo corre ~BaseSinVirtual

std::cout << "Con virtual:" << std::endl;
BaseConVirtual* p2 = new DerivadaConVirtual();
delete p2; // correcto: derivada primero, luego base

    std::cout << "\nConclusion: sin virtual, el string de 1MB de la derivada" <<
std::endl;
    std::cout << "jamás se destruye (fuga + indefinido). Con virtual, todo" <<
std::endl;
    std::cout << "se libera en orden. Por eso: clase base con funciones" << std::endl;
    std::cout << "virtuales => destructor virtual. Siempre." << std::endl;
    return 0;
}

```

Explicación: en el caso sin virtual veras solo "~BaseSinVirtual": el destructor de la derivada nunca corre, su string de un millón de caracteres fuga y el programa incurre en comportamiento indefinido (hoy "solo" fuga; mañana puede corromper memoria). Con virtual ves ambos destructores en orden (derivada, luego base) y todo se libera. Esta es la pregunta 3 de la guía de entrevistas demostrada con código: si la puedes explicar con este ejemplo, la tienes dominada.

Ejercicio 5

Escribe un micro-benchmark que compare insertar 500,000 enteros en un vector SIN reserve vs CON reserve, midiendo con <chrono> y repitiendo 5 veces cada caso (reporta el promedio). Luego responde en un comentario: que flag de optimización usaste y por qué el benchmark protege el trabajo con volatile o consumiendo el resultado.

Solución:

```

#include <iostream>
#include <vector>
#include <chrono>

template <typename F>
long medir(F f, int repeticiones) {
    auto inicio = std::chrono::high_resolution_clock::now();
    volatile long long acumulador = 0;
    for (int i = 0; i < repeticiones; i++) acumulador += f();
    auto fin = std::chrono::high_resolution_clock::now();
    (void)acumulador;
    return std::chrono::duration_cast<std::chrono::milliseconds>(
        fin - inicio).count();
}

long sinReserve() {
    std::vector<int> v;
    for (int i = 0; i < 500000; i++) v.push_back(i);
    return v.size();
}

long conReserve() {
    std::vector<int> v;
    v.reserve(500000);
    for (int i = 0; i < 500000; i++) v.push_back(i);
    return v.size();
}

int main() {
    const int REPS = 5;
    long t1 = medir(sinReserve, REPS);
    long t2 = medir(conReserve, REPS);
    std::cout << "Sin reserve: " << t1 / REPS << " ms promedio" << std::endl;
    std::cout << "Con reserve: " << t2 / REPS << " ms promedio" << std::endl;

    // Comentario pedido:
    // Flag: -O2, porque es la optimizacion tipica de produccion; medir en -O0
    // contaria una historia distinta a la que vivira el usuario.
    // El acumulador volatile (y devolver v.size()) impide que el compilador
    // elimine los ciclos por considerarlos trabajo inutil: sin esa proteccion,
    // el benchmark mediria cero milisegundos de nada.
    return 0;
}

```

Explicacion: sin reserve el vector realoca logaritmicamente (cada realocacion copia todo lo acumulado); con reserve hay una sola asignacion. La diferencia se nota en el promedio de 5 repeticiones, que filtra el ruido del sistema. Este ejercicio reúne las tres lecciones de la seccion de perfilado: medir con repeticiones, proteger el benchmark del optimizador y usar las flags de produccion.

Tu bitacora de aprendizaje: el habito que multiplica todo

Un ultimo consejo que no es tecnico pero vale mas que una característica del lenguaje: lleva una bitacora de lo que aprendes. Cada vez que un bug te quite mas de una hora, escribe que sintoma viste, cual fue la causa raiz y que herramienta te dio la pista. Cada vez que un concepto por fin "haga clic" (el primer `weak_ptr` que entiendas de verdad, el primer benchmark que te sorprenda), anotalo con tus palabras.

Esta bitacora hace tres cosas por ti. Una: convierte errores en activos, porque el bug documentado no se repite. Dos: te da material honesto para entrevistas, donde "cuentame de un bug dificil" se responde con historias reales en vez de improvisacion. Tres: dentro de seis meses, cuando releas tus primeras entradas, veras con claridad cuanto avanzaste, y esa evidencia es el mejor antidoto contra el sindrome del impostor. Los programadores que documentan lo que aprenden crecen mas rapido que los que solo acumulan horas frente al editor.

Cierre: lo que sigue es tuyo

Terminaste la expansion y con ella el curso completo. Repasa lo que ahora tienes en las manos: jerarquias de excepciones que comunican, recursos que se limpian solos aunque todo falle, bitacoras que cuentan la historia, formatos de datos que aguantan el mundo real, punteros que observan sin poseer, movimiento en vez de copia, mediciones honestas antes de optimizar, respuestas para entrevistas, un mapa para open source y un plan de 90 dias.

Ningun libro te hace programador: los programas que escribas a partir de hoy si. Elige el primer proyecto de la lista, abre tu editor y empieza. Cuando te atores (porque te vas a atorar, todos nos atoramos), vuelve a estas paginas: no para memorizarlas, sino para recordar que cada problema que encuentres ya lo resolvió alguien con las herramientas que ahora conoces.

Nos vemos en el codigo. Lucas EGA Academy.

EXAMEN FINAL

Llegó la hora de la verdad. Este examen tiene 50 preguntas que cubren todo el libro: opción múltiple, verdadero o falso y preguntas abiertas donde escribirás código. Resuélvelo sin mirar las soluciones; al final encontrarás "Respuestas y soluciones" con cada respuesta explicada.

Reglas: para las preguntas de código, escríbelo completo y compilable mentalmente. No vale "más o menos": en C++ los detalles importan. Si sacas 40 o más, estás listo para programar C++ en serio. ¡Suerte!

Preguntas de opción múltiple

PREGUNTA 1. ¿Quién creó C++ y en qué década nació?

A) Bjarne Stroustrup, años 70 B) Bjarne Stroustrup, años 80 C) Dennis Ritchie, años 80 D) Guido van Rossum, años 90

PREGUNTA 2. ¿Qué significa el nombre "C++"?

A) C elevado al cuadrado B) Es la segunda versión de C C) "C incrementado", por el operador ++ D) Son las iniciales de sus creadores

PREGUNTA 3. ¿Cuál es la principal ventaja de que C++ sea un lenguaje compilado?

A) Los programas se escriben más rápido B) El programa final es código máquina rápido y el compilador detecta errores antes de ejecutar C) No necesita sistema operativo D) Funciona sin compilar en cualquier máquina

PREGUNTA 4. ¿Qué comando compila programa.cpp generando el ejecutable "programa" con advertencias y estándar moderno?

A) g++ programa.cpp B) gcc -o programa programa.cpp C) g++ -Wall -Wextra -std=c++17 programa.cpp -o programa D) compilar programa.cpp -o programa

PREGUNTA 5. ¿Qué hace la línea `std::cout << "Hola" << std::endl;`?

A) Lee texto del teclado B) Muestra "Hola" en pantalla y baja una línea C) Guarda "Hola" en una variable D) Compila el programa

PREGUNTA 6. ¿Cuál es el problema de declarar `int x;` sin inicializar y luego usar `x`?

A) Ninguno, siempre vale 0 B) El programa no compila C) Contiene basura (valor indeterminado) y usarla es un error grave D) Siempre vale -1

PREGUNTA 7. ¿Cuánto vale 7 / 2 en C++ si ambos son int?

A) 3.5 B) 3 C) 4 D) Error de compilación

PREGUNTA 8. ¿Qué imprime este código? `int a = 5; std::cout << (a % 2 == 0);`

A) 1 B) 0 C) true D) Error

PREGUNTA 9. ¿Cuál es la diferencia entre ++x y x++?

A) No hay diferencia nunca B) ++x incrementa después de usar el valor; x++ antes C) ++x incrementa primero y luego usa el valor; x++ usa el valor primero y luego incrementa D) Solo ++x existe en C++

PREGUNTA 10. ¿Qué error hay en `if (x = 5) { ... }`?

A) Ninguno, compara correctamente B) Usa asignación en vez de comparación: asigna 5 a x y siempre es verdadero C) Falta el punto y coma D) x debe ser double

PREGUNTA 11. En un switch, ¿qué pasa si olvidas el break al final de un case?

A) Error de compilación B) La ejecución continúa con el siguiente case (fallthrough) C) El programa se cierra D) Se ejecuta el default automáticamente

PREGUNTA 12. ¿Cuál es la diferencia esencial entre while y do-while?

A) while es más rápido B) do-while ejecuta el bloque al menos una vez; while puede ejecutarse cero veces C) do-while no usa condición D) No hay diferencia

PREGUNTA 13. ¿Para qué sirve la instrucción continue dentro de un ciclo?

A) Termina el programa B) Sale del ciclo inmediatamente C) Salta a la siguiente repetición del ciclo D) Pausa el ciclo 1 segundo

PREGUNTA 14. ¿Qué ventaja tiene el for de rango `for (int n : v)`?

A) Es más rápido que el for normal siempre B) Recorre la colección sin índices, eliminando errores de límites C) Funciona solo con arreglos de enteros D) Permite modificar el tamaño del vector durante el recorrido

PREGUNTA 15. En C++, los parámetros de función se pasan:

A) Siempre por referencia B) Por valor (copias) por defecto C) Siempre por puntero D) No se pueden pasar parámetros

PREGUNTA 16. ¿Qué es la sobrecarga de funciones?

A) Llamar a una función muchas veces B) Definir varias funciones con el mismo nombre pero distintos parámetros C) Una función que se llama a sí misma D) Pasar demasiados argumentos a una función

PREGUNTA 17. ¿Por qué NO se puede sobrecargar solo cambiando el tipo de retorno?

A) Porque el compilador elige la función por los argumentos, no por lo que esperas recibir B) Porque es ilegal en todos los lenguajes C) Porque los tipos de retorno no existen en C++ D) Sí se puede, sin problema

PREGUNTA 18. En un arreglo `int a[5]`, ¿cuáles son los índices válidos?

A) 1 a 5 B) 0 a 5 C) 0 a 4 D) 1 a 4

PREGUNTA 19. ¿Qué hace `std::getline(std::cin, texto)`?

A) Lee una palabra sin espacios B) Lee una línea completa incluyendo espacios C) Lee un número entero D) Limpia la pantalla

PREGUNTA 20. Dado `int x = 10; int* p = &x;`, ¿qué vale `*p`?

A) La dirección de x B) 10 C) La dirección de p D) Error de compilación

PREGUNTA 21. ¿Qué es una fuga de memoria?

A) Cuando el programa usa demasiada CPU B) Memoria reservada con `new` que nunca se libera con `delete` C) Cuando un vector crece demasiado D) Un error de sintaxis

PREGUNTA 22. ¿Qué es una referencia en C++?

A) Un puntero a una función B) Un alias: otro nombre para una variable existente C) Una copia de una variable D) Un comentario en el código

PREGUNTA 23. ¿Cuál es la forma profesional de pasar un string grande a una función que solo lo lee?

A) `void f(std::string s)` B) `void f(std::string* s)` C) `void f(const std::string& s)` D) `void f(std::string& s)`

PREGUNTA 24. ¿Qué es un constructor?

A) Un método que se ejecuta al destruir el objeto B) Un método especial con el mismo nombre de la clase que inicializa el objeto al crearse C) Una función global D) Un tipo de variable

PREGUNTA 25. ¿Qué significa RAI?

A) Un tipo de herencia múltiple B) Los recursos se adquieren al inicializar el objeto y se liberan en su destructor, automáticamente C) Una biblioteca de la STL D) Un modo de compilación rápida

Preguntas de verdadero o falso

PREGUNTA 26. (V/F) En C++ moderno se recomienda usar `std::vector` en vez de arreglos crudos con `new[]` en la mayoría de los casos.

PREGUNTA 27. (V/F) Los atributos `private` pueden ser modificados directamente desde el `main` si el objeto es local.

PREGUNTA 28. (V/F) Una clase derivada puede acceder a los miembros `protected` de su clase base.

PREGUNTA 29. (V/F) Para que haya polimorfismo en tiempo de ejecución, el método de la clase base debe ser virtual.

PREGUNTA 30. (V/F) Si una clase tiene funciones virtuales, su destructor debería ser virtual.

PREGUNTA 31. (V/F) Se puede instanciar directamente una clase abstracta (con funciones virtuales puras).

PREGUNTA 32. (V/F) La sobrecarga de operadores permite crear operadores completamente nuevos como `o //`.

PREGUNTA 33. (V/F) Las plantillas (templates) generan código en tiempo de compilación, sin costo en ejecución.

PREGUNTA 34. (V/F) En un `std::map`, el operador `[]` crea la clave si no existe.

PREGUNTA 35. (V/F) Un `std::set` permite elementos duplicados.

PREGUNTA 36. (V/F) Cuando se lanza una excepción dentro de un `try`, el código que sigue al `throw` dentro del mismo `try` sí se ejecuta.

PREGUNTA 37. (V/F) `std::unique_ptr` permite que varios punteros compartan la propiedad de la misma memoria.

PREGUNTA 38. (V/F) `std::make_unique` es preferible a usar `new` directamente con `unique_ptr`.

PREGUNTA 39. (V/F) Al pasar un arreglo a una función, la función conoce automáticamente su tamaño.

PREGUNTA 40. (V/F) El operador `==` en `std::string` compara el contenido del texto, no las direcciones de memoria.

Preguntas abiertas (escribe el código)

PREGUNTA 41. Escribe un programa completo que pida al usuario un número entero `N` y muestre la suma de todos los números pares desde 2 hasta `N` (inclusive). Usa un ciclo `for`.

PREGUNTA 42. Escribe una función plantilla llamada `encontrarMaximo` que reciba un `std::vector<T>` y devuelva el elemento mayor. Si el vector está vacío, lanza una excepción `std::runtime_error`. Incluye un `main` que la pruebe con enteros.

PREGUNTA 43. Diseña una clase `RectanguloPro` con atributos privados `base` y `altura`; constructor con validación (si un valor es `<= 0`, lanza `std::invalid_argument`); métodos `area()`, `perimetro()` y operador `<<` para imprimirlo como `"Rectangulo(base*altura)"`. Escribe la clase completa y un `main` de prueba.

PREGUNTA 44. Crea una jerarquía polimórfica: clase abstracta `Empleado2` con nombre (`string`), método virtual puro `calcularSalario()` `const` y destructor virtual. Deriva `EmpleadoFijo` (salario mensual) y `EmpleadoPorHoras` (tarifa por hora y horas trabajadas). Escribe las tres clases y un `main` que guarde ambos en un `vector<unique_ptr<Empleado2>>`, muestre cada salario y el total de la nómina.

PREGUNTA 45. Escribe un programa que lea el archivo `"ventas.txt"` (una cantidad por línea), calcule el total y el promedio usando `vector`, y guarde el resultado en `"reporte.txt"` con el formato `"Total: X"` y `"Promedio: Y"`. Maneja con excepciones el caso de que el archivo no exista.

PREGUNTA 46. Explica con tus palabras la diferencia entre puntero y referencia, y da un ejemplo de código donde una referencia sea la mejor opción y otro donde un puntero (o puntero inteligente) sea necesario.

PREGUNTA 47. Escribe una función `void ordenarBurbuja(std::vector<int>& v)` que ordene el vector con el método de burbuja, y un `main` que la pruebe mostrando el vector antes y después. (Pista: dos `for` anidados comparando vecinos.)

PREGUNTA 48. Diseña una clase plantilla `PilaSegura<T>` con capacidad máxima fija: constructor que recibe la capacidad; métodos `push` (lanza excepción si está llena), `pop` (lanza excepción si está vacía), `cima()` `const` y `vacia()` `const`. Usa un vector interno. Escribe la clase y un `main` de prueba con `PilaSegura<std::string>`.

PREGUNTA 49. Explica qué es el encapsulamiento y por qué es mejor validar en los `setters` que dejar los atributos públicos. Ilústralo con una clase `Edad` con atributo privado `años` y `setAnios` que solo acepte valores entre 0 y 150.

PREGUNTA 50. PROYECTO MINI: escribe un sistema de contactos completo en un solo programa: struct o clase `Contacto` (`nombre`, `telefono`, `email`); clase `Agenda` con vector de contactos y métodos `agregar`, `buscarPorNombre` (devuelve índice o -1), `eliminarPorNombre`, `listar` y `guardar/cargar` en `"agenda.txt"` (formato `nombre|telefono|email`). Incluye un menú en `main` con `do-while` y `switch`. Usa lo aprendido: strings, vector, archivos y validación básica.

Respuestas y soluciones

RESPUESTA 1. B. Bjarne Stroustrup creó C++ en los laboratorios Bell a inicios de los años 80 (el proyecto empezó en 1979 como "C con clases" y el nombre C++ llegó en 1983). Dennis Ritchie creó C; Guido van Rossum creó Python.

RESPUESTA 2. C. En C, `++` es el operador de incremento, así que C++ significa "C incrementado": un chiste de programador que se volvió nombre oficial.

RESPUESTA 3. B. El compilador traduce a código máquina (rápido en ejecución) y revisa tipos y errores antes de que el programa corra. La desventaja es el paso extra de compilación.

RESPUESTA 4. C. `-Wall` `-Wextra` activan advertencias, `-std=c++17` fija el estándar moderno, `-o` nombra el ejecutable. Compilar sin banderas funciona, pero te pierdes la ayuda del compilador.

RESPUESTA 5. B. `cout` envía a pantalla con `<<`, y `endl` inserta salto de línea. `cin` (no `cout`) lee del teclado.

RESPUESTA 6. C. Las variables no inicializadas contienen basura. A veces "funciona" por casualidad, lo que hace el error más traicionero. Inicializa siempre.

RESPUESTA 7. B. División entre enteros = resultado entero truncado: $7/2 = 3$. Para 3.5 usa $7.0/2$.

RESPUESTA 8. B. $a \% 2 = 1$, y $1 == 0$ es falso, que se imprime como 0. Los bool se muestran como 0/1 por defecto.

RESPUESTA 9. C. En expresiones: $++x$ incrementa y luego entrega el valor nuevo; $x++$ entrega el valor viejo y luego incrementa. Como instrucción sola no hay diferencia.

RESPUESTA 10. B. $=$ asigna. La condición queda con valor 5 (verdadero) y además modifica x. El clásico error: usa $==$ para comparar.

RESPUESTA 11. B. Sin break hay fallthrough: se ejecuta el siguiente case. A veces es intencional (casos agrupados), casi siempre es un descuido.

RESPUESTA 12. B. while revisa antes (cero o más vueltas); do-while ejecuta primero y revisa después (una o más vueltas).

RESPUESTA 13. C. continue salta a la siguiente iteración; break sale del ciclo por completo.

RESPUESTA 14. B. Elimina los índices y con ellos los errores de límites. No siempre es más rápido; es más seguro y legible.

RESPUESTA 15. B. Por valor: la función recibe una copia. Para modificar el original usa referencias o punteros.

RESPUESTA 16. B. Mismo nombre, distintos parámetros; el compilador elige según los argumentos. No se distingue por tipo de retorno.

RESPUESTA 17. A. La resolución de sobrecarga usa los argumentos. Con el mismo parámetro y distinto retorno, el compilador no sabría cuál elegir.

RESPUESTA 18. C. Los índices van de 0 a tamaño-1: en $a[5]$, de 0 a 4. $a[5]$ no existe y accederlo es un error grave.

RESPUESTA 19. B. getline lee la línea completa con espacios. $\text{cin} \gg$ se detiene en el primer espacio.

RESPUESTA 20. B. p guarda la dirección de x; *p "sigue" esa dirección y obtiene el valor: 10.

RESPUESTA 21. B. Memoria pedida con new y jamás liberada con delete. En programas largos agota la memoria. Los punteros inteligentes la evitan.

RESPUESTA 22. B. Un alias: $\text{int\& ref} = x$; hace que ref sea x con otro nombre. Debe inicializarse al declararse y no puede reasignarse a otra variable.

RESPUESTA 23. C. `const std::string&` evita copiar el string (eficiente) e impide modificarlo (seguro). Es el estándar profesional.

RESPUESTA 24. B. Se llama igual que la clase, no tiene retorno y se ejecuta solo al crear el objeto. Ideal con lista de inicialización.

RESPUESTA 25. B. Resource Acquisition Is Initialization: el constructor adquiere el recurso y el destructor lo libera. Es la base de la memoria segura en C++ moderno.

RESPUESTA 26. Verdadero. vector crece solo, sabe su tamaño y se libera solo. Los arreglos crudos con `new[]` quedan para casos especiales.

RESPUESTA 27. Falso. `private` es `private`: solo los métodos de la clase pueden tocar esos atributos, sin importar desde dónde se llame.

RESPUESTA 28. Verdadero. `protected` es visible para la clase y sus derivadas, pero no para el código externo. Por eso existe.

RESPUESTA 29. Verdadero. Sin `virtual` hay ocultamiento estático: se llama según el tipo de la variable, no del objeto real. `virtual + override` = polimorfismo real.

RESPUESTA 30. Verdadero. Sin destructor `virtual`, al destruir un derivado vía puntero a la base solo se ejecuta el destructor de la base: fugas y errores. Regla de oro.

RESPUESTA 31. Falso. Las clases abstractas no se instancian; sirven como interfaz/base. Solo sus clases derivadas concretas se pueden crear.

RESPUESTA 32. Falso. Solo puedes redefinir operadores existentes; no inventar nuevos ni cambiar su precedencia o aridad.

RESPUESTA 33. Verdadero. El compilador genera una versión por cada tipo usado. En ejecución es tan rápido como código escrito a mano.

RESPUESTA 34. Verdadero. `mapa["nueva"]` crea la entrada con valor por defecto. Para buscar sin crear usa `find`.

RESPUESTA 35. Falso. El set ignora duplicados: insertar dos veces el mismo valor deja un solo elemento.

RESPUESTA 36. Falso. Al lanzar, se abandona el `try` inmediatamente y se salta al `catch`. Lo que sigue al `throw` no se ejecuta.

RESPUESTA 37. Falso. Eso es `shared_ptr` (propiedad compartida con conteo). `unique_ptr` es propiedad exclusiva y no se puede copiar.

RESPUESTA 38. Verdadero. `make_unique` evita fugas en expresiones complejas y es más legible. Es la forma recomendada desde C++14.

RESPUESTA 39. Falso. El arreglo decae a puntero al primer elemento; la función no sabe el tamaño. Siempre pasa el tamaño como parámetro aparte (o usa vector).

RESPUESTA 40. Verdadero. `std::string` sobrecarga `==` para comparar contenido. (Con `char*` de C compararía direcciones: otra razón para usar `string`.)

RESPUESTA 41.

```
#include <iostream>

int main() {
    int n;
    std::cout << "N: ";
    std::cin >> n;
    int suma = 0;
    for (int i = 2; i <= n; i += 2) {
        suma += i;
    }
    std::cout << "Suma de pares hasta " << n << ": " << suma << std::endl;
    return 0;
}
```

Explicación: el `for` va de 2 en 2 (`i += 2`), así solo visita pares. Si `N` es impar, el último par menor que `N` se incluye por la condición `i <= n`.

RESPUESTA 42.

```

#include <iostream>
#include <vector>
#include <stdexcept>

template <typename T>
T encontrarMaximo(const std::vector<T>& v) {
    if (v.empty()) {
        throw std::runtime_error("Vector vacio: no hay maximo");
    }
    T maximo = v[0];
    for (const T& e : v) {
        if (e > maximo) maximo = e;
    }
    return maximo;
}

int main() {
    try {
        std::vector<int> v = {3, 9, 1, 7};
        std::cout << "Maximo: " << encontrarMaximo(v) << std::endl;
        std::vector<int> vacio;
        std::cout << encontrarMaximo(vacio) << std::endl;
    } catch (const std::runtime_error& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
    return 0;
}

```

Explicación: plantilla para cualquier T comparable con >. El vector vacío es un caso de error real: lanzamos excepción en vez de devolver basura. El parámetro es const& por eficiencia.

RESPUESTA 43.

```

#include <iostream>
#include <stdexcept>

class RectanguloPro {
private:
    double base, altura;

public:
    RectanguloPro(double b, double h) {
        if (b <= 0 || h <= 0) {
            throw std::invalid_argument("Base y altura deben ser positivas");
        }
        base = b;
        altura = h;
    }

    double area() const { return base * altura; }
    double perimetro() const { return 2 * (base + altura); }

    friend std::ostream& operator<<(std::ostream& os, const RectanguloPro& r) {
        return os << "Rectangulo(" << r.base << "x" << r.altura << ")";
    }
};

int main() {
    try {
        RectanguloPro r(5, 3);
        std::cout << r << " area=" << r.area()
                  << " perimetro=" << r.perimetro() << std::endl;
        RectanguloPro malo(-2, 3);
    } catch (const std::invalid_argument& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
    return 0;
}

```

Explicación: el constructor valida e impide objetos imposibles (encapsulamiento). operator<< como friend permite imprimir con cout. Métodos const porque no modifican.

RESPUESTA 44.

```

#include <iostream>
#include <string>
#include <vector>
#include <memory>

```

```

class Empleado2 {
protected:
    std::string nombre;
public:
    Empleado2(const std::string& n) : nombre(n) {}
    virtual double calcularSalario() const = 0;
    virtual ~Empleado2() {}
    std::string getNombre() const { return nombre; }
};

class EmpleadoFijo : public Empleado2 {
private:
    double salarioMensual;
public:
    EmpleadoFijo(const std::string& n, double s)
        : Empleado2(n), salarioMensual(s) {}
    double calcularSalario() const override { return salarioMensual; }
};

class EmpleadoPorHoras : public Empleado2 {
private:
    double tarifa, horas;
public:
    EmpleadoPorHoras(const std::string& n, double t, double h)
        : Empleado2(n), tarifa(t), horas(h) {}
    double calcularSalario() const override { return tarifa * horas; }
};

int main() {
    std::vector<std::unique_ptr<Empleado2>> nomina;
    nomina.push_back(std::make_unique<EmpleadoFijo>("Ana", 15000));
    nomina.push_back(std::make_unique<EmpleadoPorHoras>("Luis", 200, 120));

    double total = 0;
    for (const auto& e : nomina) {
        double s = e->calcularSalario();
        std::cout << e->getNombre() << ": $" << s << std::endl;
        total += s;
    }
    std::cout << "Total nomina: $" << total << std::endl;
    return 0;
}

```

Explicación: clase abstracta con virtual puro, destructor virtual, dos derivadas con su cálculo. El vector de unique_ptr demuestra polimorfismo con memoria segura: al terminar, todo se libera solo.

RESPUESTA 45.

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <stdexcept>

int main() {
    try {
        std::ifstream in("ventas.txt");
        if (!in.is_open()) {
            throw std::runtime_error("No se pudo abrir ventas.txt");
        }
        std::vector<double> ventas;
        std::string linea;
        while (std::getline(in, linea)) {
            if (!linea.empty()) ventas.push_back(std::stod(linea));
        }
        in.close();

        if (ventas.empty()) throw std::runtime_error("Sin datos");

        double total = 0;
        for (double v : ventas) total += v;
        double promedio = total / ventas.size();

        std::ofstream out("reporte.txt");
        if (!out.is_open()) throw std::runtime_error("No se pudo crear
reporte.txt");
        out << "Total: " << total << "\n";
        out << "Promedio: " << promedio << "\n";
        std::cout << "Reporte generado." << std::endl;
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
    return 0;
}
```

Explicación: lectura línea por línea con `getline`, conversión con `stod`, vector para acumular, y escritura del reporte. Cada fallo posible (archivo inexistente, sin datos, no se puede escribir) se convierte en excepción con mensaje claro.

RESPUESTA 46. Un puntero guarda una dirección y puede ser nulo o cambiar de objetivo; una referencia es un alias fijo que siempre refiere a algo válido y usa sintaxis limpia. Ejemplo donde la referencia gana: parámetros de función que se modifican o se leen sin copiar (`void procesar(const std::string& s)`). Ejemplo donde el puntero es necesario: cuando el dato puede no existir (búsqueda que devuelve `nullptr` si no encuentra) o con propiedad dinámica (`unique_ptr` para un objeto creado en tiempo de ejecución cuya vida controla el programa). Regla: referencia por defecto; puntero (inteligente) cuando necesites nulidad, reasignación o propiedad dinámica.

RESPUESTA 47.

```

#include <iostream>
#include <vector>

void ordenarBurbuja(std::vector<int>& v) {
    int n = v.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (v[j] > v[j + 1]) {
                int temp = v[j];
                v[j] = v[j + 1];
                v[j + 1] = temp;
            }
        }
    }
}

void mostrar(const std::vector<int>& v) {
    for (int x : v) std::cout << x << " ";
    std::cout << std::endl;
}

int main() {
    std::vector<int> v = {5, 2, 9, 1, 7};
    std::cout << "Antes: ";
    mostrar(v);
    ordenarBurbuja(v);
    std::cout << "Despues: ";
    mostrar(v);
    return 0;
}

```

Explicación: la burbuja compara vecinos y los intercambia si están desordenados; tras cada pasada del ciclo externo, el mayor "flota" al final (por eso $j < n-1-i$). El vector se pasa por referencia no constante porque la función lo modifica. En código real usarías `std::sort`, pero implementar burbuja enseña cómo piensan los algoritmos.

RESPUESTA 48.

```

#include <iostream>
#include <string>
#include <vector>
#include <stdexcept>

template <typename T>

```

```

class PilaSegura {
private:
    std::vector<T> datos;
    size_t capacidad;

public:
    PilaSegura(size_t cap) : capacidad(cap) {}

    void push(const T& valor) {
        if (datos.size() >= capacidad) {
            throw std::runtime_error("Pila llena");
        }
        datos.push_back(valor);
    }

    void pop() {
        if (datos.empty()) throw std::runtime_error("Pila vacia");
        datos.pop_back();
    }

    T cima() const {
        if (datos.empty()) throw std::runtime_error("Pila vacia");
        return datos.back();
    }

    bool vacia() const { return datos.empty(); }
};

int main() {
    try {
        PilaSegura<std::string> p(2);
        p.push("uno");
        p.push("dos");
        std::cout << "Cima: " << p.cima() << std::endl;
        p.push("tres"); // lanza: llena
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
    return 0;
}

```

Explicación: plantilla + vector + excepciones combinados: la pila es genérica, segura (verifica límites) y reporta errores con excepciones en vez de comportamientos indefinidos. cima() es const porque solo lee.

RESPUESTA 49. El encapsulamiento esconde los datos (private) y los expone por métodos que hacen cumplir las reglas. Con el atributo público, cualquiera puede poner años = -500 y el objeto queda en estado imposible sin que nadie se entere. Con setter validado, el objeto se protege solo:

```
class Edad {
private:
    int anios;
public:
    Edad(int a) { setAnios(a); }
    void setAnios(int a) {
        if (a >= 0 && a <= 150) anios = a;
    }
    int getAnios() const { return anios; }
};
```

El constructor usa el setter para no duplicar la validación. Si la regla cambia (por ejemplo, máximo 120), se cambia en un solo lugar. Además, depurar es fácil: los años solo pueden cambiar por setAnios.

RESPUESTA 50. (Programa completo; adáptalo y compílalo para verificar.)

```
#include <iostream>
#include <string>
#include <vector>
#include <fstream>

struct Contacto {
    std::string nombre;
    std::string telefono;
    std::string email;
};

class Agenda {
private:
    std::vector<Contacto> contactos;

    int indiceDe(const std::string& nombre) const {
        for (size_t i = 0; i < contactos.size(); i++) {
            if (contactos[i].nombre == nombre) return i;
        }
        return -1;
    }
};
```

```

public:
    void agregar(const Contacto& c) {
        if (c.nombre.empty()) {
            std::cout << "El nombre no puede estar vacio." << std::endl;
            return;
        }
        contactos.push_back(c);
        std::cout << "Agregado." << std::endl;
    }

    void buscar(const std::string& nombre) const {
        int i = indiceDe(nombre);
        if (i < 0) { std::cout << "No encontrado." << std::endl; return; }
        const Contacto& c = contactos[i];
        std::cout << c.nombre << " | " << c.telefono << " | " << c.email << std::endl;
    }

    void eliminar(const std::string& nombre) {
        int i = indiceDe(nombre);
        if (i < 0) { std::cout << "No encontrado." << std::endl; return; }
        contactos.erase(contactos.begin() + i);
        std::cout << "Eliminado." << std::endl;
    }

    void listar() const {
        if (contactos.empty()) { std::cout << "(vacía)" << std::endl; return; }
        for (const auto& c : contactos) {
            std::cout << c.nombre << " | " << c.telefono << " | " << c.email <<
std::endl;
        }
    }

    void guardar(const std::string& archivo) const {
        std::ofstream out(archivo);
        for (const auto& c : contactos) {
            out << c.nombre << "|" << c.telefono << "|" << c.email << "\n";
        }
    }

    void cargar(const std::string& archivo) {
        std::ifstream in(archivo);
        if (!in.is_open()) return;
        std::string linea;
        while (std::getline(in, linea)) {
            size_t p1 = linea.find('|');

```

```

        size_t p2 = linea.find('|', p1 + 1);
        if (p1 == std::string::npos || p2 == std::string::npos) continue;
        Contacto c;
        c.nombre = linea.substr(0, p1);
        c.telefono = linea.substr(p1 + 1, p2 - p1 - 1);
        c.email = linea.substr(p2 + 1);
        contactos.push_back(c);
    }
}
};

int main() {
    Agenda agenda;
    agenda.cargar("agenda.txt");
    int opcion;
    std::string nombre;

    do {
        std::cout << "\n1.Agregar 2.Buscar 3.Eliminar 4.Listar 0.Salir: ";
        std::cin >> opcion;
        std::cin.ignore();

        switch (opcion) {
            case 1: {
                Contacto c;
                std::cout << "Nombre: "; std::getline(std::cin, c.nombre);
                std::cout << "Telefono: "; std::getline(std::cin, c.telefono);
                std::cout << "Email: "; std::getline(std::cin, c.email);
                agenda.agregar(c);
                break;
            }
            case 2:
                std::cout << "Nombre: "; std::getline(std::cin, nombre);
                agenda.buscar(nombre);
                break;
            case 3:
                std::cout << "Nombre: "; std::getline(std::cin, nombre);
                agenda.eliminar(nombre);
                break;
            case 4:
                agenda.listar();
                break;
            case 0:
                break;
            default:

```

```

        std::cout << "Opcion invalida." << std::endl;
    }
} while (opcion != 0);

agenda.guardar("agenda.txt");
std::cout << "Agenda guardada. Adios." << std::endl;
return 0;
}

```

Explicación: integra struct, clase, vector, búsqueda lineal, archivos con formato separado por |, menú con do-while + switch y cin.ignore() antes de cada getline. Es un programa completo y funcional: compílalo, úsalo varias veces y verifica que los contactos persisten entre ejecuciones. Si llegaste hasta aquí y lo entiendes todo, el curso está dominado. ¡Felicidades!

Despedida

Terminaste el "Curso Completo de C++" de Lucas EGA Academy. Empezaste compilando tu primer "Hola, mundo" y terminaste diseñando sistemas orientados a objetos con persistencia. En el camino dominaste punteros, clases, herencia, polimorfismo, plantillas, la STL, excepciones y archivos: el arsenal completo del programador de C++.

Lo que sigue depende de ti: construye proyectos, contribuye a código abierto, sigue los cursos avanzados de la academia en lucasegaacademy.com y nunca dejes de compilar con -Wall. Nos vemos en el siguiente nivel. ¡A programar!

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