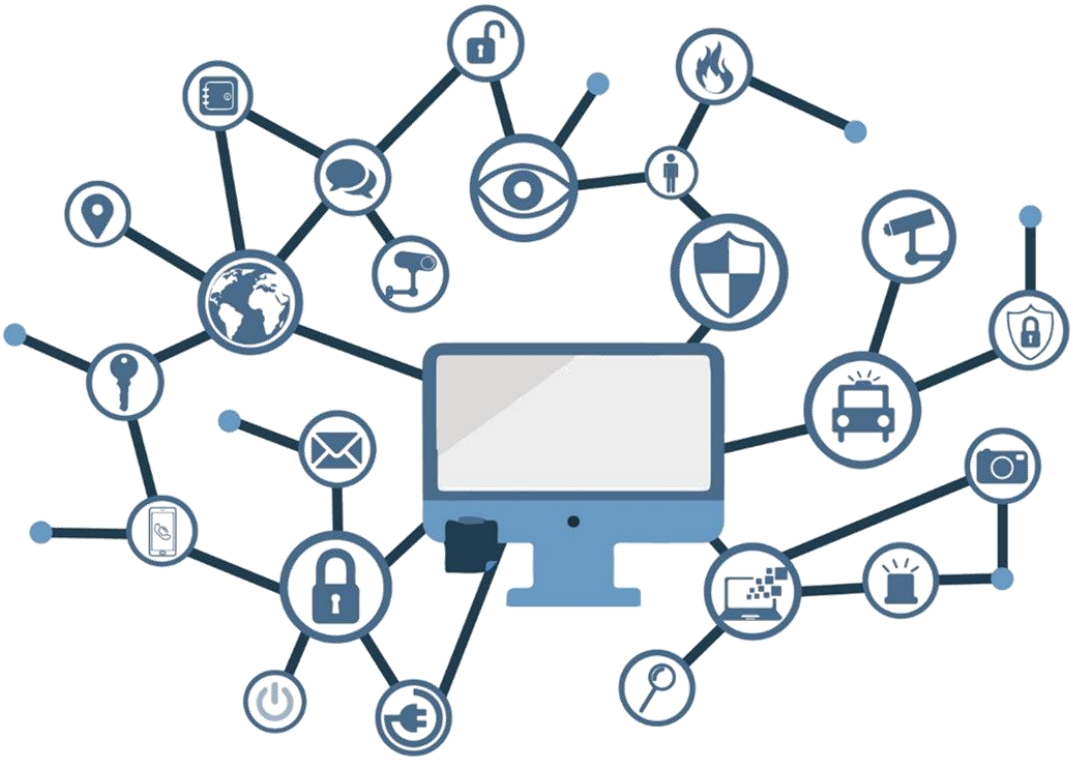
centro criptológico nacional

Guía de Desarrollo Seguro
CCN-STIC 481 – Anexo B

Codificación Segura para C++



Noviembre 2026

MINISTERIO
DE DEFENSA

cpage.mpr.gob.es

Catálogo de Publicaciones de la Administración General del Estado
cpage.mpr.gob.es

Edita:

MINISTERIO DE DEFENSA
SUBSECRETARÍA DE DEFENSA
SECRETARÍA GENERAL TÉCNICA

centro criptológico nacional

Pº de la Castellana 109, 28046 Madrid
© Centro Criptológico Nacional, 2026
NIPO (edición en línea)

CCN
c=ES, l=MADRID, o=CCN,
ou=SELLO ELECTRONICO,
2.5.4.97=VATES-S2800155J,
serialNumber=S2800155J,
cn=CCN
2026.11.23 11:35:52 +01:00

LIMITACIÓN DE RESPONSABILIDAD

El presente documento se proporciona de acuerdo con los términos en él recogidos, rechazando expresamente cualquier tipo de garantía implícita que se pueda encontrar relacionada. En ningún caso, el Centro Criptológico Nacional puede ser considerado responsable del daño directo, indirecto, fortuito o extraordinario derivado de la utilización de la información y software que se indican incluso cuando se advierta de tal posibilidad.

AVISO LEGAL

Quedan rigurosamente prohibidas, sin la autorización escrita del Centro Criptológico Nacional, bajo las sanciones establecidas en las leyes, la reproducción parcial o total de este documento por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares del mismo mediante alquiler o préstamo públicos.

Esta publicación se distribuye bajo licencia CC BY-NC-ND 4.0 que permite compartir el material en cualquier medio o formato con la condición de reconocer adecuadamente la procedencia «Edita: Ministerio de Defensa. Secretaría General Técnica.» No se puede modificar. No se puede utilizar con fines comerciales.

ÍNDICE

1. INTRODUCCIÓN 1

1.1 OBJETIVO 1
1.2 ALCANCE 1
1.3 ESTÁNDARES METODOLÓGICOS 1
1.3.1 C++ Core Guidelines 2
1.3.2 MISRA C++ 2023 Guidelines and Rules 2
1.3.3 SEI/CERT CPP Coding Standard 3
1.4 ESTRUCTURA DE LA GUÍA 3

2. REGLAS DE CODIFICACIÓN SEGURA 6

2.1 CONSTANTES 6
2.1.1 [cpp.const.1] - Evitar las variables globales no constantes 6
2.1.2 [cpp.const.2] - Declarar los objetos inmutables por defecto 10
2.1.3 [cpp.const.3] - Declarar "const" a toda función miembro siempre que no modifiquen el estado observable 14
2.1.4 [cpp.const.4] - Apuntar a "const" a todo puntero y referencia por defecto 19
2.1.5 [cpp.const.5] - Usar "constexpr", "consteval", "if consteval" para evaluaciones en tiempo de compilación 24
2.2 CADENAS DE CARACTERES 31
2.2.1 [cpp.str.1] - Usar "std::string" para almacenar cadenas 31
2.2.2 [cpp.str.2] - Evitar crear "std::string" a partir de un puntero nulo 36
2.2.3 [cpp.str.3] - Comprobar los límites en el acceso a cadenas 42
2.2.4 [cpp.str.4] - Usar "zstring" o tipos equivalentes para cadenas estilo C 48
2.3 ENUMERACIONES 54
2.3.1 [cpp.enum.1] - Preferir enumeraciones frente a macros para definir conjuntos de valores 54
2.3.2 [cpp.enum.2] - Preferir "enum class" frente a enumeraciones no fuertemente tipadas 60
2.3.3 [cpp.enum.3] - Evitar convertir a un valor de enumeración fuera de su rango válido..... 65
2.4 CONVERSIONES 69
2.4.1 [cpp.conv.1] - Evitar conversiones que cambien el signo o la categoría del tipo..... 69
2.4.2 [cpp.conv.2] - Usar "nullptr" como única constante de puntero nulo..... 73
2.4.3 [cpp.conv.3] - Evitar los "casts" o usar los casts prefijados 77
2.4.4 [cpp.conv.4] - Evitar utilizar "reinterpret_cast" 81
2.4.5 [cpp.conv.5] - Evitar eliminar "const" mediante "casts" 86
2.4.6 [cpp.conv.6] - Evitar convertir un puntero a una clase incompleta ni liberarlo 90
2.4.7 [cpp.conv.7] - Utilizar std::variant para representar alternativas 94
2.5 INICIALIZACIÓN 99
2.5.1 [cpp.ini.1] - Inicializar todos los objetos antes de su uso 99
2.5.2 [cpp.ini.2] - Evitar declarar o definir identificadores reservados 104
2.5.3 [cpp.ini.3] - Asegurar que las operaciones de movimiento dejan el objeto origen en un estado válido 108
2.5.4 [cpp.ini.4] - Asegurar que las operaciones de movimiento deben ser "noexcept" siempre que sea posible 114
2.5.5 [cpp.ini.5] - Evitar realizar slicing en objetos derivados 120
2.5.6 [cpp.ini.6] - Evitar asignar arrays de objetos derivados a punteros de la clase base 125
2.5.7 [cpp.ini.7] - Evitar el uso de uniones 130
2.5.8 [cpp.ini.8] - Evitar ciclos durante la inicialización de objetos estáticos 135
2.6 EXPRESIONES 140

2.6.1 [cpp.exp.1] - Evitar que las expresiones dependan de un orden de evaluación por defecto 140
2.6.2 [cpp.exp.2] - Evitar depender del orden de evaluación de los argumentos de una función 144
2.6.3 [cpp.exp.3] - Usar paréntesis para clarificar expresiones complejas 149
2.6.4 [cpp.exp.4] - Asegurar que el operando derecho de && y || no tenga efectos laterales persistentes 153
2.6.5 [cpp.exp.5] - Evitar copiar datos entre objetos solapados 158
2.6.6 [cpp.exp.6] - Evitar utilizar el operador "coma" 162
2.7 ARITMÉTICA 166
2.7.1 [cpp.arit.1] - Evitar mezclar aritméticas con signo y sin signo 166
2.7.2 [cpp.arit.2] - Evitar condiciones de overflow 170
2.7.3 [cpp.arit.3] - Evitar condiciones de underflow 175
2.7.4 [cpp.arit.4] - Evitar condiciones de división por cero 179
2.7.5 [cpp.arit.5] - Evitar usar "unsigned" para evitar el uso de valores negativos 183
2.7.6 [cpp.arit.6] - Evitar condiciones de wrap-around en aritmética sin signo con constantes 187
2.7.7 [cpp.arit.7] - Evitar usar aritmética aditiva sobre iteradores si el resultado puede desbordar 192
2.7.8 [cpp.arit.8] - Evitar usar aritmética de punteros sobre objetos polimórficos 196
2.8 ESTRUCTURA DE CONTROL Y FLUJO 202
2.8.1 [cpp.flow.1] - Usar índices, referencias e iteradores válidos 202
2.9 CLASES 207
2.9.1 [cpp.cls.1] - Exponer lo mínimo a los miembros de una clase 207
2.9.2 [cpp.cls.2] - Evitar definir una clase por defecto si no necesita operaciones especiales ... 213
2.9.3 [cpp.cls.3] - Definir un destructor en una clase con recursos 217
2.9.4 [cpp.cls.4] - Extremar la precaución del uso de una propiedad de clase con punteros o referencias 223
2.9.5 [cpp.cls.5] - Establecer al destructor de clase público y virtual, o protegido y no virtual . 228
2.9.6 [cpp.cls.6] - Usar el constructor para crear un objeto completamente inicializado 235
2.9.7 [cpp.cls.7] - Establecer en el constructor todos los invariantes de la clase 239
2.9.8 [cpp.cls.8] - Lanzar una excepción si un constructor no puede crear un objeto válido 244
2.9.9 [cpp.cls.9] - Nunca lanzar excepciones desde un destructor 250
2.9.10 [cpp.cls.10] - Liberar todos los recursos adquiridos por la clase desde el destructor ... 256
2.9.11 [cpp.cls.11] - Evitar llamar funciones virtuales desde constructores o destructores ... 262
2.10 FUNCIONES 268
2.10.1 [cpp.func.1] - Declarar las precondiciones de una función usando mecanismos formales 268
2.10.2 [cpp.func.2] - Declarar las postcondiciones de una función usando mecanismos formales 273
2.10.3 [cpp.func.3] - Señalizar forma explícita cuando una función no pueda cumplir su tarea 277
2.10.4 [cpp.func.4] - Asegurar el uso de todo valor devuelto por una función "no void" 282
2.10.5 [cpp.func.5] - Evitar mantener código inalcanzable en una función 287
2.10.6 [cpp.func.6] - Asegurar que toda función "no void" devuelve un valor en todas sus rutas de ejecución 291
2.10.7 [cpp.func.7] - Evitar definir una función variádica en estilo C 296
2.10.8 [cpp.func.8] - Asegurar que las expresiones de control no sean invariantes 301
2.10.9 [cpp.func.9] - Preferir expresiones lambda frente a "std::bind" 305
2.10.10 [cpp.func.10] - Utilizar "std::optional" para representar valores opcionales 309
2.11 PUNTEROS 314
2.11.1 [cpp.ptr.1] - Usar "unique_ptr" para expresar exclusividad en la propiedad 314

2.11.2 [cpp.ptr.2] - Usar "shared_ptr" cuando sea necesario compartir propiedad 319
2.11.3 [cpp.ptr.3] - Usar "weak_ptr" para romper los ciclos de "shared_ptr" 324
2.11.4 [cpp.ptr.4] - Evitar almacenar un puntero con propietario en un puntero inteligente no relacionado 329
2.11.5 [cpp.ptr.5] - Usar punteros y referencias crudos solo para accesos no propietarios ... 334
2.11.6 [cpp.ptr.6] - Evitar transferir la propiedad mediante punteros o referencias crudas .. 339
2.11.7 [cpp.ptr.7] - Declarar explícitamente los punteros que no pueden ser nulos 344
2.11.8 [cpp.ptr.8] - Evitar pasar arrays como punteros simples 349
2.11.9 [cpp.ptr.9] - Usar "std::span" para representar secuencias de elementos 354
2.11.10 [cpp.ptr.10] - Evitar devolver un puntero o referencia a un objeto automático local .. 358
2.11.11 [cpp.ptr.11] - Evitar asignar la dirección de un objeto automático a uno con mayor duración 362
2.11.12 [cpp.ptr.12] - Evitar acceder a un objeto fuera de su tiempo de vida 367
2.11.13 [cpp.ptr.13] - Evitar desreferenciar punteros inválidos 372
2.11.14 [cpp.ptr.14] - Asegurar que los accesos a contenedores estén siempre dentro de rango 376
2.12 GESTIÓN DE MEMORIA 381
2.12.1 [cpp.mem.1] - Evitar el uso de "malloc" y "free" 381
2.12.2 [cpp.mem.2] - Evitar el uso directo de "new" y "delete" salvo encapsulación clara y justificada 385
2.12.3 [cpp.mem.3] - Evitar acceder a memoria ya liberada 389
2.12.4 [cpp.mem.4] - Borrar de forma segura la memoria con información sensible antes de su liberación 394
2.13 GESTIÓN DE RECURSOS 398
2.13.1 [cpp.res.1] - Gestionar automáticamente los recursos mediante RAII 398
2.13.2 [cpp.res.2] - Transferir al RAII toda asignación explícita 402
2.13.3 [cpp.res.3] - Los archivos deben cerrarse cuando ya no sean necesarios 407
2.14 GESTIÓN DE ERRORES 411
2.14.1 [cpp.err.1] - Gestionar todas las excepciones adecuadamente 411
2.14.2 [cpp.err.2] - Asegurar la liberación de recursos al gestionar excepciones 416
2.14.3 [cpp.err.3] - Evitar lanzar excepciones a través de límites de ejecución 420
2.14.4 [cpp.err.4] - Evitar finalizar abruptamente ante errores recuperables 426
2.14.5 [cpp.err.5] - Garantizar la seguridad frente a excepciones 431
2.14.6 [cpp.err.6] - Utilizar std::expected para devolver un valor o un error cuando no se utilicen excepciones 436
2.15 CONCURRENCIA 441
2.15.1 [cpp.sync.1] - Asumir el código como potencialmente multihilo 441
2.15.2 [cpp.sync.2] - Evitar condiciones de carrera 447
2.15.3 [cpp.sync.3] - Minimizar el uso compartido de datos modificables 453
2.15.4 [cpp.sync.4] - Evitar usar "volatile" como mecanismo de sincronización 458
2.15.5 [cpp.sync.5] - Gestionar los bloqueos mediante RAII 463
2.15.6 [cpp.sync.6] - Evitar llamar a código no controlado durante un bloqueo 468
2.15.7 [cpp.sync.7] - Evitar producir interbloqueos bloqueando recursos en orden inconsistente 474
2.15.8 [cpp.sync.8] - Evitar destruir un mutex mientras esté bloqueado 480
2.15.9 [cpp.sync.9] - Evitar desacoplar hilos mediante detach 484
2.15.10 [cpp.sync.10] - Evitar una espera sin una condición asociada 488
2.15.11 [cpp.sync.11] - Evitar implementar doble comprobación de bloqueo para inicialización 492
2.15.12 [cpp.sync.12] - Usar variables de condición para preservar la seguridad y el avance del programa 499

2.16 NÚMEROS ALEATORIOS....504
2.16.1 [cpp.rand.1] - Utilizar generadores criptográficamente seguros....504
2.17 BIBLIOTECAS....508
2.17.1 [cpp.lib.1] - Sustituir funciones de libc por utilidades seguras de la stdlib en C++....508

3. ANEXO: METODOLOGÍA DE CÁLCULO DEL SCORE CWSS....515

1. Introducción

1.1 Objetivo

El propósito de este documento es ofrecer un **catálogo exhaustivo de controles técnicos de ciberseguridad**, diseñado para servir como referencia práctica en la implementación de medidas de seguridad efectivas durante el desarrollo de software.

Dicho catálogo se encuentra alineado con las mejores prácticas en materia de desarrollo seguro y tiene como objetivo **proporcionar soluciones técnicas seguras**, validadas y contrastadas frente a las vulnerabilidades más comunes que pueden presentarse en cada ámbito de la implementación.

El documento pretende facilitar la **integración sistemática de la seguridad en el ciclo de vida del software** (S-SDLC), promoviendo un enfoque preventivo, estandarizado y verificable que contribuya a la reducción de riesgos y a la mejora continua.

1.2 Alcance

La correcta comprensión y aplicación de **los principios** contenidos en este documento **son de carácter obligatorio**, ya que su cumplimiento garantiza la alineación con los estándares, políticas internas y regulaciones de seguridad vigentes, así como con las mejores prácticas internacionales en materia de desarrollo seguro.

Se recomienda la lectura y conocimiento **a todos los miembros del equipo de desarrollo**, incluyendo aquellos que participen directa o indirectamente en cualquiera de las fases restantes del S-SDLC, como análisis de requisitos, pruebas, despliegue u operaciones. Esto permitirá fomentar una cultura de seguridad transversal, asegurando que todos los involucrados comprendan la importancia de integrar la seguridad como un elemento esencial a lo largo de todo el ciclo de vida del software.

1.3 Estándares metodológicos

Las **reglas de codificación segura** descritas en este documento **se fundamentan en las recomendaciones y buenas prácticas** definidas por las guías más utilizadas y mejor respaldadas por la industria.

Cada regla hace referencia explícita a la fuente normativa o estándar de seguridad que lo respalda, asegurando su trazabilidad y su alineación con los marcos regulatorios vigentes.

1.3.1 C++ Core Guidelines

https://cpp-core-guidelines-docs.vercel.app/

Son un conjunto de directrices elaboradas por expertos de la comunidad de C++, incluidos miembros destacados como Bjarne Stroustrup, y mantenidas de forma colaborativa en GitHub. Su objetivo es proporcionar reglas claras, prácticas y modernas para escribir código C++ seguro, eficiente, mantenible y robusto. Las guías cubren aspectos esenciales como gestión de recursos, uso seguro de punteros, diseño orientado a objetos, concurrencia, manejo de errores, abstracciones eficientes y prácticas recomendadas del lenguaje moderno (C++11 en adelante). Además, incluyen soporte para verificación automática mediante herramientas como clang-tidy o analizadores estáticos capaces de aplicar las reglas de forma sistemática. Las C++ Core Guidelines buscan estandarizar buenas prácticas, reducir defectos comunes y mejorar la calidad del software en proyectos de cualquier tamaño.

Las referencias a las reglas de este estándar aparecerán descritas con un formato similar al de los siguientes ejemplos:

- **C++ Core-I.5**: State preconditions (if any).
- **C++ Core-I.12**: Declare a pointer that must not be null as not_null.
- **C++ Core-F.50**: Use a lambda when a function won't do (to capture local variables, or to write a local function).

1.3.2 MISRA C++ 2023 Guidelines and Rules

https://es.mathworks.com/help/bugfinder/misra-cpp-2023-rules-and-directives.html

Constituyen un estándar industrial ampliamente adoptado para el desarrollo seguro y fiable de software en C++ dentro de sistemas embebidos y entornos críticos, como automoción, aeroespacial, defensa o maquinaria industrial. Este marco establece un conjunto riguroso de reglas y directrices destinadas a minimizar comportamientos indefinidos, reducir ambigüedades del lenguaje y evitar patrones de codificación propensos a errores. La versión 2023 actualiza y amplía el estándar para alinearse con C++ moderno, integrando controles más estrictos sobre tipos, memoria, concurrencia, excepciones, inicialización y uso de bibliotecas. Las reglas MISRA son verificables, generalmente mediante herramientas automáticas de análisis estático, y proporcionan una base sólida para demostrar conformidad normativa, mejorar la calidad del software, incrementar la seguridad funcional y asegurar un comportamiento predecible en sistemas donde la fiabilidad es esencial.

Las referencias a las reglas de este estándar aparecerán descritas con un formato similar al de los siguientes ejemplos:

- **MISRA CPP-7.0.5**: Usual arithmetic conversions shall not change signedness/type category.

- **MISRA CPP-12.3.1**: union shall not be used.
- **MISRA CPP-8.0.1**: Parentheses should clarify expression meaning.

1.3.3 SEI/CERT CPP Coding Standard

https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Es un conjunto de normas desarrollado por el Software Engineering Institute (SEI) de Carnegie Mellon para promover la creación de software en C++ seguro, fiable y libre de errores críticos. Este estándar proporciona reglas y recomendaciones que abordan problemas comunes del lenguaje, incluyendo gestión segura de memoria, prevención de accesos fuera de rango, manipulación correcta de punteros, uso seguro de hilos y concurrencia, control robusto de excepciones, manejo adecuado de tipos y restricciones estrictas sobre comportamientos indefinidos o no especificados. Cada regla está acompañada de una justificación técnica, ejemplos de código correcto e incorrecto y, en muchos casos, métodos para la detección automatizada mediante herramientas de análisis estático. Su objetivo es reducir vulnerabilidades, mejorar la estabilidad del software y reforzar la seguridad en sistemas donde la confianza y predictibilidad del comportamiento son esenciales.

Las referencias a las reglas de este estándar aparecerán descritas con un formato similar al de los siguientes ejemplos:

- **CERT/CPP-CTR52**: Guarantee that library functions do not overflow.
- **CERT/CPP-CTR53**: Use valid iterator ranges.
- **CERT/CPP-OOP50**: Do not invoke virtual functions from constructors or destructors.

1.4 Estructura de la guía

La guía está constituida por una lista de reglas de codificación segura, organizada en las siguientes categorías:

- Principios generales.
- Constantes.
- Cadenas de caracteres.
- Enumeraciones.
- Conversiones.
- Inicialización.
- Expresiones.
- Aritmética.

- Estructura de control y flujo.
- Clases.
- Funciones.
- Punteros.
- Gestión de memoria.
- Gestión de recursos.
- Gestión de errores.
- Concurrencia.
- Números aleatorios.
- Bibliotecas.

Los epígrafes de cada categoría representan reglas de la categoría. Cada una de ellas definida por un **identificador** y el **nombre** de la regla.

Debajo de cada regla, se muestra una pequeña tabla, a **modo de cabecera**, que resume las características más relevantes como son:

- **ID**: identificador de la regla.
- **Uso:** obligatorio o recomendado.
- **CWSS Score**: la puntuación asignada mediante evaluación CWSS (0-100).
- **Impacto técnico**: la severidad del daño que podría producir una codificación insegura de la regla (Muy alto, Alto, Medio, Bajo, Muy bajo). Su impacto queda reflejado en la puntuación del CWSS.
- **Dimensión**: la dimensión de seguridad afectada (Integridad, Confidencialidad, Disponibilidad).
- **S-SDLC**: la fase del S-SDLC donde debería tenerse presente la regla de seguridad. De entre todas las fases, los únicos valores afectados son **Design**, **Build** y **Test**.

ID	**cpp.const.1**	**Uso**	**Obligatorio**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Detalle gráfico de todos los campos de la cabecera de cada regla

Después de la cabecera se muestra la siguiente información:

- Una **breve descripción**, que pondrá en antecedentes al usuario sobre el significado de la regla.
- Dos ejemplos compilables de código: una **solución no recomendada** y la **solución segura recomendada**.
- Una descripción extensa y detallada para profundizar en el significado de la regla, y las consecuencias de ignorarla durante el desarrollo.
- Una **lista** con todos los **riesgos de seguridad** en los que se incurriría en caso de ignorar la regla.
- Una **lista de las buenas prácticas** recomendadas para aplicar la regla en las situaciones más comunes junto con ejemplos de malas y buenas implementaciones.
- Los **estándares metodológicos** a los que hace referencia.

En todos los ejemplos de código, las soluciones no recomendadas se mostrarán siempre en **color rojo** para que puedan ser identificarlas rápidamente como implementaciones a evitar. Y las soluciones seguras, o recomendadas, en **color azul**.

2. Reglas de codificación segura

2.1 Constantes

2.1.1 [cpp.const.1] - Evitar las variables globales no constantes

ID	cpp.const.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

El empleo de variables globales, que no sean constantes, representa un riesgo para la seguridad, la integridad del estado interno y la mantenibilidad del software. Las variables globales modificables facilitan condiciones de carrera, accesos no autorizados, comportamientos impredecibles y acoplamiento excesivo entre componentes. Se recomienda reemplazarlas con alternativas seguras como parámetros, clases con encapsulación, inmutabilidad o un ámbito de uso limitado, con el fin de reducir la superficie de ataque en aplicaciones complejas.

Codificación insegura

Uso de una variable global modificable para compartir estado entre funciones. Se busca mostrar cómo el acceso global facilita cambios no controlados desde cualquier parte del código, generando acoplamiento fuerte, efectos secundarios inesperados y dificultades para mantener o probar la aplicación.

```
#include <iostream>

//-- Variable global modificable accesible desde cualquier parte del programa
int contadorGlobal = 0;

//-- Incrementa el contador global
void incrementar() {
  //-- Modifica directamente el estado global
  contadorGlobal++;
}

//-- Reinicia el contador global
void reiniciar() {
  //-- Cualquier función puede alterar el valor sin restricciones
  contadorGlobal = 0;
}

int main() {
  //-- Uso del contador global
  incrementar();
  incrementar();
```

```
  std::cout << "Valor actual: " << contadorGlobal << std::endl;
  reiniciar();
  std::cout << "Valor tras reinicio: " << contadorGlobal << std::endl;
  return 0;
}
```

Codificación recomendada

Encapsulamiento del estado dentro de una clase y eliminación de variables globales modificables. Se muestra cómo limitar el alcance de los datos, controlar su acceso mediante métodos públicos y reducir efectos secundarios no deseados, mejorando la mantenibilidad y la seguridad del diseño.

```
#include <iostream>

//-- Clase que encapsula el contador
class Contador {
  //-- Atributo privado, no accesible directamente desde fuera
  int valor;

public:
  //-- Constructor que inicializa el estado
  Contador() : valor(0) {}

  //-- Incrementa el contador de forma controlada
  void incrementar() {
    valor++;
  }

  //-- Reinicia el contador de forma controlada
  void reiniciar() {
    valor = 0;
  }

  //-- Devuelve el valor actual sin permitir modificación directa
  int obtenerValor() const {
    return valor;
  }
};

int main() {
  //-- Instancia local, sin variables globales modificables
  Contador contador;
  contador.incrementar();
  contador.incrementar();
  std::cout << "Valor actual: " << contador.obtenerValor() << std::endl;
  contador.reiniciar();
  std::cout << "Valor tras reinicio: " << contador.obtenerValor() << std::endl;
```

```
  return 0;
}
```

2.1.1.1 Descripción

El uso de variables globales no constantes podría representar una fuente significativa de problemas de seguridad, calidad y mantenibilidad. Disponer de accesos ubicuos y sin restricciones a estas variables incrementa la probabilidad de errores lógicos, corrupción del estado interno y comportamientos no deterministas entre módulos, permitiendo que cualquier parte del programa altere su valor sin un mecanismo de control adecuado. Esto podría causar vulnerabilidades difíciles de rastrear. Es importante limitar su uso exclusivamente a aquellos escenarios donde su presencia sea estrictamente necesaria, y se recomienda implementar soluciones alternativas más seguras siempre que se pueda.

Riesgos asociados

- **Condiciones de carrera**: múltiples hilos pueden modificar el mismo estado global sin sincronización adecuada.
- **Corrupción del estado**: cualquier error en un módulo puede afectar a otros al alterar accidentalmente la variable global.
- **Acoplamiento excesivo**: los componentes quedan dependientes de variables que no controlan directamente.
- **Dificultad de pruebas**: los tests se vuelven menos deterministas porque dependen del estado global compartido.
- **Superficie de ataque ampliada**: una variable global accesible incrementa el riesgo de abuso por parte de módulos comprometidos.

Buenas prácticas

- Limitar el alcance de las variables utilizando variables locales en lugar de globales.

```
//-- Variable global modificable accesible desde cualquier parte
int contador = 0;
```

```
//-- Variable local limitada al ámbito de la función
void procesar() {
  int contador = 0;
  contador++;
}
```

- Si se requiere acceso global, declarar la variable como constante.

```
//-- Variable global modificable que puede alterarse accidentalmente
int tiempo_espera = 30;
```

```
//-- Variable global inmutable que evita modificaciones no autorizadas
const int TIEMPO_ESPERA = 30;
```

- Encapsular el estado global dentro de clases o estructuras con control de acceso.

```
//-- Variable global expuesta directamente
int saldo = 1000;
```

```
//-- Estado encapsulado con control mediante métodos públicos
class Cuenta {
  private:
    int saldo;
  public:
    Cuenta(int s) : saldo(s) {}
    int obtenerSaldo() const { return saldo; }
};
```

- Utilizar inyección de dependencias para evitar dependencias implícitas de variables globales.

```
//-- Dependencia implícita de variable global compartida
Config config_global;

void iniciar() {
  usar(config_global);
}
```

```
//-- Dependencia explícita pasada como parámetro
void iniciar(Config& config) {
  usar(config);
}
```

2.1.1.2 Estándares metodológicos

- **C++ Core-I.2**: Avoid non-const global variables.
- **C++ Core-R.6**: Avoid non-const global variables.

2.1.2 [cpp.const.2] - Declarar los objetos inmutables por defecto

ID	cpp.const.2	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Se recomienda la declaración de los objetos de forma inmutable por defecto, con el fin de limitar los escenarios donde estrictamente la mutabilidad sea necesaria. Esta solución reduce la superficie de ataque asociada a modificaciones inesperadas del estado interno, previene efectos laterales no controlados y refuerza la **integridad** del software. Esta práctica es especialmente relevante en contextos concurrentes, componentes críticos y sistemas donde la predictibilidad del estado es esencial para la seguridad y la robustez del código.

Codificación insegura

Gestión de variables y objetos que no cambian después de su inicialización. La ausencia de inmutabilidad permite modificaciones accidentales que alteran el estado del programa, y cómo el uso adecuado de `const` ayuda a proteger los datos y hacer el código más predecible.

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

int main() {
  //-- Variable que representa un límite que no debería cambiar
  int maxConnections = 100;

  //-- Objeto que representa el nombre de la aplicación
  std::string appName = "SecureServer";

  //-- Modificación accidental del límite
  maxConnections = 1000;

  //-- Modificación accidental del nombre
  appName = "TestServer";

  //-- Uso de valores modificados inesperadamente
  std::cout << "Max connections: " << maxConnections << std::endl;
  std::cout << "Application name: " << appName << std::endl;

  return 0;
}
```

Codificación recomendada

Protección de variables y objetos que no deben cambiar durante la ejecución mediante el uso de `const`. Se evita la modificación accidental del estado, mejorando la seguridad, la mantenibilidad y facilitando que el compilador detecte errores en tiempo de compilación.

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

int main() {
  //-- Variable declarada como inmutable
  const int maxConnections = 100;

  //-- Objeto declarado como inmutable
  const std::string appName = "SecureServer";

  //-- Intentar modificar estas variables produciría error de compilación
  //-- maxConnections = 1000;
  //-- appName = "TestServer";

  //-- Uso seguro de valores constantes
  std::cout << "Max connections: " << maxConnections << std::endl;
  std::cout << "Application name: " << appName << std::endl;

  return 0;
}
```

2.1.2.1 Descripción

La inmutabilidad constituye una de las prácticas más sólidas para reforzar la seguridad y el comportamiento predecible del software. Al declarar los objetos como inmutables por defecto, el desarrollador minimiza la posibilidad de que el estado interno de una instancia cambie de forma inesperada debido a modificaciones accidentales, accesos concurrentes descoordinados o errores lógicos que faciliten manipulación no autorizada del flujo de datos. Esta práctica está basada en los principios de diseño seguro al reducir el número de variables que pueden ser alteradas y limitar el impacto de un abuso de punteros, referencias o aliasing no controlado.

La inmutabilidad se logra principalmente mediante el uso extensivo de la palabra clave `const` aplicada tanto a objetos como a métodos, parámetros y vistas sobre datos. La intención es clara: si una operación no necesita modificar el estado interno, no debe tener permitido hacerlo. Se reduce drásticamente la complejidad cognitiva del código, se mejora la trazabilidad de dependencias y se facilita la verificación tanto formal como dinámica del comportamiento del programa bajo diferentes condiciones de ejecución.

La inmutabilidad proporciona múltiples beneficios:

- Contribuye a la **integridad**, evitando modificaciones no contempladas que puedan afectar cálculos, estados o decisiones críticas,
- Mejora la **disponibilidad,** al reducir los defectos relacionados con condiciones de carrera que suelen producir fallos en tiempo de ejecución.
- Limita la superficie de ataque en manipulaciones de objetos compartidos, especialmente en estructuras complejas como contenedores, caches, árboles de decisión, configuraciones o modelos de datos.

Riesgos asociados

- El aliasing de referencias puede permitir modificaciones accidentales o maliciosas al estado interno.
- El uso de punteros que apuntan a regiones compartidas sin const-correctness habilita al atacante a alterar estructuras que deberían ser solo de lectura.
- El acceso concurrente a objetos mutables provoca condiciones de carrera, que pueden derivar en corrupción de memoria, fallos lógicos o estados inconsistentes utilizados para escalar privilegios o alterar flujos de ejecución críticos.

Buenas prácticas

- Declarar las variables como inmutables por defecto para evitar modificaciones accidentales del estado.

```
//-- variable modificable que puede alterar el flujo esperado
String rol = obtenerRolUsuario();
rol = "ADMIN";
```

```
//-- variable declarada como inmutable desde su creación
final String rolSeguro = obtenerRolUsuario();
```

- Marcar como `final` los atributos de clase que no deben cambiar tras la construcción del objeto.

```
//-- atributo modificable desde cualquier método de la clase
class Usuario {
  String id;
}
```

```
//-- atributo inmutable asignado únicamente en el constructor
class Usuario {
  final String id;
  Usuario(String id) {
    this.id = id;
```

```
  }
}
```

- Evitar exponer referencias mutables internas; usar copias defensivas y campos finales.

```
//-- exposición directa de objeto mutable interno
class Pedido {
  private Date fecha;
  Date getFecha() {
    return fecha;
  }
}
```

```
//-- atributo final y devolución de copia defensiva
class Pedido {
  private final Date fecha;
  Pedido(Date fecha) {
    this.fecha = new Date(fecha.getTime());
  }
  Date getFecha() {
    return new Date(fecha.getTime());
  }
}
```

- Utilizar colecciones inmutables cuando no se requieran modificaciones.

```
//-- lista modificable compartida entre componentes
List<String> roles = new ArrayList<>();
roles.add("USER");
```

```
//-- lista inmutable creada desde el inicio
final List<String> rolesSeguros = List.of("USER");
```

- Definir clases como inmutables cuando representen datos de solo lectura.

```
//-- clase con setters que permiten modificar el estado interno
class Configuracion {
  private String entorno;
  void setEntorno(String entorno) {
    this.entorno = entorno;
  }
}
```

```
//-- clase inmutable sin setters y con atributos finales
final class Configuracion {
```

```
  private final String entorno;
  Configuracion(String entorno) {
    this.entorno = entorno;
  }
  String getEntorno() {
    return entorno;
  }
}
```

2.1.2.2 Estándares metodológicos

- **C++ Core-Con.1**: By default, make objects immutable.

2.1.3 [cpp.const.3] - Declarar "const" a toda función miembro siempre que no modifiquen el estado observable

ID	cpp.const.3	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Design

Se recomienda marcar como `const` todas las funciones miembro de una clase que no modifiquen el estado observable del objeto, reforzando así la inmutabilidad lógica y permitiendo garantías formales sobre el comportamiento del código. Este principio mejora la verificabilidad, reduce el riesgo de mutaciones inesperadas y facilita la detección de vulnerabilidades relacionadas con corrupción de datos, aliasing imprevisto o efectos secundarios no controlado, incrementa la robustez de la API interna y externa, favorece la programación defensiva, y permite las optimizaciones seguras del compilador.

Codificación insegura

Se muestra una clase que modela una cuenta bancaria con un método para consultar el saldo. El objetivo es observar el impacto de no marcar como `const` un método que no modifica el estado interno, y cómo esto limita su uso cuando el objeto es constante.

```
#include <iostream>

//-- Clase que representa una cuenta bancaria simple
class CuentaBancaria {
  private:
    double saldo;

  public:
    //-- Constructor que inicializa el saldo
    CuentaBancaria(double s) : saldo(s) {}
```

```
    //-- Método para obtener el saldo (no modifica el estado)
    double obtenerSaldo() {
      return saldo;
    }
};

int main() {
  //-- Objeto constante de la clase CuentaBancaria
  const CuentaBancaria cuenta(1000.0);

  //-- Error de compilación: el método no está declarado como const
  std::cout << "Saldo: " << cuenta.obtenerSaldo() << std::endl;

  return 0;
}
```

Codificación recomendada

Se muestra la misma clase, asegurando que el método que solo consulta el estado esté declarado como `const`. El objetivo es permitir su uso sobre objetos constantes y reforzar la intención de que no modifica el estado observable, mejorando la seguridad y claridad del diseño.

```
#include <iostream>

//-- Clase que representa una cuenta bancaria simple
class CuentaBancaria {
  private:
    double saldo;

  public:
    //-- Constructor que inicializa el saldo
    CuentaBancaria(double s) : saldo(s) {}

    //-- Método para obtener el saldo sin modificar el estado
    double obtenerSaldo() const {
      return saldo;
    }
};

int main() {
  //-- Objeto constante de la clase CuentaBancaria
  const CuentaBancaria cuenta(1000.0);

  //-- Llamada válida porque el método está declarado como const
  std::cout << "Saldo: " << cuenta.obtenerSaldo() << std::endl;

  return 0;
```

```
}
```

2.1.3.1 Descripción

Declarar funciones miembro como `const` cuando no modifican el estado observable del objeto contribuye directamente a preservar la integridad interna del software. En el contexto de la codificación segura, esta regla evita mutaciones no intencionadas que podrían dar lugar a comportamientos inconsistentes, errores lógicos o vulnerabilidades basadas en corrupción de memoria. El enfoque de inmutabilidad lógica impide que funciones aparentemente de solo lectura introduzcan efectos colaterales, permitiendo a los desarrolladores, herramientas de análisis estático y compiladores asumir garantías sólidas sobre el comportamiento del programa.

La ausencia de la palabra clave `const` en funciones que conceptualmente no alteran el estado también puede causar que objetos sean tratados erróneamente como mutables en contextos críticos. Por ejemplo, estructuras de datos compartidas entre hilos o componentes deben respetar estrictamente la inmutabilidad. Sin esta protección, podría existir una ventana de explotación donde un atacante desencadena condiciones de carrera, mutaciones inesperadas o lecturas inconsistentes. Marcar funciones como `const` permite al compilador imponer que ninguna variable miembro accesible sea modificada, evitando comportamientos indeterminados y reforzando la seguridad del sistema.

Cuando se declaran funciones miembro como `const` permite su uso con objetos igualmente constantes, proporcionando un **contrato explícito de no modificación** que incrementa la robustez del diseño. Sin esta garantía, la API interna tiende a degradarse hacia estructuras más permisivas, donde cualquier componente puede modificar cualquier estado, lo que debilita los límites de encapsulación. Es crucial que las funciones accedentes, getters, consultas sobre el estado u operaciones puramente de inspección se declaren como `const` para reforzar la claridad semántica y prevenir ambigüedades potencialmente peligrosas.

Es importante considerar distinguir entre **inmutabilidad física e inmutabilidad lógica**. La palabra clave `mutable` permite excepciones justificadas, pero su uso indiscriminado es un riesgo de seguridad. Solo debe aplicarse para miembros que pertenecen exclusivamente a la infraestructura técnica, como caches de cálculo internamente protegidas o contadores de uso thread-safe, pero nunca para datos sensibles o estados relevantes para el comportamiento visible del objeto.

Riesgos asociados

- **Inconsistencias internas de objetos**: errores causados por modificaciones inesperadas.
- **Fugas de estado mutable**: APIs que permiten alterar datos sensibles de forma

involuntaria.

- **Condiciones de carrera**: alteraciones simultáneas del estado donde debería existir inmutabilidad lógica.
- **Bypass de auditorías internas**: funciones que se asume no modifican estado, pero en realidad sí lo hacen.
- **Errores de aliasing**: punteros o referencias a estados mutables donde no deben existir.

Buenas prácticas

- Declarar como `const` los métodos que no modifican el estado observable del objeto para evitar cambios accidentales y permitir su uso con instancias constantes.

```
//-- Método que no modifica el estado pero no está declarado como const
class Usuario {
  std::string nombre;
public:
  std::string obtenerNombre() {
    return nombre;
  }
};
```

```
//-- Método declarado como const al no alterar el estado del objeto
class Usuario {
  std::string nombre;
public:
  std::string obtenerNombre() const {
    return nombre;
  }
};
```

- Marcar como `const` los métodos de acceso (getters) para permitir su invocación sobre objetos constantes y mejorar la seguridad del diseño.

```
//-- Getter no const que impide su uso con objetos constantes
class Config {
  int puerto;
public:
  int getPuerto() {
    return puerto;
  }
};
```

```
//-- Getter declarado como const para garantizar que no modifica el estado
```

```
class Config {
  int puerto;
public:
  int getPuerto() const {
    return puerto;
  }
};
```

- Declarar como `const` los métodos que realizan cálculos internos sin modificar atributos para reforzar la inmutabilidad lógica.

```
//-- Método de cálculo sin const que podría permitir modificaciones futuras no
deseadas
class Rectangulo {
  int ancho;
  int alto;
public:
  int area() {
    return ancho * alto;
  }
};
```

```
//-- Método de cálculo marcado como const al no alterar el estado
class Rectangulo {
  int ancho;
  int alto;
public:
  int area() const {
    return ancho * alto;
  }
};
```

- Aplicar `const` en sobrecargas de métodos para diferenciar comportamientos entre objetos constantes y no constantes.

```
//-- Sobrecarga sin versión const que limita el acceso seguro
class Buffer {
  std::string datos;
public:
  char& operator[](size_t i) {
    return datos[i];
  }
};
```

```
//-- Sobrecarga con versión const que protege objetos constantes
class Buffer {
  std::string datos;
```

```
public:
  char& operator[](size_t i) {
    return datos[i];
  }
  const char& operator[](size_t i) const {
    return datos[i];
  }
};
```

2.1.3.2 Estándares metodológicos

- **C++ Core-Con.2**: By default, make member functions const.

2.1.4 [cpp.const.4] - Apuntar a "const" a todo puntero y referencia por defecto

ID	cpp.const.4	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Usar sistemáticamente el calificador `const` en punteros y referencias refuerza la seguridad y reduce la posibilidad de errores. Aplicar `const` por defecto ayuda a minimizar modificaciones accidentales de datos, evitar comportamientos indefinidos y endurecer la semántica del código frente a ataques que exploten corrupciones de memoria. Al limitar explícitamente qué puede modificarse, se mejora la fiabilidad del software, se facilita el razonamiento seguro del flujo de datos y se refuerza la protección frente a vulnerabilidades relacionadas con integridad.

Codificación insegura

Se muestra una función que recibe un puntero y una referencia a datos que no deben modificarse. El objetivo es consultar e imprimir información. La intención es evidenciar cómo la ausencia de `const` permite modificaciones accidentales sobre datos que solo deberían leerse.

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

//-- Función que solo debería mostrar datos, pero puede modificarlos
accidentalmente
void mostrarDatos(std::string* nombre, int& edad) {
  //-- Modificación accidental del contenido apuntado
  *nombre = "Modificado";
  //-- Modificación accidental de la referencia
  edad = 0;
```

```
  //-- Impresión de datos
  std::cout << "Nombre: " << *nombre << std::endl;
  std::cout << "Edad: " << edad << std::endl;
}

int main() {
  //-- Datos originales
  std::string nombre = "Carlos";
  int edad = 30;

  //-- Llamada a la función
  mostrarDatos(&nombre, edad);

  //-- Verificación posterior a la llamada
  std::cout << "Despues de la llamada:" << std::endl;
  std::cout << "Nombre: " << nombre << std::endl;
  std::cout << "Edad: " << edad << std::endl;

  return 0;
}
```

Codificación recomendada

Se presenta la misma funcionalidad, pero aplicando `const` por defecto a punteros y referencias cuando los datos no deben modificarse. El objetivo es reforzar la inmutabilidad, evitar efectos secundarios y permitir que el compilador detecte intentos de modificación no autorizados.

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

//-- Función que solo muestra datos y garantiza que no los modificará
void mostrarDatos(const std::string* nombre, const int& edad) {
  //-- Solo lectura de los datos
  std::cout << "Nombre: " << *nombre << std::endl;
  std::cout << "Edad: " << edad << std::endl;
}

int main() {
  //-- Datos originales
  std::string nombre = "Carlos";
  int edad = 30;

  //-- Llamada a la función con garantías de solo lectura
  mostrarDatos(&nombre, edad);

  //-- Verificación posterior a la llamada
  std::cout << "Despues de la llamada:" << std::endl;
```

```
    std::cout << "Nombre: " << nombre << std::endl;
    std::cout << "Edad: " << edad << std::endl;

    return 0;
}
```

2.1.4.1 Descripción

Aplicar `const` a punteros y referencias por defecto establece un contrato explícito que define qué datos pueden modificarse y cuáles no. Esta estrategia contribuye a eliminar clases completas de errores, al mismo tiempo que fortalece la integridad del sistema ante posibles vulnerabilidades explotables derivadas de cambios inesperados del estado interno. El uso de `const` actúa como una barrera preventiva: bloquea operaciones de escritura no deseadas y obliga a los desarrolladores a analizar explícitamente cualquier posibilidad de mutación. Esta característica resulta especialmente relevante en entornos donde la **integridad** es un requisito estricto.

La ausencia de `const` puede introducir problemas críticos como modificaciones accidentales, condiciones de carrera, inconsistencias lógicas, corrupción de memoria o estados inválidos. En particular, en sistemas que manejan estructuras complejas o recursos compartidos, la mutabilidad no controlada incrementa exponencialmente el riesgo de errores de integridad. La presencia estricta de `const` reduce la superficie de ataque y ofrece protección estructural al garantizar que las funciones y objetos solo realicen acciones permitidas.

En interfaces o bibliotecas utilizadas por múltiples módulos, el uso de `const` garantiza que los consumidores no puedan alterar estados internos críticos. Esta técnica también es útil para auditorías de código, ya que facilita el análisis del flujo de datos, permite detectar rápidamente puntos de mutabilidad peligrosa y ayuda a construir invariantes más robustos.

En entornos concurrentes, el uso de `const` actúa como un filtro de seguridad, ya que previene mutaciones inesperadas que puedan generar condiciones de carrera. Si bien `const` no convierte automáticamente un objeto en thread-safe, sí introduce restricciones que obligan al desarrollador a utilizar mecanismos explícitos de sincronización para cualquier modificación, reduciendo así errores inadvertidos.

Un sistema más predecible es un sistema más seguro. La práctica de declarar por defecto punteros y referencias como `const` respalda la filosofía de construcción de software robusto basada en la minimización de privilegios: un componente solo debe poder modificar aquello que explícitamente está autorizado a cambiar. Este principio coincide con estrategias de endurecimiento típicas en sistemas críticos.

Riesgos asociados

- **Modificación accidental de datos compartidos**: sin `const` en punteros o

referencias, se permite la mutación inadvertida de objetos que deberían ser inmutables, lo que puede alterar estados críticos y romper invariantes de seguridad.

- **Violación de contratos implícitos de API**: al no declarar parámetros como `const`, se pierde la garantía formal de solo lectura, debilitando el diseño por contrato y facilitando efectos secundarios no documentados.
- **Escalada de privilegios lógica**: funciones que reciben punteros no `const` pueden modificar estructuras sensibles (por ejemplo, tablas de permisos o configuraciones), ampliando la superficie de ataque interna.
- **Corrupción de memoria por modificaciones indebidas**: la ausencia de `const` permite escribir sobre memoria que conceptualmente debería tratarse como solo lectura, aumentando el riesgo de estados inconsistentes o corrupción de heap/stack.
- **Dificultad para aplicar análisis estático efectivo**: herramientas de análisis estático y verificadores formales dependen de cualificadores como `const` para razonar sobre inmutabilidad y flujo de datos. Sin ellos, disminuye la capacidad de detectar vulnerabilidades.
- **Mayor probabilidad de condiciones de carrera**: en entornos concurrentes, la falta de `const` impide diferenciar claramente datos de solo lectura, favoreciendo accesos concurrentes de escritura no sincronizados.
- **Facilitación de vulnerabilidades por sobreescritura lógica**: funciones que deberían operar sobre copias o vistas inmutables pueden modificar estructuras internas, introduciendo fallos explotables (por ejemplo, alteración de buffers antes de validación).
- **Imposibilidad de ubicar datos en segmentos de solo lectura**: sin `const`, el compilador no puede optimizar ubicando ciertos objetos en secciones de memoria protegidas contra escritura, reduciendo defensas pasivas del binario.
- **Aumento del acoplamiento y menor encapsulamiento**: la mutabilidad implícita favorece diseños donde múltiples componentes pueden alterar el mismo estado, complicando el control de integridad y la auditoría de seguridad.
- **Mayor riesgo en integración con código legado o de terceros**: APIs que no marcan correctamente parámetros como `const` pueden permitir que módulos externos modifiquen datos internos, ampliando el vector de ataque en sistemas modulares.

Buenas prácticas

- Declarar punteros como `const` cuando no se requiere modificar el dato apuntado.

```
//-- Puntero permite modificar el contenido sin necesidad
int value = 10;
int* ptr = &value;
*ptr = 20;
```

```
//-- Puntero a dato constante evita modificaciones accidentales
int value2 = 10;
const int* ptr2 = &value2;
//*ptr2 = 20;  // modificación restringida
```

- Usar referencias constantes en parámetros de funciones para evitar copias y modificaciones.

```
//-- Parámetro por referencia permite modificar el objeto original
void printData(std::string& text) {
  text.clear();
}
```

```
//-- Referencia constante evita modificaciones y copias innecesarias
void printDataSafe(const std::string& text) {
  std::cout << text;
}
```

- Aplicar `const` a métodos que no modifican el estado del objeto.

```
//-- Método puede alterar el estado interno accidentalmente
class User {
  std::string name;
public:
  std::string getName() {
    return name;
  }
};
```

```
//-- Método marcado como constante garantiza inmutabilidad
class UserSafe {
  std::string name;
public:
  std::string getName() const {
    return name;
  }
};
```

- Declarar punteros constantes cuando la dirección no debe cambiar.

```
//-- Puntero puede ser reasignado accidentalmente
```

```
int a = 1;
int b = 2;
int* ptr = &a;
ptr = &b;
```

```
//-- Puntero constante mantiene fija la dirección asignada
int x = 1;
int y = 2;
int* const ptrFixed = &x;
//ptrFixed = &y;  // reasignación restringida
```

- Combinar constancia del dato y del puntero cuando corresponda.

```
//-- Puntero y dato pueden modificarse libremente
int data = 5;
int* ptr = &data;
*ptr = 6;
ptr = nullptr;
```

```
//-- Ni el dato ni la dirección pueden modificarse
int dataSafe = 5;
const int* const ptrSafe = &dataSafe;
//*ptrSafe = 6;      // modificación restringida
//ptrSafe = nullptr; // reasignación restringida
```

2.1.4.2 Estándares metodológicos

- **C++ Core-Con.3**: By default, pass pointers and references to consts.

2.1.5 [cpp.const.5] - Usar "constexpr", "consteval", "if consteval" para evaluaciones en tiempo de compilación

ID	cpp.const.5	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Usar mecanismos de evaluación en tiempo de compilación, como `constexpr`, `consteval` e `if consteval`, mejora la seguridad, robustez y predictibilidad del código. El objetivo es permitir que más errores lógicos y estructurales sean detectados por el compilador antes de la ejecución, reduciendo riesgos de comportamiento indefinido, corrupción de memoria o cálculos inseguros en tiempo de ejecución. Además, fomenta la generación de invariantes, la verificación de restricciones y la eliminación de rutas inseguras en función de valores conocidos en compilación.

Codificación insegura

Se muestra el cálculo de un tamaño de buffer y de un valor matemático que conceptualmente deberían conocerse en tiempo de compilación. El objetivo es observar cómo, al no usar mecanismos de evaluación en compilación, los valores se calculan en tiempo de ejecución y pueden depender de datos externos, perdiendo garantías y validaciones tempranas.

```
#include <iostream>
#include <vector>

//-- Función que podría evaluarse en tiempo de compilación pero no está marcada
como constexpr
int square(int x) {
  return x * x;
}

int main() {
  //-- Valor que se espera constante, pero se calcula en tiempo de ejecución
  int base = 5;
  int bufferSize = square(base);

  //-- El tamaño del vector depende de un cálculo en runtime
  std::vector<int> buffer(bufferSize);

  //-- Uso del buffer
  for (int i = 0; i < bufferSize; ++i) {
    buffer[i] = i;
  }

  std::cout << "Buffer size: " << buffer.size() << std::endl;

  return 0;
}
```

Codificación recomendada

Se implementa el mismo cálculo usando `constexpr`, `consteval` e `if consteval` para forzar y controlar la evaluación en tiempo de compilación. Se busca garantizar que ciertos valores críticos se determinen en compilación, reduciendo errores y mejorando la seguridad al evitar dependencias en tiempo de ejecución.

```
#include <iostream>
#include <array>

//-- Fuerza evaluación en tiempo de compilación
consteval int square(int x) {
  return x * x;
```

```
}

//-- Puede evaluarse en compilación o en ejecución según el contexto
constexpr int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}

//-- Cambia comportamiento según el momento de evaluación
constexpr int computeValue(int x) {
  if consteval {
    return x * 2;
  } else {
    return x + 2;
  }
}

int main() {
  //-- Evaluado obligatoriamente en compilación
  constexpr int bufferSize = square(5);

  //-- std::array requiere tamaño conocido en compilación
  std::array<int, bufferSize> buffer{};

  //-- factorial evaluado en compilación
  constexpr int fact5 = factorial(5);

  //-- computeValue evaluado en compilación
  constexpr int compiledValue = computeValue(10);

  std::cout << "Buffer size: " << buffer.size() << std::endl;
  std::cout << "Factorial(5): " << fact5 << std::endl;
  std::cout << "Computed value: " << compiledValue << std::endl;

  return 0;
}
```

2.1.5.1 Descripción

El uso adecuado de `constexpr`, `consteval` e `if consteval` se ha convertido en un pilar esencial en la codificación moderna de C++. Estos descriptores permiten trasladar validaciones críticas y cálculos sensibles desde el tiempo de ejecución hacia el tiempo de compilación, reduciendo la superficie de ataque y evitando condiciones de error que, de otro modo, podrían derivar en vulnerabilidades explotables. Su aplicación facilita auditorías estáticas, análisis de seguridad y garantías formales sobre el comportamiento del programa.

Los beneficios son múltiples:

- **Reduce el riesgo de comportamientos indefinidos**, puesto que cálculos potencialmente peligrosos se evalúan en un contexto controlado por el compilador.
- **Evita que datos calculados dinámicamente introduzcan inconsistencias lógicas**, especialmente en sistemas complejos.
- **Contribuye a la detección precoz de errores**, permitiendo a los equipos corregir fallos antes de que sean desplegados.

constexpr

El descriptor `constexpr` permite indicar que una función o variable puede ser evaluada en tiempo de compilación si se cumplen las condiciones necesarias. Su importancia radica en que obliga a definir funciones sin efectos secundarios, puras y deterministas, lo cual contribuye al desarrollo de código más seguro. De esta forma, **las invariantes de seguridad pueden expresarse como evaluaciones en tiempo de compilación**, previniendo la manipulación de valores críticos o configuraciones peligrosas.

Por ejemplo, la generación de claves constantes internas, máscaras de bits, validaciones de rangos predefinidos y cálculos estructurales pueden realizarse con `constexpr` para impedir su modificación accidental o maliciosa en tiempo de ejecución.

```
constexpr int max_allowed = 1024;
constexpr int compute_limit(int x) { return x * 2; }
static_assert(compute_limit(100) == 200);
```

consteval

El identificador `consteval` obliga a que una función sea evaluada exclusivamente en tiempo de compilación. Esto añade una capa adicional de protección, ya que **impide que valores críticos sean generados dinámicamente**, garantizando que toda la lógica depende de información conocida y fija en compilación. Esta propiedad resulta esencial cuando se requieren decisiones de alta seguridad, como cálculos de tamaños de buffers, diseños de estructuras internas, determinación de parámetros sensibles o validación de configuraciones que no deben depender del entorno de ejecución.

Su uso también elimina posibles vulnerabilidades relacionadas con entradas manipuladas por un atacante, dado que los parámetros deben ser constantes en compilación, lo cual imposibilita la inyección o alteración en tiempo de ejecución.

```
consteval int secure_mul(int x) { return x * 4; }
constexpr int v = secure_mul(5);
```

if consteval

La construcción `if consteval` habilita una técnica poderosa: elegir rutas distintas según si el código se evalúa en compilación o en ejecución. Desde una perspectiva de seguridad, esto permite **aislar operaciones críticas y sensibles en contextos controlados y verificados**, mientras se reservan rutas alternativas para escenarios dinámicos adecuados.

Este mecanismo facilita estrategias para mitigar comportamientos indefinidos, generar mensajes de error específicos y adaptar funciones para que sean seguras tanto en ejecución como en compilación.

```
template<typename T>
auto process(T v) {
  if consteval {
    return v + 1;
  } else {
    return v * 2;
  }
}
```

Riesgos asociados

- **Pérdida de garantías de invariantes en tiempo de compilación**: No usar `constexpr` o `consteval` permite que validaciones críticas (rangos, tamaños, precondiciones estructurales) se difieran a tiempo de ejecución, donde pueden ser explotadas antes de detectarse.
- **Mayor superficie de fallo en runtime**: Cálculos que podrían resolverse en compilación (por ejemplo, tamaños de buffers, máscaras de bits, offsets) quedan expuestos a errores dinámicos, aumentando el riesgo de desbordamientos o accesos fuera de límites.
- **Ausencia de fallos tempranos (fail-fast)**: `consteval` fuerza evaluación en compilación; sin él, errores semánticos pasan a producción en lugar de bloquear la compilación.
- **Inconsistencia entre configuraciones de compilación**: Sin evaluación constante garantizada, distintas rutas de ejecución pueden depender de valores calculados dinámicamente, generando divergencias entre builds o entornos.
- **Validaciones incompletas de parámetros template**: No emplear `constexpr` en funciones auxiliares de meta-programación puede permitir instanciaciones inválidas que deberían descartarse en tiempo de compilación.
- **Pérdida de optimización determinista**: Valores no evaluados en compilación pueden impedir al compilador ubicar datos en secciones inmutables o eliminar ramas muertas, afectando previsibilidad y reduciendo endurecimiento pasivo del

binario.

- **Mayor riesgo en lógica de seguridad condicional**: Sin `if consteval`, puede mezclarse lógica destinada exclusivamente a compilación con rutas de ejecución runtime, introduciendo caminos no auditados o inconsistentes.
- **Dependencia excesiva de validaciones dinámicas**: Al no mover comprobaciones al dominio de compilación, se incrementa la carga de validaciones en runtime, ampliando el espacio donde entradas maliciosas pueden interactuar con el sistema.
- **Errores silenciosos en constantes derivadas**: Cálculos como hashes, identificadores, tamaños estructurales o claves derivadas pueden variar si no se fuerzan como expresiones constantes, afectando integridad o compatibilidad binaria.
- **Mayor complejidad en auditorías de seguridad**: El uso sistemático de `constexpr` y `consteval` delimita claramente qué se resuelve en compilación; sin ello, resulta más difícil razonar formalmente sobre qué valores son verdaderamente constantes y cuáles dependen de estado dinámico.

Buenas prácticas

- Utilizar `constexpr` para funciones y variables que puedan evaluarse en tiempo de compilación, reduciendo errores y mejorando el rendimiento.

```
//-- Cálculo en tiempo de ejecución sin restricción de compilación
int square(int x) {
  return x * x;
}
const int value = square(5);
```

```
//-- Cálculo garantizado en tiempo de compilación cuando es posible
constexpr int square(int x) {
  return x * x;
}
constexpr int value = square(5);
```

- Emplear `consteval` cuando una función deba evaluarse obligatoriamente en tiempo de compilación para evitar usos indebidos en ejecución.

```
//-- Función invocable en tiempo de ejecución
constexpr int getBufferSize(int base) {
  return base * 2;
}
int size = getBufferSize(userInput);
```

```
//-- Función obligatoriamente evaluada en compilación
consteval int getBufferSize(int base) {
  return base * 2;
}
constexpr int size = getBufferSize(64);
```

- Usar `if consteval` para diferenciar lógica de compilación y ejecución, evitando comportamientos inesperados o inseguros.

```
//-- Lógica sin distinción entre compilación y ejecución
constexpr int compute(int x) {
  return x > 0 ? x : throw "error";
}
```

```
//-- Separación explícita según el contexto de evaluación
constexpr int compute(int x) {
  if consteval {
    return x > 0 ? x : 0;
  } else {
    if (x <= 0) throw "error";
    return x;
  }
}
```

- Declarar objetos como `constexpr` cuando representen valores inmutables conocidos en compilación para reforzar la integridad y evitar modificaciones accidentales.

```
//-- Variable constante evaluada en ejecución
const int maxConnections = getConfigValue();
```

```
//-- Valor fijo determinado en compilación
constexpr int maxConnections = 100;
```

2.1.5.2 Estándares metodológicos

- **C++ Core-F.4**: If a function might have to be evaluated at compile time, declare it constexpr.

2.2 Cadenas de caracteres

2.2.1 [cpp.str.1] - Usar "std::string" para almacenar cadenas

ID	cpp.str.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Para una gestión segura y robusta del texto debe utilizarse `std::string` en lugar de punteros crudos o buffers manuales. El uso de esta clase estándar proporciona controles automáticos de memoria, evita accesos fuera de límites y reduce riesgos asociados con fugas, corrupción de memoria y uso de memoria no inicializada. Además, facilita la aplicación coherente de buenas prácticas modernas de C++ y contribuye a mejorar la seguridad, legibilidad y mantenibilidad del código.

Codificación insegura

Gestión de cadenas usando memoria dinámica manual con arrays de caracteres. Se busca mostrar cómo la copia y manipulación directa de buffers puede provocar desbordamientos y errores de memoria cuando no se controlan correctamente los tamaños.

```
#include <iostream>
#include <cstring>

int main() {
  //-- Reserva manual de memoria para una cadena pequeña
  char* nombre = new char[8];

  //-- Copia sin validar el tamaño real del contenido
  std::strcpy(nombre, "Alejandro");

  //-- Muestra el contenido almacenado
  std::cout << "Nombre: " << nombre << std::endl;

  //-- Liberación manual de memoria
  delete[] nombre;

  return 0;
}
```

Codificación recomendada

Gestión de cadenas utilizando la clase estándar para que la memoria sea administrada automáticamente. Se pretende aprender cómo evitar desbordamientos y errores de liberación usando tipos seguros del estándar.

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

int main() {
  //-- Declaración de cadena con gestión automática de memoria
  std::string nombre;

  //-- Asignación segura sin riesgo de desbordamiento
  nombre = "Alejandro";

  //-- Muestra el contenido almacenado
  std::cout << "Nombre: " << nombre << std::endl;

  //-- No es necesario liberar memoria manualmente
  return 0;
}
```

2.2.1.1 Descripción

El uso de `std::string` permite que se gestione automáticamente la memoria necesaria para almacenar cadenas, reduciendo los riesgos de errores relacionados con accesos fuera de rango y evita fugas de memoria que suelen aparecer cuando se manipulan cadenas mediante punteros crudos o funciones heredadas de C. La gestión automática, junto con el control de excepciones y los mecanismos RAII (Resource Acquisition Is Initialization), convierte a `std::string` en un elemento esencial para la seguridad del software en entornos modernos.

El empleo de punteros crudos y funciones como `strcpy`, `strcat` o `strlen` introduce una serie de riesgos bien conocidos: desbordamientos de buffer, escritura fuera de los límites, uso indebido de memoria no inicializada y errores difíciles de depurar vinculados a la gestión manual de memoria dinámica. **Estos problemas representan una superficie de ataque significativa y son una fuente habitual de vulnerabilidades explotables**. En contraste, `std::string` asigna memoria de forma controlada, redimensiona automáticamente el buffer cuando es necesario y valida las operaciones con mayor seguridad que las funciones de bajo nivel.

`std::string`, al integrarse de forma natural con otras estructuras estándar del lenguaje, permite aplicar correctamente prácticas modernas como el paso por valor eficiente mediante `move semantics`, evitando copias innecesarias. La seguridad del programa se ve fortalecida porque las operaciones sobre cadenas quedan encapsuladas en una abstracción segura, limitando la exposición directa a manipulación de memoria sin control adecuado.

Riesgos asociados

- **Desbordamientos de buffer (buffer overflow)**: el uso de arrays `char[]` o

punteros `char*` sin control de tamaño facilita escrituras fuera de límites, especialmente al emplear funciones como `strcpy`, `sprintf` o `gets`.

- **Falta de gestión automática de memoria**: las cadenas C requieren asignación y liberación manual (`malloc/free`, `new/delete`), aumentando el riesgo de fugas, dobles liberaciones y uso después de liberar (use-after-free).
- **Errores de terminación nula**: olvidar el carácter `'\0'` puede provocar lecturas fuera de límites, filtración de información o comportamiento indefinido.
- **Vulnerabilidades de desbordamiento por concatenación insegura**: operaciones como `strcat` no validan tamaño destino, lo que puede ser explotado mediante entradas maliciosas.
- **Ausencia de control de longitud explícito**: en C-style strings la longitud depende del terminador nulo, lo que impide validaciones robustas basadas en tamaño real del buffer.
- **Mayor superficie para ataques de formato**: el uso combinado de cadenas C y funciones como `printf` puede facilitar vulnerabilidades de format string si no se sanitizan adecuadamente.
- **Dificultad en manejo seguro de excepciones**: las cadenas manuales no integran semántica RAII, lo que complica garantizar liberación segura ante fallos.
- **Problemas de interoperabilidad y copia superficial**: pasar punteros crudos puede derivar en aliasing inseguro, modificaciones inesperadas o corrupción de datos compartidos.
- **Errores en redimensionamiento dinámico**: reasignaciones manuales incorrectas pueden provocar pérdida de datos o punteros colgantes.
- **Mayor complejidad en auditoría y análisis estático**: el uso de punteros y buffers manuales dificulta la verificación formal y la detección automática de vulnerabilidades frente a `std::string`, que encapsula tamaño, capacidad y gestión de memoria.

Buenas prácticas

- Preferir `std::string` frente a buffers `char[]` para almacenamiento dinámico.

```
//-- No recomendado
void setName(const char* input) {
  char name[16];
  std::strcpy(name, input);
}
```

```
//-- Se elimina el riesgo de desbordamiento y gestión manual de memoria.
void setName(const std::string& input) {
```

```
  std::string name = input;
}
```

- Usar `std::string::size()` en lugar de `strlen`.

```
//-- No recomendado
std::size_t length(const char* s) {
  return std::strlen(s);
}
```

```
//-- La longitud es parte del objeto y no depende de un terminador nulo.
std::size_t length(const std::string& s) {
  return s.size();
}
```

- Evitar concatenación insegura; usar operadores o métodos propios.

```
//-- No recomendado
void build(char* dest, const char* a, const char* b) {
  std::strcpy(dest, a);
  std::strcat(dest, b);
}
```

```
//-- Se elimina el riesgo de escribir fuera del buffer.
std::string build(const std::string& a, const std::string& b) {
  return a + b;
}
```

- Pasar cadenas como `const std::string&`.

```
//-- No recomendado
void log(std::string s) {
}
```

```
//-- Se evita una copia innecesaria y se garantiza que la función no modifica la
cadena.
void log(const std::string& s) {
}
```

- Usar std::string_view cuando no se requiere propiedad.

```
//-- No recomendado
void process(const std::string& s) {
}
```

```
//-- Reduce asignaciones y mantiene semántica segura sin copiar datos.
void process(std::string_view s) {
}
```

- Validar tamaño para evitar un potencial uso abusivo de los recursos.

```
//-- Se reserva memoria basándose en una entrada no validada (sin límite)
void validate(const std::string& input) {
  std::vector<char> buffer(input.size());
  std::copy(input.begin(), input.end(), buffer.begin());
}
```

```
//-- Se establecen límites explícitos contra abusos de recursos
void validate(const std::string& input) {
  constexpr std::size_t MAX_INPUT_SIZE = 256;
  if (input.size() > MAX_INPUT_SIZE) {
    throw std::length_error("Input too large");
  }
  std::vector<char> buffer(input.size());
  std::copy(input.begin(), input.end(), buffer.begin());
}
```

- Evitar exponer punteros crudos mediante `c_str()` salvo interoperabilidad.

```
//-- Riesgoso
const char* expose(const std::string& s) {
  return s.c_str();
}
```

```
//-- Mejor diseño: evita que el consumidor gestione punteros dependientes del ciclo
de vida interno
const std::string& expose(const std::string& s) {
  return s;
}
```

- Reservar capacidad cuando se conoce el tamaño esperado.

```
//-- No se reserva capacidad aunque se conoce el tamaño aproximado
std::string build() {
  std::string result;
  result += "header:";
  result += "data";
  return result;
}
```

```
//-- Minimiza realocaciones y mejora la previsibilidad.
```

```
std::string build() {
  std::string result;
  result.reserve(128);
  result += "header:";
  result += "data";
  return result;
}
```

- Usar funciones miembro seguras en lugar de APIs C.

```
//-- No recomendado
void format(char* buffer, int value) {
  std::sprintf(buffer, "Value: %d", value);
}
```

```
//-- Se elimina el riesgo de sobrescritura de memoria
std::string format(int value) {
  return "Value: " + std::to_string(value);
}
```

2.2.1.2 Estándares metodológicos

- **C++ Core-SL.str.1**: Use std::string to own character sequences.

2.2.2 [cpp.str.2] - Evitar crear "std::string" a partir de un puntero nulo

ID	**cpp.str.2**	**Uso**	**Obligatorio**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Nunca debe intentarse construir un objeto `std::string` a partir de un puntero nulo, dado que este patrón provoca comportamientos indefinidos y puede derivar en fallos críticos, interrupciones del programa o vulnerabilidades explotables. Esta recomendación promueve validar siempre los punteros antes de su uso, así como emplear alternativas seguras que garanticen la correcta inicialización de la cadena.

Codificación insegura

Construcción de un objeto `std::string` a partir de un puntero que puede ser nulo, simulando la recepción de una cadena desde una API externa. Se busca mostrar el impacto en tiempo de ejecución cuando no se valida el puntero antes de usarlo.

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

```
const char* obtenerNombreDesdeAPI(bool devolverNulo) {
  //-- Simula una API externa que puede devolver un puntero nulo
  if (devolverNulo) {
    return nullptr;
  }
  return "Carlos";
}

int main() {
  //-- Se solicita un valor que podría ser nulo
  const char* nombrePtr = obtenerNombreDesdeAPI(true);

  //-- Construcción directa sin validar el puntero
  std::string nombre(nombrePtr);

  //-- Uso del string (comportamiento indefinido si el puntero era nulo)
  std::cout << "Nombre: " << nombre << std::endl;

  return 0;
}
```

Codificación recomendada

Validación previa de un puntero antes de construir un objeto `std::string`, asegurando que el valor no sea nulo y aplicando una alternativa segura en caso contrario. Se refuerza la prevención de comportamiento indefinido y fallos en ejecución.

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

const char* obtenerNombreDesdeAPI(bool devolverNulo) {
  //-- Simula una API externa que puede devolver un puntero nulo
  if (devolverNulo) {
    return nullptr;
  }
  return "Carlos";
}

int main() {
  //-- Se solicita un valor que podría ser nulo
  const char* nombrePtr = obtenerNombreDesdeAPI(true);

  //-- Validación explícita antes de construir std::string
  std::string nombre;
  if (nombrePtr != nullptr) {
    nombre = std::string(nombrePtr);
  } else {
    //-- Valor por defecto seguro en caso de puntero nulo
```

```
    nombre = "Desconocido";
  }

  //-- Uso seguro del objeto std::string
  std::cout << "Nombre: " << nombre << std::endl;

  return 0;
}
```

2.2.2.1 Descripción

La construcción de objetos `std::string` a partir de punteros nulos constituye una fuente clásica de comportamiento indefinido en C++. Este problema aparece frecuentemente al manipular funciones heredadas de C que retornan punteros de tipo `char*`, especialmente cuando dichas funciones pueden devolver `nullptr` para indicar errores o ausencia de datos. La biblioteca estándar no define el comportamiento de `std::string` cuando recibe un puntero nulo; por ello, el resultado suele ser un fallo en tiempo de ejecución que compromete la estabilidad del proceso.

El uso inadvertido de un puntero nulo al inicializar un `std::string` puede generar múltiples impactos en la aplicación. Entre los más relevantes se encuentran la terminación inesperada del proceso, la corrupción de memoria, el acceso inválido a zonas reservadas y la apertura de vectores de ataque que pueden ser aprovechados para explotar vulnerabilidades de denegación de servicio. Debido a que el comportamiento es indefinido, la manifestación del fallo puede variar según compilador, optimizaciones o plataforma, lo cual incrementa la dificultad del diagnóstico y reproduce escenarios peligrosamente inconsistentes. El requisito destaca la importancia de evitar construcciones implícitas no controladas y reforzar la revisión sistemática de todos los puntos donde cadenas y punteros interactúan.

Para aumentar la consistencia, puede definirse una envoltura que valide siempre los punteros. De este modo se reduce la duplicación de lógica y se incrementa la fiabilidad del sistema. Las funciones auxiliares facilitan asegurar que todas las rutas de ejecución aplican correctamente la verificación previa.

```
std::string safeString(const char *ptr) {
  if (ptr == nullptr) {
    return std::string();
  }
  return std::string(ptr);
}
```

Los sistemas que integran código moderno C++ con bibliotecas tradicionales en C deben aplicar controles estrictos, ya que estas bibliotecas suelen indicar errores devolviendo punteros nulos. En tales casos es imprescindible implementar comprobaciones

sistemáticas justo después de cada llamada, manteniendo así un límite claro entre el mundo inseguro heredado y el entorno robusto del estándar C++. El riesgo aumenta cuando la interfaz forma parte de una capa crítica del sistema, como la comunicación con dispositivos, procesamiento de datos masivos o accesos a ficheros. El requisito enfatiza la importancia de revisar con rigor los flujos de datos mixtos para impedir el uso accidental de punteros nulos en contextos donde no existe tolerancia a fallos.

C++ moderno proporciona alternativas más seguras para gestionar cadenas. Emplear `std::optional` o valores por defecto reduce la probabilidad de pasar inadvertidamente un puntero nulo. Paradigmas como RAII, asignaciones explícitas y validación invariantes también contribuyen a reducir las ocasiones donde podrían surgir estados no válidos. El requisito fomenta el uso de estas capacidades para asegurar un flujo de datos estable. De igual modo, el empleo de funciones como `std::string::assign` debe cuidarse, pues su comportamiento frente a punteros nulos mantiene las mismas restricciones y riesgos que los constructores directos.

Riesgos asociados

- **Comportamiento indefinido (Undefined Behavior)**: Construir `std::string` desde un puntero nulo (`std::string s(nullptr);`) viola las precondiciones del constructor y activa UB, lo que puede derivar en fallos impredecibles.
- **Denegación de servicio (crash inmediato)**: Muchas implementaciones intentan calcular la longitud con `traits::length`, lo que provoca desreferenciación de `nullptr` y abortos en ejecución.
- **Exposición de vulnerabilidades explotables**: En entornos sin protecciones modernas, el acceso inválido puede abrir la puerta a explotación mediante manipulación del flujo de ejecución.
- **Corrupción de memoria**: Dependiendo de la implementación y optimizaciones, el uso de un puntero nulo puede desencadenar lecturas fuera de límites o estados internos inconsistentes.
- **Propagación silenciosa de errores lógicos**: Si el puntero proviene de una API C y no se valida, el fallo puede aparecer lejos del origen, dificultando el diagnóstico.
- **Falta de validación de entrada externa**: APIs que reciben `const char*` desde fuentes no confiables (red, FFI, bibliotecas legacy) pueden inyectar `nullptr`, generando fallos inesperados.
- **Ruptura de contratos de API**: Las funciones que aceptan `const char*` pero no documentan la prohibición de `nullptr` generan ambigüedad y comportamiento no definido.
- **Incompatibilidad entre compiladores/implementaciones**: Diferentes bibliotecas estándar pueden reaccionar de forma distinta ante `nullptr`, produciendo comportamientos inconsistentes entre plataformas.

- **Dificultad en análisis estático y auditoría**: La ausencia de validaciones explícitas antes de construir el `std::string` impide a herramientas de análisis inferir seguridad en el flujo de datos.
- **Escalada de errores en sistemas críticos**: En código de infraestructura (parsers, autenticación, configuración), un `nullptr` no controlado puede interrumpir servicios completos o invalidar procesos de validación.

Buenas prácticas

- Validar explícitamente el puntero antes de construir el `std::string`.

```
//-- No recomendado
void process(const char* input) {
  std::string s(input);
}
```

```
//-- Se garantiza el cumplimiento de la precondición del constructor.
void process(const char* input) {
  if (input == nullptr) {
    throw std::invalid_argument("input is null");
  }
  std::string s(input);
}
```

- Preferir `std::string` en interfaces públicas en lugar de `const char*`.

```
//-- No recomendado
void setName(const char* name);
```

```
//-- Recomendado: se elimina la posibilidad de recibir nullptr desde el contrato de
la API.
void setName(const std::string& name);
```

- Usar `std::optional<std::string>` cuando la ausencia sea válida.

```
//-- No recomendado
std::string getValue(const char* input) {
  return std::string(input);
}
```

```
//-- Se modela explícitamente la posibilidad de ausencia
std::optional<std::string> getValue(const char* input) {
  if (input == nullptr) {
    return std::nullopt;
  }
  return std::string(input);
```

```
}
```

- Usar operador ternario para fallback seguro cuando el diseño lo permita.

```
//-- No recomendado
std::string safeString(const char* input) {
  return input ? std::string(input) : std::string{};
}
```

```
//-- Centraliza la validación y reduce errores repetitivos
std::string toStringSafe(const char* input) {
  if (!input) {
    return {};
  }
  return std::string(input);
}
```

- Usar `std::string_view` solo si el puntero está validado previamente.

```
//-- No recomendado
void process(const char* input) {
  std::string_view view(input);
}
```

```
//-- std::string_view tampoco tolera nullptr si se asume terminación nula
void process(const char* input) {
  if (!input) {
    throw std::invalid_argument("null pointer");
  }
  std::string_view view(input);
}
```

- Definir precondiciones claras mediante contratos o aserciones: útil en código interno donde la nulidad es un error de programación.

```
//-- No se verifica la precondición antes de utilizar el puntero
void process(const char* input) {
  std::string s(input);
}
```

```
void process(const char* input) {
  assert(input != nullptr);
  std::string s(input);
}
```

- Evitar propagación directa de punteros provenientes de APIs C sin saneamiento.

```
//-- Se utiliza directamente el puntero devuelto por una API C
const char* raw = legacy_api();
std::string s(raw)
```

```
const char* raw = legacy_api();

//-- Se controla explícitamente la frontera entre código moderno y legacy
if (raw == nullptr) {
  // manejo explícito
} else {
  std::string s(raw);
}
```

2.2.2.2 Estándares metodológicos

- **CERT/CPP-STR51**: Do not attempt to create a std::string from a null pointer.

2.2.3 [cpp.str.3] - Comprobar los límites en el acceso a cadenas

ID	cpp.str.3	Uso	Obligatorio
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Establece la necesidad de aplicar verificaciones estrictas de límites al manipular y acceder a cadenas, con el fin de reducir el riesgo de accesos fuera de rango, lecturas indebidas o escrituras no autorizadas en memoria. Su aplicación adecuada contribuye a mantener la robustez del software y a prevenir vulnerabilidades asociadas a corrupción de memoria.

Codificación insegura

Acceso a posiciones de una cadena usando un índice proporcionado por el usuario sin validar si está dentro de los límites. Se busca mostrar cómo un acceso directo puede provocar comportamiento indefinido cuando el índice es inválido y qué riesgo implica no comprobar los límites.

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

int main() {
  //-- Cadena de ejemplo
  std::string texto = "Seguridad";

  //-- Índice introducido por el usuario sin validación
```

```
  int indice;
  std::cout << "Introduce una posicion: ";
  std::cin >> indice;

  //-- Acceso directo sin comprobar limites
  char caracter = texto[indice];

  //-- Puede provocar comportamiento indefinido si el indice es invalido
  std::cout << "Caracter en la posicion indicada: " << caracter << std::endl;

  return 0;
}
```

Codificación recomendada

Acceso seguro a posiciones de una cadena validando previamente que el índice esté dentro de los límites permitidos. Se pretende demostrar cómo evitar accesos fuera de rango y cómo manejar adecuadamente los errores de entrada.

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

int main() {
  //-- Cadena de ejemplo
  std::string texto = "Seguridad";

  //-- Índice introducido por el usuario
  int indice;
  std::cout << "Introduce una posicion: ";
  std::cin >> indice;

  //-- Validacion explicita del rango antes de acceder
  if (indice >= 0 && indice < static_cast<int>(texto.size())) {
    //-- Acceso seguro usando at que tambien valida limites
    char caracter = texto.at(indice);
    std::cout << "Caracter en la posicion indicada: " << caracter << std::endl;
  } else {
    //-- Manejo controlado de error por indice fuera de rango
    std::cerr << "Error: indice fuera de rango" << std::endl;
    return 1;
  }

  return 0;
}
```

2.2.3.1 Descripción

El acceso a cadenas representa una de las superficies de ataque más comunes en aplicaciones nativas. Una gestión inadecuada de límites puede conllevar accesos fuera de rango, corrupción de memoria, ejecución arbitraria de código y fugas de información sensible. Dado que el lenguaje proporciona distintos tipos de cadenas, desde arrays `char` hasta `std::string` y variantes modernas como `std::string_view`, es imprescindible comprender y aplicar prácticas que garanticen un acceso seguro y controlado. Este requisito enfatiza la necesidad de validar índices, verificar tamaños, utilizar funciones seguras y evitar patrones de riesgo que expongan al sistema a vulnerabilidades críticas.

El uso de arrays de caracteres tradicionales implica riesgos intrínsecos por su falta de verificación automática de límites. Cuando se utilizan funciones heredadas de C, como `strcpy`, `strcat` o `sprintf`, se generan escenarios propensos a desbordamientos de buffer. La recomendación principal es evitar estas funciones y reemplazarlas por alternativas más seguras, como `strncpy` o funciones basadas en C++ que gestionan memoria de forma automática. En todos los casos, es esencial confirmar explícitamente que el tamaño del destino es suficiente para contener la información sin producir truncamientos inesperados ni comportamientos indefinidos.

En el contexto de `std::string`, aunque este tipo de dato ofrece ciertas garantías al proporcionar gestión automática de memoria, ello no implica que el acceso por índice sea seguro. Un índice no verificado puede llevar a lecturas indebidas que comprometen la integridad del programa. Las funciones como `std::string::at` proporcionan un mecanismo más seguro al incluir verificaciones de límites que lanzan excepciones en caso de valores fuera de rango. El uso de `at` resulta especialmente aconsejable en sistemas críticos o cuando los datos provienen de fuentes externas que no se pueden considerar confiables.

Adicionalmente, las operaciones de concatenación pueden resultar peligrosas cuando se realizan sin medir previamente el tamaño resultante. Aunque `std::string` gestiona memoria de manera dinámica, la concatenación repetida dentro de bucles o la combinación de cadenas muy extensas puede producir un crecimiento abrupto, afectando tanto al rendimiento como a la disponibilidad. **El control explícito de la capacidad mediante `reserve` constituye una buena práctica** para evitar realocaciones innecesarias y mejorar la estabilidad de la aplicación.

El estándar moderno de C++ ha introducido `std::string_view`, un tipo que permite representar cadenas sin copiarlas. No obstante, su uso incorrecto presenta riesgos significativos, en especial cuando la vista hace referencia a datos que resultan inválidos o que expiran prematuramente. Es fundamental garantizar que el origen del `string_view` mantenga un ciclo de vida mayor al de cualquier referencia que se derive de él. Asimismo, se debe evitar modificar cadenas subyacentes cuando existan vistas activas, puesto que ello puede provocar invalidaciones o accesos ilegítimos.

El acceso basado en punteros también exige extremo cuidado. Los punteros mal calculados o los incrementos arbitrarios pueden llevar fácilmente a áreas de memoria no previstas. Para este tipo de manipulación, se recomienda preferir iteradores sobre punteros directos. Los iteradores de C++ contienen mecanismos más seguros para evitar errores comunes, aunque no eliminan por completo el riesgo. **Es esencial validar siempre que los iteradores se encuentren dentro de los rangos permitidos** y que no se utilicen tras operaciones que los invaliden, como inserciones o eliminaciones en contenedores.

En operaciones que implican acceso calculado, como el uso de funciones matemáticas para derivar posiciones dentro de cadenas, se debe realizar una validación exhaustiva. Esto incluye comprobar expresiones aritméticas, evitar desbordamientos de enteros y asegurarse de que cualquier índice derivado respete los límites del contenedor. **Las vulnerabilidades por integer overflow son especialmente críticas** cuando se emplean valores suministrados por el usuario o datos externos sin sanitización previa.

Los bucles que recorren cadenas deben implementarse evitando condiciones dependientes de valores no validados o de punteros que se desplacen sin supervisión. Un error común consiste en suponer que una cadena está correctamente terminada en nulo; sin embargo, la ausencia del terminador puede generar desbordamientos durante operaciones iterativas. El uso de funciones como `std::strlen` también debe ir acompañado de precauciones, ya que su ejecución implica recorrer la memoria hasta encontrar un `\0`, lo cual puede causar accesos ilegítimos si el buffer no está adecuadamente delimitado.

En el contexto de la interoperabilidad con APIs externas, bibliotecas de terceros o comunicación entre módulos, es imprescindible normalizar y validar la información antes de integrarla con estructuras nativas de C++. Los datos provenientes de redes, archivos, sistemas remotos o entradas de usuario deben tratarse como inherentemente no confiables. La aplicación de **validaciones estrictas de tamaño, contenido y formato** constituye un elemento esencial para impedir la explotación de vulnerabilidades mediante cadenas especialmente diseñadas para provocar fallos en el sistema.

Riesgos asociados

- **Desbordamiento de buffer**: acceder fuera de los límites válidos puede sobrescribir memoria adyacente, comprometiendo la integridad del proceso.
- **Lectura fuera de límites (out-of-bounds read)**: permite exfiltración de información sensible almacenada en memoria contigua.
- **Comportamiento indefinido**: el acceso inválido a memoria puede producir resultados impredecibles, fallos intermitentes o corrupción silenciosa de datos.
- **Denegación de servicio**: accesos inválidos pueden provocar abortos, excepciones no controladas o caídas del proceso.

- **Vulnerabilidades explotables**: un atacante puede manipular índices o longitudes para redirigir flujo de ejecución o alterar estructuras críticas.
- **Corrupción de heap o stack**: escribir fuera de rango puede modificar metadatos del heap o variables locales sensibles.
- **Errores lógicos encadenados**: una lectura incorrecta puede alimentar validaciones posteriores con datos inválidos, amplificando el impacto.
- **Fallos en validación de entrada**: no verificar tamaños antes de acceder facilita explotación mediante entradas especialmente construidas.
- **Inconsistencias en entornos concurrentes**: accesos fuera de límites combinados con condiciones de carrera pueden producir estados no deterministas difíciles de reproducir.
- **Dificultad de auditoría y análisis estático**: la ausencia de comprobaciones explícitas de límites complica la verificación formal y la detección temprana de defectos.

Buenas prácticas

- Validar explícitamente índices antes de acceder: comprobar que el índice esté dentro del rango válido.

```
void access(const std::string& s, std::size_t i) {
  //-- No se valida el índice
  char c1 = s[i];
```

```
//-- Validación explícita de límites
  if (i < s.size()) {
    char c2 = s[i];
    std::cout << c2 << "\n";
  }
}
```

- Preferir `at()` cuando se requiera verificación automática: `at()` lanza excepción si el índice es inválido.

```
void access(const std::string& s, std::size_t i) {

  //-- Acceso sin comprobación
  char c1 = s[i];
```

```
//-- Acceso con comprobación interna
  try {
    char c2 = s.at(i);
    std::cout << c2 << "\n";
```

```
  } catch (const std::out_of_range&) {
    std::cerr << "índice fuera de rango\n";
  }
}
```

- Validar tamaño antes de copiar o extraer subcadenas: evitar asumir longitudes mínimas.

```
std::string extractPrefix(const std::string& s) {

  //-- Asume tamaño mínimo
  std::string p1 = s.substr(0, 10);
```

```
//-- Verifica longitud antes de extraer
  if (s.size() >= 10) {
    return s.substr(0, 10);
  }
  return {};
}
```

- Comprobar límites en bucles manuales: evitar iteraciones que excedan el tamaño real.

```
void iterate(const std::string& s) {

  //-- Límite fijo potencialmente mayor
  for (std::size_t i = 0; i < 100; ++i) {
    char c = s[i];
  }
```

```
//-- Límite basado en tamaño real
  for (std::size_t i = 0; i < s.size(); ++i) {
    char c = s[i];
  }
}
```

- Validar entrada externa antes de indexar: nunca confiar en valores proporcionados por el usuario.

```
char getChar(const std::string& s, int indexFromUser) {

  //-- Uso directo de entrada externa
  char c1 = s[indexFromUser];
```

```
//-- Validación estricta
```

```
  if (indexFromUser < 0 ||
      static_cast<std::size_t>(indexFromUser) >= s.size()) {
    throw std::out_of_range("índice inválido");
  }

  return s[indexFromUser];
}
```

- Evitar aritmética manual con punteros internos: usar APIs seguras en lugar de manipular data() directamente.

```
void pointerAccess(const std::string& s) {

  //-- Aritmética sin validación
  const char* p = s.data();
  char c1 = *(p + 50);
```

```
//-- Control explícito del desplazamiento
  std::size_t offset = 50;
  if (offset < s.size()) {
    char c2 = s[offset];
  }
}
```

2.2.3.2 Estándares metodológicos

- **CERT/CPP-STR53**: Range check element access.

2.2.4 [cpp.str.4] - Usar "zstring" o tipos equivalentes para cadenas estilo C

ID	cpp.str.4	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se recomienda el uso de `zstring` u otros tipos equivalentes seguros para gestionar cadenas de estilo C en C++. El objetivo es reducir riesgos comunes asociados al manejo manual de buffers, terminadores nulos y desbordamientos que suelen aparecer cuando se usan punteros `char*` tradicionales. Con este enfoque se mejora la seguridad, la claridad y la corrección del código, evitando vulnerabilidades de corrupción de memoria y permitiendo una integración más segura entre APIs modernas basadas en `std::string_view` y componentes heredados que aún requieren cadenas terminadas en nulo.

Codificación insegura

Gestión de cadenas estilo C usando punteros a `char` sin garantizar terminación en null. Se busca mostrar los riesgos de asumir que un buffer es una cadena válida terminada en `'\0'`, lo que puede provocar lecturas fuera de límites y comportamiento indefinido.

```
#include <iostream>
#include <cstring>

//-- Función que asume que recibe una cadena terminada en null
void imprimirLongitud(const char* texto) {
  //-- Uso inseguro de strlen sin validar terminación
  size_t longitud = std::strlen(texto);
  std::cout << "Longitud: " << longitud << std::endl;
}

int main() {
  //-- Buffer sin terminador null
  char buffer[5] = {'H','o','l','a','!'};

  //-- Llamada que provoca comportamiento indefinido
  imprimirLongitud(buffer);

  return 0;
}
```

Codificación recomendada

Uso de un tipo específico para representar cadenas estilo C terminadas en null. Se pretende reforzar el contrato de que la función recibe una cadena válida y correctamente terminada, evitando ambigüedades y reduciendo el riesgo de errores por buffers no terminados.

```
#include <iostream>
#include <cstring>

//-- Definición de tipo equivalente a zstring (cadena terminada en null)
using zstring = const char*;

//-- Función que exige explícitamente una cadena terminada en null
void imprimirLongitud(zstring texto) {
  //-- Uso seguro al respetar el contrato de terminación
  size_t longitud = std::strlen(texto);
  std::cout << "Longitud: " << longitud << std::endl;
}

int main() {
  //-- Cadena correctamente terminada en null
```

```
    const char* mensaje = "Hola!";

    //-- Llamada segura respetando el contrato de zstring
    imprimirLongitud(mensaje);

    return 0;
}
```

2.2.4.1 Descripción

El uso de `zstring` y tipos equivalentes para representar cadenas de estilo C tiene como propósito proporcionar una interfaz más segura y expresiva que la que ofrece un simple puntero `char*`. En C++, el manejo de memoria manual y la ausencia de información explícita sobre la longitud de las cadenas suelen ser fuente de fallos de seguridad, especialmente cuando se interactúa con APIs heredadas o bibliotecas externas que esperan cadenas terminadas en nulo. Un tipo como `zstring` hace explícito el contrato de que el recurso apuntado debe estar terminado en nulo, permitiendo así que herramientas automáticas, revisores de código y el propio compilador puedan detectar estados ilegales o usos incorrectos. El objetivo principal es evitar errores críticos como accesos fuera de límites, desbordamientos de buffer y lecturas de memoria no inicializada, los cuales pueden comprometer la integridad del sistema.

En C++, uno de los mayores retos vinculados al manejo de cadenas proviene del hecho de que los punteros a caracteres carecen de semántica y garantías de seguridad. Un puntero crudo no proporciona información sobre la longitud de la cadena, si esta está terminada correctamente en nulo o si la memoria asociada sigue siendo válida. El empleo de **zstring** explicita la expectativa del programador y evita interpretaciones incorrectas o inseguras. Cuando una API recibe o devuelve un `zstring`, se comunica de forma clara qué tipo de cadena es aceptable y cuáles son los requisitos para su validez. Esto contribuye a mejorar la robustez del software y a evitar vulnerabilidades frecuentes como las causadas por funciones clásicas inseguras (`strcpy`, `strcat`, `strlen` mal utilizadas, entre otras).

El uso de un tipo seguro para cadenas estilo C permite introducir controles fuertes desde el propio compilador, evitando así errores en tiempo de ejecución. Además, facilita migrar gradualmente de cadenas heredadas a alternativas más modernas como `std::string` o `std::string_view` sin sacrificar compatibilidad. Esta práctica también mejora la auditabilidad del código, ya que permite identificar de inmediato qué funciones dependen de cadenas terminadas en nulo y cuáles no. Como resultado, se reducen significativamente riesgos asociados a la integridad de memoria, incluyendo corrupción de pilas y montículos, que pueden ser explotadas para ejecutar código arbitrario o comprometer datos sensibles.

Cuando se trabaja con APIs externas escritas en C o con código heredado, es habitual encontrar funciones que requieren cadenas terminadas en nulo. Usar `zstring` permite

documentar explícitamente esta expectativa, evitando pasar erróneamente un objeto `std::string` cuya gestión de memoria sea más compleja o cuyo contenido no garantice un terminador válido. También se facilita la adaptación de código moderno basado en `std::string_view`, el cual no garantiza la presencia de un terminador nulo. Por tanto, distinguir entre cadenas vistas (`string_view`) y cadenas terminadas en nulo (`zstring`) permite un diseño más seguro y previsivo.

Además del uso de `zstring`, se recomienda aplicar otras prácticas de seguridad complementarias para el manejo de cadenas, tales como el uso de **funciones seguras de copia**, la validación estricta de longitudes, el establecimiento de límites fijos y el empleo de análisis estático para detectar usos indebidos. La combinación de estos enfoques proporciona una defensa robusta contra vulnerabilidades de corrupción de memoria.

Riesgos asociados

- **Comportamiento indefinido por ausencia de terminador nulo**: tratar un `const char*` genérico como cadena C sin garantizar `'\0'` puede provocar lecturas fuera de límites.
- **Lectura fuera de rango (out-of-bounds read)**: funciones que dependen de terminación nula (`strlen`, `printf`, etc.) pueden recorrer memoria adyacente si la cadena no está correctamente terminada.
- **Desbordamiento de buffer en operaciones de copia o concatenación**: sin un tipo semánticamente claro como `zstring`, es más probable usar APIs inseguras sobre buffers no verificados.
- **Ambigüedad contractual en APIs**: aceptar `const char*` sin especificar que debe ser cadena terminada en cero genera contratos implícitos y propensos a error.
- **Confusión entre puntero a datos binarios y cadena textual**: un `char*` puede apuntar a datos no textuales; sin tipado explícito, se pueden aplicar funciones de cadena sobre datos arbitrarios.
- **Errores de interoperabilidad con código legacy**: interfaces C pueden devolver punteros no terminados en cero o con ciclo de vida incierto, introduciendo riesgos si no se tipan adecuadamente.
- **Uso indebido de memoria no válida o liberada**: al no diferenciar claramente tipos propietarios y no propietarios, aumenta el riesgo de `use-after-free`.
- **Falta de validación de nulidad**: sin convenciones como `zstring` (cadena C válida no nula), puede pasarse `nullptr` a funciones que asumen una cadena válida.
- **Dificultad para análisis estático y revisión de seguridad**: el uso indiscriminado de `char*` complica determinar si el puntero representa una cadena válida, datos binarios o memoria no inicializada.
- **Mayor superficie para vulnerabilidades clásicas de C**: el uso de punteros crudos

en lugar de tipos semánticos explícitos incrementa la probabilidad de errores históricos asociados a cadenas C.

Buenas prácticas

- Diferenciar explícitamente cadenas C válidas de punteros genéricos: usar alias semánticos para expresar que el puntero debe estar terminado en '\0'.

```
using zstring = const char*;

//-- puntero genérico sin semántica clara
void print1(const char* p) {
  std::cout << p << "\n";
}
```

```
//-- contrato explícito de cadena C válida
void print2(zstring p) {
  std::cout << p << "\n";
}
```

- Validar nulidad antes de usar un `zstring`: garantizar precondiciones antes de invocar funciones dependientes de terminador nulo.

```
using zstring = const char*;

//-- no valida nullptr
void log1(zstring s) {
  std::cout << s << "\n";
}
```

```
//-- validación explícita
void log2(zstring s) {
  if (s == nullptr) {
    throw std::invalid_argument("zstring null");
  }
  std::cout << s << "\n";
}
```

- Convertir inmediatamente a `std::string` cuando se requiera propiedad: evitar depender del ciclo de vida externo.

```
using zstring = const char*;

//-- almacena puntero no propietario
class User1 {
public:
  User1(zstring name) : name_(name) {}
```

```
private:
  const char* name_;
};
```

```
//-- copia segura con propiedad
class User2 {
public:
  User2(zstring name) : name_(name ? name : "") {}
private:
  std::string name_;
};
```

- No asumir terminación nula si el origen no lo garantiza: validar antes de usar funciones dependientes de '\0'.

```
using zstring = const char*;

//-- asume terminación válida
std::size_t length1(zstring s) {
  return std::strlen(s);
}
```

```
//-- valida antes de usar
std::size_t length2(zstring s) {
  if (s == nullptr) {
    throw std::invalid_argument("null zstring");
  }
  return std::strlen(s);
}
```

- Usar tipos más fuertes cuando sea posible (`std::string_view`): expresar no-propiedad sin depender exclusivamente de terminador nulo.

```
using zstring = const char*;

//-- depende solo de terminación nula
void show1(zstring s) {
  std::cout << s << "\n";
}
```

```
//-- semántica explícita de vista no propietaria
void show2(std::string_view s) {
  std::cout << s << "\n";
}
```

2.2.4.2 Estándares metodológicos

- **C++ Core-F.25**: Use a zstring or a not_null<zstring> to designate a C-style string.

2.3 Enumeraciones

2.3.1 [cpp.enum.1] - Preferir enumeraciones frente a macros para definir conjuntos de valores

ID	cpp.enum.1	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

El uso de enumeraciones aporta una capa adicional de seguridad y robustez frente al empleo de macros tradicionales para definir valores constantes. Las enumeraciones, especialmente las **enum class**, ofrecen tipado fuerte, reducen colisiones de nombres y facilitan el análisis estático, evitando comportamientos inseguros o ambiguos asociados a macros expansivas del preprocesador. Esta práctica mejora la claridad del código, previene errores involuntarios y disminuye la superficie de vulnerabilidades generadas por un manejo inadecuado de valores constantes en módulos críticos.

Codificación insegura

Uso de macros para definir un conjunto de estados posibles de una aplicación. Se busca representar distintos estados de un proceso y mostrar un mensaje según el valor seleccionado, destacando cómo se gestionan constantes enteras simbólicas.

```
#include <iostream>

//-- Definición de estados mediante macros
#define ESTADO_INICIO 0
#define ESTADO_PROCESANDO 1
#define ESTADO_FINALIZADO 2

int main() {
  //-- Variable que representa el estado actual
  int estado = ESTADO_PROCESANDO;

  //-- Evaluación del estado
  if (estado == ESTADO_INICIO) {
    std::cout << "Estado: Inicio" << std::endl;
  } else if (estado == ESTADO_PROCESANDO) {
    std::cout << "Estado: Procesando" << std::endl;
  } else if (estado == ESTADO_FINALIZADO) {
    std::cout << "Estado: Finalizado" << std::endl;
  } else {
    //-- Valor no contemplado
```

```
    std::cout << "Estado desconocido" << std::endl;
  }

  return 0;
}
```

Codificación recomendada

Uso de una enumeración fuertemente tipada para representar los mismos estados del proceso. Se pretende mostrar cómo limitar los valores posibles, mejorar la seguridad de tipos y evitar conversiones implícitas no deseadas.

```
#include <iostream>

//-- Definición de estados mediante enumeración fuertemente tipada
enum class Estado {
  Inicio,
  Procesando,
  Finalizado
};

int main() {
  //-- Variable que representa el estado actual
  Estado estado = Estado::Procesando;

  //-- Evaluación del estado
  switch (estado) {
    case Estado::Inicio:
      std::cout << "Estado: Inicio" << std::endl;
      break;
    case Estado::Procesando:
      std::cout << "Estado: Procesando" << std::endl;
      break;
    case Estado::Finalizado:
      std::cout << "Estado: Finalizado" << std::endl;
      break;
  }

  return 0;
}
```

2.3.1.1 Descripción

En C++, el reemplazo de macros que representan conjuntos de valores por enumeraciones fuertemente tipadas constituye una medida esencial para reforzar la integridad del software y minimizar riesgos asociados al preprocesador. Las macros, al ser simples sustituciones de texto, carecen de comprobaciones de tipo, pueden

provocar colisiones de nombres y pueden expandirse de manera inesperada, dificultando tanto el análisis estático como la depuración. En contraste, las enumeraciones proporcionan un nivel de protección estructural que contribuye directamente a evitar comportamientos inseguros o resultados inconsistentes en tiempo de ejecución.

Las macros utilizadas para definir valores constantes suelen emplearse bajo la forma de `#define`, lo cual carece de cualquier tipo de verificación semántica. El compilador no valida ni el tipo ni el contexto, de modo que su utilización puede introducir errores sutiles capaces de producir fallos lógicos o vulnerabilidades. Además, las macros se expanden literalmente en el código, generando escenarios donde resulta difícil determinar el origen del valor expandido, comprometiendo la mantenibilidad y capacidad de auditoría. Por ello, sustituirlas por enumeraciones reduce significativamente estos riesgos.

Las **enum class** introducidas en C++11 han mejorado notablemente la seguridad en el manejo de conjuntos de valores. A diferencia de las enumeraciones tradicionales, **enum class** no permiten conversiones implícitas a enteros, ofrecen un espacio de nombres propio y evitan colisiones con identificadores externos. Estas propiedades también permiten que herramientas de análisis estático identifiquen con precisión el flujo de valores, reforzando la capacidad de detección temprana de errores.

- **Tipado fuerte**: impide asignaciones o comparaciones con valores no previstos.
- **Encapsulamiento de nombres**: evita colisiones con otros identificadores del programa.
- **Mayor control semántico**: el compilador detecta usos incorrectos.
- **Compatibilidad segura**: se integran mejor con plantillas y sobrecarga.

El empleo de enumeraciones en vez de macros permite garantizar que los valores utilizados en estructuras lógicas o módulos críticos no se modifiquen accidentalmente. Esto resulta primordial en sistemas donde la integridad de estados o transiciones es fundamental para evitar comportamientos indeterminados que puedan comprometer la seguridad. Al tratarse de tipos con reglas estrictas, se reduce la probabilidad de asignar valores incorrectos o recibir datos maliciosamente manipulados que exploten la naturaleza flexible de las macros.

Una de las debilidades del preprocesador es la opacidad que introduce en el código compilado. Las macros no dejan rastro semántico, solo sustituciones textuales. En cambio, las enumeraciones registran su significado estructural explícitamente en el AST del compilador, lo cual facilita herramientas de análisis forense o revisiones de seguridad. Esto mejora la trazabilidad y permite identificar flujos de datos sospechosos con mayor exactitud.

Las **enum class** se integran de forma naturalmente segura con plantillas genéricas, patrones de diseño y APIs modernas. Al tratarse de tipos concretos, pueden usarse sin

riesgo en funciones sobrecargadas, estructuras condicionales optimizadas y sistemas basados en meta-programación. Este comportamiento coherente es fundamental para evitar errores silenciosos en bibliotecas complejas de terceros, así como en desarrollos internos donde la seguridad depende del correcto manejo de estados.

Riesgos asociados

- Confusión en el tipo de dato, ya que las macros no proporcionan verificación de tipo, lo que puede provocar errores de asignación y uso incorrecto.
- Difícil depuración, porque las macros se expanden en tiempo de preprocesamiento y no aparecen en el depurador como entidades con nombre propio.
- Posibilidades de colisiones de nombres, dado que las macros no respetan el alcance (namespace), aumentando la probabilidad de sobrescribir identificadores existentes.
- Errores inadvertidos, ya que las macros no respetan las reglas de ámbito del lenguaje, pudiendo resultar en comportamientos inesperados en diferentes partes del código.
- Menor legibilidad del código, porque el uso de macros dificulta la comprensión del propósito y el valor de los elementos, frente a las enumeraciones que aportan contexto y sentido de conjunto.
- Problemas de mantenimiento, ya que modificar macros puede tener efectos secundarios no esperados en todo el proyecto.
- Vulnerabilidades de seguridad derivadas de la suplantación y redefinición accidental/maliciosa de macros.
- Ausencia de garantías de unicidad, ya que múltiples macros pueden tener el mismo valor, mientras que las enumeraciones garantizan un único identificador por valor.

Buenas prácticas

- Utilizar `enum` o `enum class` en lugar de macros `#define` para definir conjuntos cerrados de valores constantes.

```
//-- Definición de constantes mediante macros sin tipado ni ámbito
#define STATUS_OK 0
#define STATUS_ERROR 1

int status = STATUS_OK;
```

```
//-- Definición de conjunto de valores tipado y limitado al ámbito del enum
enum class Status {
```

```
  Ok = 0,
  Error = 1
};

Status status = Status::Ok;
```

- Preferir `enum class` para evitar conversiones implícitas y colisiones en el espacio de nombres.

```
//-- Uso de macros que permiten asignaciones arbitrarias sin control de tipo
#define ROLE_ADMIN 1
#define ROLE_USER 2

int role = ROLE_ADMIN;
role = 999;
```

```
//-- Enumeración fuertemente tipada que restringe valores y evita conversiones
implícitas
enum class Role {
  Admin = 1,
  User = 2
};

Role role = Role::Admin;
// role = 999;
```

- Encapsular los conjuntos de valores en un ámbito específico para reducir exposición innecesaria.

```
//-- Macros globales visibles en toda la unidad de compilación
#define MODE_READ 0
#define MODE_WRITE 1

int mode = MODE_READ;
```

```
//-- Enumeración declarada dentro de una clase para limitar su alcance
class File {
public:
  enum class Mode {
    Read = 0,
    Write = 1
  };
};

File::Mode mode = File::Mode::Read;
```

- Evitar el uso de macros para valores que participan en lógica condicional, favoreciendo enumeraciones que mejoran la legibilidad y el análisis estático.

```
//-- Comparaciones basadas en macros sin validación semántica
#define LEVEL_LOW 0
#define LEVEL_HIGH 1

int level = LEVEL_LOW;
if (level == LEVEL_HIGH) {
  // ...
}
```

```
//-- Uso de enumeración que define explícitamente los estados permitidos
enum class Level {
  Low = 0,
  High = 1
};

Level level = Level::Low;
if (level == Level::High) {
  // ...
}
```

- Especificar el tipo subyacente de la enumeración cuando el tamaño o la interoperabilidad binaria sea relevante.

```
//-- Macros utilizadas para definir códigos que se envían por red
#define CODE_SUCCESS 0x01
#define CODE_FAILURE 0xFF

unsigned char code = CODE_SUCCESS;
```

```
//-- Enumeración con tipo subyacente explícito para controlar representación
binaria
enum class Code : unsigned char {
  Success = 0x01,
  Failure = 0xFF
};

Code code = Code::Success;
```

2.3.1.2 Estándares metodológicos

- **C++ Core-Enum.1**: Prefer enumerations over macros.

2.3.2 [cpp.enum.2] - Preferir "enum class" frente a enumeraciones no fuertemente tipadas

ID	cpp.enum.2	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Este control promueve el uso de `enum class` para reforzar la seguridad del código al impedir conversiones implícitas, mejorar la claridad semántica y reducir riesgos de errores relacionados con valores no válidos o mezclas accidentales de enumeraciones. Al ser fuertemente tipados, estos enumeradores permiten un mayor control de integridad y disminuyen fallos en operaciones lógicas y de comparación, aportando mayor robustez y mantenibilidad dentro de entornos de desarrollo seguro.

Codificación insegura

Uso de una enumeración tradicional para representar estados de un sistema. Se muestra cómo la falta de tipado fuerte permite conversiones implícitas a enteros y comparaciones no deseadas, lo que puede introducir errores lógicos difíciles de detectar en tiempo de compilación.

```
#include <iostream>

enum Estado {
  //-- Definición de una enumeración no fuertemente tipada
  APAGADO = 0,
  ENCENDIDO = 1
};

int main() {
  //-- La enumeración se convierte implícitamente a entero
  Estado estado = ENCENDIDO;

  //-- Comparación con un valor entero sin restricción de tipo
  if (estado == 1) {
    std::cout << "Sistema encendido" << std::endl;
  }

  //-- Asignación desde un entero sin validación de dominio
  estado = static_cast<Estado>(5);

  //-- El compilador no impide valores fuera del rango lógico
  std::cout << "Valor actual de estado: " << estado << std::endl;
```

```
  return 0;
}
```

Codificación recomendada

Uso de una enumeración fuertemente tipada para representar estados del sistema. Se observa cómo el tipado fuerte evita conversiones implícitas a enteros y obliga a comparaciones explícitas, reduciendo errores y mejorando la seguridad del código en tiempo de compilación.

```
#include <iostream>

enum class Estado {
  //-- Definición de una enumeración fuertemente tipada
  APAGADO = 0,
  ENCENDIDO = 1
};

int main() {
  //-- La enumeración no se convierte implícitamente a entero
  Estado estado = Estado::ENCENDIDO;

  //-- Comparación segura usando el mismo tipo
  if (estado == Estado::ENCENDIDO) {
    std::cout << "Sistema encendido" << std::endl;
  }

  //-- Para obtener el valor entero se requiere conversión explícita
  int valor = static_cast<int>(estado);
  std::cout << "Valor entero del estado: " << valor << std::endl;

  return 0;
}
```

2.3.2.1 Descripción

El uso de **enum class** en C++ constituye una práctica esencial de codificación segura, ya que reemplaza a las enumeraciones tradicionales que carecen de tipado estricto y permiten conversiones implícitas hacia tipos numéricos. Estas conversiones pueden introducir fallos sutiles, pérdida de integridad lógica y comportamientos inesperados durante la ejecución. Esta recomendación busca asegurar que las enumeraciones en C++ mantengan integridad semántica, evitando colisiones de nombres, inconsistencias y accesos indebidos a valores que no representan estados válidos dentro de un contexto determinado. El principio general consiste en reforzar la integridad mediante el control estricto del tipo y un espacio de nombres claramente delimitado.

La enumeración tradicional en C++ expone varios riesgos relevantes. En primer lugar, múltiples enumeraciones distintas pueden compartir nombres idénticos, generando ambigüedades y reduciendo la claridad del código. En segundo lugar, la conversión implícita hacia `int` habilita la posibilidad de mezclar tipos sin advertencia del compilador, lo cual es especialmente peligroso desde la perspectiva de la seguridad, dado que se pueden ejecutar comparaciones o asignaciones ilícitas sin informar a los desarrolladores. Además, estas debilidades pueden facilitar la introducción de errores que comprometen la integridad de estados internos, particularmente en sistemas que dependen de condiciones lógicas estrictas o flujos de control sensibles.

El uso de **enum class** mitiga estas amenazas mediante tres propiedades fundamentales. La primera es la **fuerte tipificación**, que obliga a realizar conversiones explícitas cuando es necesario, disminuyendo drásticamente el riesgo de errores accidentales. La segunda es el **encapsulamiento de nombres**, ya que los enumeradores se exponen solo dentro del ámbito del tipo declarado, eliminando colisiones y mejorando la legibilidad del código. La tercera es la **definición clara de límites**, que obliga al programador a tratarlos como entidades lógicas independientes, reforzando la precisión al manipular estados en componentes críticos.

En contextos de seguridad, esta práctica aporta beneficios significativos como aislar representaciones internas evitando asignaciones arbitrarias provenientes de variables externas o de datos manipulados por entradas no confiables, o evitar que estados inválidos sean introducidos en estructuras de control, lo que podría desencadenar fallos lógicos, corrupción de datos o incluso vulnerabilidades explotables en sistemas complejos. El uso de enumeraciones fuertemente tipadas, por tanto, incrementa la seguridad defensiva al endurecer los límites del tipo y promover patrones de validación consistentes.

Además de evitar comparaciones incorrectas, este enfoque evita que valores enteros no válidos sean asignados accidentalmente a las variables que representan estados críticos. Cualquier conversión hacia o desde un tipo numérico debe realizarse de forma explícita, lo que introduce puntos claros de revisión y validación durante auditorías de seguridad. En un proyecto de gran escala o con múltiples equipos, esta propiedad mejora la cohesión conceptual y evita discrepancias en la interpretación de estados.

Para aumentar la robustez, es recomendable seguir ciertas pautas adicionales al emplear `enum class`. Por ejemplo, definir explícitamente los valores internos solo cuando sea necesario para interoperabilidad o compatibilidad con protocolos binarios; de lo contrario, delegar la asignación al compilador reduce errores. También se aconseja acompañar estas enumeraciones con funciones auxiliares de conversión controlada, donde puedan implementarse validaciones estrictas, sanitización de entradas y manejo cuidadoso de errores.

En sistemas críticos donde la fiabilidad es esencial, la combinación del tipado fuerte junto con reglas estrictas de validación y documentación de las enumeraciones proporciona un marco sólido para prevenir degradaciones de integridad. Del mismo

modo, en aplicaciones modernas orientadas a microservicios y arquitecturas distribuidas, emplear tipos bien definidos para representar estados reduce errores de comunicación entre módulos y facilita el mantenimiento seguro a largo plazo.

Riesgos asociados

- Colisiones de nombres entre enumeradores de diferentes enum.
- Conversión implícita de enumeradores a tipos enteros, lo que puede facilitar errores de lógica y vulnerabilidades.
- Uso de valores fuera del rango válido del enum, facilitando valores no definidos o inválidos.
- Comparaciones entre diferentes tipos de enum que pueden generar comportamientos inesperados o inseguros.
- Pérdida de control sobre el ámbito (scope) de los elementos del enum, dificultando el mantenimiento y la legibilidad.
- Incremento en la superficie de ataque para errores de tipo que pueden derivar en corrupción de memoria.
- Dificultad para garantizar la validación y manejo seguro de los valores del enum.

Buenas prácticas

- Utilizar `enum class` para evitar conversiones implícitas a tipos enteros.

```
//-- Conversión implícita a entero
enum Estado { ACTIVO, INACTIVO };

Estado e = ACTIVO;
int valor = e;
```

```
//-- Uso de enum class evitando conversión implícita
enum class Estado { ACTIVO, INACTIVO };

Estado e2 = Estado::ACTIVO;
int valor2 = static_cast(e2);
```

- Evitar colisiones de nombres utilizando el ámbito propio de `enum class`.

```
//-- Posible colisión de identificadores en el mismo ámbito
enum Color { ROJO, VERDE };
enum Alerta { VERDE, AMARILLO };
```

```
//-- Enumeraciones con ámbito propio evitando colisiones
enum class Color { ROJO, VERDE };
```

```
enum class Alerta { VERDE, AMARILLO };

Color c = Color::VERDE;
Alerta a = Alerta::VERDE;
```

- Evitar comparaciones entre distintos tipos de enumeraciones usando `enum class`.

```
//-- Comparación implícita entre enumeraciones distintas
enum Estado { ENCENDIDO, APAGADO };
enum Puerta { ABIERTA, CERRADA };

bool condicion = (ENCENDIDO == ABIERTA);
```

```
//-- Tipos fuertemente tipados que impiden comparaciones inválidas
enum class Estado { ENCENDIDO, APAGADO };
enum class Puerta { ABIERTA, CERRADA };

Estado e = Estado::ENCENDIDO;
Puerta p = Puerta::ABIERTA;
```

- Especificar explícitamente el tipo subyacente en `enum class` cuando se requiera control de tamaño o interoperabilidad.

```
//-- Tipo subyacente implícito dependiente del compilador
enum Estado { OK = 0, ERROR = 1 };
```

```
//-- Tipo subyacente explícito para control de tamaño
enum class Estado : unsigned char { OK = 0, ERROR = 1 };

Estado e = Estado::OK;
```

2.3.2.2 Estándares metodológicos

- **C++ Core-Enum.3**: Prefer enum classes over "plain" enums.

2.3.3 [cpp.enum.3] - Evitar convertir a un valor de enumeración fuera de su rango válido

ID	cpp.enum.3	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este control de seguridad busca asegurar que toda conversión hacia tipos de `enum` se realice únicamente con valores válidos, evitando así comportamientos indefinidos, fallos lógicos y vulnerabilidades explotables. Convertir valores numéricos arbitrarios en enumeraciones puede permitir que un estado inesperado llegue a flujos críticos, habilite accesos no previstos o afecte la lógica de validación. El requisito promueve el uso de comprobaciones explícitas, tipos con alcance (`enum class`) y verificaciones estrictas antes de realizar cast, reforzando la integridad del software.

Codificación insegura

Uso de un valor entero externo para asignarlo directamente a un tipo enumerado sin validar su rango. Se busca mostrar qué ocurre cuando se convierte un valor fuera de los definidos en la enumeración y cómo esto puede generar estados lógicos inválidos en el programa.

```
#include <iostream>

enum class Estado {
  INACTIVO = 0,
  ACTIVO = 1,
  SUSPENDIDO = 2
};

int main() {
  //-- Valor recibido de una fuente externa (ej. usuario o red)
  int valorRecibido = 5;

  //-- Conversión directa sin validar el rango
  Estado estado = static_cast<Estado>(valorRecibido);

  //-- Uso del valor convertido
  if (estado == Estado::ACTIVO) {
    std::cout << "Estado ACTIVO" << std::endl;
  } else if (estado == Estado::INACTIVO) {
    std::cout << "Estado INACTIVO" << std::endl;
  } else if (estado == Estado::SUSPENDIDO) {
    std::cout << "Estado SUSPENDIDO" << std::endl;
  } else {
    std::cout << "Estado desconocido" << std::endl;
  }
```

```
  return 0;
}
```

Codificación recomendada

Validación explícita del valor entero antes de convertirlo al tipo enumerado. Se pretende mostrar cómo asegurar que solo los valores definidos en la enumeración sean utilizados, evitando estados inválidos y comportamientos lógicos inesperados.

```
#include <iostream>
#include <optional>

enum class Estado {
  INACTIVO = 0,
  ACTIVO = 1,
  SUSPENDIDO = 2
};

std::optional<Estado> convertirAEstado(int valor) {
  //-- Validación explícita del rango permitido
  switch (valor) {
    case 0: return Estado::INACTIVO;
    case 1: return Estado::ACTIVO;
    case 2: return Estado::SUSPENDIDO;
    default:
      return std::nullopt;
  }
}

int main() {
  //-- Valor recibido de una fuente externa (ej. usuario o red)
  int valorRecibido = 1;

  //-- Conversión segura con validación previa
  std::optional<Estado> estado = convertirAEstado(valorRecibido);

  //-- Uso del valor solo si es válido
  if (estado.has_value()) {
    std::cout << "Conversión válida" << std::endl;
  } else {
    std::cout << "Valor fuera de rango permitido" << std::endl;
  }

  return 0;
}
```

2.3.3.1 Descripción

La conversión de valores enteros hacia tipos `enum` en C++ es una operación potencialmente peligrosa cuando no se valida previamente el rango permitido. C++ permite realizar cast explícito, aunque el valor no pertenezca a ninguno de los elementos definidos en la enumeración, lo que conduce a la creación de valores inválidos que pueden propagarse silenciosamente por el programa. Los resultados incluyen comportamientos indefinidos, decisiones lógicas incorrectas, acceso a recursos no autorizados y exposición a fallos de seguridad derivados de condiciones inesperadas.

Los valores fuera de rango en tipos `enum` suelen ser peligrosos porque se utilizan como selectores lógicos, indicadores de estado, discriminadores en estructuras complejas, índices o claves en tablas. Si un atacante logra influir en los datos de entrada o si un error interno origina valores inválidos, el programa puede ejecutar caminos no previstos. Esto puede materializar vulnerabilidades como corrupción de memoria, comprobaciones omitidas o selección de operaciones inseguras. **Evitar valores no definidos en enumeraciones es esencial para salvaguardar la integridad de la lógica del sistema**.

Riesgos asociados

- Comportamiento indefinido en tiempo de ejecución.
- Generación de errores lógicos y fallos inesperados.
- Exposición a vulnerabilidades de seguridad (por ejemplo, ataques de corrupción de memoria).
- Dificultad en la depuración y el mantenimiento del código.
- Posibilidad de comprometer la integridad de los datos.
- Provocar excepciones, caídas del programa o bloqueos.
- Resultados inesperados por uso de valores no contemplados en lógica de negocio.
- Incremento en la superficie de ataque para exploits.

Buenas prácticas

- Validar explícitamente el rango antes de convertir un valor entero a un tipo enumerado.

```
//-- Conversión directa sin validar el rango permitido
enum class Estado { Inactivo = 0, Activo = 1, Suspendido = 2 };

int valor = obtenerEstadoExterno();
Estado e = static_cast<Estado>(valor);
```

```
//-- Validación previa del rango antes de realizar la conversión
enum class Estado { Inactivo = 0, Activo = 1, Suspendido = 2 };

int valor = obtenerEstadoExterno();
if (valor >= static_cast<int>(Estado::Inactivo) && valor <=
static_cast<int>(Estado::Suspendido)) {
  Estado e = static_cast<Estado>(valor);
}
```

- Encapsular la conversión en una función segura que controle valores inválidos.

```
//-- Conversión repetida sin control centralizado
enum class Nivel { Bajo = 1, Medio = 2, Alto = 3 };

Nivel n = static_cast<Nivel>(dato);
```

```
//-- Función que valida y controla valores fuera de rango
enum class Nivel { Bajo = 1, Medio = 2, Alto = 3 };

std::optional<Nivel> convertirANivel(int valor) {
  if (valor >= 1 && valor <= 3) {
    return static_cast<Nivel>(valor);
  }
  return std::nullopt;
}
```

- Evitar conversiones implícitas usando `enum class` y deshabilitando cast inseguros.

```
//-- Uso de enum tradicional que permite conversiones implícitas peligrosas
enum Rol { Usuario = 0, Administrador = 1 };

int valor = 5;
Rol r = (Rol)valor;
```

```
//-- Uso de enum class que exige conversión explícita y validación previa
enum class Rol { Usuario = 0, Administrador = 1 };

int valor = 1;
if (valor == 0 || valor == 1) {
  Rol r = static_cast<Rol>(valor);
}
```

2.3.3.2 Estándares metodológicos

- **CERT/CPP-INT50**: Do not cast to an out-of-range enumeration value.

2.4 Conversiones

2.4.1 [cpp.conv.1] - Evitar conversiones que cambien el signo o la categoría del tipo

ID	cpp.conv.1	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Establece la necesidad de prevenir conversiones implícitas o explícitas que puedan modificar el signo o la categoría del tipo original, ya que tales transformaciones generan comportamientos inesperados, vulnerabilidades de memoria, errores de límites y desviaciones lógicas. El objetivo es garantizar que el código preserve la integridad de los datos, evitando pérdidas de información, desbordamientos o reinterpretaciones peligrosas que puedan comprometer la seguridad. Se deben aplicar validaciones, utilizar tipos explícitos y favorecer APIs seguras que eviten conversiones inseguras y mantengan un control estricto sobre el flujo de datos.

Codificación insegura

Gestión de valores numéricos donde se mezclan tipos con signo y sin signo. Se busca mostrar cómo una conversión implícita puede cambiar el valor real almacenado y provocar resultados inesperados en tiempo de ejecución, especialmente al trabajar con tamaños o contadores.

```
#include <iostream>
#include <vector>

int main() {
  //-- Se declara un valor negativo que representa un tamaño inválido
  int cantidad = -5;

  //-- Conversión implícita de int (con signo) a size_t (sin signo)
  size_t tam = cantidad;

  //-- Se intenta crear un vector usando el tamaño convertido
  std::vector<int> datos(tam);

  //-- Se muestra el tamaño real del vector
  std::cout << "Tamano del vector: " << datos.size() << std::endl;

  return 0;
}
```

Codificación recomendada

Control de valores antes de realizar conversiones entre tipos con signo y sin signo. Se valida el rango del dato original y se evita la conversión implícita peligrosa, garantizando que el tamaño usado sea coherente y seguro.

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

int main() {
  //-- Se declara un valor que podría provenir de entrada externa
  int cantidad = -5;

  //-- Validación explícita antes de convertir a un tipo sin signo
  if (cantidad < 0) {
    //-- Se rechaza el valor negativo antes de la conversión
    std::cerr << "Error: cantidad negativa no permitida" << std::endl;
    return 1;
  }
  //-- Conversión explícita y segura tras la validación
  size_t tam = static_cast<size_t>(cantidad);

  //-- Uso seguro del tamaño validado
  std::vector<int> datos(tam);
  std::cout << "Tamano del vector: " << datos.size() << std::endl;
  return 0;
}
```

2.4.1.1 Descripción

Las conversiones de tipos pueden introducir vulnerabilidades severas cuando modifican el signo o la categoría numérica del valor original sin un control explícito. Hay que evitar que los tipos con signo sean convertidos a tipos sin signo, o viceversa, de forma implícita, accidental o mediante cast inapropiado. También, se debe evitar el cambio de categoría entre tipos estrechos y amplios sin validación previa, como pasar de `long` a `short` o de `double` a `int`. Estas acciones pueden generar valores inesperados que alteran la lógica de seguridad, los límites de memoria o los cálculos sensibles.

El cast implícito puede ocurrir incluso sin que el desarrollador sea completamente consciente de ello, ya que el lenguaje prioriza la flexibilidad y la conversión automática en muchos contextos. Por ejemplo, al hacer operaciones con diferentes tipos numéricos, el compilador puede promocionar o degradar tipos basándose en reglas complejas que no siempre están alineadas con la intención del programador. **El riesgo aumenta notablemente cuando se manipulan punteros, tamaños de memoria, índices de vectores o variables utilizadas en validaciones de seguridad**, ya que un valor reinterpretado como sin signo podría dar lugar a condiciones siempre verdaderas o

índices que recorren memoria fuera de los límites permitidos. Para minimizar estos riesgos es imprescindible comprender cómo interactúan los tipos de datos numéricos entre sí y cómo el compilador decide realizar conversiones automáticas.

Una buena práctica es **evitar cualquier uso de cast estilo C**, ya que estos no dejan claras las intenciones y permiten conversiones inseguras sin control. En su lugar, deben preferirse utilizar `static_cast`, `numeric_cast` (cuando exista), o funciones de conversión seguras que incluyan validación previa. La introducción de comprobaciones explícitas evita pérdidas de rango y mantiene la integridad de los datos. Por otro lado, es recomendable emplear funciones como `std::clamp` o validaciones manuales para asegurar que un valor cabe en el tipo destino antes de proceder con la conversión.

Dentro del desarrollo seguro, se recomienda mantener una consistencia de tipos en todas las interfaces internas y externas, utilizando tipado fuerte cuando sea posible. Esto reduce adquisiciones accidentales de datos en formato incompatible y evita conversiones implícitas durante la interacción de módulos. El uso de alias tipados o clases envoltorio puede ayudar a imponer restricciones, haciéndose evidente cuando una conversión no autorizada intenta ejecutarse.

Riesgos asociados

- Pérdida de datos por truncamiento al convertir entre tipos de diferente tamaño o signo.
- Errores aritméticos inesperados, como desbordamientos o comportamientos indefinidos.
- Vulnerabilidad a fallos de seguridad como buffer overflows debido a cálculos incorrectos de tamaños.
- Comportamiento inesperado durante comparaciones y operaciones lógicas.
- Introducción de bugs difíciles de detectar que se manifiestan sólo bajo ciertas condiciones o plataformas.
- Explotación potencial por atacantes que manipulen los valores para provocar fallos de lógica o corrupción de memoria.
- Problemas de portabilidad entre sistemas con diferentes representaciones de tipos de datos.
- Posibilidad de violaciones de los límites de los arrays y otros contenedores.

Buenas prácticas

- Evitar conversiones implícitas entre tipos con distinto signo en comparaciones.

```
//-- Comparación entre entero con signo y sin signo
int valor = -1;
unsigned int limite = 10;
```

```
if (valor < limite) {
  procesar();
}
```

```
//-- Conversión explícita validando rango antes de comparar
int valor = -1;
unsigned int limite = 10;
if (valor >= 0 && static_cast(valor) < limite) {
  procesar();
}
```

- Evitar asignaciones que conviertan implícitamente tipos con posible pérdida de signo.

```
//-- Asignación implícita de entero negativo a tipo sin signo
int resultado = calcular();
unsigned int almacenamiento = resultado;
```

```
//-- Validación previa del rango antes de convertir
int resultado = calcular();
if (resultado >= 0) {
  unsigned int almacenamiento = static_cast(resultado);
}
```

- Evitar conversiones entre tipos de diferente categoría (por ejemplo, entero a booleano) sin validación explícita.

```
//-- Conversión implícita de entero a booleano
int estado = obtenerEstado();
bool activo = estado;
```

```
//-- Comparación explícita que define la condición esperada
int estado = obtenerEstado();
bool activo = (estado != 0);
```

- Preferir tipos coherentes en operaciones aritméticas para evitar promociones inesperadas.

```
//-- Mezcla de tipos con distinta categoría en operación aritmética
int a = -5;
unsigned int b = 3;
auto resultado = a + b;
```

```
//-- Uso de tipos homogéneos antes de realizar la operación
int a = -5;
```

```
int b = 3;
auto resultado = a + b;
```

2.4.1.2 Estándares metodológicos

- **C++ Core-ES.46**: Avoid narrowing conversions.
- **MISRA CPP-7.0.5**: Usual arithmetic conversions shall not change signedness/type category.

2.4.2 [cpp.conv.2] - Usar "nullptr" como única constante de puntero nulo

ID	cpp.conv.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

El uso consistente y exclusivo de `nullptr` evita ambigüedades presentes en antiguas construcciones como `NULL` o `0`, reduciendo riesgos de conversión implícita y errores sutiles en el manejo de punteros. Esta práctica fortalece la seguridad del código al facilitar la detección de fallos, mejorar la precisión del tipo y prevenir comportamientos inesperados durante la resolución de sobrecarga. Además, contribuye a un código más claro, mantenible y robusto frente a vulnerabilidades que pueden derivar de comparaciones incorrectas, inicializaciones peligrosas o ramas lógicas no intencionadas.

Codificación insegura

Se muestra el uso de constantes antiguas para representar punteros nulos en un escenario con sobrecarga de funciones. El objetivo es observar cómo el uso de 0 o NULL puede generar ambigüedad o comportamientos no deseados cuando existen funciones sobrecargadas que aceptan distintos tipos de parámetros.

```
#include <iostream>

//-- Definición de dos funciones sobrecargadas
void procesar(int valor) {
  //-- Se ejecuta cuando se recibe un entero
  std::cout << "Procesando entero: " << valor << std::endl;
}

void procesar(int* puntero) {
  //-- Se ejecuta cuando se recibe un puntero a entero
  if (puntero == NULL) {
    //-- Validación de puntero nulo usando NULL
    std::cout << "Procesando puntero nulo" << std::endl;
  } else {
    std::cout << "Procesando puntero con valor: " << *puntero << std::endl;
```

```
  }
}

int main() {
  //-- Uso de 0 como constante de puntero nulo
  procesar(0);
  //-- Uso de NULL como constante de puntero nulo
  procesar(NULL);
  return 0;
}
```

Codificación recomendada

Se presenta el mismo escenario utilizando nullptr como única constante para representar punteros nulos. El objetivo es mostrar cómo nullptr elimina ambigüedades en sobrecargas y aporta mayor claridad y seguridad de tipos en el código.

```
#include <iostream>

//-- Definición de dos funciones sobrecargadas
void procesar(int valor) {
  //-- Se ejecuta cuando se recibe un entero
  std::cout << "Procesando entero: " << valor << std::endl;
}

void procesar(int* puntero) {
  //-- Se ejecuta cuando se recibe un puntero a entero
  if (puntero == nullptr) {
    //-- Validación de puntero nulo usando nullptr
    std::cout << "Procesando puntero nulo" << std::endl;
  } else {
    std::cout << "Procesando puntero con valor: " << *puntero << std::endl;
  }
}

int main() {
  //-- Uso explícito de nullptr para indicar puntero nulo
  procesar(nullptr);
  return 0;
}
```

2.4.2.1 Descripción

El uso exclusivo de `nullptr` representa una de las prácticas fundamentales de codificación segura orientadas a eliminar ambigüedades históricas heredadas del lenguaje. Antes de la introducción de `nullptr` en C++11, era común emplear las constantes `NULL` o `0` para indicar punteros nulos. Sin embargo, estas alternativas

presentan problemas relacionados con conversiones implícitas, ambigüedades en la resolución de sobrecarga y una menor robustez en tiempo de compilación. Adoptar `nullptr` como única forma válida de expresar un puntero nulo mejora la seguridad del software, reduce la superficie de error, y ayuda a mantener un código más claro y menos propenso a defectos que puedan dar lugar a vulnerabilidades explotables.

El mayor beneficio de usar `nullptr` es la eliminación de conversiones implícitas entre valores enteros y punteros. Tanto `NULL` como `0` pueden ser confundidos con enteros ordinarios, lo que permite compilar código incorrecto que más tarde se comporta de forma inesperada. **Esto puede derivar en desreferencias de puntero inválidas, fallos de segmentación, corrupción de memoria y condiciones inseguras**. Al usar `nullptr`, el compilador puede inferir correctamente el tipo de un puntero nulo y detectar usos indebidos en tiempo de compilación, actuando como una barrera preventiva contra errores críticos.

`nullptr` contribuye a la robustez del mecanismo de sobrecarga. Con `NULL` o `0`, la selección de funciones sobrecargadas puede ser ambigua, ya que dichas constantes no son estrictamente punteros. **Al utilizar `nullptr`, la sobrecarga siempre selecciona la variante que recibe un puntero, evitando llamadas peligrosas o inesperadas**.

Esta práctica **mitiga errores de puntero nulo, evita conversiones no deseadas y previene fallos de control de flujo**. Esto es importante en sistemas embebidos, software crítico, módulos de comunicaciones, bibliotecas de acceso a hardware y componentes donde una desreferencia indebida puede comprometer la integridad o disponibilidad del sistema.

Cuando se usan herramientas de análisis estático de código, el uso consistente de `nullptr` mejora la facilidad para detectar errores al quedar la semántica del código más explícita, lo que permite la reducción de falsos positivos y el reporte de informes más precisos.

En bases de código antiguas, la sustitución sistemática de `NULL` por `nullptr` puede hacerse progresivamente para evitar riesgos. Es recomendable establecer reglas de análisis estático que detecten usos prohibidos de `NULL` y emitir advertencias. Esta migración **reduce deuda técnica y aumenta la resiliencia del código a largo plazo**.

Riesgos asociados

- Confusión y ambigüedad entre valores nulos y enteros al usar 0 o NULL como puntero nulo.
- Erros de compilación o warnings en código moderno (C++11 en adelante) por ambigüedad en sobrecarga de funciones.
- Incremento de la superficie de ataque para vulnerabilidades, como desreferenciación de punteros no válidos.
- Pérdida de portabilidad y compatibilidad con herramientas y bibliotecas

modernas de C++.

- Dificultad en el análisis estático y revisión de código, aumentando la probabilidad de errores introducidos.
- Menor legibilidad y mantenibilidad del código, llevando a defectos sutiles difíciles de detectar y corregir.
- Potencial introducción de fallos de lógica por sobrecargas equivocadas al pasar 0/NULL donde se espera un puntero nulo.

Buenas prácticas

- Evitar el uso de 0 o NULL para representar punteros nulos y utilizar exclusivamente nullptr para mejorar la seguridad de tipos.

```
//-- Uso de 0 como puntero nulo, genera ambigüedad en sobrecargas
void procesar(int valor);
void procesar(char* ptr);
procesar(0);
```

```
//-- Uso de nullptr, resolución clara y tipada
void procesar(int valor);
void procesar(char* ptr);
procesar(nullptr);
```

- Inicializar siempre los punteros con nullptr en el momento de su declaración para evitar valores indeterminados.

```
//-- Puntero sin inicializar, posible uso de valor basura
int* ptr;
if (ptr) {
  *ptr = 10;
}
```

```
//-- Inicialización explícita con nullptr
int* ptr = nullptr;
if (ptr != nullptr) {
  *ptr = 10;
}
```

- Comparar punteros explícitamente contra nullptr para mejorar legibilidad y evitar conversiones implícitas peligrosas.

```
//-- Comparación implícita que depende de conversión a booleano
int* ptr = obtenerRecurso();
if (ptr) {
```

```
  usar(ptr);
}
```

```
//-- Comparación explícita contra nullptr
int* ptr = obtenerRecurso();
if (ptr != nullptr) {
  usar(ptr);
}
```

2.4.2.2 Estándares metodológicos

- **C++ Core-ES.47**: Use nullptr rather than 0 or NULL.
- **MISRA CPP-7.11.1**: nullptr shall be the only null pointer constant.

2.4.3 [cpp.conv.3] - Evitar los "casts" o usar los casts prefijados

ID	cpp.conv.3	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este requisito establece la necesidad de minimizar los **casts inseguros** y favorecer el uso de casts prefijados como `static_cast`, `dynamic_cast`, `const_cast` y `reinterpret_cast` para asegurar una conversión explícita, controlada y verificable por el compilador. Su objetivo es reducir errores de tipo, evitar accesos indebidos a memoria y mitigar vulnerabilidades asociadas a comportamientos indefinidos. Con ello se mejora la robustez del código, la trazabilidad de las conversiones y la seguridad general de la aplicación.

Codificación insegura

Conversión de tipos numéricos utilizando un cast tradicional de estilo C. Se busca mostrar cómo una conversión implícita y poco controlada puede ocultar pérdidas de información o comportamientos inesperados, dificultando la revisión y el mantenimiento del código.

```
#include <iostream>

int main() {
  //-- Declaración de un valor en punto flotante
  double precio = 10.75;

  //-- Conversión usando cast de estilo C
  int precioEntero = (int)precio;

  //-- Impresión del resultado convertido
```

```
  std::cout << "Precio original: " << precio << std::endl;
  std::cout << "Precio convertido: " << precioEntero << std::endl;
  return 0;
}
```

Codificación recomendada

Conversión explícita de tipos utilizando casts prefijados del lenguaje C++. Se busca mostrar cómo el uso de `static_cast` hace más clara la intención del programador y permite un mejor control y revisión del tipo de conversión realizada.

```
#include <iostream>

int main() {
  //-- Declaración de un valor en punto flotante
  double precio = 10.75;

  //-- Conversión explícita usando static_cast
  int precioEntero = static_cast<int>(precio);

  //-- Impresión del resultado convertido
  std::cout << "Precio original: " << precio << std::endl;
  std::cout << "Precio convertido: " << precioEntero << std::endl;
  return 0;
}
```

2.4.3.1 Descripción

El lenguaje C++ ofrece una gran flexibilidad en la conversión de tipos, pero también impone riesgos cuando se emplean mecanismos de conversión inseguros como el cast estilo C (`(T)x`) o el cast funcional (`T(x)`). Estos últimos pueden ejecutar conversiones diferentes según el contexto, a menudo sin control ni visibilidad del programador, lo que conduce a comportamientos indefinidos difíciles de detectar. El uso de **casts prefijados** proporciona una semántica clara, explícita y restrictiva, permitiendo al desarrollador comunicar de manera inequívoca la intención de la conversión, así como limitar los riesgos inherentes al manejo de punteros y objetos complejos.

Las directrices de C++ Core Guidelines, en particular ES.48 y ES.49, destacan la importancia de evitar los cast estilo C y preferir los prefijados. La razón fundamental es que cada tipo de cast prefijado cumple un propósito delimitado. Por ejemplo, `static_cast` controla la conversión entre tipos numéricos o clases relacionadas con herencia; `dynamic_cast` ofrece comprobaciones en tiempo de ejecución que reducen conversiones erróneas; `const_cast` permite modificar cualificadores de const-correctness cuando es estrictamente necesario; y `reinterpret_cast` se reserva para conversiones de bajo nivel y potencialmente peligrosas, pero útiles en casos específicos donde la manipulación de bits o interfaces externas es inevitable.

En términos de seguridad, evitar casts inseguros contribuye a reducir vulnerabilidades relacionadas con interpretaciones incorrectas de memoria, accesos fuera de límites, corrupción de datos y errores de aliasing. Un uso indebido del cast estilo C puede conducir a que un puntero se trate como un tipo incompatible sin verificación, abriendo la puerta a fallos que pueden ser explotados por un atacante mediante técnicas como desbordamientos, confusión de tipos o acceso a memoria no válida. Con los casts prefijados se logra un entorno más seguro al forzar al compilador a verificar la validez de la conversión.

Es frecuente que los desarrolladores empleen cast estilo C por costumbre o por brevedad, pero esto oculta intenciones y vuelve más difícil la auditoría del código. Un cast explícito y semánticamente preciso facilita la revisión manual, las herramientas de análisis estático y la trazabilidad durante el ciclo de vida del software. Además, el código que usa casts prefijados es más portable y menos propenso a disparar comportamientos dependientes del compilador.

El uso de casts en punteros es especialmente delicado. Tomemos como ejemplo un puntero base que apunta a una clase derivada. Realizar un cast estilo C a la clase derivada puede conducir a un error silencioso si la relación de herencia no es válida. En cambio, `dynamic_cast` comprueba si la conversión es segura y devuelve `nullptr` en caso contrario, lo que permite al desarrollador añadir controles explícitos. Esta capacidad de comprobación resulta fundamental cuando se manipulan objetos polimórficos provenientes de fuentes externas o ingresados por el usuario.

Aunque `reinterpret_cast` es el mecanismo más peligroso entre los prefijados, también aporta claridad al indicar que se está realizando una conversión de bajo nivel. Su uso debe ser excepcional y estrictamente justificado. Un error común consiste en utilizarlo para reinterpretar punteros entre tipos no relacionados, lo que puede inducir aliasing ilegal o romper alineaciones de memoria. No obstante, para interacciones con APIs del sistema o conversión a tipos enteros de tamaño fijo, suele ser la opción correcta siempre que se documente y verifique el comportamiento en todas las plataformas objetivo.

Riesgos asociados

- Desbordamiento de buffer por conversión incorrecta de tipos de punteros.
- Pérdida de información o truncamiento de datos al convertir entre tipos incompatibles.
- Acceso a memoria no válida, provocando errores en tiempo de ejecución o vulnerabilidades de seguridad.
- Evasión de controles de tipo en tiempo de compilación, dificultando la detección de errores.
- Introducción de vulnerabilidades de corrupción de memoria (por ejemplo, use-after-free, double free).

- Comportamiento indefinido debido a conversiones inadecuadas entre tipos de datos.
- Permitir la manipulación insegura de objetos o estructuras, generando inconsistencias internas.
- Dificultad para mantener el código y aumentar la probabilidad de errores futuros.
- Mayor exposición a ataques de escalada de privilegios o ejecución de código arbitrario.
- Buenas prácticas
- Preferir `static_cast` en lugar de casts de estilo C para conversiones explícitas y controladas.

```
//-- Conversión explícita y verificable en tiempo de compilación
double valor = 10.5;
int resultado = (int)valor;
```

```
//-- Conversión explícita con intención clara
double valor2 = 10.5;
int resultado2 = static_cast<int>(valor2);
```

- Usar `dynamic_cast` para conversiones seguras en jerarquías polimórficas.

```
//-- Conversión sin verificación de tipo en jerarquía
Base* base = new Derivada();
Derivada* d = (Derivada*)base;
```

```
//-- Conversión con comprobación en tiempo de ejecución
Base* base2 = new Derivada();
Derivada* d2 = dynamic_cast<Derivada*>(base2);
if (d2 == nullptr) {
  /* manejo de error */
}
```

- Emplear `const_cast` únicamente cuando sea estrictamente necesario y seguro.

```
//-- Eliminación de const sin claridad de intención
void funcion(const int* p) {
  int* mod = (int*)p;
  *mod = 5;
}
```

```
//-- Eliminación explícita de const con intención documentada
```

```
void funcion2(const int* p) {
  int* mod = const_cast<int*>(p);
  *mod = 5;
}
```

- Evitar `reinterpret_cast` salvo en escenarios de muy bajo nivel claramente justificados.

```
//-- Conversión insegura entre tipos no relacionados
int numero = 65;
char* ptr = (char*)№
```

```
//-- Conversión explícita de bajo nivel con intención clara
int numero2 = 65;
char* ptr2 = reinterpret_cast<char*>(&numero2);
```

2.4.3.2 Estándares metodológicos

- **C++ Core-ES.48**: Avoid casts.
- **C++ Core-ES.49**: If you must use a cast, use a named cast.

2.4.4 [cpp.conv.4] - Evitar utilizar "reinterpret_cast"

ID	**cpp.conv.4**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Este control establece la necesidad de minimizar el uso de `reinterpret_cast`, ya que dicho operador permite transformar un tipo en otro sin validación de compatibilidad y puede introducir fallos sutiles de memoria, corrupción de datos, violaciones de aliasing y vulnerabilidades explotables. Su empleo debe sustituirse por alternativas más seguras como `static_cast`, `dynamic_cast` o tipos bien definidos. El objetivo es reforzar la **robustez** del código, reducir comportamientos indefinidos, garantizar que las conversiones sean explícitas y controladas y evitar violaciones de integridad y confidencialidad derivadas de accesos ilegítimos a la memoria.

Codificación insegura

Conversión forzada de tipos incompatibles usando `reinterpret_cast` para tratar un bloque de memoria como si fuera otro tipo distinto. Se pretende mostrar cómo una conversión insegura puede producir comportamiento indefinido y resultados inesperados al interpretar datos con un layout diferente.

```
#include <iostream>
```

```
//-- Estructura con dos enteros
struct Data {
  int a;
  int b;
};

int main() {
  //-- Variable de tipo double
  double value = 3.1415926535;

  //-- Conversión forzada e insegura de double* a Data*
  Data* dataPtr = reinterpret_cast<Data*>(&value);

  //-- Acceso a memoria interpretada como si fuera otra estructura
  std::cout << "a: " << dataPtr->a << std::endl;
  std::cout << "b: " << dataPtr->b << std::endl;

  //-- El programa compila pero el comportamiento es indefinido
  return 0;
}
```

Codificación recomendada

Uso de tipos compatibles y conversiones seguras evitando reinterpretaciones de memoria. Se muestra cómo definir estructuras adecuadas y trabajar con datos del mismo tipo para mantener coherencia en el layout de memoria y evitar comportamiento indefinido.

```
#include <iostream>

//-- Estructura con dos enteros
struct Data {
  int a;
  int b;
};

int main() {
  //-- Inicialización correcta de la estructura
  Data data;
  data.a = 10;
  data.b = 20;

  //-- Uso seguro del puntero del mismo tipo
  Data* dataPtr = &data;

  //-- Acceso válido a los miembros de la estructura
  std::cout << "a: " << dataPtr->a << std::endl;
```

```
    std::cout << "b: " << dataPtr->b << std::endl;

    //-- El comportamiento es definido y seguro
    return 0;
}
```

2.4.4.1 Descripción

El operador `reinterpret_cast` en C++ permite convertir entre tipos sin ninguna garantía de compatibilidad semántica o binaria. Esta capacidad puede resultar extremadamente peligrosa en entornos donde la memoria y la integridad del sistema son críticas, ya que habilita comportamientos indefinidos y accesos a memoria no válidos. Por ello, es fundamental comprender profundamente los riesgos asociados y aplicar patrones seguros que eliminen o reduzcan la necesidad de utilizarlo. Este requisito se enfoca en exponer los peligros, explicar alternativas seguras y definir normas técnicas estrictas para asegurar que el código C++ mantenga integridad, trazabilidad y protección frente a vulnerabilidades asociadas con conversiones inseguras.

El uso de `reinterpret_cast` implica un riesgo directo de corromper memoria, modificar representaciones internas de objetos o provocar violaciones de aliasing. Estas situaciones pueden ser explotadas por atacantes o generar fallos no deterministas difíciles de depurar. Entre los riesgos más relevantes destacan:

- **Corrupción de memoria:** La conversión forzada puede permitir interpretar regiones de memoria bajo un tipo incompatible.
- **Comportamiento indefinido:** La norma C++ no garantiza estabilidad si se fuerza una conversión binaria inválida.
- **Violación del strict aliasing:** Optimización del compilador que puede generar resultados imprevisibles.
- **Posibles fugas de información:** Interpretar bytes como estructuras puede exponer contenido sensible.
- **Acceso fuera de límites:** Conversión de punteros puede desalinear accesos o habilitar lecturas/escrituras peligrosas.

Debido a estos riesgos, se considera que **cualquier uso de `reinterpret_cast` debe ser exhaustivamente justificado, revisado y, en la mayoría de los sistemas, prohibido**. Es preferible utilizar construcciones del lenguaje más seguras y explícitas.

El lenguaje C++ proporciona múltiples operadores de conversión más seguros que permiten controlar la intención real de la operación. Es importante conocer estas alternativas para reemplazar adecuadamente el uso de `reinterpret_cast`:

- **static_cast:** Para conversiones conocidas y compatibles en tiempo de compilación.

- **dynamic_cast:** Para conversiones entre tipos polimórficos con verificación en tiempo de ejecución.
- **const_cast:** Para eliminar o agregar cualificadores de constancia de forma segura.
- **std::bit_cast:** Alternativa segura y definida desde C++20 para reinterpretaciones bit a bit.
- **Funciones específicas del dominio:** Creación de rutinas de conversión explícitas para asegurar validaciones previas.

Riesgos asociados

- Violación de los límites de tipo y ruptura de la seguridad de tipos, que puede llevar a comportamientos indefinidos.
- Lectura o escritura de memoria incorrecta, lo que puede provocar fugas de información o corrupción de memoria.
- Ejecución de código malicioso debido a la manipulación incorrecta de punteros o referencias.
- Incremento de la posibilidad de errores críticos como fallos de segmentación (segmentation faults) y accesos inválidos a memoria.
- Dificultad en el mantenimiento y depuración del código, aumentando el riesgo de introducir vulnerabilidades inadvertidas.
- Portabilidad reducida, pues el comportamiento de "reinterpret_cast" puede variar entre diferentes plataformas o compiladores.
- Incompatibilidad con técnicas de análisis estático y herramientas de seguridad que dependen de la validez de los tipos.

Buenas prácticas

- Preferir conversiones seguras y explícitas como `static_cast` cuando la conversión es válida y conocida en tiempo de compilación.

```
//-- Conversión forzada entre tipos no relacionados
int valor = 65;
char* ptr = reinterpret_cast<char*>(&valor);
```

```
//-- Conversión explícita y controlada entre tipos compatibles
int valor2 = 65;
char caracter = static_cast<char>(valor2);
```

- Utilizar polimorfismo y `dynamic_cast` en jerarquías de clases en lugar de forzar

conversiones entre punteros.

```
//-- Conversión forzada entre tipos de una jerarquía
class Base { public: virtual ~Base(){} };
class Derivada : public Base { };
Base* b = new Derivada();
Derivada* d = reinterpret_cast<Derivada*>(b);
```

```
//-- Conversión verificada en tiempo de ejecución
class Base2 { public: virtual ~Base2(){} };
class Derivada2 : public Base2 { };
Base2* b2 = new Derivada2();
Derivada2* d2 = dynamic_cast<Derivada2*>(b2);
```

- Evitar reinterpretar estructuras de memoria; utilizar funciones de serialización o `std::memcpy` para transformar datos binarios.

```
//-- Reinterpretación directa de memoria entre tipos distintos
struct A { int x; };
struct B { int y; };
A a{10};
B* b = reinterpret_cast<B*>(&a);
```

```
//-- Copia explícita de representación binaria
struct A2 { int x; };
struct B2 { int y; };
A2 a2{10};
B2 b2;
std::memcpy(&b2, &a2, sizeof(int));
```

- Usar contenedores tipados o `std::variant` en lugar de conversiones de punteros genéricos.

```
//-- Uso de puntero genérico con conversión forzada e incorrecta
void* dato;
double numero = 5.0;
dato = &numero;
int* p = reinterpret_cast<int*>(dato);
```

```
//-- Uso de variante tipada y acceso seguro
std::variant<int, double> dato2 = 5;
int numero2 = std::get<int>(dato2);
```

2.4.4.2 Estándares metodológicos

- **MISRA CPP-8.2.5**: reinterpret_cast shall not be used.

2.4.5 [cpp.conv.5] - Evitar eliminar "const" mediante "casts"

ID	cpp.conv.5	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este requisito establece que se debe evitar eliminar la cualificación `const` mediante cualquier tipo de conversión, en especial usando `const_cast`, ya que puede introducir riesgos críticos de seguridad y comportamiento indefinido. Forzar la eliminación de inmutabilidad permite modificar memoria que fue diseñada para no cambiarse, lo que puede derivar en corrupción de memoria, violaciones de integridad, fallos de concurrencia, vulnerabilidades de escalada lógica y errores difíciles de depurar. Mantener la inmutabilidad mediante el respeto a `const` constituye una medida esencial para garantizar estabilidad, integridad y claridad de intención del código.

Codificación insegura

Modificación de un objeto declarado como `const` mediante el uso de `const_cast`. Se busca mostrar cómo eliminar el modificador `const` permite compilar el código, pero genera comportamiento indefinido al alterar un objeto que fue definido como inmutable.

```
#include <iostream>

//-- Demuestra la eliminación de const mediante const_cast
int main() {
  //-- Variable declarada como constante
  const int valor = 10;

  //-- Se elimina const usando const_cast
  int* puntero = const_cast<int*>(&valor);

  //-- Modificación del valor original (comportamiento indefinido)
  *puntero = 20;

  //-- Se imprime el valor
  std::cout << "Valor: " << valor << std::endl;

  return 0;
}
```

Codificación recomendada

Uso correcto del modificador `const` respetando la inmutabilidad del objeto. Se muestra cómo diseñar la función para no modificar datos constantes y evitar el uso de `const_cast`, manteniendo la coherencia y seguridad del código.

```
#include <iostream>

//-- Función que solo lee el valor y no lo modifica
void imprimirValor(const int& valor) {
  //-- Se accede al valor sin modificarlo
  std::cout << "Valor: " << valor << std::endl;
}

int main() {
  //-- Variable constante correctamente protegida
  const int valor = 10;

  //-- Se pasa como referencia constante sin eliminar const
  imprimirValor(valor);

  return 0;
}
```

2.4.5.1 Descripción

La palabra clave `const` constituye uno de los mecanismos fundamentales para expresar inmutabilidad, limitar efectos laterales y permitir al compilador optimizar y detectar usos indebidos de memoria. Este requisito enfatiza que se debe evitar eliminar la cualificación `const` mediante operaciones de conversión explícita, en especial usando `const_cast`, reinterpretaciones forzadas o conversiones heredadas del estilo C. La eliminación arbitraria de `const` puede introducir vulnerabilidades de integridad, afectar la coherencia del estado interno de los objetos y permitir modificaciones no autorizadas de estructuras internas o buffers sensibles.

Eliminando `const` se rompe el contrato implícito entre el diseño del software y su implementación, permitiendo que código no autorizado modifique memoria que fue definida como inmutable por diseño. Esto puede generar vulnerabilidades graves, especialmente en entornos donde múltiples componentes dependen de la garantía de inmutabilidad (por ejemplo, datos compartidos en concurrencia, cadenas literales, estructuras accesibles desde múltiples módulos o bibliotecas externas). Además, puede llevar a comportamiento indefinido cuando se escribe en memoria originalmente declarada como sólo lectura.

La mayoría de los compiladores sitúan las cadenas literales y otros valores constantes en secciones de memoria de solo lectura. Intentar alterar estos valores mediante eliminación de `const` puede provocar errores difíciles de reproducir, como violaciones de segmento, corrupciones silenciosas o fallos aleatorios bajo carga. Esta situación es especialmente peligrosa en sistemas críticos o embebidos, donde los efectos pueden ser catastróficos.

En entornos donde una API antigua requiere punteros no-const, debe revisarse la interfaz y agregar sobrecargas modernas, evitando exigir al usuario eliminar `const`. Asimismo, cuando se interactúa con código C, es recomendable encapsular las llamadas en funciones puente que mantengan la inmutabilidad en el lado de C++.

Para garantizar integridad, seguridad y coherencia del software, los desarrolladores deben abstenerse de eliminar `const` mediante conversiones directas y deben revisar cualquier código que requiera tal cast, pues indica un defecto de diseño o una API mal definida.

Riesgos asociados

- Violación de la inmutabilidad de los datos, permitiendo modificaciones no intencionadas a objetos definidos como constantes.
- Introducción de errores sutiles y difíciles de detectar que pueden causar comportamientos impredecibles durante la ejecución del programa.
- Posibilidad de corromper la memoria al escribir en áreas que están protegidas contra escritura, potencialmente provocando fallos (crashes) o corrupción de datos.
- Compromiso de la seguridad del sistema, ya que ciertos mecanismos de protección pueden depender de la inmutabilidad de los objetos.
- Generación de condiciones de carrera si varios hilos asumen que los datos constantes no cambian durante su uso.
- Difícil mantenimiento del código al romper las expectativas sobre el uso seguro de las constantes y la semántica de los modificadores "const".
- Comportamiento indefinido según el estándar de C++ si se modifica un objeto originalmente declarado como "const", resultando en problemas de portabilidad y fiabilidad.

Buenas prácticas

- Respetar la inmutabilidad de objetos declarados como `const` y rediseñar la interfaz en lugar de usar `const_cast`.

```
//-- Se elimina const para modificar un objeto declarado como inmutable
void procesar(const int* dato) {
  int* mod = const_cast<int*>(dato);
  *mod = 10;
}
```

```
//-- Se ajusta la interfaz para recibir un objeto modificable cuando sea necesario
void procesar(int* dato) {
  *dato = 10;
```

```
}
```

- Definir correctamente métodos `const` en clases para evitar eliminar `const` al invocar funciones miembro.

```
class Usuario {
  std::string nombre;
public:
  //-- Método no declarado como const obliga a eliminar const del objeto
  std::string obtenerNombre() {
    return nombre;
  }
};

void imprimir(const Usuario& u) {
  Usuario& ref = const_cast<Usuario&>(u);
  ref.obtenerNombre();
}
```

```
class UsuarioSeguro {
  std::string nombre;
public:
  //-- Método declarado como const permite uso seguro sin casts
  std::string obtenerNombre() const {
    return nombre;
  }
};
```

- Evitar modificar datos originalmente definidos como `const`, especialmente si están en memoria de solo lectura.

```
//-- Se intenta modificar una variable global definida como const
const int LIMITE = 100;

void cambiar() {
  int* p = const_cast<int*>(&LIMITE);
  *p = 200;
}
```

```
//-- Se usa una variable no const cuando el valor debe cambiar en tiempo de
ejecución
int limite = 100;

void cambiar() {
  limite = 200;
}
```

2.4.5.2 Estándares metodológicos

- **C++ Core-ES.50**: don't cast away const.

2.4.6 [cpp.conv.6] - Evitar convertir un puntero a una clase incompleta ni liberarlo

ID	cpp.conv.6	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este control establece la necesidad de garantizar que los punteros no se conviertan ni se liberen cuando apuntan a tipos de clase incompleta, ya que esta práctica puede generar comportamientos indefinidos, corrupción de memoria y fallas en tiempo de ejecución. El requisito busca promover un uso seguro y adecuado de tipos completos en operaciones de liberación, conversión o manipulación de memoria, reforzando así la **integridad** y **estabilidad** del código, especialmente en entornos donde la gestión manual de memoria es crítica.

Codificación insegura

Gestión manual de memoria usando una clase adelantada (forward declaration) y liberando el objeto cuando el tipo aún no está completamente definido. Se busca mostrar los riesgos de eliminar un objeto cuando el compilador no conoce su definición completa, lo que puede provocar comportamiento indefinido o errores de compilación en proyectos reales con múltiples archivos.

```
#include <iostream>

//-- Declaración adelantada de la clase
class Recurso;

//-- Función que libera memoria usando un tipo incompleto
void liberar(Recurso* ptr) {
  //-- Eliminación del objeto sin conocer la definición completa
  delete ptr;
}

//-- Definición completa de la clase (ocurre después)
class Recurso {
public:
  Recurso() {
    //-- Constructor
    std::cout << "Recurso creado" << std::endl;
  }
  ~Recurso() {
    //-- Destructor
```

```
    std::cout << "Recurso destruido" << std::endl;
  }
};

int main() {
  //-- Reserva dinámica de memoria
  Recurso* r = new Recurso();
  //-- Liberación a través de función que usa tipo incompleto
  liberar(r);
  return 0;
}
```

Codificación recomendada

Gestión segura de memoria asegurando que el tipo esté completamente definido antes de realizar conversiones o liberaciones. Se muestra cómo definir primero la clase y después implementar cualquier función que elimine objetos de ese tipo, garantizando que el compilador conozca su tamaño y destructor.

```
#include <iostream>

//-- Definición completa de la clase antes de su uso
class Recurso {
public:
  Recurso() {
    //-- Constructor
    std::cout << "Recurso creado" << std::endl;
  }
  ~Recurso() {
    //-- Destructor
    std::cout << "Recurso destruido" << std::endl;
  }
};

//-- Función que libera memoria con el tipo ya completo
void liberar(Recurso* ptr) {
  //-- Eliminación segura porque el tipo es completo
  delete ptr;
}

int main() {
  //-- Reserva dinámica de memoria
  Recurso* r = new Recurso();
  //-- Liberación segura
  liberar(r);
  return 0;
}
```

2.4.6.1 Descripción

Un tipo de clase se considera incompleto cuando se conoce su declaración, pero no su definición. Esto suele ocurrir al usar declaraciones adelantadas (forward declarations). Aunque estas declaraciones facilitan la separación de dependencias y reducen el acoplamiento entre módulos, también pueden introducir riesgos cuando se realizan operaciones que requieren un conocimiento completo del tipo. **Operaciones como eliminar objetos mediante `delete`, realizar conversiones de tipo, evaluar el tamaño con `sizeof` o acceder a miembros requieren la definición completa del tipo**. Intentar efectuar dichas operaciones sobre un tipo incompleto puede provocar un comportamiento indefinido.

Este requisito enfatiza que, antes de liberar memoria o convertir punteros a un tipo específico, es obligatorio asegurarse de que la clase está completamente definida. Cuando se libera un puntero cuyo tipo es incompleto, el compilador no conoce el destructor asociado. Aunque algunos compiladores permiten esta operación, la norma C++ establece que el resultado es indefinido y puede provocar fugas de memoria, destructores no ejecutados o corrupción interna del heap.

Para mitigar estos riesgos, es fundamental diseñar de forma coherente la estructura de los archivos de cabecera y la organización del código. El uso adecuado de declaraciones adelantadas permite un acoplamiento débil, pero siempre debe ir acompañado de verificaciones claras para evitar operaciones no permitidas.

Cuando se desarrollan bibliotecas o módulos que exponen interfaces públicas, es habitual utilizar tipos incompletos para ocultar detalles internos. Sin embargo, el desarrollador debe garantizar que las operaciones peligrosas nunca estén expuestas sin la definición completa. **Un error frecuente es intentar liberar memoria en el cliente de la biblioteca cuando el destructor correcto solo está disponible internamente**. Esto compromete tanto el diseño como la seguridad del sistema.

Los punteros inteligentes ayudan a reducir errores, pero solo si el tipo es completo en el momento en que se destruye el objeto. Por ejemplo, `std::unique_ptr` con un destructor por defecto necesita que el tipo sea completo en el punto donde se evalúa el destructor. Si el destructor se encuentra en un archivo separado, esto puede generar fallos. Para estos casos, es usual implementar destructores personalizados que gestionen el tipo internamente.

Riesgos asociados

- Comportamiento indefinido al acceder o liberar memoria de un objeto cuya definición está incompleta.
- Fugas de memoria por liberar incorrectamente objetos no construidos totalmente.
- Corrupción de memoria y errores difíciles de depurar.

- Fallos de segmentación (segmentation faults) al tratar de acceder a miembros no definidos.
- Violación de contratos de la interfaz pública de la clase.
- Dificultad para mantener y auditar el código debido a dependencias circulares u ocultas.
- Posibilidad de llamar destructores incorrectos o no llamar destructores necesarios.
- Exposición a exploits por uso inadecuado de punteros en contextos inseguros.

Buenas prácticas

- Incluir la definición completa de la clase antes de realizar conversiones de punteros o liberar memoria asociada.

```
//-- Conversión y liberación con tipo incompleto
class Foo;

void liberar(void* p) {
  Foo* f = static_cast<Foo*>(p);
  delete f;
}
```

```
//-- Inclusión de la definición completa antes de usar delete
class Foo {
  public:
    ~Foo() {}
};

void liberarSeguro(void* p) {
  Foo* f = static_cast<Foo*>(p);
  delete f;
}
```

- Evitar eliminar objetos a través de punteros a tipos incompletos definidos solo por declaración adelantada.

```
//-- Eliminación con declaración adelantada sin definición visible
class Bar;

void destruir(Bar* ptr) {
  delete ptr;
}
```

```
//-- Definición completa incluida antes de la eliminación
```

```
class Bar {
  public:
    ~Bar() {}
};

void destruirSeguro(Bar* ptr) {
  delete ptr;
}
```

- Utilizar punteros inteligentes en el archivo donde la clase esté completamente definida para garantizar destrucción segura.

```
//-- Uso de unique_ptr con tipo incompleto en el mismo ámbito
class Baz;

std::unique_ptr<Baz> crear();

void usar() {
  auto ptr = crear();
}
```

```
//-- Definición completa antes de instanciar unique_ptr
class Baz {
  public:
    ~Baz() {}
};

std::unique_ptr<Baz> crearSeguro() {
  return std::unique_ptr<Baz>(new Baz());
}
```

2.4.6.2 Estándares metodológicos

- **CERT/CPP-EXP57**: Do not cast or delete pointers to incomplete classes.

2.4.7 [cpp.conv.7] - Utilizar std::variant para representar alternativas

ID	cpp.conv.7	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este control propone utilizar `std::variant` para representar alternativas de tipos de un modo seguro, explícito y verificable en tiempo de compilación. Su uso evita construcciones inseguras como uniones tradicionales o la gestión manual de banderas que indiquen qué tipo está activo. Además, reduce riesgos de corrupción de memoria,

accesos inválidos y comportamientos indefinidos. El requisito impulsa prácticas que fortalecen la robustez del código, hacen más claro el contrato de uso de las APIs y permiten aplicar comprobaciones exhaustivas mediante `std::visit`.

Codificación insegura

Gestión de diferentes tipos de datos (entero o texto) usando un mecanismo manual basado en unión y un indicador de tipo. Se pretende mostrar los riesgos de representar alternativas sin control de tipo fuerte, lo que puede provocar accesos incorrectos a memoria y comportamiento indefinido.

```
#include <iostream>
#include <cstring>

//-- Definición manual de tipos posibles
enum TipoDato { ENTERO, TEXTO };

//-- Unión sin control automático del tipo activo
union Valor {
  int numero;
  char texto[32];
};

int main() {
  //-- Se crea un valor y se indica que es ENTERO
  TipoDato tipo = ENTERO;
  Valor v;
  v.numero = 42;

  //-- Error lógico: se cambia el tipo sin actualizar correctamente la unión
  tipo = TEXTO;

  //-- Acceso inseguro: se interpreta memoria como texto cuando contiene un entero
  if (tipo == TEXTO) {
    std::cout << "Texto: " << v.texto << std::endl;
  }
  return 0;
}
```

Codificación recomendada

Representación segura de valores que pueden ser de distintos tipos (entero o texto) utilizando `std::variant`. Se busca mostrar cómo el tipado fuerte y las verificaciones en tiempo de compilación y ejecución reducen errores de interpretación de memoria y mejoran la mantenibilidad.

```
#include <iostream>
#include <variant>
```

```
#include <string>

//-- Definición de un tipo seguro que puede contener int o std::string
using Valor = std::variant<int, std::string>;

int main() {
  //-- Se asigna un entero al variant
  Valor v = 42;

  //-- Verificación segura del tipo activo
  if (std::holds_alternative<int>(v)) {
    std::cout << "Numero: " << std::get<int>(v) << std::endl;
  }

  //-- Cambio seguro a string
  v = std::string("Hola mundo");

  //-- Acceso seguro al nuevo tipo
  if (std::holds_alternative<std::string>(v)) {
    std::cout << "Texto: " << std::get<std::string>(v) << std::endl;
  }
  return 0;
}
```

2.4.7.1 Descripción

El uso de `std::variant` constituye una de las mejores prácticas actuales para modelar opciones mutuamente excluyentes de forma segura en C++. Este mecanismo sustituye enfoques inseguros como las uniones clásicas (`union`) sin control de tipo o estructuras que combinan un campo tipo y un `void*`. Tales soluciones tradicionales pueden exponer la aplicación a fallos graves, inconsistencias o comportamientos indefinidos al no contar con salvaguardas sobre el tipo activo. **El control de seguridad fomenta una representación clara, robusta y comprobable en tiempo de compilación**, que impida accesos no válidos o interpretaciones erróneas de datos.

La necesidad de un mecanismo seguro para modelar alternativas tipadas aparece en módulos que manejan flujos de datos heterogéneos, mensajes, resultados de operaciones, errores o eventos. Cuando existen varias posibilidades mutuamente excluyentes, usar `std::variant` permite evitar la sobrescritura accidental de memoria, mejorar la legibilidad, y apoyar al compilador a detectar usos incorrectos mediante chequeos exhaustivos. **Este requisito contribuye directamente a la integridad de la aplicación**, al prevenir efectos colaterales no controlados derivados de la manipulación insegura de memoria.

El empleo de `std::variant` debe integrarse desde fases tempranas de diseño. Esto implica que para cada punto del sistema donde exista un conjunto finito y bien definido de posibles alternativas, dichas alternativas se representen explícitamente en el tipo. La

idea principal es modelar el dominio con tipos robustos en vez de utilizar indicadores numéricos o banderas que requieran lógica manual para verificar cuál alternativa está activa. Asimismo, **la claridad del modelo de datos mejora la mantenibilidad, la revisabilidad y la capacidad de auditoría**, factores clave en la seguridad del software.

Acceder al contenido de un `std::variant` exige usar interfaces seguras como `std::get` (cuando se conoce inequívocamente el tipo), `std::get_if` (cuando se requiere comprobarlo dinámicamente), o bien `std::visit` para forzar una comprobación exhaustiva de las alternativas. **El uso de `std::visit` asegura que todo nuevo tipo añadido al `variant` deba ser tratado explícitamente**, evitando que casos no manejados queden ocultos y generen errores en tiempo de ejecución. Para operaciones sensibles o asociadas a datos críticos, usar `std::visit` aporta mayor fortaleza y transparencia.

Con uniones tradicionales, el programa puede acceder a un campo no activo y producir corrupción silenciosa de memoria. Este tipo de problema resulta difícil de detectar, especialmente en sistemas complejos o embebidos. Por el contrario, `std::variant` impide accesos peligrosos: si se intenta extraer un tipo que no está activo, se lanza una excepción `std::bad_variant_access` o se obtiene un puntero nulo mediante `std::get_if`. Esta protección evita estados inconsistentes que podrían conducir a fallos más graves. **Controlar de manera rigurosa las operaciones ilegales protege contra vulnerabilidades de integridad** y reduce la superficie de ataque asociada a la manipulación insegura de memoria.

El uso de `std::variant` facilita la redacción de pruebas automatizadas. Al estar el conjunto de alternativas definido de manera estructurada, es posible crear baterías de casos que cubran cada rama. Además, el compilador fuerza a quienes desarrollan el sistema a implementar todas las alternativas si se utiliza `std::visit` con lambdas genéricas o visitantes especializados. En pruebas de seguridad, esta estructura evita omisiones en rutas de ejecución y ayuda a verificar qué ocurre en condiciones límite, entradas inesperadas o transiciones entre estados.

En arquitecturas modernas basadas en microservicios, sistemas distribuidos o módulos con comunicación interproceso, los datos suelen viajar en formatos estructurados. Representar estados con `std::variant` facilita mapearlos a formatos externos como JSON o XML sin comprometer la integridad interna. **Este enfoque refuerza la separación entre componentes y reduce el riesgo de que los datos se interpreten como tipos incorrectos, evitando inyecciones semánticas**. También permite validar entradas antes de deserializarlas a un `variant`, reduciendo riesgos de fallos provocados por usuarios maliciosos.

Riesgos asociados

- Desbordamientos de buffer por manejo manual incorrecto de uniones (union) o conversiones de punteros.
- Lectura de memoria no válida al acceder a un miembro incorrecto de una unión.

- Errores de tipo (type confusion) que pueden derivar en comportamientos indefinidos o vulnerabilidades de seguridad.
- Fugas de memoria por no liberar correctamente recursos en uniones que contienen tipos no triviales.
- Inyección de errores lógicos al asumir incorrectamente el tipo activo almacenado.
- Menor robustez ante excepciones, provocando posibles caídas del sistema o corrupción de datos.
- Dificultad para asegurar la inicialización y destrucción apropiada de los objetos almacenados.
- Sobrecarga para el mantenimiento de código debido a la ausencia de comprobaciones en tiempo de compilación.
- Exposición a vulnerabilidades de seguridad por interpretación incorrecta de los datos almacenados.

Buenas prácticas

- Preferir `std::variant` en lugar de `union` para representar tipos alternativos con seguridad de tipo.

```
//-- Uso de union sin control de tipo activo
union Dato {
  int entero;
  double decimal;
};
```

```
//-- Uso de std::variant con verificación segura del tipo
std::variant dato;
dato = 10;
if (std::holds_alternative(dato)) {
  int valor = std::get(dato);
}
```

- Acceder al contenido de `std::variant` mediante `std::visit` para evitar excepciones por acceso incorrecto.

```
//-- Acceso directo que puede lanzar excepción si el tipo no coincide
std::variant valor = "texto";
int numero = std::get(valor);
```

```
//-- Acceso controlado usando std::visit
std::variant valor2 = "texto";
```

```
std::visit([](auto&& arg) {
  using T = std::decay_t;
  if constexpr (std::is_same_v) {
    int numero = arg;
  }
}, valor2);
```

- Definir claramente todas las alternativas necesarias en `std::variant` para evitar estados inválidos o uso de punteros inseguros.

```
//-- Uso de puntero base que puede quedar en estado nulo o inconsistente
struct Base { virtual ~Base() = default; };
struct A : Base {};
struct B : Base {};
Base* ptr = nullptr;
```

```
//-- Uso de std::variant para representar alternativas explícitas
struct A2 {};
struct B2 {};
std::variant objeto;
objeto = A2{};
```

2.4.7.2 Estándares metodológicos

- **C++ Core-C.183**: don't use a union for type punning.

2.5 Inicialización

2.5.1 [cpp.ini.1] - Inicializar todos los objetos antes de su uso

ID	cpp.ini.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este requisito establece que cualquier objeto, variable o estructura de datos debe quedar correctamente inicializado antes de su primer uso, evitando comportamientos indefinidos que pueden derivar en vulnerabilidades críticas. Una inicialización segura previene accesos a memoria no válida, fugas de información, estados inconsistentes y fallos de seguridad explotables. Aplicar este principio implica el uso sistemático de constructores explícitos, valores por defecto bien definidos y verificaciones de estado cuando sea necesario, garantizando operaciones confiables incluso en entornos con memoria manual o semiautomática.

Codificación insegura

Uso de variables y objetos sin inicializar para realizar cálculos. Se busca mostrar cómo un valor indeterminado puede propagarse por el programa generando resultados impredecibles y comportamiento indefinido.

```
#include <iostream>

class Calculadora {
public:
  int resultado;

  void sumar(int a) {
    resultado += a;
  }
};

int main() {
  //-- Se crea el objeto pero no se inicializa el atributo resultado
  Calculadora calc;
  //-- Se usa resultado sin haber sido inicializado
  calc.sumar(10);
  //-- Se imprime un valor indeterminado (comportamiento indefinido)
  std::cout << "Resultado: " << calc.resultado << std::endl;
  return 0;
}
```

Codificación recomendada

Inicialización explícita de todos los atributos en el momento de la construcción del objeto. Se pretende garantizar que el estado interno sea válido antes de cualquier uso y evitar comportamiento indefinido.

```
#include <iostream>

class Calculadora {
public:
  int resultado;

  //-- Constructor que garantiza la inicialización del atributo
  Calculadora() : resultado(0) {
  }

  void sumar(int a) {
    resultado += a;
  }
};

int main() {
```

```
    //-- El objeto se crea con estado interno válido
    Calculadora calc;
    //-- Se usa el objeto después de estar correctamente inicializado
    calc.sumar(10);
    //-- Se imprime un valor predecible y seguro
    std::cout << "Resultado: " << calc.resultado << std::endl;
    return 0;
}
```

2.5.1.1 Descripción

C++ ofrece una gran flexibilidad al basarse en un modelo de memoria complejo con diferentes tipos de almacenamiento, reglas de vida de los objetos, inicialización condicional y posibles combinaciones de constructores implícitos o explícitos, pero también múltiples oportunidades para cometer errores sutiles. Por ello, los desarrolladores deben aplicar un conjunto de prácticas rigurosas orientadas a garantizar que todos los objetos posean un estado válido antes de cualquier operación.

La problemática surge especialmente en escenarios donde se permite que variables queden sin inicializar, como ocurre con variables locales de tipo primitivo, punteros crudos y estructuras definidas por el usuario que carecen de constructores adecuados. En estos casos, el contenido inicial de la memoria es indeterminado y puede contener valores residuales, lo que habilita vulnerabilidades como corrupción de memoria, lecturas fuera de rango, uso de punteros no válidos o incluso la filtración de datos sensibles. **La inicialización explícita reduce drásticamente este riesgo** al establecer estados controlados y coherentes desde el momento de la creación del objeto.

Para aplicar correctamente esta práctica, resulta esencial comprender las diferentes formas de inicialización disponibles: inicialización por defecto, inicialización directa, inicialización uniforme mediante llaves, inicialización de lista, inicialización estática y asignación posterior. Cada una de estas variantes implica matices semánticos importantes relacionados con la llamada a constructores, la preparación de memoria y la vida del objeto. **Se recomienda la inicialización uniforme con llaves** debido a su claridad y menor propensión a generar ambigüedad en la selección de constructores.

Es fundamental destacar que, en tipos definidos por el usuario, el uso de inicializadores miembro mejora significativamente la fiabilidad. Mediante esta técnica, todos los atributos quedan inicializados antes de la ejecución del constructor, reduciendo la posibilidad de olvidar inicializar alguno de ellos. Los punteros internos deben inicializarse a `nullptr` y los contenedores deben empezar en un estado vacío predecible. Esto elimina inconsistencias internas y evita fallos lógicos posteriores.

Una de las estrategias más eficaces para garantizar una inicialización segura en C++ moderno es el uso del patrón RAII junto con punteros inteligentes como `std::unique_ptr` y `std::shared_ptr`. Estos mecanismos aseguran la adquisición correcta de recursos durante la construcción del objeto y liberación automática al final de su vida,

evitando fugas de memoria, referencias colgantes y comportamiento indefinido. **La combinación de RAII con reglas de inicialización estrictas reduce notablemente el riesgo de fallos de seguridad**.

En entornos concurrentes, la inicialización segura tiene un papel especialmente crítico. Los objetos compartidos entre hilos deben estar completamente inicializados antes de ser publicados para evitar condiciones de carrera y estados intermedios inconsistentes. Es recomendable utilizar mecanismos de sincronización como `std::mutex`, inicializadores estáticos garantizados por el estándar o patrones de diseño como double-checked locking correctamente implementado.

Riesgos asociados

- Acceso a memoria no inicializada, lo que puede provocar resultados indefinidos o comportamientos erráticos.
- Fugas de información sensible, ya que la memoria no inicializada puede contener datos antiguos de procesos previos.
- Condiciones de carrera difíciles de detectar en programas multihilo.
- Exposición a vulnerabilidades de seguridad como ejecución de código arbitrario al manipular memoria no controlada.
- Errores de lógica y fallos de programa al depender de valores basura.
- Dificultad para depurar e identificar la causa raíz de errores, ya que el comportamiento es no determinista.
- Aumento de la superficie de ataque para exploits como buffer overflows y ataques de tipo use-after-free.

Buenas prácticas

- Inicializar variables primitivas en el momento de su declaración para evitar valores indeterminados.

```
//-- variable declarada sin inicializar
int contador;
contador++;
```

```
//-- variable inicializada en la declaración
int contadorSeguro = 0;
contadorSeguro++;
```

- Utilizar listas de inicialización en constructores para garantizar la correcta inicialización de atributos.

```
//-- atributo no inicializado en el constructor
```

```
class Usuario {
  int id;
public:
  Usuario() { }
};
```

```
//-- atributo inicializado mediante lista de inicialización
class UsuarioSeguro {
  int id;
public:
  UsuarioSeguro() : id(0) { }
};
```

- Inicializar punteros y objetos dinámicos antes de su uso para prevenir accesos a memoria inválida.

```
//-- puntero utilizado sin inicializar
int* ptr;
*ptr = 10;
```

```
//-- puntero inicializado antes de su uso
int valor = 0;
int* ptrSeguro = &valor;
*ptrSeguro = 10;
```

- Inicializar contenedores y objetos complejos antes de acceder a sus elementos.

```
//-- acceso a posición sin garantizar inicialización
std::vector datos;
int primero = datos[0];
```

```
//-- contenedor inicializado con valores antes del acceso
std::vector datosSeguro(1, 0);
int primeroSeguro = datosSeguro[0];
```

2.5.1.2 Estándares metodológicos

- **C++ Core-ES.20**: Always initialize an object.

2.5.2 [cpp.ini.2] - Evitar declarar o definir identificadores reservados

ID	cpp.ini.2	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Design

Este control establece la necesidad de impedir que desarrolladores utilicen identificadores reservados por el estándar de C++, ya sean palabras clave, nombres con prefijos reservados, o símbolos utilizados por la implementación del compilador y la biblioteca estándar. El objetivo es evitar colisiones, comportamientos indefinidos y vulnerabilidades que puedan surgir al redefinir entidades cuya semántica está controlada por el lenguaje o el entorno de ejecución. Una gestión adecuada de identificadores garantiza mayor previsibilidad del código, reduce riesgos de incoherencias y mejora la seguridad, confiabilidad y mantenibilidad.

Codificación insegura

Uso de identificadores que comienzan con doble guion bajo o con guion bajo seguido de mayúscula, los cuales están reservados para la implementación del compilador y la biblioteca estándar. Se pretende mostrar cómo una mala elección de nombres puede generar comportamiento indefinido o conflictos con el entorno de compilación.

```
#include <iostream>

int __contadorGlobal = 0;

int main() {
  //-- Se declara un identificador con doble guion bajo, reservado por el estándar
  int __Valor = 10;
  //-- Se utiliza el identificador reservado en una operación simple
  __contadorGlobal = __Valor + 5;
  //-- Se muestra el resultado por consola
  std::cout << "Resultado: " << __contadorGlobal << std::endl;
  return 0;
}
```

Codificación recomendada

Uso de identificadores definidos por el desarrollador que no invaden el espacio de nombres reservado por el estándar. Se busca demostrar cómo nombrar variables de forma segura y portable, evitando prefijos reservados y posibles conflictos con el compilador o la biblioteca estándar.

```
#include <iostream>

int contadorGlobal = 0;
```

```
int main() {
  //-- Se declara un identificador válido que no usa prefijos reservados
  int valor = 10;
  //-- Se realiza una operación simple con identificadores seguros
  contadorGlobal = valor + 5;
  //-- Se muestra el resultado por consola
  std::cout << "Resultado: " << contadorGlobal << std::endl;
  return 0;
}
```

2.5.2.1 Descripción

El uso de identificadores reservados en C++ constituye una fuente frecuente de errores complejos y comportamientos indefinidos. El estándar del lenguaje define un conjunto extenso de palabras clave, convenciones de nomenclatura, y patrones de símbolos cuyo empleo por parte del programador está prohibido o estrictamente limitado. Este requisito de seguridad propone una guía exhaustiva para comprender qué tipos de identificadores deben evitarse, por qué su utilización puede comprometer la integridad del software y cómo aplicar prácticas sistemáticas de verificación para garantizar que ningún identificador reservado se introduzca accidentalmente en el código fuente.

En primer lugar, el lenguaje define como reservadas todas sus palabras clave, incluyendo las tradicionales (`int`, `class`, `virtual`, `public`, etc.) y las incorporadas en versiones modernas del estándar (como `constexpr`, `concept`, `co_await`, entre muchas otras). El desarrollador nunca debe utilizar estas palabras para declarar variables, funciones, clases, espacios de nombres u otros identificadores. Además, el estándar reserva ciertos patrones, como cualquier identificador que comience con un guion bajo seguido de una letra mayúscula, o cualquier identificador que empiece con dos guiones bajos. Estos patrones son utilizados por implementaciones del compilador y la biblioteca estándar, y su uso por aplicaciones puede generar conflictos internos que se manifiestan de formas impredecibles.

Un aspecto crítico es que el uso de identificadores reservados puede no generar errores de compilación inmediatos. En muchos casos, el compilador permite la compilación, pero produce un comportamiento indefinido, lo que abre la puerta a fallos lógicos, corrupción de memoria o situaciones donde la ejecución depende de cómo el compilador gestione internamente ciertos símbolos. Por esta razón, los problemas derivados de identificadores reservados pueden tardar en ser detectados y manifestarse solo bajo condiciones específicas, convirtiéndose en una amenaza directa para la integridad y confiabilidad del software.

Entre las categorías comunes de identificadores reservados que deben evitarse se encuentran las siguientes:

- **Palabras clave del estándar:** Uso prohibido en cualquier contexto que defina

nuevos identificadores.

- **Identificadores con prefijos de guion bajo:** Cualquier nombre que empiece por `_A` o por `__` está reservado para el compilador.
- **Nombres pertenecientes a la biblioteca estándar:** Identificadores declarados en los encabezados estándar o en espacios como `std` o `std::experimental` no pueden redefinirse.
- **Identificadores con significado especial para el preprocesador:** Macros como `__LINE__`, `__FILE__` o `__cplusplus` son exclusivos del compilador.

El impacto de violar estas reglas se amplifica en proyectos de gran escala, donde diversas bibliotecas, herramientas de análisis y versiones del compilador interactúan entre sí. Un identificador reservado puede causar incompatibilidades entre plataformas o introducir vulnerabilidades que emergen cuando el proceso de optimización del compilador altera estructuras internas esperando que los nombres reservados respeten sus roles originales. Para evitar estos problemas, es esencial adoptar prácticas sistemáticas de validación.

Una técnica fundamental consiste en integrar herramientas automáticas de análisis estático que detecten violaciones de nomenclatura reservada. Herramientas como clang-tidy ofrecen reglas específicas que inspeccionan el código en busca de patrones prohibidos. Asimismo, es buena práctica revisar periódicamente la lista oficial de palabras clave del estándar ISO C++ y las convenciones documentadas por cada compilador, como GCC, Clang o MSVC, ya que pueden existir identificadores adicionales reservados para extensiones particulares.

Los equipos de desarrollo deben establecer una política interna de definición de nombres que incorpore convenciones de prefijos, sufijos o esquemas de nombres que eviten colisiones con la sintaxis reservada del estándar. Además, es recomendable incluir controles automáticos en sistemas de integración continua que intercepten commits que introduzcan identificadores prohibidos. Adoptar estas prácticas reduce riesgos y garantiza que el código mantenga un comportamiento consistente bajo diferentes compiladores y niveles de optimización.

Riesgos asociados

- Conflictos con las implementaciones estándar del compilador, lo que puede provocar resultados impredecibles.
- Comportamiento indefinido o inesperado en tiempo de compilación y/o ejecución.
- Reducción de la portabilidad del código al depender de implementaciones específicas del compilador.
- Dificultad en el diagnóstico y depuración de errores debido a la ambigüedad

causada por identificadores reservados.

- Fallas de compilación en futuras versiones del compilador o al migrar el código a otros entornos.
- Potencial para introducción de vulnerabilidades de seguridad debido a confusiones en la vinculación de símbolos.

Buenas prácticas

- No declarar identificadores que comiencen con doble guion bajo o guion bajo seguido de mayúscula.

```
//-- Uso de identificador reservado por el estándar
int __ContadorGlobal = 0;
```

```
//-- Uso de identificador válido definido por el desarrollador
int contadorGlobal = 0;
```

- Evitar definir macros con nombres que comiencen con guion bajo.

```
//-- Macro con nombre reservado que puede colisionar con la implementación
#define _MAX_BUFFER 1024
```

```
//-- Macro con nombre descriptivo no reservado
#define MAX_BUFFER 1024
```

- No declarar funciones con nombres que coincidan con identificadores reservados del compilador o la biblioteca estándar.

```
//-- Declaración de función con identificador reservado
void _InitSystem();
```

```
//-- Declaración de función con identificador seguro y descriptivo
void initSystem();
```

- Evitar el uso de nombres que comiencen con guion bajo en el espacio de nombres global.

```
//-- Variable global con prefijo reservado
int _configValue = 5;
```

```
//-- Variable global con nombre válido y claro
int configValue = 5;
```

- No redefinir identificadores pertenecientes a la biblioteca estándar.

```
//-- Redefinición de identificador perteneciente al estándar
int std = 10;
```

```
//-- Uso adecuado respetando el espacio de nombres estándar
int valor = 10;
```

2.5.2.2 Estándares metodológicos

- **CERT/CPP-DCL51**: Do not declare or define a reserved identifier.

2.5.3 [cpp.ini.3] - Asegurar que las operaciones de movimiento dejan el objeto origen en un estado válido

ID	cpp.ini.3	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este requisito establece que, al implementar operaciones de movimiento, es imprescindible **preservar un estado coherente, seguro y válido en el objeto fuente tras la operación**. Esto evita usos indebidos posteriores, fallos de memoria, comportamientos indefinidos y vulnerabilidades asociadas a accesos a recursos liberados o estados inconsistentes. El control exige que todo objeto movido permanezca en un estado definido y seguro, aun cuando ya no mantenga sus recursos originales. Cuando **el objeto gestione información sensible**, la operación de movimiento debería, siempre que sea técnica y funcionalmente viable, **eliminar o sobrescribir de forma segura los datos residuales** que permanezcan en el objeto origen, reduciendo el riesgo de exposición accidental de credenciales, claves, tokens u otra información confidencial. Esto contribuye a la robustez, integridad y confiabilidad del software en entornos de ejecución complejos.

Codificación insegura

Gestión de recursos dinámicos en una clase que implementa constructor y operador de movimiento. Se muestra cómo transferir la propiedad de un puntero dinámico entre objetos y qué ocurre cuando el objeto origen no se deja en un estado válido tras el movimiento.

```
#include <iostream>
#include <utility>

class Buffer {
  int* data;
```

```
  size_t size;

public:
  //-- Constructor que reserva memoria
  Buffer(size_t s) : size(s) {
    data = new int[size];
  }

  //-- Destructor que libera memoria
  ~Buffer() {
    delete[] data;
  }

  //-- Constructor de movimiento que transfiere el puntero sin invalidar el origen
  Buffer(Buffer&& other) noexcept {
    data = other.data;
    size = other.size;
    //-- ERROR: no se deja el objeto origen en estado válido
  }

  //-- Operador de movimiento que tampoco invalida el origen
  Buffer& operator=(Buffer&& other) noexcept {
    if (this != &other) {
      delete[] data;
      data = other.data;
      size = other.size;
      //-- ERROR: other sigue apuntando a la misma memoria
    }
    return *this;
  }

  void print() const {
    std::cout << "Size: " << size << std::endl;
  }
};

int main() {
  //-- Crear objeto original
  Buffer b1(10);
  //-- Mover b1 a b2
  Buffer b2(std::move(b1));
  b2.print();
  //-- Al finalizar, ambos destructores intentarán liberar la misma memoria (double
free)
  return 0;
}
```

Codificación recomendada

Gestión segura de recursos dinámicos implementando correctamente el constructor y el operador de movimiento, asegurando que el objeto origen quede en un estado válido y seguro tras la transferencia de recursos.

```
#include <iostream>
#include <utility>

class Buffer {
  int* data;
  size_t size;

public:
  //-- Constructor que reserva memoria
  Buffer(size_t s) : data(new int[s]), size(s) {
  }

  //-- Destructor que libera memoria si existe
  ~Buffer() {
    delete[] data;
  }

  //-- Constructor de movimiento que deja el origen en estado válido
  Buffer(Buffer&& other) noexcept : data(other.data), size(other.size) {
    other.data = nullptr;
    other.size = 0;
  }

  //-- Operador de movimiento seguro
  Buffer& operator=(Buffer&& other) noexcept {
    if (this != &other) {
      delete[] data;
      data = other.data;
      size = other.size;
      other.data = nullptr;
      other.size = 0;
    }
    return *this;
  }

  void print() const {
    std::cout << "Size: " << size << std::endl;
  }
};

int main() {
  //-- Crear objeto original
  Buffer b1(10);
  //-- Mover b1 a b2
```

```
    Buffer b2(std::move(b1));
    b2.print();
    //-- b1 queda en estado válido y su destructor es seguro
    return 0;
}
```

2.5.3.1 Descripción

Las operaciones de movimiento permiten transferir recursos desde un objeto origen hacia un objeto destino de manera más eficiente que las copias tradicionales, eliminando duplicidades innecesarias y optimizando el rendimiento. Sin embargo, estas mismas optimizaciones introducen riesgos importantes de seguridad si no se aplican controles adecuados. La implementación incorrecta de constructores de movimiento y operadores de asignación por movimiento puede generar estados inválidos, punteros colgantes, dobles liberaciones de memoria y uso de recursos no inicializados. Esta regla establece que **el objeto fuente debe quedar siempre en un estado válido, seguro y predecible después de cualquier operación de movimiento**.

Dentro del ecosistema moderno de C++, un estado válido para un objeto movido no implica necesariamente que mantenga su información original, sino que su representación interna esté consistente y no provoque errores al usarse en operaciones posteriores. Esto significa que cualquier recurso que haya sido transferido al objeto destino debe eliminarse adecuadamente del objeto fuente o dejarse en un estado neutral que garantice integridad y seguridad. **Cuando el objeto gestione información sensible**, como claves criptográficas, contraseñas, tokens o credenciales, **la operación de movimiento de eliminar o sobrescribir de forma segura cualquier dato residual que permanezca en el objeto origen** (ver **cpp.mem.4**), siempre que ello sea técnica y funcionalmente viable, reduciendo así el riesgo de exposición accidental de información confidencial. El estándar recomienda establecer el objeto origen en un estado que cumpla invariantes esenciales y **permita su destrucción, reasignación o reutilización sin comportamientos indefinidos**.

Cuando un objeto movido queda en un estado inconsistente, el software queda expuesto a fallos críticos. Entre los problemas más frecuentes se encuentran accesos a memoria ya liberada, manipulación de punteros colgantes, corrupción de estructuras internas y violaciones de integridad de datos. Estos problemas no solo pueden provocar fallos de ejecución, sino también abrir la puerta a ataques por corrupción de memoria cuando el entorno permite explotación de errores lógicos. Mantener un estado válido en el objeto fuente reduce la superficie de ataque asociada al mal uso del objeto tras el movimiento.

En entornos donde la seguridad es prioritaria, como servicios de red, componentes criptográficos o sistemas empotrados, es recomendable aplicar controles adicionales tales como validación explícita de estados internos tras el movimiento, auditoría de invariantes críticos y aplicación de políticas estrictas que prohíban estados parcialmente

movidos no documentados. Asimismo, cuando se manipulen datos sensibles, conviene emplear mecanismos de borrado seguro de memoria en el objeto origen, utilizando funciones o bibliotecas que impidan que el compilador elimine dichas operaciones durante la optimización. Las bibliotecas seguras deben especificar claramente sus garantías de movimiento y realizar pruebas exhaustivas para evitar inconsistencias internas.

Riesgos asociados

- Acceso a datos no válidos tras el movimiento del objeto origen (uso de objetos en estado indefinido).
- Fugas de memoria debido a no liberar correctamente los recursos del objeto origen.
- Comportamiento indefinido al llamar métodos sobre el objeto origen después de la operación de movimiento.
- Condiciones de carrera en aplicaciones multihilo si el objeto origen queda en un estado inconsistente.
- Errores en la destrucción de objetos, como doble liberación de recursos (double free).
- Violación de invariantes de clase al no dejar el objeto origen en un estado seguro tras el movimiento.
- Exposición de información sensible si los recursos del objeto origen no se limpian correctamente (ver **cpp.mem.4**).
- Difícil rastreo y depuración debido a fallos intermitentes relacionados con el estado inválido de objetos movidos.

Buenas prácticas

- Restablecer explícitamente los recursos del objeto origen después del movimiento.

```
//-- Movimiento que no reinicia el puntero origen
class Buffer {
  char* data;
  size_t size;
  Buffer(Buffer&& other) noexcept {
    data = other.data;
    size = other.size;
  }
};
```

```
//-- Movimiento que deja el objeto origen en estado válido
```

```
class Buffer {
  char* data;
  size_t size;
  Buffer(Buffer&& other) noexcept {
    data = other.data;
    size = other.size;
    other.data = nullptr;
    other.size = 0;
  }
};
```

- Garantizar que el objeto movido pueda destruirse o reutilizarse sin provocar comportamiento indefinido.

```
//-- Destructor puede liberar dos veces si no se limpia el origen
class File {
  FILE* handle;
  File(File&& other) noexcept : handle(other.handle) {}
  ~File() { if(handle) fclose(handle); }
};
```

```
//-- Se asegura que el origen no conserve el recurso
class File {
  FILE* handle;
  File(File&& other) noexcept : handle(other.handle) {
    other.handle = nullptr;
  }
  ~File() { if(handle) fclose(handle); }
};
```

- Mantener invariantes internos coherentes tras la operación de movimiento.

```
//-- Invariante inconsistente tras mover contenedor
class Container {
  std::vector data;
  size_t count;
  Container(Container&& other) noexcept : data(std::move(other.data)),
count(other.count) {}
};
```

```
//-- Se sincroniza el estado interno del objeto origen
class Container {
  std::vector data;
  size_t count;
  Container(Container&& other) noexcept : data(std::move(other.data)),
count(other.count) {
    other.count = 0;
```

```
  }
};
```

2.5.3.2 Estándares metodológicos

- **C++ Core-C.64**: A move operation should move and leave its source in a valid state.
- **CERT/CPP-EXP63**: Do not rely on the value of a moved-from object.

2.5.4 [cpp.ini.4] - Asegurar que las operaciones de movimiento deben ser "noexcept" siempre que sea posible

ID	cpp.ini.4	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las operaciones de movimiento deben marcarse como `noexcept` siempre que sea seguro hacerlo, con el fin de reforzar el comportamiento predecible, mejorar la eficiencia y evitar rutas de ejecución inseguras asociadas a excepciones inesperadas. Al declarar que **estas operaciones no pueden lanzar excepciones**, se facilita la optimización por parte del compilador, se reduce el riesgo de corrupción del estado interno y se fortalecen las garantías de seguridad y robustez del código.

Codificación insegura

Gestión de recursos dinámicos en una clase que implementa constructor y operador de movimiento. Se busca mostrar cómo la ausencia de `noexcept` en las operaciones de movimiento puede afectar al comportamiento de contenedores estándar y a la seguridad ante excepciones durante realocaciones.

```
#include <iostream>
#include <vector>

class Buffer {
  //-- Puntero a recurso dinámico
  int* data;
  size_t size;

public:
  //-- Constructor que reserva memoria
  Buffer(size_t s) : data(new int[s]), size(s) {
  }

  //-- Destructor que libera el recurso
  ~Buffer() {
```

```
    delete[] data;
  }

  //-- Constructor de movimiento sin noexcept
  Buffer(Buffer&& other) : data(other.data), size(other.size) {
    other.data = nullptr;
    other.size = 0;
  }

  //-- Operador de movimiento sin noexcept
  Buffer& operator=(Buffer&& other) {
    if (this != &other) {
      delete[] data;
      data = other.data;
      size = other.size;
      other.data = nullptr;
      other.size = 0;
    }
    return *this;
  }
};

int main() {
  //-- Uso en un contenedor estándar
  std::vector<Buffer> v;
  v.reserve(2);
  v.emplace_back(10);
  v.emplace_back(20);
  std::cout << "Buffers creados" << std::endl;
  return 0;
}
```

Codificación recomendada

Gestión de recursos dinámicos en una clase que implementa correctamente constructor y operador de movimiento declarados como `noexcept`. Se pretende mostrar cómo garantizar que las operaciones de movimiento no lancen excepciones mejora la seguridad y permite a los contenedores estándar utilizar movimientos en lugar de copias.

```
#include <iostream>
#include <vector>

class Buffer {
  //-- Puntero a recurso dinámico
  int* data;
  size_t size;
```

```
public:
  //-- Constructor que reserva memoria
  Buffer(size_t s) : data(new int[s]), size(s) {
  }

  //-- Destructor que libera el recurso
  ~Buffer() {
    delete[] data;
  }

  //-- Constructor de movimiento declarado noexcept
  Buffer(Buffer&& other) noexcept : data(other.data), size(other.size) {
    other.data = nullptr;
    other.size = 0;
  }

  //-- Operador de movimiento declarado noexcept
  Buffer& operator=(Buffer&& other) noexcept {
    if (this != &other) {
      delete[] data;
      data = other.data;
      size = other.size;
      other.data = nullptr;
      other.size = 0;
    }
    return *this;
  }
};

int main() {
  //-- Uso en un contenedor estándar que aprovechará el movimiento seguro
  std::vector<Buffer> v;
  v.reserve(2);
  v.emplace_back(10);
  v.emplace_back(20);
  std::cout << "Buffers creados con movimiento noexcept" << std::endl;
  return 0;
}
```

2.5.4.1 Descripción

Las operaciones de movimiento representan un mecanismo central para optimizar eficiencia y minimizar copias innecesarias, pero también influyen de forma crítica en la robustez y seguridad del software. Marcar estas operaciones como `noexcept` siempre que sea posible constituye una medida de protección esencial, ya que evita que el flujo de ejecución se vea desviado por excepciones inesperadas, lo que podría dejar objetos en estados inconsistentes o provocar comportamientos indefinidos. Esta práctica está en consonancia con el principio de minimizar las operaciones susceptibles de lanzar

excepciones durante fases críticas como construcciones, asignaciones o realocaciones internas de contenedores.

El uso de `noexcept` en las operaciones de movimiento permite a la biblioteca estándar y al compilador aplicar optimizaciones adicionales, como la realocación segura en contenedores de la STL (Standard Template library). Cuando estas operaciones no están marcadas como `noexcept` o no garantizan que no pueden lanzar excepciones, los contenedores utilizan copias en lugar de movimientos por motivos de seguridad. Esto, además de disminuir el rendimiento, puede introducir riesgos asociados al manejo de excepciones, estados intermedios inconsistentes o rutas de recuperación (rollback) no previstas. **La seguridad del sistema mejora cuando se garantiza que estas operaciones no pueden lanzar excepciones**.

Un aspecto especialmente crítico en el ámbito de la seguridad es evitar que una operación de movimiento parcialmente ejecutada deje un objeto en un estado no válido. Las excepciones dentro de los constructores o asignadores de movimiento pueden generar vulnerabilidades como uso posterior de memoria corrupta, fallos de validación de datos o violaciones de invariantes. Marcar estas operaciones como `noexcept` obliga a diseñar implementaciones robustas que mantengan siempre invariantes consistentes y evita condiciones en las que un atacante pudiera explotar estados intermedios no controlados.

Este tipo de patrones inseguros introduce rutas de error inesperadas en operaciones que deberían ser triviales, generando puntos de fallo difíciles de detectar y facilitando potenciales vulnerabilidades explotables bajo determinadas condiciones de carga o manipulación de datos. Mantener todas las operaciones de movimiento marcadas con `noexcept` obliga a un diseño más seguro, reduce la complejidad del control de errores y ayuda a mantener la integridad del programa a lo largo de todo su ciclo de vida.

Riesgos asociados

- Fugas de memoria y recursos debido a operaciones fallidas y administradas incorrectamente durante el movimiento de objetos.
- Inestabilidad del sistema o de la aplicación por lanzar excepciones inesperadas en contextos donde se espera no fallar (por ejemplo, durante la propagación de contenedores STL).
- Riesgo de terminación no controlada (std::terminate) al utilizar clases en algoritmos que requieren operaciones noexcept.
- Disminución del rendimiento, ya que contenedores estándar optimizan el movimiento solo para tipos noexcept; de no ser así, pueden provocar copias costosas innecesarias.
- Fallas de seguridad por el mal manejo de estados inconsistentes si ocurre una

excepción en una operación de movimiento que no es noexcept.

- Mayor superficie de ataque, ya que los errores de excepción no controlada pueden ser explotables para denegación de servicio (DoS).
- Dificultad en la mantenibilidad del software y en la identificación de errores, ya que los lugares donde se podrían lanzar excepciones son menos obvios.
- Comportamientos indefinidos al usar APIs o frameworks que asumen que las operaciones de movimiento no lanzan excepciones.

Buenas prácticas

- Declarar explícitamente el constructor de movimiento y el operador de asignación por movimiento como `noexcept` cuando no lancen excepciones.

```
//-- Movimiento sin especificar noexcept
class Recurso {
  std::unique_ptr ptr;
public:
  Recurso(Recurso&& other) {
    ptr = std::move(other.ptr);
  }
};
```

```
//-- Movimiento declarado como noexcept
class Recurso {
  std::unique_ptr ptr;
public:
  Recurso(Recurso&& other) noexcept {
    ptr = std::move(other.ptr);
  }
};
```

- Marcar como `noexcept` el operador de asignación por movimiento cuando solo transfiera recursos que ya son seguros.

```
//-- Operador de movimiento sin noexcept
class Buffer {
  std::vector datos;
public:
  Buffer& operator=(Buffer&& other) {
    datos = std::move(other.datos);
    return *this;
  }
};
```

```
//-- Operador de movimiento con noexcept
```

```
class Buffer {
  std::vector datos;
public:
  Buffer& operator=(Buffer&& other) noexcept {
    datos = std::move(other.datos);
    return *this;
  }
};
```

- Utilizar `= default` junto con `noexcept` cuando el compilador pueda generar automáticamente la operación de movimiento segura.

```
//-- Constructor de movimiento generado sin especificar noexcept
class Conexion {
  std::unique_ptr handle;
public:
  Conexion(Conexion&&) = default;
};
```

```
//-- Constructor de movimiento generado como noexcept
class Conexion {
  std::unique_ptr handle;
public:
  Conexion(Conexion&&) noexcept = default;
};
```

- Usar especificación condicional `noexcept` cuando la seguridad dependa de los miembros internos.

```
//-- Movimiento sin especificación condicional
template<typename T>
class Contenedor {
  T valor;
public:
  Contenedor(Contenedor&& other) {
    valor = std::move(other.valor);
  }
};
```

```
//-- Movimiento con noexcept condicional
template<typename T>
class Contenedor {
  T valor;
public:
  Contenedor(Contenedor&& other)
noexcept(std::is_nothrow_move_constructible<T>::value) {
    valor = std::move(other.valor);
```

```
  }
};
```

2.5.4.2 Estándares metodológicos

- **C++ Core-C.66**: Make move operations noexcept.

2.5.5 [cpp.ini.5] - Evitar realizar slicing en objetos derivados

ID	cpp.ini.5	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se centra en prevenir situaciones en las que un objeto derivado es copiado o asignado a una variable de tipo base sin utilizar referencias o punteros, provocando la pérdida silenciosa de la información específica de la clase hija. Este fenómeno, conocido como **object slicing**, puede introducir vulnerabilidades, comportamientos inesperados y violaciones de invariantes de seguridad. Garantizar el uso correcto de polimorfismo seguro reduce errores lógicos, minimiza estados inconsistentes y evita exposiciones accidentales de datos al perder atributos o métodos esenciales de la clase derivada.

Codificación insegura

Uso de herencia para representar distintos tipos de empleados y mostrar su salario. Se pretende observar cómo el paso de objetos derivados por valor provoca pérdida de comportamiento polimórfico, afectando al resultado esperado.

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

class Empleado {
public:
  virtual double calcularSalario() const {
    return 1000.0;
  }

  virtual std::string tipo() const {
    return "Empleado";
  }

  virtual ~Empleado() = default;
};

//-- Función que recibe el objeto por valor, provocando slicing
void mostrarSalario(Empleado e) {
  std::cout << e.tipo() << " cobra " << e.calcularSalario() << std::endl;
```

```
}

class EmpleadoTemporal : public Empleado {
public:
  double calcularSalario() const override {
    return 800.0;
  }
  std::string tipo() const override {
    return "EmpleadoTemporal";
  }
};

int main() {
  EmpleadoTemporal temp;
  //-- Se pierde el comportamiento derivado al pasarlo por valor
  mostrarSalario(temp);
  return 0;
}
```

Codificación recomendada

Uso de herencia para representar distintos tipos de empleados y mostrar su salario manteniendo el comportamiento polimórfico. Se busca preservar la información del objeto derivado evitando copias por valor y utilizando referencias para mantener la integridad del tipo dinámico.

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

class Empleado {
public:
  virtual double calcularSalario() const {
    return 1000.0;
  }

  virtual std::string tipo() const {
    return "Empleado";
  }

  virtual ~Empleado() = default;
};

//-- Función que recibe el objeto por referencia constante para evitar slicing
void mostrarSalario(const Empleado& e) {
  std::cout << e.tipo() << " cobra " << e.calcularSalario() << std::endl;
}

class EmpleadoTemporal : public Empleado {
```

```
public:
  double calcularSalario() const override {
    return 800.0;
  }
  std::string tipo() const override {
    return "EmpleadoTemporal";
  }
};

int main() {
  EmpleadoTemporal temp;
  //-- Se mantiene el comportamiento polimórfico al usar referencia
  mostrarSalario(temp);
  return 0;
}
```

2.5.5.1 Descripción

El **object slicing** es un problema frecuente que aparece cuando un objeto derivado es copiado en una variable de tipo base. Esta acción provoca que los atributos y comportamientos específicos de la clase hija se pierdan al reconstruirse únicamente la parte correspondiente a la clase padre. Aunque el lenguaje proporciona mecanismos potentes de polimorfismo y herencia, también permite copias implícitas que pueden conducir a vulnerabilidades y errores lógicos, especialmente en sistemas donde la integridad de los datos y de los invariantes de clase es fundamental. Por esta razón, es una práctica de codificación segura evitar cualquier operación que pueda resultar en slicing, ya que afecta directamente la integridad del objeto y puede originar fallos en tiempo de ejecución difíciles de detectar.

El mecanismo que desencadena el slicing puede surgir en asignaciones directas, en pasaje de parámetros por valor o en retornos de funciones. Cuando se utiliza un objeto derivado en un contexto donde se esperaría polimorfismo, pero la operación se realiza por valor, la semántica de C++ dicta que solo la parte base del objeto se copiará, ignorando atributos internos, información de estado extendida e incluso la implementación de métodos virtuales que dependen del tipo dinámico real. Esto puede provocar que ciertas validaciones críticas queden deshabilitadas o que componentes de seguridad definidos en la clase derivada no se ejecuten correctamente. En sistemas donde los objetos representan entidades sensibles o configuraciones de seguridad, la pérdida silenciosa de datos puede ser especialmente peligrosa.

Para evitar este problema, debe fomentarse sistemáticamente el uso de **referencias** y **punteros inteligentes** en todos los escenarios donde se deba mantener el comportamiento polimórfico. Estos mecanismos permiten preservar el tipo dinámico real del objeto y evitan copias implícitas que generen slicing. Resulta igualmente importante evitar funciones o constructores que acepten parámetros por valor cuando se trabaja con jerarquías de clases destinadas al polimorfismo. Una buena práctica

recomendada es hacer que las clases base tengan métodos y destructores virtuales, asegurando que la estructura polimórfica sea robusta y coherente entre componentes.

Riesgos asociados

- Pérdida de información de las subclases al copiar un objeto derivado en una variable de tipo base.
- Comportamiento inesperado o incorrecto al invocar métodos virtuales, ya que se pierde el polimorfismo.
- Difícil depuración y mantenimiento del código por resultados incoherentes o errores silenciosos.
- Exposición a vulnerabilidades debido a la manipulación incorrecta de los datos (ejemplo: acceso a datos ya no presentes tras el slicing).
- Fallo en mecanismos de serialización o persistencia al omitir atributos específicos de la clase derivada.
- Posibles errores de lógica e inconsistencias en estructuras de datos heterogéneas.
- Imposibilidad de aplicar correctamente patrones de diseño orientados a objetos, como el patrón polimórfico.

Buenas prácticas

- Usar referencias o punteros a la clase base en lugar de copias por valor para evitar slicing.

```
class Base {
  public:
    virtual void imprimir() const {}
};

class Derivada : public Base {
  public:
    void imprimir() const override {}
};

//-- Copia por valor que provoca pérdida de la parte derivada
void procesar(Base obj) {
  obj.imprimir();
}
```

```
//-- Uso de referencia para preservar el polimorfismo
void procesarSeguro(const Base& obj) {
  obj.imprimir();
```

```
}
```

- Evitar almacenar objetos derivados por valor en contenedores de objetos base.

```
class Base {
  public:
    virtual void ejecutar() const {}
};

class Derivada : public Base {
  public:
    void ejecutar() const override {}
};

//-- Almacenamiento por valor que elimina el comportamiento derivado
std::vector lista;
lista.push_back(Derivada());
```

```
//-- Uso de punteros inteligentes para mantener el tipo dinámico
std::vector> listaSegura;
listaSegura.push_back(std::make_unique());
```

- Declarar destructores virtuales en clases base para asegurar destrucción correcta al usar polimorfismo.

```
class Base {
  public:
    ~Base() {}
};

class Derivada : public Base {
  public:
    ~Derivada() {}
};

//-- Eliminación a través de puntero base sin destructor virtual
Base* ptr = new Derivada();
delete ptr;
```

```
//-- Destructor virtual que garantiza liberación completa
class BaseSegura {
  public:
    virtual ~BaseSegura() {}
};
```

- Eliminar o proteger constructores de copia en la clase base si no está diseñada

para ser copiada por valor.

```
class Base {
  public:
    Base() = default;
    Base(const Base&) = default;
};

//-- Copia implícita que permite slicing
Derivada d;
Base b = d;
```

```
//-- Constructor de copia protegido para evitar copias por valor
class BaseSegura {
  public:
    BaseSegura() = default;
  protected:
    BaseSegura(const BaseSegura&) = default;
};
```

2.5.5.2 Estándares metodológicos

- **CERT/CPP-OOP51**: Do not slice derived objects.

2.5.6 [cpp.ini.6] - Evitar asignar arrays de objetos derivados a punteros de la clase base

ID	cpp.ini.6	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este requisito de seguridad enfatiza la importancia de no asignar arrays de objetos de una clase derivada a punteros de la clase base, ya que esta práctica provoca un comportamiento indefinido al acceder a los elementos del array. El problema se origina porque el tamaño real de cada elemento del array no coincide con el tamaño que la clase base espera, lo que causa desalineación de memoria, corrupción interna de objetos y fallos de seguridad. Esta guía ayuda a asegurar que las manipulaciones de memoria sean correctas y seguras cuando se utilicen jerarquías de clases y polimorfismo.

Codificación insegura

Uso de herencia con asignación de un array de objetos derivados a un puntero de la clase base para recorrerlos polimórficamente. Se pretende mostrar el impacto que tiene

esta práctica cuando se realiza aritmética de punteros sobre un array cuyo tipo real no coincide con el tipo del puntero utilizado.

```
#include <iostream>

class Base {
public:
  //-- Método virtual para mostrar información
  virtual void mostrar() const {
    std::cout << "Base" << std::endl;
  }

  //-- Destructor virtual
  virtual ~Base() {}
};

class Derivada : public Base {
  int valor;

public:
  //-- Constructor que inicializa el valor
  Derivada(int v) : valor(v) {}

  //-- Sobrescritura del método mostrar
  void mostrar() const override {
    std::cout << "Derivada: " << valor << std::endl;
  }
};

int main() {
  //-- Se crea un array de objetos Derivada
  Derivada arrayDerivadas[2] = { Derivada(1), Derivada(2) };
  //-- Se asigna el array de Derivada a un puntero Base*
  Base* punteroBase = arrayDerivadas;
  //-- Se recorre el array usando aritmética de punteros sobre Base*
  for (int i = 0; i < 2; ++i) {
    punteroBase[i].mostrar();
  }
  return 0;
}
```

Codificación recomendada

Gestión segura de una colección de objetos derivados utilizando contenedores y punteros polimórficos, evitando la conversión de arrays completos entre tipos relacionados. Se busca mostrar cómo mantener el polimorfismo sin incurrir en problemas de aritmética de punteros ni en comportamientos indefinidos.

```
#include <iostream>
#include <vector>
#include <memory>

class Base {
public:
  //-- Método virtual para mostrar información
  virtual void mostrar() const {
    std::cout << "Base" << std::endl;
  }

  //-- Destructor virtual
  virtual ~Base() {}
};

class Derivada : public Base {
  int valor;

public:
  //-- Constructor que inicializa el valor
  Derivada(int v) : valor(v) {}

  //-- Sobrescritura del método mostrar
  void mostrar() const override {
    std::cout << "Derivada: " << valor << std::endl;
  }
};

int main() {
  //-- Vector de punteros inteligentes a Base
  std::vector<std::unique_ptr<Base>> elementos;
  //-- Se agregan instancias de Derivada al contenedor
  elementos.push_back(std::make_unique<Derivada>(1));
  elementos.push_back(std::make_unique<Derivada>(2));
  //-- Se recorren los elementos de forma polimórfica y segura
  for (const auto& elem : elementos) {
    elem->mostrar();
  }
  return 0;
}
```

2.5.6.1 Descripción

El uso incorrecto de punteros a clases base para manipular arrays de objetos derivados representa una fuente significativa de errores de integridad y fallos de memoria. El lenguaje no proporciona un mecanismo automático para gestionar el tamaño dinámico de objetos polimórficos dentro de arrays contiguos, por lo que asignar un array de objetos de una clase derivada y tratarlo como un array de la clase base conduce a

comportamientos no definidos. Este requisito profundiza en las razones técnicas que justifican evitar este patrón, en los riesgos asociados, en los mecanismos alternativos permitidos y seguros, y en ejemplos prácticos que ilustran el problema.

Cuando se declara un array de objetos, cada elemento debe tener el mismo tamaño en memoria. Esto funciona bien para tipos triviales o clases sin herencia, pero en clases derivadas, el tamaño de los objetos suele ser mayor que el de la clase base debido a miembros adicionales o alineación de memoria. Por ello, si se asigna un array de objetos derivados a un puntero de la clase base, el código que recorre el array usando la aritmética de punteros basada en la clase base calculará direcciones incorrectas. Este desacoplamiento provoca **corrupción de memoria**, **lecturas fuera de límites**, **desbordamientos de buffer implícitos** y, en consecuencia, vulnerabilidades explotables.

El comportamiento indefinido derivado de esta práctica puede dar lugar a múltiples problemas de seguridad. Una desalineación de punteros puede sobrescribir otros objetos en memoria, comprometer el funcionamiento del programa o exponer estructuras internas que podrían filtrarse involuntariamente.

Las colecciones de punteros inteligentes proporcionan un modo seguro de almacenar objetos derivados en estructuras homogéneas tratadas desde la interfaz de la clase base. A diferencia de los arrays de objetos, los punteros respetan el tamaño real del objeto, evitan la reinterpretación errónea de memoria y permiten una administración de ciclo de vida automatizada. Proteger la integridad del programa mediante estas prácticas es esencial para garantizar un comportamiento robusto y libre de vulnerabilidades al gestionar familias de clases polimórficas.

Riesgos asociados

- **Corrupción de memoria por desbordamiento**: Al acceder a los elementos a través del puntero de la clase base, el tamaño de un objeto derivado puede ser mayor que el tamaño de la clase base, provocando escritura o lectura fuera de límites.
- **Comportamiento indefinido**: El acceso incorrecto a los miembros de los objetos derivados puede llevar a estados impredecibles o fallos graves en la ejecución del programa.
- **Fugas de memoria**: En operaciones de destrucción (por ejemplo, utilizando delete[] en un puntero base), los destructores de los objetos derivados podrían no ser invocados correctamente, impidiendo la liberación adecuada de recursos.
- **Vulnerabilidades de seguridad**: El manejo incorrecto de arrays puede ser explotado por atacantes para ejecutar código arbitrario, acceder a información sensible o realizar ataques de buffer overflow.
- **Errores lógicos difíciles de detectar**: Los fallos derivados de esta práctica pueden manifestarse de forma intermitente y compleja, dificultando el diagnóstico y la

corrección.

- **Pérdida de información o datos corruptos**: El mal mapeo de memoria puede sobrescribir información crítica de los objetos gestionados, dañando la integridad de los datos.

Buenas prácticas

- Evitar la conversión implícita de arrays de objetos derivados a punteros de la clase base, ya que puede provocar comportamiento indefinido al acceder a los elementos.

```
//-- Conversión implícita de array derivado a puntero base
class Base { public: virtual ~Base() {} };
class Derivada : public Base { int x; };

Derivada arr[3];
Base* p = arr;
p[1] = Base();
```

```
//-- Uso de contenedores polimórficos con punteros inteligentes
class Base { public: virtual ~Base() {} };
class Derivada : public Base { int x; };

std::vector> v;
v.push_back(std::make_unique());
v.push_back(std::make_unique());
```

- Preferir el uso de contenedores estándar en lugar de arrays estáticos cuando se trabaje con jerarquías de herencia.

```
//-- Array estático de objetos derivados tratado como base
class Base { public: virtual void f() {} };
class Derivada : public Base { public: void f() override {} };

Derivada arr[2];
Base* p = arr;
p[0].f();
```

```
//-- Contenedor estándar con referencias o punteros a base
class Base { public: virtual void f() {} };
class Derivada : public Base { public: void f() override {} };

std::vector> v;
Derivada d1, d2;
v.push_back(d1);
v.push_back(d2);
```

```
v[0].get().f();
```

- Si se requiere almacenamiento contiguo, utilizar arrays o contenedores del tipo base y almacenar objetos homogéneos del mismo tipo.

```
//-- Array de derivados asignado a puntero base
class Base { public: virtual ~Base() {} };
class Derivada : public Base { int x; };

Derivada arr[4];
Base* p = arr;
```

```
//-- Array homogéneo del mismo tipo base
class Base { public: virtual ~Base() {} };

Base arr[4];
Base* p = arr;
```

2.5.6.2 Estándares metodológicos

- **C++ Core-C.152**: Never assign a pointer to an array of derived class objects to a pointer to its base.

2.5.7 [cpp.ini.7] - Evitar el uso de uniones

ID	cpp.ini.7	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Las uniones permiten superponer diferentes tipos de datos en la misma región de memoria, lo que puede introducir riesgos de seguridad debido a accesos inesperados, rompimiento de invariantes de tipo y comportamientos indefinidos. Este requisito promueve evitar su uso en la medida de lo posible, sustituyéndolas por alternativas más seguras como **std::variant** o clases bien encapsuladas. El objetivo es prevenir errores de memoria, corrupción de datos, fallos de tipo y vulnerabilidades explotables asociadas a la interpretación incorrecta de datos.

Codificación insegura

Uso de una unión para almacenar distintos tipos en la misma región de memoria, accediendo a un miembro diferente al último escrito. Se busca mostrar cómo esta práctica puede generar comportamiento indefinido y errores difíciles de detectar al no existir control de tipo ni gestión segura del estado activo.

```
#include <iostream>

//-- Definición de una unión para almacenar distintos tipos
union Data {
  int intValue;
  float floatValue;
  char charValue;
};

int main() {
  //-- Se crea una instancia de la unión
  Data data;
  //-- Se asigna un valor entero
  data.intValue = 65;
  //-- Se accede a un miembro distinto al último escrito
  //-- Esto provoca comportamiento indefinido
  std::cout << "Valor como float: " << data.floatValue << std::endl;
  //-- Se vuelve a interpretar la misma memoria como char
  std::cout << "Valor como char: " << data.charValue << std::endl;
  return 0;
}
```

Codificación recomendada

Uso de una alternativa segura basada en `std::variant` para representar distintos tipos en una misma variable, garantizando control de tipo en tiempo de compilación y ejecución. Se busca mostrar cómo evitar reinterpretaciones inseguras de memoria y reducir riesgos de comportamiento indefinido.

```
#include <iostream>
#include <variant>

//-- Definición de un tipo seguro que puede contener varios tipos
using Data = std::variant<int, float, char>;

int main() {
  //-- Se crea una instancia del tipo seguro
  Data data;
  //-- Se asigna un valor entero
  data = 65;
  //-- Se verifica y accede de forma segura al tipo almacenado
  if (std::holds_alternative<int>(data)) {
    std::cout << "Valor entero: " << std::get<int>(data) << std::endl;
  }
  //-- Se cambia el valor a tipo char de forma controlada
  data = 'A';
  if (std::holds_alternative<char>(data)) {
    std::cout << "Valor char: " << std::get<char>(data) << std::endl;
```

```
    }
    return 0;
}
```

2.5.7.1 Descripción

El uso de uniones (`union`) en C++ proviene del legado de C y permite superponer múltiples campos en la misma ubicación de memoria. Aunque existen casos donde podían considerarse útiles por motivos de optimización, en la actualidad se consideran constructos peligrosos debido a su impacto en la seguridad, la legibilidad del código y la integridad de los datos. Las uniones rompen las garantías del sistema de tipos al permitir interpretar una misma región de memoria como distintas entidades sin control, lo que puede generar comportamientos indefinidos, corrupción silenciosa de datos y condiciones propicias para vulnerabilidades explotables. Este requisito establece evitar su uso salvo en casos extremadamente excepcionales y controlados, sustituyendo estas construcciones por alternativas modernas, seguras y expresivas del lenguaje.

La principal amenaza que introducen las uniones es la pérdida de las garantías de integridad del sistema de tipos. Cuando un valor escrito como un tipo se lee como otro diferente, los datos pueden quedar truncados, reinterpretados o invalidados de forma silenciosa. Este comportamiento es especialmente peligroso en sistemas donde se manejan estructuras críticas, como datos de red, valores utilizados en lógica de control, indicadores de estado, señales de hardware o información sensible. Asimismo, las uniones pueden generar inconsistencias en la memoria cuando un desarrollador asume que el valor interpretado es válido, sin que exista un mecanismo automático para validar cuál de los campos es efectivamente el activo. Estas condiciones conducen a errores lógicos y vulnerabilidades en tiempo de ejecución.

El comportamiento indefinido es un riesgo frecuente al usar uniones. Esto se presenta cuando se accede a un miembro diferente al que se escribió por última vez, salvo contadas excepciones describiendo tipos triviales compatibles. Cuando el lenguaje no puede garantizar la validez del acceso, el programa puede producir resultados erráticos o permitir aprovechar corrupciones para explotar el sistema. En sistemas críticos o de alta disponibilidad, estas condiciones son especialmente peligrosas. Algunos errores comunes incluyen reinterpretación de estructuras internas, pérdidas de punteros, lectura de padding no inicializado, dependencia de alineación no garantizada y accesos ilegítimos a memoria.

En C++ moderno existen mecanismos seguros que permiten representar variantes de datos sin sacrificar rendimiento ni claridad. Entre las alternativas recomendadas destacan **std::variant** y **clases envoltorio** que encapsulan claramente la intención y el estado. Las plantillas modernas permiten definir objetos que representen múltiples tipos posibles, y aportan seguridad en tiempo de compilación al obligar al desarrollador a manejar explícitamente todos los casos posibles.

Desde una perspectiva de integridad, las uniones representan una amenaza directa porque rompen la estructura lógica de los datos, permitiendo crear estados inconsistentes que pueden generar vulnerabilidades difíciles de detectar. También complican la trazabilidad en depuración o auditoría, ya que el estado interno depende de la interpretación realizada y no existe un mecanismo automático que indique cuál miembro es válido. Su uso afecta la previsibilidad del programa y reduce la capacidad de análisis estático. En entornos seguros o sistemas de alta criticidad, estas debilidades son inaceptables, por lo que deben reemplazarse por mecanismos modernos del lenguaje que proporcionen garantías explícitas.

Riesgos asociados

- Lectura de datos inválidos o corruptos debido a la superposición de memoria entre los miembros de la unión.
- Acceso a un miembro de la unión diferente al que fue escrito, lo que puede causar comportamiento indefinido.
- Fugas de información sensible si distintos tipos o estructuras comparten la misma región de memoria.
- Difícil trazabilidad y depuración debido a la naturaleza poco transparente del almacenamiento en uniones.
- Incompatibilidad con constructores, destructores y miembros no triviales, lo que puede llevar a fallos en la inicialización o destrucción de objetos.
- Problemas de portabilidad, ya que la representación en memoria de las uniones puede variar entre compiladores o arquitecturas.
- Vulnerabilidades a ataques de tipo type-punning, facilitando la manipulación inesperada de datos.
- Error en el alineamiento de la memoria, causando fallos de acceso a memoria en ciertas arquitecturas.
- Dificultad en la aplicación de controles de acceso y validaciones de tipo en tiempo de ejecución.

Buenas prácticas

- Reemplazar uniones por estructuras o clases con tipos bien definidos para evitar ambigüedad de memoria.

```
//-- Uso de union que comparte memoria entre tipos incompatibles
union Dato {
  int entero;
  float decimal;
};
```

```
//-- Uso de struct con campos diferenciados y control explícito
struct DatoSeguro {
  int entero;
  float decimal;
};
```

- Utilizar std::variant para representar múltiples tipos de forma segura.

```
//-- Union sin control del tipo activo
union Valor {
  int entero;
  double real;
};
```

```
//-- Uso de std::variant con verificación de tipo en tiempo de ejecución
std::variant valor;
valor = 10;
if (std::holds_alternative(valor)) {
  int v = std::get(valor);
}
```

- Evitar reinterpretaciones peligrosas de memoria y usar conversiones seguras.

```
//-- Acceso a miembro distinto al último asignado en union
union Numero {
  int entero;
  float flotante;
};
Numero n;
n.entero = 1065353216;
float f = n.flotante;
```

```
//-- Conversión explícita y controlada entre tipos
int entero = 1065353216;
float flotante = static_cast(entero);
```

2.5.7.2 Estándares metodológicos

- **MISRA CPP-12.3.1**: union shall not be used.

2.5.8 [cpp.ini.8] - Evitar ciclos durante la inicialización de objetos estáticos

ID	cpp.ini.8	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este requisito busca minimizar riesgos derivados de dependencias cíclicas entre objetos estáticos, las cuales pueden provocar comportamientos indefinidos, fallos de seguridad y resultados inesperados en tiempo de ejecución. La inicialización estática ocurre antes del inicio del programa, y su orden no está garantizado entre unidades de traducción, lo que puede exponer datos no inicializados o provocar accesos inválidos. Aplicar este control implica estructurar el código para evitar referencias cruzadas en constructores estáticos, emplear técnicas seguras como funciones `factory`, inicialización diferida y patrones de diseño que reduzcan el acoplamiento global.

Codificación insegura

Se muestra una situación donde dos objetos estáticos globales dependen entre sí durante su construcción. El objetivo es entender cómo el orden de inicialización entre objetos estáticos en distintas unidades puede generar comportamientos indefinidos o fallos difíciles de detectar cuando existe dependencia circular.

```
#include <iostream>

class B; // Declaración adelantada

class A {
public:
  A();
  void saludar() const {
    std::cout << "Hola desde A" << std::endl;
  }
};

class B {
public:
  B();
  void saludar() const {
    std::cout << "Hola desde B" << std::endl;
  }
};

//-- Objetos estáticos globales con dependencia circular
A objetoA;
B objetoB;

A::A() {
```

```
  //-- Uso de objetoB durante la inicialización de A
  objetoB.saludar();
}

B::B() {
  //-- Uso de objetoA durante la inicialización de B
  objetoA.saludar();
}

int main() {
  //-- Punto de entrada del programa
  std::cout << "Programa iniciado" << std::endl;
  return 0;
}
```

Codificación recomendada

Se presenta una forma segura de evitar dependencias circulares durante la inicialización estática utilizando inicialización diferida mediante variables estáticas locales en funciones. El objetivo es garantizar que cada objeto se construya la primera vez que se necesita, evitando problemas de orden de inicialización.

```
#include <iostream>

class B; // Declaración adelantada

class A {
public:
  A() {
    //-- Constructor sin dependencias externas
  }
  void saludar() const {
    std::cout << "Hola desde A" << std::endl;
  }
};

class B {
public:
  B() {
    //-- Constructor sin dependencias externas
  }
  void saludar() const {
    std::cout << "Hola desde B" << std::endl;
  }
};

//-- Inicialización diferida mediante función
A& obtenerA() {
```

```
  static A instanciaA;
  return instanciaA;
}

//-- Inicialización diferida mediante función
B& obtenerB() {
  static B instanciaB;
  return instanciaB;
}

int main() {
  //-- Los objetos se crean al usarse por primera vez
  obtenerA().saludar();
  obtenerB().saludar();
  return 0;
}
```

2.5.8.1 Descripción

La inicialización estática puede introducir riesgos severos cuando existen dependencias cruzadas entre objetos globales o estáticos, tanto dentro de una misma unidad de traducción como, especialmente, entre módulos distintos. El lenguaje no garantiza un orden de inicialización coherente más allá de la secuencia dentro de una misma unidad de traducción, y este comportamiento puede derivar en ciclos indirectos o directos que expongan valores sin inicializar, desborden recursos en constructores o produzcan fallos de ejecución. La aplicación disciplinada de este control permite proteger la integridad de datos críticos y prevenir accesos ilegítimos a estados incoherentes.

Los ciclos en la inicialización estática provocan situaciones en las cuales un objeto emplea otro que aún no ha sido inicializado. En contextos de seguridad, esto puede ser especialmente problemático: estructuras criptográficas, gestores de memoria, instancias globales de control de acceso o buffers de registro pueden quedar expuestos a operaciones sobre memoria no válida. Los efectos pueden ir desde valores corruptos hasta fallas reproducibles que faciliten la explotación. Es crucial comprender que este comportamiento puede manifestarse de manera no determinista en distintas plataformas o compiladores.

Existen patrones ampliamente adoptados en el desarrollo seguro para eliminar los ciclos de inicialización y mitigar los riesgos predecibles. Una de las estrategias más eficaces es la inicialización diferida (**lazy initialization**), donde un objeto global se reemplaza por una función que devuelve una instancia estática interna correctamente inicializada cuando se necesita. Esto reduce drásticamente el riesgo de inicialización circular. Asimismo, el modelo **construct on first use** permite que la creación del objeto se realice de forma controlada, evitando el orden de inicialización global.

Otra técnica recomendada implica encapsular dependencias mediante patrones como **Singleton seguro** o **Service Locator** acotado, que producen instancias en el momento oportuno y no durante la fase estática del programa. Reducir el uso de objetos estáticos complejos, sustituir dependencias globales por inyección manual o diferida, y desacoplar módulos minimiza los riesgos sustancialmente.

Riesgos asociados

- Condiciones de carrera durante la inicialización de variables globales o estáticas.
- Desreferenciación de punteros nulos si se accede a un objeto aún no inicializado.
- Comportamiento indefinido por el uso de objetos parcialmente construidos.
- Dificultad para depurar errores, ya que pueden aparecer de manera intermitente o en entornos específicos.
- Fugas de memoria por asignaciones realizadas antes de una falla en la inicialización.
- Dependencia oculta y acoplamiento fuerte entre módulos o clases.
- Elevado riesgo de fallos en el arranque de la aplicación o en el uso de bibliotecas compartidas.

Buenas prácticas

- Utilizar la técnica de inicialización bajo demanda (construct on first use) para evitar dependencias circulares entre objetos estáticos globales.

```
//-- Inicialización directa de objetos estáticos con dependencia cruzada
class B;
class A {
  public:
  A();
};
class B {
  public:
  B();
};
A a;
B b;
```

```
//-- Inicialización diferida mediante función estática local
class B;
class A {
  public:
  static A& instance() {
    static A a;
    return a;
```

```
  }
};
class B {
  public:
  static B& instance() {
    static B b;
    return b;
  }
};
```

- Evitar que constructores de objetos estáticos invoquen funciones que dependan de otros objetos estáticos globales.

```
//-- Constructor que usa otro objeto estático no garantizado
class Logger {
  public:
  void log(const char* msg);
};
Logger globalLogger;
class Config {
  public:
  Config() {
    globalLogger.log("Init");
  }
};
Config globalConfig;
```

```
//-- Uso explícito tras inicialización controlada
class Logger {
  public:
  void log(const char* msg);
};
Logger& getLogger() {
  static Logger logger;
  return logger;
}
class Config {
  public:
  void init() {
    getLogger().log("Init");
  }
};
```

- Centralizar la creación de objetos estáticos relacionados dentro de una misma unidad de compilación para controlar el orden de inicialización.

```
//-- Objetos estáticos definidos en múltiples archivos con dependencias implícitas
```

```
extern int value;
int result = value + 1;
```

```
//-- Agrupación en una única función de inicialización controlada
int& getValue() {
  static int value = 10;
  return value;
}
int computeResult() {
  return getValue() + 1;
}
```

2.5.8.2 Estándares metodológicos

- **CERT/CPP-DCL56**: Avoid cycles during initialization of static objects.

2.6 Expresiones

2.6.1 [cpp.exp.1] - Evitar que las expresiones dependan de un orden de evaluación por defecto

ID	cpp.exp.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este control exige que la lógica del software no dependa de un orden de evaluación no definido o no garantizado por el estándar del lenguaje, donde la secuencia de evaluación de argumentos, operadores y expresiones compuestas puede ser indeterminada o no especificada. Cuando el código depende implícitamente de ese orden no garantizado, pueden aparecer comportamientos inesperados, vulnerabilidades de integridad, errores en tiempo de ejecución y resultados inconsistentes entre compiladores. Para prevenir estos riesgos, es esencial estructurar las expresiones para que cada operación tenga un orden claro y explícito, evitando efectos laterales y asegurando un comportamiento determinista.

Codificación insegura

Uso de múltiples incrementos sobre la misma variable dentro de una llamada a función, asumiendo que los argumentos se evaluarán en un orden específico. La intención es mostrar cómo depender del orden de evaluación puede generar resultados impredecibles y comportamientos distintos según el compilador.

```
#include <iostream>
```

```
//-- Función que recibe dos enteros y los muestra
void imprimir(int a, int b) {
  std::cout << "a: " << a << ", b: " << b << std::endl;
}

int main() {
  //-- Variable inicial
  int x = 5;
  //-- Se incrementa la misma variable en dos argumentos
  //-- El orden de evaluación de los argumentos no está definido
  imprimir(x++, x++);
  //-- Valor final de x
  std::cout << "x final: " << x << std::endl;
  return 0;
}
```

Codificación recomendada

Separación explícita de las operaciones que modifican el estado antes de utilizarlas como argumentos, eliminando cualquier dependencia del orden de evaluación. Se busca garantizar un comportamiento determinista y portable entre compiladores.

```
#include <iostream>

//-- Función que recibe dos enteros y los muestra
void imprimir(int a, int b) {
  std::cout << "a: " << a << ", b: " << b << std::endl;
}

int main() {
  //-- Variable inicial
  int x = 5;
  //-- Se realizan los incrementos en instrucciones separadas
  int primero = x++;
  int segundo = x++;
  //-- Se pasan valores ya definidos, sin ambigüedad
  imprimir(primero, segundo);
  //-- Valor final de x
  std::cout << "x final: " << x << std::endl;
  return 0;
}
```

2.6.1.1 Descripción

Uno de los riesgos más frecuentes y sutiles al escribir código inseguro o propenso a errores consiste en depender del orden de evaluación de expresiones que, por especificación del lenguaje, no está garantizado. Este problema puede producir

comportamientos inconsistentes entre compiladores, versiones de compiladores o incluso entre diferentes optimizaciones del mismo compilador. Desde una perspectiva de **desarrollo seguro**, estas inconsistencias constituyen un riesgo crítico, porque facilitan la aparición de errores de integridad, condiciones de carrera internas, corrupción de memoria y fallos lógicos difíciles de detectar. Evitar depender de un orden no garantizado mejora la previsibilidad del código y reduce la superficie de vulnerabilidad asociada al comportamiento indefinido.

El estándar C++ especifica que el compilador tiene libertad para elegir el orden en el que evalúa los operandos de muchos operadores. En expresiones que incluyen `&&`, `||`, `,`, asignación o llamadas a funciones con parámetros que tienen efectos laterales, esto puede resultar en resultados inesperados. Asimismo, ciertas construcciones están marcadas explícitamente como comportamiento indefinido cuando un mismo objeto es leído y modificado sin una secuencia de puntos bien definida. En un entorno de seguridad, el comportamiento indefinido se considera una amenaza importante porque un atacante puede manipular efectos laterales, generar inconsistencias o provocar fallos difíciles de rastrear.

La dependencia de un orden no garantizado constituye un riesgo de integridad porque permite que la lógica interna del sistema se corrompa sin un patrón reproducible, generando fallos intermitentes que pueden ser explotados para corromper datos, activar comportamientos inesperados o desestabilizar módulos dependientes. En aplicaciones críticas, como sistemas empotrados, SCADA, automoción o aplicaciones financieras, estas inconsistencias pueden comprometer decisiones de control, cálculos sensibles o flujos transaccionales. Al evitar estas construcciones ambiguas, el código adquiere mayor estabilidad, predictibilidad y resistencia frente a entradas maliciosas o escenarios inesperados.

Riesgos asociados

- Comportamiento indefinido que puede provocar fallos de programa.
- Dificultad para detectar y depurar errores debido a resultados inconsistentes.
- Diferencias en la ejecución del código entre compiladores o plataformas.
- Introducción inadvertida de vulnerabilidades de seguridad explotables.
- Pérdida de portabilidad y mantenibilidad del código.
- Lectura o escritura de datos en un estado parcial o incorrecto.
- Posibilidad de corrupción de memoria u otras estructuras críticas.

Buenas prácticas

- Separar expresiones complejas en múltiples sentencias para evitar depender del orden de evaluación.

```
//-- Incrementos múltiples en una misma expresión
int i = 0;
int result = i++ + i++;
```

```
//-- Separación de operaciones para definir el orden explícitamente
int i = 0;
int a = i++;
int b = i++;
int result = a + b;
```

- No modificar una misma variable más de una vez dentro de una sola expresión.

```
//-- Modificación múltiple de la misma variable
int x = 5;
x = x + (++x);
```

```
//-- Uso de variable temporal para evitar ambigüedad
int x = 5;
int temp = x + 1;
x = x + temp;
```

- Evitar llamadas a funciones con efectos laterales dentro de una misma expresión.

```
//-- Llamadas con efectos laterales en una misma expresión
int counter = 0;
int increment() { return ++counter; }
int total = increment() + increment();
```

```
//-- Evaluaciones separadas para controlar el orden
int counter = 0;
int increment() { return ++counter; }
int a = increment();
int b = increment();
int total = a + b;
```

- No asumir el orden de evaluación de los argumentos de una función.

```
//-- Dependencia del orden de evaluación de argumentos
void log(int a, int b){
  ...
}

int i = 0;
log(i++, i++);
```

```
//-- Evaluación previa de argumentos antes de la llamada
void log(int a, int b){
  ...
}

int i = 0;
int first = i++;
int second = i++;
log(first, second);
```

- Evitar combinar operadores lógicos con efectos laterales en una misma condición.

```
//-- Efectos laterales dentro de una condición lógica
int i = 0;
if (i++ > 0 && i++ < 5) {
  process();
}
```

```
//-- Evaluaciones separadas para controlar efectos
int i = 0;
bool firstCheck = (i > 0);
i++;
bool secondCheck = (i < 5);
i++;
if (firstCheck && secondCheck) {
  process();
}
```

2.6.1.2 Estándares metodológicos

- **C++ Core-ES.43**: Avoid expressions with undefined order of evaluation.
- **MISRA CPP-4.6.1**: Operations on a memory location shall be sequenced appropriately.

2.6.2 [cpp.exp.2] - Evitar depender del orden de evaluación de los argumentos de una función

ID	cpp.exp.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este requisito de seguridad se centra en prevenir comportamientos indefinidos y vulnerabilidades derivadas de asumir un orden específico de evaluación de argumentos

en llamadas a funciones o expresiones complejas. En C++ el estándar no garantiza un orden definido en muchos contextos, lo que puede provocar fallos lógicos, corrupciones de memoria, accesos inesperados o resultados inconsistentes. Aplicar este control ayuda a asegurar que el código produzca resultados deterministas, facilite las auditorías de seguridad y reduzca los riesgos de errores difíciles de detectar que puedan afectar a la confiabilidad, integridad y robustez del software.

Codificación insegura

Se muestra una función que recibe dos argumentos y cómo el uso de una misma variable con efectos secundarios en ambos parámetros puede producir resultados distintos según el compilador o la versión del estándar. El objetivo es evidenciar los riesgos de asumir un orden de evaluación específico en los argumentos de una llamada.

```
#include <iostream>

//-- Función que suma dos enteros
int sumar(int a, int b) {
  return a + b;
}

int main() {
  //-- Variable compartida
  int x = 5;

  //-- Uso de la misma variable con post-incremento en ambos argumentos
  int resultado = sumar(x++, x++);
  //-- El resultado puede variar según el orden de evaluación
  std::cout << "Resultado: " << resultado << std::endl;
  std::cout << "Valor final de x: " << x << std::endl;
  return 0;
}
```

Codificación recomendada

Se presenta la misma operación, pero separando explícitamente las modificaciones de estado antes de la llamada a la función. El objetivo es garantizar un comportamiento determinista y evitar efectos secundarios múltiples en una misma expresión, eliminando la dependencia del orden de evaluación.

```
#include <iostream>

//-- Función que suma dos enteros
int sumar(int a, int b) {
  return a + b;
}
```

```
int main() {
  //-- Variable compartida
  int x = 5;

  //-- Evaluaciones separadas y controladas
  int a = x;
  x++;
  int b = x;
  x++;

  //-- Llamada sin efectos secundarios en los argumentos
  int resultado = sumar(a, b);
  std::cout << "Resultado: " << resultado << std::endl;
  std::cout << "Valor final de x: " << x << std::endl;
  return 0;
}
```

2.6.2.1 Descripción

La ausencia de un orden de evaluación garantizado en ciertos contextos del lenguaje C++ es una fuente frecuente de comportamientos indefinidos que afectan directamente la integridad del software. Las expresiones en las que se combinan múltiples evaluaciones con efectos secundarios, como incrementos, accesos a memoria o modificaciones de variables compartidas, pueden producir resultados inesperados cuando el compilador elige libremente el orden de ejecución. Este requisito busca establecer una guía clara para que los desarrolladores eviten situaciones donde la semántica del programa dependa implícitamente del orden de evaluación, lo cual es especialmente crítico en entornos de alta seguridad, sistemas embebidos, aplicaciones concurrentes o software que manipule recursos sensibles.

El estándar especifica explícitamente que el orden de evaluación de los argumentos de una función, así como de los operandos de la mayoría de los operadores, queda sin definir. Esto significa que expresiones como `f(a++, a)`, `v[g(i)] = h(i++)` o `x = y++ + z++` pueden ejecutarse en un orden no predecible, generando cambios inesperados en el estado del programa. Este comportamiento puede introducir vulnerabilidades graves cuando las variables afectadas intervienen en validaciones de seguridad, comprobaciones de límites, operaciones criptográficas o algoritmos sensibles. Minimizar este riesgo requiere aplicar una estrategia sistemática de estructuración del código para evitar cualquier dependencia implícita del orden evaluado.

Una técnica fundamental es aislar los efectos secundarios en expresiones separadas, asegurando así que las operaciones críticas se ejecuten de forma explícita y controlada. En lugar de combinar cambios de estado y cálculos en una misma expresión, se deben separar los pasos necesarios en instrucciones independientes. De esta manera se fuerza un orden claro y definido, reduciendo significativamente el margen para errores sutiles. Además, se recomienda no utilizar expresiones complejas en llamadas a funciones

cuando alguno de los parámetros puede alterar el estado del programa o del entorno que comparten otros argumentos.

Un patrón general de prevención consiste en asignar previamente el valor de cada argumento a variables temporales, especialmente cuando incluyen operaciones como incrementos, accesos a punteros, manipulaciones de índices o interacciones con estructuras compartidas. Esto garantiza que el valor utilizado en la llamada a función es exactamente el esperado y que no se verá afectado por decisiones internas del compilador respecto a la evaluación. También es recomendable evitar expresiones encadenadas donde un mismo identificador aparece varias veces en contextos que modifican su valor o dependen del resultado de otra parte de la expresión.

El riesgo aumenta sustancialmente en contextos multihilo, donde el orden de evaluación puede interactuar con condiciones de carrera. Si el estado modificado por una expresión depende de acciones concurrentes, una evaluación inesperada puede hacer que los valores leídos o modificados sean incorrectos, generando vulnerabilidades de integridad o provocando interbloqueos inesperados. Evitar este tipo de ambigüedad contribuye directamente a la robustez del sistema, ya que reduce puntos débiles en los que el compilador, el orden de ejecución y la concurrencia se combinan de forma peligrosa.

Además, cuando se integran módulos escritos en C, ensamblador o bibliotecas de terceros, el riesgo asociado se amplifica porque cualquier dependencia no explícita del orden de evaluación puede propagar errores o corrupciones a través de capas de ejecución. Utilizar valores precalculados evita que las rutinas externas reciban parámetros inconsistentes o inesperados que comprometan el flujo de control o generen accesos fuera de rango.

Riesgos asociados

- Resultados inesperados debido a valores de variables modificados en un orden no previsto.
- Errores de lógica difíciles de detectar y depurar.
- Comportamiento no portátil entre compiladores o plataformas diferentes.
- Vulnerabilidades de seguridad por acceso indebido a memoria o recursos.
- Corrupción de datos almacenados en variables o estructuras.
- Condiciones de carrera en sistemas concurrentes.
- Fallo en operaciones críticas por recibir argumentos incorrectos.
- Dificultad en el mantenimiento y pruebas del código.

Buenas prácticas

- Separar las expresiones con efectos secundarios antes de pasarlas como argumentos a una función.

```
int i = 0;
int j = 0;

//-- Llamada con múltiples incrementos en los argumentos
f(i++, j++);
```

```
//-- Evaluación previa de los incrementos antes de la llamada
int tmp1 = i++;
int tmp2 = j++;
f(tmp1, tmp2);
```

- No modificar una misma variable más de una vez dentro de la misma llamada a función.

```
int x = 5;

//-- Variable modificada múltiples veces en la misma expresión
g(x++, x);
```

```
//-- Modificación separada y uso posterior del valor estable
int original = x;
x++;
g(original, x);
```

- Evitar combinar acceso y modificación de contenedores dentro de una misma invocación.

```
std::vector v = {1,2,3};

//-- Acceso y modificación del contenedor en la misma llamada
h(v[0], v.push_back(4));
```

```
//-- Operaciones realizadas en pasos independientes
int first = v[0];
v.push_back(4);
h(first, v.size());
```

2.6.2.2 Estándares metodológicos

- **C++ Core-ES.44**: don't depend on order of evaluation of function arguments.

2.6.3 [cpp.exp.3] - Usar paréntesis para clarificar expresiones complejas

ID	cpp.exp.3	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Se promueve el uso estricto y consistente de paréntesis en expresiones complejas para evitar ambigüedades, errores lógicos y comportamientos inesperados derivados de la precedencia y asociatividad de operadores. El objetivo es mejorar la legibilidad del código, reducir defectos introducidos por interpretaciones erróneas del compilador o del programador y fortalecer la seguridad evitando evaluaciones inesperadas que puedan conducir a vulnerabilidades. Mediante paréntesis adicionales se fuerza una semántica clara, robusta y explícita, especialmente en contextos donde intervienen operadores de comparación, lógicos, aritméticos y combinación de incrementos o llamadas a funciones.

Codificación insegura

Evaluación de una expresión lógica y aritmética combinada donde intervienen operadores de comparación y operadores lógicos. Se pretende mostrar cómo la falta de paréntesis puede generar ambigüedad en la lectura, errores de mantenimiento y resultados distintos a los esperados debido a la precedencia de operadores.

```
#include <iostream>

//-- Punto de entrada del programa
int main() {
  //-- Variables de ejemplo
  int a = 5;
  int b = 10;
  int c = 15;

  //-- Expresión compleja sin paréntesis que clarifiquen la intención
  //-- La precedencia de operadores puede no ser evidente a simple vista
  bool resultado = a + b > c && b - a < c || a == 5;

  //-- Mostrar resultado
  if (resultado) {
    std::cout << "Condicion verdadera" << std::endl;
  } else {
    std::cout << "Condicion falsa" << std::endl;
  }
  return 0;
}
```

Codificación recomendada

Evaluación de la misma expresión lógica y aritmética, pero agrupando explícitamente las operaciones mediante paréntesis para reflejar la intención lógica. El objetivo es mejorar la claridad, reducir ambigüedades y evitar errores derivados de la precedencia implícita de operadores.

```
#include <iostream>

//-- Punto de entrada del programa
int main() {
  //-- Variables de ejemplo
  int a = 5;
  int b = 10;
  int c = 15;

  //-- Expresión compleja con paréntesis para clarificar la intención
  //-- Se agrupan explícitamente las condiciones aritméticas y lógicas
  bool resultado = ((a + b) > c && (b - a) < c) || (a == 5);

  //-- Mostrar resultado
  if (resultado) {
    std::cout << "Condicion verdadera" << std::endl;
  } else {
    std::cout << "Condicion falsa" << std::endl;
  }
  return 0;
}
```

2.6.3.1 Descripción

El uso consistente de paréntesis en expresiones complejas es una recomendación esencial para garantizar que el código se evalúe exactamente en el orden previsto por el programador. Aunque el lenguaje define reglas estrictas de precedencia y asociatividad, estas reglas pueden resultar difíciles de recordar, especialmente cuando se combinan operadores aritméticos, lógicos, comparativos y de asignación. Por ello, **el uso de paréntesis adicionales aporta claridad inmediata** tanto a humanos como a herramientas de análisis estático y reduce de forma significativa defectos de integridad y fallos lógicos.

Esta práctica también contribuye a la seguridad del software, ya que muchas vulnerabilidades se originan por evaluaciones inesperadas de expresiones. Una expresión aparentemente simple, al combinar varios operadores, puede ejecutarse en un orden no anticipado que derive en un comportamiento incorrecto. Por ejemplo, expresiones mal construidas, bajo ciertas circunstancias pueden habilitar caminos lógicos no deseados que conduzcan a estados inconsistentes del programa. Al rodear

subexpresiones relevantes con paréntesis, el desarrollador controla explícitamente el flujo de evaluación, minimizando riesgos.

Riesgos asociados

- Evaluación inesperada de operadores debido a la precedencia o asociatividad incorrecta.
- Errores lógicos que provocan comportamientos inesperados o incorrectos del programa.
- Dificultad para identificar bugs durante el mantenimiento del código.
- Introducción de vulnerabilidades por interpretación errónea de la lógica del programa.
- Posibilidad de explotar las expresiones ambiguas para manipulación maliciosa de control de flujo.
- Reducción de la legibilidad y comprensión del código por parte de otros desarrolladores.
- Problemas de portabilidad si distintos compiladores interpretan la expresión de manera diferente.

Buenas prácticas

- Agregar paréntesis para eliminar ambigüedad en expresiones con múltiples operadores lógicos y relacionales.

```
//-- Expresión compleja sin agrupación explícita
bool acceso = rol == ADMIN || rol == USER && activo;
```

```
//-- Agrupación explícita para clarificar precedencia
bool acceso = (rol == ADMIN) || ((rol == USER) && (activo));
```

- Encerrar en paréntesis las operaciones aritméticas combinadas con desplazamientos o comparaciones.

```
//-- Operación mixta sin paréntesis claros
int resultado = a + b << 2;
```

```
//-- Agrupación explícita del orden de evaluación
int resultado = (a + b) << 2;
```

- Usar paréntesis en expresiones que combinan operadores bit a bit y lógicos.

```
//-- Mezcla de operadores bit a bit y lógicos
```

```
if (flags & FLAG_ADMIN == FLAG_ADMIN) {
  autorizar();
}
```

```
//-- Evaluación explícita del resultado bit a bit
if ((flags & FLAG_ADMIN) == FLAG_ADMIN) {
  autorizar();
}
```

- Encapsular asignaciones dentro de condiciones con paréntesis adicionales para evitar ambigüedad.

```
//-- Asignación dentro de condición poco clara
if (ptr = obtenerRecurso()) {
  usar(ptr);
}
```

```
//-- Asignación explícitamente delimitada
if ((ptr = obtenerRecurso()) != nullptr) {
  usar(ptr);
}
```

- Agregar paréntesis en expresiones ternarias anidadas para mejorar la legibilidad y evitar errores.

```
//-- Operador ternario anidado sin agrupación
int nivel = prioridad > 5 ? alto : prioridad > 2 ? medio : bajo;
```

```
//-- Agrupación explícita del operador ternario
int nivel = (prioridad > 5) ? alto : ((prioridad > 2) ? medio : bajo);
```

2.6.3.2 Estándares metodológicos

- **MISRA CPP-8.0.1**: Parentheses should clarify expression meaning.

2.6.4 [cpp.exp.4] - Asegurar que el operando derecho de && y || no tenga efectos laterales persistentes

ID	cpp.exp.4	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se busca garantizar que, en expresiones lógicas que utilizan los operadores `&&` y `||`, el operando situado a la derecha no produzca **efectos laterales persistentes** que modifiquen el estado del programa de forma inesperada. Debido al cortocircuito inherente a estos operadores, dicho operando puede no llegar a evaluarse, generando inconsistencias, fallos lógicos o vulnerabilidades si las operaciones del lado derecho alteran memoria, estado interno u objetos. Aplicar esta práctica permite mantener un flujo de control predecible, seguro y resistente a manipulaciones.

Codificación insegura

Se muestra una validación compuesta con operadores lógicos donde una de las condiciones ejecuta una función que modifica estado global. El objetivo es observar cómo colocar una operación con efectos laterales en el operando derecho de `&&` puede generar cambios persistentes inesperados debido a la evaluación condicional de corto circuito.

```
#include <iostream>

int globalCounter = 0;

bool isUserValid() {
  //-- Simula una validación de usuario
  return false;
}

bool incrementCounter() {
  //-- Incrementa un contador global (efecto lateral persistente)
  globalCounter++;
  return true;
}

int main() {
  //-- Uso del operador && donde el operando derecho tiene efecto lateral
  if (isUserValid() && incrementCounter()) {
    std::cout << "Acceso concedido" << std::endl;
  }
  //-- Se muestra el valor del contador global
  std::cout << "Valor del contador: " << globalCounter << std::endl;
  return 0;
```

```
}
```

Codificación recomendada

Se presenta la misma lógica de validación separando explícitamente las operaciones con efectos laterales antes de la evaluación lógica. El objetivo es asegurar que los operandos de && sean expresiones sin efectos persistentes, mejorando la claridad, la previsibilidad y el cumplimiento de buenas prácticas de codificación segura.

```
#include <iostream>

int globalCounter = 0;

bool isUserValid() {
  //-- Simula una validación de usuario
  return false;
}

bool incrementCounter() {
  //-- Incrementa un contador global (efecto lateral persistente)
  globalCounter++;
  return true;
}

int main() {
  //-- La operación con efecto lateral se ejecuta fuera de la condición lógica
  bool counterResult = incrementCounter();

  //-- El operador && usa únicamente valores sin efectos laterales
  if (isUserValid() && counterResult) {
    std::cout << "Acceso concedido" << std::endl;
  }
  //-- Se muestra el valor del contador global
  std::cout << "Valor del contador: " << globalCounter << std::endl;
  return 0;
}
```

2.6.4.1 Descripción

Los operadores lógicos && y || aplican **evaluación de cortocircuito**, lo que implica que el segundo operando sólo se ejecutará si el resultado no puede determinarse exclusivamente con el primero. Esta característica, aunque útil para aumentar la eficiencia, puede generar riesgos de seguridad cuando el operando derecho contiene **efectos laterales persistentes**, como modificaciones de variables globales, actualización de estados de objetos, alteración de punteros o escritura en memoria compartida. Debido a que la ejecución del operando derecho no está garantizada, sus efectos pueden

producirse de forma inconsistente, provocando comportamientos no deterministas que impactan directamente en la integridad del sistema.

El principal riesgo radica en que la presencia de efectos laterales persistentes hace que el estado del programa dependa de una evaluación condicional no obvia para el lector del código. En entornos de seguridad crítica o en código con alta concurrencia, esta falta de determinismo puede originar fallos difíciles de reproducir, errores silenciosos y vulnerabilidades potenciales. La posibilidad de que el operando derecho no se ejecute puede dejar operaciones críticas sin realizar, o realizarse de manera parcial, lo cual compromete la integridad del flujo lógico. Cuando estas modificaciones se aplican sobre estructuras de memoria compartida, punteros o contadores internos, el desalineamiento resultante puede abrir puertas a estados corruptos y ataques basados en manipulación del flujo.

En escenarios donde se utilizan operadores sobrecargados, especialmente en clases que implementan lógica interna compleja, el riesgo se amplifica. La sobrecarga de operadores puede introducir efectos laterales inadvertidos si no se revisa adecuadamente su semántica. Por ello, se recomienda evitar incluir cambios de estado dentro de operadores booleanos sobrecargados y optar por métodos explícitos que describan claramente su propósito. Además, cuando se trabaja con punteros inteligentes, referencias a objetos gestionados o contenedores de la STL, resulta esencial evitar que la evaluación del operando derecho desencadene alteraciones en la estructura interna que dependa implícitamente del orden de evaluación.

Otro aspecto crítico aparece al usar funciones lambdas, especialmente cuando capturan variables por referencia. Si la lambda situada a la derecha del operador lógico modifica variables externas, el comportamiento resultante puede ser impredecible según si se ejecuta o no. Esto agrava la posibilidad de generar condiciones de carrera o estados inconsistentes, sobre todo en aplicaciones paralelas.

Riesgos asociados

- Comportamiento inesperado de la aplicación debido a la evaluación condicional no garantizada del operando derecho.
- Inyección de errores lógicos difíciles de detectar y depurar.
- Generación de condiciones de carrera o estados inconsistentes, si el operando derecho modifica el estado global o variables compartidas.
- Posible pérdida de datos o corrupción de datos si el operando derecho produce cambios persistentes y no es evaluado por la lógica corta.
- Aumento de la complejidad del código y dificultad en el mantenimiento debido a los efectos colaterales ocultos.
- Problemas de portabilidad, dado que la dependencia de los efectos laterales puede tener diferentes resultados en diferentes compiladores o versiones del

lenguaje.

- Introducción de vulnerabilidades de seguridad al dejar variables en estados inseguros o inconsistentes.

Buenas prácticas

- Evitar llamadas a funciones con efectos laterales persistentes en el operando derecho de `&&` y `||`; evaluar primero la función y almacenar el resultado.

```
//-- Llamada con efecto lateral dentro del operando derecho
bool validar();
bool registrarAcceso();

if (validar() && registrarAcceso()) {
  procesar();
}
```

```
//-- Evaluación previa separando el efecto lateral
bool esValido = validar();
bool accesoRegistrado = false;
if (esValido) {
  accesoRegistrado = registrarAcceso();
}
if (esValido && accesoRegistrado) {
  procesar();
}
```

- No realizar asignaciones a variables globales o estáticas en el operando derecho; efectuar la asignación en una instrucción independiente.

```
//-- Asignación con efecto persistente en el operando derecho
bool autenticado = false;
bool verificarCredenciales();

if (verificarCredenciales() || (autenticado = true)) {
  accederSistema();
}
```

```
//-- Separación explícita de la asignación
bool autenticado = false;
bool credencialesValidas = verificarCredenciales();
if (!credencialesValidas) {
  autenticado = true;
}
if (credencialesValidas || autenticado) {
  accederSistema();
```

```
}
```

- Evitar modificar contadores o estados compartidos en el operando derecho; realizar las modificaciones fuera de la expresión lógica.

```
//-- Incremento con efecto lateral en el operando derecho
int intentos = 0;
bool limiteSuperado();

if (limiteSuperado() && (++intentos < 3)) {
  reintentar();
}
```

```
//-- Incremento controlado en bloque separado
int intentos = 0;
bool superaLimite = limiteSuperado();
if (superaLimite) {
  ++intentos;
}
if (superaLimite && (intentos < 3)) {
  reintentar();
}
```

- No invocar operaciones que modifiquen recursos externos (archivos, base de datos, red) en el operando derecho; aislar dichas operaciones en bloques explícitos.

```
//-- Escritura en recurso externo dentro del operando derecho
bool conexionActiva();
bool guardarLog();

if (conexionActiva() || guardarLog()) {
  notificar();
}
```

```
//-- Operación externa realizada fuera de la expresión lógica
bool activa = conexionActiva();
bool logGuardado = false;
if (!activa) {
  logGuardado = guardarLog();
}
if (activa || logGuardado) {
  notificar();
}
```

2.6.4.2 Estándares metodológicos

- **MISRA CPP-8.14.1**: RHS of logical &&/|| should not have persistent side effects.

2.6.5 [cpp.exp.5] - Evitar copiar datos entre objetos solapados

ID	cpp.exp.5	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

En las operaciones de copia se debe evitar el uso de funciones o construcciones que lean y escriban datos en regiones de memoria que se solapan, ya que esto puede generar comportamientos indefinidos, corrupción de datos y vulnerabilidades explotables. Para ello se deben emplear funciones seguras capaces de gestionar solapamientos o rediseñar la lógica de copia para garantizar que las áreas de origen y destino no se interfieran.

Codificación insegura

Manipulación de un buffer de caracteres donde se desplaza parte del contenido dentro del mismo array. Se busca entender qué ocurre cuando se copian datos entre regiones de memoria que se solapan y cómo ciertas funciones no garantizan un comportamiento definido en ese escenario.

```
#include <iostream>
#include <cstring>

int main() {
  //-- Buffer inicial con texto
  char buffer[20] = "ABCDEFGHIJ";

  //-- Se intenta desplazar el contenido 2 posiciones a la derecha
  //-- memcpy no garantiza comportamiento correcto si hay solapamiento
  std::memcpy(buffer + 2, buffer, 8);

  //-- Mostrar resultado
  std::cout << "Resultado: " << buffer << std::endl;
  return 0;
}
```

Codificación recomendada

Manipulación del mismo escenario de desplazamiento dentro de un buffer, asegurando que la copia entre regiones solapadas se realiza con una función que gestiona correctamente el orden interno de copia para evitar comportamiento indefinido.

```
#include <iostream>
#include <cstring>

int main() {
  //-- Buffer inicial con texto
  char buffer[20] = "ABCDEFGHIJ";

  //-- Se desplaza el contenido 2 posiciones a la derecha
  //-- memmove gestiona correctamente regiones de memoria solapadas
  std::memmove(buffer + 2, buffer, 8);

  //-- Mostrar resultado
  std::cout << "Resultado: " << buffer << std::endl;
  return 0;
}
```

2.6.5.1 Descripción

Las operaciones de copia entre regiones de memoria solapadas constituyen una de las fuentes más habituales de **comportamiento indefinido**. En entornos de bajo nivel donde se manejan punteros de forma directa o estructuras complejas con asignadores personalizados, el riesgo aumenta sustancialmente, dado que el estándar del lenguaje no garantiza un orden determinista de lectura y escritura cuando el origen y el destino se superponen. Este requisito se orienta a evitar tales patrones y a promover el uso de funciones, algoritmos y diseños que preserven tanto la integridad como la coherencia de los datos.

El problema surge típicamente cuando se emplean funciones como `memcpy`, `std::copy` o equivalentes con regiones parcialmente coincidentes. En estos casos, el comportamiento puede variar entre compiladores, arquitecturas y niveles de optimización, provocando desde fallos difíciles de depurar hasta vulnerabilidades explotables por atacantes. En particular, un atacante podría manipular offsets o tamaños para inducir solapamientos intencionados que generen corrupción de memoria, abriendo la puerta a sobreescritura de estructuras internas, bypass de controles lógicos o incluso ejecución de código malicioso.

Para mitigar estos riesgos se deben adoptar prácticas de codificación que garanticen que ninguna operación de copia conlleve solapamiento no controlado. La regla principal consiste en usar **funciones explícitamente seguras para solapamientos**, como `memmove`, cuando se requieran movimientos dentro del mismo buffer. Además, se recomienda un rediseño cuidadoso de estructuras de datos o algoritmos, de modo que la copia se realice en secuencias bien definidas que impidan cualquier superposición accidental entre origen y destino.

El acceso directo a memoria permite construir soluciones de alto rendimiento, pero también introduce riesgos significativos si no se aplica un control estricto. Un

solapamiento entre regiones ocurre cuando parte del rango de destino coincide con parte del rango fuente. En estas situaciones, el orden en que se leen y escriben bytes se vuelve crítico. Si la escritura se adelanta a la lectura de bytes aún no copiados, se destruye información necesaria y se produce un estado inconsistente del objeto. Este efecto cascada puede alterar la semántica de clases, invalidar iteradores o incluso sobrepasar límites seguros, generando fallos de memoria.

Adicionalmente, en sistemas con requisitos estrictos de rendimiento, puede resultar necesario analizar el impacto de sustituir funciones como `memcpy` por alternativas más seguras. Sin embargo, incluso en estos casos, la integridad de los datos y la estabilidad del sistema deben prevalecer sobre micro-optimizaciones prematuras. El uso de análisis estático, sanitizadores de memoria y pruebas exhaustivas bajo condiciones de estrés es esencial para verificar que las rutas de copia se comportan de manera segura en todos los escenarios posibles.

Riesgos asociados

- Corrupción de datos debido a la superposición de memoria fuente y destino.
- Comportamiento indefinido, lo que puede llevar a fallos de la aplicación.
- Introducción de vulnerabilidades de seguridad explotables, como escritura fuera de límites.
- Pérdida de información o truncamiento de datos durante la copia.
- Dificultad para rastrear y depurar errores relacionados con la gestión de memoria.
- Potencial ejecución de código malicioso si la corrupción de datos es manipulada intencionadamente.

Buenas prácticas

- Utilizar `memmove` en lugar de `memcpy` cuando exista posibilidad de solapamiento de memoria.

```
//-- Copia directa sin considerar solapamiento
char buffer[10] = "abcdef";
memcpy(buffer + 2, buffer, 4);
```

```
//-- Copia segura que maneja regiones solapadas
char buffer2[10] = "abcdef";
memmove(buffer2 + 2, buffer2, 4);
```

- Preferir `std::copy_backward` cuando el rango destino se encuentra dentro del rango origen.

```
//-- Copia hacia adelante con rangos solapados
std::vector v = {1,2,3,4,5};
std::copy(v.begin(), v.begin() + 3, v.begin() + 1);
```

```
//-- Copia hacia atrás evitando sobrescritura prematura
std::vector v2 = {1,2,3,4,5};
std::copy_backward(v2.begin(), v2.begin() + 3, v2.begin() + 4);
```

- Validar explícitamente que los rangos origen y destino no se solapan antes de realizar copias manuales con punteros.

```
//-- Copia manual sin verificación de solapamiento
int data[5] = {1,2,3,4,5};
int* src = data;
int* dst = data + 1;
for(int i = 0; i < 3; ++i) {
  dst[i] = src[i];
}
```

```
//-- Verificación previa del rango antes de copiar
int data2[5] = {1,2,3,4,5};
int* src2 = data2;
int* dst2 = data2 + 1;
if(dst2 >= src2 + 3 || dst2 + 3 <= src2) {
  for(int i = 0; i < 3; ++i) {
    dst2[i] = src2[i];
  }
} else {
  memmove(dst2, src2, 3 * sizeof(int));
}
```

- Evitar el uso de funciones de bajo nivel cuando existan abstracciones seguras de la STL que gestionen correctamente los rangos.

```
//-- Manipulación directa de buffer interno con riesgo de solapamiento
std::string s = "abcdef";
memcpy(&s[2], &s[0], 4);
```

```
//-- Uso de métodos seguros de la clase para reorganizar contenido
std::string s2 = "abcdef";
s2.insert(2, s2.substr(0,4));
```

2.6.5.2 Estándares metodológicos

- **MISRA CPP-8.18.1**: No copying to overlapping object.

2.6.6 [cpp.exp.6] - Evitar utilizar el operador "coma"

ID	cpp.exp.6	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Design

El uso del operador "coma" puede introducir comportamientos confusos y difíciles de rastrear, afectando la claridad del flujo de ejecución y aumentando el riesgo de errores lógicos que deriven en fallos de seguridad. Este requisito promueve evitar su utilización en expresiones complejas, especialmente dentro de bucles o condiciones, donde su semántica puede ser interpretada incorrectamente.

Codificación insegura

Uso del operador "coma" para ejecutar múltiples expresiones en una sola sentencia, especialmente en condiciones o bucles. Se busca mostrar cómo esta práctica reduce la claridad del código, dificulta el mantenimiento y puede introducir errores lógicos al evaluar expresiones de forma poco evidente.

```
#include <iostream>

int main() {
  //-- Declaración de variables
  int a = 5;
  int b = 10;
  int resultado = 0;

  //-- Uso del operador "coma" en una condición
  if ((resultado = a + b, resultado > 10)) {
    //-- Se imprime el resultado si es mayor que 10
    std::cout << "Resultado mayor que 10: " << resultado << std::endl;
  }

  //-- Uso del operador "coma" en un bucle for
  for (int i = 0, j = 5; i < j; i++, j--) {
    //-- Se muestran los valores actuales
    std::cout << "i: " << i << ", j: " << j << std::endl;
  }
  return 0;
}
```

Codificación recomendada

Separación explícita de instrucciones para evitar el operador "coma", mejorando la claridad, mantenibilidad y reduciendo ambigüedades en la evaluación de expresiones. Se busca que cada operación tenga una intención clara y fácil de auditar.

```
#include <iostream>

int main() {
  //-- Declaración de variables
  int a = 5;
  int b = 10;
  int resultado = 0;

  //-- Cálculo explícito del resultado
  resultado = a + b;

  //-- Evaluación clara de la condición
  if (resultado > 10) {
    //-- Se imprime el resultado si es mayor que 10
    std::cout << "Resultado mayor que 10: " << resultado << std::endl;
  }

  //-- Inicialización separada de variables de control
  int i = 0;
  int j = 5;

  //-- Bucle con actualizaciones claras y separadas
  while (i < j) {
    //-- Se muestran los valores actuales
    std::cout << "i: " << i << ", j: " << j << std::endl;

    //-- Actualización explícita de variables
    i++;
    j--;
  }
  return 0;
}
```

2.6.6.1 Descripción

El operador "coma" ejecuta sus operandos de izquierda a derecha y devuelve el valor del último operando. Aunque forma parte del lenguaje desde sus inicios, su empleo suele generar ambigüedad y opacidad en el código, afectando directamente la **integridad** de la lógica programada y reduciendo la capacidad de detectar fallos. Este requisito recomienda evitar su uso salvo casos extremadamente justificados y explícitos. La claridad es un principio fundamental del desarrollo seguro, y el operador "coma" rara vez contribuye positivamente a ella.

Desde una perspectiva de seguridad, el operador "coma" es especialmente problemático en contextos donde la intención del programador debe ser inequívoca. Su semántica puede provocar que variables cambien de valor en momentos inesperados o que algunas expresiones se evalúen sin que el programador lo haya advertido,

introduciendo errores que pueden afectar la lógica de validaciones, cálculos críticos y secuencias de inicialización. En sistemas que dependen de estados coherentes o cálculos deterministas, estas alteraciones mínimas pueden escalar hasta producir fallos de seguridad.

Aunque el operador "coma" es parte del estándar del lenguaje y tiene usos válidos, la recomendación general en contextos de desarrollo seguro es **evitar emplearlo salvo en casos estrictamente necesarios**, documentando su uso de forma explícita y asegurándose de que no afecte la integridad ni claridad del software.

Riesgos asociados

- Confusión en la lógica del programa, dificultando el mantenimiento y la comprensión del código.
- Posibilidad de introducir errores sutiles por la evaluación secuencial de expresiones, lo que puede alterar el flujo programado.
- Saltarse intencionalmente valores o asignaciones, lo que puede causar inicializaciones incorrectas o no deseadas.
- Complicaciones al depurar el código, ya que los errores generados por el uso indebido del operador "coma" suelen ser difíciles de identificar.
- Código propenso a errores en refactorizaciones o evoluciones posteriores, donde el uso del operador "coma" puede pasar desapercibido.
- Aumento de la superficie para vulnerabilidades de seguridad, especialmente si el flujo de ejecución inesperado es explotable.

Buenas prácticas

- Evitar el uso del operador "coma" para ejecutar múltiples expresiones en una sola sentencia; separar las instrucciones mejora claridad y mantenibilidad.

```
//-- Uso del operador "coma" en una misma sentencia
int a = 0, b = 0;
a = 5, b = 10;
```

```
//-- Separación explícita de instrucciones
int a = 0;
int b = 0;
a = 5;
b = 10;
```

- No utilizar el operador "coma" en estructuras de control; dividir las operaciones previas en sentencias independientes.

```
//-- Inicialización y condición combinadas con operador "coma"
int x = 0;
if (x = obtenerValor(), x > 10) {
  procesar();
}
```

```
//-- Separación de la asignación y la evaluación condicional
int x = obtenerValor();
if (x > 10) {
  procesar();
}
```

- Evitar el operador "coma" en bucles para múltiples efectos laterales; encapsular la lógica en el cuerpo del bucle.

```
//-- Incrementos múltiples usando operador "coma"
for (int i = 0, j = 10; i < j; i++, j--) {
  usar(i, j);
}
```

```
//-- Control explícito de variables dentro del bucle
int i = 0;
int j = 10;
for (; i < j;) {
  usar(i, j);
  i++;
  j--;
}
```

- No emplear el operador "coma" en expresiones de retorno; retornar valores explícitos mejora la legibilidad y evita ambigüedades.

```
//-- Retorno usando operador "coma"
int funcion() {
  int x = 5;
  return x++, x;
}
```

```
//-- Operaciones separadas antes del retorno
int funcion() {
  int x = 5;
  x++;
  return x;
}
```

2.6.6.2 Estándares metodológicos

- **MISRA CPP-8.19.1**: Comma operator should not be used.

2.7 Aritmética

2.7.1 [cpp.arit.1] - Evitar mezclar aritméticas con signo y sin signo

ID	cpp.arit.1	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Establece la necesidad de impedir que en el código se mezclen valores o expresiones aritméticas con signo y sin signo, ya que esta práctica puede provocar conversiones implícitas peligrosas, desbordamientos inesperados, pérdidas de precisión y comportamientos no evidentes para el desarrollador. Aplicar este control permite mejorar la **integridad** del software, evitar errores lógicos difíciles de detectar y reducir la superficie de vulnerabilidades relacionadas con cálculos incorrectos, validaciones fallidas o comparaciones inseguras, manteniendo así un código más seguro, estable y consistente.

Codificación insegura

Se muestra una situación donde se comparan y operan valores con signo y sin signo en una condición y en una operación aritmética. El objetivo es observar cómo las conversiones implícitas pueden generar resultados inesperados, especialmente cuando se manejan valores negativos.

```
#include <iostream>

//-- Función principal
int main() {
  //-- Variable con signo que puede tomar valores negativos
  int cantidad = -1;

  //-- Variable sin signo que representa un tamaño o contador
  unsigned int limite = 10;

  //-- Comparación entre entero con signo y sin signo
  if (cantidad < limite) {
    //-- Se espera que -1 sea menor que 10
    std::cout << "cantidad es menor que limite" << std::endl;
  } else {
    //-- Resultado inesperado por conversión implícita a unsigned
    std::cout << "cantidad NO es menor que limite" << std::endl;
  }
```

```
  //-- Operación aritmética mezclando tipos
  unsigned int resultado = limite + cantidad;

  //-- Se imprime el resultado convertido implícitamente
  std::cout << "Resultado: " << resultado << std::endl;
  return 0;
}
```

Codificación recomendada

Se muestra la misma lógica evitando conversiones implícitas entre tipos con y sin signo. El objetivo es mantener coherencia en los tipos utilizados y validar explícitamente los valores antes de realizar comparaciones u operaciones aritméticas.

```
#include <iostream>

//-- Función principal
int main() {
  //-- Variable con signo que puede tomar valores negativos
  int cantidad = -1;

  //-- Variable con el mismo tipo para evitar conversiones implícitas
  int limite = 10;

  //-- Validación explícita antes de operar con valores negativos
  if (cantidad >= 0 && cantidad < limite) {
    //-- Comparación segura entre tipos homogéneos
    std::cout << "cantidad es valida y menor que limite" << std::endl;
  } else {
    //-- Manejo controlado de valores fuera de rango
    std::cout << "cantidad es negativa o mayor que limite" << std::endl;
  }

  //-- Operación aritmética entre tipos consistentes
  int resultado = 0;
  if (cantidad >= 0) {
    resultado = limite + cantidad;
  }

  //-- Se imprime el resultado de forma controlada
  std::cout << "Resultado: " << resultado << std::endl;
  return 0;
}
```

2.7.1.1 Descripción

Cuando un valor con signo se combina con uno sin signo en una operación aritmética o comparativa, C++ realiza una conversión implícita del valor con signo hacia un tipo sin

signo. Esta conversión puede transformar un número negativo en un valor extremadamente grande, alterando la lógica esperada y generando fallos de integridad que pueden derivar en comportamientos incorrectos del programa, comprobaciones de límites fallidas, bucles infinitos o accesos indebidos a memoria.

El desarrollador puede creer que está comparando un valor potencialmente negativo con un límite superior, cuando en realidad la conversión a tipo sin signo hace que la comparación sea siempre verdadera, o siempre falsa. Por ello, es fundamental comprender que **las conversiones automáticas en C++ nunca deben considerarse un mecanismo seguro** cuando están involucrados tipos con signo y sin signo.

Las consecuencias de ignorar este control pueden extenderse más allá de simples errores lógicos. En contextos de seguridad, una comparación incorrecta puede permitir que un valor supuestamente inválido sea aceptado, que se produzcan desbordamientos aritméticos o que se acceda a posiciones de memoria fuera de rango. Estos escenarios pueden aprovecharse para provocar fallos programados, omitir validaciones o comprometer la integridad de estructuras internas.

Riesgos asociados

- Desbordamientos silenciosos al realizar operaciones entre variables signed y unsigned.
- Resultados inesperados debido a la conversión implícita de tipos (por ejemplo, un valor negativo convertido a unsigned se transforma en un valor muy grande).
- Vulnerabilidades de seguridad por errores en validaciones de límites o rangos (p.ej., posibles desbordamientos de buffer o accesos fuera de límites de arrays).
- Pérdida de información al convertir valores negativos en enteros unsigned.
- Comportamiento indefinido o dependiente del compilador, dificultando la portabilidad y mantenimiento del código.
- Dificultad en el análisis y revisión de código debido a resultados no intuitivos en operaciones y comparaciones.
- Explotación por parte de atacantes para evadir controles de seguridad basados en cálculos de tamaño, condiciones o índices.

Buenas prácticas

- Usar tipos coherentes en operaciones aritméticas, evitando mezclar variables con y sin signo en la misma expresión.

```
//-- Mezcla de int y unsigned int en comparación aritmética
int a = -1;
unsigned int b = 10;
if (a < b) {
```

```
  /* ... */
}
```

```
//-- Uso de tipos homogéneos con signo para comparación consistente
int a2 = -1;
int b2 = 10;
if (a2 < b2) {
  /* ... */
}
```

- Realizar conversiones explícitas y controladas cuando sea imprescindible combinar tipos con y sin signo.

```
//-- Conversión implícita de valor negativo a unsigned
int size = -5;
unsigned int total = size + 10;
```

```
//-- Validación previa y conversión explícita controlada
int size2 = -5;
if (size2 >= 0) {
  unsigned int total2 = static_cast(size2) + 10;
}
```

- Evitar el uso de tipos sin signo para valores que puedan representar estados negativos o errores.

```
//-- Uso de unsigned para función que puede devolver error
unsigned int getBalance();
unsigned int balance = getBalance();
if (balance < 0) {
  /* manejo de error */
}
```

```
//-- Uso de tipo con signo para permitir valores negativos de error
int getBalance2();
int balance2 = getBalance2();
if (balance2 < 0) {
  /* manejo de error */
}
```

- Controlar cuidadosamente las comparaciones entre tamaños (size_t) y enteros con signo en bucles.

```
//-- Comparación entre size_t e int con posible conversión implícita
std::vector v = {1,2,3};
```

```
for (int i = 0; i < v.size(); ++i) {
  /* ... */
}
```

```
//-- Uso de size_t coherente con el tipo devuelto por size()
std::vector v2 = {1,2,3};
for (std::size_t i = 0; i < v2.size(); ++i) {
  /* ... */
}
```

2.7.1.2 Estándares metodológicos

- **C++ Core-ES.100**: don't mix signed and unsigned arithmetic.

2.7.2 [cpp.arit.2] - Evitar condiciones de overflow

ID	cpp.arit.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se trata de impedir que operaciones aritméticas o de memoria excedan los límites válidos de los tipos o estructuras utilizadas, lo que podría provocar comportamientos indefinidos, corrupción de datos o vulnerabilidades explotables, así como reforzar el uso de tipos seguros, verificaciones explícitas de rango, utilidades estándar como `std::numeric_limits` y funciones seguras para gestionar memoria y cálculos. El objetivo es reducir errores sutiles derivados de conversiones implícitas, cálculos intermedios incorrectos o asignaciones no verificadas, garantizando así código más robusto y resistente frente a ataques que aprovechan desbordamientos.

Codificación insegura

Se muestra una suma de enteros donde se manejan valores cercanos al límite máximo del tipo `int`. La intención es evidenciar cómo una operación aritmética sin validación previa puede producir un desbordamiento y generar resultados incorrectos o indefinidos.

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

int main() {
  //-- Se definen dos valores grandes cercanos al límite de int
  int a = std::numeric_limits<int>::max();
  int b = 1;

  //-- Se realiza la suma sin validar posibles desbordamientos
```

```
  int resultado = a + b;

  //-- Se muestra el resultado, que puede ser incorrecto por overflow
  std::cout << "Resultado: " << resultado << std::endl;
  return 0;
}
```

Codificación recomendada

Se aborda la misma operación de suma de enteros, incorporando una validación previa para comprobar si la operación excederá los límites del tipo `int`. Se pretende demostrar cómo prevenir el desbordamiento mediante comprobaciones explícitas antes de realizar la operación.

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

bool sumaSegura(int a, int b, int& resultado) {
  //-- Verifica si la suma excede el límite superior
  if (b > 0 && a > std::numeric_limits<int>::max() - b) {
    return false;
  }

  //-- Verifica si la suma excede el límite inferior
  if (b < 0 && a < std::numeric_limits<int>::min() - b) {
    return false;
  }

  //-- Si las validaciones pasan, se realiza la suma
  resultado = a + b;
  return true;
}

int main() {
  //-- Se definen dos valores grandes cercanos al límite de int
  int a = std::numeric_limits<int>::max();
  int b = 1;
  int resultado = 0;

  //-- Se intenta realizar la suma de forma segura
  if (sumaSegura(a, b, resultado)) {
    std::cout << "Resultado: " << resultado << std::endl;
  } else {
    //-- Se informa que la operación produciría overflow
    std::cout << "Error: la suma produce overflow" << std::endl;
  }
  return 0;
```

```
}
```

2.7.2.1 Descripción

Los desbordamientos pueden clasificarse en dos grupos principales: **desbordamientos aritméticos** y **desbordamientos de memoria**. El manejo incorrecto de límites numéricos y memoria es una de las fuentes más frecuentes de vulnerabilidades en C++, especialmente debido a la combinación de operaciones aritméticas sin comprobaciones, gestión manual de memoria y ausencia de protecciones automáticas en el lenguaje. Este requisito trata de evitar que se produzcan desbordamientos tanto aritméticos como de memoria, aplicando verificaciones explícitas, uso de tipos y funciones seguras y una disciplina de programación orientada a minimizar comportamientos indefinidos. El objetivo es mitigar los fallos tradicionales asociados al acceso fuera de rango, las conversiones incorrectas y la sobreasignación o subasignación de buffers, que pueden derivar en corrupción de memoria o ejecución arbitraria.

Un enfoque recomendado consiste en analizar los valores antes de realizar la operación. La biblioteca estándar facilita la verificación mediante `std::numeric_limits`. Estas comprobaciones deben aplicarse antes de sumas, multiplicaciones o incrementos utilizados para gestionar tamaños, índices o capacidades.

Los desbordamientos de memoria suelen aparecer al escribir más datos de los disponibles en un array, buffer o estructura. Para prevenir estos fallos es crucial priorizar estructuras seguras que gestionen su propio tamaño, como `std::vector` y `std::string`. Cuando sea necesario usar arrays o punteros crudos, deben realizarse comprobaciones explícitas antes de cualquier acceso o copia.

La STL proporciona herramientas que permiten gestionar memoria de forma más robusta. Por ejemplo, `std::vector::at` lanza una excepción si el índice está fuera de rango. También existen clases como `std::span`, que permiten manejar vistas seguras sobre datos sin perder información del tamaño.

Los compiladores modernos pueden optimizar código de formas que asumen ausencia de desbordamientos, lo que puede ocultar errores. Por ejemplo, se eliminan comprobaciones si el compilador infiere que un valor jamás será negativo. Por ello deben emplearse opciones que refuercen la detección de condiciones peligrosas, como `-fsanitize=undefined` o `-ftrapv`.

Riesgos asociados

- Ejecutar código incorrecto debido a resultados aritméticos imprevistos (comportamiento indefinido).
- Vulnerabilidades de seguridad que permiten la explotación por parte de atacantes (ej. ejecución de código arbitrario).
- Corrupción de memoria y datos, afectando la estabilidad y fiabilidad del sistema.

- Caídas inesperadas (crash) de la aplicación.
- Pérdida o modificación de información crítica por cálculos erróneos.
- Desviación del flujo de control, permitiendo saltos a instrucciones no deseadas.
- Posibilidad de denegación de servicio debido a fallos repetidos.
- Dificultad en el diagnóstico y depuración de errores.

Buenas prácticas

- Validar explícitamente los límites antes de realizar operaciones aritméticas con enteros.

```
//-- Suma sin validar límites puede desbordar
int a = INT_MAX;
int b = 1;
int c = a + b;
```

```
//-- Validación previa contra el límite máximo permitido
int a = INT_MAX;
int b = 1;
if (a > INT_MAX - b) {
  throw std::overflow_error("overflow");
}
int c = a + b;
```

- Utilizar tipos de datos con mayor capacidad cuando el rango esperado pueda superar el tipo original.

```
//-- Uso de tipo con rango insuficiente
int total = 1000000;
int factor = 3000;
int resultado = total * factor;
```

```
//-- Uso de tipo con mayor capacidad para evitar desbordamiento
long long total = 1000000;
long long factor = 3000;
long long resultado = total * factor;
```

- Comprobar límites antes de multiplicaciones, ya que son propensas a overflow silencioso.

```
//-- Multiplicación sin verificación previa
int x = 50000;
int y = 50000;
```

```
int z = x * y;
```

```
//-- Verificación previa usando división inversa
int x = 50000;
int y = 50000;
if (x != 0 && y > INT_MAX / x) {
  throw std::overflow_error("overflow");
}
int z = x * y;
```

- Usar funciones o utilidades seguras que detecten overflow en tiempo de ejecución.

```
//-- Operación aritmética directa sin detección
int a = 2000000000;
int b = 2000000000;
int c = a + b;
```

```
//-- Uso de función que detecta overflow (C++20)
int a = 2000000000;
int b = 2000000000;
int c;
if (__builtin_add_overflow(a, b, &c)) {
  throw std::overflow_error("overflow");
}
```

- Evitar conversiones implícitas entre tipos con distinto signo o tamaño que puedan provocar desbordamientos inesperados.

```
//-- Conversión implícita que puede provocar desbordamiento
unsigned int size = 4000000000U;
int signedSize = size;
```

```
//-- Validación explícita antes de convertir
unsigned int size = 4000000000U;
if (size > static_cast(INT_MAX)) {
  throw std::overflow_error("overflow");
}
int signedSize = static_cast(size);
```

2.7.2.2 Estándares metodológicos

- **C++ Core-ES.103**: don't overflow.
- **CERT/CPP-CTR52**: Guarantee that library functions do not overflow.

2.7.3 [cpp.arit.3] - Evitar condiciones de underflow

ID	cpp.arit.3	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Prevenir de forma sistemática cualquier condición de **underflow** al manipular tipos numéricos evita que operaciones aritméticas inseguras generen resultados inesperados, violaciones de integridad de datos, errores lógicos y vulnerabilidades explotables cuando el valor resultante es inferior al mínimo representable por el tipo. La mitigación requiere aplicar verificaciones explícitas, tipos seguros, funciones estándar y prácticas rigurosas de validación previa al cálculo. Este control es especialmente importante en código crítico, cálculos de índices, operaciones con memoria y transformaciones matemáticas dependientes de límites.

Codificación insegura

Gestión de un contador sin validar límites inferiores. Se muestra cómo una operación de decremento sobre un tipo sin signo puede provocar un underflow cuando el valor llega a cero, generando resultados inesperados y posibles fallos lógicos en la aplicación.

```
#include <iostream>

int main() {
  //-- Se define un contador sin signo inicializado en cero
  unsigned int contador = 0;

  //-- Se decrementa el contador sin validar el límite inferior
  contador--;

  //-- Se imprime el valor resultante tras el underflow
  std::cout << "Valor del contador: " << contador << std::endl;
  return 0;
}
```

Codificación recomendada

Control seguro de un contador validando el límite inferior antes de realizar el decremento. Se aprende a prevenir el underflow mediante comprobaciones explícitas que garantizan que la operación es válida antes de ejecutarse.

```
#include <iostream>

int main() {
  //-- Se define un contador sin signo inicializado en cero
  unsigned int contador = 0;
```

```
  //-- Se valida que el contador sea mayor que cero antes de decrementar
  if (contador > 0) {
    contador--;
  } else {
    //-- Se informa que no es posible decrementar para evitar underflow
    std::cout << "No se puede decrementar: el contador ya esta en cero" <<
std::endl;
  }
  //-- Se imprime el valor final del contador
  std::cout << "Valor del contador: " << contador << std::endl;
  return 0;
}
```

2.7.3.1 Descripción

El underflow numérico ocurre cuando el resultado de una operación aritmética queda por debajo del valor mínimo representable para un tipo numérico concreto. Este fenómeno puede aparecer en cálculos con enteros, operaciones con punteros, manipulación de índices de contenedores y en algoritmos donde se asumen rangos estrictos. Un underflow silencioso puede provocar valores inesperados debido a la naturaleza modular de los enteros, o generar comportamientos indefinidos en escenarios donde el estándar lo especifica de esa forma. Para garantizar un desarrollo seguro, es imprescindible aplicar verificaciones preventivas, utilizar tipos ajustados al rango necesario y apoyarse en las funciones de la biblioteca estándar diseñadas para evitar desbordamientos y underflows.

El underflow puede comprometer la integridad del programa y producir efectos colaterales que se extienden más allá de la operación numérica inicial. Entre los riesgos más destacados se encuentran errores de cálculo, acceso fuera de límites en estructuras de datos, vulnerabilidades de control de flujo, bucles infinitos y errores de seguridad al manipular tamaños o desplazamientos. Un atacante podría aprovechar un cálculo numérico insuficientemente validado para provocar una indexación negativa reinterpretada como un valor masivo no intencionado en tipos sin signo. Por ello, es esencial evitar cualquier dependencia en comportamientos indefinidos o implícitos del lenguaje cuando se realizan operaciones cercanas a los límites numéricos.

C++ ofrece utilidades esenciales que permiten detectar y evitar condiciones de límite. La plantilla `std::numeric_limits` proporciona valores mínimo y máximo para cada tipo, lo que permite realizar verificaciones seguras antes de cualquier operación. Igualmente, `std::clamp` permite limitar valores dentro de rangos seguros sin necesidad de escribir verificaciones repetitivas. Estas herramientas deben integrarse sistemáticamente en el código que interactúa con datos potencialmente cercanos a límites numéricos.

Además, las funciones de verificación y los algoritmos basados en comprobaciones explícitas mejoran la trazabilidad del código al documentar claramente las expectativas

respecto a los valores permitidos. El uso de **programación defensiva** es especialmente recomendable en calculadoras de tamaño, índices y operaciones relacionadas con memoria. Cualquier cálculo que afecte a desplazamientos, asignaciones o dimensionamiento de buffers debe incorporar controles explícitos para evitar valores negativos o no representables en su tipo asociado.

Riesgos asociados

- Desbordamientos negativos (underflow) pueden permitir la obtención de valores inesperados, provocando fallos lógicos en el programa.
- Errores de validación en comprobaciones de límites, provocando accesos fuera de rango en arrays o buffers.
- Vulnerabilidades de seguridad explotables, como ejecución de código arbitrario o escalado de privilegios.
- Elusión de controles de seguridad, ya que los underflow pueden evadir condiciones de verificación basadas en límites inferiores.
- Pérdida o corrupción de datos debido a cálculos erróneos derivados de valores desbordados.
- Condiciones para ataques de denegación de servicio (DoS) al hacer que el software se comporte de forma indefinida o impredecible.
- Posibilidad de que el programa se bloquee o produzca resultados no deterministas, afectando la estabilidad general de la aplicación.

Buenas prácticas

- Validar límites antes de realizar operaciones aritméticas que puedan provocar underflow en tipos enteros sin signo.

```
//-- Resta sin validar puede provocar underflow en unsigned
unsigned int total = 10;
unsigned int descuento = 20;
unsigned int resultado = total - descuento;
```

```
//-- Validación previa para evitar resultado fuera de rango
unsigned int total2 = 10;
unsigned int descuento2 = 20;
unsigned int resultado2 = 0;
if (descuento2 <= total2) {
  resultado2 = total2 - descuento2;
}
```

- Evitar el uso de tipos sin signo cuando la lógica permita valores negativos intermedios.

```
//-- Uso de unsigned provoca underflow al decrementar bajo cero
unsigned int contador = 0;
contador--;
```

```
//-- Uso de tipo con signo para representar correctamente valores negativos
int contador2 = 0;
contador2--;
```

- Comprobar resultados antes de decrementar variables usadas como índices o tamaños.

```
//-- Decremento directo del índice puede generar valor máximo por underflow
size_t indice = 0;
indice--;
```

```
//-- Validación del índice antes de decrementar
size_t indice2 = 0;
if (indice2 > 0) {
  indice2--;
}
```

- Utilizar funciones o utilidades seguras para operaciones aritméticas con control de límites.

```
//-- Multiplicación sin control puede generar underflow en cálculos posteriores
unsigned int saldo = 5;
unsigned int retiro = 10;
unsigned int nuevoSaldo = saldo - retiro;
```

```
//-- Uso de función de resta segura con verificación explícita
unsigned int saldo2 = 5;
unsigned int retiro2 = 10;
unsigned int nuevoSaldo2 = 0;
bool restaValida = (retiro2 <= saldo2);
if (restaValida) {
  nuevoSaldo2 = saldo2 - retiro2;
}
```

2.7.3.2 Estándares metodológicos

- **C++ Core-ES.104**: don't underflow.

2.7.4 [cpp.arit.4] - Evitar condiciones de división por cero

ID	cpp.arit.4	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este requisito establece la necesidad de incorporar comprobaciones explícitas, validaciones previas y mecanismos de manejo seguro de errores para impedir cualquier intento de división entre cero durante la ejecución. Este tipo de condición es una de las fuentes más comunes de fallos lógicos y vulnerabilidades explotables, especialmente cuando los valores provienen de entradas externas o cálculos intermedios no controlados. Mediante el uso de defensas consistentes, como verificaciones previas, manejadores centralizados y contratos de código, se reduce de forma significativa la superficie de ataque y el riesgo de interrupciones inesperadas.

Codificación insegura

Programa que solicita dos números al usuario y realiza una división directa entre ellos. La intención es mostrar cómo una operación aritmética aparentemente simple puede generar un fallo en tiempo de ejecución cuando no se validan las entradas, especialmente ante valores cero en el denominador.

```
#include <iostream>

int main() {
  //-- comentario
  //-- Solicita dos números y realiza una división sin validar el denominador
  double numerador = 0.0;
  double denominador = 0.0;

  std::cout << "Ingrese el numerador: ";
  std::cin >> numerador;

  std::cout << "Ingrese el denominador: ";
  std::cin >> denominador;

  //-- comentario
  //-- División directa sin comprobar si el denominador es cero
  double resultado = numerador / denominador;

  std::cout << "Resultado: " << resultado << std::endl;
  return 0;
}
```

Codificación recomendada

Programa que solicita dos números al usuario y realiza una división validando previamente el denominador. El objetivo es demostrar cómo prevenir errores de ejecución y comportamientos indefinidos mediante una verificación explícita antes de efectuar la operación.

```
#include <iostream>

int main() {
  //-- comentario
  //-- Solicita dos números y valida el denominador antes de dividir
  double numerador = 0.0;
  double denominador = 0.0;

  std::cout << "Ingrese el numerador: ";
  std::cin >> numerador;

  std::cout << "Ingrese el denominador: ";
  std::cin >> denominador;

  //-- comentario
  //-- Verifica que el denominador no sea cero antes de dividir
  if (denominador == 0.0) {
    std::cerr << "Error: division por cero no permitida." << std::endl;
    return 1;
  }

  double resultado = numerador / denominador;

  std::cout << "Resultado: " << resultado << std::endl;
  return 0;
}
```

2.7.4.1 Descripción

El control relativo a evitar divisiones entre cero constituye una medida esencial de integridad y robustez del software, dado que este lenguaje ofrece un control muy preciso sobre las operaciones aritméticas y, por consiguiente, deja al programador una mayor responsabilidad en la prevención de comportamientos indefinidos. El estándar del lenguaje no define el resultado de una división entre cero, lo cual abre la puerta a interrupciones súbitas, fallos de ejecución, excepciones de hardware, corrupción de datos y potenciales vectores de ataque cuando los valores divisores pueden ser manipulados desde el exterior.

En escenarios donde intervienen cálculos provenientes de sensores, archivos, parámetros de usuario o respuestas de sistemas remotos, resulta crítico validar de forma sistemática que el denominador sea distinto de cero y que no se encuentre dentro

de un rango considerado inestable o peligroso, por ejemplo, valores extremadamente próximos a cero en variables de tipo `double`. Además, cuando la lógica del negocio requiere manipular valores fraccionarios pequeños, se deben establecer umbrales y estrategias de manejo alternativo para evitar que condiciones de precisión insuficiente generen fallos lógicos o rutas no previstas. La falta de dichas validaciones puede comprometer la estabilidad del proceso, generar condiciones de denegación de servicio y, en casos muy específicos, permitir que un atacante fuerce la ejecución hacia estados corruptos.

El empleo de **bloques de verificación previos** es la medida más directa y efectiva. Estas comprobaciones deben integrarse como una parte natural de la rutina de validación de parámetros, evitando tanto el exceso de confianza en datos internos como la suposición de que ciertos valores no pueden ser cero. En desarrollos complejos, el uso de técnicas como **programación defensiva**, **contratos de código** y **aserciones explícitas** añade capas adicionales de prevención. Asimismo, es relevante que estas verificaciones no se limiten al punto de uso, sino que se replique un modelo de validación consistente a lo largo de toda la base de código para reforzar la coherencia.

Para entornos donde el rendimiento es crítico, como sistemas embebidos o de computación de alto rendimiento, las validaciones deben diseñarse de forma que minimicen el impacto en ciclos de CPU, pero sin sacrificar la seguridad. Es preferible incorporar verificaciones ligeras, pero sistemáticas, en lugar de omitirlas completamente. Cuando se manejen excepciones, resulta adecuado capturar condiciones relacionadas con aritmética ilegal y derivarlas a rutas seguras, generando registros auditables para análisis posteriores.

En sistemas que requieren cumplimiento estricto de normas de seguridad o niveles elevados de confiabilidad, es recomendable acompañar estas medidas con análisis estático especializado en detección de defectos aritméticos. Herramientas modernas de análisis para C++ permiten identificar rutas de ejecución donde un valor divisor podría aproximarse a cero sin control, facilitando la corrección temprana en etapas de desarrollo. Además, se pueden emplear pruebas unitarias con valores límite o fuzzing para maximizar la cobertura de escenarios atípicos donde la operación aritmética pueda verse comprometida. Esto permite no solo prevenir fallos sino mejorar la resiliencia global del software ante condiciones inesperadas.

Riesgos asociados

- Fallo en tiempo de ejecución de la aplicación.
- Comportamiento indefinido del programa.
- Vulnerabilidad ante ataques de denegación de servicio (DoS).
- Pérdida o corrupción de datos.
- Exposición a errores difíciles de depurar y mantener.

- Posible explotación para desestabilizar el sistema.
- Interrupción inesperada de servicios críticos.
- Potencial fuga de información a través de mensajes de error.

Buenas prácticas

- Validar explícitamente que el divisor no sea cero antes de realizar la operación.

```
//-- División directa sin validar el divisor
double dividir(double a, double b) {
  return a / b;
}
```

```
//-- Validación previa del divisor antes de dividir
double dividir(double a, double b) {
  if (b == 0.0) {
    throw std::invalid_argument("División por cero");
  }
  return a / b;
}
```

- Comprobar datos provenientes de entrada externa antes de usarlos en operaciones aritméticas.

```
//-- Uso directo de entrada externa sin validación
void procesar(int total, int cantidad) {
  int promedio = total / cantidad;
}
```

```
//-- Validación de entrada antes de realizar la división
void procesar(int total, int cantidad) {
  if (cantidad == 0) {
    return;
  }
  int promedio = total / cantidad;
}
```

- Implementar funciones utilitarias seguras para centralizar el control de divisiones.

```
//-- División repetida sin control centralizado
int calcularRatio(int a, int b) {
  return a / b;
}
```

```
//-- Función segura reutilizable que controla división por cero
int dividirSeguro(int a, int b) {
  if (b == 0) {
    throw std::runtime_error("Error aritmético");
  }
  return a / b;
}
```

2.7.4.2 Estándares metodológicos

- **C++ Core-ES.105**: don't divide by integer zero.

2.7.5 [cpp.arit.5] - Evitar usar "unsigned" para evitar el uso de valores negativos

ID	cpp.arit.5	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Este control de seguridad aborda el uso inapropiado de tipos `unsigned`, los cuales pueden generar comportamientos inesperados al producir desbordes silenciosos, comparaciones incorrectas y errores lógicos que afectan a la integridad de las operaciones. El requisito promueve el uso de tipos con signo cuando sea necesario representar valores negativos o realizar comparaciones que puedan verse alteradas por conversión implícita. Aplicar este criterio reduce riesgos de vulnerabilidades vinculadas a corrupción de datos, fallos lógicos y vectores de ataque derivados de cálculos numéricos no controlados.

Codificación insegura

Gestión de una cantidad introducida por el usuario y posterior resta de un valor fijo. Se busca mostrar cómo el uso de tipos sin signo puede provocar comportamientos inesperados cuando se realizan operaciones que pueden producir resultados negativos, y cómo esto impacta en la lógica del programa.

```
#include <iostream>

int main() {
  //-- Declaración de variable sin signo para almacenar una cantidad
  unsigned int cantidad;

  //-- Solicita al usuario una cantidad numérica
  std::cout << "Introduce una cantidad: ";
  std::cin >> cantidad;
```

```
  //-- Resta un valor fijo que puede generar un resultado negativo lógico
  unsigned int resultado = cantidad - 10;

  //-- Muestra el resultado calculado
  std::cout << "Resultado despues de restar 10: " << resultado << std::endl;
  return 0;
}
```

Codificación recomendada

Gestión de una cantidad introducida por el usuario y posterior resta de un valor fijo, utilizando tipos con signo y validaciones explícitas. Se pretende asegurar que las operaciones aritméticas reflejen correctamente valores negativos y que la lógica del programa sea clara y segura.

```
#include <iostream>

int main() {
  //-- Declaración de variable con signo para permitir valores negativos
  int cantidad;

  //-- Solicita al usuario una cantidad numérica
  std::cout << "Introduce una cantidad: ";
  std::cin >> cantidad;

  //-- Resta un valor fijo y conserva correctamente valores negativos
  int resultado = cantidad - 10;

  //-- Validación explícita del resultado
  if (resultado < 0) {
    //-- Informa que el resultado es negativo
    std::cout << "El resultado es negativo: " << resultado << std::endl;
  } else {
    //-- Informa que el resultado es positivo o cero
    std::cout << "Resultado despues de restar 10: " << resultado << std::endl;
  }
  return 0;
}
```

2.7.5.1 Descripción

Los tipos `unsigned` puede ser útiles en situaciones muy concretas, como operaciones aritméticas puramente binarias o manipulación de datos donde se garantiza que el rango no incluye valores negativos. Sin embargo, en la mayoría de los escenarios de desarrollo seguro, su utilización introduce riesgos sutiles pero significativos, especialmente cuando intervienen cálculos, comparaciones, conversiones implícitas y validación de datos provenientes de fuentes externas. Un error numérico que pase

inadvertido puede convertirse en una vulnerabilidad si permite a un atacante manipular valores para alterar el comportamiento del programa. Por ello, **evitar el uso de tipos unsigned cuando no sea estrictamente necesario** contribuye directamente a la integridad del sistema.

Los tipos `unsigned` generan problemas frecuentes debido al comportamiento definido por el estándar del lenguaje cuando ocurre un desbordamiento. A diferencia de los tipos con signo, los desbordes en tipos `unsigned` no son indefinidos: realizan un wrap-around modular. Aunque esto parece beneficioso desde una perspectiva matemática, en términos de seguridad implica que un desbordamiento puede pasar completamente desapercibido para el programador. Además, el sistema de promociones aritméticas obliga a conversiones que pueden generar resultados muy distintos a los esperados.

Uno de los mayores riesgos es la conversión implícita entre tipos con signo y sin signo. En muchos contextos de seguridad, las validaciones dependen de comparaciones correctas. Si un valor de entrada negativo es convertido automáticamente a `unsigned`, se obtiene un valor inmenso que puede hacer que la validación parezca correcta cuando en realidad no lo es. Este comportamiento puede ser aprovechado, por ejemplo, para manipular índices de acceso a memoria o forzar bucles a ejecutar iteraciones inesperadas.

El valor de la integridad en seguridad implica que los datos y operaciones se mantengan consistentes durante la ejecución. El uso inapropiado de `unsigned` contribuye a violaciones de integridad al permitir comportamientos no detectados. Esto incluye errores de validación, corrupción silenciosa de datos y resultados inconsistentes en cálculos fundamentales para el control del flujo o la lógica de negocio. En aplicaciones donde la integridad es crítica —como sistemas financieros, redes, telecomunicaciones o software industrial— estos fallos pueden escalar a problemas serios.

Si se requiere utilizar un tipo `unsigned` por motivos de interoperabilidad, optimización o requerimientos de un API externa, se deben implementar medidas adicionales. Entre ellas se encuentran comprobaciones explícitas de límites, encapsulación en funciones seguras y el uso de bibliotecas que proporcionen aritmética segura. También es fundamental documentar claramente la expectativa de rango de los datos.

Riesgos asociados

- Comportamiento inesperado en operaciones aritméticas (por ejemplo, desbordamiento hacia valores grandes y no negativos en vez de errores al usar valores negativos).
- Incompatibilidad con funciones o bibliotecas que esperan valores signed.
- Posibles vulnerabilidades de seguridad debido a conversiones implícitas entre signed y unsigned, lo que puede ser explotado en validaciones o límites.
- Comparaciones incorrectas, ya que comparar tipos signed y unsigned puede dar

resultados no intuitivos y errores lógicos.

- Dificultad en el manejo adecuado de señalización de errores usando valores negativos (por ejemplo, retornar -1 para indicar error no es posible con unsigned).
- Introducción de bugs al realizar bucles decrecientes, ya que un unsigned puede "enrollarse" (underflow) y conducir a bucles infinitos.
- Reducción de la legibilidad y mantenibilidad del código al tener que gestionar explícitamente las conversiones y comprobaciones adicionales.

Buenas prácticas

- Preferir tipos enteros con signo cuando se realicen operaciones aritméticas que puedan producir resultados negativos.

```
//-- Resta entre valores puede generar desbordamiento silencioso
unsigned int a = 5;
unsigned int b = 10;
unsigned int resultado = a - b;
```

```
//-- Uso de tipo con signo para reflejar correctamente valores negativos
int a = 5;
int b = 10;
int resultado = a - b;
```

- Evitar comparaciones mixtas entre tipos signed y unsigned para prevenir conversiones implícitas peligrosas.

```
//-- Comparación produce conversión implícita inesperada
int limite = -1;
unsigned int valor = 10;
if (valor > limite) {
  /* ... */
}
```

```
//-- Uso consistente de tipos con signo en la comparación
int limite = -1;
int valor = 10;
if (valor > limite) {
  /* ... */
}
```

- No usar unsigned como mecanismo de validación de valores no negativos; validar explícitamente las entradas.

```
//-- Uso de unsigned para impedir números negativos sin validación explícita
void procesarCantidad(unsigned int cantidad) {
  /* ... */
}
```

```
//-- Validación explícita del rango permitido usando tipo con signo
void procesarCantidad(int cantidad) {
  if (cantidad < 0) {
    throw std::invalid_argument("cantidad negativa");
  }
  /* ... */
}
```

2.7.5.2 Estándares metodológicos

- **C++ Core-ES.106**: don't try to avoid negative values by using unsigned.

2.7.6 [cpp.arit.6] - Evitar condiciones de wrap-around en aritmética sin signo con constantes

ID	cpp.arit.6	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se deben establecer medidas para evitar que las operaciones aritméticas realizadas con tipos sin signo y valores constantes produzcan condiciones de wrap-around que alteren de forma inesperada el resultado numérico. Dichas condiciones pueden generar desbordamientos silenciosos, errores lógicos, comportamientos no deseados y vulnerabilidades explotables. Mediante verificaciones explícitas, límites seguros, conversiones adecuadas y validación previa de rangos, este requisito busca asegurar que la aritmética se comporte de manera determinista y confiable incluso ante valores límite o expresiones constantes potencialmente peligrosas.

Codificación insegura

Procesamiento de un valor sin signo al que se le resta una constante fija. Se busca mostrar cómo una resta directa puede provocar un wrap-around cuando el valor es menor que la constante, generando un resultado inesperado. El objetivo es entender el impacto de la aritmética modular en tipos sin signo y cómo puede afectar la lógica del programa.

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

int main() {
```

```
  //-- Valor inicial menor que la constante a restar
  unsigned int value = 5;
  const unsigned int decrement = 10;

  //-- Resta directa sin validación previa
  unsigned int result = value - decrement;

  //-- Se imprime el resultado, que puede haber sufrido wrap-around
  std::cout << "Valor original: " << value << std::endl;
  std::cout << "Resultado tras resta: " << result << std::endl;
  return 0;
}
```

Codificación recomendada

Procesamiento de un valor sin signo al que se le resta una constante fija aplicando validación previa. Se pretende evitar el wrap-around comprobando que el valor sea mayor o igual que la constante antes de realizar la operación, manteniendo la coherencia lógica y evitando resultados inesperados.

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

int main() {
  //-- Valor inicial menor que la constante a restar
  unsigned int value = 5;
  const unsigned int decrement = 10;
  unsigned int result = 0;

  //-- Validación previa para evitar wrap-around
  if (value >= decrement) {
    result = value - decrement;
  } else {
    //-- Manejo seguro cuando la resta no es válida
    std::cerr << "Error: la resta provocaria wrap-around" << std::endl;
    return 1;
  }

  //-- Se imprime el resultado validado
  std::cout << "Valor original: " << value << std::endl;
  std::cout << "Resultado tras resta segura: " << result << std::endl;
  return 0;
}
```

2.7.6.1 Descripción

El **wrap-around** es el comportamiento por el cual **un valor se desborda y continúa desde el extremo opuesto del rango**. El uso de tipos sin signo con operaciones que involucran constantes puede originar condiciones de wrap-around debido a las reglas semánticas del lenguaje en aritmética modular. Cuando un valor excede el rango representable, no se produce un error o excepción, sino un desbordamiento silencioso que reinicia el valor desde 0 o desde su límite equivalente. Este comportamiento puede introducir vulnerabilidades si el código depende de comparaciones, límites o cálculos que deben preservarse sin alteraciones involuntarias. Es esencial comprender que el wrap-around no es un comportamiento indefinido en tipos sin signo, sino un comportamiento definido, pero potencialmente peligroso si se asume incorrectamente que sobrepasar un límite generará un fallo detectable. Para prevenir estas situaciones, se requiere aplicar una estrategia sistemática de validación, uso cuidadoso de constantes, tipos con tamaño suficiente y controles explícitos sobre rangos y resultados numéricos.

El wrap-around puede introducir desviaciones lógicas que comprometan propiedades críticas de seguridad. Por ejemplo, si una operación aparentemente inofensiva utiliza una constante grande o un desplazamiento numérico y excede el rango del tipo sin signo, el resultado puede disminuir drásticamente o quedar fuera del dominio esperado. Este comportamiento puede ser explotado en situaciones donde la seguridad dependa de cálculos deterministas, como verificaciones de límites, construcción de offsets, generación de índices, validación de tamaños de buffer o comprobaciones de condiciones previas. En contextos más sensibles, un atacante puede desencadenar un wrap-around manipulando valores de entrada que interactúan con constantes internas, provocando fallos en la lógica de control o habilitando accesos fuera de rango en memoria.

La prevención del wrap-around debe abordarse mediante un conjunto estructurado de prácticas que fortalezcan la robustez del código. Una de las medidas fundamentales es validar los operandos antes de ejecutar cualquier operación susceptible de generar un valor excedente. Asimismo, es recomendable usar tipos con mayor anchura o tipos enteros con signo cuando el comportamiento requerido no necesite modularidad implícita. La aplicación explícita de `safe_cast`, comprobaciones de límites y operaciones condicionadas minimiza el riesgo de comportamientos inesperados. En sistemas críticos, el uso de bibliotecas de enteros seguros o envoltorios que detecten desbordamientos de manera explícita puede fortalecer significativamente la seguridad. Otra estrategia útil consiste en evitar el uso de constantes numéricas sin verificar su rango efectivo para el tipo destino, asegurando que ninguna operación produzca una transición silenciosa a valores incorrectos. Para expresiones complejas, la evaluación anticipada en tipos ampliados y la verificación de límites antes de la conversión final reduce notablemente el riesgo de errores.

La interpretación precisa de las reglas numéricas puede diferir entre compiladores debido a optimizaciones o evaluaciones constantes. Aunque el wrap-around en tipos sin signo está definido por el estándar, su interacción con plegado de constantes, análisis estático o transformaciones de optimización puede producir resultados difíciles de analizar si el código no está explícitamente protegido. Por ello, resulta fundamental garantizar que las expresiones que involucren constantes numéricas se mantengan dentro de rangos seguros o se realicen bajo un modelo explícito de verificación, evitando confiar en supuestos implícitos de portabilidad. Cuando se empleen macros, `constexpr` o expresiones de tiempo de compilación, se deben validar igualmente los límites y garantizar que el resultado final no excede el rango del tipo objetivo.

Riesgos asociados

- Desbordamiento silencioso de valores, generando resultados inesperados en operaciones matemáticas.
- Ejecución de lógica incorrecta en decisiones condicionales debido a resultados fuera del rango esperado.
- Vulnerabilidades de seguridad, tales como bypass de límites o validación insuficiente en bucles y validación de entradas.
- Exposición a ataques de desbordamiento de enteros (integer overflow), potencialmente explotables para denegación de servicio o escalado de privilegios.
- Dificultad para detectar y depurar errores, ya que el wrap-around en enteros sin signo no genera excepciones ni advertencias en tiempo de compilación/ejecución.
- Posible corrupción o pérdida de datos al superar los límites máximos de las variables sin signo.
- Anulación de controles de seguridad dependientes del rango correcto de los valores numéricos, facilitando evasión de restricciones.

Buenas prácticas

- Validar explícitamente los límites antes de realizar operaciones aritméticas con constantes sobre tipos sin signo.

```
//-- Suma directa con constante sin validar límite superior
unsigned int size = input;
unsigned int total = size + 100;
```

```
//-- Validación previa contra el máximo representable
unsigned int size = input;
if (size > std::numeric_limits::max() - 100) {
```

```
  throw std::overflow_error("overflow");
}
unsigned int total = size + 100;
```

- Evitar comparaciones lógicas posteriores a una operación que ya pudo haber producido wrap-around.

```
//-- Comparación después de una resta que puede desbordar
unsigned int length = value;
if (length - 1 < 10) {
  process();
}
```

```
//-- Verificación previa antes de realizar la resta
unsigned int length = value;
if (length == 0) {
  throw std::underflow_error("underflow");
}
if (length - 1 < 10) {
  process();
}
```

- Usar tipos más amplios o conversiones controladas antes de operar con constantes que puedan provocar desbordamiento.

```
//-- Multiplicación con constante en tipo sin signo limitado
unsigned int count = items;
unsigned int bytes = count * 1024;
```

```
//-- Promoción a tipo más amplio antes de multiplicar
unsigned int count = items;
unsigned long long temp = static_cast(count) * 1024ULL;
if (temp > std::numeric_limits::max()) {
  throw std::overflow_error("overflow");
}
unsigned int bytes = static_cast(temp);
```

- Evitar depender del comportamiento modular implícito de enteros sin signo en validaciones de seguridad.

```
//-- Uso implícito de wrap-around para controlar límites
unsigned int index = position + 1;
if (index > maxSize) {
  handleError();
}
```

```
//-- Validación explícita antes de incrementar
unsigned int position = input;
if (position >= maxSize) {
  handleError();
}
unsigned int index = position + 1;
```

2.7.6.2 Estándares metodológicos

- **MISRA CPP-8.20.1**: Unsigned arithmetic with constants should not wrap.

2.7.7 [cpp.arit.7] - Evitar usar aritmética aditiva sobre iteradores si el resultado puede desbordar

ID	cpp.arit.7	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Para evitar desbordamientos numéricos y accesos fuera de límites se recomienda verificar de forma estricta cualquier operación aritmética realizada sobre los iteradores. Al manipular iteradores mediante operaciones como suma, resta o desplazamientos, es posible que se generen posiciones no válidas si no se evalúan previamente los límites del contenedor. Este tipo de errores puede conducir a corrupción de memoria, lecturas o escrituras ilegales y vulnerabilidades explotables. Aplicar este requisito ayuda a garantizar una manipulación segura de contenedores y un código más robusto.

Codificación insegura

Acceso a un elemento de un contenedor usando iteradores y un desplazamiento calculado externamente. Se busca mostrar cómo un desplazamiento mayor al tamaño del contenedor puede generar un iterador fuera de rango y provocar comportamiento indefinido al desreferenciarlo.

```
#include <iostream>
#include <vector>

int main() {
  //-- Crear un vector con 5 elementos
  std::vector<int> values = {1, 2, 3, 4, 5};

  //-- Obtener iterador al inicio
  auto it = values.begin();

  //-- Desplazamiento calculado externamente (valor no validado)
  std::size_t offset = 10;
```

```
  //-- Aritmética aditiva sin verificar límites
  auto itOutOfRange = it + offset;

  //-- Desreferenciación de un iterador potencialmente inválido (comportamiento
indefinido)
  std::cout << "Valor: " << *itOutOfRange << std::endl;
  return 0;
}
```

Codificación recomendada

Acceso seguro a un elemento de un contenedor validando previamente que el desplazamiento no supere el tamaño disponible. Se pretende mostrar cómo evitar desbordamientos de iteradores comprobando límites antes de aplicar aritmética o utilizando utilidades estándar de forma controlada.

```
#include <iostream>
#include <vector>
#include <iterator>

int main() {
  //-- Crear un vector con 5 elementos
  std::vector<int> values = {1, 2, 3, 4, 5};

  //-- Obtener iterador al inicio
  auto it = values.begin();

  //-- Desplazamiento calculado externamente
  std::size_t offset = 3;

  //-- Validar que el desplazamiento no exceda el tamaño del contenedor
  if (offset <= values.size()) {
    //-- Avanzar el iterador solo después de validar límites
    auto safeIt = std::next(it, offset);

    //-- Verificar que no sea el iterador end antes de desreferenciar
    if (safeIt != values.end()) {
      std::cout << "Valor: " << *safeIt << std::endl;
    }
  } else {
    //-- Manejo de error si el desplazamiento es inválido
    std::cerr << "Error: desplazamiento fuera de rango" << std::endl;
  }
  return 0;
}
```

2.7.7.1 Descripción

La aritmética sobre iteradores es una funcionalidad potente del lenguaje que permite desplazarse por estructuras de datos como arrays, vectores o cadenas. Sin embargo, cuando estas operaciones se realizan sin verificar límites o sin controlar adecuadamente la validez de los valores intermedios, pueden producirse **desbordamientos numéricos** o movimientos hacia posiciones no definidas del contenedor. Este tipo de condiciones pueden generar comportamientos indefinidos, errores de memoria y vulnerabilidades explotables.

La expresión `it + n` o `it - n` implica la creación de un nuevo iterador desplazado. En muchos contenedores, especialmente aquellos que cumplen la semántica de iteradores aleatorios como `std::vector` o `std::array`, estas operaciones son válidas siempre que: el desplazamiento se mantenga dentro del rango del contenedor y el cálculo de la posición resultante no origine un desbordamiento del tipo entero usado internamente. Cuando estas condiciones no se verifican, es posible que ocurra un **acceso fuera de límites**, uno de los problemas de seguridad más críticos en C++.

En sistemas de alta seguridad o software crítico, se recomienda integrar herramientas de análisis estático para detectar desbordamientos potenciales y accesos fuera de límites. También es posible usar modelos de memoria seguros como "safe iterators" o contenedores instrumentados que realizan comprobaciones automáticas. Otra estrategia consiste en crear wrappers internos para operaciones con iteradores que centralizan las verificaciones necesarias.

Riesgos asociados

- Desbordamiento de memoria, causando lecturas o escrituras fuera de los límites válidos del contenedor.
- Corrupción de datos o estructuras internas del programa.
- Ejecución de código indefinido que puede llevar a comportamientos impredecibles.
- Vulnerabilidades de seguridad, tales como posibilidad de explotación para ejecución de código malicioso.
- Fugas de información debido al acceso a zonas de memoria no autorizadas.
- Fallas o bloqueos de la aplicación (crashes).
- Dificultad en el diagnóstico y depuración de errores relacionados.
- Inestabilidad general en la ejecución del programa.

Buenas prácticas

- Validar los límites antes de realizar aritmética con iteradores para evitar desbordamientos fuera del rango del contenedor.

```
//-- Avanza el iterador sin comprobar límites
std::vector v = {1,2,3,4,5};
auto it = v.begin();
it = it + 10;
int value = *it;
```

```
//-- Verifica que el desplazamiento esté dentro del rango válido
std::vector v2 = {1,2,3,4,5};
auto it2 = v2.begin();
std::size_t offset = 3;
if (offset <= v2.size()) {
  it2 = it2 + offset;
  int value2 = *it2;
}
```

- Preferir funciones seguras como `std::next` junto con validaciones explícitas del tamaño restante.

```
//-- Usa aritmética directa sin comprobar distancia restante
std::list lst = {1,2,3,4};
auto it = lst.begin();
it = std::next(it, 5);
int value = *it;
```

```
//-- Calcula la distancia disponible antes de avanzar
std::list lst2 = {1,2,3,4};
auto it2 = lst2.begin();
auto remaining = std::distance(it2, lst2.end());
if (remaining >= 2) {
  it2 = std::next(it2, 2);
  int value2 = *it2;
}
```

- Evitar cálculos manuales con índices derivados de iteradores cuando puedan reemplazarse por métodos del contenedor.

```
//-- Calcula una posición sumando manualmente al iterador
std::vector v = {10,20,30};
auto it = v.begin();
std::size_t pos = 5;
int value = *(it + pos);
```

```
//-- Utiliza acceso controlado con verificación de rango
std::vector v2 = {10,20,30};
std::size_t pos2 = 1;
if (pos2 < v2.size()) {
```

```
  int value2 = v2.at(pos2);
}
```

- Usar tipos adecuados para el cálculo de desplazamientos y evitar conversiones implícitas que puedan provocar desbordamientos.

```
//-- Usa un entero con posible desbordamiento en el desplazamiento
std::vector v = {1,2,3};
auto it = v.begin();
int offset = -1;
it = it + offset;
```

```
//-- Emplea std::ptrdiff_t y valida el rango antes de aplicar el desplazamiento
std::vector v2 = {1,2,3};
auto it2 = v2.begin();
std::ptrdiff_t offset2 = 1;
if (offset2 >= 0 && static_cast(offset2) <= v2.size()) {
  it2 = it2 + offset2;
}
```

2.7.7.2 Estándares metodológicos

- **CERT/CPP-CTR55**: Do not use an additive operator on an iterator if the result would overflow.

2.7.8 [cpp.arit.8] - Evitar usar aritmética de punteros sobre objetos polimórficos

ID	**cpp.arit.8**	**Uso**	**Obligatorio**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

El uso de aritmética de punteros sobre objetos polimórficos puede producir comportamientos indefinidos, corrupciones de memoria y violaciones de acceso. En los objetos polimórficos, la disposición interna de memoria depende de la presencia de una tabla virtual y puede variar entre compiladores, versiones y configuraciones. Por este motivo, manipular punteros con operaciones aritméticas puede romper la integridad de las estructuras internas, poniendo en riesgo la seguridad, la estabilidad y la correcta ejecución del programa.

Codificación insegura

Se muestra una jerarquía polimórfica donde se almacenan objetos derivados y se accede a ellos mediante un puntero al tipo base usando aritmética de punteros. La intención es

recorrer varios elementos, pero se utiliza una técnica insegura que puede provocar comportamiento indefinido cuando se trabaja con objetos polimórficos.

```
#include <iostream>

class Base {
public:
  //-- Clase base con método virtual para habilitar polimorfismo
  virtual void print() const {
    std::cout << "Base" << std::endl;
  }
  virtual ~Base() = default;
};

class Derived : public Base {
  int value;
public:
  //-- Constructor que inicializa un valor interno
  Derived(int v) : value(v) {}

  //-- Sobrescritura del método virtual
  void print() const override {
    std::cout << "Derived: " << value << std::endl;
  }
};

int main() {
  //-- Array de objetos derivados
  Derived derivedArray[2] = { Derived(10), Derived(20) };

  //-- Conversión implícita a puntero base
  Base* basePtr = derivedArray;

  //-- Uso de aritmética de punteros sobre tipo polimórfico
  //-- Esto puede provocar comportamiento indefinido
  for (int i = 0; i < 2; ++i) {
    basePtr[i].print();
  }
  return 0;
}
```

Codificación recomendada

Se presenta una jerarquía polimórfica donde los objetos derivados se gestionan mediante contenedores y punteros inteligentes, evitando cualquier aritmética de punteros. La intención es recorrer varios objetos polimórficos de forma segura, manteniendo el comportamiento definido y respetando las buenas prácticas de gestión de memoria.

```
#include <iostream>
#include <vector>
#include <memory>

class Base {
public:
  //-- Clase base con método virtual para habilitar polimorfismo
  virtual void print() const = 0;
  virtual ~Base() = default;
};

class Derived : public Base {
  int value;
public:
  //-- Constructor que inicializa un valor interno
  Derived(int v) : value(v) {}

  //-- Implementación del método virtual
  void print() const override {
    std::cout << "Derived: " << value << std::endl;
  }
};

int main() {
  //-- Uso de contenedor con punteros inteligentes
  std::vector<std::unique_ptr<Base>> objects;

  objects.push_back(std::make_unique<Derived>(10));
  objects.push_back(std::make_unique<Derived>(20));

  //-- Recorrido seguro sin aritmética de punteros
  for (const auto& obj : objects) {
    obj->print();
  }
  return 0;
}
```

2.7.8.1 Descripción

El uso de punteros es una capacidad poderosa, pero también una de las causas más frecuentes de errores de memoria y vulnerabilidades graves. Cuando se emplean con estructuras simples, la aritmética de punteros permite recorrer arrays o áreas contiguas de memoria con corrección. Sin embargo, cuando se trata de objetos polimórficos, esta propiedad deja de ser válida debido a que la distribución de estos objetos en memoria no es lineal ni homogénea, ya que suele incluir al menos un puntero a la tabla virtual (`vptr`) y otros metadatos dependientes del compilador. Por ello, realizar expresiones

como `ptr + 1` o `*(ptr + n)` sobre instancias polimórficas puede ocasionar accesos a zonas desconocidas o inconsistentes.

El lenguaje no garantiza en absoluto que los objetos polimórficos sean ubicados de manera contigua, ni que el tamaño real del objeto sea estático o idéntico entre jerarquías de clases. Los compiladores son libres de aplicar optimizaciones, reordenamientos o ampliaciones internas que dificultan cualquier intento de manipulación a través de aritmética de punteros. Así, operaciones aparentemente simples pueden llevar al programa a interpretar fragmentos de memoria aleatorios como si fuesen instancias válidas, lo que puede resultar en comportamientos erráticos, pérdida de datos, violaciones de integridad y aperturas a vulnerabilidades explotables.

Además, el uso de aritmética de punteros sobre objetos polimórficos puede provocar la ruptura de los principios de aliasing seguro. El compilador puede aplicar reglas de aliasing que asumen que ciertas referencias no se solapan, y la manipulación incorrecta de memoria con punteros puede invalidar esas asunciones, causando transformaciones inesperadas durante la optimización. Estos comportamientos indefinidos son especialmente problemáticos en entornos de seguridad, donde cualquier inconsistencia puede convertirse en una vía de explotación si un atacante manipula o fuerza condiciones límite.

El riesgo se incrementa cuando los objetos polimórficos contienen datos sensibles o estado interno crítico que debe preservarse con integridad. Un desplazamiento incorrecto puede permitir a un atacante influir indirectamente en punteros internos, modificar atributos, corromper memoria adyacente o forzar flujos de ejecución anómalos. Por ello, debe emplearse siempre un enfoque seguro que evite la posibilidad misma de realizar aritmética de punteros sobre este tipo de objetos. En su lugar, deben usarse contenedores seguros, iteradores especializados o referencias estándar de C++, que garantizan comportamiento definido y evitan accesos fuera de límites.

Otro factor importante es que, en muchos casos, los desarrolladores asumen erróneamente que los objetos polimórficos tienen el mismo tamaño o están contiguos en memoria cuando son almacenados en estructuras como arrays de C. Esta suposición es incorrecta y causa una falsa sensación de seguridad. Las colecciones heterogéneas deben gestionarse a través de clases contenedor como `std::vector<std::unique_ptr<base>>` o `std::vector<std::shared_ptr<base>>`, donde cada objeto se maneja de forma aislada y no se presupone ninguna estructura de memoria secuencial.

El uso de **smart pointers** resulta especialmente apropiado, ya que obligan a los desarrolladores a mantener un control explícito sobre la propiedad del objeto y ayudan a evitar operaciones inseguras. Asimismo, los iteradores de las colecciones estándar están diseñados para impedir aritmética insegura que pudiera derivar en un comportamiento indefinido. Estas soluciones modernas reducen enormemente la probabilidad de que un desarrollo incurra en errores de memoria, fortaleciendo la

integridad del sistema y manteniendo un comportamiento predecible incluso bajo carga o condiciones extremas.

Otra situación común ocurre cuando se intenta almacenar objetos polimórficos por valor en arrays estáticos o buffers de bajo nivel. Esta práctica es incompatiblemente insegura porque el tamaño concreto de los objetos derivados no puede conocerse estáticamente, y su manipulación puede destruir información esencial o producir sobre lecturas y sobreescrituras.

Riesgos asociados

- Comportamiento indefinido al acceder a miembros o métodos de objetos polimórficos.
- Violación del principio de alineación de objetos, provocando fallos de acceso a memoria (segmentation fault).
- Corrupción de datos debido a acceder a partes incorrectas del objeto.
- Bugs difíciles de detectar y reproducir al alterar la disposición interna de los objetos.
- Riesgo de explotación de vulnerabilidades para ejecución de código arbitrario.
- Pérdida de portabilidad y dependencia de detalles específicos de la implementación del compilador.
- Fallo en la destrucción adecuada de objetos, provocando fugas de memoria o comportamientos inesperados.

Buenas prácticas

- Evitar la aritmética de punteros sobre objetos polimórficos; utilizar contenedores estándar o estructuras adecuadas para almacenar múltiples instancias.

```
class Base {
public:
  virtual void foo();
};
class Derivada : public Base {
  void foo() override;
};
```

```
//-- Iteración basada en índice sobre array de objetos
Derivada arr[2];
Base* ptr = arr;
ptr++;
ptr->foo();
```

```
//-- Uso de contenedor seguro para objetos polimórficos
std::vector<std::unique_ptr<Base>> vec;
vec.push_back(std::make_unique<Derivada>());
vec.push_back(std::make_unique<Derivada>());
vec[1]->foo();
```

- No asumir que objetos derivados son contiguos ni del mismo tamaño que la clase base; evitar cálculos manuales de desplazamiento.

```
class Base {
public:
  virtual ~Base() = default;
};
class Derivada : public Base {
  int datos[10];
};
```

```
//-- Cálculo manual de desplazamiento sobre tipo base
Derivada objetos[2];
Base* basePtr = objetos;
Base* siguiente = basePtr + 1;
```

```
//-- Acceso mediante índice tipado correctamente
Derivada objetos2[2];
Derivada* derivPtr = objetos2;
Derivada* siguienteSeguro = &derivPtr[1];
```

- Cuando se requiera recorrer colecciones polimórficas, emplear iteradores o bucles basados en rango en lugar de aritmética de punteros.

```
class Base {
public:
  virtual void imprimir();
};
class Derivada : public Base {
  void imprimir() override;
};
```

```
//-- Recorrido mediante incremento manual de puntero
Derivada elementos[3];
Base* p = elementos;
```

```
for(int i = 0; i < 3; ++i) {
  p->imprimir();
  ++p;
}
```

```
//-- Recorrido usando contenedor e iterador seguro
std::vector<std::unique_ptr<Base>> lista;
lista.push_back(std::make_unique<Derivada>());
lista.push_back(std::make_unique<Derivada>());
for(const auto& elem : lista) {
  elem->imprimir();
}
```

2.7.8.2 Estándares metodológicos

- **CERT/CPP-CTR56**: Do not use pointer arithmetic on polymorphic objects.

2.8 Estructura de control y flujo

2.8.1 [cpp.flow.1] - Usar índices, referencias e iteradores válidos

ID	cpp.flow.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se debería garantizar que todos los índices, referencias e iteradores utilizados sean siempre válidos antes de acceder a ellos, con el fin de evitar errores graves como accesos fuera de rango, corrupción de memoria, fugas de información o fallos del programa. Esto incluye: verificar rangos, utilizar métodos seguros de acceso, prevenir la invalidación de iteradores, y asegurar que las referencias no apunten a memoria liberada o fuera de su ámbito.

Codificación insegura

Acceso a elementos de un `std::vector` utilizando índices e iteradores sin validar su rango ni su validez. Se muestra cómo un acceso fuera de límites y el uso de un iterador invalidado pueden provocar comportamiento indefinido, fallos en tiempo de ejecución o vulnerabilidades.

```
#include <iostream>
#include <vector>

int main() {
  //-- Crear un vector con tres elementos
  std::vector<int> numeros = {10, 20, 30};
```

```
  //-- Acceso directo a un índice fuera de rango (comportamiento indefinido)
  std::cout << "Acceso fuera de rango: " << numeros[5] << std::endl;

  //-- Obtener un iterador válido inicialmente
  auto it = numeros.begin();

  //-- Modificar el vector provocando posible realocación e invalidación del
iterador
  numeros.push_back(40);
  numeros.push_back(50);

  //-- Uso de un iterador potencialmente invalidado (comportamiento indefinido)
  std::cout << "Iterador invalidado: " << *it << std::endl;
  return 0;
}
```

Codificación recomendada

Acceso seguro a elementos de un `std::vector` validando índices antes de usarlos y gestionando correctamente la validez de iteradores tras modificar el contenedor. Se muestra cómo evitar accesos fuera de rango y cómo readquirir iteradores después de operaciones que pueden invalidarlos.

```
#include <iostream>
#include <vector>
#include <stdexcept>

int main() {
  //-- Crear un vector con tres elementos
  std::vector<int> numeros = {10, 20, 30};

  //-- Validar el índice antes de acceder
  size_t indice = 2;
  if (indice < numeros.size()) {
    std::cout << "Acceso válido: " << numeros.at(indice) << std::endl;
  } else {
    std::cerr << "Índice fuera de rango" << std::endl;
  }

  //-- Obtener un iterador y reacquirirlo tras modificar el contenedor
  auto it = numeros.begin();

  //-- Reservar capacidad para evitar invalidaciones innecesarias
  numeros.reserve(10);
  numeros.push_back(40);
  numeros.push_back(50);
```

```
  //-- Reobtener el iterador tras modificaciones del contenedor
  it = numeros.begin();
  if (it != numeros.end()) {
    std::cout << "Iterador válido: " << *it << std::endl;
  }
  return 0;
}
```

2.8.1.1 Descripción

El uso seguro de índices, referencias e iteradores es fundamental para una codificación robusta, debido a que cualquier acceso indebido puede comprometer no solo la estabilidad del programa sino también su seguridad. Las operaciones relacionadas con contenedores, punteros y estructuras dinámicas implican un alto nivel de responsabilidad por parte del desarrollador, ya que el lenguaje ofrece gran flexibilidad, pero poca protección automática. Por ello, resulta esencial aplicar controles estrictos que aseguren que todos los índices empleados se mantienen dentro de sus límites, que las referencias continúan apuntando a objetos válidos, y que los iteradores no han sido invalidados por operaciones previas.

Un aspecto crítico es que los errores relacionados con índices e iteradores no siempre generan fallos inmediatos, sino que pueden comportarse de forma indefinida, dando lugar a vulnerabilidades difíciles de detectar. Entre las consecuencias más graves destacan fugas de datos, corrupción de memoria, **overflows** y ejecuciones no controladas.

El control de índices comienza por asegurar que cualquier acceso a un contenedor, array o estructura indexada se encuentre dentro de los límites válidos. Para ello es necesario emplear verificaciones explícitas cuando sea apropiado, así como utilizar métodos seguros proporcionados por la biblioteca estándar. En particular, `std::vector` y otros contenedores ofrecen el método `at()`, que realiza comprobaciones de rango y lanza excepciones ante valores incorrectos. Su uso mejora significativamente la seguridad cuando el rendimiento no constituye un factor crítico.

Las referencias son más seguras que los punteros en cuanto a sintaxis, pero no eliminan la posibilidad de errores. Una referencia puede quedar invalidada si el objeto al que apunta sale de ámbito, se destruye o se mueve. Por esta razón, resulta esencial garantizar que ninguna referencia se devuelva desde funciones que crean objetos locales y que no se mantengan referencias a elementos almacenados en estructuras que puedan cambiar su asignación interna. Asimismo, es fundamental evitar conservar referencias a elementos de un contenedor cuando este pueda reasignar memoria debido a operaciones como `push_back()` o `resize()`.

El lenguaje y la biblioteca estándar proporcionan herramientas que contribuyen a mantener la seguridad en el uso de iteradores. Entre ellas se encuentran los contenedores que invalidan menos iteradores, como `std::list` o `std::forward_list`,

que sólo invalidan iteradores en operaciones específicas, así como el uso de algoritmos que no requieren manipular los iteradores de forma manual. También es recomendable utilizar iteradores constantes cuando sea posible, lo que reduce la probabilidad de modificaciones inesperadas sobre el contenedor.

Riesgos asociados

- Desbordamiento de buffer y corrupción de memoria.
- Lectura o escritura en ubicaciones de memoria no definidas (comportamiento indefinido).
- Fugas de información sensible por acceso a memoria indebida.
- Condiciones de carrera y errores difíciles de depurar.
- Caídas de la aplicación (crash) por segmentation faults.
- Ejecución de código malicioso si la corrupción es aprovechada.
- Pérdida de integridad en los datos procesados.
- BYPASS de controles de seguridad basados en estructura de memoria.
- Acceso a objetos ya destruidos (dangling references/iterators), lo que puede llevar a exploits.
- Dificultad en el mantenimiento y evolución del código debido a errores semánticos sutiles.

Buenas prácticas

- Validar siempre los límites antes de acceder a contenedores indexados.

```
//-- Acceso sin validación de límites
std::vector v = {1,2,3};
int x = v[5];
```

```
//-- Validación explícita del índice antes del acceso
std::vector v2 = {1,2,3};
size_t idx = 2;
if(idx < v2.size()) {
  int y = v2[idx];
}
```

- Preferir el uso de `at()` cuando se requiera verificación automática de rango.

```
//-- Acceso directo sin comprobación automática
std::vector datos = {10,20,30};
int a = datos[3];
```

```
//-- Uso de at() con verificación de rango en tiempo de ejecución
std::vector datos2 = {10,20,30};
try {
  int b = datos2.at(1);
} catch(const std::out_of_range&) {
}
```

- No desreferenciar iteradores sin comprobar que sean válidos y distintos de `end()`.

```
//-- Desreferenciación sin verificar si el iterador es válido
std::vector nums = {4,5,6};
auto it = nums.end();
int v = *it;
```

```
//-- Verificación del iterador antes de desreferenciar
std::vector nums2 = {4,5,6};
auto it2 = nums2.begin();
if(it2 != nums2.end()) {
  int w = *it2;
}
```

- Evitar usar referencias a objetos cuyo tiempo de vida ha finalizado.

```
//-- Referencia a variable local fuera de su ámbito válido
int& obtener() {
  int x = 5;
  return x;
}
```

```
//-- Retorno por valor evitando referencia colgante
int obtenerSeguro() {
  int x = 5;
  return x;
}
```

- Reinicializar o invalidar iteradores tras modificar la estructura del contenedor.

```
//-- Uso de iterador después de modificar el contenedor
std::vector valores = {1,2,3};
auto it = valores.begin();
valores.push_back(4);
int n = *it;
```

```
//-- Reasignación del iterador tras la modificación
std::vector valores2 = {1,2,3};
```

```
auto it3 = valores2.begin();
valores2.push_back(4);
it3 = valores2.begin();
int m = *it3;
```

2.8.1.2 Estándares metodológicos

- **CERT/CPP-CTR51**: Use valid references, pointers, and iterators to reference elements of a container.
- **CERT/CPP-CTR53**: Use valid iterator ranges.

2.9 Clases

2.9.1 [cpp.cls.1] - Exponer lo mínimo a los miembros de una clase

ID	cpp.cls.1	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La práctica de restringir el acceso a los miembros internos de una clase reduce la superficie de ataque, evita manipulaciones indebidas y minimiza la dependencia entre componentes. Al encapsular correctamente los datos y comportamientos, se mejora la capacidad de mantener la integridad del estado interno de los objetos y se facilita la evolución segura del software sin introducir vulnerabilidades relacionadas con accesos no controlados.

Codificación insegura

Se muestra una clase que gestiona el saldo de una cuenta bancaria. El objetivo es observar cómo la exposición directa de los atributos permite modificar el estado interno sin control, generando inconsistencias y posibles fallos de seguridad o de lógica de negocio.

```
#include <iostream>

class Cuenta {
  //-- Atributos expuestos públicamente sin control
  public:
    std::string titular;
    double saldo;

  //-- Constructor simple
  Cuenta(const std::string& t, double s) {
    titular = t;
    saldo = s;
```

```
  }
};

int main() {
  //-- Creación de una cuenta con saldo inicial válido
  Cuenta cuenta("Juan", 1000.0);

  //-- Modificación directa del saldo sin validación
  cuenta.saldo = -500.0;

  //-- Se imprime un estado inconsistente
  std::cout << "Titular: " << cuenta.titular << std::endl;
  std::cout << "Saldo: " << cuenta.saldo << std::endl;
  return 0;
}
```

Codificación recomendada

Se presenta una versión de la clase donde los atributos internos están encapsulados y solo se exponen los métodos necesarios. El objetivo es aprender a reducir la superficie de exposición, aplicar validaciones y mantener la coherencia del estado interno.

```
#include <iostream>
#include <stdexcept>

class Cuenta {
  //-- Atributos privados para evitar acceso directo
  private:
    std::string titular;
    double saldo;

  //-- Solo se exponen los métodos estrictamente necesarios
  public:
    Cuenta(const std::string& t, double s) {
      if (s < 0) {
        throw std::invalid_argument("Saldo inicial invalido");
      }
      titular = t;
      saldo = s;
    }

    double obtenerSaldo() const {
      return saldo;
    }

    std::string obtenerTitular() const {
      return titular;
    }
```

```
    void depositar(double cantidad) {
      if (cantidad <= 0) {
        throw std::invalid_argument("Cantidad invalida");
      }
      saldo += cantidad;
    }
};

int main() {
  //-- Creación de una cuenta con validación de datos
  Cuenta cuenta("Juan", 1000.0);

  //-- Uso de métodos públicos controlados
  cuenta.depositar(500.0);

  //-- Consulta del estado sin exponer atributos internos
  std::cout << "Titular: " << cuenta.obtenerTitular() << std::endl;
  std::cout << "Saldo: " << cuenta.obtenerSaldo() << std::endl;
  return 0;
}
```

2.9.1.1 Descripción

El objetivo de esta regla es restringir el acceso a datos y funciones de manera que los consumidores de la clase solo puedan interactuar con una interfaz pública cuidadosamente diseñada. El uso incorrecto de niveles de acceso suele derivar en exposición accidental de detalles internos, lo cual puede permitir comportamientos inesperados, acceso a datos sensibles o modificación indebida del estado interno de los objetos. Con este diseño, además de fortalecer la integridad del software, nos ayuda a desacoplar componentes y a reducir la complejidad general del sistema.

C++ dispone de tres niveles fundamentales de acceso: `public`, `protected` y `private`. La regla de seguridad más relevante es que **la mayor parte de los datos miembro deben ser privados**, mientras que los métodos expuestos públicamente deben representar únicamente la interfaz estable y controlada por el desarrollador. Con ello se evita que código externo dependa de la estructura interna de la clase. Exponer atributos directamente, especialmente aquellos que contienen información crítica o estados intermedios, supone un riesgo importante porque facilita que actores internos o externos manipulen dichos valores sin validación ni restricciones.

Desde el punto de vista de seguridad, esta práctica está estrechamente alineada con los principios de fundamentales del diseño seguro como son: **encapsulamiento**, **separación de responsabilidades** y **defensa en profundidad**. Al limitar la visibilidad se reduce de forma natural la superficie vulnerable.

La encapsulación impide que consumidores externos modifiquen directamente los atributos, haciendo obligatorio el uso de métodos controlados como getters y setters cuando sea estrictamente necesario. Estos métodos no deben limitarse a exponer los datos o establecer valores de forma trivial, deben implementar la **validación de los datos**, **controles de consistencia** y **restricciones lógicas**. En entornos críticos, permitir que un valor se modifique libremente puede desencadenar condiciones de carrera, corrupción de estructuras o accesos fuera de rango. Las funciones deben actuar como puntos de control que aseguren que cualquier operación realizada sobre datos internos se ajusta a los criterios definidos. Además, permiten mantener invariantes y proteger reglas internas que deben cumplirse para evitar comportamientos inseguros.

El nivel `protected` es útil para permitir que clases derivadas accedan a ciertos elementos internos, pero un uso incorrecto puede expandir la superficie de exposición más allá de la intención original. Es importante evaluar cuidadosamente si un miembro realmente debe ser accesible a clases derivadas o si conviene encapsular ese comportamiento mediante métodos públicos o privados. El uso excesivo de miembros `protected` puede generar jerarquías frágiles y aumentar las vías por las que puede alterarse el estado interno.

Riesgos asociados

- Exposición innecesaria de datos sensibles o privados, facilitando accesos no autorizados.
- Dificultad para mantener y evolucionar el código, debido a una interfaz pública demasiado amplia.
- Incremento de dependencias externas sobre detalles internos de la clase, complicando el mantenimiento.
- Posibilidad de modificación indebida del estado interno de la clase, violando el encapsulamiento.
- Reducción del control sobre la validación y consistencia de los datos miembros.
- Aumento del riesgo de uso incorrecto o inesperado de la clase por otros desarrolladores.
- Dificultad para implementar controles de acceso o restricciones de seguridad a futuro.
- Superficie de ataque más amplia para exploits basados en manipulación de miembros expuestos.
- Complicación al realizar pruebas unitarias debido a la falta de control sobre las dependencias internas.
- Posible violación del principio de mínimo privilegio, clave en la seguridad del software.

Buenas prácticas

- Declarar los miembros de datos como `private` y exponer únicamente métodos públicos controlados.

```
//-- Miembros expuestos directamente
class Usuario {
public:
  std::string nombre;
  int edad;
};
```

```
//-- Encapsulamiento con control de acceso
class Usuario {
public:
  void setEdad(int e) {
    if (e >= 0) edad = e;
  }
  int getEdad() const {
    return edad;
  }
private:
  std::string nombre;
  int edad;
};
```

- Exponer solo las operaciones estrictamente necesarias, evitando métodos innecesarios en la interfaz pública.

```
//-- Métodos internos visibles públicamente
class Configuracion {
public:
  void cargarDesdeArchivo();
  void validarDatos();
  void aplicarCambios();
};
```

```
//-- Solo se expone la operación de alto nivel
class Configuracion {
public:
  void inicializar() {
    cargarDesdeArchivo();
    validarDatos();
    aplicarCambios();
  }
private:
  void cargarDesdeArchivo();
  void validarDatos();
```

```
  void aplicarCambios();
};
```

- Evitar devolver referencias o punteros modificables a miembros internos.

```
//-- Exposición directa de referencia modificable
class Cuenta {
public:
  double& obtenerSaldo() {
    return saldo;
  }
private:
  double saldo;
};
```

```
//-- Devolución por valor o referencia constante
class Cuenta {
public:
  double obtenerSaldo() const {
    return saldo;
  }
private:
  double saldo;
};
```

- Restringir la herencia innecesaria marcando la clase o métodos como `final` cuando no deban extenderse.

```
//-- Clase diseñada para herencia sin necesidad real
class Validador {
public:
  virtual bool validar(int dato);
};
```

```
//-- Restricción de extensión no prevista
class Validador final {
public:
  bool validar(int dato);
};
```

- Utilizar el principio de menor privilegio también en clases amigas (`friend`).

```
//-- Acceso total otorgado a otra clase
class Secreto {
  friend class Auditor;
private:
```

```
  std::string clave;
};
```

```
//-- Acceso limitado mediante método público controlado
class Secreto {
public:
  bool verificarClave(const std::string& intento) const {
    return intento == clave;
  }
private:
  std::string clave;
};
```

2.9.1.2 Estándares metodológicos

- **C++ Core-C.9**: Minimize exposure of members.

2.9.2 [cpp.cls.2] - Evitar definir una clase por defecto si no necesita operaciones especiales

ID	**cpp.cls.2**	**Uso**	**Recomendado**
CWSS Score	**74**	**Impacto técnico**	**Alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Declarar constructores, destructores u operadores por defecto cuando la clase no necesita realizar operaciones especiales introduce errores sutiles, comportamientos inseguros o contradicciones con las garantías del lenguaje. La generación implícita que ofrece el lenguaje suele ser más segura y eficiente, y evita problemas relacionados con la gestión manual de recursos, copias inconsistentes o semántica incorrecta al modificar artificialmente el comportamiento estándar de los objetos.

Codificación insegura

Se muestra una clase sencilla que solo almacena y muestra un mensaje. El objetivo es observar el impacto de declarar manualmente constructor, destructor y copia cuando no se necesita lógica especial, y cómo esto añade código innecesario que puede afectar mantenimiento, legibilidad y seguridad.

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

class Message {
  //-- Constructor definido manualmente sin necesidad real
  public:
    Message() {
```

```
    //-- Inicialización trivial que el compilador podría generar
  }

  //-- Constructor de copia definido sin lógica especial
  Message(const Message& other) {
    text = other.text;
  }

  //-- Destructor definido sin liberar recursos
  ~Message() {
    //-- No hay recursos manuales que liberar
  }

  void setText(const std::string& t) {
    text = t;
  }

  void print() const {
    std::cout << text << std::endl;
  }

private:
  std::string text;
};

int main() {
  Message msg;
  msg.setText("Hola mundo");
  msg.print();
  return 0;
}
```

Codificación recomendada

Se presenta la misma funcionalidad aplicando el principio de “regla de cero”. La clase no define constructores ni destructores porque no gestiona recursos manuales. Se delega al compilador la generación automática de operaciones especiales, reduciendo superficie de error y mejorando mantenibilidad.

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

class Message {
  //-- No se definen constructor, copia ni destructor
  //-- Se aplica la regla de cero
  public:
    void setText(const std::string& t) {
      text = t;
```

```
    }

    void print() const {
      std::cout << text << std::endl;
    }

  private:
    std::string text;
};

int main() {
  Message msg;
  msg.setText("Hola mundo");
  msg.print();
  return 0;
}
```

2.9.2.1 Descripción

El método más adecuado es permitir que el compilador genere estas funciones de forma automática siempre que sea posible. Cuando un desarrollador define manualmente estas funciones sin que exista una necesidad real, se incrementa la superficie de error, se pueden romper garantías implícitas del lenguaje y se introducen riesgos relacionados con la gestión de recursos, la semántica de copia o la consistencia del estado interno.

La gestión automática que realiza el compilador para funciones como `default constructor`, `copy constructor`, `move constructor`, `copy assignment`, `move assignment` y `destructor` está extremadamente optimizada y probada. Estas funciones siguen reglas predeterminadas que garantizan coherencia y evitan errores comunes, tales como inicializar atributos en orden incorrecto, olvidar copiar elementos fundamentales, aplicar semántica de movimiento inapropiada o liberar recursos varias veces. Cuando el desarrollador redefine alguno de estos elementos sin entender completamente las implicaciones, puede introducir vulnerabilidades difíciles de detectar, como corrupción de memoria, fugas de recursos o estados inconsistentes en la aplicación.

En casos donde existe herencia, recursos externos, identificadores únicos o restricciones lógicas, el riesgo aumenta de forma significativa. Además, definir estas funciones puede inhibir optimizaciones del lenguaje, como "copy elision", generando código menos eficiente.

Si existe un recurso especial, como un `file descriptor`, un `mutex`, una conexión o cualquier otro estado no copiable o no movible, entonces sí se debe implementar de forma explícita la semántica adecuada. En estos escenarios, resulta especialmente útil aplicar el concepto RAII para encapsular la gestión del recurso dentro de un objeto que implemente correctamente su constructor y destructor.

Riesgos asociados

- Exposición accidental de operaciones peligrosas como la copia, asignación o destrucción, que pueden permitir modificaciones inesperadas del estado del objeto.
- Fugas de recursos (resource leaks) al no manejar correctamente el ciclo de vida de recursos administrados por la clase (por ejemplo, punteros, archivos, descriptores de sockets).
- Condiciones de carrera y problemas en entornos multihilo por copia o destrucción incorrecta de objetos compartidos.
- Violaciones de invariantes internos del objeto, resultando en estados inconsistentes o corrupción de datos.
- Incremento de la superficie de ataque, ya que funciones por defecto pueden ser utilizadas maliciosamente o de formas no anticipadas.
- Posibilidad de duplicación no controlada de objetos, llevando a dobles liberaciones de memoria (double free) o uso posterior a liberar memoria (dangling pointer/use-after-free).
- Dificultad para realizar auditorías de seguridad, ya que el comportamiento implícito puede ocultar riesgos no evidentes.

Buenas prácticas

- Aplicar la regla de cero y no declarar constructores, destructores u operadores especiales cuando no gestionan recursos.

```
class Logger {
  public:
    //-- Define constructor y destructor vacíos sin gestionar recursos
    Logger() {}
    ~Logger() {}
  };
```

```
class Logger {
  public:
    //-- Se apoya en las funciones especiales generadas implícitamente
  };
```

- Evitar declarar explícitamente funciones especiales como `=default` si la clase no requiere comportamiento personalizado.

```
class Config {
  public:
    //-- Declara explícitamente operaciones especiales innecesarias
```

```
    Config() = default;
    ~Config() = default;
    Config(const Config&) = default;
    Config& operator=(const Config&) = default;
  };
```

```
class Config {
  public:
    //-- Omite declaraciones redundantes y mantiene la clase simple
  };
```

- No definir clases vacías como marcadores si no aportan comportamiento o restricciones; usar alias o tipos existentes.

```
class UserId {
  //-- Define clase vacía sin comportamiento adicional
};
```

```
//-- Utiliza alias de tipo estándar sin crear clase innecesaria
using UserId = std::string;
```

2.9.2.2 Estándares metodológicos

- **C++ Core-C.20**: If you can avoid defining any default operations, do.

2.9.3 [cpp.cls.3] - Definir un destructor en una clase con recursos

ID	cpp.cls.3	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Establece la necesidad de implementar correctamente un destructor en clases que gestionen recursos como memoria dinámica, descriptores de archivo o conexiones de red. Un destructor seguro previene fugas de recursos, dobles liberaciones y comportamientos indefinidos derivados de la gestión manual. Su uso adecuado, junto con el principio RAII, permite garantizar que los recursos sean liberados de forma automática incluso ante excepciones, fallos inesperados o salidas tempranas del flujo del programa.

Codificación insegura

Gestión manual de memoria dinámica dentro de una clase que reserva recursos en el constructor, pero no define un destructor para liberarlos. Se pretende mostrar el

impacto de no liberar recursos adquiridos y cómo esto provoca fugas de memoria al destruir el objeto.

```
#include <iostream>

class Buffer {
  //-- Puntero a memoria dinámica
  char* data;
  size_t size;

public:
  //-- Constructor que reserva memoria dinámica
  Buffer(size_t s) : size(s) {
    data = new char[size];
    std::cout << "Buffer reservado" << std::endl;
  }

  //-- Método para usar el buffer
  void write(char value) {
    for (size_t i = 0; i < size; ++i) {
      data[i] = value;
    }
  }

  //-- No se define destructor
};

int main() {
  //-- Creación del objeto en el stack
  Buffer buffer(1024);
  buffer.write('A');

  //-- Al salir del alcance no se libera la memoria reservada
  return 0;
}
```

Codificación recomendada

Gestión adecuada de memoria dinámica dentro de una clase que reserva recursos en el constructor y define explícitamente un destructor para liberarlos. Se busca aplicar el principio RAII y evitar fugas de memoria asegurando la liberación del recurso al finalizar el ciclo de vida del objeto.

```
#include <iostream>

class Buffer {
  //-- Puntero a memoria dinámica
  char* data;
```

```
  size_t size;

public:
  //-- Constructor que reserva memoria dinámica
  Buffer(size_t s) : data(nullptr), size(s) {
    data = new char[size];
    std::cout << "Buffer reservado" << std::endl;
  }

  //-- Destructor que libera el recurso
  ~Buffer() {
    delete[] data;
    std::cout << "Buffer liberado" << std::endl;
  }

  //-- Método para usar el buffer
  void write(char value) {
    for (size_t i = 0; i < size; ++i) {
      data[i] = value;
    }
  }
};

int main() {
  //-- Creación del objeto en el stack
  Buffer buffer(1024);
  buffer.write('A');

  //-- Al salir del alcance se invoca el destructor automáticamente
  return 0;
}
```

2.9.3.1 Descripción

Toda clase que gestione directamente un recurso finito del sistema debe definir un destructor seguro que se responsabilice de su liberación siguiendo los principios RAII. Esta práctica evita fugas de memoria, descriptores abiertos y estados inconsistentes en el sistema. Un destructor correctamente implementado garantiza que los recursos asociados a una instancia se liberen de forma ordenada y determinista cuando el objeto abandona su ámbito, se destruye manualmente o se produce una excepción.

Un manejo inadecuado de los recursos puede provocar problemas de seguridad como fugas de memoria explotables, accesos a memoria ya liberada, estados corruptos y bloqueos del sistema. Un destructor mal implementado o inexistente también puede conducir a dobles liberaciones, condición que puede ser aprovechada por atacantes para manipular estructuras internas del heap. Por ello, la definición correcta del

destructor es una medida esencial para preservar la integridad y coherencia en aplicaciones críticas.

Para que un destructor sea considerado seguro y robusto, debe cumplir varios principios: liberar exactamente los recursos adquiridos, mantener un estado válido y coherente después de la destrucción parcial y evitar comportamientos indefinidos, especialmente ante excepciones. También debe anticipar posibles casos de uso erróneo, como destrucciones múltiples o liberaciones prematuras. Además, el destructor debe ser simple, determinista y no debe lanzar excepciones en ninguna circunstancia.

En escenarios más complejos, como clases que gestionan múltiples recursos o estructuras compartidas, el destructor debe coordinar la liberación sin introducir condiciones de carrera, especialmente en contextos multihilo. Si la clase es base, el destructor debe ser virtual para evitar fugas derivadas de la destrucción incompleta al eliminar objetos a través de punteros a la clase base. Para contenedores personalizados o buffers alineados, deben respetarse los requisitos de alineación durante la liberación. El uso de smart pointers como `unique_ptr` o `shared_ptr` reduce la carga de implementar manualmente la lógica del destructor, minimizando errores.

Un destructor jamás debe lanzar excepciones. Si durante la liberación de un recurso pueden producirse errores, estos deben registrarse o manejarse silenciosamente. En contextos críticos, puede utilizarse una función auxiliar no destructora que genere registros detallados del fallo sin comprometer la estabilidad del programa. Esta regla es especialmente crítica cuando la destrucción ocurre durante el desenrollado de pila tras una excepción previa.

El destructor debe garantizar que, una vez liberado el recurso, el objeto no deje referencias colgantes ni exponga memoria previamente usada a través de interfaces públicas. Debe analizarse si el recurso requiere un borrado seguro, especialmente cuando maneje información sensible. Para estructuras como buffers criptográficos, puede ser necesario sobrescribir la memoria antes de liberarla si la plataforma no garantiza su limpieza.

Riesgos asociados

- Fuga de memoria (memory leaks), al no liberar adecuadamente la memoria asignada dinámicamente.
- Fuga de recursos del sistema, como archivos abiertos, sockets, o conexiones a bases de datos no cerradas.
- Pérdida de información o corrupción de datos si los recursos no se cierran o guardan correctamente.
- Comportamiento indefinido al reutilizar recursos no liberados correctamente.
- Crecimiento continuo en el consumo de memoria o recursos, que puede derivar en la negación de servicio (DoS).

- Imposibilidad de gestionar correctamente la copia o traslado (copy/move) de objetos de la clase.
- Errores de doble liberación si la gestión manual de los recursos se realiza en otro lugar y no en el destructor.
- Posibilidad de comprometer la estabilidad y seguridad de la aplicación, facilitando ataques como el uso después de liberar (use-after-free).

Buenas prácticas

- Implementar un destructor para liberar explícitamente los recursos adquiridos (memoria dinámica, archivos, sockets, etc.).

```
class FileHandler {
  FILE* file;
public:
  FileHandler(const char* path) {
    file = fopen(path, "r");
  }
  //-- Recurso no liberado al destruir el objeto
  ~FileHandler() {
  }
};
```

```
class FileHandlerSafe {
  FILE* file;
public:
  FileHandlerSafe(const char* path) {
    file = fopen(path, "r");
  }
  //-- Liberación explícita del recurso adquirido
  ~FileHandlerSafe() {
    if (file) {
      fclose(file);
    }
  }
};
```

- Aplicar el principio RAII y asegurar que el destructor sea responsable único de la liberación del recurso.

```
class Buffer {
  int* data;
public:
  Buffer(size_t size) {
    data = new int[size];
  }
```

```
  //-- Liberación manual externa propensa a omisiones
  void release() {
    delete[] data;
  }
};
```

```
class BufferSafe {
  int* data;
public:
  BufferSafe(size_t size) {
    data = new int[size];
  }
  //-- Liberación automática del recurso en el destructor
  ~BufferSafe() {
    delete[] data;
  }
};
```

- Declarar el destructor como virtual cuando la clase esté destinada a ser usada como clase base polimórfica.

```
class Base {
public:
  //-- Destructor no virtual en jerarquía polimórfica
  ~Base() { }
};

class Derived : public Base {
  int* data;
public:
  Derived() {
    data = new int[10];
  }
  ~Derived() {
    delete[] data;
  }
};
```

```
class BaseSafe {
public:
  //-- Destructor virtual para asegurar destrucción completa
  virtual ~BaseSafe() { }
};
```

2.9.3.2 Estándares metodológicos

- **C++ Core-C.30**: Define a destructor if a class needs an explicit action at object

destruction.

- **C++ Core-C.33**: If a class has an owning pointer member, define a destructor.

2.9.4 [cpp.cls.4] - Extremar la precaución del uso de una propiedad de clase con punteros o referencias

ID	cpp.cls.4	Uso	Recomendado
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se debe gestionar con extremo cuidado cualquier propiedad de una clase que utilice **punteros** o **referencias**, ya que estas construcciones pueden introducir vulnerabilidades graves si no se definen reglas claras de propiedad, liberación de memoria, transferencia de recursos y ciclos de vida. Hay que asegurar que el objeto mantenga un modelo coherente, seguro y auditable del recurso al que apunta, evitando accesos inválidos, fugas de memoria, dobles liberaciones o aliasing no controlado que derive en corrupción de memoria.

Codificación insegura

Gestión de la propiedad de memoria en una clase que almacena un puntero dinámico. Se busca mostrar cómo la falta de definición clara de la propiedad del recurso puede provocar liberaciones duplicadas, copias superficiales inseguras y comportamiento indefinido al copiar objetos.

```
#include <iostream>
#include <cstring>

//-- Clase que gestiona manualmente un puntero sin definir correctamente la
propiedad
class Buffer {
public:
  char* data;

  //-- Constructor que reserva memoria
  Buffer(const char* text) {
    data = new char[strlen(text) + 1];
    std::strcpy(data, text);
  }

  //-- Destructor que libera la memoria
  ~Buffer() {
    delete[] data;
  }
};
```

```
int main() {
  //-- Se crea un objeto con memoria dinámica
  Buffer b1("Hola mundo");

  //-- Copia superficial: ambos objetos apuntan al mismo bloque de memoria
  Buffer b2 = b1;

  //-- Se imprime el contenido
  std::cout << b2.data << std::endl;

  //-- Al finalizar, ambos destructores intentarán liberar el mismo puntero
  //-- Esto provoca doble liberación y comportamiento indefinido
  return 0;
}
```

Codificación recomendada

Gestión segura de la propiedad del recurso utilizando RAII y tipos estándar que modelan claramente la propiedad exclusiva. Se muestra cómo evitar copias inseguras y dobles liberaciones usando `std::unique_ptr` y deshabilitando la copia.

```
#include <iostream>
#include <memory>
#include <cstring>

//-- Clase que define claramente la propiedad exclusiva del recurso
class Buffer {
private:
  std::unique_ptr<char[]> data;

public:
  //-- Constructor que reserva memoria
  Buffer(const char* text) {
    size_t len = std::strlen(text) + 1;
    data = std::make_unique<char[]>(len);
    std::strcpy(data.get(), text);
  }

  //-- Se elimina la copia para evitar duplicar la propiedad
  Buffer(const Buffer&) = delete;
  Buffer& operator=(const Buffer&) = delete;

  //-- Se permite el movimiento para transferir la propiedad
  Buffer(Buffer&&) noexcept = default;
  Buffer& operator=(Buffer&&) noexcept = default;

  //-- Método para acceder al contenido
  const char* get() const {
```

```
    return data.get();
  }
};

int main() {
  //-- Se crea un objeto con propiedad exclusiva del recurso
  Buffer b1("Hola mundo");

  //-- Transferencia segura de propiedad mediante movimiento
  Buffer b2 = std::move(b1);

  //-- Se imprime el contenido
  std::cout << b2.get() << std::endl;

  //-- La memoria se libera automáticamente una sola vez
  return 0;
}
```

2.9.4.1 Descripción

El uso de propiedades internas de clase representadas mediante punteros o referencias exige un modelo de propiedad coherente y seguro. Una clase que almacena una dirección de memoria puede convertirse fácilmente en origen de múltiples vulnerabilidades si no se define claramente quién es responsable de administrar la vida útil del recurso. El diseño inseguro de estas propiedades suele provocar errores complejos como desreferenciaciones inválidas, dobles liberaciones, fugas de memoria, aliasing no intencionado o modificaciones inesperadas del estado interno. Para evitar estos problemas, es fundamental establecer una estrategia rigurosa que cubra aspectos de propiedad, transferencia, restricciones de aliasing y validación del estado del puntero o referencia.

Dentro de las buenas prácticas seguras, la propiedad del recurso debe quedar explícita. Esto implica definir si la clase **posee** el recurso, si **lo comparte** con otros componentes o si simplemente lo **observa** sin responsabilidades sobre su liberación. En clases modernas de C++ es habitual recurrir a contenedores inteligentes como `std::unique_ptr`, `std::shared_ptr` o `std::weak_ptr` para prevenir fugas y garantizar normas claras de propiedad. Cuando dichas estructuras no se utilizan, se deben implementar controles manuales que impidan condiciones inseguras. Además, cuando se utilicen referencias, debe garantizarse que el objeto referenciado viva al menos tanto como la clase que lo almacena.

El diseño de clases con punteros exige abordar explícitamente el ciclo de vida del recurso. La clase debe evitar asumir que un puntero recibido es siempre válido. Se recomienda la inicialización determinista, ya sea con `nullptr` o con valores bien definidos, evitando situaciones donde un puntero quede colgante tras la destrucción del recurso referenciado. El constructor, el destructor y los operadores de copia y

movimiento deben alinearse con las reglas de propiedad establecidas. Es una práctica crítica evitar inconsistencias entre estas funciones, ya que provocarían duplicación de liberaciones o pérdidas de memoria.

Cuando se usen punteros crudos estos deben ser tratados con estrictas medidas de seguridad. Se sugiere no exponer punteros públicos como propiedades accesibles sin controles, ya que pueden permitir la modificación del estado interno de forma inesperada. También se recomienda evitar que la clase asuma la propiedad de un puntero recibido por parámetros sin confirmación explícita. Los punteros crudos deben ser considerados sólo para acceso sin propiedad o para integraciones donde no se pueda usar memoria gestionada automáticamente. El uso incorrecto de punteros puede derivar en corrupción de memoria, un tipo de vulnerabilidad extremadamente grave.

Las referencias no pueden ser nulas y deben inicializarse en el constructor. Esto implica que una clase que almacena una referencia depende completamente de que el autor del código garantice la persistencia del objeto referenciado. Las referencias no deben utilizarse cuando exista cualquier posibilidad de que el objeto referenciado desaparezca antes de la instancia propietaria. En estos casos es más seguro recurrir a punteros controlados, punteros débiles o soluciones de almacenamiento alternativo.

Riesgos asociados

- Fugas de memoria (memory leaks) por no liberar correctamente la memoria asignada a punteros.
- Condiciones de doble liberación (double free) al liberar memoria en más de un lugar o varias veces.
- Accesso a memoria no válida (dangling pointers) si un puntero queda apuntando a espacio liberado.
- Violaciones de acceso (segmentation fault) por desreferenciar punteros o referencias nulas o inválidas.
- Corrupción de memoria debido a escrituras no controladas a través de punteros mal gestionados.
- Propagación involuntaria de datos sensibles si un objeto comparte memoria sin una copia segura.
- Problemas de concurrencia si múltiples instancias acceden o modifican memoria compartida sin sincronización.
- Falta de claridad en la propiedad de los datos, dificultando el mantenimiento y la seguridad del código.
- Errores lógicos al modificar o destruir instancias referenciadas por varios objetos (p. ej. double delete).

Buenas prácticas

- Definir explícitamente la propiedad del recurso cuando se utilicen punteros en una clase (aplicar RAII).

```
//-- Uso de puntero crudo sin gestión clara de propiedad
class Recurso {
public:
  int* datos;
  Recurso() {
    datos = new int(10);
  }
};
```

```
//-- Uso de std::unique_ptr para expresar propiedad exclusiva
class Recurso {
public:
  std::unique_ptr datos;
  Recurso() : datos(std::make_unique(10)) {}
};
```

- Evitar compartir propiedad con punteros crudos; utilizar mecanismos seguros para propiedad compartida.

```
//-- Compartición de puntero crudo sin reglas claras de liberación
class Nodo {
public:
  int* valor;
  Nodo(int* v) : valor(v) {}
};
```

```
//-- Uso de std::shared_ptr para propiedad compartida controlada
class Nodo {
public:
  std::shared_ptr valor;
  Nodo(std::shared_ptr v) : valor(v) {}
};
```

- Evitar almacenar referencias como miembros si no se garantiza su ciclo de vida.

```
//-- Almacenamiento de referencia sin garantizar tiempo de vida
class Usuario {
  const std::string& nombre;
public:
  Usuario(const std::string& n) : nombre(n) {}
};
```

```
//-- Almacenamiento por valor para controlar el ciclo de vida
class Usuario {
  std::string nombre;
public:
  Usuario(const std::string& n) : nombre(n) {}
};
```

- Definir correctamente las operaciones de copia y movimiento cuando la clase gestiona recursos dinámicos.

```
//-- Clase con puntero crudo sin regla de tres/cinco
class Buffer {
  int* datos;
public:
  Buffer(size_t n) {
    datos = new int[n];
  }
};
```

```
//-- Eliminación de copia y habilitación de movimiento para propiedad exclusiva
class Buffer {
  std::unique_ptr datos;
public:
  Buffer(size_t n) : datos(std::make_unique(n)) {}
  Buffer(const Buffer&) = delete;
  Buffer& operator=(const Buffer&) = delete;
  Buffer(Buffer&&) noexcept = default;
  Buffer& operator=(Buffer&&) noexcept = default;
};
```

2.9.4.2 Estándares metodológicos

- **C++ Core-C.32**: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning.

2.9.5 [cpp.cls.5] - Establecer al destructor de clase público y virtual, o protegido y no virtual

ID	cpp.cls.5	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La configuración correcta del destructor en clases destinadas a heredarse es esencial para evitar comportamientos indefinidos, fugas de recursos y fallos de seguridad asociados con destrucciones incompletas. Establecer el destructor como **público y**

virtual cuando la clase se utiliza mediante punteros a la clase base, o como **protegido y no virtual** cuando la clase no debe ser destruida externamente, evita escenarios donde un destructor incorrecto provoque corrupción de memoria o violaciones de integridad.

Codificación insegura

Uso de herencia y polimorfismo donde una clase base es utilizada mediante un puntero base para destruir un objeto derivado. Se busca observar qué ocurre cuando el destructor de la clase base no está correctamente definido para soportar destrucción polimórfica, lo que puede provocar comportamiento indefinido o liberación incompleta de recursos.

```
#include <iostream>

//-- Clase base con destructor no virtual (problema en uso polimórfico)
class Base {
public:
  Base() {
    std::cout << "Constructor Base" << std::endl;
  }

  //-- Destructor no virtual
  ~Base() {
    std::cout << "Destructor Base" << std::endl;
  }
};

//-- Clase derivada que gestiona recursos
class Derivada : public Base {
public:
  Derivada() {
    data = new int[5];
    std::cout << "Constructor Derivada" << std::endl;
  }

  ~Derivada() {
    delete[] data;
    std::cout << "Destructor Derivada" << std::endl;
  }

private:
  int* data;
};

int main() {
  //-- Uso polimórfico
  Base* obj = new Derivada();

  //-- Eliminación a través de puntero base
```

```
  delete obj;
  return 0;
}
```

Codificación recomendada

Uso correcto de herencia y polimorfismo donde la clase base define un destructor público y virtual, permitiendo que la destrucción a través de un puntero base invoque correctamente los destructores de las clases derivadas y garantice la liberación adecuada de recursos.

```
#include <iostream>

//-- Clase base con destructor público y virtual
class Base {
public:
  Base() {
    std::cout << "Constructor Base" << std::endl;
  }

  //-- Destructor virtual para permitir destrucción polimórfica segura
  virtual ~Base() {
    std::cout << "Destructor Base" << std::endl;
  }
};

//-- Clase derivada que gestiona recursos
class Derivada : public Base {
public:
  Derivada() {
    data = new int[5];
    std::cout << "Constructor Derivada" << std::endl;
  }

  ~Derivada() override {
    delete[] data;
    std::cout << "Destructor Derivada" << std::endl;
  }

private:
  int* data;
};

int main() {
  //-- Uso polimórfico seguro
  Base* obj = new Derivada();

  //-- Eliminación correcta gracias al destructor virtual
```

```
  delete obj;
  return 0;
}
```

2.9.5.1 Descripción

Las reglas para la definición de destructores tienen un impacto directo en la seguridad del software, especialmente en contextos donde las clases participan en jerarquías de herencia o manejan recursos críticos como memoria dinámica, descriptores de archivos, sockets o manejadores de sistemas externos. La destrucción incorrecta de objetos derivando en destructores no invocados, comportamiento indefinido o fugas, puede exponer la aplicación a vulnerabilidades de integridad, disponibilidad e incluso confidencialidad. Por ello, la aplicación rigurosa de principios como la definición de destructores **públicos y virtuales** para clases base polimórficas, o **protegidos y no virtuales** para clases que no deben destruirse externamente, resulta altamente relevante para un diseño seguro.

El destructor de una clase controla la fase final del ciclo de vida del objeto y la liberación de recursos. Cuando una clase se utiliza como tipo base y se elimina mediante un puntero al tipo base, el destructor debe ser virtual para asegurar que se destruya correctamente el objeto dinámico completo. No hacerlo provoca un comportamiento indefinido que puede desencadenar efectos como memoria no liberada, destrucción parcial de estado interno, inconsistencias en estructuras de datos y fallos impredecibles. Estos fallos pueden ser explotados para manipular estructuras de memoria o alterar el funcionamiento lógico del software.

Por el contrario, hay casos donde no se desea permitir la destrucción directa a través de punteros, como cuando una clase base actúa como una interfaz o cuando el ciclo de vida de los objetos debe ser gestionado por fábricas u otros mecanismos de control. En estos casos, un destructor protegido y no virtual ayuda a restringir esta operación, evitando una destrucción involuntaria o insegura desde contexto externo.

La regla se basa en dos escenarios principales:

- **Clases destinadas a herencia polimórfica:** Cuando una clase será usada como base y los objetos derivados podrán ser administrados a través de punteros o referencias a la clase base.
- **Clases no diseñadas para manejo de ciclos de vida externos:** Cuando la destrucción debe permanecer controlada, evitando que un consumidor elimine un objeto arbitrariamente.

En el primer escenario, el destructor debe declararse como `virtual` para garantizar que el destructor del objeto derivado sea invocado. Cuando el destructor no es virtual, eliminar un objeto derivado usando un puntero al tipo base termina invocando solo el destructor base, sin ejecutar el destructor derivado. Esto deja partes del objeto en un

estado no destruido correctamente. Si la clase derivada gestiona memoria, buffers, punteros crudos o recursos del SO, esto puede derivar en fugas o corrupción.

Los errores en la gestión del ciclo de vida pueden ser explotados en escenarios donde se accede a memoria ya liberada, se provoca doble liberación o se mantienen referencias colgantes. Estos fallos se clasifican frecuentemente como vulnerabilidades de integridad o disponibilidad y pueden usarse para provocar cierres forzados, corrupción de heap o modificación de datos internos. El simple hecho de garantizar que el destructor adecuado se ejecuta completamente contribuye a mantener el sistema en estado seguro.

Riesgos asociados

- Fugas de memoria debido a que los destructores base no son virtuales y la destrucción vía puntero base no libera correctamente los recursos.
- Ejecución incompleta del destructor de clases derivadas, resultando en liberación parcial de recursos.
- Comportamiento indefinido al eliminar objetos de una jerarquía de clases a través de punteros o referencias a la clase base.
- Pérdida de recursos críticos como locks, archivos abiertos o conexiones de red al no invocar destructores de las clases derivadas.
- Vulnerabilidad a ataques de denegación de servicio (DoS) por fuga de recursos debido a destrucción incompleta de objetos complejos.
- Exposición accidental de la interfaz de destrucción (si el destructor debe ser protegido, pero es público), permitiendo destrucción no controlada por código externo.
- Dificultad para implementar el patrón RAII de manera segura, comprometiendo la administración automática de recursos.
- Fallo en la correcta gestión de herencia polimórfica, generando errores sutiles y difíciles de detectar en grandes bases de código.

Buenas prácticas

- Declarar el destructor como público y virtual cuando la clase esté diseñada para ser usada como clase base polimórfica.

```
//-- Eliminación polimórfica sin destructor virtual en clase base
class Base {
public:
  ~Base() {}
};
class Derivada : public Base {
public:
```

```
  ~Derivada() { liberar(); }
  void liberar();
};
void f() {
  Base* p = new Derivada();
  delete p;
}
```

```
//-- Destructor público y virtual para permitir destrucción polimórfica segura
class Base {
public:
  virtual ~Base() {}
};
class Derivada : public Base {
public:
  ~Derivada() override { liberar(); }
  void liberar();
};
void f() {
  Base* p = new Derivada();
  delete p;
}
```

- Definir el destructor como protegido y no virtual cuando la clase no esté pensada para eliminación polimórfica, evitando su destrucción a través de punteros base.

```
//-- Destructor público no virtual permite destrucción indebida vía puntero base
class Base {
public:
  ~Base() {}
};
class Derivada : public Base {
public:
  ~Derivada() {}
};
void f() {
  Base* p = new Derivada();
  delete p;
}
```

```
//-- Destructor protegido no virtual restringe la destrucción externa
class Base {
protected:
  ~Base() {}
};
class Derivada : public Base {
public:
```

```
  ~Derivada() {}
};
void f() {
  Derivada* p = new Derivada();
  delete p;
}
```

- Marcar explícitamente el destructor virtual con `override` en clases derivadas para reforzar la intención y evitar errores de mantenimiento.

```
//-- Destructor derivado sin override puede ocultar errores de firma
class Base {
public:
  virtual ~Base() {}
};
class Derivada : public Base {
public:
  ~Derivada() {}
};
```

```
//-- Uso de override para garantizar coincidencia con la jerarquía
class Base {
public:
  virtual ~Base() {}
};
class Derivada : public Base {
public:
  ~Derivada() override {}
};
```

2.9.5.2 Estándares metodológicos

- **C++ Core-C.35**: A base class destructor should be either public and virtual, or protected and non-virtual.
- **CERT/CPP-OOP52**: Do not delete a polymorphic object without a virtual destructor.

2.9.6 [cpp.cls.6] - Usar el constructor para crear un objeto completamente inicializado

ID	cpp.cls.6	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Inicializar completamente todos los atributos de un objeto dentro de su constructor evita estados parciales o no definidos que puedan introducir fallos de seguridad, comportamientos indefinidos o vulnerabilidades explotables. Al asegurar que un objeto se crea siempre en un estado válido, se reducen riesgos como fugas de memoria, accesos indebidos a punteros no inicializados, uso de valores basura y errores lógicos que afectan la robustez del software.

Codificación insegura

Se define una clase que representa una cuenta bancaria. La intención es mostrar cómo un objeto puede quedar en un estado inconsistente cuando no se utiliza el constructor para inicializar completamente sus atributos, obligando a realizar configuraciones posteriores y aumentando el riesgo de uso incorrecto.

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

class CuentaBancaria {
public:
  std::string titular;
  double saldo;

  CuentaBancaria() {
    //-- Constructor por defecto que no garantiza inicialización válida
    saldo = -1; // Valor inválido para simular estado inconsistente
  }

  void mostrar() {
    //-- Muestra el estado actual de la cuenta
    std::cout << "Titular: " << titular << std::endl;
    std::cout << "Saldo: " << saldo << std::endl;
  }
};

int main() {
  //-- Creación del objeto sin inicialización completa
  CuentaBancaria cuenta;

  //-- Inicialización manual posterior, propensa a errores
  cuenta.titular = "Carlos";
```

```
  cuenta.saldo = 1000.0;

  cuenta.mostrar();
  return 0;
}
```

Codificación recomendada

Se define la misma clase asegurando que el objeto quede completamente inicializado desde el momento de su creación mediante un constructor con parámetros y lista de inicialización. Se busca aprender cómo garantizar estados válidos y reducir riesgos de uso indebido o valores inconsistentes.

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

class CuentaBancaria {
private:
  std::string titular;
  double saldo;

public:
  //-- Constructor que garantiza inicialización completa y válida
  CuentaBancaria(const std::string& nombre, double saldoInicial)
    : titular(nombre), saldo(saldoInicial) {
    //-- Validación básica del saldo inicial
    if (saldoInicial < 0) {
      throw std::invalid_argument("El saldo no puede ser negativo");
    }
  }

  void mostrar() const {
    //-- Muestra el estado actual de la cuenta
    std::cout << "Titular: " << titular << std::endl;
    std::cout << "Saldo: " << saldo << std::endl;
  }
};

int main() {
  //-- Creación del objeto completamente inicializado
  CuentaBancaria cuenta("Carlos", 1000.0);
  cuenta.mostrar();
  return 0;
}
```

2.9.6.1 Descripción

Un constructor define el estado inicial de un objeto, estableciendo valores concretos para todos sus atributos y evitando que variables internas queden en estados indeterminados. El acceso a memoria no inicializada puede causar comportamientos no definidos, que incluyen lecturas de datos aleatorios, escritura en direcciones no esperadas o ejecución de rutas de código no planeadas. Estos efectos no solo comprometen la fiabilidad del sistema, sino que pueden derivar en vulnerabilidades serias capaces de ser explotadas por atacantes que manipulen las condiciones de ejecución del programa.

La inicialización segura implica que cada miembro del objeto reciba un valor válido durante la construcción. Para asegurarlo, es recomendable usar la lista de inicialización del constructor, evitar la lógica compleja antes de establecer estados internos y prohibir accesos a miembros antes de ser inicializados. Además, se debe priorizar la inicialización sobre la asignación, ya que esta última puede introducir sobrecostes o ventanas de inconsistencia.

Riesgos asociados

- Lectura de variables miembro no inicializadas, provocando comportamientos indefinidos.
- Acceso a datos con valores basura o aleatorios, pudiendo causar fallos inesperados.
- Vulnerabilidades de seguridad explotables, como divulgación de información sensible desde memoria no inicializada.
- Uso de punteros salvajes (dangling pointers) o referencias no válidas.
- Fugas de memoria por objetos construidos parcialmente y fallida gestión de recursos.
- Problemas de consistencia y corrupción de datos al usar miembros no inicializados.
- Errores difíciles de detectar en tiempo de ejecución, incrementando la complejidad de depuración.
- Condiciones de carrera al usar objetos en entornos multihilo sin inicialización adecuada.
- Comportamiento imprevisible de funciones o métodos que dependen del estado completo del objeto.

Buenas prácticas

- Inicializar todos los atributos obligatorios mediante listas de inicialización en el constructor.

```
//-- Inicialización diferida fuera del constructor
class Usuario {
  std::string nombre;
  int edad;
public:
  Usuario() {}
  void inicializar(const std::string& n, int e) {
    nombre = n;
    edad = e;
  }
};
```

```
//-- Inicialización completa usando lista de inicialización
class Usuario {
  std::string nombre;
  int edad;
public:
  Usuario(const std::string& n, int e)
    : nombre(n), edad(e) {}
};
```

- Evitar constructores por defecto que permitan objetos en estado inconsistente.

```
//-- Objeto creado sin valores válidos obligatorios
class Conexion {
  std::string host;
  int puerto;
public:
  Conexion() {}
};
```

```
//-- Constructor que exige parámetros mínimos para estado válido
class Conexion {
  std::string host;
  int puerto;
public:
  Conexion(const std::string& h, int p)
    : host(h), puerto(p) {}
};
```

- Validar los parámetros dentro del constructor antes de completar la creación del objeto.

```
//-- Asignación sin validación de parámetros
class Cuenta {
  double saldo;
public:
```

```
  Cuenta(double s) : saldo(s) {}
};
```

```
//-- Validación de parámetros durante la construcción
class Cuenta {
  double saldo;
public:
  Cuenta(double s) {
    if (s < 0) {
      throw std::invalid_argument("Saldo inválido");
    }
    saldo = s;
  }
};
```

2.9.6.2 Estándares metodológicos

- **C++ Core-C.41**: A constructor should create a fully initialized object.

2.9.7 [cpp.cls.7] - Establecer en el constructor todos los invariantes de la clase

ID	cpp.cls.7	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Una clase debe garantizar que todos sus invariantes internos queden correctamente inicializados y validados dentro del constructor, evitando estados incompletos o inconsistentes. Al asegurar que ningún objeto pueda existir sin cumplir sus condiciones esenciales, se reduce el riesgo de accesos indebidos a memoria, comportamientos indefinidos, corrupción de datos y vulnerabilidades asociadas al uso de objetos mal formados. Además, refuerza la seguridad orientada a objetos asegurando que cada instancia nace en un estado válido y estable.

Codificación insegura

Gestión de una clase que representa una cuenta bancaria donde el saldo no debe ser negativo. Se muestra cómo la falta de validación en el constructor permite crear objetos en un estado inválido, lo que rompe los invariantes desde el momento de la construcción y puede generar errores lógicos posteriores.

```
#include <iostream>
#include <stdexcept>

//-- Clase que no garantiza el invariante saldo >= 0 en el constructor
class Account {
```

```
  double balance;

public:
  //-- Constructor que recibe un saldo inicial sin validarlo
  Account(double initialBalance) {
    balance = initialBalance;
  }

  //-- Método para obtener el saldo actual
  double getBalance() const {
    return balance;
  }
};

int main() {
  //-- Se crea una cuenta con saldo negativo, violando el invariante
  Account acc(-100.0);

  //-- Se muestra un estado inválido permitido por el constructor
  std::cout << "Saldo actual: " << acc.getBalance() << std::endl;
  return 0;
}
```

Codificación recomendada

Gestión de la misma clase de cuenta bancaria asegurando que el saldo nunca sea negativo. El constructor valida el dato de entrada y garantiza que el objeto se crea siempre en un estado consistente, estableciendo el invariante desde el primer momento.

```
#include <iostream>
#include <stdexcept>

//-- Clase que garantiza el invariante saldo >= 0 desde el constructor
class Account {
  double balance;

public:
  //-- Constructor que valida el saldo inicial
  Account(double initialBalance) {
    if (initialBalance < 0.0) {
      throw std::invalid_argument("El saldo inicial no puede ser negativo");
    }
    balance = initialBalance;
  }

  //-- Método para obtener el saldo actual
  double getBalance() const {
```

```
    return balance;
  }
};

int main() {
  try {
    //-- Intento de crear una cuenta con saldo inválido
    Account acc(-100.0);
    std::cout << "Saldo actual: " << acc.getBalance() << std::endl;
  } catch (const std::exception& ex) {
    //-- Manejo controlado del error al violar el invariante
    std::cout << "Error al crear la cuenta: " << ex.what() << std::endl;
  }
  return 0;
}
```

2.9.7.1 Descripción

Un invariante de clase es cualquier condición lógica que debe cumplirse para que un objeto pueda considerarse válido y seguro de usar. La posibilidad de que existan estados parciales, de que se invoquen métodos antes de que el objeto esté completamente construido o de que una excepción interrumpa la inicialización, crea un espacio de vulnerabilidades que podría evitarse. Un constructor debe dejar el objeto en un estado consistente garantizando que sus atributos tienen valores seguros, que sus dependencias han sido correctamente configuradas y que cualquier recurso crítico ha sido adquirido de forma exitosa.

C++ permite controlar memoria manualmente, ejecutar código durante la inicialización de atributos, utilizar herencia, lanzar excepciones y administrar recursos nativos del sistema. Cada uno de estos elementos introduce puntos de fallo si los invariantes no están claramente definidos y verificados. Un objeto cuyo estado no es válido puede causar problemas como accesos ilegales a punteros, corrupción de memoria, fugas, comportamientos indefinidos o incluso condiciones aprovechables para ataques. Por este motivo, establecer los invariantes en el constructor forma parte de las bases de la programación defensiva.

Los invariantes deben definirse con claridad y aplicarse siempre antes de que el objeto quede disponible para uso externo. Esto implica que los constructores deben realizar toda la inicialización necesaria y, en caso de fallo, la construcción debe fracasar de forma segura, idealmente mediante una excepción bien controlada que asegure que no queda memoria sucia o recursos asignados de forma parcial. Es fundamental que ningún método público dependa de una condición que no haya sido verificada dentro del constructor. Es igualmente importante que estos invariantes se mantengan durante toda la vida del objeto, y que cualquier modificación posterior preserve dichas condiciones.

Los invariantes suelen estar relacionados con la validez de los miembros internos de la clase.

Algunos ejemplos comunes incluyen:

- **Punteros no nulos:** Cuando la clase depende de un recurso externo cuyo puntero debe existir.
- **Rango válido:** Valores numéricos que deben permanecer dentro de un intervalo seguro.
- **Memoria asignada correctamente:** Garantizar que los buffers están inicializados y no contienen valores indeterminados.
- **Recursos adquiridos:** Archivos abiertos, sockets conectados o locks obtenidos.
- **Configuraciones coherentes:** Relaciones lógicas entre atributos que deben cumplirse para evitar comportamientos incorrectos.

Riesgos asociados

- Estado inconsistente del objeto: El objeto puede crearse en un estado no válido, lo que puede provocar fallos durante la ejecución.
- Errores en tiempo de ejecución: El acceso a miembros de la clase no inicializados puede causar errores difíciles de detectar.
- Violación de encapsulamiento: Al exponer o permitir la modificación de invariantes fuera del constructor, se debilita la protección de la lógica interna de la clase.
- Comportamiento indefinido: Uso de variables o punteros no inicializados puede llevar a acceso a memoria inválida o corrupción de datos.
- Explotación de vulnerabilidades: Un atacante podría crear instancias malformadas deliberadamente aprovechando la falta de inicialización, generando brechas de seguridad.
- Dificultad para el mantenimiento del código: La lógica dispersa de inicialización hace más complejo entender y modificar el código de manera segura.
- Desbordamiento de buffer o acceso fuera de límites: Si los invariantes definen límites, no establecerlos adecuadamente puede facilitar este tipo de errores de seguridad.
- Perdida de fiabilidad en validaciones internas: Las funciones miembro que dependen de los invariantes pueden devolver resultados incorrectos o fallar.

Buenas prácticas

- Inicializar todos los atributos obligatorios en la lista de inicialización del

constructor para garantizar invariantes desde la creación del objeto.

```
//-- Inicialización diferida que puede dejar el objeto en estado inválido
class Usuario {
  std::string nombre;
  int edad;
public:
  Usuario() {}
  void setEdad(int e) { edad = e; }
};
```

```
//-- Inicialización completa en la lista de inicialización
class Usuario {
  std::string nombre;
  int edad;
public:
  Usuario(const std::string& n, int e) : nombre(n), edad(e) {}
};
```

- Validar parámetros en el constructor y lanzar excepciones si no cumplen las restricciones del dominio.

```
//-- Constructor que acepta valores inválidos sin validación
class Cuenta {
  double saldo;
public:
  Cuenta(double s) : saldo(s) {}
};
```

```
//-- Validación de invariante antes de asignar el estado interno
class Cuenta {
  double saldo;
public:
  Cuenta(double s) : saldo(s) {
    if (s < 0) {
      throw std::invalid_argument("Saldo negativo");
    }
  }
};
```

- Evitar estados parcialmente construidos utilizando miembros constantes o referencias cuando formen parte del invariante.

```
//-- Miembro modificable que puede romper el invariante tras la construcción
class Configuracion {
  int puerto;
public:
```

```
  Configuracion(int p) : puerto(p) {}
  void setPuerto(int p) { puerto = p; }
};
```

```
//-- Miembro constante que garantiza el valor establecido en el constructor
class Configuracion {
  const int puerto;
public:
  Configuracion(int p) : puerto(p) {
    if (p <= 0 || p > 65535) {
      throw std::out_of_range("Puerto inválido");
    }
  }
};
```

2.9.7.2 Estándares metodológicos

- **C++ Core-C.40**: Define a constructor if a class has an invariant.

2.9.8 [cpp.cls.8] - Lanzar una excepción si un constructor no puede crear un objeto válido

ID	cpp.cls.8	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Cuando un constructor no pueda inicializar correctamente un objeto en un estado válido, debe lanzar una excepción en lugar de permitir la creación de un objeto parcialmente construido o inválido. Esto evita estados inconsistentes, reduce errores lógicos complejos y mejora la robustez del sistema. Al utilizar excepciones como mecanismo de señalización temprana, se promueve un flujo de control seguro y predecible, facilitando la detección y manejo de fallos críticos en tiempo de ejecución y fortaleciendo la integridad del sistema.

Codificación insegura

Se muestra una clase que representa una conexión a un recurso crítico. La intención es construir el objeto incluso cuando el recurso no está disponible, permitiendo que el objeto quede en un estado inválido. Se pretende observar qué ocurre cuando no se garantiza la validez del objeto tras el constructor y cómo esto puede provocar fallos en tiempo de ejecución.

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

```
class FileLoader {
  //-- Ruta del archivo
  std::string path;
  //-- Indicador de estado interno
  bool valid;

public:
  //-- Constructor que no valida correctamente la creación
  FileLoader(const std::string& filePath) : path(filePath), valid(true) {
    //-- Intenta abrir el archivo
    std::ifstream file(path);
    if (!file.is_open()) {
      //-- Marca el objeto como inválido pero permite su creación
      valid = false;
    }
  }

  //-- Método que asume que el objeto es válido
  void process() {
    //-- Uso del objeto sin garantizar estado válido
    if (!valid) {
      std::cout << "Estado inválido, pero el objeto existe." << std::endl;
    } else {
      std::cout << "Procesando archivo correctamente." << std::endl;
    }
  }
};

int main() {
  //-- Se crea el objeto aunque el archivo no exista
  FileLoader loader("no_existe.txt");
  //-- Se invoca un método sobre un objeto potencialmente inválido
  loader.process();
  return 0;
}
```

Codificación recomendada

Se presenta una clase que representa una conexión a un recurso crítico. La intención es garantizar que el objeto solo se construya si el recurso está disponible. En caso contrario, el constructor lanza una excepción para evitar que el programa continúe con un objeto en estado inválido, reforzando la integridad y la seguridad del sistema.

```
#include <iostream>
#include <fstream>
#include <string>
#include <stdexcept>
```

```
class FileLoader {
  //-- Ruta del archivo
  std::string path;

public:
  //-- Constructor que garantiza objeto válido o lanza excepción
  FileLoader(const std::string& filePath) : path(filePath) {
    //-- Intenta abrir el archivo
    std::ifstream file(path);
    if (!file.is_open()) {
      //-- Lanza excepción si no se puede crear un objeto válido
      throw std::runtime_error("No se pudo abrir el archivo: " + path);
    }
  }

  //-- Método que asume estado siempre válido
  void process() {
    //-- Operación segura porque el objeto solo existe si es válido
    std::cout << "Procesando archivo correctamente." << std::endl;
  }
};

int main() {
  try {
    //-- Se intenta crear el objeto
    FileLoader loader("no_existe.txt");
    loader.process();
  } catch (const std::exception& ex) {
    //-- Manejo controlado del error
    std::cout << "Error: " << ex.what() << std::endl;
  }
  return 0;
}
```

2.9.8.1 Descripción

Los constructores son responsables de establecer un objeto en un estado válido desde el momento de su creación. En el contexto de desarrollo seguro, es fundamental garantizar que ningún objeto entre en el sistema en un estado inconsistente, ya que ello podría desencadenar fallos lógicos, vulnerabilidades por uso indebido de memoria, comportamientos indefinidos o errores que comprometan la integridad general de la aplicación. Cuando un constructor no puede validar correctamente sus parámetros, no puede asignar recursos de forma segura o no puede establecer las invariantes internas del objeto, debe lanzar una excepción para evitar la construcción incompleta.

Permitir que se creen objetos inválidos puede derivar en efectos secundarios muy peligrosos. Algunos programadores intentan establecer un indicador interno de error o

usan métodos de inicialización posteriores, pero estos enfoques representan riesgos considerables, especialmente cuando otros componentes asumen que el objeto creado está correctamente inicializado. Este problema se manifiesta en estructuras complejas como manejadores de recursos, clases RAII, contenedores con dependencias encadenadas y capas de abstracción de bajo nivel. La garantía crítica es que ningún método debería operar sobre un objeto cuyo constructor ha fallado.

Un constructor seguro debe cumplir dos principios: establecer completamente todas sus invariantes o fallar de manera explícita. La forma estándar y recomendada de gestionar fallos en un constructor es lanzar una excepción. C++ garantiza que, si un constructor lanza una excepción, los campos ya inicializados serán liberados de forma segura y los destructores relevantes serán invocados automáticamente. Esta propiedad evita pérdidas de memoria, fugas de recursos y estados intermedios indefinidos. La semántica RAII se mantiene intacta.

Las invariantes de una clase constituyen un conjunto de condiciones que debe cumplir un objeto para considerarse válido. Un constructor seguro debe garantizar que estas invariantes se establezcan correctamente antes de que el objeto sea utilizado. Entre estas invariantes pueden incluirse: recursos asignados de forma consistente, valores de parámetros validados, configuraciones internas correctas y punteros inicializados. Si alguna de estas invariantes no puede garantizarse, el mecanismo de excepción actúa como una barrera que evita que un objeto defectuoso continúe existiendo.

El empleo de excepciones en constructores también mejora la trazabilidad de errores y permite una auditoría más clara del flujo del programa. Al lanzar una excepción, se proporciona información específica sobre la naturaleza del fallo, lo que permite a los componentes superiores tomar decisiones informadas, corregir parámetros, reintentar operaciones o registrar eventos críticos. Esta capacidad proporciona un fuerte soporte para sistemas que requieren altos niveles de resiliencia y observabilidad. En entornos críticos, esta práctica asegura que ningún objeto en estado inestable entre en la cadena de ejecución.

Es importante considerar que en algunos casos los desarrolladores intentan evitar excepciones debido a políticas internas, restricciones de sistemas embebidos o requisitos de rendimiento. Sin embargo, incluso en esos entornos deben incluirse mecanismos equivalentes que bloqueen la creación del objeto si falla la inicialización, como constructores de fábrica que devuelven tipos opcionales. No obstante, en un desarrollo seguro estándar en C++, la recomendación explícita sigue siendo lanzar una excepción en el constructor cuando no es posible establecer un estado válido.

Otro aspecto relevante es el manejo seguro de los recursos adquiridos parcialmente. Si un constructor necesita asignar varias estructuras internas y una parte falla durante el proceso, lanzar una excepción garantiza que los destructores de los recursos previamente adquiridos se ejecuten automáticamente, evitando fugas. Este comportamiento es parte esencial del modelo RAII, lo cual refuerza tanto la calidad del código como la solidez de la aplicación ante errores inesperados.

Riesgos asociados

- Creación de objetos en estado inconsistente o inválido.
- Fallos en tiempo de ejecución debido a acceso a miembros no inicializados.
- Dificultad para depurar errores debido a comportamientos inesperados en los objetos.
- Violación de invariantes de clase, lo que puede causar corrupción de datos.
- Propagación de errores silenciosos a otras partes del sistema.
- Posibles ataques de seguridad aprovechando objetos parcialmente construidos.
- Pérdida de recursos o fugas de memoria por inicialización incompleta.
- Fallo en el cumplimiento de garantías de excepción (exception safety).
- Incompatibilidades con contenedores y algoritmos estándar que asumen objetos válidos.
- Pérdida de confianza en la robustez y predictibilidad del código.

Buenas prácticas

- Validar explícitamente las precondiciones del constructor y lanzar una excepción estándar adecuada cuando no se cumplan.

```
//-- Constructor que no valida parámetros y permite estado inválido
class FileReader {
  std::string path;
public:
  FileReader(const std::string& p) : path(p) {}
};
```

```
//-- Constructor que valida y lanza excepción si el recurso no es válido
class FileReader {
  std::string path;
public:
  FileReader(const std::string& p) : path(p) {
    if (path.empty()) {
      throw std::invalid_argument("Ruta vacia");
    }
  }
};
```

- Verificar la adquisición de recursos críticos dentro del constructor y lanzar una excepción si falla.

```
//-- Constructor que ignora fallo al abrir archivo
```

```
class Config {
  std::ifstream file;
public:
  Config(const std::string& name) {
    file.open(name);
  }
};
```

```
//-- Constructor que valida apertura y lanza excepción si falla
class Config {
  std::ifstream file;
public:
  Config(const std::string& name) {
    file.open(name);
    if (!file.is_open()) {
      throw std::runtime_error("No se pudo abrir el archivo");
    }
  }
};
```

- Evitar dejar objetos parcialmente inicializados utilizando listas de inicialización y validaciones inmediatas.

```
//-- Inicialización diferida que permite uso previo a validación
class User {
  int id;
public:
  User(int value) {
    id = value;
  }
};
```

```
//-- Validación previa en lista de inicialización con comprobación posterior
class User {
  int id;
public:
  User(int value) : id(value) {
    if (id <= 0) {
      throw std::out_of_range("Identificador invalido");
    }
  }
};
```

- No capturar y suprimir excepciones dentro del constructor si el objeto no puede quedar en estado válido.

```
//-- Constructor que captura y oculta error dejando estado inconsistente
```

```
class Connection {
  Socket sock;
public:
  Connection(const std::string& host) {
    try {
      sock.connect(host);
    } catch (...) {
    }
  }
};
```

```
//-- Constructor que propaga excepción para impedir objeto inválido
class Connection {
  Socket sock;
public:
  Connection(const std::string& host) {
    sock.connect(host);
  }
};
```

2.9.8.2 Estándares metodológicos

- **C++ Core-C.42**: If a constructor cannot construct a valid object, throw an exception.
- **C++ Core-E.5**: Let a constructor establish an invariant, and throw if it cannot.

2.9.9 [cpp.cls.9] - Nunca lanzar excepciones desde un destructor

ID	**cpp.cls.9**	**Uso**	**Obligatorio**
CWSS Score	**74**	**Impacto técnico**	**Alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Los destructores no deben lanzar excepciones, debido a que la terminación de un objeto ocurre frecuentemente durante el manejo de errores y un lanzamiento adicional puede conducir a la terminación del programa, pérdida de datos o inconsistencias internas. Este control se centra en gestionar los fallos dentro del destructor sin propagación, registrando errores o tomando medidas seguras, garantizando así un comportamiento predecible y estable durante la liberación de recursos.

Codificación insegura

Gestión de recursos mediante una clase que libera un recurso en su destructor. Se busca mostrar qué ocurre cuando durante la liberación se produce un error y se decide lanzar una excepción, analizando el impacto que esto tiene en la estabilidad del programa.

```
#include <iostream>
#include <stdexcept>

class FileHandler {
public:
  //-- Constructor que simula la apertura de un recurso
  FileHandler() {
    std::cout << "Recurso adquirido" << std::endl;
  }

  //-- Destructor que lanza una excepción si ocurre un error
  ~FileHandler() {
    std::cout << "Liberando recurso" << std::endl;
    throw std::runtime_error("Error al liberar el recurso");
  }
};

int main() {
  //-- Creación del objeto en el ámbito automático
  FileHandler handler;

  //-- Al salir del ámbito se invoca el destructor y se produce la excepción
  return 0;
}
```

Codificación recomendada

Gestión segura de recursos donde el destructor garantiza que no se propaguen excepciones hacia el exterior. Se demuestra cómo manejar internamente los errores durante la liberación del recurso para evitar la terminación inesperada del programa.

```
#include <iostream>
#include <stdexcept>

class FileHandler {
public:
  //-- Constructor que simula la adquisición de un recurso
  FileHandler() {
    std::cout << "Recurso adquirido" << std::endl;
  }

  //-- Destructor que captura internamente cualquier error y no lanza excepciones
  ~FileHandler() noexcept {
    try {
      std::cout << "Liberando recurso" << std::endl;
      throw std::runtime_error("Error al liberar el recurso");
    } catch (const std::exception& e) {
      //-- comentario
```

```
      //-- Registro del error sin propagar la excepción
      std::cerr << "Error controlado en destructor: " << e.what() << std::endl;
    }
  }
};

int main() {
  //-- Creación del objeto en el ámbito automático
  FileHandler handler;

  //-- El programa finaliza normalmente sin terminación inesperada
  return 0;
}
```

2.9.9.1 Descripción

Los destructores representan el mecanismo fundamental para la liberación determinista de recursos, especialmente cuando se aplican patrones como RAII. Sin embargo, su ejecución en contextos donde ya existe una excepción activa convierte a estos métodos en puntos extremadamente sensibles desde la perspectiva de la estabilidad, robustez y seguridad del software. **Lanzar una excepción desde un destructor puede provocar la terminación inmediata del programa** si ocurre durante el proceso de stack unwinding, lo que genera riesgos significativos como pérdida de datos, corrupción de estados internos, interrupción inesperada del flujo de ejecución y fallos de seguridad derivados de dejar el sistema en un estado inconsistente. Debido a esto, se considera una regla crítica evitar que los destructores propaguen excepciones y, en cambio, encapsular cualquier posible error dentro del propio destructor.

La razón subyacente de esta directriz se basa en el diseño del modelo de excepciones de C++. Cuando una excepción se lanza y no se captura inmediatamente, el runtime comienza un proceso de liberación de recursos llamado stack unwinding, en el cual se invocan los destructores de los objetos ya construidos. Si durante ese proceso un destructor lanza una nueva excepción, el comportamiento resulta catastrófico, ya que el lenguaje no posee un mecanismo estándar para manejar múltiples excepciones simultáneas en este flujo. En consecuencia, la ejecución no puede continuar de forma ordenada, lo cual desemboca en la llamada a `std::terminate`, eliminando cualquier capacidad de recuperación. Desde un punto de vista de seguridad, esto significa que el sistema pierde control en un momento crítico del flujo, dificultando la protección frente a estados vulnerables o no verificados.

Para cumplir adecuadamente este requisito, es necesario adoptar una estrategia donde el destructor nunca deje escapar excepciones. Esto implica que cualquier operación susceptible de lanzarlas debe gestionarse localmente mediante mecanismos como bloques `try/catch` internos. Cuando se detecta una condición de error, el destructor debe optar por alternativas seguras que no interfieran con el proceso de liberación de

recursos. Entre estas alternativas se encuentran: el registro del fallo, el marcado de un estado interno como inválido o la ejecución de acciones de recuperación menores que no introduzcan riesgo. El objetivo principal es preservar la integridad del sistema, incluso en escenarios donde ya existe un fallo en curso.

Otra consideración relevante es el uso de la especificación `noexcept` en destructores. Declarar explícitamente un destructor como `noexcept` comunica la intención de que no deben escapar excepciones y permite al compilador optimizar la lógica de manejo de errores. Aunque los destructores son `noexcept` por defecto en C++11 y posteriores, la declaración explícita mejora la claridad y evita confusiones en contextos con plantillas, herencia o estructuras complejas. De igual forma, cuando un destructor necesita realizar operaciones que pueden fallar, es preferible delegarlas a métodos dedicados, donde el usuario de la clase pueda manejar o consultar el error de manera explícita y controlada.

Las clases que gestionan recursos sensibles, como manejadores de archivos, sockets, conexiones de red, transacciones de base de datos, bloqueos de sincronización, memoria dinámica o buffers compartidos, deben ser particularmente cuidadosas con esta regla. Un lanzamiento inesperado dentro de un destructor podría dejar recursos sin liberar, provocar fugas, romper invariantes internos del sistema o exponer datos en estados transitorios inseguros. Este tipo de fallos es especialmente crítico en sistemas de alta disponibilidad, sistemas embebidos, software financiero, entornos que requieren cumplimiento estricto de estándares de seguridad o aplicaciones que operan con permisos elevados.

También es necesario mencionar que, en ciertos casos, un destructor puede enfrentarse a errores inevitables, como fallos al cerrar descriptores o liberar recursos de terceros. La respuesta adecuada consiste en registrar el error de forma segura y permitir que el programa continúe con el proceso de limpieza sin riesgo de sufrir una terminación abrupta. Para ello, se recomienda usar un sistema seguro de logging que no dependa de recursos inestables o en proceso de liberación, evitando así efectos secundarios inesperados.

Riesgos asociados

- Pérdida de recursos por terminación abrupta del programa (resource leaks).
- Invocación de std::terminate y detención inesperada de la aplicación.
- Corrupción de memoria o estado inconsistente del programa.
- Dificultad para depurar y diagnosticar problemas.
- Imposibilidad de realizar una recuperación segura de errores.
- Interferencia en el manejo adecuado de excepciones anidadas.
- Violación de las garantías de excepción (exception safety guarantees).
- Riesgo de denegación de servicio (DoS) en sistemas críticos.

Buenas prácticas

- Capturar internamente cualquier excepción dentro del destructor para evitar la propagación durante el stack unwinding.

```
//-- Destructor que propaga excepción
class Recurso {
  ~Recurso() {
    liberar();
  }
  void liberar() {
    throw std::runtime_error("Error al liberar");
  }
};
```

```
//-- Destructor que captura y gestiona la excepción internamente
class RecursoSeguro {
  ~RecursoSeguro() {
    try {
      liberar();
    } catch (const std::exception& e) {
      registrarError(e.what());
    }
  }
  void liberar() {
    throw std::runtime_error("Error al liberar");
  }
  void registrarError(const std::string& msg) {
    /* registro seguro */
  }
};
```

- Declarar el destructor como `noexcept` para garantizar explícitamente que no lanzará excepciones.

```
//-- Destructor sin especificación noexcept
class Conexion {
  ~Conexion() {
    cerrar();
  }
  void cerrar() {
    throw std::runtime_error("Fallo al cerrar");
  }
};
```

```
//-- Destructor declarado noexcept y manejo interno de errores
class ConexionSegura {
  ~ConexionSegura() noexcept {
```

```
    try {
      cerrar();
    } catch (...) {
      /* manejo alternativo */
    }
  }
  void cerrar() {
    /* operación que puede fallar */
  }
};
```

- Proporcionar un método explícito de liberación que permita reportar errores antes de que el destructor sea invocado.

```
//-- Liberación implícita en destructor con posible excepción
class Archivo {
  ~Archivo() {
    if (!cerrar()) {
      throw std::runtime_error("No se pudo cerrar");
    }
  }
  bool cerrar();
};
```

```
//-- Método explícito de cierre con verificación previa
class ArchivoSeguro {
  ~ArchivoSeguro() noexcept {
    if (abierto) {
      cerrar();
    }
  }
  bool cerrar() {
    if (!abierto) return true;
    abierto = false;
    return true;
  }
  bool abierto{true};
};
```

2.9.9.2 Estándares metodológicos

- **C++ Core-C.36**: A destructor must not fail.
- **C++ Core-C.37**: Make destructors noexcept.

2.9.10 [cpp.cls.10] - Liberar todos los recursos adquiridos por la clase desde el destructor

ID	cpp.cls.10	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Disponibilidad	S-SDLC	Build

Toda clase debería gestionar de forma adecuada la liberación de recursos en su destructor para prevenir fugas de memoria, desbordamientos de recursos del sistema y comportamientos indefinidos. Implementar una política clara de liberación asegura que los recursos adquiridos por la clase, como memoria dinámica, descriptores de archivo, manejadores del sistema operativo o buffers temporales, sean correctamente devueltos. Esto contribuye a mantener la estabilidad, seguridad y disponibilidad del sistema, evitando fallos que puedan ser explotados o provocar interrupciones inesperadas.

Codificación insegura

Gestión de recursos adquiridos por una clase que reserva memoria dinámica y abre un archivo durante su construcción. Se busca mostrar la importancia de liberar explícitamente todos los recursos en el destructor para evitar fugas de memoria y de descriptores de archivo cuando el objeto sale de su ámbito.

```
#include <iostream>
#include <cstdio>

class FileManager {
public:
  FileManager(const char* path) {
    //-- Reserva memoria dinámica para un buffer
    buffer = new int[10];

    //-- Abre un archivo para escritura
    file = std::fopen(path, "w");
  }

  ~FileManager() {
    //-- Destructor vacío: no libera los recursos adquiridos
  }

  void write() {
    if (file) {
      //-- Escribe un mensaje en el archivo
      std::fputs("Hola mundo\n", file);
    }
  }
```

```
private:
  int* buffer;
  FILE* file;
};

int main() {
  //-- Crea un objeto que adquiere recursos
  FileManager manager("salida.txt");
  manager.write();

  //-- Al salir de main se invoca el destructor
  return 0;
}
```

Codificación recomendada

Gestión correcta de recursos adquiridos por una clase que reserva memoria dinámica y abre un archivo. Se demuestra cómo el destructor libera explícitamente todos los recursos, asegurando que no queden fugas cuando el objeto finaliza su ciclo de vida.

```
#include <iostream>
#include <cstdio>

class FileManager {
public:
  FileManager(const char* path) {
    //-- Reserva memoria dinámica para un buffer
    buffer = new int[10];

    //-- Abre un archivo para escritura
    file = std::fopen(path, "w");
  }

  ~FileManager() {
    //-- Libera la memoria dinámica si fue asignada
    delete[] buffer;

    //-- Cierra el archivo si fue abierto correctamente
    if (file) {
      std::fclose(file);
    }
  }

  void write() {
    if (file) {
      //-- Escribe un mensaje en el archivo
      std::fputs("Hola mundo\n", file);
    }
```

```
  }

private:
  int* buffer;
  FILE* file;
};

int main() {
  //-- Crea un objeto que adquiere recursos
  FileManager manager("salida.txt");
  manager.write();

  //-- Al salir de main el destructor libera todos los recursos
  return 0;
}
```

2.9.10.1 Descripción

La gestión de memoria y de otros recursos no es automática, de modo que cada clase que adquiera recursos debe liberar explícitamente dichos recursos en su destructor. Este requisito se fundamenta en el principio RAII, que establece que la vida de un recurso debe estar ligada a la vida de un objeto. Cuando se incumple esta norma, pueden aparecer fugas de memoria, pérdida de descriptores, comportamiento indefinido o incluso condiciones que afecten a la disponibilidad del sistema. Por ello, es fundamental que cada clase que gestione recursos propios implemente un destructor fiable, seguro y coherente con el diseño de la clase.

Existen múltiples recursos que una clase puede adquirir: memoria mediante `new` o `new[]`, descriptores de archivo abiertos mediante APIs del sistema operativo, sockets, mutex o bloqueos, áreas asignadas mediante bibliotecas de terceros o estructuras complejas que deben liberarse mediante funciones personalizadas. Cada uno de estos recursos debe ser cuidadosamente liberado en el destructor para garantizar que no queden elementos residuales que comprometan la eficiencia o seguridad del programa. Además, debe verificarse que la liberación se produzca de manera segura incluso en presencia de excepciones.

Uno de los aspectos clave de este requisito es que el destructor debe ser **exento de excepciones**. Un destructor que lanza excepciones compromete la estabilidad del sistema, especialmente si se produce durante el proceso de stack unwinding. La liberación debe realizarse dentro de bloques seguros, evitando lanzar errores o depender de funciones externas poco fiables. Una implementación adecuada debe rodear la liberación con controles que garanticen que no se generarán situaciones anómalas durante el apagado del objeto.

Cuando se utilizan punteros gestionados manualmente, la liberación en el destructor debe realizarse usando la operación correspondiente a la asignación original. Si se

usó `new`, debe usarse `delete`; si se usó `new[]`, debe emplearse `delete[]`. Es crucial no mezclar estas operaciones, ya que esto genera comportamientos indefinidos que pueden derivar en vulnerabilidades explotables. Además, se debe evitar liberar un puntero dos veces y no intentar liberar punteros no inicializados.

Los recursos no pertenecientes al lenguaje, como descriptores de archivo o sockets, deben cerrarse explícitamente en el destructor. Estos descriptores suelen venir referenciados en handles para permitir acceder y gestionar los recursos del sistema, sin exponer directamente su implementación interna. No cerrar adecuadamente los recursos puede provocar fuga de handles (handle leaks), agotamiento de recursos globales y denegación de servicio.

El destructor debe llamar a funciones como `close()` o a los métodos adecuados de liberación según la API correspondiente. Además, debe asegurarse de que no se llame dos veces al cierre para un mismo recurso, para lo cual se recomienda mantener estados internos claros que controlen si el recurso está activo o no.

El uso de punteros inteligentes como `std::unique_ptr`, `std::shared_ptr` o `std::weak_ptr` reduce significativamente la necesidad de liberar recursos manualmente en el destructor. Estos tipos aplican RAII automáticamente, asegurando que los recursos se liberen cuando el objeto deja de estar en uso. No obstante, incluso con punteros inteligentes, el destructor debe contemplar la liberación de cualquier otro recurso adquirido fuera del ámbito de estos mecanismos.

Debido a la naturaleza crítica de la gestión de recursos, es recomendable que el destructor no realice más tareas de las estrictamente necesarias. Debe evitarse que incluya lógica compleja o dependencias de alto nivel, las cuales podrían fallar o producir efectos colaterales inesperados durante la destrucción del objeto. La sencillez y determinismo del destructor son clave para prevenir vulnerabilidades asociadas a corrupción de memoria o estados inconsistentes en el programa.

La liberación adecuada de recursos tiene también un impacto en la seguridad del sistema. Liberar memoria evita escenarios donde restos de información sensible queden disponibles en áreas no limpiadas, mitigando filtraciones accidentales. Cerrar correctamente archivos y sockets reduce la superficie de ataque al impedir accesos residuales desde procesos huérfanos. Además, garantizar que los mutex se liberen en el destructor blinda el sistema contra bloqueos lógicos o deadlocks persistentes.

Riesgos asociados

- Fugas de memoria (memory leaks) al no liberar memoria dinámica asignada a lo largo del ciclo de vida del objeto.
- Fugas de recursos del sistema, como descriptores de archivos, sockets o manejadores de bases de datos, lo que puede agotar recursos del sistema operativo.

- Inhabilidad para reutilizar recursos, causando un uso ineficiente del sistema y posibles bloqueos (deadlocks).
- Incremento en el consumo de memoria y reducción en el rendimiento de la aplicación debido a recursos no liberados.
- Comportamiento indefinido si otros objetos dependen de la correcta liberación de recursos.
- Vulnerabilidades de seguridad, como Denegación de Servicio (DoS), al agotar recursos críticos.
- Errores difíciles de depurar, ya que las fugas de recursos pueden manifestarse sólo bajo ciertas condiciones o en ejecuciones prolongadas.
- Pérdida de datos si no se cierran y liberan adecuadamente recursos relacionados con almacenamiento persistente.

Buenas prácticas

- Implementar el principio RAII asegurando que el destructor libere explícitamente todos los recursos dinámicos adquiridos por la clase.

```
//-- Recurso dinámico adquirido pero no liberado en el destructor
class Archivo {
  FILE* f;
public:
  Archivo(const char* nombre) {
    f = fopen(nombre, "r");
  }
  ~Archivo() {
    // recurso no liberado
  }
};
```

```
//-- Liberación explícita del recurso en el destructor
class Archivo {
  FILE* f;
public:
  Archivo(const char* nombre) {
    f = fopen(nombre, "r");
  }
  ~Archivo() {
    if (f != nullptr) {
      fclose(f);
    }
  }
};
```

- Utilizar punteros inteligentes para delegar la liberación automática de memoria y evitar fugas en el destructor.

```
//-- Uso de puntero crudo sin liberar memoria dinámica
class Buffer {
  int* data;
public:
  Buffer(size_t n) {
    data = new int[n];
  }
  ~Buffer() {
    // memoria no liberada
  }
};
```

```
//-- Uso de std::unique_ptr para liberación automática
class Buffer {
  std::unique_ptr data;
public:
  Buffer(size_t n) : data(std::make_unique(n)) {
  }
  ~Buffer() {
    // liberación automática gestionada por unique_ptr
  }
};
```

- Garantizar que todos los recursos múltiples adquiridos sean liberados incluso en presencia de excepciones.

```
//-- Múltiples recursos sin liberación completa en el destructor
class Conexion {
  int* datos;
  FILE* log;
public:
  Conexion() {
    datos = new int[10];
    log = fopen("log.txt", "w");
  }
  ~Conexion() {
    delete[] datos;
    // archivo no cerrado
  }
};
```

```
//-- Liberación completa y validación de cada recurso adquirido
class Conexion {
  int* datos;
```

```
  FILE* log;
public:
  Conexion() {
    datos = new int[10];
    log = fopen("log.txt", "w");
  }
  ~Conexion() {
    if (datos != nullptr) {
      delete[] datos;
    }
    if (log != nullptr) {
      fclose(log);
    }
  }
};
```

2.9.10.2 Estándares metodológicos

- **C++ Core-C.31**: All resources acquired by a class must be released by the class's destructor.
- **CERT/CPP-MEM51**: Properly deallocate dynamically allocated resources.

2.9.11 [cpp.cls.11] - Evitar llamar funciones virtuales desde constructores o destructores

ID	cpp.cls.11	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se debe evitar la invocación de funciones **virtuales** desde constructores o destructores, ya que durante estas etapas el objeto no está completamente construido o ya está en proceso de destrucción. Llamar a funciones virtuales en estos momentos puede generar comportamientos inesperados, romper invariantes de seguridad, producir accesos a memoria no válida, o permitir que lógica dependiente del polimorfismo se ejecute de forma incompleta o insegura. Este requisito reduce riesgos comunes relacionados con errores de integridad y corrupción de memoria en jerarquías de clases complejas.

Codificación insegura

Demostración de una jerarquía de clases donde se define un método virtual que es invocado desde el constructor de la clase base. Se busca mostrar el comportamiento real de la resolución dinámica durante la construcción de objetos y cómo puede generar resultados inesperados al no ejecutarse la versión redefinida en la clase derivada.

```
#include <iostream>
```

```
class Base {
public:
  //-- Constructor que llama a un método virtual
  Base() {
    inicializar();
  }

  virtual void inicializar() {
    std::cout << "Inicializacion desde Base" << std::endl;
  }

  virtual ~Base() = default;
};

class Derivada : public Base {
public:
  Derivada() {
    std::cout << "Constructor de Derivada" << std::endl;
  }

  void inicializar() override {
    std::cout << "Inicializacion desde Derivada" << std::endl;
  }
};

int main() {
  //-- Se crea un objeto derivado
  Derivada obj;
  return 0;
}
```

Codificación recomendada

Implementación de la misma jerarquía evitando la llamada a métodos virtuales dentro del constructor. La inicialización dependiente del polimorfismo se realiza mediante un método público que se invoca después de que el objeto esté completamente construido, asegurando que la resolución dinámica funcione como se espera.

```
#include <iostream>

class Base {
public:
  Base() {
    //-- Constructor sin llamadas a metodos virtuales
  }

  virtual void inicializar() {
```

```
    std::cout << "Inicializacion desde Base" << std::endl;
  }

  virtual ~Base() = default;
};

class Derivada : public Base {
public:
  Derivada() {
    std::cout << "Constructor de Derivada" << std::endl;
  }

  void inicializar() override {
    std::cout << "Inicializacion desde Derivada" << std::endl;
  }
};

int main() {
  //-- Se crea el objeto
  Derivada obj;

  //-- Se invoca el metodo virtual despues de la construccion completa
  obj.inicializar();
  return 0;
}
```

2.9.11.1 Descripción

El despacho de funciones virtuales (**virtual dispatch**) es el mecanismo utilizado por C++ para resolver las llamadas a funciones declaradas como virtual. Cuando la llamada se realiza a través de un puntero o una referencia a una clase base, el lenguaje debe decidir qué implementación concreta ejecutar. Si el objeto pertenece a una clase derivada que ha sobrescrito la función, se invoca dicha implementación; en caso contrario, se ejecuta la versión definida en la clase base. Este proceso de selección en tiempo de ejecución se conoce como despacho dinámico (**dynamic dispatch**) y constituye el fundamento del polimorfismo en C++.

El despacho dinámico solo funciona plenamente cuando un objeto ha completado su proceso de construcción. Mientras se ejecuta el constructor de una clase base, el objeto aún no constituye una instancia completa de la clase más derivada, ya que los subobjetos derivados todavía no han sido construidos. En consecuencia, cualquier llamada a una función virtual realizada durante esa fase se resuelve como si el objeto perteneciera únicamente a la clase base. En las implementaciones habituales de C++, este comportamiento se materializa haciendo que la tabla virtual (vtable) activa corresponda a la clase cuyo constructor se está ejecutando. Solo cuando finaliza la construcción de toda la jerarquía de herencia el objeto adquiere su tipo dinámico

definitivo y el despacho dinámico comienza a seleccionar las implementaciones sobrescritas de la clase más derivada.

Como consecuencia, **las llamadas a funciones virtuales realizadas desde constructores no presentan un comportamiento polimórfico**. Asumir que se ejecutará la implementación redefinida en una clase derivada puede provocar errores lógicos, ya que dicha parte del objeto todavía no existe y sus invariantes aún no han sido establecidas. Esta situación resulta especialmente crítica cuando las funciones virtuales encapsulan operaciones de seguridad, como validaciones, inicialización de recursos, comprobaciones de integridad o establecimiento de invariantes.

Un comportamiento análogo se produce durante la destrucción del objeto. Cuando comienza la ejecución del destructor de una clase base, los subobjetos de las clases derivadas ya han sido destruidos. Por este motivo, l**as llamadas a funciones virtuales realizadas desde un destructor tampoco se despachan hacia las implementaciones de las clases derivadas**, sino hacia las correspondientes a la clase cuyo destructor se está ejecutando. Invocar funciones virtuales en esta fase puede dar lugar a operaciones incompletas, acceso a recursos ya liberados, incumplimiento de invariantes o resultados impredecibles.

El uso indebido del despacho de funciones virtuales en constructores y destructores puede comprometer la integridad del estado interno del objeto y originar vulnerabilidades difíciles de detectar. Estos problemas aparecen con frecuencia en diseños orientados a objetos que dependen intensamente del polimorfismo para implementar validaciones, inicializaciones o mecanismos de seguridad. Cuando la secuencia de construcción o destrucción no respeta las garantías del lenguaje, el objeto puede quedar temporalmente en estados inconsistentes, aumentando la probabilidad de errores lógicos, corrupción del estado interno o comportamientos potencialmente explotables.

Riesgos asociados

- Comportamiento inesperado en tiempo de ejecución debido a que el tipo dinámico del objeto aún no está completamente construido o ya está parcialmente destruido.
- Invocación de la implementación base en vez de la sobrescrita, lo que puede provocar omisión de lógica específica de la subclase.
- Posible acceso a recursos no inicializados o ya liberados, ocasionando fallos o corrupción de memoria.
- Violación de invariantes de clase, lo que puede dejar el objeto en un estado inconsistente.
- Problemas de seguridad como uso de punteros colgantes o ejecución de código no deseado si la función virtual accede a miembros no válidos.

- Dificultad en el mantenimiento y depuración debido a efectos secundarios poco evidentes.

Buenas prácticas

- Evitar invocar métodos virtuales dentro del constructor; delegar la inicialización dependiente de polimorfismo a un método explícito posterior.

```
class Base {
public:
  Base() {
    //-- Invoca método virtual durante la construcción
    init();
  }
  virtual void init() { }
};

class Derivada : public Base {
public:
  void init() override {
    // implementación específica
  }
};
```

```
class BaseSeguro {
public:
  BaseSeguro() { }
  void initialize() {
    //-- Inicialización controlada fuera del constructor
    init();
  }
  virtual void init() { }
};

class DerivadaSegura : public BaseSeguro {
public:
  void init() override {
    // implementación específica
  }
};

DerivadaSegura obj;
obj.initialize();
```

- No llamar a funciones virtuales desde destructores; utilizar métodos no virtuales o mecanismos explícitos de liberación.

```
class Base {
```

```
public:
  virtual ~Base() {
    //-- Llamada virtual durante destrucción
    cleanup();
  }
  virtual void cleanup() { }
};

class Derivada : public Base {
public:
  void cleanup() override {
    // liberar recursos específicos
  }
};
```

```
class BaseSeguro {
public:
  virtual ~BaseSeguro() {
    //-- Destructor sin dependencias polimórficas
    cleanupImpl();
  }
protected:
  void cleanupImpl() { }
};
```

- Diseñar constructores que sólo inicialicen datos propios y dejar el comportamiento polimórfico para factorías o métodos estáticos.

```
class Base {
public:
  Base() {
    //-- Lógica dependiente de método virtual
    configure();
  }
  virtual void configure() = 0;
};

class Derivada : public Base {
public:
  void configure() override {
    // configuración específica
  }
};
```

```
class BaseSegura {
public:
  BaseSegura() { }
```

```
  virtual void configure() = 0;
  static void createAndConfigure(BaseSegura& obj) {
    //-- Configuración posterior a la construcción completa
    obj.configure();
  }
};
```

2.9.11.2 Estándares metodológicos

- **C++ Core-C.82**: don't call virtual functions in constructors and destructors.
- **CERT/CPP-OOP50**: Do not invoke virtual functions from constructors or destructors.

2.10 Funciones

2.10.1 [cpp.func.1] - Declarar las precondiciones de una función usando mecanismos formales

ID	cpp.func.1	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se deben declarar de manera visible y verificable las condiciones que tienen que cumplir las funciones antes de su ejecución, reduciendo fallos lógicos y evitando accesos indebidos a memoria. La definición clara de estas reglas permite detectar errores tempranos, fortalece la integridad del estado interno y facilita la auditoria del código. Al emplear validaciones explicitas, contratos, `asserts` o comprobaciones programáticas, se garantiza que las funciones operan bajo supuestos correctos, evitando escenarios inseguros derivados de parámetros incorrectos o estados inconsistentes.

Codificación insegura

Se muestra una función que realiza una división entera asumiendo que el divisor siempre es distinto de cero. La intención es observar qué ocurre cuando no se declaran ni validan formalmente las precondiciones de la función y cómo esto puede provocar fallos en tiempo de ejecución o comportamiento indefinido.

```
#include <iostream>

//-- Función que asume que el divisor es distinto de cero
int dividir(int a, int b) {
  //-- No se valida la precondición b != 0
  return a / b;
}
```

```
int main() {
  int resultado1 = dividir(10, 2);
  std::cout << "Resultado 1: " << resultado1 << std::endl;

  //-- Caso que viola la precondición (división por cero)
  int resultado2 = dividir(10, 0);
  std::cout << "Resultado 2: " << resultado2 << std::endl;
  return 0;
}
```

Codificación recomendada

Se presenta la misma operación de división, pero declarando explícitamente la precondición mediante un mecanismo formal del lenguaje (`assert`). El objetivo es mostrar cómo documentar y verificar en tiempo de ejecución las condiciones que deben cumplirse antes de ejecutar la lógica de la función, reduciendo el riesgo de comportamiento indefinido.

```
#include <iostream>
#include <cassert>

//-- Función que exige como precondición que el divisor sea distinto de cero
int dividir(int a, int b) {
  //-- Verificación formal de la precondición
  assert(b != 0);
  return a / b;
}

int main() {
  int resultado1 = dividir(10, 2);
  std::cout << "Resultado 1: " << resultado1 << std::endl;

  //-- Caso que viola la precondición y será detectado por assert
  int resultado2 = dividir(10, 0);
  std::cout << "Resultado 2: " << resultado2 << std::endl;
  return 0;
}
```

2.10.1.1 Descripción

Declarar de forma explícita las precondiciones en funciones y métodos constituye una medida fundamental para proteger la **integridad** del sistema y minimizar comportamientos indefinidos. En un lenguaje que ofrece acceso directo a memoria y donde el compilador no siempre puede inferir los supuestos del desarrollador, definir claramente los requisitos necesarios antes de invocar una función es un mecanismo esencial de codificación segura. Una precondición establece que ciertos parámetros

deben cumplir rangos, que los estados de los objetos deben ser válidos, que los punteros deben ser no nulos o que un contrato lógico debe mantenerse. Su ausencia puede provocar corrupción de memoria, fallos en ejecución o vulnerabilidades criticas derivadas de estados inconsistentes.

Las precondiciones pueden declararse mediante documentación clara, mecanismos tradicionales como `assert`, validaciones manuales en tiempo de ejecución o mediante herramientas modernas del estándar como `requires`, `concepts` o `static_assert`. Aunque los contratos formales propuestos para C++ aún no se encuentran completamente estandarizados, su filosofía orientada a la definición explícita de requisitos debe incorporarse usando los medios disponibles. La adopción consistente de estas técnicas permite expresar supuestos de funcionamiento, habilitar verificaciones estáticas y reducir significativamente la superficie de error.

En sistemas complejos, las precondiciones sirven como un lenguaje universal para comunicar expectativas entre módulos. Cuando una función indica exactamente lo que necesita para operar, se elimina la ambigüedad que puede llevar a un abuso accidental de la API. Ejemplos comunes incluyen: índices que deben pertenecer a un rango valido, objetos que deben estar inicializados, contenedores que deben disponer de datos previos o estados que requieren una secuencia de invocación especifica. Esta claridad reduce los riesgos de corrupción de memoria y fallos de integridad.

La definición explícita de precondiciones mejora además la capacidad de las herramientas de análisis estático para detectar defectos. Cuando las reglas son visibles y expresadas mediante mecanismos formales, herramientas como clang-tidy, Coverity o SonarQube pueden razonar sobre la lógica de la función y advertir tempranamente sobre violaciones. Asimismo, los desarrolladores que integran el código pueden ver inmediatamente que condiciones deben cumplirse, sin depender de interpretaciones o supuestos implícitos.

También es recomendable emplear `static_assert` cuando las precondiciones pueden verificarse en compilación, especialmente para invariantes de tipos o restricciones que no dependen de valores en tiempo de ejecución. Esta solución evita errores de forma temprana y sirve como documentación embebida directamente en el código. En contextos de alta criticidad, las validaciones nunca deben eliminarse en compilaciones de producción cuando su ausencia pueda introducir vulnerabilidades. Un enfoque consistente de programación defensiva garantiza que los usos indebidos de la API sean detectados inmediatamente.

Riesgos asociados

- Ejecución de funciones con argumentos inválidos o mal formados que pueden causar comportamientos indefinidos.
- Mayor probabilidad de errores difíciles de detectar debido a la falta de validación de entrada.

- Vulnerabilidad a ataques como desbordamientos de buffer, acceso fuera de límites y corrupción de memoria.
- Dificultad para mantener y evolucionar el código, ya que los contratos no están explícitos.
- Reducción de la eficacia de pruebas automáticas, ya que no se conoce el rango permitido de valores de entrada.
- Posibilidad de violar invariantes internas de las funciones o estructuras de datos.
- Incremento de la posibilidad de errores de seguridad por suposiciones incorrectas sobre el estado de los datos.
- Dificultad para documentar y comunicar claramente los requisitos de uso de la función entre desarrolladores.

Buenas prácticas

- Usar `assert` para expresar precondiciones internas que deben cumplirse en tiempo de desarrollo.

```
//-- función sin validación explícita de precondición
int dividir(int a, int b) {
  return a / b;
}
```

```
//-- verificación explícita de que el divisor no sea cero
#include <cassert>
int dividir(int a, int b) {
  assert(b != 0);
  return a / b;
}
```

- Utilizar excepciones estándar para formalizar el incumplimiento de precondiciones en interfaces públicas.

```
//-- función que asume puntero válido sin comprobarlo
void procesar(const std::string* texto) {
  std::cout << texto->size();
}
```

```
//-- validación explícita del puntero como precondición
#include <stdexcept>
void procesar(const std::string* texto) {
  if (texto == nullptr) {
    throw std::invalid_argument("texto no puede ser nulo");
  }
```

```
  std::cout << texto->size();
}
```

- Expresar precondiciones mediante `[[expects]]` (cuando esté disponible) o mecanismos equivalentes de contratos.

```
//-- función sin contrato explícito
int raizCuadrada(int valor) {
  return std::sqrt(valor);
}
```

```
//-- contrato formal que define valor no negativo como precondición
int raizCuadrada(int valor)
  [[expects: valor >= 0]]
{
  return std::sqrt(valor);
}
```

- Definir precondiciones en tiempo de compilación usando `static_assert` en plantillas.

```
//-- plantilla sin restricción sobre el tipo
template<typename T>
T incrementar(T valor) {
  return valor + 1;
}
```

```
//-- restricción formal del tipo mediante verificación en compilación
#include <type_traits>
template<typename T>
T incrementar(T valor) {
  static_assert(std::is_arithmetic<T>::value, "T debe ser numerico");
  return valor + 1;
}
```

- Documentar precondiciones en interfaces públicas utilizando anotaciones o atributos estáticos compatibles con herramientas de análisis.

```
//-- función sin anotaciones formales
void copiar(char* destino, const char* origen, size_t tam);
```

```
//-- anotación SAL que formaliza condiciones sobre los parámetros
#include <sal.h>
void copiar(
  _Out_writes_bytes_(tam) char* destino,
```

```
  _In_reads_bytes_(tam) const char* origen,
  size_t tam);
```

2.10.1.2 Estándares metodológicos

- **C++ Core-I.5**: State preconditions (if any).

2.10.2 [cpp.func.2] - Declarar las postcondiciones de una función usando mecanismos formales

ID	cpp.func.2	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Establecer postcondiciones formales en funciones permite definir con claridad las garantías que deben mantenerse al finalizar su ejecución, reforzando la integridad del estado interno, la consistencia de estructuras de datos y la correcta gestión de memoria y recursos. Este control reduce riesgos asociados a corrupciones de datos, estados inconsistentes, efectos secundarios no documentados o resultados inválidos. Al proporcionar expectativas verificables, se facilita el análisis estático, la depuración, la revisión de código y la detección temprana de desviaciones críticas que podrían convertirse en vulnerabilidades explotables, apoyando un desarrollo seguro y robusto.

Codificación insegura

Se implementa una función que calcula el cuadrado de un número entero y se espera que el resultado siempre sea mayor o igual que cero. El objetivo es mostrar cómo la ausencia de un mecanismo formal para declarar y verificar la postcondición puede permitir errores lógicos sin ser detectados en tiempo de ejecución.

```
#include <iostream>

int cuadrado(int x) {
  //-- Se espera que el resultado sea mayor o igual que cero
  int resultado = x * x;

  //-- Error lógico intencional para valores negativos
  if (x < 0) {
    resultado = -resultado;
  }
  return resultado;
}

int main() {
  int valor = -5;
```

```
  int res = cuadrado(valor);
  std::cout << "Resultado: " << res << std::endl;
  return 0;
}
```

Codificación recomendada

Se implementa la misma función, pero declarando y verificando formalmente la postcondición mediante una aserción. El objetivo es garantizar en tiempo de ejecución que el resultado siempre cumpla la condición esperada, detectando cualquier desviación de forma temprana y controlada.

```
#include <iostream>
#include <cassert>

int cuadrado(int x) {
  int resultado = x * x;

  //-- Postcondición: el resultado debe ser mayor o igual que cero
  assert(resultado >= 0);
  return resultado;
}

int main() {
  int valor = -5;
  int res = cuadrado(valor);
  std::cout << "Resultado: " << res << std::endl;
  return 0;
}
```

2.10.2.1 Descripción

Declarar postcondiciones formales en funciones es fundamental para garantizar que, después de la ejecución de una operación, el estado del sistema permanezca íntegro y consistente. Una postcondición define explícitamente qué debe ser cierto cuando la función retorna, incluyendo el valor devuelto, los efectos en memoria, las modificaciones en estructuras de datos, la validez de referencias y punteros, y cualquier compromiso contractual respecto a recursos y comportamiento. Su correcta definición permite detectar desviaciones peligrosas, identificar silenciosamente comportamientos indefinidos y fortalecer el diseño seguro.

La ausencia de comprobaciones automáticas de estado, sumada al acceso directo a memoria y a la gestión manual de recursos, hace que una función pueda dejar el sistema en un estado corrupto sin indicios inmediatos. Por ello, establecer postcondiciones reduce riesgos de uso de memoria no inicializada, desbordamientos, inconsistencias en invariantes de clases o fugas y dobles liberaciones de recursos. Las postcondiciones se

convierten así en un mecanismo de autocontrol interno que refuerza la seguridad lógica del programa.

Una postcondición debe ser precisa, determinista, verificable y libre de ambigüedades. Debe describir el estado esperado de los datos manipulados, indicando qué estructuras deben permanecer válidas, qué efectos secundarios están permitidos y qué valores deben cumplir propiedades específicas. Esto contribuye a una comprensión clara del comportamiento de la función y habilita la verificación mediante herramientas de análisis estático o dinámico, además de facilitar revisiones manuales de manera más rigurosa.

Los programas suelen manipular vectores, listas, árboles u otras estructuras avanzadas en las que la más mínima inconsistencia puede desencadenar comportamientos inseguros. Una postcondición bien definida asegura que cada operación preserva invariantes estructurales, manteniendo todos los enlaces, tamaños, referencias y validaciones internas en un estado seguro.

Las postcondiciones son especialmente importantes en funciones que asignan, liberan o transfieren propiedad de memoria. Definirlas evita fugas, dobles liberaciones y referencias colgantes, problemas que pueden comprometer gravemente la seguridad. En C++, donde la responsabilidad del desarrollador sigue siendo significativa incluso usando RAII, una verificación formal refuerza la fiabilidad del código.

Riesgos asociados

- Comportamiento inesperado de la función debido a la falta de garantías explícitas sobre el resultado.
- Introducción de errores lógicos difíciles de detectar y corregir.
- Incremento en la probabilidad de vulnerabilidades de seguridad, tales como corrupción de memoria o ejecución de código malicioso.
- Dificultad para verificar, auditar o mantener el código por falta de documentación clara sobre las expectativas de salida.
- Dependencia implícita en supuestos no documentados, lo que puede llevar a errores durante la integración o uso de la función por terceros.
- Reducción de la capacidad para aplicar análisis estático y pruebas automatizadas de manera efectiva.
- Posibilidad de inconsistencias en el estado del sistema si la función no cumple las condiciones requeridas al finalizar.
- Mayor probabilidad de fallas en el funcionamiento cuando cambian los requisitos del sistema o se realizan refactorizaciones.

Buenas prácticas

- Utilizar contratos del lenguaje (cuando estén disponibles) para expresar postcondiciones explícitas.

```
//-- Postcondición implícita no verificada
int dividir(int a,int b){
  return a/b;
}
```

```
//-- Postcondición formal usando contrato ensures
int dividir(int a,int b) [[ensures r: r*b==a]]{
  return a/b;
}
```

- Complementar las postcondiciones con aserciones en tiempo de ejecución.

```
//-- Resultado retornado sin verificación posterior
int incrementar(int x){
  return x+1;
}
```

```
//-- Verificación explícita de la postcondición con assert
int incrementar(int x){
  int r=x+1;
  assert(r>x);
  return r;
}
```

- Usar bibliotecas de contratos como GSL para documentar y verificar postcondiciones.

```
//-- Postcondición documentada solo en comentario
int obtenerEdad(){
  return calcularEdad();
}
```

```
//-- Postcondición expresada formalmente con Ensures
int obtenerEdad(){
  int edad=calcularEdad();
  Ensures(edad>=0);
  return edad;
}
```

- Aplicar `static_assert` cuando la postcondición pueda verificarse en tiempo de compilación.

```
//-- Suposición implícita sobre tamaño de tipo
size_t tam(){
  return sizeof(int);
}
```

```
//-- Verificación formal en compilación de la postcondición esperada
size_t tam(){
  static_assert(sizeof(int)>=4,"int debe ser >=4 bytes");
  return sizeof(int);
}
```

- Encapsular resultados en tipos que garanticen la postcondición mediante invariantes.

```
//-- Retorno de valor que puede ser inválido
int crearPuerto(int p){
  return p;
}
```

```
//-- Uso de tipo que valida la postcondición en su construcción
class Puerto{
  int valor;
  public:
  explicit Puerto(int p):valor(p){
    assert(p>0 && p<=65535);
  }
};
```

2.10.2.2 Estándares metodológicos

- **C++ Core-I.7**: State postconditions.

2.10.3 [cpp.func.3] - Señalizar forma explícita cuando una función no pueda cumplir su tarea

ID	**cpp.func.3**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Este requisito establece que toda función debe comunicar explícitamente cuando no pueda completar su operación prevista, evitando comportamientos silenciosos, estados inconsistentes o fallos impredecibles. La señalización clara de errores mejora la trazabilidad, la robustez y la seguridad del software empleando mecanismos como excepciones, valores de retorno bien definidos, tipos opcionales o códigos de error

estructurados, reduciendo así vectores de ataque derivados de estados erróneos o accesos inválidos.

Codificación insegura

Función que realiza una división entera y puede fallar cuando el divisor es cero. Se muestra cómo la falta de señalización explícita del error provoca ambigüedad en el resultado y dificulta al código que invoca la función saber si la operación fue válida.

```
#include <iostream>

int dividir(int a, int b) {
  //-- No se valida correctamente el divisor
  if (b == 0) {
    //-- Se devuelve 0 para indicar error, pero 0 puede ser un resultado válido
    return 0;
  }
  return a / b;
}

int main() {
  int resultado = dividir(10, 0);

  //-- El llamador no puede distinguir entre error y resultado válido
  std::cout << "Resultado: " << resultado << std::endl;
  return 0;
}
```

Codificación recomendada

Función que realiza una división entera y utiliza un mecanismo explícito para indicar que no puede cumplir su tarea cuando el divisor es cero. Se emplea un tipo de retorno que permite diferenciar claramente entre un resultado válido y una condición de error.

```
#include <iostream>
#include <optional>

std::optional<int> dividir(int a, int b) {
  //-- Se valida explícitamente la precondición
  if (b == 0) {
    //-- Se devuelve std::nullopt para indicar que la operación no pudo realizarse
    return std::nullopt;
  }
  return a / b;
}

int main() {
  std::optional<int> resultado = dividir(10, 0);
```

```
  //-- El llamador verifica explícitamente si la operación fue exitosa
  if (!resultado.has_value()) {
    std::cerr << "Error: division por cero" << std::endl;
    return 1;
  }
  std::cout << "Resultado: " << resultado.value() << std::endl;
  return 0;
}
```

2.10.3.1 Descripción

Una función tiene que ser capaz de **señalizar de forma explícita** cualquier situación en la que no pueda completar la tarea para la que fue diseñada. Un fallo silencioso, un retorno ambiguo o la omisión de la comunicación del error pueden generar vulnerabilidades graves, especialmente cuando la función forma parte de un flujo crítico o interactúa con estructuras de datos sensibles. Se debe evitar crear funciones que fallen de forma implícita o que generen efectos laterales inesperados sin avisar. Esto incluye retornos por defecto no verificados, valores especiales carentes de significado semántico o el uso excesivo de `nullptr` para indicar errores. Cada mecanismo de señalización debe estar vinculado a un contrato claro entre quien implementa y quien consume la función.

Algunos de los mecanismos para señalizar errores son:

- **Excepciones:** útiles cuando el error representa una condición excepcional y la recuperación puede manejarse en un nivel superior.
- **Tipos de retorno opcionales:** como `std::optional` para indicar operaciones que pueden no tener un resultado válido.
- **Códigos de error estructurados:** como `std::error_code` y `std::expected`.
- **Retornos booleanos con verificación obligatoria:** siempre acompañados de estructuras que permitan obtener la causa del fallo.

En C++, donde la gestión de memoria y el control de recursos es crítica, no reportar adecuadamente una condición de error puede provocar desde fugas de memoria hasta vulnerabilidades como desbordamientos, doble liberación o corrupción de estructuras internas. Implementar una estrategia clara, consistente y verificable para comunicar el fallo es esencial para prevenir dichas condiciones.

Los mecanismos modernos de la biblioteca estándar, como `std::optional`, `std::expected` y `std::error_code`, proporcionan una base robusta para establecer protocolos de señalización más seguros. A su vez, las excepciones permiten aislar el manejo del error y separarlo del flujo normal cuando éste no pueda ser resuelto en el

contexto inmediato. Todo ello debe hacerse evitando la ambigüedad, la ocultación de fallos y los retornos silenciosos.

Riesgos asociados

- Propagación de errores silenciosos que pueden derivar en fallas impredecibles del sistema.
- Corrupción de datos debido a suposiciones incorrectas sobre el resultado de la función.
- Condiciones de carrera y vulnerabilidades de seguridad al operar sobre recursos en estado inconsistente.
- Difícil trazabilidad y diagnóstico de errores, lo que complica el mantenimiento y depuración.
- Incremento del riesgo de desbordamientos, accesos fuera de límites y uso de punteros nulos o no inicializados.
- Comportamiento indefinido de la aplicación.
- Exposición a ataques de Denegación de Servicio (DoS), al no manejar correctamente fallos en llamadas críticas.
- Violación de la lógica de negocio o reglas de validación al asumir éxito en operaciones fallidas.

Buenas prácticas

- Utilizar excepciones para señalizar errores cuando una función no puede completar su tarea.

```
//-- La función retorna un valor ambiguo cuando falla
int dividir(int a, int b) {
  if (b == 0) {
    return 0;
  }
  return a / b;
}
```

```
//-- La función lanza una excepción para indicar fallo explícito
int dividir(int a, int b) {
  if (b == 0) {
    throw std::invalid_argument("Division por cero");
  }
  return a / b;
}
```

- Usar tipos de retorno explícitos como `std::optional` cuando la ausencia de

resultado es válida.

```
//-- Se devuelve un valor centinela que puede confundirse con uno válido
int buscarUsuario(const std::string& nombre) {
  if (nombre.empty()) {
    return -1;
  }
  return 42;
}
```

```
//-- Se utiliza std::optional para indicar ausencia de resultado
std::optional buscarUsuario(const std::string& nombre) {
  if (nombre.empty()) {
    return std::nullopt;
  }
  return 42;
}
```

- Marcar funciones cuyo resultado no debe ignorarse con `[[nodiscard]]`.

```
//-- El llamador puede ignorar el resultado sin advertencia
bool validarToken(const std::string& token) {
  return token == "ABC123";
}
```

```
//-- Se obliga a considerar el resultado devuelto
[[nodiscard]] bool validarToken(const std::string& token) {
  return token == "ABC123";
}
```

- Evitar modificar parámetros de salida sin indicar claramente el estado de la operación.

```
//-- Se modifica el parámetro aunque la operación falle
void obtenerClave(bool autorizado, std::string& clave) {
  if (!autorizado) {
    clave = "";
    return;
  }
  clave = "SECRETA";
}
```

```
//-- Se devuelve un indicador explícito del estado de la operación
bool obtenerClave(bool autorizado, std::string& clave) {
  if (!autorizado) {
    return false;
```

```
  }
  clave = "SECRETA";
  return true;
}
```

2.10.3.2 Estándares metodológicos

- **C++ Core-E.2**: Throw an exception to signal that a function can't perform its assigned task.
- **C++ Core-I.10**: Use exceptions to signal a failure to perform a required task.

2.10.4 [cpp.func.4] - Asegurar el uso de todo valor devuelto por una función "no void"

ID	cpp.func.4	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Todo valor devuelto por una función que no sea de tipo `void` debe ser utilizado, validado o manejado explícitamente para evitar comportamientos indefinidos, errores silenciosos, omisiones lógicas o fallos de seguridad. Ignorar el valor devuelto puede producir estados inconsistentes, pérdida de comprobaciones de error, corrupción de datos o vulnerabilidades explotables. La aplicación rigurosa de este principio contribuye a reforzar la robustez del software, asegurando que las funciones sean evaluadas correctamente y que sus resultados influyan en la lógica prevista del programa.

Codificación insegura

Se muestra una función que devuelve un valor booleano para indicar éxito o fallo al guardar un archivo de configuración. El objetivo es ilustrar qué ocurre cuando el valor devuelto por una función no `void` se ignora, y cómo esto puede provocar que el flujo del programa continúe como si todo hubiera salido bien, ocultando errores relevantes.

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

bool saveConfig(const std::string& path) {
  //-- Intenta abrir el archivo para escritura
  std::ofstream file(path);

  //-- Simula un fallo si no se pudo abrir el archivo
  if (!file.is_open()) {
    return false;
  }
```

```
  //-- Escribe datos de configuración
  file << "config=true" << std::endl;
  file.close();
  return true;
}

int main() {
  //-- Llamada a la función sin verificar el valor devuelto
  saveConfig("/ruta/invalida/config.txt");

  //-- El programa asume que todo salió bien
  std::cout << "Configuracion guardada correctamente" << std::endl;
  return 0;
}
```

Codificación recomendada

Se utiliza la misma función que devuelve un valor booleano para indicar éxito o fallo al guardar un archivo. El objetivo es mostrar cómo validar explícitamente el valor devuelto y tomar decisiones en función del resultado, evitando estados inconsistentes y mejorando la robustez del programa.

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

bool saveConfig(const std::string& path) {
  //-- Intenta abrir el archivo para escritura
  std::ofstream file(path);

  //-- Verifica si el archivo se abrió correctamente
  if (!file.is_open()) {
    return false;
  }
  //-- Escribe datos de configuración
  file << "config=true" << std::endl;
  file.close();
  return true;
}

int main() {
  //-- Llamada a la función verificando el valor devuelto
  bool result = saveConfig("/ruta/invalida/config.txt");

  //-- Control explícito del flujo ante error
  if (!result) {
    std::cerr << "Error al guardar la configuracion" << std::endl;
    return 1;
```

```
    }
    //-- Solo se informa de éxito si realmente ocurrió
    std::cout << "Configuracion guardada correctamente" << std::endl;
    return 0;
}
```

2.10.4.1 Descripción

El lenguaje permite a los desarrolladores definir funciones que retornan información esencial, incluyendo códigos de error, estados internos, referencias, punteros, contenedores, estructuras y objetos complejos. Sin embargo, cuando estos valores no se utilizan, se descartan sin intención o no se validan correctamente, surgen riesgos significativos de integridad, comportamiento indefinido y fallos silenciosos. Este requisito impone la obligación de procesar explícitamente dichos valores, reforzando la solidez del flujo de control, la trazabilidad lógica del programa y la seguridad general de la aplicación.

El valor retornado por una función es un canal primario de comunicación entre componentes. Su uso adecuado es especialmente relevante debido a la amplia gama de semánticas asociadas al retorno: desde objetos devueltos por valor o referencia, hasta el uso de `std::optional`, `std::expected`, punteros inteligentes, códigos numéricos y estados lógicos. Ignorar este valor puede causar desde fugas de recursos hasta el bypass involuntario de mecanismos de verificación. Cada retorno posee un propósito funcional y, en muchos casos, un significado crítico para la seguridad operativa. Por ello, se debe evitar de forma estricta que cualquier invocación a funciones no `void` quede sin procesar.

Cuando un valor devuelto no es utilizado, se abren múltiples vectores de riesgo. Entre ellos se encuentran la omisión de comprobaciones de estado en operaciones sensibles, la pérdida de indicadores de error o la falta de validación de datos externos. Esto puede agravarse por características específicas como la semántica de movimiento, los constructores de copia y la creación implícita de objetos. Además, si la función retorna un puntero o referencia, ignorarla puede dejar recursos sin liberar, estructuras sin validar o comportamientos anómalos sin identificar. Dichos errores podrían eventualmente transformarse en fallos de seguridad explotables, especialmente en contextos donde el flujo de control depende de la evaluación correcta del retorno.

El tratamiento seguro del valor retornado consiste en utilizarlo de forma explícita e inequívoca. El desarrollador debe recoger dicho valor, procesarlo, verificarlo o documentar su intención mediante patrones seguros. En particular, aplicar **validaciones exhaustivas** y **comprobaciones de error robustas** es esencial cuando el retorno representa un estado crítico. También se recomienda que las funciones diseñadas para retornos significativos utilicen tipos autoexplicativos como `std::optional` o mecanismos modernos como `std::expected`, los cuales obligan a un tratamiento más estructurado del resultado.

En proyectos complejos, el uso sistemático de anotaciones, atributos del compilador y wrappers modernos puede elevar el nivel de seguridad del código. Emplear atributos como `[[nodiscard]]` obliga a tratar el valor retornado, reduciendo la posibilidad de omisiones. Asimismo, combinar este principio con patrones de diseño como RAII permite que el retorno represente estados transitorios útiles para verificar la validez de objetos, evitando fugas y situaciones de inconsistencia.

Riesgos asociados

- Pérdida de información crítica y resultados de errores no detectados.
- Dificultad para depurar el programa debido a comportamientos inesperados.
- Saltos silenciosos de errores que pueden derivar en fallos de seguridad posteriores.
- Propagación de estados inconsistentes en la ejecución del programa.
- Vulnerabilidades de seguridad al ignorar códigos de error que indican fallos en operaciones seguras (por ejemplo, verificación de límites, autenticación, lectura de archivos).
- Acceso a recursos no inicializados o liberación inapropiada de recursos.
- Posible denegación de servicio (DoS) si se ignoran los fallos en operaciones críticas.

Buenas prácticas

- Verificar y utilizar siempre el valor devuelto por funciones que indican estado o error.

```
  //-- Se invoca la función sin comprobar el resultado
  bool autenticado = autenticarUsuario(usuario, clave);
  accederSistema();
```

```
//-- Se valida el resultado antes de continuar
  bool autenticado = autenticarUsuario(usuario, clave);
  if (!autenticado) {
    registrarError("Fallo de autenticación");
    return;
  }
  accederSistema();
```

- Comprobar códigos de retorno en operaciones críticas como acceso a archivos o recursos.

```
  //-- Se abre un archivo sin verificar el resultado
  std::ofstream archivo("datos.txt");
```

```
  archivo << "Información sensible";
```

```
//-- Se valida que el recurso esté disponible antes de usarlo
  std::ofstream archivo("datos.txt");
  if (!archivo.is_open()) {
    registrarError("No se pudo abrir el archivo");
    return;
  }
  archivo << "Información sensible";
```

- Utilizar el valor devuelto para tomar decisiones de control de flujo seguras.

```
  //-- Se ignora el valor devuelto por la función de validación
  validarIntegridad(datos);
  procesarDatos(datos);
```

```
//-- Se utiliza el resultado para decidir si continuar
  if (!validarIntegridad(datos)) {
    registrarError("Datos corruptos");
    return;
  }
  procesarDatos(datos);
```

- Aplicar el atributo `[[nodiscard]]` para evitar que se ignoren valores devueltos importantes.

```
  //-- Función cuyo valor puede ser ignorado accidentalmente
  int calcularChecksum(const std::string& datos) {
    return datos.size();
  }
  calcularChecksum("abc");
```

```
//-- Se fuerza el uso del valor devuelto
  [[nodiscard]] int calcularChecksum(const std::string& datos) {
    return datos.size();
  }
  int checksum = calcularChecksum("abc");
```

2.10.4.2 Estándares metodológicos

- **MISRA CPP-0.1.2**: The value returned by a function shall be used.

2.10.5 [cpp.func.5] - Evitar mantener código inalcanzable en una función

ID	cpp.func.5	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Establece la necesidad de identificar, evitar y eliminar fragmentos de código que nunca pueden ejecutarse dentro de una función. La presencia de código inalcanzable puede indicar errores lógicos, fallos en la gestión del control de flujo o restos de implementaciones anteriores, lo que introduce riesgos de mantenimiento, genera confusión y puede enmascarar vulnerabilidades reales. Mantener un flujo de ejecución claro y verificable contribuye a mejorar la seguridad, la auditabilidad y la calidad del software, reduciendo comportamientos inesperados y facilitando la revisión del código.

Codificación insegura

Se muestra una función que calcula el valor absoluto de un número entero. La intención es entender cómo la presencia de instrucciones después de un `return` genera código inalcanzable, lo que dificulta el mantenimiento, puede ocultar errores lógicos y afecta la calidad del código.

```
#include <iostream>

int valorAbsoluto(int numero) {
  //-- Si el número es negativo, se retorna su inverso
  if (numero < 0) {
    return -numero;
  }
  //-- Retorno directo si es positivo
  return numero;

  //-- Código inalcanzable después del return
  std::cout << "Este mensaje nunca se mostrará" << std::endl;
}

int main() {
  //-- Prueba de la función
  int resultado = valorAbsoluto(-5);
  std::cout << "Resultado: " << resultado << std::endl;
  return 0;
}
```

Codificación recomendada

Se presenta la misma funcionalidad de cálculo del valor absoluto, reorganizando la lógica para evitar instrucciones innecesarias después de un `return`. El objetivo es mantener un

flujo claro, sin código inalcanzable, facilitando el mantenimiento y reduciendo riesgos de errores ocultos.

```
#include <iostream>

int valorAbsoluto(int numero) {
  //-- Se utiliza una variable local para almacenar el resultado
  int resultado;

  //-- Evaluación del signo del número
  if (numero < 0) {
    resultado = -numero;
  } else {
    resultado = numero;
  }
  //-- Único punto de retorno, sin código posterior
  return resultado;
}

int main() {
  //-- Prueba de la función
  int resultado = valorAbsoluto(-5);
  std::cout << "Resultado: " << resultado << std::endl;
  return 0;
}
```

2.10.5.1 Descripción

El código inalcanzable constituye un problema relevante tanto de calidad como de seguridad. No solo implica incoherencias en el flujo de ejecución, sino que además puede ocultar fallos lógicos, generar falsas expectativas sobre el comportamiento del programa y dificultar tareas de auditoría y análisis estático. Su eliminación es fundamental para asegurar que el flujo de control sea claro, verificable y coherente, especialmente en sistemas críticos donde cualquier comportamiento inesperado puede derivar en fallos graves. Este requisito busca fomentar un desarrollo disciplinado mediante la detección sistemática de secciones de código que nunca serán ejecutadas debido a retornos anticipados, condiciones imposibles, interrupciones de flujo o construcciones erróneas.

El código inalcanzable puede introducir múltiples riesgos. Entre los más relevantes se encuentra el deterioro de la **integridad** del sistema, ya que puede ocultar rutas de ejecución mal implementadas o inconsistentes. También afecta la **auditabilidad**, dificultando los análisis estáticos y fomentando la creación de rutas muertas que nunca son evaluadas en pruebas. Este tipo de fragmentos también promueve el incremento del tamaño binario sin aportar funcionalidad real, lo que resulta perjudicial en contextos embebidos. Evitar estas secciones aumenta la claridad del flujo de control, reduce la

complejidad cognitiva y minimiza posibles errores introducidos durante cambios futuros.

Una práctica para evitar implementar código inalcanzable consiste en estructurar funciones de forma que cada una tenga una única responsabilidad y un conjunto limitado y coherente de rutas de retorno. Esto no solo reduce la probabilidad de crear código inalcanzable, sino que también mejora significativamente la seguridad y mantenibilidad. Las funciones extensas presentan mayor tendencia a acumular bloques obsoletos o fragmentos inactivos tras refactorizaciones. Dividirlas en unidades pequeñas y bien definidas permite detectar más rápidamente incoherencias en el flujo. Otra estrategia recomendable es establecer patrones de retorno consistentes, evitando mezclar múltiples estilos o criterios de terminación.

La presencia de código inalcanzable suele generar advertencias en compiladores modernos y herramientas de análisis estático. Atender estas advertencias ayuda a evitar defectos más amplios, dado que en muchos casos la inalcanzabilidad es un síntoma de condiciones lógicas mal planteadas. En entornos donde se exigen certificaciones de seguridad o cumplimiento de estándares, la eliminación sistemática de código inalcanzable es un requisito estricto. Además, su ausencia mejora la cobertura de pruebas unitarias, ya que permite garantizar que todas las rutas relevantes son efectivamente ejercidas.

Riesgos asociados

- Pérdida de la claridad y legibilidad del código, dificultando el mantenimiento.
- Incremento en la superficie de posibles defectos y errores lógicos no detectados.
- Confusión para los desarrolladores, provocando malas interpretaciones de la lógica del programa.
- Potencial introducción de vulnerabilidades si el código inalcanzable contiene fallos de seguridad o llamadas a funciones inseguras.
- Generación de advertencias o errores en herramientas de análisis estático, entorpeciendo la revisión de código.
- Compilaciones innecesariamente pesadas si el código inalcanzable no es eliminado automáticamente por el compilador.
- Posibilidad de que el código inalcanzable sea "reactivado" por cambios futuros, reintroduciendo riesgos inadvertidamente.

Buenas prácticas

- Eliminar o refactorizar cualquier instrucción ubicada después de una sentencia de retorno, ruptura o lanzamiento de excepción.

```
//-- Código con instrucciones posteriores a return que nunca se ejecutan
```

```
int calcular(int a) {
  if (a < 0) {
    return -1;
    a = 0;
  }
  return a * 2;
}
```

```
//-- Reestructuración eliminando la instrucción inalcanzable
int calcular(int a) {
  if (a < 0) {
    return -1;
  }
  return a * 2;
}
```

- Evitar condiciones lógicas que siempre evalúan al mismo resultado y generan ramas inalcanzables.

```
//-- Condición redundante que impide ejecutar el else
void procesar(int x) {
  if (x > 0 || x <= 0) {
    ejecutar();
  } else {
    registrarError();
  }
}
```

```
//-- Simplificación eliminando la rama innecesaria
void procesar(int x) {
  ejecutar();
}
```

- Retirar bloques de código posteriores a una excepción lanzada dentro del mismo flujo.

```
//-- Código posterior a throw que nunca se alcanza
void validar(int edad) {
  if (edad < 0) {
    throw std::invalid_argument("Edad inválida");
    registrarAuditoria();
  }
}
```

```
//-- Separación de la auditoría antes del lanzamiento de la excepción
void validar(int edad) {
```

```
  if (edad < 0) {
    registrarAuditoria();
    throw std::invalid_argument("Edad inválida");
  }
}
```

2.10.5.2 Estándares metodológicos

- **CERT/CPP-DCL50**: Do not define a C-style variadic function.

2.10.6 [cpp.func.6] - Asegurar que toda función "no void" devuelve un valor en todas sus rutas de ejecución

ID	cpp.func.6	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este control exige que toda función cuyo tipo de retorno no sea `void` garantice la devolución de un valor válido en todas las posibles rutas de ejecución, evitando comportamientos indefinidos, fugas de flujo, inconsistencias lógicas y vulnerabilidades explotables. El objetivo es reforzar la integridad y predictibilidad del software, reduciendo fallos en tiempo de ejecución y previniendo usos de memoria no inicializada, así como asegurar que el código cumpla prácticas seguras y deterministas.

Codificación insegura

Se muestra una función que calcula un descuento en base a un tipo de cliente. La intención es devolver un valor entero según el caso evaluado. Se puede observar qué ocurre cuando no se cubren todas las rutas de ejecución en una función que no es `void` y cómo esto puede generar comportamiento indefinido.

```
#include <iostream>

//-- Función que calcula el descuento según el tipo de cliente
int calcularDescuento(int tipoCliente) {
  //-- Cliente estándar
  if (tipoCliente == 1) {
    return 5;
  }
  //-- Cliente premium
  else if (tipoCliente == 2) {
    return 10;
  }
  //-- No se contempla ningún otro caso
}
```

```
int main() {
  int tipo = 3;
  int descuento = calcularDescuento(tipo);

  std::cout << "Descuento aplicado: " << descuento << "%" << std::endl;
  return 0;
}
```

Codificación recomendada

Se muestra la misma lógica de cálculo de descuento, asegurando que todas las rutas de ejecución devuelvan un valor. Se añade un caso por defecto que garantiza un retorno válido, evitando comportamiento indefinido y cumpliendo la regla de que toda función no `void` debe retornar un valor.

```
#include <iostream>

//-- Función que calcula el descuento según el tipo de cliente
int calcularDescuento(int tipoCliente) {
  //-- Cliente estándar
  if (tipoCliente == 1) {
    return 5;
  }
  //-- Cliente premium
  else if (tipoCliente == 2) {
    return 10;
  }
  //-- Caso por defecto para cualquier otro tipo
  else {
    return 0;
  }
}

int main() {
  int tipo = 3;
  int descuento = calcularDescuento(tipo);

  std::cout << "Descuento aplicado: " << descuento << "%" << std::endl;
  return 0;
}
```

2.10.6.1 Descripción

El incumplimiento de devolver un valor en una función definida con un tipo de retorno distinto a `void` constituye una fuente clásica de comportamientos indefinidos. Este comportamiento puede introducir riesgos de seguridad al permitir que partes críticas del programa operen basadas en valores residuales, memoria no inicializada o rutas de

ejecución no anticipadas. Por ello, **garantizar explícitamente el retorno en todas las rutas** es una práctica fundamental tanto para la robustez como para la seguridad del software.

Una función con tipo de retorno no `void` puede estar sujeta a múltiples ramas de ejecución: condiciones, bucles, excepciones o rutas implícitas. Si alguna de estas rutas omite la instrucción `return`, el programa podría utilizar un valor basura, provocar un fallo lógico o quedar expuesto a manipulaciones maliciosas. **En el contexto de C++ moderno**, garantizar retornos explícitos también reduce riesgos derivados de optimizaciones del compilador, ejecución especulativa y transformaciones de código realizadas durante el proceso de construcción.

En funciones con múltiples pasos lógicos, especialmente en sistemas críticos o embebidos, un error de retorno puede pasar desapercibido. Herramientas de análisis estático como clang-tidy, Cppcheck o los análisis integrados de compiladores como GCC y Clang suelen emitir advertencias cuando detectan rutas sin retorno. Sin embargo, no siempre logran identificar casos en estructuras complejas, por lo que es responsabilidad del desarrollador adoptar un patrón de diseño robusto.

Algunos patrones recomendados son:

- **Return único**: Garantiza claridad y reduce errores humanos.
- **Uso de tipos opcionales**: Con `std::optional` se puede gestionar retornos condicionados sin perder claridad.
- **Valores centinela controlados**: Cuando corresponda, devolver un valor conocido que indique condición especial.
- **Enfoque RAII**: Mantener consistencia incluso en escenarios de liberación automática de recursos.
- **Validación temprana**: Retornar al inicio en casos de error simplifica el diseño.

La ausencia de retornos definidos puede derivar en vulnerabilidades relacionadas con corrupción de memoria, uso de datos no inicializados, fuga de información sensible o bypass de controles lógicos. En particular, funciones que determinan permisos, validaciones o cálculos criptográficos deben ser extremadamente rigurosas en cumplir esta regla. **Un retorno incorrecto o indefinido puede comprometer la integridad de todo el sistema**.

Cuando una función puede lanzar una excepción, esto constituye una ruta válida para abandonar la función sin devolver un valor. Sin embargo, es crucial que el código sea consistente: o todas las rutas devuelven un valor o determinadas rutas lanzan explícitamente una excepción. La mezcla de ambas estrategias sin un diseño claro puede confundir al lector e introducir defectos difíciles de rastrear.

Riesgos asociados

- Comportamiento indefinido en tiempo de ejecución.
- Generación de valores basura o aleatorios como resultado de la función.
- Fallos de seguridad por toma de decisiones basadas en valores incorrectos.
- Dificultad en la detección de errores por parte de herramientas de análisis estático.
- Posible explotación de vulnerabilidades si el valor retornado es usado en rutas críticas.
- Inestabilidad de la aplicación por comportamiento inconsistente.
- Errores de lógica y fallos funcionales difíciles de depurar.
- Potencial para causar condiciones de carrera o corrupción de datos si el valor retornado es usado en operaciones críticas.

Buenas prácticas

- Garantizar que todas las rutas de ejecución retornan explícitamente un valor.

```
//-- Función con ruta sin retorno explícito
int calcular(int x) {
  if (x > 0) {
    return x * 2;
  }
}
```

```
//-- Todas las rutas retornan un valor definido
int calcular(int x) {
  if (x > 0) {
    return x * 2;
  }
  return 0;
}
```

- Incluir un caso `default` en estructuras `switch` para asegurar el retorno.

```
//-- Switch sin cubrir todos los casos posibles
int obtenerValor(int opcion) {
  switch (opcion) {
    case 1:
      return 10;
    case 2:
      return 20;
  }
```

```
}
```

```
//-- Switch con caso default que garantiza retorno
int obtenerValor(int opcion) {
  switch (opcion) {
    case 1:
      return 10;
    case 2:
      return 20;
    default:
      return -1;
  }
}
```

- Evitar retornos condicionales incompletos en bloques `try-catch`.

```
//-- Posible ruta sin retorno si no se lanza excepción
int dividir(int a, int b) {
  try {
    if (b != 0) {
      return a / b;
    }
  } catch (...) {
    return 0;
  }
}
```

```
//-- Retorno garantizado tras bloque try-catch
int dividir(int a, int b) {
  try {
    if (b != 0) {
      return a / b;
    }
  } catch (...) {
    return 0;
  }
  return 0;
}
```

- Agregar retorno final de seguridad cuando existan múltiples condiciones encadenadas.

```
//-- Condiciones múltiples sin retorno final
int evaluar(int x) {
  if (x == 1) {
    return 100;
  } else if (x == 2) {
```

```
    return 200;
  }
}
```

```
//-- Retorno final que cubre valores no contemplados
int evaluar(int x) {
  if (x == 1) {
    return 100;
  } else if (x == 2) {
    return 200;
  }
  return -1;
}
```

2.10.6.2 Estándares metodológicos

- **MISRA CPP-9.6.5**: Non-void function must return a value on all paths.

2.10.7 [cpp.func.7] - Evitar definir una función variádica en estilo C

ID	cpp.func.7	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

El uso de funciones variádicas de estilo C mediante `...` (puntos suspensivos) introduce riesgos significativos de seguridad y fiabilidad, ya que omiten verificaciones de tipos y dependen de convenciones de llamada que pueden ser mal utilizadas o provocar comportamientos indefinidos. Este requisito fomenta el uso de alternativas seguras como plantillas variádicas, sobrecarga u objetos contenedores, las cuales proporcionan comprobaciones de tipos en tiempo de compilación y reducen errores de memoria. Su aplicación minimiza vulnerabilidades asociadas al manejo inseguro de parámetros, corrupciones de pila y posibles vectores de explotación.

Codificación insegura

Gestión de una función que recibe un número variable de argumentos para sumar valores enteros. Se muestra el uso de una función variádica al estilo C con `va_list`, donde el compilador no puede verificar tipos ni cantidad real de parámetros, lo que puede provocar comportamientos indefinidos si se usan de forma incorrecta.

```
#include <iostream>
#include <cstdarg>

int sumar(int cantidad, ...) {
  //-- inicio de lista de argumentos variables
```

```
  va_list args;
  va_start(args, cantidad);

  int total = 0;

  //-- recorrido de los argumentos sin verificación de tipo
  for (int i = 0; i < cantidad; ++i) {
    total += va_arg(args, int);
  }
  //-- fin de uso de la lista
  va_end(args);
  return total;
}

int main() {
  //-- llamada correcta pero sin garantías de seguridad de tipos
  int resultado = sumar(3, 10, 20, 30);

  std::cout << "Resultado: " << resultado << std::endl;
  return 0;
}
```

Codificación recomendada

Implementación de una función para sumar un número variable de valores enteros utilizando plantillas variádicas de C++. El compilador valida los tipos en tiempo de compilación, evitando conversiones inseguras y errores típicos de las funciones variádicas en estilo C.

```
#include <iostream>

template<typename T>
T sumar(T valor) {
  //-- caso base de la recursión
  return valor;
}

template<typename T, typename... Args>
T sumar(T primero, Args... resto) {
  //-- expansión segura de parámetros con verificación de tipos en compilación
  return primero + sumar(resto...);
}

int main() {
  //-- llamada con comprobación de tipos en tiempo de compilación
  int resultado = sumar(10, 20, 30);

  std::cout << "Resultado: " << resultado << std::endl;
```

```
  return 0;
}
```

2.10.7.1 Descripción

Usar funciones variádicas de estilo C mediante el operador `...` constituye una práctica insegura en C++, debido a que elimina cualquier tipo de comprobación de tipos y delega la gestión de argumentos variables a mecanismos de bajo nivel basados en la pila. Este enfoque, heredado del lenguaje C, contradice los principios modernos del C++ orientados a la seguridad, la robustez y la comprobación estática. Las funciones variádicas en estilo C dependen de especificadores implícitos y del uso de macros como `va_list`, `va_start` y `va_arg`, las cuales pueden ser mal utilizadas y provocar lectura o escritura fuera de límites, desbordamientos, corrupciones de pila y errores lógicos de difícil diagnóstico.

El primer problema crítico es la **ausencia total de verificación de tipos**. En una función definida con `...`, el compilador no puede garantizar que los tipos suministrados por el llamador coincidan con los esperados internamente por la función. Si el desarrollador asume un tipo incorrecto, el acceso mediante `va_arg` efectuará conversiones inválidas, alteraciones de memoria y posibles fallos en tiempo de ejecución. Esta ausencia de contratos de tipo genera fragilidad y oportunidades para que un atacante manipule entradas malformadas con fines maliciosos.

Otro aspecto crítico es que la interfaz variádica estilo C es **intrínsecamente dependiente del orden** y del tamaño de los argumentos, lo cual dificulta enormemente su análisis estático y dinámico. Las herramientas de análisis carecen de información semántica para evaluar la corrección del número y tipo de parámetros, lo que complica la detección temprana de errores. En contextos de sistemas empotrados, automoción o entornos regulados, esta opacidad provoca incompatibilidades con requisitos de certificación y calidad.

Adicionalmente, las funciones variádicas estilo C **no permiten la propagación segura de excepciones** ni la utilización coherente de RAII, pilares fundamentales del diseño seguro en C++. Tampoco interactúan adecuadamente con mecanismos modernos como `constexpr`, `noexcept` o la deducción automática de tipos, limitando su integración en bases de código seguras y robustas. En escenarios multihilo, este estilo de funciones puede agravar condiciones de carrera si se utilizan en contextos con memoria compartida o sin sincronización explícita.

Alternativas seguras modernas

C++ ofrece alternativas potentes y seguras para sustituir por completo las funciones variádicas estilo C. Las **plantillas variádicas** constituyen la alternativa más robusta, ya que permiten recibir un número arbitrario de argumentos con comprobación completa en tiempo de compilación. Gracias al mecanismo de parámetros de plantilla, el

compilador garantiza que el tipo de cada argumento esté correctamente representado en la definición de la función, posibilitando optimizaciones y evitando errores de conversión.

La **sobrecarga de funciones** es una solución eficaz cuando el conjunto de combinaciones de parámetros es limitado. Este enfoque prioriza la claridad, permite a las herramientas de análisis comprender relaciones directas entre firma y uso, y proporciona al compilador información precisa de tipos, eliminando riesgos de accesos indebidos a memoria.

Otra alternativa segura consiste en emplear **contenedores estándar** como `std::vector<T>` o `std::initializer_list<T>`, que encapsulan la variabilidad del número de argumentos dentro de estructuras manejadas de manera segura. Esto facilita la gestión de memoria, reduce el riesgo de errores de lectura y escritura, y permite usar las facilidades modernas de iteración segura del estándar.

Riesgos asociados

- Lectura de memoria no inicializada o fuera de los límites (out-of-bounds read) al acceder a argumentos no definidos correctamente.
- Condiciones de desbordamiento de buffer debido a la falta de control de tipo y número de argumentos.
- Inyección de datos maliciosos, ya que el compilador no puede verificar el tipo de los argumentos pasados.
- Comportamiento indefinido en tiempo de ejecución si los argumentos no coinciden con lo esperado por la función variádica.
- Difícil mantenimiento y menor legibilidad del código, incrementando el riesgo de errores en futuras modificaciones.
- Dificultad para depurar y encontrar errores debido a la ausencia de comprobaciones automáticas del compilador.
- Vulnerabilidad ante exploits como formateo de cadenas (por ejemplo, ataques relacionados con `printf/scanf`).
- Mayor propensión a errores de seguridad en aplicaciones críticas, especialmente en manejo de datos sensibles.

Buenas prácticas

- Utilizar plantillas variádicas (variadic templates) en lugar de funciones variádicas estilo C con `...` y `va_list`.

```
//-- Uso de función variádica estilo C sin verificación de tipos
int sumar(int count, ...) {
  va_list args;
```

```
  va_start(args, count);
  int total = 0;
  for(int i = 0; i < count; ++i) {
    total += va_arg(args, int);
  }
  va_end(args);
  return total;
}
```

```
//-- Uso de plantilla variádica con comprobación de tipos en compilación
template<typename... Args>
int sumar(Args... args) {
  return (args + ...);
}
```

- Evitar la falta de seguridad de tipos reemplazando argumentos indefinidos por sobrecarga o contenedores tipados.

```
//-- Uso de argumentos variables sin control de tipo
void log(const char* format, ...) {
  va_list args;
  va_start(args, format);
  vprintf(format, args);
  va_end(args);
}
```

```
//-- Uso de sobrecarga y tipos seguros
void log(const std::string& mensaje) {
  std::cout << mensaje;
}

void log(int valor) {
  std::cout << valor;
}
```

- Preferir estructuras modernas como `std::initializer_list` o contenedores estándar en lugar de listas de argumentos variables.

```
//-- Recepción de múltiples argumentos mediante estilo C
double promedio(int count, ...) {
  va_list args;
  va_start(args, count);
  double suma = 0;
  for(int i = 0; i < count; ++i) {
    suma += va_arg(args, double);
  }
  va_end(args);
```

```
  return suma / count;
}
```

```
//-- Uso de std::initializer_list con tipo definido
double promedio(std::initializer_list<double> valores) {
  double suma = 0;
  for(double v : valores) {
    suma += v;
  }
  return suma / valores.size();
}
```

2.10.7.2 Estándares metodológicos

- **MISRA CPP-8.2.2**: No C-style or functional-notation casts.

2.10.8 [cpp.func.8] - Asegurar que las expresiones de control no sean invariantes

ID	cpp.func.8	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Las expresiones utilizadas en estructuras de control como `if`, `while`, `for` y `switch` no deberían ser invariantes o constantes, con el fin de evitar comportamientos inesperados, errores lógicos y vulnerabilidades explotables. Validar dinámicamente que las condiciones realmente dependan de variables o estados cambiantes permite reducir riesgos de fallos silenciosos, evitar bucles infinitos y mitigar vulnerabilidades relacionadas con flujos de ejecución incorrectamente definidos o manipulables.

Codificación insegura

Se muestra un programa donde la condición de una estructura de control es constante y no depende del estado del programa. La intención es observar cómo una expresión invariante en un bucle hace que la condición pierda sentido lógico, dificultando el mantenimiento y pudiendo generar errores si se elimina la salida forzada del bucle.

```
#include <iostream>

int main() {
  //-- Variable de ejemplo
  int contador = 0;

  //-- Bucle con expresión de control invariante (siempre verdadera)
  while (10 > 5) {
```

```
    //-- Incrementa el contador
    contador++;

    //-- Salida forzada para evitar bucle infinito
    break;
  }
  //-- Muestra el valor final
  std::cout << "Contador: " << contador << std::endl;
  return 0;
}
```

Codificación recomendada

Se muestra un programa donde la condición del bucle depende de una variable que cambia durante la ejecución. La intención es asegurar que la expresión de control refleje el estado real del programa, facilitando la comprensión, el mantenimiento y evitando comportamientos inesperados.

```
#include <iostream>

int main() {
  //-- Variable de control del bucle
  int contador = 0;

  //-- Bucle con expresión de control dependiente del estado
  while (contador < 1) {
    //-- Incrementa el contador modificando la condición
    contador++;
  }
  //-- Muestra el valor final
  std::cout << "Contador: " << contador << std::endl;
  return 0;
}
```

2.10.8.1 Descripción

Una expresión invariante es aquella cuyo resultado no cambia durante la ejecución del programa, ya sea porque está basada únicamente en valores constantes o porque depende de variables que nunca se modifican antes o durante la evaluación de la condición. Cuando este problema ocurre, el flujo del programa se vuelve rígido e inalterable, lo que puede ocultar fallos lógicos, permitir ejecuciones repetitivas no deseadas o exponer la aplicación a estados inconsistentes que son difíciles de detectar y depurar.

En contextos de seguridad, este tipo de fallos puede derivar en **denegaciones de servicio**, errores persistentes o ramas de ejecución nunca alcanzadas, lo cual representa un problema especialmente crítico en componentes de seguridad como validadores de

entrada, parsers, lógica criptográfica o mecanismos de autorización. Una condición invariable en un sistema complejo puede inutilizar una comprobación esencial y permitir el avance de una operación que debería bloquearse.

Herramientas modernas de análisis estático como clang-tidy, Cppcheck o SonarQube permiten identificar expresiones invariantes mediante reglas específicas. Estas herramientas detectan patrones peligrosos tales como condiciones constantes, expresiones booleanas triviales o comparaciones redundantes. El uso de estas herramientas debe integrarse en el pipeline de compilación para asegurar que los desarrolladores no introduzcan este tipo de fallos de manera accidental.

Una expresión de control invariable puede tener consecuencias serias en la seguridad del software. Por ejemplo, un control de acceso basado en una condición incorrectamente definida podría permitir que un atacante ejecute una operación sin las autorizaciones necesarias. De igual manera, un bucle infinito generado por una condición constante podría permitir ataques de denegación de servicio que saturen los recursos del sistema.

Riesgos asociados

- Introducción de puertas traseras o funcionalidad indeseada debido a condiciones de control que siempre se evalúan igual (por ejemplo, bucles infinitos o condicionales que nunca se ejecutan).
- Pérdida de alcance de pruebas y cobertura de código, ya que rutas muertas nunca serán ejecutadas o testeadas.
- Vulnerabilidad a ataques de denegación de servicio (DoS) por bucles sin condición de salida.
- Desviación accidental del flujo normal del programa causando errores de lógica y estados inconsistentes.
- Dificultad en el mantenimiento del código y aumento en la posibilidad de introducción de vulnerabilidades.
- Exposición de información sensible si la invarianza expone rutas o funciones internas por error.
- Ignorancia inadvertida de controles de seguridad, dejando el sistema en un estado inseguro permanente.

Buenas prácticas

- Evitar condiciones constantes en estructuras de control que no dependan del estado real del programa.

```
//-- Condición constante que hace el bloque inalcanzable
if (false) {
```

```
  procesarDatos();
}
```

```
//-- Condición dependiente del estado real del programa
if (datosDisponibles) {
  procesarDatos();
}
```

- No utilizar variables que no cambian dentro del flujo como condición de bucles.

```
//-- Variable que nunca cambia dentro del bucle
bool continuar = true;
while (continuar) {
  procesar();
}
```

```
//-- Variable de control actualizada dentro del bucle
bool continuar = true;
while (continuar) {
  continuar = verificarCondicion();
  procesar();
}
```

- Evitar comparaciones tautológicas o contradictorias que siempre evalúan igual.

```
//-- Comparación que siempre es verdadera
int x = obtenerValor();
if (x == x) {
  autorizarAcceso();
}
```

```
//-- Comparación significativa contra valor esperado
int x = obtenerValor();
if (x == valorPermitido) {
  autorizarAcceso();
}
```

- No usar macros o constantes que conviertan expresiones de control en invariantes.

```
//-- Macro que fuerza condición siempre verdadera
#define VALIDAR 1
if (VALIDAR) {
  ejecutarOperacion();
}
```

```
//-- Resultado de función que evalúa condición real
if (validarOperacion()) {
  ejecutarOperacion();
}
```

- Revisar que las condiciones no dependan de valores no modificados dentro del mismo alcance.

```
//-- Variable no modificada que mantiene condición invariable
int intentos = 3;
while (intentos > 0) {
  procesarIntento();
}
```

```
//-- Variable actualizada que altera la condición del bucle
int intentos = 3;
while (intentos > 0) {
  procesarIntento();
  --intentos;
}
```

2.10.8.2 Estándares metodológicos

- **MISRA CPP-0.0.2**: Controlling expressions should not be invariant.

2.10.9 [cpp.func.9] - Preferir expresiones lambda frente a "std::bind"

ID	cpp.func.9	Uso	Recomendado
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Se promueve el uso consistente de expresiones `lambda` como una medida de seguridad y claridad frente al empleo de `std::bind`, que puede generar comportamientos ambiguos, pérdida de legibilidad y errores difíciles de depurar. Las `lambda` proporcionan un mecanismo más explícito, seguro y moderno para capturar variables, evitando riesgos relacionados con referencias colgantes, errores de conversión y omisiones involuntarias en la semántica de captura.

Codificación insegura

Uso de `std::bind` para adaptar una función y utilizarla como predicado en un algoritmo estándar. Se busca filtrar valores mayores que un umbral fijo dentro de un contenedor. Permite observar cómo se enlazan parámetros y cómo esto puede afectar a la legibilidad y mantenibilidad del código.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

//-- Función que compara si un valor es mayor que un umbral
bool esMayorQue(int valor, int umbral) {
  return valor > umbral;
}

int main() {
  //-- Vector de datos de ejemplo
  std::vector<int> datos = {1, 5, 10, 15, 20};

  //-- Se fija el umbral en 10 usando std::bind
  auto predicado = std::bind(esMayorQue, std::placeholders::_1, 10);

  //-- Se recorren y muestran los valores que cumplen la condición
  for (int valor : datos) {
    if (predicado(valor)) {
      std::cout << valor << std::endl;
    }
  }
  return 0;
}
```

Codificación recomendada

Uso de una expresión `lambda` para definir directamente el predicado dentro del mismo contexto donde se utiliza. Se persigue el mismo objetivo funcional, pero mejorando la claridad, reduciendo errores por desajuste de parámetros y facilitando el mantenimiento.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
  //-- Vector de datos de ejemplo
  std::vector<int> datos = {1, 5, 10, 15, 20};

  //-- Umbral definido en el mismo contexto
  int umbral = 10;

  //-- Lambda que captura el umbral y evalúa la condición
  auto predicado = [umbral](int valor) {
    return valor > umbral;
  };
```

```
  //-- Se recorren y muestran los valores que cumplen la condición
  for (int valor : datos) {
    if (predicado(valor)) {
      std::cout << valor << std::endl;
    }
  }
  return 0;
}
```

2.10.9.1 Descripción

El uso excesivo o inapropiado de `std::bind` puede introducir problemas de legibilidad, errores sutiles y riesgos de seguridad derivados de capturas implícitas o transformaciones inesperadas de parámetros. Por ello, se recomienda **preferir el uso de expresiones lambda**, las cuales ofrecen una semántica más clara, una captura más explícita y reducen la probabilidad de comportamientos indeterminados o vulnerabilidades derivadas de usos incorrectos del lenguaje. Este enfoque, además de alinearse con las recomendaciones modernas del estándar C++, contribuye a un estilo de codificación más seguro y mantenible.

El uso de expresiones `lambda` presenta beneficios significativos en escenarios donde la seguridad y la robustez del código son prioritarias. Gracias a su expresividad, las `lambdas` permiten definir directamente el comportamiento deseado sin recurrir a un nivel adicional de indeterminación como ocurre con `std::bind`. Además, su sintaxis posibilita exponer de forma explícita la captura de variables y el modelo utilizado para gestionar su vida útil. Esto reduce riesgos relacionados con accesos a memoria ya liberada, errores de concurrencia o inconsistencias lógicas.

La semántica explícita de las `lambdas` también facilita la aplicación de herramientas automáticas de análisis estático, que pueden detectar con mayor precisión problemas de integridad y flujos de datos peligrosos. Por su parte, `std::bind` oculta parte de la lógica interna, generando estructuras auxiliares poco transparentes que dificultan el seguimiento de dependencias. Este tipo de opacidad puede ser aprovechado por actores maliciosos si el código se combina con otros patrones inseguros.

El mecanismo de `std::bind` fue diseñado originalmente para facilitar la aplicación parcial de funciones. Sin embargo, su sintaxis compacta frecuentemente introduce confusión respecto al orden en que se aplican los parámetros o la forma en que se capturan y transforman los valores. Esto puede derivar en vulnerabilidades que afecten a la integridad del flujo de ejecución. En cambio, las expresiones lambda permiten declarar la lógica de forma explícita, mostrando directamente qué valores se están usando, cómo se capturan y qué comportamiento se espera en el ámbito de ejecución. Al emplear una `lambda`, el lector entiende con exactitud qué datos se están manejando y en qué forma se pasan a la función, lo que reduce la superficie de error y

facilita auditorías de integridad. Este control resulta especialmente relevante cuando los parámetros incluyen datos sensibles o configuraciones críticas del sistema.

Riesgos asociados

- Código menos legible, dificultando la revisión y auditoría de seguridad.
- Errores sutiles de captura de variables o parámetros, aumentando la probabilidad de bugs y vulnerabilidades.
- Menor claridad en el control del alcance y ciclo de vida de los objetos capturados, lo que puede derivar en "dangling references" o fugas de memoria.
- Complejidad innecesaria en el manejo de errores, pudiendo ocultar condiciones peligrosas o excepciones mal gestionadas.
- Posibilidades de errores en la vinculación de argumentos, generando un comportamiento inesperado o no intencional.
- Dificultad para aplicar análisis estático de código eficiente, reduciendo la capacidad de detectar patrones inseguros automáticamente.
- Susceptibilidad a errores de tipo en tiempo de compilación o ejecución, al no aprovechar las comprobaciones de `type safety` que las expresiones `lambda` pueden proporcionar.

Buenas prácticas

- Utilizar expresiones `lambda` para mejorar la legibilidad y mantener el contexto explícito en lugar de usar `std::bind`.

```
//-- Uso de std::bind que reduce claridad y dificulta mantenimiento
auto f = std::bind(&MiClase::procesar, this, std::placeholders::_1);
```

```
//-- Uso de lambda que hace explícita la captura y el parámetro
auto f = [this](int valor) {
  procesar(valor);
};
```

- Preferir `lambdas` para evitar errores derivados del orden de parámetros con `std::placeholders`.

```
//-- Uso de std::bind con placeholders que puede inducir errores en el orden
auto f = std::bind(sumar, std::placeholders::_2, std::placeholders::_1);
```

```
//-- Uso de lambda que define explícitamente el orden de los argumentos
auto f = [](int a, int b) {
  return sumar(b, a);
```

```
};
```

- Utilizar `lambdas` para mejorar la seguridad al capturar variables, evitando copias o referencias implícitas no deseadas.

```
//-- Uso de std::bind que copia argumentos sin hacerlo evidente
auto f = std::bind(procesarDato, dato);
```

```
//-- Uso de lambda con captura explícita por referencia o valor
auto f = [&dato]() {
  procesarDato(dato);
};
```

- Emplear `lambdas` para facilitar la compatibilidad con herramientas modernas y evitar construcciones obsoletas o menos recomendadas.

```
//-- Uso de std::bind para adaptar función a algoritmo estándar
std::for_each(v.begin(), v.end(), std::bind(&Logger::log, &logger,
std::placeholders::_1));
```

```
//-- Uso de lambda que se integra naturalmente con el algoritmo estándar
std::for_each(v.begin(), v.end(), [&logger](const std::string& msg) {
  logger.log(msg);
});
```

2.10.9.2 Estándares metodológicos

- **C++ Core-F.50**: Use a lambda when a function won't do (to capture local variables, or to write a local function).

2.10.10 [cpp.func.10] - Utilizar "std::optional" para representar valores opcionales

ID	**cpp.func.10**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Se establece el uso de `std::optional` como mecanismo seguro para representar valores que pueden o no estar presentes, evitando estados inválidos, punteros nulos y convenciones implícitas que generan ambigüedad o vulnerabilidades. Al adoptar este patrón, se refuerza la claridad semántica, se previenen accesos indebidos a memoria y se facilita la verificación explícita del estado del valor, lo que reduce defectos lógicos y minimiza riesgos de seguridad relacionados con el uso indebido de objetos no

inicializados. Este enfoque fortalece la robustez del código y contribuye a una ingeniería defensiva más consistente.

Codificación insegura

Gestión de un resultado que puede no existir al buscar un usuario por identificador. Se muestra cómo representar la ausencia de valor usando un valor centinela, lo que obliga al consumidor a conocer reglas implícitas y puede provocar errores de validación o lógica.

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

int buscarEdadUsuario(int id) {
  //-- Simula una búsqueda en base de datos
  if (id == 1) {
    return 30;
  }
  //-- -1 indica que el usuario no existe (valor centinela)
  return -1;
}

int main() {
  int id = 2;

  //-- Se obtiene la edad del usuario
  int edad = buscarEdadUsuario(id);

  //-- Se valida manualmente el valor centinela
  if (edad == -1) {
    std::cout << "Usuario no encontrado" << std::endl;
  } else {
    std::cout << "Edad: " << edad << std::endl;
  }
  return 0;
}
```

Codificación recomendada

Gestión del mismo escenario utilizando un tipo que representa explícitamente la posible ausencia de valor. Se busca que el contrato de la función sea claro y que el compilador obligue a tratar el caso en el que no exista resultado.

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

std::optional<int> buscarEdadUsuario(int id) {
```

```
  //-- Simula una búsqueda en base de datos
  if (id == 1) {
    return 30;
  }
  //-- Retorna std::nullopt si el usuario no existe
  return std::nullopt;
}

int main() {
  int id = 2;

  //-- Se obtiene la edad del usuario como valor opcional
  std::optional<int> edad = buscarEdadUsuario(id);

  //-- Se comprueba explícitamente si existe valor
  if (!edad.has_value()) {
    std::cout << "Usuario no encontrado" << std::endl;
  } else {
    std::cout << "Edad: " << edad.value() << std::endl;
  }
  return 0;
}
```

2.10.10.1 Descripción

El uso de `std::optional` constituye una práctica esencial para mejorar la seguridad, claridad y robustez del software desarrollado en C++. La razón fundamental es que permite representar de forma explícita la ausencia de un valor, eliminando la necesidad de emplear convenciones arriesgadas como valores centinela, punteros nulos o estados no inicializados. Estas convenciones históricas introducen debilidades serias, especialmente en sistemas complejos donde los valores opcionales pueden influir en decisiones de seguridad, integridad de datos y flujos de control que, si no se verifican correctamente, abren la puerta a comportamientos no definidos. **El empleo de `std::optional` permite expresar claramente la intención del diseño y obliga al desarrollador a verificar su validez antes del uso**, reduciendo así un conjunto amplio de errores frecuentes.

En entornos con requerimientos estrictos de seguridad, el uso de valores opcionales no declarados explícitamente puede generar fallos críticos. Valores no inicializados, punteros nulos o estados ilegítimos permiten que un atacante provoque desbordamientos, accesos indebidos a memoria o errores lógicos explotables. Al definir un valor como `std::optional`, se establece un contrato semántico claro en la API: quien consume el valor debe comprobar si existe antes de acceder a él. Esta obligación mecanizada disminuye la probabilidad de fallos de control de flujo y evita que el compilador pase por alto errores que de otro modo no detectaría.

Su aplicación en C++ resulta especialmente relevante en funciones que pueden devolver o no un resultado, parámetros que pueden omitirse o estructuras que representan configuraciones incompletas. En estos casos, `std::optional` **reemplaza el uso de valores especiales o punteros, promoviendo verificaciones explícitas visibles durante la revisión de código**. Además, las herramientas modernas de análisis estático pueden detectar de forma precisa los posibles caminos en los que un valor podría no estar presente, facilitando auditorías de seguridad.

Para reforzar aún más el control de seguridad, es recomendable seguir una serie de prácticas adicionales. **Evitar el uso de** `operator*` **o** `operator->` **sin una verificación explícita previa** reduce riesgos derivados de desatenciones durante la programación. Asimismo, emplear `value_or` cuando existan valores por defecto permite evitar flujos de código inseguros. Otra medida relevante consiste en no utilizar `std::optional` para tipos pesados, dado que podría introducir overhead o incentivar patrones de copia ineficientes. En estos casos, resulta preferible `std::optional<std::reference_wrapper<T>>` o diseños alternativos más optimizados.

El diseño de APIs que emplean `std::optional` debe contemplar cómo interactúa con otros tipos estándar. Por ejemplo, cuando se trata de cadenas, su uso permite distinguir entre cadena vacía y ausencia de cadena, una diferencia crítica en varias validaciones de entrada y controles de autenticación. De igual forma, al integrarse con contenedores u otros tipos derivados, se clarifica la semántica del estado y se evitan errores de lógica que puedan comprometer integridad de datos o controles de acceso internamente en la aplicación.

En aplicaciones concurrentes, **el uso de** `std::optional` **reduce condiciones de carrera relacionadas con estados no inicializados**. Un valor opcional correctamente encapsulado evita que múltiples hilos infieran o asuman estados incorrectos del objeto antes de que sea inicializado. Aunque no sustituye mecanismos de sincronización apropiados, sí contribuye a definir estados consistentes que permiten que estos mecanismos trabajen de forma más confiable.

El enfoque explícito que aporta `std::optional` mejora la trazabilidad lógica del código y simplifica las auditorías de seguridad. Los revisores pueden identificar con facilidad los puntos donde el valor puede no estar disponible y verificar que se realicen validaciones apropiadas. Además, al eliminar ambigüedades, se reduce la probabilidad de que un atacante se beneficie de supuestos incorrectos o de rutas de ejecución poco claras.

Riesgos asociados

- Desreferenciación de punteros nulos (null pointer dereference), causando fallos en tiempo de ejecución.
- Comportamiento indefinido al acceder a datos no inicializados.
- Errores lógicos al suponer que una variable tiene siempre un valor válido.

- Dificultad para distinguir entre valores no válidos y valores válidos predeterminados.
- Propagación inadvertida de valores basura o inválidos en el programa.
- Aumento de la complejidad del manejo de errores y validación de entradas.
- Introducción de vulnerabilidades por falta de comprobaciones de existencia de valor.
- Ataques de denegación de servicio (DoS) debido a fallos inesperados o manejo incorrecto de estados inválidos.
- Incremento del costo de mantenimiento y depuración por errores ocultos relacionados con valores opcionales mal gestionados.

Buenas prácticas

- Evitar el uso de valores centinela (como -1 o `nullptr`) y emplear `std::optional` para representar ausencia de valor explícitamente.

```
//-- Uso de valor centinela para indicar ausencia
int buscarUsuario(int id) {
  if (id == 0) {
    return -1;
  }
  return 42;
}
```

```
//-- Uso de std::optional para expresar ausencia de resultado
std::optional buscarUsuario(int id) {
  if (id == 0) {
    return std::nullopt;
  }
  return 42;
}
```

- Evitar parámetros de salida mediante punteros o referencias para indicar éxito o fracaso y devolver `std::optional` como resultado.

```
//-- Uso de puntero de salida y booleano para controlar éxito
bool obtenerToken(const std::string& user, std::string* token) {
  if (user.empty()) {
    return false;
  }
  *token = "ABC123";
  return true;
}
```

```
//-- Retorno directo de std::optional para simplificar el contrato
std::optional obtenerToken(const std::string& user) {
  if (user.empty()) {
    return std::nullopt;
  }
  return std::string("ABC123");
}
```

- Comprobar explícitamente la presencia de valor antes de acceder a `std::optional` para evitar accesos inseguros.

```
//-- Acceso directo sin verificar presencia
std::optional codigo = obtenerCodigo();
int valor = codigo.value();
```

```
//-- Verificación previa antes de acceder al valor
std::optional codigo = obtenerCodigo();
if (codigo.has_value()) {
  int valor = codigo.value();
}
```

2.10.10.2 Estándares metodológicos

- **C++ Core-F.20**: For "out" output values, prefer return values to output parameters.

2.11 Punteros

2.11.1 [cpp.ptr.1] - Usar "unique_ptr" para expresar exclusividad en la propiedad

ID	cpp.ptr.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

El uso de `unique_ptr` permite gestionar la memoria y los recursos con seguridad, evitando fugas, dobles liberaciones y accesos indebidos. Este mecanismo de propiedad exclusiva garantiza que solo exista un propietario válido del recurso administrado, reduciendo la complejidad de la gestión manual y fortaleciendo la integridad del programa. Además, previene errores comunes asociados a la manipulación de punteros crudos, facilitando la transferencia explícita de propiedad y permitiendo una destrucción automática y segura del recurso cuando deja de ser necesario.

Codificación insegura

Gestión de un recurso dinámico representando la propiedad exclusiva de un objeto. Se muestra cómo se crea, utiliza y libera un recurso en memoria dinámica, permitiendo observar los riesgos cuando la propiedad no está claramente definida y cómo expresar correctamente esa exclusividad en el diseño.

```
#include <iostream>

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 procesar(Recurso* r) {
  //-- Se asume que la función puede usar el recurso
  r->usar();
  //-- Error de diseño: se libera aquí sin ser el propietario real
  delete r;
}

int main() {
  //-- Se reserva memoria dinámicamente
  Recurso* recurso = new Recurso();

  //-- Se pasa el puntero sin definir claramente la propiedad
  procesar(recurso);

  //-- Posible doble liberación si se intenta liberar nuevamente
  // delete recurso; // Provocaría comportamiento indefinido

  return 0;
}
```

Codificación recomendada

Gestión del mismo recurso dinámico expresando explícitamente la propiedad exclusiva mediante un mecanismo estándar de C++. Se observa cómo se transfiere la propiedad de forma segura y cómo la liberación del recurso se realiza automáticamente sin ambigüedades.

```
#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 procesar(std::unique_ptr<Recurso> r) {
  //-- La propiedad se transfiere explícitamente a esta función
  r->usar();
  //-- El recurso se liberará automáticamente al salir del ámbito
}

int main() {
  //-- Se crea el recurso con propiedad exclusiva
  std::unique_ptr<Recurso> recurso = std::make_unique<Recurso>();

  //-- Se transfiere la propiedad usando std::move
  procesar(std::move(recurso));

  //-- recurso ahora es nullptr y no puede usarse
  if (!recurso) {
    std::cout << "Propiedad transferida correctamente" << std::endl;
  }
  return 0;
}
```

2.11.1.1 Descripción

El empleo de `unique_ptr` representa una de las prácticas más relevantes para la gestión segura de memoria y recursos, al proporcionar un mecanismo automático que evita errores graves derivados de la manipulación directa de punteros crudos. El concepto central se basa en la **propiedad exclusiva**, lo que implica que en cada instante solo puede existir una instancia responsable de la vida del recurso. Esta restricción estructural impide copias accidentales del puntero y la liberación múltiple del recurso, promoviendo un comportamiento más previsible y seguro. Además, obliga a que cualquier transferencia de propiedad sea explícita mediante `std::move`, reforzando la claridad del flujo de responsabilidades dentro del código.

Un aspecto fundamental es que `unique_ptr` no solo gestiona memoria dinámica con `new`, sino que también puede administrar cualquier recurso que requiera una liberación explícita, como descriptores de archivo, manejadores de red o recursos de bibliotecas externas. La capacidad de personalizar un **deleter** permite encapsular la lógica de liberación, lo que resulta especialmente valioso en aplicaciones de sistemas, motores de alto rendimiento y entornos donde la liberación incorrecta podría causar comportamientos indeterminados. La gestión automática reduce la probabilidad de fugas y contribuye a la robustez general del software, promoviendo un estilo RAII seguro y consistente.

La estructura de propiedad exclusiva fortalece la integridad del programa al impedir que varios componentes intenten modificar o liberar el mismo recurso. Este enfoque es crucial en contextos de seguridad, donde la integridad de la memoria debe preservarse para evitar corrupción, escalado de privilegios o ejecución de código arbitrario. Si un recurso es propiedad de múltiples punteros crudos, un atacante podría explotar inconsistencias mediante manipulación de punteros, provocando ataques como uso después de liberación o escrituras fuera de rango. La adopción de `unique_ptr` limita este riesgo mediante un modelo estricto de posesión.

Las ventajas de usar `unique_ptr` son:

- **Propiedad exclusiva:** asegura que solo exista un propietario válido del recurso.
- **Transferencias explícitas:** toda cesión requiere `std::move`, garantizando claridad en el flujo de control.
- **Prevención de copias peligrosas:** el puntero no es copiable, evitando duplicación accidental.
- **Destrucción automática:** el recurso se libera de forma segura y determinista cuando el puntero sale de ámbito.
- **Soporte de deleters personalizados:** permite liberar recursos no asociados a memoria dinámica.
- **Integración segura con RAII:** elimina la necesidad de llamadas manuales a `delete`.

La adopción sistemática de `unique_ptr` fomenta un diseño más claro, reduce riesgos inherentes al manejo manual de memoria y evita vulnerabilidades críticas asociadas a corrupción de memoria o fuga de recursos. Su aplicación es especialmente potente en módulos que gestionan buffers, recursos del sistema, estructuras dinámicas y componentes donde la integridad del flujo de posesión es esencial.

Riesgos asociados

- Fugas de memoria por manejo manual inadecuado del ciclo de vida de punteros.
- Doble liberación (double free) del recurso al no dejar clara la propiedad del

puntero.

- Acceso a memoria liberada (dangling pointer) por transferencias incorrectas de propiedad.
- No manejo seguro de excepciones, potencial causando recursos no liberados ante errores.
- Condiciones de carrera en entornos multihilo al compartir punteros sin control explícito de propiedad.
- Confusión sobre la responsabilidad de borrar recursos, dificultando el mantenimiento del código.
- Aumento de la superficie para vulnerabilidades de corrupción de memoria.
- Dificultad para hacer análisis estático y pruebas al desconocer el dueño real del recurso.

Buenas prácticas

- Utilizar `std::unique_ptr` para representar propiedad exclusiva en lugar de punteros crudos.

```
//-- Uso de puntero crudo con liberación manual
MyClass* ptr = new MyClass();
ptr->doSomething();
delete ptr;
```

```
//-- Uso de unique_ptr para propiedad exclusiva y liberación automática
std::unique_ptr ptr = std::make_unique();
ptr->doSomething();
```

- Transferir la propiedad explícitamente mediante `std::move` al pasar `unique_ptr` entre funciones.

```
//-- Copia implícita de puntero crudo sin semántica clara de propiedad
void setResource(MyClass* res);
MyClass* ptr = new MyClass();
setResource(ptr);
```

```
//-- Transferencia explícita de propiedad con std::move
void setResource(std::unique_ptr res);
std::unique_ptr ptr = std::make_unique();
setResource(std::move(ptr));
```

- Evitar compartir propiedad cuando no sea necesario, usando `unique_ptr` en vez de `shared_ptr`.

```
//-- Uso de shared_ptr sin requerir propiedad compartida
std::shared_ptr ptr = std::make_shared();
process(ptr);
```

```
//-- Uso de unique_ptr cuando la propiedad es exclusiva
std::unique_ptr ptr = std::make_unique();
process(std::move(ptr));
```

- Almacenar objetos en contenedores usando `unique_ptr` para evitar fugas y clarificar la propiedad.

```
//-- Contenedor de punteros crudos sin gestión automática
std::vector items;
items.push_back(new MyClass());
```

```
//-- Contenedor de unique_ptr con gestión automática de memoria
std::vector> items;
items.push_back(std::make_unique());
```

2.11.1.2 Estándares metodológicos

- **C++ Core-F.26**: Use a unique_ptr<T> to transfer ownership where a pointer is needed.
- **C++ Core-R.21**: Prefer unique_ptr over shared_ptr unless you need to share ownership.

2.11.2 [cpp.ptr.2] - Usar "shared_ptr" cuando sea necesario compartir propiedad

ID	cpp.ptr.2	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

El uso de `shared_ptr` debería realizarse únicamente cuando exista una necesidad real y justificada de compartir la propiedad de un recurso entre múltiples componentes del sistema, garantizando así una liberación segura y automática de memoria. La correcta gestión del contador de referencias evita fugas, dobles liberaciones y accesos inválidos. Su empleo apropiado asegura integridad estructural en arquitecturas complejas, donde varios módulos requieren acceso concurrente a un mismo objeto, y contribuye a prevenir vulnerabilidades derivadas de una gestión manual insegura de la memoria.

Codificación insegura

Gestión de memoria cuando varios objetos necesitan acceder y gestionar la misma instancia dinámica. Se muestra cómo compartir un recurso entre distintas funciones y qué ocurre cuando la propiedad no está claramente definida, permitiendo entender la importancia de un modelo de propiedad compartida bien gestionado.

```
#include <iostream>

//-- Clase simple que representa un recurso dinámico
class Recurso {
public:
  Recurso() {
    std::cout << "Recurso creado" << std::endl;
  }
  ~Recurso() {
    std::cout << "Recurso destruido" << std::endl;
  }
};

//-- Función que asume propiedad y libera el recurso
void usarRecurso(Recurso* r) {
  delete r;
}

int main() {
  //-- Se crea un recurso dinámico
  Recurso* recurso = new Recurso();

  //-- Dos partes del código creen ser responsables de liberar el recurso
  usarRecurso(recurso);

  //-- Segunda liberación provoca comportamiento indefinido (double free)
  delete recurso;
  return 0;
}
```

Codificación recomendada

Gestión del mismo escenario utilizando un modelo de propiedad compartida. Varias partes del código pueden utilizar el mismo recurso dinámico sin preocuparse por su liberación manual, delegando la responsabilidad al conteo de referencias automático.

```
#include <iostream>
#include <memory>

//-- Clase simple que representa un recurso dinámico
class Recurso {
```

```
public:
  Recurso() {
    std::cout << "Recurso creado" << std::endl;
  }
  ~Recurso() {
    std::cout << "Recurso destruido" << std::endl;
  }
};

//-- Función que comparte el uso del recurso sin asumir liberación manual
void usarRecurso(std::shared_ptr<Recurso> r) {
  std::cout << "Usando recurso" << std::endl;
}

int main() {
  //-- Se crea el recurso con shared_ptr para compartir propiedad
  std::shared_ptr<Recurso> recurso = std::make_shared<Recurso>();

  //-- Se comparte el recurso con otra función
  usarRecurso(recurso);

  //-- El recurso se liberará automáticamente cuando el último shared_ptr deje de
referenciarlo
  return 0;
}
```

2.11.2.1 Descripción

El uso de `shared_ptr` constituye un mecanismo fundamental para gestionar de forma segura la propiedad compartida de objetos en memoria dentro de sistemas donde múltiples entidades requieren acceso simultáneo y garantizado al mismo recurso. Este puntero inteligente implementa un sistema de referencia compartida basado en un conteo automático que asegura que el recurso subyacente solo será destruido cuando ya no existan instancias activas que reclamen dicha propiedad. Esta técnica, cuando se aplica correctamente, reduce riesgos como fugas de memoria, accesos a memoria ya liberada y dobles liberaciones, problemas frecuentemente derivados de una administración manual deficiente del ciclo de vida de los recursos.

La aplicación de `shared_ptr` debe realizarse con un criterio de diseño bien fundamentado. Una práctica esencial es reconocer que no debe emplearse de forma indiscriminada. La propiedad compartida introduce complejidad inherente: incrementa el costo de sincronización en entornos multihilo, dificulta el razonamiento sobre el ciclo de vida de los objetos e incrementa el riesgo de estructuras cíclicas de referencias que pueden provocar fugas silenciosas. Por ello, resulta crucial determinar si la propiedad compartida es realmente necesaria o si soluciones alternativas como `unique_ptr`, referencias constantes o punteros sin propiedad pueden proporcionar un diseño más eficiente y seguro.

Un aspecto clave para el desarrollo seguro consiste en evitar la formación de ciclos de referencia, los cuales se producen cuando dos o más objetos mantienen entre sí referencias compartidas y, como resultado, sus contadores nunca llegan a cero. Para prevenir este problema, se recomienda emplear `weak_ptr` como referencia no propietaria en aquellas relaciones donde no se requiere control de vida, contribuyendo así a una arquitectura más robusta. Este patrón es especialmente importante en modelos de relación bidireccional o en sistemas observador-observable, donde una mala elección del tipo de puntero puede generar fugas persistentes y complicadas de diagnosticar.

El uso adecuado de `make_shared` resulta igualmente esencial. Esta función garantiza una construcción eficiente del objeto y del bloque de control asociado, reduciendo la fragmentación de memoria y minimizando la posibilidad de inconsistencias al inicializar múltiples componentes relacionados. Además, contribuye a mejorar el rendimiento al evitar múltiples asignaciones de memoria. Al mismo tiempo, debe evitarse la conversión manual de punteros crudos hacia `shared_ptr` sin un análisis claro del contratante, ya que ello puede generar dobles liberaciones si el recurso es manejado por código externo cuyo ciclo de vida no esté coordinado con el del puntero inteligente.

También debe considerarse que `shared_ptr` implica sincronización interna, lo que introduce un coste adicional cuando se emplea en sistemas multihilo de alto rendimiento. Este factor requiere evaluar cuidadosamente si la seguridad y simplicidad que ofrece el modelo de propiedad compartida compensan dicha penalización. En casos donde no se necesita transferir ni compartir propiedad, el uso de `unique_ptr` o referencias resulta más adecuado, promoviendo un diseño más simple y predecible.

Finalmente, al integrar `shared_ptr` con bibliotecas heredadas o APIs que aplican estrategias propias de administración de memoria, se debe definir un deleter personalizado que asegure coherencia en el ciclo de vida del recurso. Ignorar esta práctica puede provocar dobles liberaciones o pérdidas de control sobre el recurso, comprometiendo la estabilidad e integridad de la aplicación.

Riesgos asociados

- Fugas de memoria por falta de control automático del ciclo de vida de los objetos.
- Acceso a memoria liberada (dangling pointer) si varios punteros crudos gestionan un mismo recurso y alguno libera la memoria prematuramente.
- Duplicación accidental de liberación de memoria (double free), causando errores de ejecución o fallos de seguridad.
- Dificultad en el mantenimiento y comprensión del código debido a la gestión manual y dispersa de punteros crudos.
- Posibilidad de condiciones de carrera en entornos multihilo al modificar el mismo

objeto desde múltiples ubicaciones sin la seguridad de los contadores atómicos de `shared_ptr`.

- Riesgo de corrupción de memoria por desreferenciar punteros colgantes o no inicializados.
- Mayor probabilidad de introducir vulnerabilidades explotables (por ejemplo, ejecución de código arbitrario) mediante manipulación insegura de la memoria.

Buenas prácticas

- Utilizar `std::shared_ptr` cuando múltiples objetos necesiten compartir la propiedad de un recurso dinámico.

```
//-- Uso de puntero crudo compartido sin gestión clara
class Recurso {};
Recurso* crearRecurso() {
  return new Recurso();
}
void usarRecurso(Recurso* r) {
  // uso del recurso
}
```

```
//-- Uso de shared_ptr para compartir propiedad automáticamente
class Recurso {};
std::shared_ptr crearRecurso() {
  return std::make_shared();
}
void usarRecurso(std::shared_ptr r) {
  // uso del recurso
}
```

- Crear instancias de `shared_ptr` mediante `std::make_shared` para evitar fugas y mejorar eficiencia.

```
//-- Creación manual que puede provocar fugas si hay excepciones
std::shared_ptr p(new int(10));
```

```
//-- Creación segura y eficiente con make_shared
auto p = std::make_shared(10);
```

- Evitar ciclos de referencia entre `shared_ptr` utilizando `std::weak_ptr` cuando no se requiera propiedad fuerte.

```
//-- Referencias compartidas mutuas que impiden liberar memoria
class B;
class A {
```

```
  public:
  std::shared_ptr b;
};
class B {
  public:
  std::shared_ptr a;
};
```

```
//-- Uso de weak_ptr para romper el ciclo de propiedad
class B;
class A {
  public:
  std::shared_ptr b;
};
class B {
  public:
  std::weak_ptr a;
};
```

2.11.2.2 Estándares metodológicos

- **C++ Core-F.27**: Use a shared_ptr<T> to share ownership.
- **C++ Core-R.20**: Use unique_ptr or shared_ptr to represent ownership.

2.11.3 [cpp.ptr.3] - Usar "weak_ptr" para romper los ciclos de "shared_ptr"

ID	cpp.ptr.3	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Disponibilidad	S-SDLC	Build

Establece la necesidad de emplear **weak_ptr** cuando dos o más objetos gestionados por **shared_ptr** pueden generar ciclos de referencia que impidan la liberación automática de memoria. La prevención de estos ciclos es crítica tanto para la correcta gestión de recursos como para evitar estados inconsistentes o condiciones de bloqueo en componentes donde la disponibilidad y la estabilidad del sistema son esenciales. Utilizando **weak_ptr** como referencia no poseedora, se garantiza que los objetos puedan liberarse correctamente sin fugas ni retención accidental de recursos.

Codificación insegura

Gestión de memoria compartida entre dos clases que se referencian mutuamente usando `shared_ptr`. Se busca mostrar cómo un ciclo de referencias impide que los destructores se ejecuten y provoca fugas de memoria, incluso cuando aparentemente ya no existen referencias externas.

```
#include <iostream>
#include <memory>

class B;
class A {
public:
  std::shared_ptr<B> b_ptr;

  ~A() {
    std::cout << "Destructor de A" << std::endl;
  }
};

class B {
public:
  std::shared_ptr<A> a_ptr;

  ~B() {
    std::cout << "Destructor de B" << std::endl;
  }
};

int main() {
  std::shared_ptr<A> a = std::make_shared<A>();
  std::shared_ptr<B> b = std::make_shared<B>();
  a->b_ptr = b;
  b->a_ptr = a;
  return 0;
}
```

Codificación recomendada

Gestión de memoria compartida entre dos clases que se referencian mutuamente utilizando `weak_ptr` para romper el ciclo de referencias. Se busca mostrar cómo evitar fugas de memoria cuando existe una relación bidireccional entre objetos gestionados por `shared_ptr`.

```
#include <iostream>
#include <memory>

class B;
class A {
public:
  std::shared_ptr<B> b_ptr;

  ~A() {
    std::cout << "Destructor de A" << std::endl;
  }
```

```
};

class B {
public:
  std::weak_ptr<A> a_ptr;

  ~B() {
    std::cout << "Destructor de B" << std::endl;
  }

  void useA() {
    if (auto sharedA = a_ptr.lock()) {
      std::cout << "A sigue vivo" << std::endl;
    } else {
      std::cout << "A ya fue liberado" << std::endl;
    }
  }
};

int main() {
  std::shared_ptr<A> a = std::make_shared<A>();
  std::shared_ptr<B> b = std::make_shared<B>();
  a->b_ptr = b;
  b->a_ptr = a;
  b->useA();
  return 0;
}
```

2.11.3.1 Descripción

El uso de `shared_ptr` para la gestión automática de memoria es una práctica habitual destinada para asegurar una correcta liberación de recursos sin requerir gestión manual. Sin embargo, cuando dos objetos poseen referencias mutuas mediante `shared_ptr`, se forma un ciclo de referencia que impide la liberación automática, provocando fugas de memoria y potencial degradación del rendimiento o interrupciones de servicio. La forma recomendada de romper estos ciclos es reemplazar en uno de los extremos la referencia de propiedad por un `weak_ptr`, que actúa como referencia sin incrementar el contador de posesión.

Un ciclo de referencia ocurre cuando dos o más objetos se referencian entre sí usando `shared_ptr`, creando una dependencia circular que impide el decremento del contador de referencias a cero. Mientras dicho contador no llegue a cero, el objeto no se libera, lo cual implica que queda en memoria de manera indefinida. Esto no sólo provoca pérdidas de memoria, sino que puede provocar un aumento progresivo del consumo de recursos, afectando la disponibilidad general de la aplicación y pudiendo inducir fallos inesperados en sistemas de larga ejecución.

Un `weak_ptr` permite referenciar un objeto gestionado por `shared_ptr` sin participar en su ciclo de vida. De esta forma, cuando el último `shared_ptr` propietario desaparece, el objeto es liberado incluso aunque existan referencias `weak_ptr`. Para utilizar un `weak_ptr` de forma segura, debe convertirse en `shared_ptr` mediante el método `lock()` antes de su uso, ya que el recurso puede haber sido liberado. Esta conversión genera un `shared_ptr` temporal que garantiza su validez mientras esté en uso.

Es importante abordar la conversión desde `weak_ptr` a `shared_ptr` mediante `lock()` únicamente cuando sea necesario acceder al recurso. Antes de usar el puntero resultante, se debe comprobar si la conversión ha tenido éxito, ya que el objeto puede haber sido destruido. Esta validación evita accesos inválidos y protege la integridad del programa.

En sistemas críticos y de larga duración, incluso pequeñas fugas pueden acumularse hasta agotar la memoria disponible. La prevención de ciclos mediante `weak_ptr` contribuye directamente a la **disponibilidad** del sistema al eliminar fugas silenciosas que, de otra manera, podrían causar fallos progresivos o paradas inesperadas. Además, una gestión de recursos correcta evita comportamientos erráticos relacionados con la vida útil de los objetos.

Riesgos asociados

- Fugas de memoria (memory leaks) debido a referencias circulares entre `shared_ptr` que nunca liberan los recursos.
- Consumo excesivo de memoria por acumulación de objetos no destruidos.
- Pérdida de eficiencia y degradación del rendimiento de la aplicación.
- Comportamiento indefinido al intentar acceder a objetos ya eliminados indirectamente.
- Dificultad en el rastreo y depuración de problemas de gestión de memoria.
- Desbordamiento de memoria (out-of-memory) en aplicaciones de larga duración.
- Imposibilidad de predecir el ciclo de vida de los objetos, introduciendo bugs sutiles.
- Vulnerabilidad frente a ataques por denegación de servicio (DoS) si un atacante fuerza la creación de ciclos de referencias.

Buenas prácticas

- Utilizar `std::weak_ptr` en una de las direcciones cuando dos objetos se referencian mutuamente mediante `std::shared_ptr`.

```
//-- Referencias compartidas mutuas que impiden liberar memoria
```

```
class B;
class A {
  public:
  std::shared_ptr<B> b;
};
class B {
  public:
  std::shared_ptr<A> a;
};
```

```
//-- Uso de weak_ptr para evitar el ciclo de referencias
class B;
class A {
  public:
  std::shared_ptr<B> b;
};
class B {
  public:
  std::weak_ptr<A> a;
};
```

- Acceder al objeto gestionado por `std::weak_ptr` mediante `lock()` y validar su vigencia antes de usarlo.

```
//-- Acceso directo sin comprobar expiración
void usar(std::weak_ptr<A> wp) {
  std::shared_ptr<A> sp = wp.lock();
  sp->hacerAlgo();
}
```

```
//-- Verificación de validez antes de acceder al recurso
void usar(std::weak_ptr<A> wp) {
  if (auto sp = wp.lock()) {
    sp->hacerAlgo();
  }
}
```

- Evitar almacenar `shared_ptr` en observadores o estructuras auxiliares cuando no se requiere propiedad del objeto.

```
//-- Observador mantiene propiedad innecesaria del sujeto
class Observador {
  public:
  std::shared_ptr<A> sujeto;
};
```

```
//-- Observador mantiene referencia débil sin extender el ciclo de vida
class Observador {
  public:
  std::weak_ptr<A> sujeto;
};
```

- Comprobar `expired()` cuando se requiera validar explícitamente el estado de un `weak_ptr` antes de intentar su promoción.

```
//-- Promoción sin validar estado previo
if (wp.lock()) {
  wp.lock()->hacerAlgo();
}
```

```
//-- Validación explícita del estado antes de promover
if (!wp.expired()) {
  if (auto sp = wp.lock()) {
    sp->hacerAlgo();
  }
}
```

2.11.3.2 Estándares metodológicos

- **C++ Core-R.24**: Use std::weak_ptr to break cycles of shared_ptrs.

2.11.4 [cpp.ptr.4] - Evitar almacenar un puntero con propietario en un puntero inteligente no relacionado

ID	**cpp.ptr.4**	**Uso**	**Obligatorio**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Este requisito busca prevenir fallos de memoria e inconsistencias en la gestión de recursos cuando un puntero con semántica de propiedad es almacenado dentro de un puntero inteligente que no fue diseñado para asumir la propiedad. Al mezclar modelos de propiedad se generan riesgos como dobles liberaciones, fugas de memoria, referencias colgantes y violaciones de integridad. La aplicación adecuada de este control obliga a utilizar punteros inteligentes únicamente en contextos coherentes con su semántica de propiedad y ciclo de vida, asegurando un manejo seguro, explícito y consistente de los recursos.

Codificación insegura

Gestión incorrecta de la propiedad de memoria al mezclar punteros inteligentes no relacionados, provocando doble liberación de memoria. Se muestra cómo un objeto

creado dinámicamente termina siendo gestionado por dos propietarios distintos, generando un comportamiento indefinido que puede terminar en fallo en tiempo de ejecución.

```
#include <iostream>
#include <memory>

class Recurso {
public:
  Recurso() {
    //-- Constructor
    std::cout << "Recurso adquirido" << std::endl;
  }
  ~Recurso() {
    //-- Destructor
    std::cout << "Recurso liberado" << std::endl;
  }
};

int main() {
  //-- Se crea un unique_ptr como propietario legítimo
  std::unique_ptr<Recurso> propietario = std::make_unique<Recurso>();

  //-- Se obtiene el puntero crudo gestionado por unique_ptr
  Recurso* punteroCrudo = propietario.get();

  //-- Se crea un shared_ptr independiente a partir del mismo puntero crudo
  //-- Ahora existen dos propietarios distintos sobre el mismo recurso
  std::shared_ptr<Recurso> propietarioCompartido(punteroCrudo);

  //-- Ambos intentarán liberar el mismo recurso al finalizar
  return 0;
}
```

Codificación recomendada

Transferencia correcta de propiedad entre punteros inteligentes utilizando los mecanismos previstos por el estándar. Se evita duplicar la responsabilidad sobre el mismo recurso y se garantiza que la memoria se libera exactamente una vez.

```
#include <iostream>
#include <memory>

class Recurso {
public:
  Recurso() {
    //-- Constructor
    std::cout << "Recurso adquirido" << std::endl;
```

```
  }
  ~Recurso() {
    //-- Destructor
    std::cout << "Recurso liberado" << std::endl;
  }
};

int main() {
  //-- Se crea un unique_ptr como propietario inicial
  std::unique_ptr<Recurso> propietarioUnico = std::make_unique<Recurso>();

  //-- Se transfiere la propiedad al shared_ptr usando move
  std::shared_ptr<Recurso> propietarioCompartido = std::move(propietarioUnico);

  //-- unique_ptr ya no es propietario
  if (!propietarioUnico) {
    //-- Confirmación de que la propiedad fue transferida
    std::cout << "Propiedad transferida correctamente" << std::endl;
  }
  //-- El recurso será liberado una sola vez automáticamente
  return 0;
}
```

2.11.4.1 Descripción

El uso de punteros inteligentes proporciona una herramienta poderosa para gestionar automáticamente la vida útil de los recursos. Sin embargo, su uso indebido puede causar problemas graves de integridad y estabilidad del sistema. Uno de los errores más frecuentes y peligrosos es almacenar un puntero con propietario, como `unique_ptr`, dentro de otro puntero inteligente que no respeta esa propiedad o que no está diseñado para compartirlas, rompiendo así el modelo de propiedad que el desarrollador pretende imponer. Este requisito asegura que cada objeto tenga un único responsable claro de su destrucción, evitando inconsistencias en la gestión de memoria.

Un `unique_ptr` garantiza que únicamente una entidad tenga propiedad exclusiva del objeto administrado. Si dicho puntero es almacenado accidentalmente dentro de otra estructura inteligente, como `shared_ptr` o un contenedor que copie valores en lugar de moverlos, se pierde el control sobre su ciclo de vida. Esto puede provocar dobles liberaciones si la memoria es liberada dos veces o fugas si el puntero se copia inadvertidamente sin liberar su contenido.

Cada tipo de puntero inteligente sigue reglas y contratos muy específicos: `unique_ptr` representa propiedad exclusiva, `shared_ptr` representa propiedad compartida mediante conteo de referencias, y `weak_ptr` sirve como referencia no propietaria. Estas reglas existen para evitar errores difíciles de detectar en tiempo de ejecución, pero la mezcla indebida de estas semánticas provoca que los mecanismos automáticos de liberación actúen en momentos inesperados o bajo condiciones incorrectas. **Usar un**

tipo de puntero inteligente para un fin distinto del que fue diseñado constituye un fallo de diseño crítico de seguridad.

El enfoque correcto consiste en respetar siempre el contrato del puntero inteligente que se utiliza. Cuando se necesita un puntero no propietario, deben emplearse punteros sin semántica de propiedad explícita: `A*`, `weak_ptr<A>` o referencias. Cuando se desea compartir recursos, se debe promover un `shared_ptr` desde el inicio, sin intentar adaptar un `unique_ptr` de forma forzada. Además, para almacenar punteros propietarios en estructuras contenedoras se debe garantizar que estas usen semántica de movimiento y no de copia.

Las violaciones del modelo de propiedad suelen derivar en comportamientos indefinidos, los cuales constituyen riesgos críticos desde la perspectiva de la seguridad. Un recurso liberado dos veces puede comprometer la estabilidad del proceso, permitir corrupción de memoria o abrir vías para ataques de ejecución arbitraria. Además, referencias colgantes generadas por una liberación prematura pueden ser explotadas por un atacante si se ejecuta código posterior que accede a memoria inválida. Por estos motivos, mantener la coherencia del modelo de propiedad se considera una medida esencial para garantizar la integridad del sistema.

Riesgos asociados

- Fugas de memoria (memory leaks) al perder el control claro sobre la propiedad y liberación de la memoria asignada.
- Double free o corrupción de memoria al liberar múltiples veces el mismo recurso desde ubicaciones diferentes.
- Condiciones de carrera por posesión ambigua del recurso compartido.
- Referencias colgantes (dangling pointers) al liberar la memoria desde un smart pointer no relacionado mientras otros punteros aún la referencian.
- Comportamiento indefinido debido a la falta de claridad sobre quién es responsable de liberar el recurso.
- Violaciones del principio RAII, lo que puede llevar a dificultades en la gestión y liberación automática de recursos.
- Dificultad en la depuración y mantenimiento del código por la gestión inconsistente de la propiedad del recurso.

Buenas prácticas

- Utilizar el mismo tipo de puntero inteligente para transferir o compartir propiedad, evitando asignar punteros propietarios a punteros inteligentes incompatibles.

```
//-- Asignar puntero propietario a shared_ptr no relacionado
```

```
std::unique_ptr up(new int(10));
std::shared_ptr sp(up.get());
```

```
//-- Transferir propiedad correctamente con move
std::unique_ptr up2(new int(20));
std::shared_ptr sp2 = std::move(up2);
```

- No crear múltiples punteros inteligentes independientes a partir del mismo puntero crudo propietario.

```
//-- Crear dos shared_ptr desde el mismo puntero crudo
int* raw = new int(30);
std::shared_ptr sp1(raw);
std::shared_ptr sp2(raw);
```

```
//-- Crear un shared_ptr y copiarlo para compartir propiedad
std::shared_ptr sp3(new int(40));
std::shared_ptr sp4 = sp3;
```

- Evitar extraer el puntero crudo con `get()` para ceder propiedad a otro puntero inteligente.

```
//-- Transferir propiedad usando get() hacia otro unique_ptr
std::unique_ptr up3(new int(50));
std::unique_ptr up4(up3.get());
```

```
//-- Transferir propiedad usando move
std::unique_ptr up5(new int(60));
std::unique_ptr up6 = std::move(up5);
```

2.11.4.2 Estándares metodológicos

- **CERT/CPP-MEM56**: Do not store an already-owned pointer value in an unrelated smart pointer.

2.11.5 [cpp.ptr.5] - Usar punteros y referencias crudos solo para accesos no propietarios

ID	cpp.ptr.5	Uso	Obligatorio
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Es esencial evitar que el código asuma la propiedad de memoria mediante punteros crudos o referencias directas, ya que esto incrementa la probabilidad de fugas de memoria, condiciones de carrera, dobles liberaciones y usos después de liberar. El objetivo es promover el uso de mecanismos modernos de gestión segura de recursos, como `std::unique_ptr` y `std::shared_ptr`, asegurando una administración explícita, verificable y resistente a fallos. Aplicar este requisito reduce la complejidad, mejora la trazabilidad de la propiedad y mitiga vulnerabilidades críticas asociadas a una gestión insegura de memoria.

Codificación insegura

Gestión manual de memoria usando punteros crudos para representar propiedad de un recurso dinámico. Se pretende mostrar cómo la responsabilidad de liberar memoria puede provocar fugas o errores cuando no está claramente definida.

```
#include <iostream>

class Usuario {
public:
  Usuario(const std::string& nombre) : nombre_(nombre) {}
  void saludar() const {
    std::cout << "Hola, soy " << nombre_ << std::endl;
  }
private:
  std::string nombre_;
};

int main() {
  //-- Se reserva memoria dinámicamente y se asume propiedad con un puntero crudo
  Usuario* usuario = new Usuario("Carlos");

  //-- Uso normal del objeto
  usuario->saludar();

  //-- Error: no se libera la memoria reservada (fuga de memoria)
  // delete usuario;  // Olvidado intencionalmente
  return 0;
}
```

Codificación recomendada

Gestión automática de memoria utilizando un smart pointer para representar propiedad exclusiva de un recurso dinámico. Se busca aprender cómo delegar la liberación de memoria al estándar para evitar fugas y errores de liberación.

```
#include <iostream>
#include <memory>

class Usuario {
public:
  Usuario(const std::string& nombre) : nombre_(nombre) {}
  void saludar() const {
    std::cout << "Hola, soy " << nombre_ << std::endl;
  }
private:
  std::string nombre_;
};

int main() {
  //-- Se usa std::unique_ptr para expresar propiedad exclusiva
  std::unique_ptr<Usuario> usuario = std::make_unique<Usuario>("Carlos");

  //-- Uso normal del objeto
  usuario->saludar();

  //-- La memoria se libera automáticamente al salir del alcance
  return 0;
}
```

2.11.5.1 Descripción

Históricamente, el lenguaje ha permitido el uso de punteros crudos como mecanismo principal para controlar la vida de los objetos, pero esta práctica presenta numerosos riesgos que se manifiestan en forma de vulnerabilidades severas. El uso de punteros crudos facilita la introducción involuntaria de errores como **double-free**, **use-after-free**, **dangling pointers** y **memory leaks**. Estos fallos, además de comprometer la estabilidad, permiten potencialmente la ejecución arbitraria de código o la corrupción silenciosa de datos.

Cuando un puntero crudo tiene semántica de propiedad, su significado no es evidente para otros desarrolladores ni para herramientas automáticas de análisis estático. Esto dificulta la verificación formal, la auditoría de seguridad y la detección temprana de fugas o errores de acceso. En contraste, los punteros inteligentes establecen un contrato semántico claro: **quién posee un recurso, cuánto dura su vida y en qué condiciones se transfiere su propiedad**. Esta claridad reduce significativamente la superficie de fallo e incrementa la seguridad del sistema.

El uso de punteros crudos para gestionar recursos sensibles amplifica múltiples vectores de ataque. Entre los más relevantes se encuentran los siguientes:

- **Double-free:** liberar memoria dos veces debido a propiedad no documentada o transferida implícitamente.
- **Use-after-free:** acceder a memoria que ya ha sido liberada, dando lugar a corrupción de memoria.
- **Fugas de memoria:** al no definir claramente quién debe liberar un recurso.
- **Condiciones de carrera:** en situaciones donde varios hilos manipulan recursos sin sincronización adecuada.
- **Inconsistencia de propiedad:** descoordinación entre módulos sobre quién es responsable de destruir un objeto.

La forma moderna, segura y recomendada de gestionar recursos es aplicar el principio **RAII**. Los tipos RAII aseguran la liberación de recursos automáticamente al salir del ámbito. Este mecanismo, complementado con punteros inteligentes, elimina la mayoría de los riesgos asociados a la gestión manual de memoria. Los punteros inteligentes proporcionan garantías explícitas:

- `std::unique_ptr`**:** expresa propiedad exclusiva; sólo un dueño puede gestionar el recurso.
- `std::shared_ptr`**:** permite propiedad compartida con recuento de referencias.
- `std::weak_ptr`**:** evita ciclos de referencia peligrosos.

El uso de estos constructos estandarizados evita errores lógicos y facilita la lectura y análisis del código. Esto permite a los auditores verificar fácilmente cómo se distribuye la propiedad y en qué condiciones se transfiere.

Riesgos asociados

- Fugas de memoria (memory leaks) por falta de liberación adecuada de recursos.
- Condiciones de doble liberación (double free) y comportamiento indefinido al liberar memoria más de una vez.
- Desreferenciación de punteros colgantes (dangling pointers), provocando corrupción de memoria o fallos de segmentación (segmentation faults).
- Condiciones de carrera en entornos multihilo al manejar recursos manualmente.
- Posibilidad de acceso a memoria no válida, causando vulnerabilidades de seguridad como ejecución de código arbitrario.
- Dificultad para mantener y auditar el código, incrementando el riesgo de errores sutiles y fallos de seguridad.

- Incremento de la superficie de ataque para exploits que aprovechan punteros mal gestionados.
- Problemas en la transferencia de propiedad que pueden derivar en inconsistencias en la gestión de recursos.
- Imposibilidad de garantizar la excepción de seguridad (exception safety), exponiendo el sistema a recursos mal gestionados en caso de errores.

Buenas prácticas

- Utilizar `std::unique_ptr` para representar propiedad exclusiva de recursos dinámicos.

```
//-- Propiedad manual con puntero crudo
class Recurso {
public:
  void usar();
};

class Gestor {
  Recurso* recurso;
public:
  Gestor() {
    recurso = new Recurso();
  }
  ~Gestor() {
    delete recurso;
  }
};
```

```
//-- Propiedad exclusiva con std::unique_ptr
class Gestor {
  std::unique_ptr<Recurso> recurso;
public:
  Gestor() : recurso(std::make_unique<Recurso>()) {}
};
```

- Utilizar `std::shared_ptr` cuando la propiedad del recurso es compartida explícitamente.

```
//-- Compartición manual con puntero crudo
class Nodo {
public:
  Nodo* siguiente;
};

void conectar(Nodo* a, Nodo* b) {
```

```
  a->siguiente = b;
}
```

```
//-- Propiedad compartida con std::shared_ptr
class Nodo {
public:
  std::shared_ptr<Nodo> siguiente;
};

void conectar(const std::shared_ptr<Nodo>& a, const std::shared_ptr<Nodo>& b) {
  a->siguiente = b;
}
```

- Pasar objetos por referencia o puntero sin propiedad solo cuando no se transfiere la responsabilidad del ciclo de vida.

```
//-- Transferencia implícita de propiedad con puntero crudo
void procesar(Recurso* r) {
  delete r;
}
```

```
//-- Uso de referencia sin transferencia de propiedad
void procesar(Recurso& r) {
  r.usar();
}
```

- Evitar exponer punteros crudos propietarios en interfaces públicas.

```
//-- Exposición de puntero propietario en la interfaz
class Servicio {
public:
  Recurso* crear();
};
```

```
//-- Retorno de propiedad mediante std::unique_ptr
class Servicio {
public:
  std::unique_ptr<Recurso> crear();
};
```

2.11.5.2 Estándares metodológicos

- **C++ Core-I.11**: Never transfer ownership by a raw pointer (T*) or reference (T&).
- **C++ Core-R.3**: A raw pointer (a T*) is non-owning.

- **C++ Core-R.4**: A raw reference (a T&) is non-owning.

2.11.6 [cpp.ptr.6] - Evitar transferir la propiedad mediante punteros o referencias crudas

ID	cpp.ptr.6	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La transferencia de propiedad de recursos no debe realizarse mediante punteros o referencias crudas, ya que esto dificulta el seguimiento de la vida del recurso, favorece fugas de memoria, dobles liberaciones y acceso a memoria inválida. En su lugar, deben emplearse mecanismos seguros como `std::unique_ptr` o `std::shared_ptr`, que permiten expresar explícitamente las reglas de propiedad y facilitan al compilador y a las herramientas de análisis estático detectar errores.

Codificación insegura

Gestión de memoria dinámica al transferir la propiedad de un objeto entre funciones usando punteros crudos. Se pretende mostrar cómo una transferencia implícita de responsabilidad puede provocar fugas de memoria o dobles liberaciones cuando no queda claro quién debe liberar el recurso.

```
#include <iostream>

//-- Clase simple que representa un recurso dinámico
class Recurso {
public:
  Recurso() { std::cout << "Recurso adquirido\n"; }
  ~Recurso() { std::cout << "Recurso liberado\n"; }
  void usar() { std::cout << "Usando recurso\n"; }
};

//-- Función que recibe un puntero crudo y asume la propiedad
void procesar(Recurso* r) {
  if (r) {
    r->usar();
    delete r; //-- Libera el recurso
  }
}

int main() {
  //-- Reserva dinámica del recurso
  Recurso* recurso = new Recurso();

  //-- Se pasa el puntero sin dejar claro quién es responsable
  procesar(recurso);
```

```
  //-- Error: se vuelve a liberar el mismo recurso (doble delete)
  delete recurso;
  return 0;
}
```

Codificación recomendada

Gestión segura de la propiedad de objetos dinámicos utilizando smart pointers para dejar explícita la transferencia de responsabilidad. Se busca que el compilador y la semántica de movimiento controlen la liberación automática del recurso evitando fugas y dobles liberaciones.

```
#include <iostream>
#include <memory>

//-- Clase simple que representa un recurso dinámico
class Recurso {
public:
  Recurso() { std::cout << "Recurso adquirido\n"; }
  ~Recurso() { std::cout << "Recurso liberado\n"; }
  void usar() { std::cout << "Usando recurso\n"; }
};

//-- Función que recibe la propiedad mediante std::unique_ptr
void procesar(std::unique_ptr<Recurso> r) {
  if (r) {
    r->usar();
  }
  //-- El recurso se libera automáticamente al salir del ámbito
}

int main() {
  //-- Creación segura del recurso
  std::unique_ptr<Recurso> recurso = std::make_unique<Recurso>();

  //-- Transferencia explícita de propiedad usando std::move
  procesar(std::move(recurso));

  //-- recurso ahora es nullptr y no puede liberarse dos veces
  if (!recurso) {
    std::cout << "La propiedad fue transferida correctamente\n";
  }
  return 0;
}
```

2.11.6.1 Descripción

El uso de punteros crudos para transferir propiedad de objetos es una práctica peligrosa que puede derivar en múltiples fallos de memoria, especialmente en sistemas complejos donde la gestión de recursos es crítica. La ausencia de semánticas claras de propiedad dificulta la detección temprana de errores y expone al sistema a vulnerabilidades severas, como fugas de memoria, dobles liberaciones o desreferenciaciones de memoria liberada. Estas situaciones pueden ser explotadas, dependiendo del contexto, para provocar denegaciones de servicio o incluso para alterar el flujo de ejecución de una aplicación.

El principio fundamental es que la propiedad debe estar asociada a un constructo que la represente de manera explícita y verificable, permitiendo identificar quién es responsable de liberar un recurso y garantizando que tal liberación solo se produce una vez. En C++, la forma moderna, segura y recomendada de expresar propiedad es mediante punteros inteligentes, los cuales implementan de forma automática la liberación correcta y realizan las copias, movimientos o comparticiones de manera semánticamente consistente con la intención del desarrollador. Evitar punteros crudos es esencial para prevenir inconsistencias en la gestión de memoria.

Cuando se transfiere un puntero crudo de una función a otra, no existe un mecanismo estándar que indique con claridad si la función receptora debe liberar el recurso o no. Esta ambigüedad conduce a errores difíciles de diagnosticar. En cambio, `std::unique_ptr` expresa que existe un único propietario y que la propiedad puede ser trasladada explícitamente mediante movimiento, lo que obliga a reflexionar sobre el ciclo de vida del recurso. Si varios componentes requieren acceso compartido, entonces el uso de `std::shared_ptr` puede ser apropiado, siempre que se gestione también el peligro de referencias circulares.

Además, los punteros crudos no integran mecanismos de conteo de referencias ni destructor automático, factores que incrementan los riesgos cuando el recurso representa memoria sensible, descriptores de archivo, sockets, identificadores de GPU o buffers compartidos entre hilos. En contextos de programación concurrente, el uso descuidado de punteros crudos se vuelve aún más riesgoso, pues la vida útil del objeto puede concluir en un hilo mientras otro aún accede a él, generando condiciones de carrera que podrían derivar en corrupción de memoria o ejecución de código arbitrario.

Una ventaja adicional es que los punteros inteligentes permiten aplicar estrategias de diseño robustas como RAII, que garantiza la liberación de los recursos cuando el objeto sale de su ámbito, incluso ante excepciones u otros flujos de control no convencionales. Esto incrementa la resiliencia del software y disminuye la probabilidad de introducir vulnerabilidades relacionadas con la integridad del sistema.

En escenarios donde no resulta adecuado emplear punteros inteligentes, como en ciertos contextos donde se utilizan APIs de C heredadas, se debe especificar de forma estricta mediante documentación y convenciones internas quién es el propietario del

recurso y en qué condiciones debe liberarse. Sin embargo, incluso en tales escenarios, es recomendable encapsular los recursos crudos en clases RAII para reducir el número de interacciones directas con punteros crudos y minimizar el riesgo asociado.

Para reforzar la seguridad, es importante adoptar políticas de revisión de código que revisen toda transferencia de propiedad y verifiquen consistentemente que esta se realiza usando mecanismos explícitos. También debe promoverse el uso obligatorio de punteros inteligentes en nuevas implementaciones, evitando excepciones salvo cuando exista una razón técnica clara y documentada que justifique el empleo de punteros crudos.

Riesgos asociados

- Fugas de memoria (memory leaks) por pérdida de control sobre la liberación de los recursos.
- Doble liberación de memoria (double free) si múltiples propietarios intentan liberar el mismo recurso.
- Acceso a memoria liberada (use-after-free), provocando comportamientos indefinidos y vulnerabilidades de seguridad.
- Condiciones de carrera (race conditions) en entornos multihilo, donde el recurso pueda ser liberado o modificado inesperadamente.
- Incremento en la complejidad del código y dificultad para realizar auditorías de seguridad y análisis de fugas.
- Facilidad para introducir exploits como ejecución de código arbitrario o elevación de privilegios.
- Pérdida de información sobre la vida útil de los recursos, lo que puede causar corrupción de datos.

Buenas prácticas

- Utilizar smart pointers (`std::unique_ptr` o `std::shared_ptr`) para expresar transferencia de propiedad explícitamente.

```
//-- Transferencia implícita de propiedad mediante puntero crudo
class Recurso {
};

Recurso* crearRecurso() {
  return new Recurso();
}
```

```
//-- Transferencia explícita de propiedad con std::unique_ptr
class Recurso {
```

```
};

std::unique_ptr<Recurso> crearRecurso() {
  return std::make_unique<Recurso>();
}
```

- Evitar funciones que acepten punteros crudos cuando asumen propiedad; usar `std::unique_ptr` en parámetros.

```
//-- Función que asume propiedad mediante puntero crudo
void procesar(Recurso* r) {
  delete r;
}
```

```
//-- Función que recibe propiedad de forma explícita
void procesar(std::unique_ptr<Recurso> r) {
  // uso del recurso
}
```

- No almacenar referencias o punteros crudos como miembros cuando el objeto es propietario; usar smart pointers como miembros.

```
//-- Clase que almacena puntero crudo con responsabilidad de liberación
class Gestor {
  Recurso* recurso;
public:
  Gestor(Recurso* r) : recurso(r) {}
  ~Gestor() { delete recurso; }
};
```

```
//-- Clase que almacena std::unique_ptr como propietario
class Gestor {
  std::unique_ptr<Recurso> recurso;
public:
  Gestor(std::unique_ptr<Recurso> r) : recurso(std::move(r)) {}
};
```

- Para propiedad compartida, emplear `std::shared_ptr` en lugar de copiar punteros crudos entre objetos.

```
//-- Copia de puntero crudo sin semántica clara de propiedad
Recurso* r = new Recurso();
Recurso* copia = r;
```

```
//-- Uso de std::shared_ptr para propiedad compartida controlada
```

```
auto r = std::make_shared<Recurso>();
auto copia = r;
```

2.11.6.2 Estándares metodológicos

- **C++ Core-I.11**: Never transfer ownership by a raw pointer (T*) or reference (T&).

2.11.7 [cpp.ptr.7] - Declarar explícitamente los punteros que no pueden ser nulos

ID	cpp.ptr.7	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se deben declarar de forma explícita aquellos punteros cuyo valor no debe ser nulo durante su ciclo de vida. Al hacerlo, se refuerza la verificación de integridad, se evitan desreferencias inseguras y se permite a las herramientas de análisis estático y al compilador aplicar comprobaciones estrictas. Además, esta práctica facilita el razonamiento del código, ayuda a detectar defectos lógicos y reduce la superficie de ataque frente a fallos de memoria explotables.

Codificación insegura

Gestión de un puntero que se espera válido para imprimir un nombre. Se muestra cómo el uso de punteros que pueden ser nulos sin validación puede provocar fallos en tiempo de ejecución y comportamientos indefinidos al desreferenciarlos.

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

void imprimirNombre(std::string* nombre) {
  //-- Se asume que el puntero es válido, pero no se valida
  std::cout << "Nombre: " << *nombre << std::endl;
}

int main() {
  //-- Se declara un puntero sin inicializar correctamente
  std::string* nombre = nullptr;

  //-- Se llama a la función con un puntero nulo
  imprimirNombre(nombre);
  return 0;
}
```

Codificación recomendada

Gestión de un dato que debe existir obligatoriamente usando una referencia en lugar de un puntero. Se fuerza en tiempo de compilación que el valor no sea nulo, eliminando la posibilidad de pasar un puntero inválido y reduciendo errores en tiempo de ejecución.

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

void imprimirNombre(const std::string& nombre) {
  //-- La referencia garantiza que el objeto existe y no es nulo
  std::cout << "Nombre: " << nombre << std::endl;
}

int main() {
  //-- Se crea un objeto válido
  std::string nombre = "Carlos";

  //-- Se pasa por referencia garantizando no nulidad
  imprimirNombre(nombre);
  return 0;
}
```

2.11.7.1 Descripción

La manipulación de punteros constituye uno de los aspectos más delicados en términos de seguridad y fiabilidad. La ausencia de garantías sobre la nulidad o no nulidad de un puntero puede provocar accesos ilegales a memoria, corrupción de datos, errores de ejecución e incluso vulnerabilidades explotables como **desreferencias de punteros nulos**, **use-after-null o inconsistencias de estado**. Para mitigar estos riesgos, es esencial que aquellos punteros cuyo valor nunca debe ser nulo sean declarados y tratados explícitamente como tales, permitiendo a los desarrolladores y al compilador aplicar reglas estrictas de integridad.

La intención principal de este requisito es lograr que el código exprese con claridad sus invariantes. Si un puntero no debe ser nulo, esa condición debe quedar reflejada tanto en su tipo como en su lugar de inicialización, evitando configuraciones ambiguas o revisiones manuales innecesarias. En C++, esta práctica se refuerza mediante el uso de anotaciones, envoltorios inteligentes (`smart pointers`) y herramientas de análisis estático, capaces de comprobar automáticamente la validez de los punteros. Una declaración explícita reduce el riesgo de errores lógicos y permite al equipo comprender mejor las dependencias entre objetos.

Una de las técnicas más extendidas consiste en utilizar **constructores que requieren parámetros no nulos**, evitando que un objeto sea creado en un estado inconsistente. El propio lenguaje, unido a los mecanismos de RAII, contribuye entonces a mantener los datos en un estado válido durante todo su ciclo de vida. Además, la adopción de

punteros inteligentes como `std::unique_ptr` o `std::shared_ptr` simplifica la gestión de memoria, aunque por sí solos no garantizan la no nulidad. Para este propósito, se pueden usar tipos envoltorio diseñados específicamente para garantizar la no nulidad mediante comprobaciones en tiempo de compilación o en tiempo de ejecución.

Otra línea importante dentro de este requisito consiste en describir con precisión, mediante documentación interna o anotaciones semánticas, las expectativas sobre la validez de un puntero. Herramientas como **GSL (Guidelines Support Library)** proporcionan el tipo `not_null`, que encapsula un puntero y asegura que nunca pueda albergar un valor nulo. Esta práctica es especialmente útil en entornos industriales donde el código debe estar sujeto a verificaciones rigurosas, auditorías de seguridad o certificaciones normativas.

También es esencial considerar los casos en los que una API acepta punteros como parámetros. Si el parámetro no debe ser nulo, esta regla exige que se refleje en la firma de la función. Esto facilita a los usuarios de la API comprender sus responsabilidades y evita ambigüedades que podrían derivar en errores difíciles de rastrear. Además, habilita al compilador y a las herramientas de análisis a emitir advertencias cuando se detecta un posible flujo que podría generar un puntero nulo.

Cuando no se dispone de la GSL, es posible implementar mecanismos equivalentes mediante comprobaciones explícitas en el constructor o mediante funciones auxiliares que verifiquen la validez antes de proceder. Aunque esta estrategia agrega cierta sobrecarga, continúa siendo una medida eficaz para evitar desreferencias inseguras y garantizar la consistencia del sistema. En entornos críticos como automoción, energía, defensa o dispositivos médicos, estas validaciones pueden ser obligatorias y deben realizarse de forma sistemática.

De forma complementaria, es aconsejable acompañar este tipo de restricciones con revisiones de código automatizadas y manuales. Las herramientas de análisis estático pueden detectar rutas de ejecución que puedan llevar a estados no válidos, mientras que las pruebas unitarias y las validaciones en tiempo de ejecución refuerzan la estabilidad del sistema. Además, documentar claramente la no nulidad mejora la mantenibilidad, ya que otros desarrolladores pueden comprender rápidamente las garantías que ofrece el código y actuar en consecuencia.

Otro aspecto relevante consiste en minimizar la cantidad de punteros permitiendo, cuando sea posible, trabajar con referencias. Las referencias en C++ no pueden ser nulas por definición, y su uso comunica de manera más clara la intención de los desarrolladores. Sin embargo, deben utilizarse con responsabilidad, evitando referencias colgantes o aliasing excesivo que pueda dificultar el análisis del flujo de datos. Cuando una referencia no satisface los requisitos del diseño (por ejemplo, en estructuras opcionales), los punteros no nulos proveen una alternativa segura.

A nivel de arquitectura, este requisito fomenta diseños más robustos y modulares. Asegurar la no nulidad de los punteros reduce el acoplamiento incierto entre

componentes y mejora la predictibilidad del comportamiento de los sistemas. Al no depender de valores nulos como señales de error, los flujos de control se simplifican, permitiendo la introducción de mecanismos más seguros como tipos de retorno estructurados, excepciones o valores opcionales con semánticas bien definidas. En sistemas seguros, esta claridad es esencial para mitigar vulnerabilidades asociadas a la corrupción de memoria.

Es importante recordar que los punteros nulos pueden ser usados legítimamente en algunos casos, pero la regla exige que su uso sea explícito, consciente y documentado. Si un puntero puede ser nulo, debe usarse una representación adecuada como `std::optional`, punteros inteligentes configurados para expresar ausencia de valor, o verificaciones exhaustivas antes de cualquier acceso. Lo que este requisito pretende evitar es el uso implícito, no controlado o ambiguo de punteros que deberían considerarse invariantes.

Riesgos asociados

- Desreferenciación de punteros nulos, provocando fallos en tiempo de ejecución o vulnerabilidades de denegación de servicio.
- Dificultad para identificar y prevenir errores relacionados con la gestión de memoria, aumentando la probabilidad de fallos de estabilidad.
- Mayor superficie para la explotación de ataques como el Null Pointer Dereference (CWE-476), que pueden derivar en ejecución de código arbitrario bajo ciertas condiciones.
- Complicaciones al realizar revisiones de código y análisis estáticos, ya que no es posible inferir fácilmente la validez de los punteros.
- Pérdida de claridad y confiabilidad en las interfaces y funciones, causando malentendidos en el equipo de desarrollo y posibles usos indebidos de las APIs.
- Reducción de la robustez y mantenibilidad del código, dificultando la implementación segura de nuevas funcionalidades.

Buenas prácticas

- Utilizar referencias en lugar de punteros cuando el valor no puede ser nulo.

```
//-- Uso de puntero que puede ser nulo
void procesar(Usuario* u) {
  u->actualizar();
}
```

```
//-- Uso de referencia que garantiza objeto válido
void procesar(Usuario& u) {
  u.actualizar();
```

```
}
```

- Emplear tipos envoltorio como `std::reference_wrapper` cuando se requiera semántica de objeto no nulo en contenedores.

```
//-- Vector de punteros que podrían ser nulos
std::vector usuarios;
usuarios.push_back(nullptr);
```

```
//-- Vector de referencias que no admiten nulos
std::vector> usuarios;
usuarios.push_back(usuarioValido);
```

- Usar anotaciones o tipos específicos como `gsl::not_null` para expresar explícitamente la no nulidad.

```
//-- Puntero sin restricción explícita de nulidad
void enviar(Conexion* c) {
  c->abrir();
}
```

```
//-- Uso de gsl::not_null para impedir valores nulos
void enviar(gsl::not_null<Conexion*> c) {
  c->abrir();
}
```

- Validar en el constructor que los punteros miembros obligatorios no sean nulos.

```
//-- Constructor que acepta puntero potencialmente nulo
class Servicio {
  Repositorio* repo;
public:
  Servicio(Repositorio* r) : repo(r) {}
};
```

```
//-- Constructor que valida y rechaza punteros nulos
class Servicio {
  Repositorio* repo;
public:
  Servicio(Repositorio* r) : repo(r) {
    if (r == nullptr) throw std::invalid_argument("repo nulo");
  }
};
```

- Eliminar valores por defecto nulos en parámetros que deben ser obligatorios.

```
//-- Parámetro con valor por defecto nulo
void registrar(Log* logger = nullptr) {
  logger->escribir("inicio");
}
```

```
//-- Parámetro obligatorio sin valor por defecto
void registrar(Log* logger) {
  logger->escribir("inicio");
}
```

2.11.7.2 Estándares metodológicos

- **C++ Core-F.23**: Use a not_null<T> to indicate that "null" is not a valid value.
- **C++ Core-I.12**: Declare a pointer that must not be null as not_null.

2.11.8 [cpp.ptr.8] - Evitar pasar arrays como punteros simples

ID	cpp.ptr.8	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se recomienda el uso de construcciones seguras como `std::span`, `std::array` o contenedores estándar en lugar de punteros sin información de tamaño al pasar arrays entre funciones. Se reducen riesgos de corrupción de memoria, desbordamientos y accesos fuera de límites, ya que preserva la semántica del tamaño y proporciona mayor seguridad en tiempo de compilación y en tiempo de ejecución. Al emplear estas abstracciones modernas, se mejora la calidad del código, se previenen vulnerabilidades y se favorece una programación más robusta y mantenible.

Codificación insegura

Se muestra una función que recibe un array como puntero simple y trata de calcular su tamaño usando `sizeof`. La intención es recorrer todos los elementos del array, pero se evidencia cómo al perder la información del tamaño real se genera un comportamiento incorrecto. Se pretende entender el riesgo de pasar arrays como punteros sin información adicional.

```
#include <iostream>

void imprimir(int* datos) {
  //-- Se intenta calcular el tamaño del array usando sizeof
  size_t longitud = sizeof(datos) / sizeof(datos[0]);

  //-- Se recorre el array usando la longitud calculada
```

```
  for (size_t i = 0; i < longitud; ++i) {
    std::cout << datos[i] << " ";
  }
  std::cout << std::endl;
}

int main() {
  //-- Se declara un array de 5 elementos
  int numeros[5] = {1, 2, 3, 4, 5};

  //-- Se pasa el array a la función como puntero
  imprimir(numeros);
  return 0;
}
```

Codificación recomendada

Se presenta una función que recibe el array por referencia manteniendo su tamaño como parte del tipo mediante una plantilla. La intención es recorrer todos los elementos de forma segura sin perder la información de longitud, evitando errores comunes asociados al uso de punteros simples.

```
#include <iostream>
#include <cstddef>

template <std::size_t N>
void imprimir(int (&datos)[N]) {
  //-- Se utiliza el tamaño N deducido automáticamente por la plantilla
  for (std::size_t i = 0; i < N; ++i) {
    std::cout << datos[i] << " ";
  }
  std::cout << std::endl;
}

int main() {
  //-- Se declara un array de 5 elementos
  int numeros[5] = {1, 2, 3, 4, 5};

  //-- Se pasa el array manteniendo su tamaño en el tipo
  imprimir(numeros);
  return 0;
}
```

2.11.8.1 Descripción

El manejo seguro de arrays es un aspecto fundamental para reducir vulnerabilidades relacionadas con accesos indebidos a memoria. Una práctica insegura recurrente

consiste en pasar arrays como simples punteros, lo cual elimina información crítica del tamaño real del array y deja al código dependiente de supuestos implícitos. Este enfoque es fuente de múltiples fallos de integridad, especialmente cuando la función receptora no puede verificar límites o cuando el código evoluciona y los argumentos cambian con el tiempo.

El uso de un puntero simple para representar un array implica que se pierde toda información sobre su longitud, lo cual genera múltiples problemas potenciales. Entre los riesgos más comunes se encuentran los accesos fuera de límites, los desbordamientos de buffer, la corrupción de estructuras internas de memoria y la imposibilidad de validar adecuadamente los parámetros de entrada. Además, cuando el mismo puntero se utiliza para referirse tanto al inicio de una secuencia como a un elemento arbitrario en su interior, se dificulta establecer invariante de seguridad y se incrementa la probabilidad de errores lógicos.

Este problema se agrava cuando se mezclan bibliotecas heredadas con código moderno, o cuando se realizan refactorizaciones parciales que asumen tamaños fijos que realmente no están garantizados. Incluso en contextos donde la entrada está controlada, el desconocimiento del tamaño efectivo del array conduce a errores sutiles que pueden manifestarse en condiciones específicas, lo cual complica su detección y afecta la confiabilidad del software.

A partir de C++20, una de las herramientas más recomendadas para reemplazar el paso de arrays mediante punteros simples es `std::span`. Esta abstracción representa una vista no propietaria sobre una secuencia contigua e incluye explícitamente la información de tamaño. De esta manera, facilita la verificación de límites y evita depender de convenciones implícitas que pueden cambiar entre versiones o entre equipos de desarrollo. El uso de `std::span` resulta especialmente valioso en APIs que reciben datos desde múltiples capas del software.

Otra alternativa segura consiste en utilizar `std::array` cuando el tamaño del array es fijo y conocido en tiempo de compilación. Además de conservar la semántica del tamaño, ofrece integración con algoritmos de la STL y provee chequeos de límites mediante funciones como `at()`. Su uso refuerza la seguridad al evitar errores derivados de inconsistencias en la longitud esperada.

Riesgos asociados

- Desbordamiento de buffer (buffer overflow) debido a no conocer el tamaño real del array.
- Dificultad para validar límites y tamaño, lo que puede causar accesos inválidos a memoria.
- Vulnerabilidad a ataques de ejecución de código arbitrario por corrupción de memoria.

- Comportamiento indefinido al acceder fuera de los límites del array.
- Errores de seguridad por confusión entre punteros y arrays dinámicos o estáticos.
- Propagación involuntaria de errores si la función modifica datos fuera del array válido.
- Pérdida de información de tamaño y tipo, lo que aumenta la dificultad para realizar validaciones seguras.
- Fugas de memoria (memory leaks) si no se gestiona correctamente la memoria dinámica asociada.

Buenas prácticas

- Preferir contenedores estándar como `std::array` o `std::vector` en lugar de pasar arrays como punteros simples.

```
//-- Paso de array como puntero sin información de tamaño
void procesar(int* datos) {
  for (size_t i = 0; i < 10; ++i) {
    datos[i] = i;
  }
}
```

```
//-- Uso de std::array que mantiene tamaño y semántica segura
void procesar(std::array<int, 10>& datos) {
  for (size_t i = 0; i < datos.size(); ++i) {
    datos[i] = i;
  }
}
```

- Si se requiere flexibilidad en tamaño, utilizar `std::vector` para evitar accesos fuera de límites.

```
//-- Puntero sin validación de límites ni tamaño asociado
void limpiar(int* buffer) {
  for (size_t i = 0; i < 100; ++i) {
    buffer[i] = 0;
  }
}
```

```
//-- Uso de std::vector con control explícito del tamaño
void limpiar(std::vector<int>& buffer) {
  for (size_t i = 0; i < buffer.size(); ++i) {
    buffer[i] = 0;
  }
```

```
}
```

- Cuando se necesite interoperabilidad sin copiar datos, emplear `std::span` para mantener tamaño y límites.

```
//-- Pérdida de información de tamaño al recibir puntero
void imprimir(const int* datos) {
  for (size_t i = 0; i < 5; ++i) {
    std::cout << datos[i] << " ";
  }
}
```

```
//-- Uso de std::span para asociar puntero y tamaño
void imprimir(std::span<const int> datos) {
  for (size_t i = 0; i < datos.size(); ++i) {
    std::cout << datos[i] << " ";
  }
}
```

- Si se debe usar punteros por compatibilidad, pasar explícitamente el tamaño como parámetro adicional y validarlo.

```
//-- Puntero sin verificación de tamaño asociado
void copiar(int* destino, const int* origen) {
  for (size_t i = 0; i < 10; ++i) {
    destino[i] = origen[i];
  }
}
```

```
//-- Puntero acompañado de tamaño con validación previa
void copiar(int* destino, const int* origen, size_t tam) {
  if (destino == nullptr || origen == nullptr) return;
  for (size_t i = 0; i < tam; ++i) {
    destino[i] = origen[i];
  }
}
```

2.11.8.2 Estándares metodológicos

- **C++ Core-I.13**: Do not pass an array as a single pointer.
- **C++ Core-R.14**: Avoid [] parameters, prefer span.

2.11.9 [cpp.ptr.9] - Usar "std::span" para representar secuencias de elementos

ID	cpp.ptr.9	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Esta regla recomienda representar vistas sobre secuencias de memoria mediante `std::span` en lugar de usar punteros crudos o pares puntero-longitud. El uso de `std::span` aporta una capa adicional de claridad semántica, reduce el riesgo de desbordamientos, evita inconsistencias entre tamaño y datos, y facilita la escritura de funciones más seguras y mantenibles. Esta práctica permite expresar con precisión los límites de los datos manipulados, mitigando errores comunes asociados al manejo manual de memoria.

Codificación insegura

Procesamiento de una secuencia de enteros para calcular su suma. Se muestra cómo representar y recorrer una colección de datos que proviene de un contenedor externo, destacando la importancia de mantener coherencia entre el tamaño real y los elementos accesibles.

```
#include <iostream>
#include <vector>

//-- Función que recibe puntero y tamaño por separado
int sum(const int* data, std::size_t size) {
  //-- Acumulador de la suma
  int total = 0;
  //-- Recorre usando el tamaño recibido sin validar coherencia
  for (std::size_t i = 0; i < size; ++i) {
    total += data[i];
  }
  return total;
}

int main() {
  //-- Vector con 3 elementos
  std::vector<int> values = {1, 2, 3};

  //-- Error: se pasa un tamaño incorrecto (5 en lugar de 3)
  int result = sum(values.data(), 5);

  //-- Puede provocar acceso fuera de límites (comportamiento indefinido)
  std::cout << "Resultado: " << result << std::endl;
  return 0;
}
```

Codificación recomendada

Procesamiento de una secuencia de enteros para calcular su suma utilizando una vista segura sobre los datos. Se busca representar la secuencia como una única entidad que encapsula puntero y tamaño, reduciendo el riesgo de inconsistencias y accesos fuera de límites.

```
#include <iostream>
#include <vector>
#include <span>

//-- Función que recibe la secuencia como una vista segura
int sum(std::span<const int> data) {
  //-- Acumulador de la suma
  int total = 0;
  //-- Recorre usando el tamaño gestionado por std::span
  for (std::size_t i = 0; i < data.size(); ++i) {
    total += data[i];
  }
  return total;
}

int main() {
  //-- Vector con 3 elementos
  std::vector<int> values = {1, 2, 3};

  //-- Se pasa el contenedor completo como span
  int result = sum(values);

  //-- No es posible desincronizar puntero y tamaño accidentalmente
  std::cout << "Resultado: " << result << std::endl;
  return 0;
}
```

2.11.9.1 Descripción

El uso de `std::span` contribuye significativamente a la seguridad del software, especialmente en operaciones que manejan secuencias de elementos en memoria. Esta construcción proporciona una vista no propietaria sobre un rango contiguo de memoria, lo que permite a las funciones operar con mayor seguridad al conocer explícitamente el tamaño de la región de datos. Este enfoque reduce la dependencia de punteros crudos, disminuye la probabilidad de desbordamientos, elimina errores de desincronización entre longitud y datos, y aporta un modelo conceptual más claro para el desarrollador.

Las ventajas más claras de su uso son:

- **Seguridad por diseño**: `std::span` impone conocimiento explícito del tamaño de la secuencia, reduciendo fallos de memoria comúnmente asociados al uso de

punteros.

- **Claridad semántica**: al expresar el rango de elementos, el lector del código comprende de inmediato los límites de la operación.
- **Mitigación de errores**: evita el uso de múltiples parámetros redundantes, como puntero y longitud, reduciendo el riesgo de inconsistencias.
- **Compatibilidad con estructuras contiguas**: trabaja de forma natural con `std::array`, `std::vector`, arrays estáticos y cualquier bloque de memoria contiguo.
- **Evita copias innecesarias**: ofrece una vista ligera sin necesidad de duplicar los datos originales.

Riesgos asociados

- Desbordamiento de buffer al no verificar los límites del array.
- Lectura o escritura fuera de los límites de los datos, causando vulnerabilidades de seguridad.
- Corrupción de memoria al acceder a posiciones inválidas.
- Exposición a ataques de ejecución de código arbitrario debido a accesos indebidos en memoria.
- Condiciones de carrera y errores sutiles por manejo incorrecto del tamaño de las secuencias.
- Fugas de información al leer datos fuera del rango esperado.
- Dificultad en el mantenimiento del código al usar punteros crudos o arrays sin abstracción, incrementando la probabilidad de errores de manipulación de memoria.
- Incompatibilidades y fallas inesperadas al manipular secuencias que cambian de tamaño dinámicamente.

Buenas prácticas

- Evitar el uso de punteros crudos y tamaños separados para representar secuencias; usar `std::span` para mantener la relación segura entre datos y longitud.

```
//-- Uso de puntero crudo y tamaño independiente
void procesar(int* datos, size_t tam) {
  for(size_t i = 0; i < tam; ++i) {
    datos[i] *= 2;
  }
```

```
}
```

```
//-- Uso de std::span para vincular datos y tamaño
void procesar(std::span<int> datos) {
  for(auto& v : datos) {
    v *= 2;
  }
}
```

- Utilizar `std::span` para evitar desajustes entre el tamaño real del contenedor y el tamaño pasado como argumento.

```
//-- Tamaño pasado manualmente con riesgo de inconsistencia
void imprimir(const std::vector<int>& v, size_t tam) {
  for(size_t i = 0; i < tam; ++i) {
    std::cout << v[i];
  }
}
```

```
//-- Paso directo del contenedor como std::span
void imprimir(std::span<const int> v) {
  for(const auto& e : v) {
    std::cout << e;
  }
}
```

- Preferir `std::span` para trabajar con arrays estáticos o dinámicos sin perder información de límites.

```
//-- Array decae a puntero perdiendo información de tamaño
void calcular(int* datos) {
  datos[10] = 5;
}
```

```
//-- std::span conserva el tamaño del array
void calcular(std::span<int> datos) {
  if(datos.size() > 10) {
    datos[10] = 5;
  }
}
```

- Usar `std::span<const T>` cuando la función no deba modificar los datos, reforzando la inmutabilidad.

```
//-- Puntero no constante permite modificación accidental
```

```
void mostrar(int* datos, size_t tam) {
  datos[0] = 0;
}
```

```
//-- std::span constante evita modificaciones no deseadas
void mostrar(std::span<const int> datos) {
  for(const auto& v : datos) {
    std::cout << v;
  }
}
```

2.11.9.2 Estándares metodológicos

- **C++ Core-F.24**: Use a span<T> or a span_p<T> to designate a half-open sequence.

2.11.10 [cpp.ptr.10] - Evitar devolver un puntero o referencia a un objeto automático local

ID	**cpp.ptr.10**	**Uso**	**Obligatorio**
CWSS Score	**90**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Consiste en evitar que se devuelvan **punteros** o **referencias** a objetos con almacenamiento automático, ya que dichos objetos dejan de existir al salir del ámbito donde fueron creados. Esta práctica genera accesos a memoria liberada, comportamientos indefinidos y vulnerabilidades explotables. El requisito promueve el uso de mecanismos seguros como `std::unique_ptr`, `std::shared_ptr` o la devolución por valor para garantizar integridad, estabilidad y resistencia ante ataques relacionados con corrupción de memoria.

Codificación insegura

Se muestra una función que crea una cadena local dentro de su ámbito y devuelve un puntero a esa variable. El objetivo es observar el comportamiento cuando se intenta acceder a memoria que ya no es válida tras finalizar la función, entendiendo los riesgos asociados al tiempo de vida de los objetos automáticos.

```
#include <iostream>
#include <cstring>

char* obtenerMensaje() {
  //-- Se declara un array local en la pila
  char mensaje[] = "Hola mundo";
```

```
  //-- Se devuelve un puntero a memoria que dejará de ser válida al salir de la
función
  return mensaje;
}

int main() {
  //-- Se recibe un puntero que apunta a memoria inválida
  char* ptr = obtenerMensaje();

  //-- Comportamiento indefinido al intentar usar el puntero
  std::cout << "Mensaje: " << ptr << std::endl;
  return 0;
}
```

Codificación recomendada

Se presenta una función que genera una cadena y la devuelve por valor. El propósito es mostrar una forma segura de retornar datos creados dentro de la función, evitando problemas relacionados con el tiempo de vida del objeto y eliminando el riesgo de referencias o punteros colgantes.

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

std::string obtenerMensaje() {
  //-- Se crea un objeto local que será devuelto por valor
  std::string mensaje = "Hola mundo";

  //-- La devolución por valor garantiza una copia segura del objeto
  return mensaje;
}

int main() {
  //-- Se recibe una copia válida del objeto
  std::string msg = obtenerMensaje();

  //-- Uso seguro del objeto fuera de la función
  std::cout << "Mensaje: " << msg << std::endl;
  return 0;
}
```

2.11.10.1 Descripción

Uno de los problemas de seguridad más frecuentes y peligrosos aparece cuando una función devuelve un **puntero** o una **referencia** a un objeto local automático. Los objetos locales se destruyen automáticamente al finalizar la ejecución del bloque donde fueron declarados; por tanto, cualquier puntero o referencia hacia ellos queda apuntando a

memoria inválida. Esta situación puede causar fallos críticos, corrupción de memoria, fugas de información y condiciones explotables por un atacante. Debido a ello, evitar este patrón es una obligación de seguridad en sistemas robustos.

Los riesgos derivados de esta mala práctica pueden manifestarse tanto en tiempo de ejecución como en condiciones de explotación deliberada. El acceso a memoria ya liberada puede resultar en fallos inmediatos o en comportamientos erráticos difíciles de depurar. Un atacante podría provocar escenarios donde un objeto destruido es reemplazado por datos controlados, permitiendo ejecutar código arbitrario, sobrescribir estructuras internas o manipular el flujo del programa. Por ello, este tipo de fallo está relacionado con vulnerabilidades de severidad crítica.

Para evitar este tipo de errores es recomendable emplear estrategias seguras: devolución por valor, uso de `std::unique_ptr` para transferencia clara de propiedad, uso de `std::shared_ptr` cuando se precise propiedad compartida o uso de objetos estáticos cuidadosamente controlados. Estos mecanismos mantienen una vida útil clara y no inducen accesos a memoria liberada.

En código moderno, la devolución por valor es eficiente debido a optimizaciones como RVO. Por ello, la antigua necesidad de devolver punteros a locales para reducir costes ya no es válida. Además, la claridad del código aumenta, se evita complejidad innecesaria y se garantiza la integridad del programa. También en entornos multihilo esta práctica cobra mayor relevancia, ya que los errores de memoria se vuelven más complejos y potencialmente más peligrosos en presencia de condiciones de carrera o accesos concurrentes.

En sistemas embebidos o críticos, donde C++ es ampliamente utilizado, esta recomendación es obligatoria. El incumplimiento puede generar comportamientos no deterministas que comprometen la integridad global del sistema.

Riesgos asociados

- Dereferenciación de punteros o referencias a memoria ya liberada (dangling pointer/reference), ocasionando comportamientos indefinidos.
- Posibles fallos en tiempo de ejecución como segmentation faults o crashes de la aplicación.
- Vulnerabilidades de seguridad como corrupción de memoria, leading a posibles exploits.
- Difícil depuración y diagnóstico debido a la naturaleza errática y no determinista del error.
- Puede permitir la sobrescritura de memoria sensible, afectando la integridad y confidencialidad de los datos.
- Exposición a ataques de ejecución de código arbitrario si el atacante puede influir en la memoria liberada.

Buenas prácticas

- Devolver objetos por valor en lugar de punteros o referencias a variables locales automáticas.

```
//-- devolución de referencia a variable local
int& obtenerValor() {
  int x = 10;
  return x;
}
```

```
//-- devolución por valor evitando referencia a objeto automático
int obtenerValorSeguro() {
  int x = 10;
  return x;
}
```

- Utilizar almacenamiento dinámico gestionado (smart pointers) cuando sea necesario extender el tiempo de vida del objeto.

```
//-- devolución de puntero a variable local
int* crearObjeto() {
  int x = 20;
  return &x;
}
```

```
//-- uso de std::unique_ptr para gestionar el tiempo de vida
std::unique_ptr<int> crearObjetoSeguro() {
  return std::make_unique<int>(20);
}
```

- Pasar el objeto por parámetro de salida si se requiere modificar datos sin comprometer el tiempo de vida.

```
//-- devolución de referencia a objeto local
std::string& obtenerCadena() {
  std::string texto = "hola";
  return texto;
}
```

```
//-- uso de parámetro de salida para evitar retorno de referencia local
void obtenerCadenaSegura(std::string& resultado) {
  resultado = "hola";
}
```

2.11.10.2 Estándares metodológicos

- **C++ Core-F.43**: Never (directly or indirectly) return a pointer or a reference to a local object.

2.11.11 [cpp.ptr.11] - Evitar asignar la dirección de un objeto automático a uno con mayor duración

ID	cpp.ptr.11	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Este control de seguridad busca prevenir errores graves evitando que punteros o referencias almacenados en variables de mayor duración apunten a objetos definidos con almacenamiento automático y, por tanto, con existencia limitada al ámbito local. Esta mala práctica puede generar accesos a memoria liberada, resultados indefinidos, corrupción de datos, vulnerabilidades explotables y condiciones de carrera en casos multihilo. El requisito promueve la gestión segura del ciclo de vida de objetos usando constructores, RAII, punteros inteligentes y patrones seguros de diseño para asegurar coherencia y robustez del software.

Codificación insegura

Asignación de la dirección de una variable local (almacenada en la pila) a un puntero global. Se busca mostrar qué ocurre cuando un objeto automático sale de su ámbito y su dirección sigue siendo utilizada, provocando comportamiento indefinido por acceso a memoria inválida.

```
#include <iostream>

//-- Puntero global con mayor duración que las variables locales
int* g_ptr = nullptr;

void asignarDireccion() {
  //-- Variable automática (duración limitada al bloque)
  int valor = 42;

  //-- Se asigna su dirección a un puntero global
  g_ptr = &valor;

  //-- Se muestra el valor dentro del ámbito válido
  std::cout << "Dentro de la funcion: " << *g_ptr << std::endl;
}

int main() {
  asignarDireccion();
```

```
  //-- Acceso a memoria inválida (la variable ya no existe)
  if (g_ptr != nullptr) {
    std::cout << "Fuera de la funcion: " << *g_ptr << std::endl;
  }
  return 0;
}
```

Codificación recomendada

Uso de almacenamiento dinámico para garantizar que el objeto tenga una duración adecuada al ser referenciado por un puntero global. Se pretende asegurar que la memoria siga siendo válida hasta que se libere explícitamente, evitando referencias a objetos que ya salieron de su ámbito.

```
#include <iostream>

//-- Puntero global
int* g_ptr = nullptr;

void asignarDireccion() {
  //-- Reserva dinámica para extender la duración del objeto
  g_ptr = new int(42);

  //-- Uso seguro dentro de la función
  std::cout << "Dentro de la funcion: " << *g_ptr << std::endl;
}

int main() {
  asignarDireccion();

  //-- Uso válido mientras no se libere la memoria
  if (g_ptr != nullptr) {
    std::cout << "Fuera de la funcion: " << *g_ptr << std::endl;
  }
  //-- Liberación explícita para evitar fuga de memoria
  delete g_ptr;
  g_ptr = nullptr;
  return 0;
}
```

2.11.11.1 Descripción

Los objetos con almacenamiento automático se crean y destruyen de manera implícita al entrar y salir de un bloque de código. Cuando la dirección de uno de estos objetos se asigna a otra entidad con mayor duración, como una variable estática, global o un objeto dinámico, se genera una discrepancia peligrosa entre el tiempo de vida del puntero y el de la memoria a la que apunta. Este tipo de error, conocido como "dangling pointer",

produce comportamientos indefinidos que pueden manifestarse como fallos esporádicos, escritura fuera de límites, fugas de información sensible o corrupción de estructuras internas del programa. Por ello, resulta esencial aplicar mecanismos que garanticen que la vida útil de los objetos sea coherente con la vida útil de los punteros o referencias que los señalan.

Los errores que surgen al exponer direcciones de objetos automáticos fuera de su ámbito pueden ser especialmente difíciles de detectar porque no siempre causan fallos inmediatos. En muchos casos, el programa parece funcionar correctamente hasta que circunstancias muy específicas provocan una lectura o escritura inválida. Estos fallos intermitentes hacen que el diagnóstico resulte extremadamente complejo. Una explotación maliciosa puede consistir en forzar un patrón de ejecución que provoque reutilización de pila, alteración de datos internos o desvío del flujo de control. La naturaleza no determinista de estos problemas subraya la necesidad de evitar su origen mediante prácticas seguras de gestión de memoria.

El lenguaje moderno proporciona recursos nativos que facilitan gestionar adecuadamente el tiempo de vida de los objetos. Entre ellos destacan los punteros inteligentes `std::unique_ptr`, `std::shared_ptr` y `std::weak_ptr`, que permiten modelar la propiedad de un objeto de manera precisa. Asimismo, las referencias devueltas desde funciones deben garantizar que el objeto referenciado tenga un tiempo de vida suficiente para su uso previsto, evitando devolver referencias a variables locales. RAII es un patrón fundamental que asegura que los recursos asociados a un objeto se mantienen válidos mientras la instancia exista y se liberan correctamente al destruirse.

En contextos multihilo, el uso incorrecto de direcciones de objetos automáticos puede derivar no solo en accesos inválidos, sino también en condiciones de carrera donde varios hilos intentan acceder a memoria ya liberada o que ha sido sobrescrita. El modelo de memoria de C++ exige que cualquier dato compartido entre hilos tenga una vida útil claramente definida y protegida mediante mecanismos de sincronización. Un puntero que apunta a un objeto automático cuya vida terminó puede causar fallos que dependen del orden de ejecución, lo que dificulta su análisis. Por ello, resulta imprescindible asegurar que ningún hilo almacene o reutilice direcciones provenientes de zonas temporales de la pila.

Los patrones de diseño seguros, como "factory methods", "singleton controlado", o contenedores de objetos gestionados, permiten controlar el ciclo de vida de los recursos y evitar fugas de direcciones hacia ámbitos de mayor duración. Igualmente, encapsular los datos internos y exponer solo interfaces que devuelvan copias o punteros gestionados reduce notablemente el riesgo de errores relacionados con la memoria. El objetivo es que jamás se necesite exponer directamente la dirección de un objeto automático fuera del contexto donde reside.

Cuando se trabaja con APIs de terceros escritas en C o C++, es habitual que ciertas funciones esperen recibir punteros válidos cuya vida útil exceda el uso inmediato. En tales escenarios, nunca se debe pasar la dirección de un objeto automático si la

biblioteca puede almacenarlo internamente o usarlo después de que el ámbito local termine. Resulta indispensable revisar la documentación y las garantías sobre propiedad que ofrece cada API y, cuando sea necesario, usar almacenamiento dinámico o estructuras persistentes.

El código heredado puede contener patrones peligrosos que asignan direcciones de variables de la pila a estructuras globales o estáticas. Durante refactorizaciones, es fundamental auditar cualquier uso de direcciones locales almacenadas en entidades persistentes. La modernización a C++ contemporáneo mediante punteros inteligentes, referencias seguras y construcción explícita de objetos permite prevenir errores históricos que, por su naturaleza, pasan desapercibidos durante largos periodos. Estas labores de limpieza reducen significativamente el riesgo de vulnerabilidades relacionadas con invalidación de memoria.

Riesgos asociados

- Desreferenciación de punteros colgantes (dangling pointers).
- Lectura de datos corruptos o inesperados.
- Condiciones de carrera y comportamiento indefinido al acceder a memoria liberada.
- Acceso a memoria fuera del alcance, resultando en fallos de segmentación (segmentation faults).
- Vulnerabilidades explotables para ejecución de código arbitrario.
- Filtración de información sensible almacenada temporalmente en variables locales.
- Dificultades para identificar y depurar errores intermitentes o de difícil reproducción.
- Comportamiento no determinista y dependiente del compilador o del entorno de ejecución.

Buenas prácticas

- No almacenar la dirección de una variable local en un puntero con mayor tiempo de vida (global o estático).

```
//-- Asignación de dirección de variable automática a puntero global
int* g_ptr;

void setPointer() {
  int local = 10;
  g_ptr = &local;
}
```

```
//-- Uso de almacenamiento dinámico o estático para extender la duración
int* g_safePtr;

void setSafePointer() {
  g_safePtr = new int(10);
}
```

- Evitar retornar la dirección de una variable automática desde una función.

```
//-- Retorno de dirección de variable automática
int* createValue() {
  int value = 5;
  return &value;
}
```

```
//-- Retorno por valor para mantener validez del objeto
int createSafeValue() {
  int value = 5;
  return value;
}
```

- No almacenar referencias a objetos locales en estructuras con mayor duración.

```
//-- Almacenamiento de referencia a variable local en objeto persistente
struct Holder {
  const int& ref;
};

Holder* createHolder() {
  int local = 7;
  return new Holder{local};
}
```

```
//-- Copia del valor para evitar dependencia de duración automática
struct SafeHolder {
  int value;
};

SafeHolder* createSafeHolder() {
  int local = 7;
  return new SafeHolder{local};
}
```

2.11.11.2 Estándares metodológicos

- **C++ Core-F.43**: Never (directly or indirectly) return a pointer or a reference to a

local object.

2.11.12 [cpp.ptr.12] - Evitar acceder a un objeto fuera de su tiempo de vida

ID	cpp.ptr.12	Uso	Obligatorio
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se debe garantizar que ninguna parte del programa acceda a un objeto después de que su tiempo de vida haya finalizado. Este principio es crítico para evitar vulnerabilidades graves como **use-after-free**, desreferencias inválidas, corrupción de memoria o comportamiento indefinido. El control exige una gestión cuidadosa de la duración de variables automáticas, dinámicas y estáticas, así como un uso seguro de punteros, referencias, estructuras RAII y objetos compartidos. Su objetivo es asegurar la estabilidad del programa y preservar la integridad de los datos.

Codificación insegura

Se muestra cómo acceder a un objeto después de que su tiempo de vida ha terminado, generando un puntero colgante. La intención es observar qué ocurre cuando se devuelve la dirección de una variable local y posteriormente se intenta utilizar, entendiendo los riesgos asociados al manejo incorrecto de memoria.

```
#include <iostream>

//-- Función que devuelve la dirección de una variable local
int* obtenerDireccion() {
  //-- Variable local con almacenamiento automático
  int valor = 42;

  //-- Se retorna la dirección de una variable que dejará de existir
  return &valor;
}

int main() {
  //-- Se recibe un puntero a memoria inválida
  int* puntero = obtenerDireccion();

  //-- Acceso a memoria cuyo tiempo de vida ya terminó
  std::cout << "Valor: " << *puntero << std::endl;
  return 0;
}
```

Codificación recomendada

Se presenta una forma segura de garantizar que el objeto mantiene un tiempo de vida válido al ser utilizado fuera de la función, empleando almacenamiento dinámico gestionado automáticamente mediante RAII. El objetivo es evitar punteros colgantes y asegurar que el acceso al objeto se realiza mientras su memoria sigue siendo válida.

```
#include <iostream>
#include <memory>

//-- Función que crea un entero en memoria dinámica gestionada
std::unique_ptr<int> obtenerDireccion() {
  //-- Se reserva memoria dinámica controlada por unique_ptr
  std::unique_ptr<int> valor = std::make_unique<int>(42);

  //-- Se transfiere la propiedad al llamador
  return valor;
}

int main() {
  //-- Se recibe un objeto cuyo tiempo de vida está gestionado
  std::unique_ptr<int> puntero = obtenerDireccion();

  //-- Acceso seguro mientras el objeto esté en alcance
  std::cout << "Valor: " << *puntero << std::endl;

  //-- La memoria se libera automáticamente al salir de alcance
  return 0;
}
```

2.11.12.1 Descripción

El control preciso del tiempo de vida de los objetos es un requisito fundamental para evitar fallos de seguridad y errores de ejecución. Debido a la naturaleza del lenguaje y su gestión manual de memoria, el desarrollador debe garantizar que ningún puntero o referencia sobreviva más que el objeto al que apunta. El incumplimiento de esta regla deriva en **use-after-free**, punteros colgantes, escrituras fuera de límites y fallos críticos que pueden comprometer la integridad del sistema y, en muchos casos, abrir la puerta a la ejecución arbitraria de código.

Existen múltiples escenarios en los que los objetos pueden perder validez: finalización de bloques de ámbito automático, liberación explícita con `delete`, destrucción de contenedores estándar, invalidación de iteradores, movimientos de objetos, reasignaciones internas de contenedores y devoluciones de referencias a datos temporales. En todos los casos, el uso posterior de tales objetos conduce a un comportamiento indefinido. El riesgo aumenta cuando el código intenta usar punteros o referencias que siguen pareciendo válidos, especialmente si la memoria ha sido

reutilizada silenciosamente por el runtime o el asignador de memoria. Este requisito demanda un esfuerzo consciente por diseñar estructuras y flujos que hagan imposible acceder a objetos expirados.

Las variables automáticas solo existen dentro del bloque en el que se declaran. Devolver un puntero o referencia a ellas desde una función causa que el receptor obtenga un manejador inválido. Este patrón es una de las principales causas de errores silenciosos en C++. El uso disciplinado de valores de retorno por copia o por objetos administrados mediante RAII previene este fallo.

Cuando se libera memoria con `delete`, el puntero original queda colgante. Acceder a él, incluso para operaciones aparentemente inofensivas, provoca corrupción de memoria. Una práctica obligatoria es asignar `nullptr` al puntero tras liberar la memoria, aunque preferiblemente se debe evitar el uso directo de `new` y `delete` y adoptar punteros inteligentes.

Los punteros inteligentes como `std::unique_ptr` y `std::shared_ptr` permiten modelar la propiedad y el tiempo de vida de objetos dinámicos. De forma natural, impiden liberar accidentalmente un recurso aún referenciado o evitar fugas de memoria. No obstante, incluso con punteros inteligentes, sigue siendo indispensable evitar ciclos de referencia, referencias colgantes o transferencias incorrectas de propiedad.

Los contenedores pueden invalidar internamente iteradores, referencias o punteros a sus elementos cuando cambian de tamaño o reorganizan su contenido. Operaciones como `push_back`, `erase`, `insert`, `resize` o asignaciones completas pueden causar invalidaciones silenciosas. Para un desarrollo seguro, cada operación susceptible de reubicación debe ser seguida de una actualización explícita de los manejadores o de un rediseño que evite su uso persistente.

Cuando un objeto es movido, su estado puede quedar en una condición válida pero indeterminada. Usar referencias o punteros al objeto original después de una operación de movimiento que transfiera su contenido a otro objeto puede exponer datos inconsistentes o causar fallos de acceso. El uso seguro exige evitar retener referencias a objetos sujetos a movimientos.

RAII asocia la adquisición y liberación de recursos a la construcción y destrucción de objetos, garantizando tiempos de vida claros. El uso adecuado de RAII impide la liberación prematura y evita almacenar punteros crudos que puedan invalidarse. La utilización de objetos RAII también simplifica el control de excepciones y asegura que ningún recurso se utilice tras su destrucción.

En aplicaciones complejas, especialmente con múltiples hilos, callbacks, colas de eventos o frameworks asíncronos, el tiempo de vida de los objetos puede volverse difícil de rastrear. Para estos contextos, es recomendable adoptar estrategias como inyectar dependencias por valor o por punteros inteligentes, evitar capturas inseguras en lambdas, prevenir condiciones de carrera mediante sincronización explícita y diseñar

arquitecturas que hagan imposible el acceso a objetos destruidos, incluyendo gestionadores centralizados de recursos.

Riesgos asociados

- Desreferenciación de punteros colgantes (dangling pointers), lo que puede causar fallos de segmentación (segmentation faults) y caídas inesperadas del programa.
- Condición de carrera o comportamientos indefinidos al reutilizar memoria previamente liberada, que puede contener datos arbitrarios.
- Divulgación de información sensible almacenada en la memoria reutilizada accidentalmente.
- Ejecutar código con datos corruptos, generando errores lógicos difíciles de rastrear.
- Exposición a vulnerabilidades de seguridad como corrupción de memoria, exploits de use-after-free y potencial escalada de privilegios o ejecución remota de código.
- Inestabilidad general del sistema y difícil diagnóstico de errores intermitentes.

Buenas prácticas

- Evitar devolver punteros o referencias a variables locales.

```
//-- Retorno de referencia a variable local que deja de existir tras salir del
ámbito
int& obtenerValor() {
  int x = 10;
  return x;
}
```

```
//-- Retorno por valor para evitar referencias a memoria liberada
int obtenerValorSeguro() {
  int x = 10;
  return x;
}
```

- No utilizar punteros a objetos destruidos o liberados.

```
//-- Uso de memoria después de liberarla
void ejemplo() {
  int* ptr = new int(5);
  delete ptr;
  *ptr = 10;
```

```
}
```

```
//-- Anulación del puntero tras liberar y evitar su reutilización
void ejemploSeguro() {
  int* ptr = new int(5);
  delete ptr;
  ptr = nullptr;
}
```

- Gestionar recursos mediante RAII en lugar de manejo manual de memoria.

```
//-- Almacenamiento de dirección de objeto automático fuera de su ámbito
int* crear() {
  int valor = 20;
  return &valor;
}
```

```
//-- Uso de std::unique_ptr para asegurar liberación controlada
std::unique_ptr crearSeguro() {
  return std::make_unique(20);
}
```

- Evitar capturar por referencia objetos con menor tiempo de vida en lambdas.

```
//-- Lambda que captura por referencia una variable que sale de ámbito
std::function obtenerLambda() {
  int x = 7;
  return [&]() { return x; };
}
```

```
//-- Captura por valor para extender la vida útil dentro de la lambda
std::function obtenerLambdaSeguro() {
  int x = 7;
  return [x]() { return x; };
}
```

2.11.12.2 Estándares metodológicos

- **MISRA CPP-6.8.1**: No access outside object lifetime.

2.11.13 [cpp.ptr.13] - Evitar desreferenciar punteros inválidos

ID	cpp.ptr.13	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se busca evitar el acceso mediante punteros que ya no son válidos, nunca fueron inicializados o apuntan a regiones de memoria liberadas, lo cual puede originar fallos críticos, corrupción de datos y vulnerabilidades explotables. Para mitigar estos riesgos, se recomienda emplear mecanismos seguros de gestión de memoria como **smart pointers**, inicializar siempre los punteros, validar su estado antes de utilizarlos y limitar al máximo las operaciones que interactúan directamente con direcciones crudas.

Codificación insegura

Gestión de memoria dinámica usando punteros crudos donde se reserva memoria en el heap y posteriormente se accede a ella. Se busca observar qué ocurre cuando un puntero puede quedar en estado nulo y aun así se intenta desreferenciar, provocando un comportamiento indefinido.

```
#include <iostream>

int main() {
  //-- Declaración de un puntero sin inicializar correctamente
  int* ptr = nullptr;

  //-- Simulación de una condición donde la memoria no se asigna
  bool reservarMemoria = false;
  if (reservarMemoria) {
    ptr = new int(10);
  }
  //-- Desreferenciación sin validar si el puntero es válido
  std::cout << "Valor: " << *ptr << std::endl;

  //-- Liberación de memoria si fue asignada
  delete ptr;
  return 0;
}
```

Codificación recomendada

Gestión segura de memoria dinámica validando el estado del puntero antes de acceder a él. Se muestra cómo comprobar que el puntero contiene una dirección válida antes de desreferenciarlo, evitando comportamiento indefinido y fallos en tiempo de ejecución.

```
#include <iostream>
```

```
int main() {
  //-- Declaración e inicialización segura del puntero
  int* ptr = nullptr;

  //-- Simulación de una condición donde la memoria sí se asigna
  bool reservarMemoria = true;
  if (reservarMemoria) {
    ptr = new int(10);
  }
  //-- Validación del puntero antes de desreferenciar
  if (ptr != nullptr) {
    std::cout << "Valor: " << *ptr << std::endl;
  } else {
    std::cout << "Puntero no válido" << std::endl;
  }
  //-- Liberación segura de memoria
  delete ptr;
  return 0;
}
```

2.11.13.1 Descripción

El manejo inseguro de punteros es una de las causas más frecuentes de fallos graves en aplicaciones escritas en C++. La desreferenciación de punteros inválidos suele ocurrir cuando un puntero apunta a memoria que ya ha sido liberada, nunca fue inicializada o ha sido sobrescrita por error. Estas situaciones pueden generar errores como **segmentation faults**, corrupción silenciosa de memoria, fugas de información o comportamientos indefinidos difíciles de depurar. Por este motivo, las buenas prácticas de codificación segura recomiendan reducir al mínimo el uso de punteros sin control explícito, sustituirlos por abstracciones seguras y aplicar verificaciones sistemáticas en aquellos casos donde su uso sea estrictamente necesario.

El acceso mediante punteros inválidos puede desencadenar múltiples problemas a nivel de seguridad. Muchos de estos riesgos son aprovechados por atacantes para manipular el flujo del programa o extraer información sensible. C++ otorga un control muy fino sobre la memoria, pero esto implica una gran responsabilidad por parte del desarrollador. Entre los riesgos más relevantes destaca la posibilidad de que, al no validarse adecuadamente un puntero, se termine accediendo a una región que contiene datos residuales de otro proceso, exponiendo información sensible. También existe el riesgo de modificar memoria crítica, lo que puede comprometer la integridad del proceso en ejecución.

Para prevenir la desreferenciación de punteros inválidos, se deben aplicar diversas prácticas de programación defensiva. Una de ellas consiste en inicializar siempre los punteros en el momento de su declaración. Otra es el uso de **smart pointers** como

`std::unique_ptr` o `std::shared_ptr`, que automatizan la gestión de recursos y minimizan los riesgos de fugas o referencias colgantes. Asimismo, es importante evitar operaciones arriesgadas con memoria dinámica cuando existan alternativas más seguras como contenedores estándar de la STL. Finalmente, verificar explícitamente que un puntero no sea nulo antes de usarlo es una regla fundamental incluso cuando su validez parece garantizada.

En algunas aplicaciones de alto rendimiento, se suele justificar el uso de punteros crudos para evitar la sobrecarga de los smart pointers. Sin embargo, incluso en estos casos es posible aplicar patrones seguros. Por ejemplo, se pueden utilizar punteros crudos únicamente como vistas temporales, mientras que la propiedad del recurso reside en un contenedor o smart pointer. De esta forma, se minimiza el riesgo sin comprometer significativamente la eficiencia. También es útil aplicar herramientas de análisis como sanitizadores de memoria, que detectan accesos indebidos y ayudan a identificar errores en fases tempranas del desarrollo.

Los errores más comunes para evitar son:

- **No comprobar punteros:** Suponer que un puntero siempre será válido.
- **Ignorar el ciclo de vida:** Usar punteros fuera del ámbito donde fueron creados.
- **Asignación incorrecta:** Sobrescribir punteros sin liberar la memoria previa.
- **Desreferenciar punteros liberados:** Acceder a memoria ya devuelta al sistema.
- **Aplicar aritmética insegura:** Manipular direcciones sin validar rangos.

Riesgos asociados

- Corrupción de memoria.
- Caída inesperada del programa (crash/segmentation fault).
- Ejecución de código arbitrario (potencial explotación).
- Lectura o escritura de datos sensibles o críticos en direcciones de memoria no previstas.
- Condiciones de carrera y comportamientos indefinidos.
- Vulnerabilidades de seguridad aprovechables por atacantes (como exploit de buffer overflow).
- Dificultad en el diagnóstico y depuración de errores.
- Pérdida de integridad en los datos de la aplicación.
- Inestabilidad general del sistema.
- Aumento en los costos de mantenimiento y soporte.

Buenas prácticas

- Validar siempre que un puntero no sea nulo antes de desreferenciarlo.

```
//-- Desreferenciación sin validación previa
int* ptr = obtenerValor();
int valor = *ptr;
```

```
//-- Validación explícita antes de acceder al contenido
int* ptr = obtenerValor();
if (ptr != nullptr) {
  int valor = *ptr;
}
```

- Inicializar los punteros en el momento de su declaración.

```
//-- Puntero sin inicializar
int* ptr;
*ptr = 10;
```

```
//-- Inicialización segura del puntero
int valor = 0;
int* ptr = &valor;
*ptr = 10;
```

- Asignar `nullptr` después de liberar memoria dinámica.

```
//-- Uso de puntero después de liberar memoria
int* ptr = new int(5);
delete ptr;
int valor = *ptr;
```

```
//-- Invalidación explícita tras liberar memoria
int* ptr = new int(5);
delete ptr;
ptr = nullptr;
if (ptr != nullptr) {
  int valor = *ptr;
}
```

- Utilizar punteros inteligentes para gestionar la memoria automáticamente.

```
//-- Gestión manual con riesgo de puntero colgante
int* ptr = new int(20);
procesar(ptr);
delete ptr;
```

```
//-- Uso de std::unique_ptr para gestión automática
std::unique_ptr<int> ptr = std::make_unique<int>(20);
procesar(ptr.get());
```

- Comprobar el resultado de funciones que pueden devolver punteros inválidos.

```
//-- Uso directo del valor retornado
int valor = *buscarElemento(id);
```

```
//-- Validación del puntero retornado antes de usarlo
int* resultado = buscarElemento(id);
if (resultado != nullptr) {
  int valor = *resultado;
}
```

2.11.13.2 Estándares metodológicos

- **C++ Core-ES.65**: don't dereference an invalid pointer.

2.11.14 [cpp.ptr.14] - Asegurar que los accesos a contenedores estén siempre dentro de rango

ID	cpp.ptr.14	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se deben validar rigurosamente todos los accesos a contenedores para evitar operaciones fuera de rango que puedan comprometer la estabilidad, seguridad y comportamiento del sistema. El objetivo es prevenir accesos no controlados a memoria, corrupciones de datos y vulnerabilidades explotables derivadas de índices incorrectos, iteradores inválidos o uso inseguro de operaciones de bajo nivel. Para ello se recomienda utilizar mecanismos seguros del lenguaje, comprobaciones explícitas y estructuras proporcionadas por la biblioteca estándar que verifiquen límites y garanticen un uso seguro de cualquier contenedor.

Codificación insegura

Acceso a un elemento de un `std::vector` usando un índice proporcionado por el usuario. Se busca mostrar cómo un acceso directo sin validación puede provocar comportamiento indefinido cuando el índice está fuera de rango, y la importancia de controlar los límites antes de acceder al contenedor.

```
#include <iostream>
#include <vector>
```

```
int main() {
  //-- Crear un vector con tres elementos
  std::vector<int> numeros = {10, 20, 30};

  //-- Solicitar al usuario un índice
  std::cout << "Introduce un indice: ";
  int indice;
  std::cin >> indice;

  //-- Acceder directamente al vector sin validar el rango
  int valor = numeros[indice];

  //-- Mostrar el valor obtenido
  std::cout << "Valor: " << valor << std::endl;
  return 0;
}
```

Codificación recomendada

Acceso a un elemento de un `std::vector` validando previamente que el índice esté dentro de los límites del contenedor. Se busca demostrar cómo prevenir accesos fuera de rango mediante comprobaciones explícitas antes de utilizar el índice.

```
#include <iostream>
#include <vector>

int main() {
  //-- Crear un vector con tres elementos
  std::vector<int> numeros = {10, 20, 30};

  //-- Solicitar al usuario un índice
  std::cout << "Introduce un indice: ";
  int indice;
  std::cin >> indice;

  //-- Validar que el índice esté dentro del rango permitido
  if (indice >= 0 && indice < static_cast<int>(numeros.size())) {
    //-- Acceder al vector solo si el índice es válido
    int valor = numeros[indice];
    std::cout << "Valor: " << valor << std::endl;
  } else {
    //-- Informar de índice fuera de rango
    std::cout << "Error: indice fuera de rango" << std::endl;
  }
  return 0;
}
```

2.11.14.1 Descripción

El acceso seguro a los contenedores minimiza todos aquellos riesgos asociados a vulnerabilidades que derivan del uso indebido de índices, punteros o iteradores. Dado que C++ ofrece un alto grado de control sobre la memoria, es fundamental aplicar medidas que garanticen que cualquier operación de lectura o escritura sobre estructuras como `std::vector`, `std::array`, `std::deque` o buffers manuales se realice siempre dentro de los límites válidos. Cuando estas operaciones no se controlan, existe un potencial significativo de provocar comportamientos indefinidos, corrupción de memoria, fallos del programa o vulnerabilidades explotables que pueden comprometer la integridad y estabilidad de las aplicaciones.

Es imprescindible recordar que muchos métodos del estándar no realizan comprobaciones de rango, como el operador `[]` de `std::vector` o el acceso directo a elementos de un `std::array`. Esto permite un acceso rápido, pero también expone a riesgos considerables si el índice no ha sido validado. Por ello, es necesario preferir mecanismos como `at()` cuando se requieran validaciones automáticas, complementados con controles adicionales en casos donde la lógica de negocio dependa de cálculos dinámicos de índices. Las comprobaciones explícitas mediante comparaciones de límites siguen siendo esenciales en algoritmos personalizados u operaciones sobre memoria manual, especialmente cuando se manejan punteros crudos.

Además, resulta crítico evitar cualquier manipulación que deje iteradores en estado inválido. Operaciones como inserciones o eliminaciones sobre contenedores pueden invalidar iteradores y referencias existentes. Cuando un iterador inválido se utiliza posteriormente, puede producir accesos fuera de rango o condiciones de carrera sobre recursos compartidos. Por lo tanto, es necesario revisar cuidadosamente el diseño de las operaciones sobre contenedores, asegurando que, tras cualquier modificación estructural, los iteradores se actualicen o recalculen antes de volver a utilizarlos.

En contextos donde se gestionan buffers manuales, como arrays estáticos o memoria gestionada mediante `new` y `delete`, la responsabilidad recae por completo en el desarrollador. En estos escenarios, es obligatorio implementar validaciones exhaustivas que garanticen que los punteros no superen los límites asignados y que no se realicen desbordamientos por lectura o escritura. Asimismo, el uso de funciones de bajo nivel como `memcpy`, `memmove` o `strcpy` debe acompañarse de verificaciones estrictas del tamaño de origen y destino, evitando así sobrescrituras involuntarias.

Riesgos asociados

- Lectura o escritura fuera de los límites del contenedor (out-of-bounds), lo que puede causar corrupción de memoria.
- Posibilidad de ejecutar código malicioso debido a la sobrescritura de áreas críticas de memoria.

- Exposición de información sensible almacenada contiguamente en memoria.
- Caídas inesperadas (crash) del programa por acceso ilegal a memoria.
- Comportamiento indefinido del programa, lo que dificulta la localización y corrección de errores.
- Vulnerabilidades explotables, como buffer overflow, que pueden ser aprovechadas en ataques de escalada de privilegios.
- Violación de la integridad de los datos al modificar información fuera del contenedor esperado.
- Problemas de estabilidad del sistema o aplicación por corrupción de heap/stack.

Buenas prácticas

- Validar explícitamente los límites antes de acceder a un contenedor mediante índice.

```
//-- Acceso directo sin validar límites
std::vector v = {1,2,3};
int value = v[5];
```

```
//-- Validación previa del índice antes del acceso
std::vector v = {1,2,3};
size_t index = 5;
if(index < v.size()) {
  int value = v[index];
}
```

- Utilizar el método `at()` cuando se requiera verificación automática de rango.

```
//-- Acceso mediante operador [] sin verificación interna
std::vector v = {1,2,3};
int value = v[2];
```

```
//-- Acceso mediante at() con verificación de rango
std::vector v = {1,2,3};
try {
  int value = v.at(2);
} catch(const std::out_of_range&) {
  // manejo de error
}
```

- Evitar el uso de índices derivados de datos externos sin validación previa.

```
//-- Índice proveniente de entrada externa sin validar
```

```
std::vector v = {10,20,30};
int index;
std::cin >> index;
int value = v[index];
```

```
//-- Validación del índice recibido antes del acceso
std::vector v = {10,20,30};
int index;
std::cin >> index;
if(index >= 0 && static_cast(index) < v.size()) {
  int value = v[index];
}
```

- Utilizar bucles basados en iteradores o rango para evitar cálculos manuales de índices.

```
//-- Iteración manual con posible error en límites
std::vector v = {1,2,3};
for(size_t i = 0; i <= v.size(); ++i) {
  process(v[i]);
}
```

```
//-- Iteración basada en rango evitando acceso fuera de límites
std::vector v = {1,2,3};
for(const auto& elem : v) {
  process(elem);
}
```

- Sincronizar las operaciones de acceso cuando el contenedor pueda modificarse concurrentemente.

```
//-- Acceso concurrente sin sincronización
std::vector v = {1,2,3};
std::thread t([&](){ v.push_back(4); });
int value = v[0];
t.join();
```

```
//-- Acceso protegido mediante mutex para mantener coherencia de tamaño
std::vector v = {1,2,3};
std::mutex m;
std::thread t([&](){
  std::lock_guard<std::mutex> lock(m);
  v.push_back(4);
});
{
  std::lock_guard<std::mutex> lock(m);
```

```
  if(!v.empty()) {
    int value = v[0];
  }
}
t.join();
```

2.11.14.2 Estándares metodológicos

- **C++ Core-SL.con.3**: Avoid bounds errors.
- **CERT/CPP-CTR50**: Guarantee that container indices and iterators are within the valid range.

2.12 Gestión de memoria

2.12.1 [cpp.mem.1] - Evitar el uso de "malloc" y "free"

ID	cpp.mem.1	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Debe evitarse el uso directo de `malloc` y `free`, promoviendo el empleo de mecanismos modernos de gestión automática de memoria basados en RAII, como `new`/`delete` dentro de contenedores seguros o, preferiblemente, punteros inteligentes como `std::unique_ptr` y `std::shared_ptr`. Se reducen riesgos de fugas de memoria, dobles liberaciones, corrupción del heap y errores difíciles de depurar que afectan tanto a la seguridad como a la estabilidad del software.

Codificación insegura

Gestión manual de memoria dinámica en C++ utilizando funciones heredadas de C. Se reserva un bloque de memoria para un array de enteros, se usa y posteriormente se libera. El objetivo es mostrar cómo se realiza esta práctica y los riesgos asociados cuando no se emplean mecanismos propios del lenguaje para la gestión automática de recursos.

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

int main() {
  //-- Reserva de memoria dinámica usando malloc
  int* numeros = (int*) std::malloc(5 * sizeof(int));

  //-- Verificación básica de puntero nulo
  if (numeros == nullptr) {
    std::cerr << "Error al reservar memoria" << std::endl;
    return 1;
```

```
  }
  //-- Inicialización del array
  for (int i = 0; i < 5; ++i) {
    numeros[i] = i * 10;
  }
  //-- Uso del array
  for (int i = 0; i < 5; ++i) {
    std::cout << numeros[i] << std::endl;
  }
  //-- Liberación manual de memoria usando free
  std::free(numeros);

  //-- Posible riesgo si se usa el puntero después de liberar
  return 0;
}
```

Codificación recomendada

Gestión de memoria dinámica utilizando contenedores estándar de C++, delegando la responsabilidad de la reserva y liberación de recursos al propio lenguaje. Se crea un array dinámico mediante una estructura segura que libera automáticamente la memoria cuando sale de alcance, reduciendo riesgos como fugas o uso después de liberar.

```
#include <iostream>
#include <vector>

int main() {
  //-- Creación de un contenedor dinámico gestionado automáticamente
  std::vector<int> numeros(5);

  //-- Inicialización del array
  for (int i = 0; i < 5; ++i) {
    numeros[i] = i * 10;
  }
  //-- Uso del array
  for (int i = 0; i < 5; ++i) {
    std::cout << numeros[i] << std::endl;
  }
  //-- Liberación automática al salir del ámbito
  return 0;
}
```

2.12.1.1 Descripción

El uso directo de `malloc` y `free` constituye un patrón obsoleto y potencialmente inseguro. Estas funciones pertenecen al ecosistema de C y no integran los mecanismos de inicialización, destrucción y gestión automática de recursos característicos del

paradigma RAII. Su empleo puede generar una amplia variedad de problemas, tales como fugas de memoria, dobles liberaciones, desajustes entre asignación y liberación, así como corrupción del heap. El objetivo de este requisito es asegurar que en el código se empleen herramientas seguras, potentes y expresivas para administrar memoria, aprovechando las facilidades provistas por la STL y por los constructores y destructores automáticos.

El manejo manual de memoria mediante `malloc` y `free` introduce complejidad innecesaria y errores sutiles. Estas funciones no ejecutan constructores ni destructores, lo que obliga al desarrollador a gestionar explícitamente la inicialización y la liberación de recursos. Además, carecen de información de tipo y pueden provocar incompatibilidades si se mezclan con `new` y `delete`. **El resultado es un aumento significativo en la probabilidad de defectos que pueden explotarse para provocar corrupción de memoria**, escaladas de privilegios, inestabilidad en tiempo de ejecución o fallos graves de seguridad.

El paradigma RAII constituye el núcleo de la gestión segura de los recursos. Su principio fundamental establece que cada recurso debe estar vinculado al ciclo de vida de un objeto, de modo que la adquisición del recurso se produzca en el constructor, y la liberación en el destructor. Los punteros inteligentes, como `std::unique_ptr` y `std::shared_ptr`, implementan este enfoque de manera robusta. **Al adoptar punteros inteligentes se garantiza la liberación automática y segura de memoria incluso si se producen excepciones**. Asimismo, los contenedores de la STL, como `std::vector`, `std::string` o `std::array`, eliminan la necesidad de gestionar memoria de forma explícita.

Algunos de los errores más comunes mitigados al evitar el uso de `malloc` y `free` son:

- **Fuga de memoria**: El desarrollador olvida llamar a `free`.
- **Doble liberación**: Se ejecuta `free` dos veces sobre el mismo puntero.
- **Desajuste de asignadores**: Uso incorrecto de `malloc`/`free` con `new`/`delete`.
- **Acceso a memoria liberada**: Uso de punteros colgantes tras liberar memoria.
- **Falta de construcción y destrucción adecuada**: No se invocan constructores ni destructores.

Riesgos asociados

- Pérdida de control sobre el ciclo de vida de los objetos, ocasionando fugas de memoria.
- Posibilidad de doble liberación de memoria (“double free”), resultando en corrupción de memoria o vulnerabilidades de seguridad.
- Incompatibilidad entre la gestión de memoria de C y C++ (por ejemplo, usar delete para liberar memoria reservada con malloc), lo que puede causar

comportamiento indefinido.

- Falta de invocación de constructores y destructores, evitando la inicialización o liberación adecuada de recursos.
- Aumento del riesgo de errores de tipo o conversiones indebidas de punteros (por ejemplo, al retornar void*).
- Mayor vulnerabilidad a ataques de desbordamiento de buffer debido a una gestión de memoria más manual y propensa a errores.
- Dificultad para mantener y depurar el código, afectando la robustez y la legibilidad.
- Limitaciones en la integración con características modernas de C++, como smart pointers o RAII.

Buenas prácticas

- Preferir RAII y punteros inteligentes (`std::unique_ptr`, `std::shared_ptr`) en lugar de `malloc`/`free`.

```
//-- Reserva manual sin constructor ni liberación automática
struct Recurso {
  int valor;
};

Recurso* r = (Recurso*)malloc(sizeof(Recurso));
r->valor = 10;
free(r);
```

```
//-- Uso de RAII con liberación automática y construcción adecuada
#include <memory>

struct Recurso {
  int valor;
  Recurso(int v) : valor(v) {}
};

std::unique_ptr<Recurso> r = std::make_unique<Recurso>(10);
```

- Utilizar contenedores estándar (`std::vector`, `std::string`) para gestionar memoria dinámica.

```
//-- Gestión manual de memoria para un array dinámico
int* datos = (int*)malloc(5 * sizeof(int));
for(int i = 0; i < 5; ++i) {
  datos[i] = i;
}
```

```
free(datos);
```

```
//-- Uso de contenedor estándar con control automático de memoria
#include <vector>

std::vector<int> datos(5);
for(int i = 0; i < 5; ++i) {
  datos[i] = i;
}
```

- Evitar mezclar `malloc`/`free` con `new`/`delete` para prevenir comportamientos indefinidos.

```
//-- Mezcla de mecanismos de gestión de memoria
struct Objeto {
  ~Objeto() {}
};

Objeto* obj = (Objeto*)malloc(sizeof(Objeto));
delete obj;
```

```
//-- Uso consistente de new/delete con llamada a destructores
Objeto* obj = new Objeto();
delete obj;
```

2.12.1.2 Estándares metodológicos

- **C++ Core-R.10**: Avoid malloc() and free().

2.12.2 [cpp.mem.2] - Evitar el uso directo de "new" y "delete" salvo encapsulación clara y justificada

ID	**cpp.mem.2**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Establece la necesidad de evitar la gestión manual de memoria mediante `new` y `delete`, promoviendo el uso de abstracciones seguras como **smart pointers**, **RAII** y fábricas especializadas. El objetivo es mitigar riesgos asociados a errores de punteros, uso después de liberar, fugas de memoria y corrupción de datos. El desarrollo debe orientarse a encapsular cualquier uso necesario de memoria dinámica dentro de componentes bien definidos, auditables y revisados con el fin de reducir los vectores de ataque.

Codificación insegura

Gestión manual de memoria dinámica mediante operadores directos para crear y liberar objetos en el heap, mostrando cómo se puede producir una fuga de memoria cuando no se controla correctamente el flujo de ejecución. Se busca entender los riesgos asociados a la administración manual de recursos.

```
#include <iostream>

class Usuario {
public:
  //-- Constructor que recibe un nombre
  Usuario(const std::string& nombre) : nombre_(nombre) {}

  //-- Método que muestra el nombre
  void mostrar() const {
    std::cout << "Usuario: " << nombre_ << std::endl;
  }

private:
  std::string nombre_;
};

int main() {
  //-- Reserva manual de memoria en el heap
  Usuario* usuario = new Usuario("Carlos");

  usuario->mostrar();

  //-- Simulación de salida anticipada. No se libera la memoria
  if (true) {
    return 0;
  }
  //-- Liberación manual de memoria (nunca se ejecuta)
  delete usuario;
  return 0;
}
```

Codificación recomendada

Gestión automática de memoria utilizando punteros inteligentes para asegurar la liberación correcta del recurso incluso ante salidas anticipadas o excepciones. Se busca aprender cómo aplicar RAII para evitar fugas y mejorar la seguridad del código.

```
#include <iostream>
#include <memory>

class Usuario {
```

```
public:
  //-- Constructor que recibe un nombre
  Usuario(const std::string& nombre) : nombre_(nombre) {}

  //-- Método que muestra el nombre
  void mostrar() const {
    std::cout << "Usuario: " << nombre_ << std::endl;
  }

private:
  std::string nombre_;
};

int main() {
  //-- Uso de std::unique_ptr para gestión automática
  std::unique_ptr<Usuario> usuario = std::make_unique<Usuario>("Carlos");

  usuario->mostrar();

  //-- Simulación de salida anticipada. Liberación automática del recurso
  if (true) {
    return 0;
  }
  return 0;
}
```

2.12.2.1 Descripción

La gestión manual de memoria con `new` y `delete` es una de las fuentes históricas más frecuentes de vulnerabilidades de seguridad y errores de robustez. Este requisito establece la directriz de evitar su uso directo salvo en casos donde exista una encapsulación clara, estricta y justificada que permita ejercer control, auditoría y verificación del ciclo de vida de los objetos creados dinámicamente. En la mayoría de los escenarios modernos, las construcciones del estándar como `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`, `std::make_unique` o `std::make_shared` eliminan la necesidad de llamar manualmente a operadores de asignación de memoria. El uso de estas abstracciones reduce significativamente riesgos como fugas, doble liberación, invalidación de punteros y accesos a memoria ya liberada.

Un diseño seguro debe seguir el principio **RAII** el cual asocia la vida de un recurso al alcance de un objeto. Esto garantiza que los recursos se liberen automáticamente al salir del ámbito, incluso en condiciones excepcionales. Por ello, los punteros inteligentes son herramientas fundamentales para encapsular memoria asignada dinámicamente permitiendo que la liberación se realice de forma determinista y segura sin intervención manual. Usarlos también encapsula la semántica de propiedad, haciendo explícito si un

objeto tiene un único propietario, múltiples propietarios o simplemente un observador no propietario.

En entornos reales, situaciones como la integración con APIs heredadas, bibliotecas externas o interfaces con hardware pueden requerir el uso de `new` o `delete`. En estos casos, es crítico que dicho uso quede encapsulado dentro de funciones o clases que implementen **constructores**, **destructores** y la semántica de movimiento apropiada, evitando exponer punteros sin gestión automática al resto del código. Asimismo, cualquier código que gestione memoria manualmente debe ser sometido a revisiones más estrictas, análisis estático específico y pruebas focalizadas en fugas y condiciones de carrera.

Riesgos asociados

- Fugas de memoria (memory leaks) por no liberar correctamente la memoria asignada con `new`.
- Dereferencing de punteros colgantes (dangling pointers) por liberar memoria con `delete` y seguir accediendo a ellos.
- Condiciones de doble liberación (double free) si se llama `delete` más de una vez sobre el mismo puntero.
- Fragmentación de memoria en aplicaciones de larga duración, afectando el desempeño.
- Errores de inicialización o destrucción incompleta de objetos.
- Vulnerabilidad a exploits de corrupción de memoria, como **use-after-free**.
- Dificultad en el mantenimiento y prueba del código por la gestión manual de los ciclos de vida de los objetos.
- Pérdida de portabilidad y reutilización del código al depender de mecanismos manuales en lugar de prácticas modernas (como smart pointers).

Buenas prácticas

- Preferir el uso de smart pointers (`std::unique_ptr`, `std::shared_ptr`) en lugar de usar directamente `new` y `delete`.

```
//-- Uso directo de new y delete
MyClass* ptr = new MyClass();
ptr->doSomething();
delete ptr;
```

```
//-- Gestión automática con std::unique_ptr
std::unique_ptr ptr = std::make_unique();
ptr->doSomething();
```

- Encapsular la creación y liberación de recursos dinámicos dentro de clases que apliquen RAII.

```
//-- Reserva manual expuesta al consumidor
MyResource* createResource() {
  return new MyResource();
}
```

```
//-- Encapsulación mediante RAII
class ResourceHolder {
  std::unique_ptr resource;
public:
  ResourceHolder() : resource(std::make_unique()) {}
  MyResource* get() const { return resource.get(); }
};
```

- Utilizar contenedores estándar (`std::vector`, `std::string`, etc.) en lugar de gestionar memoria dinámica manualmente.

```
//-- Gestión manual de array dinámico
int* data = new int[10];
data[0] = 42;
delete[] data;
```

```
//-- Uso de contenedor estándar
std::vector data(10);
data[0] = 42;
```

2.12.2.2 Estándares metodológicos

- **C++ Core-R.11**: Avoid calling new and delete explicitly.

2.12.3 [cpp.mem.3] - Evitar acceder a memoria ya liberada

ID	cpp.mem.3	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se debe evitar cualquier acceso a memoria que haya sido previamente liberada mediante operaciones como `delete`, `delete[]` o llamadas a asignadores personalizados. Acceder a memoria liberada constituye un comportamiento indefinido, pudiendo causar corrupción de datos, fallos de ejecución, escaladas de privilegio o vulnerabilidades explotables. Para mitigarlo, se recomienda emplear técnicas como el uso sistemático de punteros inteligentes, la anulación explícita de punteros tras su liberación y una gestión

estricta de ciclos de vida, reduciendo así la probabilidad de errores que comprometan la integridad o seguridad del sistema.

Codificación insegura

Gestión manual de memoria dinámica con punteros, donde se reserva memoria con `new`, se libera con `delete` y posteriormente se intenta acceder al mismo puntero. Se busca mostrar el riesgo de comportamiento indefinido cuando se utiliza memoria que ya ha sido liberada.

```
#include <iostream>

int main() {
  //-- Reserva memoria dinámica para un entero
  int* numero = new int(10);

  //-- Muestra el valor almacenado
  std::cout << "Valor inicial: " << *numero << std::endl;

  //-- Libera la memoria reservada
  delete numero;

  //-- Accede a memoria ya liberada (comportamiento indefinido)
  std::cout << "Valor despues de liberar: " << *numero << std::endl;
  return 0;
}
```

Codificación recomendada

Gestión segura de memoria dinámica evitando el acceso a memoria liberada, ya sea anulando el puntero tras liberar recursos o utilizando mecanismos que previenen el uso posterior. Se busca mostrar cómo eliminar el riesgo de uso después de liberar memoria.

```
#include <iostream>

int main() {
  //-- Reserva memoria dinámica para un entero
  int* numero = new int(10);

  //-- Muestra el valor almacenado
  std::cout << "Valor inicial: " << *numero << std::endl;

  //-- Libera la memoria reservada
  delete numero;

  //-- Anula el puntero para evitar uso posterior
  numero = nullptr;
```

```
  //-- Verifica que el puntero no sea nulo antes de acceder
  if (numero != nullptr) {
    std::cout << "Valor despues de liberar: " << *numero << std::endl;
  } else {
    std::cout << "Puntero invalido, no se accede a memoria liberada" << std::endl;
  }
  return 0;
}
```

2.12.3.1 Descripción

El acceso a memoria ya liberada es una de las vulnerabilidades más peligrosas y frecuentes en C++, debido a la naturaleza no gestionada de su modelo de memoria. Cuando un desarrollador libera explícitamente un bloque dinámico utilizando `delete`, `delete[]` o mecanismos equivalentes, cualquier lectura o escritura posterior sobre la dirección previamente válida pasa a constituir un comportamiento indefinido. Este tipo de fallos notoriamente difíciles de diagnosticar pueden originar corrupción silenciosa, cuelgues intermitentes, fuga de información sensible, alteración no autorizada de datos internos y en casos críticos permitir la ejecución de código arbitrario por parte de un atacante.

El principio fundamental de este requisito es que la memoria liberada deja inmediatamente de pertenecer al proceso, aun si su contenido permanece temporalmente accesible en la práctica. El estándar de C++ establece que cualquier interacción con un objeto destruido carece de validez y puede provocar fallos impredecibles. Por esta razón, es vital adoptar estrategias que minimicen los riesgos asociados a la gestión manual del ciclo de vida de los objetos, favoreciendo mecanismos automáticos y patrones de diseño que reduzcan el margen de error humano.

En aplicaciones multihilo, bibliotecas compartidas, servicios de larga ejecución o módulos que interactúan con controladores o hardware, el impacto del acceso a memoria liberada se amplifica considerablemente. Es crucial que los desarrolladores eviten exponer punteros crudos entre módulos sin una definición estricta de propiedad, utilicen bloqueos o mecanismos atómicos para sincronizar la destrucción de objetos compartidos, y adopten diseños basados en colas seguras u otros patrones concurrentes. **Los accesos diferidos**, como los empleados en colas de eventos o en comportamientos asíncronos, constituyen una fuente común de errores cuando se programan callbacks que dependen de objetos destruidos prematuramente.

Los sistemas embebidos o de tiempo real requieren especial precaución. La reutilización rápida de la memoria por parte del asignador puede permitir sobrescrituras casi inmediatas. En estos entornos, es recomendable emplear asignadores deterministas, pools de memoria estáticos o mecanismos que limiten el coste del análisis en tiempo de ejecución, manteniendo la seguridad sin degradar el rendimiento.

Riesgos asociados

- Acceso a memoria no válida que puede causar fallos de segmentación (crash de la aplicación).
- Lectura de datos sensibles residuales en memoria ya liberada (fuga de información).
- Escritura sobre memoria reutilizada por el sistema o por otras partes del programa, ocasionando corrupción de datos.
- Comportamiento indefinido que puede ser explotado por atacantes para ejecutar código arbitrario.
- Dificultad en el diagnóstico y depuración del software, aumentando los costes de mantenimiento.
- Vulnerabilidad a ataques de tipo use-after-free, que permiten escalado de privilegios o toma de control del sistema afectado.
- Fallas de seguridad que pueden propiciar denegación de servicio (DoS).

Buenas prácticas

- Asignar `nullptr` inmediatamente después de liberar memoria dinámica para evitar accesos posteriores accidentales.

```
//-- Puntero liberado pero reutilizado accidentalmente
int* ptr = new int(10);
delete ptr;
*ptr = 20;
```

```
//-- Puntero invalidado tras liberar memoria
int* ptr2 = new int(10);
delete ptr2;
ptr2 = nullptr;
if (ptr2 != nullptr) {
  *ptr2 = 20;
}
```

- Preferir punteros inteligentes (`std::unique_ptr`, `std::shared_ptr`) para gestionar automáticamente el ciclo de vida de los recursos.

```
//-- Gestión manual con riesgo de uso tras liberación
int* data = new int(5);
delete data;
if (data) {
  *data = 8;
}
```

```
//-- Gestión automática mediante RAII
std::unique_ptr<int> data2 = std::make_unique<int>(5);
data2.reset();
if (data2) {
  *data2 = 8;
}
```

- Evitar múltiples liberaciones del mismo recurso centralizando la responsabilidad de liberación en un único propietario.

```
//-- Liberación duplicada del mismo recurso
int* buffer = new int[10];
delete[] buffer;
delete[] buffer;
```

```
//-- Propiedad única claramente definida
std::unique_ptr<int[]> buffer2(new int[10]);
buffer2.reset();
```

- No conservar punteros o referencias a memoria liberada al salir de un ámbito o tras destruir el objeto propietario.

```
//-- Referencia colgante tras liberar memoria
int* create() {
  int* value = new int(42);
  delete value;
  return value;
}
```

```
//-- Transferencia segura de propiedad
std::unique_ptr<int> createSafe() {
  std::unique_ptr<int> value = std::make_unique<int>(42);
  return value;
}
```

2.12.3.2 Estándares metodológicos

- **CERT/CPP-MEM50**: Do not access freed memory.

2.12.4 [cpp.mem.4] - Borrar de forma segura la memoria con información sensible antes de su liberación

ID	cpp.mem.4	Uso	Recomendado
CWSS Score	21	Impacto técnico	Alto
Dimensión	Confidencialidad	S-SDLC	Build

Este control establece la necesidad de eliminar de forma fiable cualquier dato confidencial que permanezca en memoria antes de que la aplicación libere dichos recursos. La acción evita que información como claves criptográficas, credenciales o tokens de sesión queden accesibles por procesos posteriores, herramientas de depuración o posibles atacantes que logren obtener acceso al espacio de memoria tras la liberación. La práctica incluye el uso de funciones seguras de borrado, la sobrescritura explícita de buffers y la prevención de optimizaciones del compilador que puedan omitir este borrado, fortaleciendo así la protección ante fugas de información.

Codificación insegura

El siguiente fragmento reserva memoria para una contraseña, la utiliza y luego la libera sin realizar un borrado seguro, dejando los datos sensibles expuestos en memoria.

```
#include <iostream>
#include <cstring>
#include <cstdlib>

int main() {
  char *password = (char*)std::malloc(32);
  if (!password) return 1;

  std::strcpy(password, "SuperSecreta123!");
  std::cout << "Procesando..." << std::endl;

  // Liberaci&oacute;n insegura: no se borra el contenido sensible
  std::free(password);
  return 0;
}
```

Codificación recomendada

El siguiente ejemplo ilustra el borrado seguro de un buffer sensible usando `memset_s` antes de liberar la memoria.

```
#include <cstring>
#include <cstddef>
#include <cstdlib>
```

```
void secure_zero(void *ptr, size_t len) {
  volatile unsigned char *p = static_cast(ptr);
  while (len--) {
    *p++ = 0;
  }
}

int main() {
  char *secret = static_cast<char *>(std::malloc(32));
  if (!secret) return 1;

  std::strncpy(secret, "ClaveSuperSecreta", 31);
  secret[31] = '\0';

  secure_zero(secret, 32);
  std::free(secret);
  return 0;
}
```

2.12.4.1 Descripción

La eliminación segura de datos sensibles en memoria constituye un elemento fundamental para evitar filtraciones involuntarias de información en aplicaciones desarrolladas en C++. Debido a las características de bajo nivel del lenguaje, la gestión de memoria queda en manos del desarrollador, lo que introduce riesgos adicionales si no se implementan técnicas adecuadas para limpiar buffers antes de liberarlos. En C++, es común manejar datos extremadamente delicados, como contraseñas, claves simétricas, claves privadas o tokens temporales, los cuales pueden permanecer en memoria incluso después de liberar el espacio, si no se sobrescriben explícitamente. **Un atacante con acceso al entorno de ejecución, volcados de memoria, técnicas forenses o explotación de use-after-free** puede extraer información que no debería permanecer disponible.

Cuando se libera memoria sin limpiarla, el contenido previo se mantiene hasta que otro proceso, hilo o parte de la misma aplicación sobrescriba esa área. Esto significa que datos privados pueden quedar intactos durante un periodo prolongado. Este fenómeno se incrementa en sistemas con memoria compartida, buffers estáticos o regiones fácilmente inspeccionables mediante volcados. Algunas vulnerabilidades aprovechan estos comportamientos para reconstruir información crítica. De hecho, herramientas comunes de análisis permiten visualizar estos restos sin necesidad de grandes capacidades técnicas, lo que vuelve aún más importante la implementación del borrado seguro.

El borrado seguro no consiste en un simple `memset()` estándar de los bytes a 0. El compilador puede eliminar la llamada si considera que el programa no utilizará el buffer después, ya que dicha operación se percibe como una optimización válida. Es

fundamental emplear funciones especialmente diseñadas o patrones que eviten este comportamiento. Algunas bibliotecas modernas incluyen funciones que fuerzan la sobrescritura sin permitir al compilador eliminarlas. Cuando estas funciones no están disponibles, deben aplicarse técnicas alternativas, como el uso de `volatile`, aunque este mecanismo no siempre garantiza un comportamiento a prueba de optimizaciones. En entornos críticos, es recomendable encapsular estas operaciones en utilidades específicas de seguridad de la aplicación.

Ejemplo de borrado seguro en C++

```
void secure_zero(void *ptr, size_t len) {
  volatile unsigned char *p = static_cast(ptr);
  while (len--) {
    *p++ = 0;
  }
}
```

Esta técnica usa un puntero `volatile` para impedir la optimización agresiva del compilador. Aun así, no siempre garantiza un comportamiento completamente seguro en todos los compiladores o plataformas, por lo que se recomienda emplear funciones como `std::memset_s` en entornos compatibles. Las bibliotecas criptográficas profesionales suelen ofrecer sus propias rutinas optimizadas y auditadas para borrar memoria, y se recomienda su uso.

Consideraciones para datos sensibles en estructuras complejas

Los datos sensibles no siempre residen en simples arrays. En C++ es frecuente utilizar estructuras anidadas, objetos con asignadores personalizados, contenedores de la STL o buffers temporales asociados a operaciones criptográficas. En tales casos, es necesario que el desarrollador implemente borrado seguro dentro de destructores, asignadores move, operadores de copia y mecanismos RAII. Es fundamental asegurarse de que, cuando un objeto se destruya, cualquier rastro de información sensible se sobrescriba explícitamente antes de liberar memoria. Asimismo, en contenedores estándar, como `std::vector<char>`, la memoria interna puede reubicarse automáticamente, copiando información sensible a nuevas zonas. Esto obliga a adoptar estrategias de control de capacidad fija o contenedores especializados para seguridad.

Riesgos asociados

- Filtración de información sensible a través de volcados de memoria.
- Recuperación posterior de datos sensibles por parte de atacantes mediante técnicas forenses.
- Exposición de contraseñas, claves criptográficas o datos personales en memoria reutilizada.

- Incumplimiento de regulaciones o normativas de protección de datos (por ejemplo, GDPR).
- Aumento de la superficie de ataque frente a malware o exploits que buscan leer información residual en la memoria.
- Riesgo de ataques internos, donde usuarios privilegiados puedan acceder a datos remanentes.
- Pérdida de confianza de usuarios o clientes debido a posibles brechas de seguridad.

Buenas prácticas

- Utilizar funciones específicas de borrado seguro que no puedan ser optimizadas por el compilador antes de liberar memoria con datos sensibles.

```
//-- Liberación directa de memoria sin limpiar el contenido sensible
char *password = malloc(32);
strcpy(password, "Secreto123");
free(password);
```

```
//-- Limpieza explícita de la memoria antes de liberarla utilizando función segura
char *password = malloc(32);
strcpy(password, "Secreto123");
secure_zero(password, 32);
free(password);
```

- Evitar el uso de funciones estándar como memset para borrar datos sensibles cuando puedan ser eliminadas por optimización; emplear alternativas garantizadas o mecanismos volátiles.

```
//-- Uso de memset que podría ser eliminado por optimización del compilador
char token[64];
strcpy(token, "ABCDEF123456");
memset(token, 0, sizeof(token));
```

```
//-- Uso de puntero volátil para asegurar el borrado efectivo del contenido
sensible
char token[64];
strcpy(token, "ABCDEF123456");
volatile char *p = token;
for(size_t i = 0; i < sizeof(token); i++) {
  p[i] = 0;
}
```

2.12.4.2 Estándares metodológicos

- **CERT/CPP-EXP53**: Do not read uninitialized memory.
- **CERT/CPP-MEM50**: Do not access freed memory.

2.13 Gestión de recursos

2.13.1 [cpp.res.1] - Gestionar automáticamente los recursos mediante RAII

ID	cpp.res.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Design

La RAII permite garantizar que los recursos del sistema, como memoria, archivos o manejadores de red, sean liberados de forma segura y determinista mediante la asociación de su ciclo de vida con el de los objetos. Esta práctica reduce riesgos frecuentes como fugas de memoria, dobles liberaciones, estados incoherentes y errores de seguridad derivados de la gestión manual.

Codificación insegura

Gestión manual de memoria dinámica con `new` y `delete` para trabajar con un recurso en memoria. Se pretende mostrar cómo un flujo con múltiples puntos de salida o errores puede provocar fugas de memoria cuando no se libera el recurso correctamente.

```
#include <iostream>
#include <stdexcept>

//-- Función que procesa datos usando memoria dinámica
void procesarDatos(bool generarError) {
  //-- Reserva manual de memoria
  int* datos = new int[5];

  //-- Inicialización del array
  for (int i = 0; i < 5; ++i) {
    datos[i] = i * 10;
  }

  //-- Simulación de error que interrumpe el flujo
  if (generarError) {
    throw std::runtime_error("Error durante el procesamiento");
  }

  //-- Liberación manual del recurso
  delete[] datos;
}
```

```
int main() {
  try {
    procesarDatos(true);
  }
  catch (const std::exception& e) {
    std::cout << "Excepción capturada: " << e.what() << std::endl;
  }

  //-- Si ocurre una excepción antes del delete, hay fuga de memoria
  return 0;
}
```

Codificación recomendada

Gestión automática de memoria utilizando el principio RAII mediante `std::unique_ptr`. Se busca demostrar cómo el recurso se libera automáticamente al salir del ámbito, incluso cuando se producen excepciones, evitando fugas de memoria.

```
#include <iostream>
#include <memory>
#include <stdexcept>

//-- Función que procesa datos usando RAII
void procesarDatos(bool generarError) {
  //-- Reserva de memoria gestionada automáticamente
  std::unique_ptr<int[]> datos(new int[5]);

  //-- Inicialización del array
  for (int i = 0; i < 5; ++i) {
    datos[i] = i * 10;
  }

  //-- Simulación de error que interrumpe el flujo
  if (generarError) {
    throw std::runtime_error("Error durante el procesamiento");
  }

  //-- No es necesario liberar manualmente el recurso
}

int main() {
  try {
    procesarDatos(true);
  }
  catch (const std::exception& e) {
    std::cout << "Excepción capturada: " << e.what() << std::endl;
  }
```

```
    //-- La memoria se libera automáticamente al salir del ámbito
    return 0;
}
```

2.13.1.1 Descripción

El propósito de la **RAII** es garantizar que todo recurso adquirido por un objeto sea liberado automáticamente al finalizar su ciclo de vida, sin necesidad de una intervención manual por parte del desarrollador. Este mecanismo resulta esencial para prevenir un amplio abanico de vulnerabilidades, especialmente aquellas relacionadas con fugas de memoria, corrupciones de estado, liberaciones incorrectas o dobles liberaciones, entre otros fallos que pueden comprometer la integridad y estabilidad del software.

Aplicar RAII implica encapsular cada recurso en una clase cuya responsabilidad sea adquirir el recurso en su constructor y liberarlo en su destructor. Este patrón garantiza que, incluso en escenarios complejos marcados por excepciones, bifurcaciones de flujo o terminaciones abruptas del programa, los recursos asociados a cada objeto gestionado correctamente se liberen de forma determinista. En entornos de seguridad, donde la coherencia del estado y la predictibilidad del comportamiento son fundamentales, RAII proporciona una herramienta eficiente y confiable para la gestión adecuada de recursos.

Entre las ventajas más relevantes se encuentran la mitigación de fugas de memoria, la prevención de dobles eliminaciones, el saneamiento de estados inconsistentes y la imposibilidad de omitir accidentalmente la liberación de un recurso. Estas situaciones son especialmente críticas en aplicaciones orientadas a sistemas embebidos, servidores críticos o cualquier contexto donde fallos de memoria puedan derivar en incidentes graves de disponibilidad o integridad. Además, RAII no requiere mecanismos complejos adicionales ni genera sobrecostes significativos a nivel de ejecución, lo que lo convierte en un patrón extremadamente eficiente.

C++ provee construcciones de alto nivel que implementan RAII de forma nativa, como `std::unique_ptr`, `std::shared_ptr`, `std::lock_guard`, `std::scoped_lock` y `std::fstream`. Estas clases deben ser la opción preferente en la mayor parte de los casos, dado que representan implementaciones probadas, optimizadas y seguras. Por ejemplo, el uso de std::unique_ptr previene automáticamente fugas, facilita la transferencia segura de propiedad y evita errores comunes al manipular punteros crudos.

Es necesario evitar prácticas que rompan la filosofía de RAII, como la liberación manual del recurso, el uso excesivo de punteros brutos o el control disperso del ciclo de vida. Estas prácticas suelen originar dificultades en la trazabilidad del código y facilitan escenarios inseguros que pueden derivar en vulnerabilidades de integridad o corrupciones de datos complejas de depurar.

Riesgos asociados

- Fugas de memoria por no liberar recursos asignados dinámicamente.

- Pérdida de recursos del sistema (descriptores de archivos, sockets, memoria compartida) debido a su no liberación.
- Condiciones de carrera y bloqueos si los recursos no se liberan correctamente en presencia de excepciones.
- Violaciones de integridad de datos por acceso a memoria o recursos ya liberados (use-after-free).
- Comportamiento indefinido al acceder a recursos en estados inconsistentes.
- Incremento en la superficie de ataque para exploits de denegación de servicio (DoS) debido al agotamiento de recursos.
- Dificultad en el mantenimiento y prueba del código debido a la gestión manual y dispersa de recursos.
- Falta de robustez frente a errores, aumentando la probabilidad de caídas inesperadas.
- Exposición a vulnerabilidades por no cumplir el principio de excepción segura (strong exception guarantee).

Buenas prácticas

- Utilizar objetos automáticos y smart pointers en lugar de gestionar memoria manualmente con `new`/`delete`.

```
//-- Reserva y liberación manual de memoria
void procesar() {
  int* datos = new int[100];
  // uso de datos
  delete[] datos;
}
```

```
//-- Uso de std::vector para gestión automática del recurso
void procesar() {
  std::vector datos(100);
  // uso de datos
}
```

- Emplear `std::unique_ptr` o `std::shared_ptr` para expresar propiedad y evitar fugas de memoria.

```
//-- Puntero crudo retornado sin claridad de propiedad
Foo* crear() {
  return new Foo();
}
```

```
//-- Uso de std::unique_ptr para transferir propiedad de forma segura
std::unique_ptr crear() {
  return std::make_unique();
}
```

- Encapsular recursos del sistema (archivos, sockets, mutex) en clases que liberen el recurso en su destructor.

```
//-- Apertura y cierre manual de archivo
void escribir() {
  FILE* f = fopen("datos.txt", "w");
  if (f) {
    fputs("hola", f);
    fclose(f);
  }
}
```

```
//-- Uso de std::ofstream que libera el recurso automáticamente
void escribir() {
  std::ofstream f("datos.txt");
  if (f) {
    f << "hola";
  }
}
```

2.13.1.2 Estándares metodológicos

- **C++ Core-R.1**: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization).

2.13.2 [cpp.res.2] - Transferir al RAII toda asignación explícita

ID	cpp.res.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Design

Evitar asignaciones y liberaciones de memoria explícitas mediante `new` y `delete`, reduce las fugas de memoria, dobles liberaciones, uso después de liberar y otras condiciones inseguras derivadas de la gestión manual. Al obligar a que los recursos de memoria, archivos, descriptores y otros elementos sensibles se administren a través de objetos con semántica de vida bien definida, se mejora la seguridad, la estabilidad y la resiliencia del software frente a fallos inesperados y a comportamientos indefinidos.

Codificación insegura

Gestión manual de memoria dinámica usando `new` y `delete` para crear y liberar un recurso. Se busca mostrar cómo una asignación explícita puede provocar fugas de memoria cuando existen retornos tempranos o rutas de error que omiten la liberación del recurso.

```
#include <iostream>
#include <cstring>

//-- Reserva memoria manualmente y puede provocar fuga si hay retorno temprano
char* crearBuffer(size_t tam) {
  //-- Asignación explícita en el heap
  char* buffer = new char[tam];

  //-- Simulación de error de validación
  if (tam < 10) {
    //-- Retorno sin liberar memoria previamente asignada

    return nullptr;
  }

  std::strcpy(buffer, "Ejemplo");
  return buffer;
}

int main() {
  //-- Solicitud de buffer pequeño que genera fuga
  char* datos = crearBuffer(5);

  if (datos != nullptr) {
    std::cout << datos << std::endl;

    //-- Liberación manual del recurso

    delete[] datos;
  }

  return 0;
}
```

Codificación recomendada

Uso del principio RAII mediante contenedores estándar para transferir la responsabilidad de la gestión del recurso al objeto. Se pretende mostrar cómo la liberación automática evita fugas incluso cuando existen retornos tempranos o excepciones.

```
#include <iostream>
#include <vector>
#include <cstring>

//-- Uso de RAII con contenedor estándar
std::vector<char> crearBuffer(size_t tam) {
  //-- El recurso queda gestionado por el objeto vector
  std::vector<char> buffer(tam);

  //-- Validación con retorno temprano sin riesgo de fuga
  if (tam < 10) {
    return {};
  }

  std::strcpy(buffer.data(), "Ejemplo");
  return buffer;
}

int main() {
  //-- El objeto gestiona automáticamente la memoria
  std::vector<char> datos = crearBuffer(5);

  if (!datos.empty()) {
    std::cout << datos.data() << std::endl;
  }

  //-- No es necesario liberar manualmente

  return 0;
}
```

2.13.2.1 Descripción

Este requisito establece trasladar toda asignación explícita mediante `new`, `new[]`, `malloc`, o cualquier otro mecanismo manual, hacia constructores de clases que representen la propiedad de dichos recursos, confiando en su destructor para liberarlos automáticamente. Este patrón ofrece una estrategia determinista que se vuelve especialmente relevante en contextos de desarrollo seguro, donde el riesgo derivado de una gestión deficiente de memoria puede dar lugar a vulnerabilidades graves como fugas persistentes, corrupción de memoria, uso de memoria ya liberada o ejecución de código arbitrario.

En el enfoque RAII, cuando un objeto entra en un ámbito, el recurso es adquirido; cuando abandona el ámbito, el recurso es liberado. La clave de esta regla está en eliminar la intervención manual en la gestión de memoria evitando errores humanos y combinando seguridad con eficiencia. Esto se puede realizar usando **smart pointers** como `std::unique_ptr` y `std::shared_ptr`, o contenedores como `std::vector` y

`std::string`, que internamente administran la memoria sin exponer mecanismos manuales en el código cliente.

Riesgos asociados

- Fugas de memoria por no liberar recursos asignados manualmente.
- Condiciones de carrera al manipular recursos compartidos sin control adecuado.
- Acceso a recursos liberados prematuramente (use-after-free).
- Pérdida de información sobre la propiedad y el ciclo de vida de los recursos.
- Complejidad innecesaria en el manejo de excepciones y limpieza manual.
- Errores de doble liberación (double free) al intentar liberar explícitamente recursos ya gestionados.
- Vulnerabilidades de seguridad por manejo inadecuado de recursos sensibles (archivos, sockets, etc.).
- Difícil mantenimiento y evolución del código debido a la dispersión del control de recursos.
- Inconsistencia en la liberación de recursos en rutas de error.

Buenas prácticas

- Utilizar objetos automáticos y smart pointers en lugar de `new`/`delete` explícitos.

```
//-- Uso de new/delete manual con riesgo de fuga ante excepciones
void procesar() {
  Recurso* r = new Recurso();
  r->usar();
  delete r;
}
```

```
//-- Uso de std::unique_ptr para liberar automáticamente el recurso
void procesar() {
  std::unique_ptr r = std::make_unique();
  r->usar();
}
```

- Encapsular recursos del sistema (archivos, sockets, mutex) en clases RAII.

```
//-- Apertura y cierre manual de archivo
void escribir() {
  FILE* f = fopen("datos.txt", "w");
  fprintf(f, "hola");
  fclose(f);
```

```
}
```

```
//-- Uso de std::ofstream que cierra automáticamente el archivo
void escribir() {
  std::ofstream f("datos.txt");
  f << "hola";
}
```

- Evitar gestión manual de memoria dinámica usando contenedores estándar.

```
//-- Array dinámico con liberación manual
void cargar() {
  int* datos = new int[100];
  datos[0] = 1;
  delete[] datos;
}
```

```
//-- Uso de std::vector con liberación automática
void cargar() {
  std::vector datos(100);
  datos[0] = 1;
}
```

- Aplicar RAII para gestión de bloqueos en concurrencia.

```
//-- Bloqueo y desbloqueo manual de mutex
void tarea(std::mutex& m) {
  m.lock();
  procesoCritico();
  m.unlock();
}
```

```
//-- Uso de std::lock_guard para liberar el mutex automáticamente
void tarea(std::mutex& m) {
  std::lock_guard lock(m);
  procesoCritico();
}
```

2.13.2.2 Estándares metodológicos

- **C++ Core-R.12**: Immediately give the result of an explicit resource allocation to a manager object.

2.13.3 [cpp.res.3] - Los archivos deben cerrarse cuando ya no sean necesarios

ID	cpp.res.3	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Este requisito establece la obligación de asegurar que todos los archivos abiertos por un programa sean cerrados de forma consistente, explícita o automática, tan pronto como ya no se necesiten. El cierre adecuado de archivos evita problemas como fugas de recursos, corrupción de datos, condiciones de carrera, accesos concurrentes indebidos y pérdida de información. También permite reducir el riesgo de denegación de servicio por agotamiento de descriptores.

Codificación insegura

Gestión básica de archivos donde se abre un archivo para escribir información y finalizar el programa. Se busca observar el manejo del recurso archivo durante la ejecución y qué ocurre cuando no se libera explícitamente antes de terminar el proceso.

```
#include <iostream>
#include <cstdio>

int main() {
  //-- Se abre un archivo para escritura
  FILE* file = fopen("salida.txt", "w");

  if (file == nullptr) {
    //-- Se valida si el archivo no pudo abrirse
    std::cerr << "Error al abrir el archivo" << std::endl;
    return 1;
  }

  //-- Se escribe información en el archivo
  fputs("Linea de prueba\n", file);

  //-- No se cierra el archivo explícitamente
  //-- Esto puede provocar fugas de recursos en aplicaciones mas complejas

  return 0;
}
```

Codificación recomendada

Gestión adecuada de archivos donde se abre un archivo para escribir información y se libera explícitamente el recurso cuando ya no es necesario. Se busca asegurar la liberación controlada del descriptor para evitar fugas y comportamientos inesperados.

```
#include <iostream>
#include <cstdio>

int main() {
  //-- Se abre un archivo para escritura
  FILE* file = fopen("salida.txt", "w");

  if (file == nullptr) {
    //-- Se valida si el archivo no pudo abrirse
    std::cerr << "Error al abrir el archivo" << std::endl;
    return 1;
  }

  //-- Se escribe información en el archivo
  fputs("Linea de prueba\n", file);

  //-- Se cierra el archivo inmediatamente despues de usarlo
  if (fclose(file) != 0) {
    //-- Se valida si hubo error al cerrar el archivo
    std::cerr << "Error al cerrar el archivo" << std::endl;
    return 1;
  }

  //-- El recurso fue liberado correctamente
  return 0;
}
```

2.13.3.1 Descripción

La gestión de archivos implica la manipulación explícita de recursos del sistema operativo, como descriptores de archivo y buffers de E/S. Un error frecuente en aplicaciones de todos los tamaños es dejar archivos abiertos durante demasiado tiempo o, peor aún, no cerrarlos en absoluto. Esto puede ocasionar fugas de recursos, corrupción de datos o fallos impredecibles en condiciones de carga. El requisito establece que todo archivo debe cerrarse en el momento exacto en que ha dejado de ser necesario y que el diseño debe garantizar este comportamiento incluso en presencia de excepciones, errores lógicos o caminos de ejecución complejos. Para ello, las técnicas de RAII, el uso de clases contenedoras seguras y el control estricto del ciclo de vida de los objetos son herramientas esenciales.

El cierre seguro de archivos se apoya en los principios tradicionales de gestión de recursos del lenguaje, como es el RAII. RAII asegura que, al destruirse el objeto, el recurso se libera automáticamente. Esto evita depender de llamadas manuales a `close` y disminuye la probabilidad de errores humanos, especialmente en aplicaciones complejas o con múltiples rutas de salida.

C++ proporciona mecanismos de alto nivel que permiten automatizar la gestión segura de archivos. La clase `std::fstream` y sus variantes permiten apoyarse en RAII: cuando el objeto sale de ámbito, su destructor cierra el archivo de manera segura. Sin embargo, cuando se manejan directamente descriptores del sistema operativo mediante funciones de bajo nivel como `open` o `fopen`, el programador debe garantizar manualmente el cierre utilizando `close` o `fclose`. En estos casos, resulta fundamental encapsular los descriptores dentro de clases que implementen un destructor responsable del cierre seguro.

Cuando se emplean APIs POSIX, es recomendable encapsular el descriptor dentro de una clase que garantice su cierre en el destructor. Este enfoque reduce errores comunes como retornos prematuros o excepciones que omiten manualmente la llamada a `close`. Además, encapsular el descriptor permite aplicar reglas de copia y movimiento que eviten duplicar manejadores, generando un control mucho más robusto sobre el ciclo de vida.

Riesgos asociados

- Fugas de recursos del sistema debido a la acumulación de archivos abiertos.
- Agotamiento del límite de descriptores de archivos, lo que puede impedir la apertura de nuevos archivos o servicios críticos.
- Pérdida de datos si la información almacenada en buffers no se escribe correctamente en disco al no cerrar el archivo.
- Condiciones de carrera o corrupción de datos, ya que otros procesos pueden quedar a la espera de que se cierre el archivo.
- Comportamiento indefinido del programa por exceder recursos permitidos por el sistema operativo.
- Exposición a ataques de denegación de servicio (DoS) si un atacante fuerza la apertura repetida y no cierre de archivos.
- Imposibilidad de acceder o modificar el archivo por otros procesos debido a bloqueos persistentes.
- Incremento en la complejidad de depuración por la gestión incorrecta del ciclo de vida de los archivos.

Buenas prácticas

- Cerrar explícitamente los archivos cuando dejan de utilizarse.

```
//-- Apertura sin cierre explícito del archivo
void procesar() {
  std::ofstream file("datos.txt");
  file << "Información";
```

```
}
```

```
//-- Cierre explícito después de finalizar la escritura
void procesar() {
  std::ofstream file("datos.txt");
  file << "Información";
  file.close();
}
```

- Verificar que el archivo esté abierto antes de usarlo y cerrarlo en rutas de error.

```
//-- Uso del archivo sin validación ni cierre en caso de error
void leer() {
  std::ifstream file("config.txt");
  std::string linea;
  std::getline(file, linea);
}
```

```
//-- Validación del estado y cierre controlado ante fallo
void leer() {
  std::ifstream file("config.txt");
  if (!file.is_open()) {
    return;
  }
  std::string linea;
  std::getline(file, linea);
  file.close();
}
```

- Utilizar RAII y limitar el alcance del archivo para garantizar su cierre automático.

```
//-- Archivo con alcance amplio que permanece abierto más tiempo del necesario
std::ofstream file;

void escribir() {
  file.open("log.txt");
  file << "Evento";
}
```

```
//-- Archivo declarado en un alcance reducido para cierre automático
void escribir() {
  std::ofstream file("log.txt");
  file << "Evento";
}
```

2.13.3.2 Estándares metodológicos

- **CERT/CPP-FIO51**: Close files when they are no longer needed.

2.14 Gestión de errores

2.14.1 [cpp.err.1] - Gestionar todas las excepciones adecuadamente

ID	cpp.err.1	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Design

Se deben gestionar los errores y las excepciones adecuadamente con el fin de evitar comportamientos indefinidos, filtrado de información sensible y fallos inesperados en tiempo de ejecución. Se incluyen técnicas para validar condiciones de error, lanzar únicamente excepciones seguras, capturarlas de forma estructurada y asegurar que los recursos sean liberados correctamente mediante RAII. Además, se deben evitar las excepciones no documentadas, desbordamientos silenciosos, estados incoherentes del sistema y fugas de datos a través de mensajes de error inseguros.

Codificación insegura

Programa que realiza una conversión de texto a número y una división. Se busca mostrar cómo se gestionan errores en tiempo de ejecución cuando se producen excepciones, permitiendo observar el impacto de no controlarlas adecuadamente.

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

int convertirYDividir(const std::string& valor, int divisor) {
  //-- convierte el texto a entero usando stoi
  int numero = std::stoi(valor);

  //-- realiza una division que puede generar error si divisor es cero
  return numero / divisor;
}

int main() {
  //-- valores de entrada potencialmente invalidos
  std::string entrada = "abc";
  int divisor = 0;

  //-- llamada sin gestion de excepciones
  int resultado = convertirYDividir(entrada, divisor);

  std::cout << "Resultado: " << resultado << std::endl;
  return 0;
```

```
}
```

Codificación recomendada

Programa que realiza la misma conversión y división, incorporando manejo explícito de excepciones estándar y validaciones adicionales. Se pretende mostrar cómo controlar errores esperados y no esperados para evitar la finalización abrupta del proceso y mejorar la robustez.

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

int convertirYDividir(const std::string& valor, int divisor) {
  //-- valida division por cero antes de operar
  if (divisor == 0) {
    throw std::invalid_argument("Divisor no puede ser cero");
  }

  //-- convierte el texto a entero y puede lanzar excepcion
  int numero = std::stoi(valor);

  return numero / divisor;
}

int main() {
  std::string entrada = "abc";
  int divisor = 0;

  //-- bloque try-catch para capturar errores controlados
  try {
    int resultado = convertirYDividir(entrada, divisor);
    std::cout << "Resultado: " << resultado << std::endl;
  } catch (const std::invalid_argument& e) {
    //-- gestiona errores de argumentos invalidos
    std::cerr << "Error de argumento: " << e.what() << std::endl;
  } catch (const std::exception& e) {
    //-- captura cualquier otra excepcion estandar
    std::cerr << "Error general: " << e.what() << std::endl;
  } catch (...) {
    //-- captura excepciones no previstas
    std::cerr << "Error desconocido" << std::endl;
  }

  return 0;
}
```

2.14.1.1 Descripción

El manejo incorrecto de excepciones puede provocar estados inconsistentes en memoria, pérdidas de recursos, interrupciones inesperadas del programa, exposición de información sensible o rutas de ejecución no controladas que pueden ser aprovechadas por atacantes. Este requisito se basa en principios de programación defensiva y otras recomendaciones de ingeniería de software seguro orientadas a minimizar riesgos asociados a fallos no gestionados.

C++ posee un mecanismo de excepciones flexible, pero este poder implica responsabilidad. Cuando no se gestionan correctamente, las excepciones pueden escapar de límites previstos, romper invariantes de clases, causar fugas de memoria o finalizar abruptamente el programa mediante `std::terminate`. Un atacante puede explotar estos comportamientos para desestabilizar el sistema, activar condiciones de denegación de servicio o provocar ejecuciones incorrectas. Por ello es fundamental definir reglas estrictas para lanzar, propagar y capturar excepciones.

En sistemas complejos es común combinar excepciones con mecanismos alternativos de gestión de errores. Es importante mantener coherencia entre ambos enfoques. Cuando se utilicen códigos de retorno, deben integrarse adecuadamente en un marco de control que permita identificar errores críticos y convertirlos en excepciones cuando corresponda. También es recomendable centralizar la captura de excepciones en puntos definidos de la arquitectura, como en bucles de ejecución principales o puertas de entrada de módulos, para evitar que errores internos escapen sin control.

Las bibliotecas que se diseñen para terceros deben proporcionar garantías claras sobre qué funciones lanzan excepciones y en qué condiciones. Asimismo, es fundamental asegurar que las APIs no filtren información sensible en los mensajes de error; los detalles internos deben registrarse solo en entornos controlados, utilizando **registros auditables** protegidos.

En entornos de alta seguridad o donde la disponibilidad es crítica, la gestión de excepciones debe ir acompañada de estrategias de aislamiento para evitar que un fallo en un módulo afecte a otros componentes. Patrones como **circuit breakers**, contenedores de ejecución o límites de recursos pueden reforzar la resiliencia del sistema ante errores excepcionales.

Riesgos asociados

- Pérdida de control del flujo del programa, lo que puede provocar comportamientos inesperados o fallos.
- Divulgación de información sensible a través de mensajes de error no controlados.
- Condiciones de denegación de servicio (DoS) debido a caídas abruptas de la aplicación.

- Fugas de memoria y recursos al no liberar correctamente objetos tras una excepción no gestionada.
- Corrupción de datos o estados inconsistentes al no completar transacciones o cambios por manejo inadecuado de errores.
- Exposición a vulnerabilidades como ejecución de código arbitrario si el flujo salta a áreas inseguras por excepciones no controladas.
- Pérdida de trazabilidad y dificultad para identificar la causa raíz de fallos debido a excepciones no registradas ni informadas adecuadamente.
- Degradación de la experiencia del usuario por comportamientos erráticos o caídas inesperadas.

Buenas prácticas

- Capturar excepciones específicas antes que las genéricas para evitar ocultar errores críticos.

```
try {
  procesarArchivo();
} catch (...) {
  //-- Manejo genérico que oculta el detalle del error
  std::cerr << "Error desconocido";
}
```

```
try {
  procesarArchivo();
} catch (const std::ios_base::failure& e) {
  //-- Manejo específico con información detallada
  std::cerr << e.what();
} catch (const std::exception& e) {
  //-- Manejo controlado para otras excepciones estándar
  std::cerr << e.what();
}
```

- Evitar capturar excepciones por valor; capturarlas por referencia constante para prevenir copias y slicing.

```
try {
  operacionCritica();
} catch (std::exception e) {
  //-- Captura por valor que puede producir slicing
  registrar(e.what());
}
```

```
try {
```

```
  operacionCritica();
} catch (const std::exception& e) {
  //-- Captura por referencia constante preservando la jerarquía
  registrar(e.what());
}
```

- No ignorar excepciones; registrar, propagar o transformar adecuadamente el error.

```
void servicio() {
  try {
    accederRecurso();
  } catch (const std::exception&) {
    //-- Excepción ignorada sin trazabilidad
  }
}
```

```
void servicio() {
  try {
    accederRecurso();
  } catch (const std::exception& e) {
    //-- Registro y propagación controlada del error
    registrar(e.what());
    throw;
  }
}
```

- Garantizar la liberación segura de recursos utilizando RAII para evitar fugas ante excepciones.

```
void procesar() {
  Recurso* r = new Recurso();
  //-- Uso manual que puede provocar fuga si ocurre excepción
  usar(r);
  delete r;
}
```

```
void procesar() {
  std::unique_ptr<Recurso> r(new Recurso());
  //-- Gestión automática del recurso ante cualquier excepción
  usar(r.get());
}
```

2.14.1.2 Estándares metodológicos

- **CERT/CPP-ERR51**: Handle all exceptions.

2.14.2 [cpp.err.2] - Asegurar la liberación de recursos al gestionar excepciones

ID	cpp.err.2	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se deben implementar mecanismos seguros que aseguren la liberación correcta de recursos del sistema cuando se producen excepciones. Una gestión deficiente puede ocasionar pérdidas de memoria, fugas de descriptores y bloqueos que comprometen la estabilidad y seguridad de la aplicación. Aplicar patrones como RAII y asegurar que todos los objetos administren adecuadamente sus recursos evita situaciones peligrosas donde un flujo de ejecución excepcional interrumpe la liberación prevista.

Codificación insegura

Gestión manual de memoria dinámica donde se reserva un recurso y se produce una excepción antes de liberarlo. Se busca mostrar cómo una excepción puede interrumpir el flujo normal y provocar fugas de memoria cuando no se asegura la liberación del recurso.

```
#include <iostream>
#include <stdexcept>

void procesar() {
  //-- Reserva dinámica de memoria
  int* datos = new int[5];

  //-- Simulación de una condición de error
  if (true) {
    //-- Se lanza una excepción antes de liberar la memoria
    throw std::runtime_error("Error durante el procesamiento");
  }

  //-- Liberación manual del recurso (nunca se ejecuta)
  delete[] datos;
}

int main() {
  //-- Manejo básico de la excepción
  try {
    procesar();
  }
  catch (const std::exception& e) {
    std::cout << "Excepcion capturada: " << e.what() << std::endl;
  }

  //-- El programa finaliza pero la memoria no fue liberada
```

```
  return 0;
}
```

Codificación recomendada

Gestión automática de recursos utilizando el principio RAII mediante `std::unique_ptr`, garantizando que la memoria se libere automáticamente cuando ocurre una excepción. Se busca demostrar cómo asegurar la liberación de recursos incluso cuando el flujo se interrumpe por errores.

```
#include <iostream>
#include <memory>
#include <stdexcept>

void procesar() {
  //-- Reserva dinámica gestionada por RAII
  std::unique_ptr<int[]> datos(new int[5]);

  //-- Simulación de una condición de error
  if (true) {
    //-- Se lanza una excepción pero el recurso será liberado automáticamente
    throw std::runtime_error("Error durante el procesamiento");
  }

  //-- No es necesario liberar manualmente el recurso
}

int main() {
  //-- Manejo básico de la excepción
  try {
    procesar();
  }
  catch (const std::exception& e) {
    std::cout << "Excepcion capturada: " << e.what() << std::endl;
  }

  //-- La memoria fue liberada automáticamente al salir del alcance
  return 0;
}
```

2.14.2.1 Descripción

La complejidad asociada a las excepciones cuando interrumpen secuencias de liberación de recursos manuales obliga a adoptar estrategias defensivas robustas. El uso del patrón **RAII** es fundamental para garantizar que la adquisición de recursos esté intrínsecamente ligada a su liberación automática, independientemente de si el flujo de ejecución concluye de manera normal o por una excepción inesperada.

Cuando un recurso se adquiere usando llamadas directas a APIs del sistema, asignaciones de memoria con `new`, gestión manual de buffers o utilización de descriptores de archivos, existe el riesgo de que una excepción generada entre la adquisición y la liberación haga que el recurso permanezca en memoria o quede asociado al proceso de manera indefinida.

El uso de RAII debe considerarse obligatorio y no una recomendación. Este enfoque asegura que cualquier recurso sea liberado de forma automática al abandonar el ámbito, incluso cuando se produce una excepción. Las clases estándar como `std::unique_ptr`, `std::shared_ptr`, `std::lock_guard` o `std::fstream` ya implementan este patrón, pero también es responsabilidad del desarrollador crear sus propias envolturas RAII cuando trabaja con recursos no gestionados.

Riesgos asociados

- Fugas de memoria por no liberar memoria dinámica al ocurrir una excepción.
- Bloqueo o interbloqueo al no liberar mutexes, semáforos u otros sincronizadores.
- Pérdida de archivos o inconsistencias de estado al no cerrar correctamente archivos abiertos.
- Agotamiento de recursos del sistema, como descriptores de archivo o memoria, al no liberar recursos.
- Comportamiento inesperado del programa debido a la corrupción del estado interno (recursos no inicializados o no restablecidos).
- Dificultad para depurar y mantener el código debido a recursos que permanecen activos tras una excepción.
- Vulnerabilidad a ataques de denegación de servicio (DoS) al permitir acumulación de recursos no liberados.
- Pérdida de datos importantes al no realizar operaciones de limpieza/cierre necesarias tras excepciones.
- Inestabilidad general del sistema o de la aplicación por el uso prolongado de recursos no liberados.

Buenas prácticas

- Aplicar el principio RAII para que la liberación de recursos ocurra automáticamente al salir del ámbito, incluso ante excepciones.

```
//-- Reserva manual sin objeto RAII

void procesar() {
  char* buffer = new char[1024];
```

```
  realizarOperacion();
  delete[] buffer;
}
```

```
//-- Uso de contenedor RAII para liberar automáticamente
void procesar() {
  std::vector buffer(1024);
  realizarOperacion();
}
```

- Utilizar punteros inteligentes en lugar de punteros crudos para gestionar memoria dinámica.

```
//-- Uso de puntero crudo sin liberación garantizada ante excepción
void procesar() {
  Recurso* r = new Recurso();
  r->usar();
  delete r;
}
```

```
//-- Uso de std::unique_ptr para liberación automática
void procesar() {
  std::unique_ptr r = std::make_unique();
  r->usar();
}
```

- Emplear objetos de bloqueo automáticos para liberar recursos del sistema (como `mutex`) al producirse excepciones.

```
//-- Bloqueo y desbloqueo manual sin garantía ante excepción
std::mutex m;
void procesar() {
  m.lock();
  operacionCritica();
  m.unlock();
}
```

```
//-- Uso de std::lock_guard para liberar el mutex automáticamente
std::mutex m;
void procesar() {
  std::lock_guard lock(m);
  operacionCritica();
}
```

- Encapsular recursos en clases con destructores que garanticen la liberación

segura ante excepciones.

```
//-- Apertura de archivo sin cierre garantizado
void procesar() {
  FILE* f = fopen("datos.txt", "r");
  leerDatos(f);
  fclose(f);
}
```

```
//-- Uso de std::ifstream que cierra el archivo en su destructor
void procesar() {
  std::ifstream f("datos.txt");
  leerDatos(f);
}
```

2.14.2.2 Estándares metodológicos

- **CERT/CPP-ERR57**: Do not leak resources when handling exceptions.

2.14.3 [cpp.err.3] - Evitar lanzar excepciones a través de límites de ejecución

ID	cpp.err.3	Uso	Recomendado
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se debe impedir que las excepciones crucen límites de ejecución tales como interfaces binarias, callbacks, controladores externos, APIs del sistema u otros contextos donde no exista un contrato explícito que garantice el manejo seguro de dichas excepciones. Cuando una excepción atraviesa estos límites, puede provocar corrupción de memoria, finalización inesperada del proceso, fuga de información, fallos de disponibilidad o estados inconsistentes.

Codificación insegura

Uso de hilos donde una función ejecutada en un hilo secundario puede generar una excepción. Se busca mostrar cómo el flujo de una excepción a través del límite de ejecución (thread boundary) impacta el comportamiento del programa y qué ocurre cuando no se gestiona correctamente dentro del propio hilo.

```
#include <iostream>
#include <thread>
#include <stdexcept>

//-- Función que simula una operación que falla
void tarea() {
  //-- Se lanza una excepción que no será capturada dentro del hilo
```

```
  throw std::runtime_error("Error en el hilo");
}

int main() {
  //-- Se crea un hilo que ejecuta una función que lanza una excepción
  std::thread hilo(tarea);

  //-- Se espera a que el hilo termine
  hilo.join();

  //-- Esta línea nunca se ejecutará porque el programa terminará con
std::terminate
  std::cout << "Fin del programa" << std::endl;

  return 0;
}
```

Codificación recomendada

Uso de hilos donde la función ejecutada puede fallar, pero la excepción se captura dentro del propio hilo y se comunica el error de forma controlada al hilo principal. Se busca aprender a evitar que las excepciones crucen límites de ejecución y provoquen terminaciones inesperadas.

```
#include <iostream>
#include <thread>
#include <stdexcept>
#include <future>

//-- Función que simula una operación que puede fallar
void tarea(std::promise<std::string> resultado) {
  try {
    //-- Se simula un error durante la ejecución
    throw std::runtime_error("Error en el hilo");
  }
  catch (const std::exception& e) {
    //-- Se captura la excepción dentro del hilo y se comunica el error
    resultado.set_value(std::string("Fallo controlado: ") + e.what());
    return;
  }

  //-- Si no hay error, se informa éxito
  resultado.set_value("Ejecución correcta");
}

int main() {
  //-- Se prepara un mecanismo seguro para recibir el resultado del hilo
  std::promise<std::string> promesa;
```

```
    std::future<std::string> futuro = promesa.get_future();

    //-- Se lanza el hilo pasando la promesa por movimiento
    std::thread hilo(tarea, std::move(promesa));

    //-- Se espera la finalización del hilo
    hilo.join();

    //-- Se obtiene el resultado de forma controlada
    std::cout << futuro.get() << std::endl;

    return 0;
}
```

2.14.3.1 Descripción

Cuando una excepción cruza un límite de ejecución no preparado para recibirla, como una interfaz de código nativo, una frontera ABI, funciones callback hacia bibliotecas externas, o componentes escritos en otros lenguajes, se pueden generar fallos graves en tiempo de ejecución. Estos fallos se deben a que dichos entornos no conocen el mecanismo de unwinding de C++ ni garantizan la preservación del estado requerido para continuar ejecutando el programa de manera coherente. Para evitar estos riesgos, es necesario establecer políticas estrictas que limiten la propagación de excepciones más allá de los puntos de control definidos. El **unwinding** es el proceso que ocurre cuando se lanza una excepción y el programa **deshace la pila de llamadas**.

El riesgo principal proviene de la destrucción incorrecta de recursos durante el proceso de unwinding cuando la excepción escapa hacia código que no está preparado para manejarla. Esto puede originar fugas de memoria, invocación incompleta de destructores o corrupción de estructuras internas. En contextos de seguridad, estos problemas pueden desembocar en bloqueos del sistema, cierre inesperado de servicios expuestos a red o inconsistencias en operaciones críticas.

En ambientes de interoperabilidad, como la exposición de funcionalidades mediante `extern "C"`, se requiere evitar que cualquier excepción abandone el límite del lenguaje C++, ya que el lenguaje C carece de mecanismos para procesarla. Asimismo, cuando el código C++ se integra con bibliotecas o runtimes que no utilizan el manejador de excepciones propio del lenguaje, la propagación no controlada puede producir comportamientos indefinidos. Por este motivo, es esencial realizar un encapsulamiento explícito del código que podría lanzar excepciones y convertirlas en códigos de error seguros o valores de retorno definidos.

Una práctica habitual consiste en delimitar secciones de código potencialmente inseguras mediante bloques `try`/`catch` internos, garantizando que antes de abandonar una función expuesta, toda excepción ha sido manejada adecuadamente. Con frecuencia, estas capturas deben ser generales (`catch(...)`) para evitar que excepciones

desconocidas crucen el límite. Posteriormente, el error debe traducirse a un tipo de salida conocido y documentado por la interfaz.

El diseño de APIs internas también debe tener en cuenta estos riesgos. Cuando una función se define como no lanzadora de excepciones, ya sea mediante `noexcept` o mediante documentación, se vuelve crítico asegurarse de que cualquier excepción interna sea tratada adecuadamente antes de salir de la función. La violación de `noexcept` desencadena la finalización del programa, lo que puede comprometer la disponibilidad del sistema.

La interoperabilidad con hilos gestionados por frameworks externos, sistemas embebidos, controladores o bibliotecas sin soporte para excepciones exige un enfoque disciplinado basado en convertir cualquier condición excepcional en mecanismos alternativos como valores centinela, tipos de retorno con estructuras seguras, indicadores booleanos o códigos de error uniformes. **Un diseño coherente de errores** facilita evitar que excepciones se propaguen fuera de los límites previstos, permitiendo mejorar la robustez general del sistema.

Para interfaces externas, el diseño debe dejar claro que **ninguna excepción propagada es aceptable**. Las pruebas deben incluir escenarios donde el código interno genere excepciones intencionalmente, verificando que la frontera de ejecución las captura. Además, cuando se implementan callbacks invocados desde código externo, es obligatorio garantizar que ninguna excepción cruza hacia el llamador, utilizando técnicas similares a las mencionadas.

Riesgos asociados

- Pérdida de recursos y memory leaks por no ejecutar correctamente destructores o manejadores de recursos.
- Inestabilidad de la aplicación debido a estados inconsistentes cuando no se completa la limpieza de objetos o recursos.
- Comportamiento indefinido si la excepción cruza límites de ejecución donde no se puede manejar correctamente (por ejemplo, a través de código C puro o APIs de terceros que no soportan excepciones).
- Interrupción inesperada del flujo de control que puede dejar operaciones críticas incompletas.
- Dificultad para depurar y mantener el código, al no quedar claro el punto de fallo real y su contexto.
- Vulnerabilidades de seguridad si datos sensibles quedan expuestos o no protegidos correctamente tras un fallo inesperado.
- Fallo en la liberación de locks o recursos de concurrencia, potencialmente causando deadlocks o race conditions.

- Rompimiento de contratos de API al pasar excepciones a través de interfaces no preparadas para gestionarlas.

Buenas prácticas

- Capturar todas las excepciones antes de cruzar límites de hilos (`std::thread`) y comunicarlas mediante mecanismos seguros.

```
//-- Excepción no controlada dentro de un hilo que puede terminar el programa
abruptamente
void tarea() {
  throw std::runtime_error("error en hilo");
}
```

```
//-- Captura interna y comunicación controlada del error
void tarea(std::exception_ptr& eptr) {
  try {
    throw std::runtime_error("error en hilo");
  } catch (...) {
    eptr = std::current_exception();
  }
}
```

- Evitar que excepciones atraviesen interfaces C utilizando bloques `try-catch` y códigos de error.

```
//-- Excepción propagada a través de una interfaz C
extern "C" void funcion_c() {
  throw std::runtime_error("error");
}
```

```
//-- Captura y traducción a código de error compatible con C
extern "C" int funcion_c() {
  try {
    throw std::runtime_error("error");
    return 0;
  } catch (...) {
    return -1;
  }
}
```

- Controlar excepciones antes de salir de límites de bibliotecas dinámicas (DLL/so) para evitar comportamiento indefinido entre módulos compilados de manera diferente.

```
//-- Excepción cruza el límite de una biblioteca compartida
void api_publica() {
```

```
  lanzar_error();
}
```

```
//-- Manejo interno y retorno de estado controlado
bool api_publica() {
  try {
    lanzar_error();
    return true;
  } catch (const std::exception&) {
    return false;
  }
}
```

- Evitar lanzar excepciones desde destructores que puedan ejecutarse durante el desenrollado de pila.

```
//-- Destructor que lanza excepción durante stack unwinding
class Recurso {
public:
  ~Recurso() {
    throw std::runtime_error("fallo en destructor");
  }
};
```

```
//-- Destructor que captura internamente cualquier fallo
class Recurso {
public:
  ~Recurso() noexcept {
    try {
      liberar();
    } catch (...) {
      registrar_error();
    }
  }
};
```

- Definir funciones de entrada (por ejemplo, `main` o puntos de entrada de servicio) con captura global de excepciones para evitar propagación no controlada.

```
//-- Excepción no capturada en punto de entrada
int main() {
  procesar();
}
```

```
//-- Captura global para evitar propagación fuera del proceso
int main() {
```

```
  try {
    procesar();
  } catch (const std::exception& e) {
    registrar_error(e.what());
  } catch (...) {
    registrar_error("error desconocido");
  }
}
```

2.14.3.2 Estándares metodológicos

- **CERT/CPP-ERR59**: Do not throw an exception across execution boundaries.

2.14.4 [cpp.err.4] - Evitar finalizar abruptamente ante errores recuperables

ID	cpp.err.4	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Disponibilidad	S-SDLC	Build

Se debe manejar de forma segura y controlada todos los errores que puedan ser recuperables durante la ejecución, evitando la finalización abrupta de la aplicación. El propósito es prevenir comportamientos inesperados, pérdida de datos, interrupciones del servicio y posibles vulnerabilidades derivadas de estados inconsistentes.

Codificación insegura

Programa que solicita al usuario un número y lo utiliza como divisor. Se busca mostrar qué ocurre cuando se detecta un error recuperable, como una división por cero, y se decide terminar el proceso de forma abrupta en lugar de gestionarlo adecuadamente. La intención es evidenciar el impacto de finalizar la ejecución sin permitir manejo controlado del error.

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

int main() {
  //-- solicitar un número al usuario
  int divisor = 0;
  std::cout << "Ingrese un numero distinto de cero: ";
  std::cin >> divisor;

  //-- validar si el divisor es cero
  if (divisor == 0) {
    //-- finalizar el programa abruptamente ante un error recuperable
    std::cerr << "Error: division por cero detectada" << std::endl;
    std::exit(EXIT_FAILURE);
```

```
  }

  //-- realizar la division
  int resultado = 100 / divisor;
  std::cout << "Resultado: " << resultado << std::endl;

  return 0;
}
```

Codificación recomendada

Programa que solicita al usuario un número y lo utiliza como divisor. Se pretende mostrar cómo manejar un error recuperable, como la división por cero, informando al usuario y permitiendo que corrija la entrada sin terminar abruptamente la ejecución del proceso.

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

int main() {
  //-- declarar variable para el divisor
  int divisor = 0;

  //-- solicitar el numero hasta que sea valido
  while (true) {
    std::cout << "Ingrese un numero distinto de cero: ";
    std::cin >> divisor;

    //-- comprobar errores de lectura
    if (std::cin.fail()) {
      //-- limpiar el estado de error y descartar entrada invalida
      std::cin.clear();
      std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
      std::cerr << "Entrada invalida. Intente nuevamente." << std::endl;
      continue;
    }

    //-- validar que el divisor no sea cero
    if (divisor == 0) {
      std::cerr << "No se permite division por cero. Intente nuevamente." <<
std::endl;
      continue;
    }

    //-- salir del bucle cuando la entrada es valida
    break;
  }
```

```
    //-- realizar la division de forma segura
    int resultado = 100 / divisor;
    std::cout << "Resultado: " << resultado << std::endl;

    return 0;
}
```

2.14.4.1 Descripción

Este requisito se enfoca en promover que cualquier fallo que pueda ser gestionado de forma segura cuente con un mecanismo apropiado, como comprobaciones explícitas de valores de retorno, uso responsable de excepciones, estructuras de control preventivas o rutas alternativas de ejecución. Los desarrolladores deben evaluar sistemáticamente si un fallo permite continuar con la ejecución mediante una corrección local, un reinicio controlado de un módulo, una repetición segura de la operación o un mecanismo de fallback.

El diseño de un sistema robusto implica establecer criterios claros para diferenciar errores recuperables de los no recuperables. Los primeros suelen estar relacionados con condiciones ambientales no críticas como indisponibilidad temporal de un recurso, fallo en operaciones de entrada/salida, parámetros inesperados, formatos no válidos o interrupción de una comunicación externa. Cuando estos eventos ocurren, el software debe reaccionar mediante validaciones exhaustivas. Para ello es esencial incorporar bloques `try`/`catch` con excepciones específicas, así como validación sistemática de todas las funciones que devuelvan valores de retorno susceptibles de indicar error. El uso de `std::optional` y `std::expected` puede servir como mecanismo adicional para encapsular fallos de forma segura y obligar al desarrollador a tratarlos.

A la hora de implementar un manejo seguro de errores recuperables, es fundamental que la lógica del programa evite suposiciones implícitas sobre la ejecución. Cualquier operación que pueda fallar debe ser tratada explícitamente. De igual forma, se deben evitar bloqueos de recursos asociados a condiciones de error. Recursos como descriptores de archivo, buffers de memoria y mutex deben liberarse consistentemente, incluso ante fallos. En este sentido, el uso de RAII y objetos inteligentes como `std::unique_ptr` y `std::lock_guard` se convierte en una práctica clave para evitar fugas y estados no controlados que puedan amplificar el impacto del error original.

Riesgos asociados

- Pérdida de datos temporales o de usuario debido a la terminación inesperada de la aplicación.
- Corrupción de archivos o recursos abiertos si no se realiza un cierre/unlock adecuado.
- Fuga de memoria y recursos por no liberar correctamente objetos y handles al

finalizar abruptamente.

- Degradación de la experiencia del usuario al presentar cierre inesperado en lugar de manejar el error de forma controlada.
- Vulnerabilidades de seguridad si se dejan datos parciales o estructurales en estado inconsistente.
- Imposibilidad de recuperación automática o reinicio controlado del servicio/aplicación.
- Pérdida de trazabilidad y difícil diagnóstico de problemas debido a ausencia de registros y manejo apropiado de la excepción.
- Fallo en otras tareas o hilos relacionados si no se realiza una notificación o sincronización adecuada.

Buenas prácticas

- Propagar errores recuperables mediante excepciones en lugar de finalizar el proceso abruptamente.

```
//-- Finaliza el proceso ante un error recuperable
void procesarArchivo(const std::string& ruta) {
  if (!existe(ruta)) {
    std::cerr << "Archivo no encontrado";
    std::exit(EXIT_FAILURE);
  }
}
```

```
//-- Lanza una excepción para permitir manejo en niveles superiores
void procesarArchivo(const std::string& ruta) {
  if (!existe(ruta)) {
    throw std::runtime_error("Archivo no encontrado");
  }
}
```

- Devolver códigos de error o tipos seguros (`std::optional`/`std::expected`) cuando la situación es controlable.

```
//-- Termina la aplicación si falla la conversión
int convertir(const std::string& valor) {
  if (!esNumero(valor)) {
    std::abort();
  }
  return std::stoi(valor);
}
```

```
//-- Devuelve un valor opcional para que el llamador decida cómo actuar
std::optional<int> convertir(const std::string& valor) {
  if (!esNumero(valor)) {
    return std::nullopt;
  }
  return std::stoi(valor);
}
```

- Registrar el error y permitir la continuidad controlada cuando el fallo no compromete la integridad del sistema.

```
//-- Finaliza inmediatamente ante fallo de conexión secundaria
void enviarMetricas() {
  if (!conectarServidorSecundario()) {
    std::terminate();
  }
  enviarDatos();
}
```

```
//-- Registra el incidente y continúa con funcionalidad principal
void enviarMetricas() {
  if (!conectarServidorSecundario()) {
    logError("Servidor secundario no disponible");
    return;
  }
  enviarDatos();
}
```

- Evitar el uso de `assert` para manejar errores de ejecución esperables en producción.

```
//-- Usa assert que puede finalizar el programa en producción
void procesarPedido(int cantidad) {
  assert(cantidad > 0);
  calcularTotal(cantidad);
}
```

```
//-- Valida la condición y maneja el error de forma controlada
void procesarPedido(int cantidad) {
  if (cantidad <= 0) {
    throw std::invalid_argument("Cantidad invalida");
  }
  calcularTotal(cantidad);
}
```

2.14.4.2 Estándares metodológicos

- **CERT/CPP-ERR50**: Do not abruptly terminate the program.

2.14.5 [cpp.err.5] - Garantizar la seguridad frente a excepciones

ID	cpp.err.5	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Design

Se debe diseñar la aplicación para manejar y propagar de forma segura las excepciones y los errores, con el fin de evitar fugas de información, corrupción de memoria, estados inconsistentes, comportamientos indefinidos y denegaciones de servicio. Para ello se va a requerir de un tratamiento coherente, centralizado y seguro de situaciones excepcionales que puedan surgir durante la ejecución.

Codificación insegura

Gestión manual de memoria dentro de una clase que reserva recursos dinámicamente y realiza varias operaciones que pueden lanzar excepciones. Se busca observar qué ocurre cuando no se garantiza la liberación adecuada de recursos si se produce un error en medio del proceso.

```
#include <iostream>
#include <stdexcept>

class FileProcessor {
  char* buffer;
  size_t size;

public:
  FileProcessor(size_t s) : buffer(nullptr), size(s) {
    //-- Reserva memoria dinámica sin usar RAII
    buffer = new char[size];
  }

  void process(bool fail) {
    //-- Simula una operación que puede lanzar una excepción
    if (fail) {
      throw std::runtime_error("Error durante el procesamiento");
    }
    buffer[0] = 'A';
  }

  ~FileProcessor() {
    //-- Libera la memoria manualmente
    delete[] buffer;
  }
```

```
};

int main() {
  //-- Uso del objeto sin manejo adecuado de excepciones
  FileProcessor processor(1024);
  processor.process(true);
  std::cout << "Procesamiento completado" << std::endl;
  return 0;
}
```

Codificación recomendada

Uso del principio RAII y contenedores estándar para asegurar que los recursos se liberen automáticamente incluso si ocurre una excepción. Se busca garantizar consistencia del estado y evitar fugas de memoria ante fallos durante la ejecución.

```
#include <iostream>
#include <stdexcept>
#include <vector>

class FileProcessor {
  std::vector<char> buffer;

public:
  FileProcessor(size_t s) : buffer(s) {
    //-- La memoria es gestionada automáticamente por std::vector
  }

  void process(bool fail) {
    //-- Operación que puede lanzar una excepción sin comprometer recursos
    if (fail) {
      throw std::runtime_error("Error durante el procesamiento");
    }
    buffer[0] = 'A';
  }
};

int main() {
  //-- Manejo adecuado de excepciones para mantener el control del flujo
  try {
    FileProcessor processor(1024);
    processor.process(true);
    std::cout << "Procesamiento completado" << std::endl;
  } catch (const std::exception& ex) {
    //-- Captura la excepción y evita terminación inesperada
    std::cerr << "Excepción capturada: " << ex.what() << std::endl;
  }
  return 0;
```

```
}
```

2.14.5.1 Descripción

El desarrollador debe garantizar un tratamiento coherente, seguro y verificable de cualquier excepción, evitando dejar al sistema en un estado inconsistente. Este requisito abarca tanto la definición de políticas claras de gestión de errores como la implementación de constructos del lenguaje que mantengan los invariantes de seguridad incluso ante fallos inesperados. El uso inadecuado de excepciones puede llevar a comportamientos indefinidos, especialmente cuando se lanzan excepciones desde destructores, se realizan operaciones que pueden fallar en contextos no seguros, o se mezclan mecanismos de propagación de errores como códigos de retorno y excepciones sin una convención uniforme. Para evitar estos problemas, es fundamental adoptar un enfoque sistemático que cubra todo el ciclo de vida de los errores: desde su detección, pasando por la notificación, hasta su mitigación y recuperación segura.

Uno de los principios más importantes para gestionar errores es garantizar que todas las funciones que pueden lanzar una excepción estén correctamente documentadas y que el código que depende de ellas se encuentre dentro de bloques `try` apropiados. Asimismo, cualquier acción que deba mantener la integridad del sistema debe ser atómica o seguir el patrón de **commit or rollback**, asegurando que ninguna excepción deje datos parciales o estructuras inconsistentes.

Los destructores deben diseñarse para no lanzar excepciones en ninguna circunstancia. Lanzar una excepción durante el **stack unwinding** genera un estado de doble excepción que concluye en una llamada a `std::terminate()`, finalizando abruptamente la aplicación. Para evitar esta situación, los destructores deben envolver cualquier operación susceptible de fallar en bloques de captura y manejar localmente los errores.

El patrón **RAII** es fundamental para garantizar una gestión segura de recursos frente a excepciones. El uso adecuado de RAII permite que los recursos se liberen automáticamente al salir de un ámbito, sin necesidad de lógica adicional que pueda fallar o quedar sin ejecutar cuando se lanza una excepción. Este patrón resulta especialmente importante para gestionar memoria dinámica, descriptores de archivo, bloqueos de sincronización y otros recursos sensibles.

Para que el sistema mantenga su integridad ante fallos, es necesario implementar controles estrictos de validación antes y después de ejecutar acciones críticas. Esto incluye validar parámetros, verificar precondiciones y postcondiciones, y asegurar que los invariantes de clase se conserven incluso en caso de excepción. Una clase con invariantes débiles es un vector común de fallos de seguridad cuando se lanzan excepciones en momentos inesperados.

Riesgos asociados

- Pérdida de recursos (memory leaks) por manejo inadecuado de recursos en presencia de excepciones.
- Corrupción de estado interno de objetos al quedar en estados inconsistentes tras una excepción.
- Fuga de información sensible debido a la omisión de limpieza de información crítica cuando una excepción es lanzada.
- Condiciones de carrera o bloqueos al no liberar correctamente locks o semáforos en flujos excepcionales.
- Caídas abruptas del sistema (crashes) si no se gestionan adecuadamente las excepciones o se permite la propagación de excepciones sin control.
- Vulnerabilidad a ataques de denegación de servicio (DoS) si el software entra en bucles o estados erróneos por la mala gestión de excepciones.
- Pérdida o corrupción de datos al no realizar rollback o restauración del estado tras una excepción.
- Baja mantenibilidad y dificultad de depuración debido a la falta de una estrategia clara para el tratamiento de excepciones.

Buenas prácticas

- Aplicar el principio RAII para garantizar la liberación automática de recursos ante excepciones.

```
//-- Gestión manual de memoria sin liberación segura ante excepción
void procesar() {
  int* datos = new int[10];
  operacionQueLanza();
  delete[] datos;
}
```

```
//-- Uso de contenedor que libera recursos automáticamente
void procesar() {
  std::vector datos(10);
  operacionQueLanza();
}
```

- Utilizar smart pointers en lugar de punteros crudos para evitar fugas de memoria.

```
//-- Uso de puntero crudo sin control ante excepción
void ejemplo() {
```

```
  Recurso* r = new Recurso();
  usarRecurso(r);
  delete r;
}
```

```
//-- Uso de std::unique_ptr para gestión automática
void ejemplo() {
  std::unique_ptr r = std::make_unique();
  usarRecurso(r.get());
}
```

- Garantizar la consistencia del estado del objeto aplicando el patrón copy-and-swap en operadores de asignación.

```
//-- Asignación directa que puede dejar el objeto en estado inconsistente
Clase& operator=(const Clase& otra) {
  if (this != &otra) {
    delete recurso;
    recurso = new Recurso(*otra.recurso);
  }
  return *this;
}
```

```
//-- Implementación basada en copy-and-swap
Clase& operator=(Clase otra) {
  std::swap(recurso, otra.recurso);
  return *this;
}
```

- Declarar funciones `noexcept` cuando se garantiza que no lanzan excepciones.

```
//-- Destructor que podría propagar excepciones
~Clase() {
  liberar();
}
```

```
//-- Destructor declarado noexcept para evitar propagación
~Clase() noexcept {
  liberar();
}
```

- Capturar excepciones por referencia constante para evitar copias innecesarias y slicing.

```
//-- Captura por valor que puede producir slicing
```

```
try {
  operacion();
} catch (std::exception e) {
  registrar(e.what());
}
```

```
//-- Captura por referencia constante
try {
  operacion();
} catch (const std::exception& e) {
  registrar(e.what());
}
```

2.14.5.2 Estándares metodológicos

- **CERT/CPP-ERR56**: Guarantee exception safety.

2.14.6 [cpp.err.6] - Utilizar std::expected para devolver un valor o un error cuando no se utilicen excepciones

ID	cpp.err.6	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

En entornos de C++, donde las excepciones estén deshabilitadas o restringidas por motivos de seguridad, rendimiento o portabilidad, se deben emplear estructuras como **std::expected** para representar de forma explícita y segura el resultado de una operación que puede fallar. Se reduce la posibilidad de errores silenciosos y se mejora la capacidad de inspección y auditoría del código, y permite que los desarrolladores gestionen fallos sin depender de mecanismos de propagación implícita como las excepciones tradicionales.

Codificación insegura

Gestión de errores al convertir una cadena a entero sin usar excepciones. Se busca devolver tanto el resultado como el posible error, pero se utiliza un mecanismo basado en valores centinela que puede generar ambigüedad y errores de interpretación por parte del llamador.

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

//-- Convierte una cadena a entero usando un valor centinela para indicar error
int parseNumber(const std::string& input) {
  //-- Si la cadena está vacía se devuelve -1 como indicador de error
```

```
  if (input.empty()) {
    return -1;
  }

  //-- Intentar convertir manualmente
  int result = 0;
  for (char c : input) {
    //-- Validar que el carácter sea un dígito
    if (c < '0' || c > '9') {
      return -1;
    }
    result = result * 10 + (c - '0');
  }

  return result;
}

int main() {
  std::string value = "123a";

  //-- El llamador debe interpretar manualmente si hubo error
  int number = parseNumber(value);

  if (number == -1) {
    std::cout << "Error al convertir la cadena" << std::endl;
  } else {
    std::cout << "Numero convertido: " << number << std::endl;
  }

  return 0;
}
```

Codificación recomendada

Gestión de errores al convertir una cadena a entero sin usar excepciones, devolviendo explícitamente un valor válido o un error tipado mediante `std::expected`. Se busca que el contrato de la función sea claro, evitando valores ambiguos y obligando al llamador a comprobar el estado del resultado.

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

//-- Convierte una cadena a entero devolviendo valor o mensaje de error
std::expected<int, std::string> parseNumber(const std::string& input) {
  //-- Validar entrada vacía
  if (input.empty()) {
    return std::unexpected("La cadena está vacía");
```

```
  }

  int result = 0;
  for (char c : input) {
    //-- Validar que el carácter sea un dígito
    if (c < '0' || c > '9') {
      return std::unexpected("Carácter no numérico encontrado");
    }
    result = result * 10 + (c - '0');
  }

  return result;
}

int main() {
  std::string value = "123a";

  //-- El resultado obliga a comprobar si contiene valor o error
  auto result = parseNumber(value);

  if (!result.has_value()) {
    std::cout << "Error: " << result.error() << std::endl;
  } else {
    std::cout << "Numero convertido: " << result.value() << std::endl;
  }

  return 0;
}
```

2.14.6.1 Descripción

El uso de `std::expected` proporciona un mecanismo explícito para representar resultados que pueden contener un valor válido o un error. En contextos de desarrollo seguro, este patrón es especialmente relevante cuando las excepciones están deshabilitadas o no se desean emplear debido a restricciones de rendimiento, requisitos de plataformas embebidas o políticas de seguridad que desaconsejan su uso. El empleo de este enfoque obliga a que el flujo de control se mantenga visible y verificable, reduciendo los riesgos derivados de saltos implícitos y facilitando auditorías de código orientadas a garantizar la integridad y la robustez de las funciones críticas.

La adopción de **std::expected** permite evitar varios problemas comunes relacionados con la gestión de errores mediante excepciones. Al tratarse de un retorno explícito, cualquier operación que pueda fallar queda representada en la firma de la función, lo que obliga a los consumidores de la API a gestionar la posibilidad de error. Esto reduce la probabilidad de fallos no controlados, estados corruptos y flujos de ejecución impredecibles. También contribuye a mejorar la trazabilidad de errores, ya que el error forma parte del valor retornado y puede registrarse, rastrearse y transmitirse con mayor

detalle. Dentro de equipos de seguridad, este mecanismo se considera más auditable y coherente con prácticas de diseño seguro.

Es crucial que el desarrollador defina una política homogénea para la estructura del error en `std::expected`. Esta estructura debe ser lo suficientemente informativa como para permitir diagnósticos y registros precisos, pero sin exponer información sensible que pueda filtrarse. Asimismo, es recomendable que los errores sean inmutables, serializables y aptos para un análisis estructurado, ya sea mediante logs o herramientas automatizadas. La comunicación del error a capas superiores debe mantenerse clara y coherente, evitando concatenar información redundante o generar mensajes que puedan inducir confusión en procesos de depuración.

Este mecanismo es útil en sistemas embebidos, componentes criptográficos, parsers de entrada, controladores y cualquier módulo donde la resiliencia frente a fallos forma parte integral del diseño seguro. También aporta claridad cuando se implementan flujos de validación de datos provenientes de fuentes no confiables, donde los errores deben analizarse y manejarse de forma determinista.

Riesgos asociados

- **Propagación silenciosa de errores**: Si no se utiliza `std::expected`, los errores pueden no ser detectados ni gestionados, lo que produce fallos silenciosos en la aplicación.
- **Código difícil de mantener**: El manejo inconsistente de resultados y errores resulta en código menos claro y propenso a comportamientos inesperados.
- **Condiciones de carrera y corrupción de datos**: La falta de manejo explícito de errores puede provocar que los recursos no sean liberados correctamente o que los datos queden en estados inválidos.
- **Fugas de memoria y recursos**: Si ocurre un error, pero no se gestiona correctamente su propagación, pueden producirse fugas de memoria o recursos (por ejemplo, archivos abiertos, sockets, etc.).
- **Superficies de ataque ampliadas**: Los errores no gestionados pueden ser explotados por atacantes para alterar el flujo del programa, provocar denegaciones de servicio o ejecutar código malicioso.
- **Desbordamiento de pila o bucles infinitos**: Al no propagar errores adecuadamente, es posible que el programa entre en estados anómalos, como llamadas recursivas sin fin o ciclos interminables intentando operar sobre errores no gestionados.
- **Pérdida de confianza en la robustez del sistema**: La inestabilidad y falta de manejo de errores reduce la confiabilidad del sistema y puede afectar la percepción del usuario o del cliente.

Buenas prácticas

- Devolver `std::expected` en lugar de códigos de error primitivos para evitar ambigüedad en la interpretación del resultado.

```
//-- Uso de código de error entero sin contexto
int dividir(int a, int b) {
  if (b == 0) return -1;
  return a / b;
}
```

```
//-- Uso de std::expected para devolver valor o error tipado
std::expected<int, std::string> dividir(int a, int b) {
  if (b == 0) return std::unexpected("division por cero");
  return a / b;
}
```

- Definir un tipo de error específico en `std::expected` en lugar de usar tipos genéricos o booleanos.

```
//-- Uso de booleano sin detalle del fallo
bool validarUsuario(const std::string& user) {
  if (user.empty()) return false;
  return true;
}
```

```
//-- Uso de enum como tipo de error explícito
enum class ErrorValidacion { Vacio };

std::expected<void, ErrorValidacion> validarUsuario(const std::string& user) {
  if (user.empty()) return std::unexpected(ErrorValidacion::Vacio);
  return {};
}
```

- Obligar al consumidor a comprobar el resultado antes de acceder al valor devuelto.

```
//-- Acceso directo al valor sin verificación previa
auto resultado = dividir(10, 0);
int valor = resultado.value();
```

```
//-- Verificación explícita antes de usar el valor
auto resultadoSeguro = dividir(10, 2);
if (resultadoSeguro.has_value()) {
  int valor = resultadoSeguro.value();
} else {
```

```
  auto error = resultadoSeguro.error();
}
```

- Propagar errores usando `std::expected` en lugar de convertirlos o ignorarlos.

```
//-- Ignorar error y devolver valor por defecto
int obtenerResultado() {
  auto r = dividir(10, 0);
  if (!r.has_value()) return 0;
  return r.value();
}
```

```
//-- Propagación explícita del error al llamador
std::expected<int, std::string> obtenerResultado() {
  auto r = dividir(10, 0);
  if (!r.has_value()) return std::unexpected(r.error());
  return r.value();
}
```

2.14.6.2 Estándares metodológicos

- **C++ Core-E.27**: If you can't throw exceptions, use error codes systematically.

2.15 Concurrencia

2.15.1 [cpp.sync.1] - Asumir el código como potencialmente multihilo

ID	cpp.sync.1	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Todo desarrollo debe concebirse como susceptible de ejecutarse en entornos multihilo, incluso cuando la aplicación parezca inicialmente secuencial. Asumir este modelo permite evitar condiciones de carrera, corrupción de memoria, uso inseguro de recursos compartidos y bloqueos imprevisibles. La práctica implica diseñar estructuras de datos thread-safe, aplicar sincronización explícita cuando sea necesario y limitar el acceso concurrente a objetos no seguros. Asimismo, fomenta decisiones preventivas de arquitectura que reducen vulnerabilidades asociadas a la concurrencia y facilita el uso correcto de las utilidades modernas de C++.

Codificación insegura

Acceso concurrente a una variable global compartida desde múltiples hilos para incrementar un contador. Se busca mostrar el impacto de no sincronizar el acceso y cómo pueden aparecer condiciones de carrera que generan resultados inconsistentes.

```
#include <iostream>
#include <thread>
#include <vector>

int counter = 0;

void increment() {
  //-- Incrementa el contador compartido sin sincronización
  for (int i = 0; i < 100000; ++i) {
    counter++;
  }
}

int main() {
  //-- Lanza múltiples hilos que acceden al mismo recurso
  std::vector<std::thread> threads;
  for (int i = 0; i < 10; ++i) {
    threads.emplace_back(increment);
  }

  //-- Espera a que todos los hilos finalicen
  for (auto& t : threads) {
    t.join();
  }

  //-- Muestra el resultado final del contador
  std::cout << "Counter final: " << counter << std::endl;
  return 0;
}
```

Codificación recomendada

Acceso concurrente a una variable compartida utilizando mecanismos de sincronización para evitar condiciones de carrera. Se aplica un bloqueo explícito mediante mutex para garantizar la exclusión mutua en un entorno potencialmente multihilo.

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

int counter = 0;
std::mutex mtx;
```

```
void increment() {
  //-- Incrementa el contador compartido de forma sincronizada
  for (int i = 0; i < 100000; ++i) {
    std::lock_guard<std::mutex> lock(mtx);
    counter++;
  }
}

int main() {
  //-- Lanza múltiples hilos que acceden al mismo recurso
  std::vector<std::thread> threads;
  for (int i = 0; i < 10; ++i) {
    threads.emplace_back(increment);
  }

  //-- Espera a que todos los hilos finalicen
  for (auto& t : threads) {
    t.join();
  }

  //-- Muestra el resultado final del contador
  std::cout << "Counter final: " << counter << std::endl;
  return 0;
}
```

2.15.1.1 Descripción

En el desarrollo de aplicaciones modernas en C++, incluso aquellas que inicialmente no están diseñadas con concurrencia directa, resulta fundamental asumir que el código podrá ejecutarse en contextos multihilo. Esta expectativa responde a las características de los entornos actuales: bibliotecas internas que lanzan hilos, runtimes que paralelizan operaciones, frameworks que distribuyen carga o tareas asíncronas y sistemas operativos que pueden activar ejecuciones concurrentes de manera implícita. Asumir esta condición desde el diseño evita que el producto final incurra en errores clásicos como condiciones de carrera, corrupción de datos, comportamientos indefinidos y fallos de seguridad generados por accesos concurrentes no controlados.

La concurrencia en C++ combina potencia y riesgo. El lenguaje ofrece flexibilidad para manipular memoria, punteros y estructuras complejas, lo cual incrementa la posibilidad de que varios hilos modifiquen simultáneamente un mismo recurso. La ausencia de protecciones automáticas obliga al desarrollador a extremar el control. **Asumir siempre el contexto multihilo** es una estrategia preventiva que reduce enormemente la superficie de vulnerabilidades. Permite diseñar interfaces seguras, elegir estructuras thread-safe, proteger invariantes internas y documentar claramente el modelo de concurrencia esperado.

Una condición de carrera ocurre cuando más de un hilo accede a una sección crítica sin coordinación, generando estados inconsistentes. Estas situaciones pueden ser explotadas si un atacante fuerza condiciones de acceso específicas, incrementando la superficie de vulnerabilidad. Diseñar con anticipación permite que cada operación que modifique el estado interno de un objeto sea respaldada por mecanismos de protección adecuados, reduciendo riesgos de corrupción, lectura de datos inconsistentes o alteraciones no autorizadas.

La memoria compartida es una de las fuentes principales de errores en entornos multihilo en C++. Deben aplicarse principios que reduzcan su presencia siempre que sea posible. En su lugar, se favorecen mensajes inmutables, paso de datos por copia controlada y diseños basados en colas seguras. Minimizar el alcance de lo compartido reduce la necesidad de sincronización y facilita el razonamiento sobre el flujo del programa, lo que incrementa la seguridad y la robustez del sistema.

El estándar moderno proporciona herramientas que permiten implementar concurrencia con mayor seguridad. Entre ellas se encuentran `std::thread`, `std::async`, `std::future`, `std::atomic`, `std::mutex`, `std::shared_mutex` y `std::condition_variable`. El uso de estas utilidades ayuda a mantener un estilo de programación seguro, evitando construcciones peligrosas como punteros no sincronizados, bloqueos manuales no deterministas o algoritmos con dependencias implícitas. Al asumir que el código puede ser accedido por múltiples hilos, el desarrollador elige de forma natural estas herramientas y reduce el riesgo de errores estructurales.

Cualquier API expuesta por un módulo debe documentar claramente su modelo de concurrencia. Asumir multihilo desde el principio permite que la interfaz esté diseñada para no permitir escenarios ambiguos.

Cada función debe explicitar si:

- es thread-safe.
- requiere protección externa.
- se limita a un hilo concreto.

Este nivel de claridad fortalece la seguridad general, evitando intromisiones no deseadas y comportamientos indeterminados causados por malentendidos en el uso de la API.

Riesgos asociados

- Condiciones de carrera que pueden producir resultados impredecibles o erróneos.
- Interbloqueos (deadlocks) debido a la mala gestión de recursos compartidos.
- Corrupción de datos por acceso concurrente sin protección adecuada.
- Comportamiento indefinido o fallos difíciles de reproducir y depurar.

- Vulnerabilidades de seguridad explotables por atacantes para alterar el flujo o los datos del programa.
- Pérdida de información o inconsistencias de estado en sistemas críticos.
- Bajo rendimiento o bloqueos por malos patrones de sincronización.
- Exposición de condiciones de fuga de memoria al competir por inicialización o liberación de recursos.
- Incremento significativo en la complejidad técnica y en los costos de mantenimiento y pruebas.

Buenas prácticas

- Proteger los recursos compartidos con mecanismos de sincronización adecuados (`std::mutex`, `std::lock_guard`).

```
//-- Acceso a recurso compartido sin sincronización
int contador = 0;

void incrementar() {
  contador++;
}
```

```
//-- Uso de mutex para proteger el recurso compartido
int contadorSeguro = 0;
std::mutex mtx;

void incrementarSeguro() {
  std::lock_guard<std::mutex> lock(mtx);
  contadorSeguro++;
}
```

- Evitar datos globales mutables; preferir inmutabilidad o encapsulación segura para hilos.

```
//-- Variable global mutable accesible por múltiples hilos
std::string estado = "INIT";

void actualizar() {
  estado = "RUNNING";
}
```

```
//-- Encapsulación con acceso sincronizado
class EstadoSeguro {
  std::string estado = "INIT";
  std::mutex mtx;
```

```
public:
  void actualizar() {
    std::lock_guard<std::mutex> lock(mtx);
    estado = "RUNNING";
  }
};
```

- Utilizar tipos atómicos (`std::atomic`) para operaciones simples compartidas entre hilos.

```
//-- Incremento de bandera compartida sin atomicidad
bool listo = false;

void marcarListo() {
  listo = true;
}
```

```
//-- Uso de tipo atómico para evitar condición de carrera
std::atomic<bool> listoAtomico(false);

void marcarListoSeguro() {
  listoAtomico.store(true, std::memory_order_release);
}
```

- Evitar dobles bloqueos y posibles interbloqueos adquiriendo múltiples mutex en orden consistente o usando `std::scoped_lock`.

```
//-- Bloqueo de mutex en distinto orden generando riesgo de interbloqueo
std::mutex m1;
std::mutex m2;

void tarea1() {
  std::lock_guard<std::mutex> l1(m1);
  std::lock_guard<std::mutex> l2(m2);
}

void tarea2() {
  std::lock_guard<std::mutex> l2(m2);
  std::lock_guard<std::mutex> l1(m1);
}
```

```
//-- Bloqueo simultáneo con std::scoped_lock para evitar interbloqueo
void tareaSegura() {
  std::scoped_lock lock(m1, m2);
}
```

2.15.1.2 Estándares metodológicos

- **C++ Core-CP.1**: Assume that your code will run as part of a multi-threaded program.

2.15.2 [cpp.sync.2] - Evitar condiciones de carrera

ID	**cpp.sync.2**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Se deben aplicar mecanismos seguros de sincronización para impedir que hilos o procesos accedan simultáneamente a recursos compartidos de manera no controlada, lo que puede originar corrupción de datos, comportamientos impredecibles y vulnerabilidades explotables. Para mitigar estos riesgos, deben utilizarse primitivas de exclusión mutua, estructuras atómicas, modelos de memoria seguros y patrones de diseño que garanticen accesos ordenados.

Codificación insegura

Programa multihilo que incrementa un contador compartido desde varios hilos sin ningún mecanismo de sincronización. Se pretende observar cómo el acceso concurrente a datos compartidos puede producir resultados inconsistentes debido a condiciones de carrera.

```
#include <iostream>
#include <thread>
#include <vector>

int counter = 0;

void increment() {
  //-- Incrementa el contador sin sincronización
  for (int i = 0; i < 100000; ++i) {
    counter++;
  }
}

int main() {
  //-- Crea varios hilos que acceden al mismo recurso compartido
  std::vector<std::thread> threads;
  for (int i = 0; i < 4; ++i) {
    threads.emplace_back(increment);
  }

  //-- Espera a que todos los hilos terminen
  for (auto& t : threads) {
```

```
    t.join();
  }

  //-- Muestra el valor final del contador
  std::cout << "Valor final del contador: " << counter << std::endl;
  return 0;
}
```

Codificación recomendada

Programa multihilo que incrementa un contador compartido utilizando mecanismos de sincronización para proteger el acceso concurrente. Se busca mostrar cómo eliminar condiciones de carrera mediante el uso de primitivas adecuadas del lenguaje.

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

int counter = 0;
std::mutex mtx;

void increment() {
  //-- Incrementa el contador usando exclusión mutua
  for (int i = 0; i < 100000; ++i) {
    std::lock_guard<std::mutex> lock(mtx);
    counter++;
  }
}

int main() {
  //-- Crea varios hilos que acceden al mismo recurso compartido de forma segura
  std::vector<std::thread> threads;
  for (int i = 0; i < 4; ++i) {
    threads.emplace_back(increment);
  }

  //-- Espera a que todos los hilos terminen
  for (auto& t : threads) {
    t.join();
  }

  //-- Muestra el valor final del contador
  std::cout << "Valor final del contador: " << counter << std::endl;
  return 0;
}
```

2.15.2.1 Descripción

Las condiciones de carrera constituyen una fuente significativa de errores graves y vulnerabilidades, especialmente en sistemas de alta concurrencia o de tiempo real. Una condición de carrera ocurre cuando dos o más hilos acceden de forma concurrente a un recurso compartido sin que exista un mecanismo de sincronización adecuado, provocando resultados diferentes en ejecuciones sucesivas. El comportamiento es inherentemente no determinista, lo que complica su detección y puede dar lugar a fallos de integridad, valores corruptos, fugas de información o interrupciones del servicio. Para reducir estos riesgos, el lenguaje proporciona primitivas modernas como `std::mutex`, `std::scoped_lock`, `std::atomic` y los modelos de memoria definidos desde C++11, que permiten estructurar código seguro y predecible.

El uso seguro de la concurrencia parte del conocimiento profundo del modelo de memoria, de la garantía de ordenación de accesos y de la aplicación consistente de mecanismos de exclusión mutua. Es importante comprender que las condiciones de carrera no solo surgen del acceso de escritura concurrente, sino también de la lectura simultánea cuando estas lecturas no están sincronizadas con operaciones de actualización. Para minimizar riesgos, debe emplearse una estrategia sistemática basada en el uso de primitivas seguras y patrones robustos.

- **Exclusión mutua:** Garantiza que solo un hilo acceda a la sección crítica.
- **Sincronización de memoria:** Asegura orden lógico entre accesos concurrentes.
- **Estructuras atómicas:** Proveen operaciones indivisibles para evitar corrupción de datos.
- **Diseños lock-free:** Útiles solo cuando se comprenda su complejidad y restricciones.
- **Evitar datos compartidos:** Preferir inmutabilidad o encapsulamiento de recursos.

Mecanismos de sincronización

Las primitivas estándar permiten construir soluciones seguras basadas en exclusión mutua, evitando errores comunes asociados a liberaciones manuales o doble bloqueo. El mecanismo recomendado para la mayoría de los casos es `std::mutex` combinado con RAII mediante `std::lock_guard` o `std::scoped_lock`. Adicionalmente, los tipos `std::atomic` permiten realizar operaciones atómicas sin necesidad de mutex, siempre que las estructuras de datos involucradas puedan operar de modo estrictamente atómico.

```
std::mutex m;
int recurso_compartido = 0;

void incrementar() {
```

```
    std::lock_guard<std::mutex> lock(m);
    recurso_compartido++;
}
```

Este patrón asegura que un único hilo modifique el recurso compartido, eliminando interleavings peligrosos. Además, la garantía RAII protege contra fugas de bloqueo incluso si se lanzan excepciones.

Uso de variables atómicas

Las variables atómicas pueden mejorar el rendimiento cuando se requieren operaciones simples sin bloquear hilos. C++ define operaciones de lectura, escritura, intercambio y suma atómica que garantizan la ausencia de condiciones de carrera sin necesidad de un mutex. No obstante, su empleo exige comprender los modos de ordenación de memoria como `memory_order_seq_cst`, `memory_order_release` o `memory_order_acquire`, ya que un uso incorrecto puede reintroducir errores.

```
std::atomic<int> contador{0};

void incrementar() {
    contador.fetch_add(1, std::memory_order_relaxed);
}
```

El acceso compartido a datos globales, caches internas sin sincronización o el retorno de referencias a datos internos pueden generar condiciones de carrera incluso cuando la lógica parece segura. Es crucial minimizar la compartición de estados mutables y favorecer modelos de diseño como paso de mensajes, inmutabilidad o hilos propietarios de datos. También es fundamental evitar sincronización incompleta, como bloquear solo en la lectura, pero no en la escritura, o emplear dobles comprobaciones sin garantías de memoria seguras.

Riesgos asociados

- Pérdida o corrupción de datos por accesos concurrentes no controlados.
- Comportamiento indefinido de la aplicación, resultando en fallos difíciles de reproducir y depurar.
- Aumento significativo de la superficie de ataque para exploits, como escalada de privilegios o ejecución de código arbitrario.
- Interbloqueos (deadlocks) que ocasionan que el sistema o aplicación quede congelado.
- Ataques de tipo TOCTOU (Time of Check to Time of Use), permitiendo que un atacante manipule datos entre la verificación y el uso.

- Exposición de información sensible o confidencial a usuarios no autorizados mediante acceso simultáneo a recursos compartidos.
- Violaciones de integridad, por la posibilidad de que diferentes hilos modifiquen datos críticos de forma no sincronizada.

Buenas prácticas

- Proteger el acceso a recursos compartidos mediante mecanismos de sincronización como `std::mutex`.

```
//-- Incremento de variable compartida sin sincronización
int contador = 0;

void incrementar() {
  contador++;
}
```

```
//-- Incremento protegido con mutex para evitar acceso concurrente
#include <mutex>
std::mutex mtx;
int contador = 0;

void incrementar() {
  std::lock_guard<std::mutex> lock(mtx);
  contador++;
}
```

- Utilizar tipos atómicos (`std::atomic`) cuando se requieran operaciones simples compartidas entre hilos.

```
//-- Uso de variable compartida sin atomicidad
bool listo = false;

void hilo1() {
  listo = true;
}
```

```
//-- Uso de std::atomic para garantizar visibilidad entre hilos
#include <atomic>
std::atomic<bool> listo(false);

void hilo1() {
  listo.store(true);
}
```

- Evitar el doble chequeo sin sincronización adecuada en inicialización diferida.

```
//-- Inicialización diferida sin protección
MyClass* instancia = nullptr;

MyClass* obtenerInstancia() {
  if (instancia == nullptr) {
    instancia = new MyClass();
  }
  return instancia;
}
```

```
//-- Inicialización segura usando std::call_once
#include <mutex>
MyClass* instancia = nullptr;
std::once_flag flag;

void init() {
  instancia = new MyClass();
}
MyClass* obtenerInstancia() {
  std::call_once(flag, init);
  return instancia;
}
```

- Minimizar la sección crítica para reducir la probabilidad de bloqueos y condiciones de carrera.

```
//-- Sección crítica extensa que incrementa riesgo de inconsistencias
std::mutex mtx;
int dato = 0;

void procesar() {
  std::lock_guard<std::mutex> lock(mtx);
  dato++;
  realizarOperacionCostosa();
}
```

```
//-- Sección crítica limitada únicamente al recurso compartido
std::mutex mtx;
int dato = 0;

void procesar() {
  {
    std::lock_guard<std::mutex> lock(mtx);
    dato++;
  }
  realizarOperacionCostosa();
}
```

2.15.2.2 Estándares metodológicos

- **C++ Core-CP.2**: Avoid data races.
- **CERT/CPP-CON52**: Prevent data races when accessing bit-fields from multiple threads.

2.15.3 [cpp.sync.3] - Minimizar el uso compartido de datos modificables

ID	cpp.sync.3	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Reducir de forma estricta el uso compartido de datos modificables, especialmente en entornos concurrentes o de alta complejidad, genera riesgos de condiciones de carrera, corrupción de memoria, violaciones de integridad y fallos impredecibles que afectan a la seguridad y estabilidad. La recomendación se fundamenta en emplear estrategias como evitar globales mutables, favorecer la inmutabilidad, encapsular los accesos a través de abstracciones seguras y usar mecanismos de sincronización robustos cuando el intercambio sea inevitable. Su objetivo final es proteger la integridad del sistema y prevenir vulnerabilidades derivadas del acceso concurrente inseguro.

Codificación insegura

Uso de una variable global modificable compartida entre múltiples funciones para contar operaciones. Se busca mostrar cómo el estado global puede ser alterado desde cualquier parte del código, dificultando el control, mantenimiento y pruebas del programa.

```
#include <iostream>

int globalCounter = 0;

void increment() {
  //-- Incrementa el contador global
  globalCounter++;
}

void decrement() {
  //-- Decrementa el contador global
  globalCounter--;
}

int main() {
  //-- Llamadas a funciones que modifican el mismo estado global
  increment();
  increment();
```

```
  decrement();

  //-- Se imprime el valor final del contador global
  std::cout << "Valor final: " << globalCounter << std::endl;
  return 0;
}
```

Codificación recomendada

Uso de encapsulamiento para evitar datos modificables compartidos globalmente. El contador se mantiene como estado interno de una clase y sólo se modifica a través de métodos públicos controlados, reduciendo el acoplamiento y mejorando la mantenibilidad.

```
#include <iostream>

class Counter {
  //-- Atributo privado para evitar acceso directo externo
  int value;

public:
  //-- Constructor que inicializa el contador
  Counter() : value(0) {}

  //-- Método público para incrementar el contador
  void increment() {
    value++;
  }

  //-- Método público para decrementar el contador
  void decrement() {
    value--;
  }

  //-- Método público de solo lectura
  int getValue() const {
    return value;
  }
};

int main() {
  //-- Se crea una instancia local del contador
  Counter counter;

  //-- Las modificaciones se realizan mediante métodos controlados
  counter.increment();
  counter.increment();
  counter.decrement();
```

```
    //-- Se obtiene el valor sin exponer el estado interno
    std::cout << "Valor final: " << counter.getValue() << std::endl;
    return 0;
}
```

2.15.3.1 Descripción

La presencia de datos mutables accesibles por múltiples partes del código incrementa sustancialmente los riesgos de corrupción de memoria, condiciones de carrera, cambios inesperados del estado interno y comportamientos indefinidos. En este tipo de lenguajes, donde la gestión de memoria y la semántica de acceso pueden ser muy detalladas, **evitar el estado compartido mutable es una línea de defensa fundamental** en la seguridad estructural del software.

Un principio clave es que todo dato que pueda ser modificado debe analizarse cuidadosamente para determinar su necesidad real de ser compartido. En la mayoría de los casos, la compartición surge por conveniencia o por diseños poco encapsulados, cuando una arquitectura más sólida permitiría distribuir copias inmutables, flujos de datos unidireccionales o contenedores seguros para concurrencia. La práctica habitual consiste en priorizar la inmutabilidad y limitar el alcance de variables, lo cual reduce la superficie de ataque y mejora drásticamente la rastreabilidad del estado.

El uso de variables globales mutables es especialmente problemático. Este tipo de variables puede ser modificado por cualquier parte del código, lo que crea una dependencia implícita que debilita la estructura del programa. **Las globales mutables deben considerarse siempre una amenaza a la integridad**, pues dificultan la verificación formal, impiden el aislamiento entre módulos y aumentan las posibilidades de acceso no autorizado o accidental. Al limitar el uso de globales y reemplazarlas por estructuras encapsuladas, se facilita la aplicación de controles de sincronización y se reduce la complejidad cognitiva.

En el contexto de concurrencia, C++ ofrece primitivas como `std::mutex`, `std::lock_guard`, `std::shared_mutex` y otros tipos especializados. Sin embargo, depender exclusivamente de mecanismos de bloqueo puede introducir bloqueos innecesarios, degradación del rendimiento y deadlocks si no se gestionan con disciplina. Por ello, **la mejor práctica es eliminar la necesidad de sincronización siempre que sea posible**, mediante técnicas como paso de mensajes, copias inmutables, colas lock-free y estructuras de datos diseñadas para concurrencia con acceso acotado.

Aplicar el patrón RAII, para gestionar automáticamente los recursos asociados a mecanismos de sincronización, ayuda a garantizar que los bloqueos se liberen incluso ante excepciones. Asimismo, es valioso estructurar el código para que la sección crítica sea lo más pequeña posible, permitiendo minimizar el tiempo durante el cual los datos permanecen bloqueados. El control explícito y estrecho sobre la región crítica reduce

tanto la probabilidad de competencia como los efectos colaterales de errores de programación.

Riesgos asociados

- Condiciones de carrera (race conditions), que pueden causar resultados indeterminados o corrupción de datos.
- Interbloqueos (deadlocks) cuando dos o más hilos quedan bloqueados esperando recursos compartidos.
- Pérdida de integridad de los datos debido a accesos concurrentes no controlados.
- Errores difíciles de reproducir y depurar, aumentando la complejidad del mantenimiento.
- Exposición involuntaria de información sensible por estados inconsistentes de los datos.
- Mayor probabilidad de fallos de seguridad como time-of-check to time-of-use (TOCTOU).
- Dificultad para aplicar mecanismos efectivos de sincronización, lo que puede degradar el rendimiento y estabilidad de la aplicación.

Buenas prácticas

- Evitar variables globales modificables y preferir el paso de parámetros constantes.

```
int contadorGlobal = 0;

void incrementar() {
  contadorGlobal++;
}
```

```
//-- Uso de parámetro constante en lugar de estado global compartido
void incrementar(int valorInicial) {
  int contador = valorInicial + 1;
}
```

- Encapsular los datos modificables y exponer solo interfaces controladas.

```
class Configuracion {
public:
  std::string modo;
};
```

```
//-- Encapsulamiento con atributo privado y método controlado
class Configuracion {
private:
  std::string modo;
public:
  void establecerModo(const std::string& nuevoModo) {
    modo = nuevoModo;
  }
  std::string obtenerModo() const {
    return modo;
  }
};
```

- Preferir objetos inmutables o `const` para evitar modificaciones accidentales.

```
void procesar(std::string& datos) {
  datos = "modificado";
}
```

```
//-- Uso de referencia constante para impedir cambios en datos compartidos
void procesar(const std::string& datos) {
  std::string copia = datos;
}
```

- Utilizar almacenamiento local o `thread_local` en entornos concurrentes.

```
int contador = 0;
void incrementar() {
  contador++;
}
```

```
//-- Uso de almacenamiento específico por hilo para evitar estado compartido
thread_local int contador = 0;

void incrementar() {
  contador++;
}
```

2.15.3.2 Estándares metodológicos

- **C++ Core-CP.3**: Minimize explicit sharing of writable data.

2.15.4 [cpp.sync.4] - Evitar usar "volatile" como mecanismo de sincronización

ID	cpp.sync.4	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Se debe evitar el uso de `volatile` como mecanismo de sincronización, ya que su función real se limita a impedir ciertas optimizaciones del compilador y no proporciona garantías de orden ni atomicidad. Para asegurar un comportamiento concurrente correcto y seguro, deben emplearse primitivas de sincronización diseñadas para tal fin, como `std::atomic`, `std::mutex` u otros mecanismos adecuados de la biblioteca estándar. El uso incorrecto de `volatile` puede producir condiciones de carrera, corrupción de memoria y comportamientos indefinidos.

Codificación insegura

Uso de una variable compartida entre hilos para indicar cuándo una tarea ha finalizado. Se busca mostrar cómo se intenta coordinar la ejecución mediante una bandera compartida, permitiendo que un hilo espere hasta que otro cambie su valor.

```
#include <iostream>
#include <thread>
#include <chrono>

volatile bool ready = false;

void worker() {
  //-- comentario
  // Simula trabajo y luego cambia la bandera
  std::this_thread::sleep_for(std::chrono::milliseconds(500));
  ready = true;
}

int main() {
  //-- comentario
  // Lanza un hilo que modificará la variable compartida
  std::thread t(worker);

  //-- comentario
  // Espera activa hasta que la bandera cambie
  while (!ready) {
    // Espera ocupada
  }

  std::cout << "Trabajo finalizado" << std::endl;
  t.join();
  return 0;
```

```
}
```

Codificación recomendada

Uso de un mecanismo de sincronización adecuado para coordinar hilos mediante una bandera compartida. Se muestra cómo garantizar visibilidad y ausencia de condiciones de carrera utilizando primitivas diseñadas para concurrencia.

```
#include <iostream>
#include <thread>
#include <chrono>
#include <atomic>

std::atomic<bool> ready(false);

void worker() {
  //-- comentario
  // Simula trabajo y luego actualiza la bandera de forma segura
  std::this_thread::sleep_for(std::chrono::milliseconds(500));
  ready.store(true, std::memory_order_release);
}

int main() {
  //-- comentario
  // Lanza un hilo que modificará la variable compartida
  std::thread t(worker);

  //-- comentario
  // Espera hasta que la bandera cambie usando operaciones atómicas
  while (!ready.load(std::memory_order_acquire)) {
    // Espera ocupada controlada
  }

  std::cout << "Trabajo finalizado" << std::endl;
  t.join();
  return 0;
}
```

2.15.4.1 Descripción

El uso del especificador `volatile` se ha interpretado erróneamente durante años como un mecanismo de sincronización para resolver problemas de concurrencia. Sin embargo, **esto no es correcto**. El propósito real de `volatile` es indicar al compilador que una variable puede cambiar de forma inesperada debido a factores externos, como interrupciones de hardware, acceso desde memoria mapeada o registros especiales. No garantiza orden de memoria, atomicidad, visibilidad entre hilos ni protección frente a condiciones de carrera. Por ello, su empleo como herramienta de sincronización

constituye una práctica peligrosa que compromete la integridad de los datos en escenarios concurrentes.

En el contexto de la programación concurrente, resulta imprescindible utilizar herramientas específicamente diseñadas para ello, como **operaciones atómicas** mediante `std::atomic` o mecanismos de bloqueo explícito, entre los que destacan `std::mutex`, `std::scoped_lock` y `std::lock_guard`. Estos elementos proporcionan las garantías formales definidas por el estándar en materia de orden, sincronización de memoria y ausencia de condiciones de carrera siempre que el código se diseñe adecuadamente. El compilador y la CPU pueden reordenar operaciones para optimizar el rendimiento, y el uso de `volatile` no impide estos reordenamientos, lo cual puede producir resultados inesperados cuando varios hilos leen o escriben la misma variable.

Para ilustrar esta problemática, considérese una variable global marcada como `volatile` utilizada como señal entre hilos. Aunque cada acceso a la variable se cargue desde memoria y no desde un registro, **no existe garantía de que las operaciones precedentes o posteriores en cada hilo conserven el orden esperado**. Por ello, un hilo podría observar una actualización en un estado de memoria incoherente, lo que provocaría resultados imprevisibles o fallos difíciles de depurar.

Riesgos asociados

- Condiciones de carrera debido a que `volatile` no garantiza exclusión mutua ni atomicidad en operaciones sobre datos compartidos.
- Lecturas y escrituras inconsistentes de variables compartidas entre hilos, provocando comportamientos indefinidos.
- Fallas intermitentes y errores difíciles de reproducir.
- Falta de visibilidad de los cambios realizados por un hilo para otros hilos, ya que `volatile` no impone barreras de memoria.
- Exposición a vulnerabilidades de seguridad por el acceso concurrente y no protegido a recursos críticos.
- Malinterpretación del objetivo del especificador `volatile`, generando confianza infundada en la seguridad de la concurrencia.
- Dificultad para mantener y depurar el código debido a errores sutiles relacionados con sincronización.

Buenas prácticas

- Utilizar `std::atomic` para variables compartidas entre hilos en lugar de `volatile`.

```
//-- Uso de volatile para comunicación entre hilos
volatile bool flag = false;
```

```
void worker() {
  flag = true;
}

void wait() {
  while(!flag) { }
}
```

```
//-- Uso de std::atomic para sincronización segura
#include <atomic>

std::atomic<bool> flag(false);

void worker() {
  flag.store(true, std::memory_order_release);
}

void wait() {
  while(!flag.load(std::memory_order_acquire)) { }
}
```

- Emplear `std::mutex` para proteger secciones críticas en lugar de depender de `volatile`.

```
//-- Incremento compartido protegido solo con volatile
volatile int counter = 0;

void increment() {
  counter++;
}
```

```
//-- Protección mediante std::mutex y std::lock_guard
#include <mutex>

int counter = 0;
std::mutex mtx;

void increment() {
  std::lock_guard<std::mutex> lock(mtx);
  counter++;
}
```

- Utilizar mecanismos de sincronización como `std::condition_variable` en lugar de bucles de espera activa con `volatile`.

```
//-- Espera activa basada en volatile
volatile bool ready = false;
```

```
void wait() {
  while(!ready) { }
}
```

```
//-- Sincronización mediante std::condition_variable
#include <mutex>
#include <condition_variable>

bool ready = false;
std::mutex mtx;
std::condition_variable cv;

void wait() {
  std::unique_lock<std::mutex> lock(mtx);
  cv.wait(lock, [] { return ready; });
}
```

- Reservar el uso de `volatile` exclusivamente para acceso a memoria mapeada o hardware, no para sincronización entre hilos.

```
//-- Uso de volatile para sincronización entre hilos
volatile int sharedData = 0;

void writer() {
  sharedData = 42;
}
```

```
//-- Uso de volatile para acceso a registro de hardware
int readRegister() {
  volatile int* reg = reinterpret_cast<volatile int*>(0x40000000);
  return *reg;
}
```

2.15.4.2 Estándares metodológicos

- **C++ Core-CP.8**: don't try to use volatile for synchronization.

2.15.5 [cpp.sync.5] - Gestionar los bloqueos mediante RAII

ID	cpp.sync.5	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Los bloqueos y recursos asociados a la sincronización deben gestionarse mediante el patrón **RAII**, garantizando así la liberación automática y segura de mutex, semáforos o cualquier recurso crítico incluso ante excepciones u órdenes de retorno inesperadas. Al hacerlo, se evita que los hilos entren en estados de espera indefinida o condiciones de carrera provocadas por bloqueos no liberados correctamente.

Codificación insegura

Gestión manual de un mutex en una función que protege un recurso compartido. Se busca mostrar cómo un flujo de ejecución con excepciones puede dejar el recurso bloqueado si no se libera correctamente, afectando a la concurrencia y estabilidad del sistema.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <stdexcept>

std::mutex g_mutex;
int g_counter = 0;

void incrementar(bool provocarError) {
  //-- Bloqueo manual del mutex
  g_mutex.lock();

  //-- Sección crítica: modificación de recurso compartido
  g_counter++;

  //-- Simulación de error durante la ejecución
  if (provocarError) {
    //-- Excepción antes de liberar el mutex
    throw std::runtime_error("Error simulado");
  }

  //-- Liberación manual del mutex
  g_mutex.unlock();
}

int main() {
  try {
    incrementar(true);
```

```
  }
  catch (const std::exception& e) {
    //-- Captura de la excepción
    std::cout << "Excepcion capturada: " << e.what() << std::endl;
  }

  //-- Intento posterior de uso del recurso compartido
  std::cout << "Valor del contador: " << g_counter << std::endl;

  return 0;
}
```

Codificación recomendada

Gestión automática del mutex utilizando el patrón RAII mediante `std::lock_guard`. Se pretende demostrar cómo el bloqueo y desbloqueo quedan asociados al ciclo de vida de un objeto, garantizando la liberación del recurso incluso cuando se producen excepciones.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <stdexcept>

std::mutex g_mutex;
int g_counter = 0;

void incrementar(bool provocarError) {
  //-- Bloqueo automático del mutex mediante RAII
  std::lock_guard<std::mutex> lock(g_mutex);

  //-- Sección crítica: modificación de recurso compartido
  g_counter++;

  //-- Simulación de error durante la ejecución
  if (provocarError) {
    //-- Excepción dentro de la sección crítica
    throw std::runtime_error("Error simulado");
  }

  //-- No es necesario liberar manualmente el mutex
}

int main() {
  try {
    incrementar(true);
  }
  catch (const std::exception& e) {
```

```
    //-- Captura de la excepción
    std::cout << "Excepcion capturada: " << e.what() << std::endl;
  }

  //-- El mutex ya ha sido liberado automáticamente
  std::cout << "Valor del contador: " << g_counter << std::endl;

  return 0;
}
```

2.15.5.1 Descripción

En entornos multihilo, los errores derivados del olvido al liberar un `mutex`, de rutas de código que retornan de forma inesperada o de excepciones que interrumpen el flujo normal de ejecución pueden generar condiciones altamente peligrosas: deadlocks, corrupción de estados compartidos o acceso inconsistente a estructuras críticas. Con RAII, la adquisición del recurso queda ligada al ciclo de vida del objeto, lo que asegura que la liberación se produzca automáticamente al abandonar el ámbito, independientemente del camino que siga la ejecución.

Para los bloqueos, el lenguaje ofrece clases como `std::lock_guard`, `std::unique_lock` y `std::scoped_lock`, todas diseñadas bajo el patrón RAII. Estas clases encapsulan la toma y liberación de un `mutex`, eliminando la necesidad de invocar manualmente a `lock()` y `unlock()`. Este enfoque reduce drásticamente el riesgo de errores lógicos en la sincronización y protege el sistema frente a múltiples fallos derivados de la manipulación manual de los bloqueos. Es especialmente relevante en contextos de alta complejidad donde el flujo del programa puede incluir múltiples bifurcaciones, excepciones, salidas tempranas o estructuras de control anidadas.

Una implementación típica del uso seguro de `std::lock_guard` mediante RAII evita comportamientos indeterminados incluso si se lanzan excepciones dentro del ámbito protegido. El patrón fuerza un orden coherente en la adquisición y liberación de los recursos y, lo que es más importante, garantiza que la liberación siempre se realice. Este comportamiento automático hace que RAII sea una herramienta clave para mitigar vulnerabilidades derivadas de la manipulación incorrecta de primitivas de sincronización.

Al aplicar RAII, también se reduce la probabilidad de errores relacionados con la omisión accidental de liberaciones. Por ejemplo, en un manejo manual sin RAII, una función con múltiples retornos podría pasar por alto la llamada a `unlock()`. Asimismo, si una excepción se propaga antes de liberar el bloqueo, el programa quedaría en un estado inconsistente, generando fallos graves o bloqueos permanentes. Sin embargo, con RAII, la destrucción del objeto guardián ejecuta automáticamente la liberación del recurso, neutralizando este tipo de riesgos.

Otra ventaja importante es la claridad semántica. El código que utiliza RAII transmite explícitamente la intención del desarrollador: el bloqueo está acotado al ámbito del objeto RAII. Este enfoque evita efectos colaterales o fugas de alcance en las que un recurso permanece bloqueado más tiempo del previsto, lo que puede provocar que otros hilos queden esperando indefinidamente. Asimismo, la adopción de **RAII reduce la superficie de ataque** al impedir manipulaciones inseguras que podrían permitir a un atacante causar denegaciones de servicio mediante la explotación de bloqueos mal gestionados.

En sistemas complejos, el uso de RAII combinado con `std::unique_lock` permite mayor flexibilidad. Por ejemplo, permite bloquear de forma diferida, liberar temporalmente un recurso o realizar operaciones avanzadas en combinación con primitivas como `std::condition_variable`. Esta flexibilidad no elimina las garantías de seguridad proporcionadas por RAII, sino que las amplía, otorgando un control más preciso sobre la sincronización sin sacrificar la correcta liberación del recurso. De este modo, incluso en arquitecturas con hilos que esperan señales, RAII previene fugas de bloqueo y estados inconsistentes.

El uso de RAII también resulta fundamental para evitar errores derivados de intentos de bloqueo múltiples sobre un mismo recurso por parte del mismo hilo, o de desbloqueos duplicados que pueden resultar en comportamientos indefinidos. Las clases RAII impiden directamente estas prácticas, ya que su diseño se basa en la posesión exclusiva y temporal del recurso. Al apoyarse en la semántica de los objetos y en la destrucción automática al final del ámbito, RAII mitiga problemas clásicos de la concurrencia que pueden resultar extremadamente difíciles de detectar durante las pruebas o auditorías de código.

Para escenarios donde se requiere la adquisición simultánea de múltiples bloqueos, el uso de `std::scoped_lock` permite la prevención de deadlocks mediante la adquisición ordenada de múltiples `mutex`. Este patrón es particularmente vital en aplicaciones críticas donde la integridad de los datos y la disponibilidad del sistema debe mantenerse bajo cargas concurrentes intensivas. Con RAII, el riesgo de bloqueos cruzados se reduce significativamente, ya que la liberación de los recursos se produce de forma coordinada y garantizada al finalizar el ámbito.

Riesgos asociados

- Condiciones de carrera por liberación no garantizada de bloqueos.
- Pérdida de bloqueos en casos de excepciones, lo que puede provocar estados inconsistentes o deadlocks.
- Deadlocks por olvidar liberar manualmente los bloqueos.
- Fugas de recursos, ya que los bloqueos pueden no ser liberados correctamente.
- Dificultad para mantener y auditar el código, aumentando la probabilidad de

errores de concurrencia.

- Incremento en la complejidad del código por la gestión manual de los bloqueos.
- Vulnerabilidades de seguridad explotables por estados inconsistente o recursos mal gestionados.

Buenas prácticas

- Utilizar `std::lock_guard` para gestionar automáticamente la adquisición y liberación de bloqueos.

```
//-- Bloqueo y desbloqueo manual propenso a omisiones
std::mutex m;
void f() {
  m.lock();
  procesoCritico();
  m.unlock();
}
```

```
//-- Uso de std::lock_guard para liberar automáticamente al salir del ámbito
std::mutex m;
void f() {
  std::lock_guard<std::mutex> lock(m);
  procesoCritico();
}
```

- Preferir `std::unique_lock` cuando se requiera flexibilidad adicional en la gestión del bloqueo.

```
//-- Gestión manual con múltiples puntos de salida
std::mutex m;
void f() {
  m.lock();
  if(condicionError()) {
    m.unlock();
    return;
  }
  procesoCritico();
  m.unlock();
}
```

```
//-- Uso de std::unique_lock con liberación automática y control explícito
std::mutex m;
void f() {
  std::unique_lock<std::mutex> lock(m);
  if(condicionError()) {
    return;
```

```
  }
  procesoCritico();
}
```

- Evitar el uso de `lock()` y `unlock()` en bloques separados, encapsulando el bloqueo en objetos RAII.

```
//-- Bloqueo separado del ámbito lógico del recurso protegido
std::mutex m;
void f() {
  m.lock();
  procesoCritico();
}
void g() {
  m.unlock();
}
```

```
//-- Encapsulación del bloqueo en el mismo ámbito mediante RAII
std::mutex m;
void f() {
  std::lock_guard<std::mutex> lock(m);
  procesoCritico();
}
```

2.15.5.2 Estándares metodológicos

- **C++ Core-CP.20**: Use RAII, never plain lock()/unlock().

2.15.6 [cpp.sync.6] - Evitar llamar a código no controlado durante un bloqueo

ID	**cpp.sync.6**	**Uso**	**Obligatorio**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

La ejecución de funciones o callbacks externos mientras se mantiene un bloqueo activo, puede provocar bloqueos mutuos, condiciones de carrera, reentrancia insegura, degradación del rendimiento y vectores de explotación basados en inyección de comportamiento. Mantener el código ejecutado dentro de una región crítica estrictamente controlado mitiga riesgos importantes asociados a la invocación de lógica no confiable o extensible, preservando la estabilidad, predictibilidad y seguridad del sistema.

Codificación insegura

Se muestra una clase que protege un recurso compartido con `std::mutex` y permite registrar un callback externo. La intención es ilustrar qué ocurre cuando se invoca código proporcionado por terceros mientras el bloqueo sigue activo, generando riesgo de interbloqueo o bloqueo prolongado.

```
#include <iostream>
#include <mutex>
#include <functional>

class ResourceManager {
  //-- Mutex para proteger el recurso compartido
  std::mutex mtx;

  //-- Callback externo que no controlamos
  std::function<void()> callback;

public:
  void setCallback(const std::function<void()>& cb) {
    //-- Se asigna el callback sin validaciones adicionales
    callback = cb;
  }

  void process() {
    //-- Se adquiere el bloqueo del mutex
    std::lock_guard<std::mutex> lock(mtx);

    //-- Se ejecuta código crítico protegido
    std::cout << "Procesando recurso..." << std::endl;

    //-- Se invoca código externo mientras el bloqueo sigue activo
    if (callback) {
      callback();
    }
  }
};

int main() {
  ResourceManager manager;

  //-- Se registra un callback que podría intentar acceder al mismo recurso
  manager.setCallback([&manager]() {
    std::cout << "Callback ejecutado" << std::endl;
  });

  manager.process();

  return 0;
```

```
}
```

Codificación recomendada

Se muestra la misma funcionalidad, pero separando la sección crítica protegida por el mutex de la ejecución del callback externo. La intención es aprender a reducir el tiempo de bloqueo y evitar la ejecución de código no controlado dentro de una región crítica.

```
#include <iostream>
#include <mutex>
#include <functional>

class ResourceManager {
  //-- Mutex para proteger el recurso compartido
  std::mutex mtx;

  //-- Callback externo que no controlamos
  std::function<void()> callback;

public:
  void setCallback(const std::function<void()>& cb) {
    //-- Se protege la asignación del callback
    std::lock_guard<std::mutex> lock(mtx);
    callback = cb;
  }

  void process() {
    std::function<void()> localCallback;

    {
      //-- Se adquiere el bloqueo solo para la sección crítica
      std::lock_guard<std::mutex> lock(mtx);

      //-- Se ejecuta código crítico protegido
      std::cout << "Procesando recurso..." << std::endl;

      //-- Se copia el callback para usarlo fuera del bloqueo
      localCallback = callback;
    }

    //-- Se ejecuta el callback fuera de la región crítica
    if (localCallback) {
      localCallback();
    }
  }
};

int main() {
```

```
  ResourceManager manager;

  //-- Se registra un callback externo
  manager.setCallback([&manager]() {
    std::cout << "Callback ejecutado" << std::endl;
  });

  manager.process();

  return 0;
}
```

2.15.6.1 Descripción

Mantener bloqueos (por ejemplo, `std::mutex`, `std::shared_mutex` u otros mecanismos de sincronización) mientras se llama a código no controlado introduce riesgos fundamentales para la seguridad y la estabilidad del sistema. Estos riesgos se derivan de la imposibilidad de garantizar el comportamiento del código externo, incluyendo posibles demoras, fallos inesperados, comportamientos reentrantes o intentos deliberados de explotar estados compartidos. Este principio forma parte de las orientaciones críticas para garantizar integridad del flujo de ejecución y evitar vulnerabilidades como el deadlock inducido, la corrupción de datos en memoria compartida y ataques basados en la manipulación de tiempos de espera.

Separar la obtención del bloqueo de la ejecución de código externo reduce la superficie de ataque y limita el impacto de comportamientos inesperados. Un atacante que controle parte del flujo de extensión del programa no podrá manipular el tiempo de retención del bloqueo, ni provocar estados inconsistentes mientras el recurso protegido está en un estado temporalmente inestable. Además, este enfoque facilita garantizar propiedades importantes como atomicidad, no reentrancia y consistencia interna de los datos compartidos.

Para sistemas de alto rendimiento o con grandes volúmenes de concurrencia, la aplicación disciplinada de este principio permite asegurar que la contención de bloqueos no derive en efectos cascada, que pueden degradar severamente la disponibilidad del sistema. La segmentación de responsabilidades, el uso de estructuras lock-free cuando sea posible y la definición precisa de invariantes reducen riesgos sistémicos y ayudan a mantener un modelo de ejecución predecible.

Riesgos asociados

- Deadlocks (interbloqueos) inesperados por ejecución de código externo que también intente adquirir el mismo u otros bloqueos.
- Inconsistencias de estado si el código no controlado realiza operaciones que afectan recursos compartidos durante el bloqueo.

- Condiciones de carrera debido a la pérdida de control sobre el orden de ejecución del código externo bajo el bloqueo.
- Vulnerabilidades de denegación de servicio, ya que el código no controlado puede realizar operaciones prolongadas y mantener el bloqueo, deteniendo otros hilos.
- Elevación de privilegios si el código no controlado manipula recursos protegidos o el contexto del hilo bloqueado.
- Dificultad en la depuración y el mantenimiento por la imprevisibilidad introducida por el código externo bajo el bloqueo.
- Introducción de bugs secundarios y regresiones derivadas por el comportamiento no previsto del código externo llamado durante el bloqueo.
- Posibilidad de violaciones de seguridad si el código externo realiza acciones no validadas o maliciosas mientras se tiene el bloqueo.

Buenas prácticas

- Evitar invocar callbacks o funciones externas mientras se mantiene un mutex bloqueado.

```
//-- Se invoca código externo mientras el mutex está bloqueado
std::mutex m;
void procesar(std::function cb) {
  std::lock_guard<std::mutex> lock(m);
  cb();
}
```

```
//-- Se libera el mutex antes de invocar código externo
std::mutex m;
void procesar(std::function cb) {
  {
    std::lock_guard<std::mutex> lock(m);
    // preparar estado interno
  }
  cb();
}
```

- Copiar los datos necesarios bajo el bloqueo y procesarlos fuera de la sección crítica.

```
//-- Se procesa directamente el contenedor compartido bajo bloqueo
std::mutex m;
std::vector<int> datos;
void imprimir() {
  std::lock_guard<std::mutex> lock(m);
```

```
  for (int v : datos) {
    std::cout << v << std::endl;
  }
}
```

```
//-- Se copia el contenedor bajo bloqueo y se procesa fuera
std::mutex m;
std::vector<int> datos;
void imprimir() {
  std::vector<int> copia;
  {
    std::lock_guard<std::mutex> lock(m);
    copia = datos;
  }
  for (int v : copia) {
    std::cout << v << std::endl;
  }
}
```

- Evitar destructores de objetos externos mientras se mantiene un bloqueo.

```
//-- El objeto externo se destruye mientras el mutex está bloqueado
std::mutex m;
void eliminar(std::unique_ptr<Recurso> r) {
  std::lock_guard<std::mutex> lock(m);
  r.reset();
}
```

```
//-- El recurso se libera después de salir de la sección crítica
std::mutex m;
void eliminar(std::unique_ptr<Recurso> r) {
  std::unique_ptr<Recurso> tmp;
  {
    std::lock_guard<std::mutex> lock(m);
    tmp = std::move(r);
  }
  tmp.reset();
}
```

- Definir claramente la sección crítica para minimizar el tiempo de bloqueo antes de llamadas potencialmente bloqueantes.

```
//-- Se realiza una operación de red mientras el mutex está bloqueado
std::mutex m;
void enviar(Socket& s, const std::string& msg) {
  std::lock_guard<std::mutex> lock(m);
  s.send(msg);
```

```
}
```

```
//-- Se limita la sección crítica y la operación externa ocurre fuera
std::mutex m;
void enviar(Socket& s, const std::string& msg) {
  std::string copia;
  {
    std::lock_guard<std::mutex> lock(m);
    copia = msg;
  }
  s.send(copia);
}
```

2.15.6.2 Estándares metodológicos

- **C++ Core-CP.22**: Never call unknown code while holding a lock (e.g., a callback).

2.15.7 [cpp.sync.7] - Evitar producir interbloqueos bloqueando recursos en orden inconsistente

ID	cpp.sync.7	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Disponibilidad	S-SDLC	Build

El orden de adquisición de recursos, si está mal implementado, puede comprometer la **disponibilidad** del sistema, bloquear hilos críticos y provocar fallos en cascada. Para mitigarlo, se deben definir reglas estables de bloqueo, utilizar mecanismos atómicos cuando sea posible y aplicar patrones de diseño que reduzcan los riesgos asociados a la concurrencia.

Codificación insegura

Demostración de cómo dos hilos adquieren los mismos `mutex` en distinto orden, generando una condición clásica de interbloqueo. Se pretende observar cómo una diferencia en el orden de bloqueo puede provocar que ambos hilos queden bloqueados indefinidamente esperando recursos que nunca se liberan.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

std::mutex m1;
std::mutex m2;
```

```
void tarea1() {
  //-- Bloquea primero m1
  m1.lock();
  std::this_thread::sleep_for(std::chrono::milliseconds(100));

  //-- Luego intenta bloquear m2
  m2.lock();

  std::cout << "Tarea 1 ejecutada" << std::endl;

  m2.unlock();
  m1.unlock();
}

void tarea2() {
  //-- Bloquea primero m2
  m2.lock();
  std::this_thread::sleep_for(std::chrono::milliseconds(100));

  //-- Luego intenta bloquear m1
  m1.lock();

  std::cout << "Tarea 2 ejecutada" << std::endl;

  m1.unlock();
  m2.unlock();
}

int main() {
  //-- Lanza dos hilos que pueden quedar en interbloqueo
  std::thread h1(tarea1);
  std::thread h2(tarea2);

  h1.join();
  h2.join();

  return 0;
}
```

Codificación recomendada

Demostración de cómo evitar el interbloqueo asegurando un orden consistente en la adquisición de recursos compartidos. Ambos hilos bloquean los `mutex` siguiendo la misma secuencia lógica, eliminando la condición circular que provoca el bloqueo mutuo.

```
#include <iostream>
#include <thread>
#include <mutex>
```

```
#include <chrono>

std::mutex m1;
std::mutex m2;

void tarea1() {
  //-- Bloquea los mutex en un orden consistente
  std::lock(m1, m2);
  std::lock_guard<std::mutex> lock1(m1, std::adopt_lock);
  std::lock_guard<std::mutex> lock2(m2, std::adopt_lock);

  std::cout << "Tarea 1 ejecutada" << std::endl;
}

void tarea2() {
  //-- Usa el mismo orden de bloqueo que tarea1
  std::lock(m1, m2);
  std::lock_guard<std::mutex> lock1(m1, std::adopt_lock);
  std::lock_guard<std::mutex> lock2(m2, std::adopt_lock);

  std::cout << "Tarea 2 ejecutada" << std::endl;
}

int main() {
  //-- Lanza dos hilos sin riesgo de interbloqueo
  std::thread h1(tarea1);
  std::thread h2(tarea2);

  h1.join();
  h2.join();

  return 0;
}
```

2.15.7.1 Descripción

Un interbloqueo ocurre cuando dos o más hilos esperan indefinidamente recursos que nunca serán liberados porque otros hilos los poseen y también están bloqueados. Este requisito de seguridad se centra en la necesidad de establecer un ordenamiento coherente, estable y universal en el acceso a recursos sincronizados, especialmente cuando se utilizan mecanismos como `std::mutex`, `std::recursive_mutex`, `std::shared_mutex` u otros mecanismos equivalentes. El enfoque principal consiste en minimizar la complejidad del diseño de concurrencia, reducir la probabilidad de interacciones inseguras y garantizar que todos los componentes del sistema respeten un patrón de bloqueo uniforme.

Los interbloqueos emergen típicamente cuando se adquieren múltiples recursos sin una política clara de orden. Si dos hilos intentan tomar los mismos recursos en distinto

orden, es altamente probable que ambos queden bloqueados. Esto sucede incluso con bloques de código pequeños o aparentemente inofensivos. En sistemas críticos, esta condición puede causar desde la pérdida total de capacidad de respuesta hasta reinicios inesperados o indisponibilidad prolongada. La solución más efectiva y universal es definir un orden global de adquisición de recursos y garantizar que todas las rutas de código lo respeten. Además, se deben evitar patrones que deriven en dependencias cíclicas, los cuales son causas frecuentes de interbloqueo en entornos de concurrencia intensiva.

Las estrategias fundamentales para evitar interbloqueos son:

- **Establecer un orden global de bloqueo**: todos los recursos deben ser adquiridos de acuerdo con un orden consistente y documentado.
- **Evitar adquirir recursos innecesarios**: mientras más colocación de bloqueos se realice, mayor será la probabilidad de estructuras cíclicas.
- **Reducir el tiempo de retención de bloqueos**: liberar los bloqueos tan pronto como sea posible disminuye las situaciones de espera prolongada.
- **Utilizar bloqueos atómicos cuando proceda**: en algunos casos, operaciones atómicas pueden sustituir por completo mutexes.
- **Priorizar estructuras RAII**: el uso de `std::lock_guard` y `std::unique_lock` reduce errores de liberación.
- **Evitar diseños con múltiples niveles de bloqueo**: estructuras de dependencia profundas aumentan la complejidad y los riesgos.

La clave principal de este requisito es la consistencia. Si se establece un orden global, pero partes del sistema lo aplican de manera incompleta o errónea, el riesgo de interbloqueo persiste. Por este motivo, es esencial que el diseño de concurrencia sea parte integral y explícita de la arquitectura del software, y que las funciones que acceden concurrentemente a recursos sigan las reglas definidas. Debe considerarse también el impacto de futuras ampliaciones del código, ya que agregar un nuevo mutex o un nuevo patrón de acceso puede introducir relaciones cíclicas incluso si la implementación actual funciona correctamente.

Riesgos asociados

- Interbloqueos (deadlocks) que detienen indefinidamente el funcionamiento de hilos o procesos.
- Caídas o cuelgues de la aplicación debido a la imposibilidad de continuar la ejecución.
- Pérdida de disponibilidad de servicios críticos.
- Mayor dificultad para depurar y mantener el software por comportamientos no deterministas.

- Exposición a ataques de denegación de servicio (DoS) si un atacante fuerza situaciones de interbloqueo.
- Pérdida de datos o corrupción de información si el interbloqueo ocurre durante operaciones críticas.
- Escalado del consumo de recursos del sistema (CPU/memoria) por hilos bloqueados innecesariamente.
- Reducción del rendimiento general de la aplicación y del sistema anfitrión.

Buenas prácticas

- Definir y respetar un orden global y consistente de adquisición de `mutex` en todo el sistema.

```
//-- Bloqueo de mutex en orden inconsistente entre hilos
std::mutex m1, m2;

void tarea1() {
  std::lock_guard l1(m1);
  std::lock_guard l2(m2);
}

void tarea2() {
  std::lock_guard l1(m2);
  std::lock_guard l2(m1);
}
```

```
//-- Bloqueo de mutex siguiendo el mismo orden global
std::mutex m1, m2;

void tarea1() {
  std::lock_guard l1(m1);
  std::lock_guard l2(m2);
}

void tarea2() {
  std::lock_guard l1(m1);
  std::lock_guard l2(m2);
}
```

- Utilizar `std::lock` o `std::scoped_lock` para adquirir múltiples `mutex` de forma atómica.

```
//-- Adquisición manual de múltiples mutex en secuencia
std::mutex m1, m2;

void procesar() {
```

```
  m1.lock();
  m2.lock();
  // uso de recursos
  m2.unlock();
  m1.unlock();
}
```

```
//-- Adquisición conjunta evitando interbloqueos
std::mutex m1, m2;

void procesar() {
  std::scoped_lock lock(m1, m2);
  // uso de recursos
}
```

- Reducir el tiempo de posesión de los bloqueos, evitando llamadas externas o costosas mientras se mantiene un `mutex`.

```
//-- Llamada externa mientras se mantiene el bloqueo
std::mutex m;

void actualizar() {
  std::lock_guard lock(m);
  operacion_costosa();
}
```

```
//-- Liberación del bloqueo antes de la operación costosa
std::mutex m;

void actualizar() {
  {
    std::lock_guard lock(m);
    // actualización de estado compartido
  }
  operacion_costosa();
}
```

- Evitar bloqueos anidados innecesarios rediseñando las funciones para no requerir múltiples recursos simultáneamente.

```
//-- Función que bloquea un segundo mutex dentro de otro
std::mutex m1, m2;

void f2() {
  std::lock_guard l2(m2);
}
```

```
void f1() {
  std::lock_guard l1(m1);
  f2();
}
```

```
//-- Reorganización para evitar bloqueo anidado
std::mutex m1, m2;

void f2_sin_bloqueo();

void f1() {
  std::scoped_lock lock(m1, m2);
  f2_sin_bloqueo();
}
```

2.15.7.2 Estándares metodológicos

- **CERT/CPP-CON53**: Avoid deadlock by locking in a predefined order.

2.15.8 [cpp.sync.8] - Evitar destruir un mutex mientras esté bloqueado

ID	cpp.sync.8	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Cualquier recurso de sincronización de tipo `mutex` no debería ser destruido mientras continúa bloqueado por uno o varios hilos. La destrucción prematura de un `mutex` puede generar comportamientos indefinidos, corrupción de memoria, bloqueos lógicos, condiciones de carrera y fallos de seguridad que afectan la estabilidad y la integridad general de la aplicación. Para mantener una ejecución segura, coherente y predecible, es indispensable verificar siempre el estado del `mutex` antes de su liberación y destrucción, aplicando mecanismos de diseño adecuados.

Codificación insegura

Se muestra el uso de un `std::mutex` gestionado dinámicamente donde se adquiere el bloqueo y posteriormente se destruye el objeto sin liberarlo previamente. La intención es observar el impacto de destruir un `mutex` mientras aún está bloqueado y entender por qué esto provoca comportamiento indefinido.

```
#include <iostream>
#include <mutex>

int main() {
  //-- Crear un mutex dinámicamente
```

```
  std::mutex* mtx = new std::mutex();

  //-- Bloquear el mutex
  mtx->lock();

  //-- Destruir el mutex mientras sigue bloqueado (comportamiento indefinido)
  delete mtx;

  std::cout << "Mutex destruido mientras estaba bloqueado" << std::endl;

  return 0;
}
```

Codificación recomendada

Se presenta el uso seguro de un `std::mutex` empleando el patrón RAII con `std::lock_guard`, garantizando que el bloqueo se libere automáticamente antes de que el `mutex` salga de su ámbito. Se busca demostrar cómo evitar la destrucción de un `mutex` mientras permanece bloqueado.

```
#include <iostream>
#include <mutex>

int main() {
  //-- Crear un mutex en el ámbito automático
  std::mutex mtx;
 {
    //-- Adquirir el bloqueo usando RAII
    std::lock_guard<std::mutex> lock(mtx);
    std::cout << "Mutex bloqueado de forma segura" << std::endl;
    //-- El mutex se desbloquea automáticamente al salir de este bloque
  }

  //-- En este punto el mutex ya está desbloqueado y puede destruirse con seguridad
  std::cout << "Mutex desbloqueado antes de su destrucción" << std::endl;

  return 0;
}
```

2.15.8.1 Descripción

Cuando un `mutex` es destruido mientras permanece bloqueado, la aplicación entra en un estado de comportamiento indefinido que puede manifestarse en corrupciones de datos, abuso de memoria, interbloqueos permanentes y brechas de seguridad. En ecosistemas multi-hilo complejos, este tipo de fallos es especialmente peligroso porque puede pasar inadvertido durante pruebas básicas, manifestándose solo bajo cargas reales o condiciones de concurrencia extremas. Por ello, **hay que asegurar que un** `mutex`

nunca se destruya mientras esté bloqueado constituye un principio fundamental de la programación concurrente segura.

En C++ moderno, la biblioteca estándar proporciona múltiples mecanismos de sincronización, siendo `std::mutex` uno de los más comunes. Su gestión correcta implica adoptar patrones RAII, utilizar objetos contenedores seguros y establecer reglas de propiedad claras para determinar cuándo se crea y cuándo se destruye un `mutex`. Una destrucción inapropiada suele producirse cuando el desarrollador no controla adecuadamente el ciclo de vida del recurso o cuando un error lógico provoca que un hilo no libere el bloqueo antes de eliminar la instancia. Esto puede suceder, por ejemplo, si se utiliza memoria dinámica sin punteros inteligentes, se generan caminos de ejecución no previstos o se emplean estructuras globales con orden de destrucción incierto.

Una práctica esencial consiste en encapsular los `mutex` dentro de clases diseñadas para garantizar su correcta liberación. Esto se complementa con herramientas como `std::lock_guard` o `std::unique_lock`, que liberan automáticamente el bloqueo cuando el objeto sale de su ámbito. Sin embargo, incluso con RAII, si la instancia del `mutex` es destruida antes de que todos los hilos liberan el bloqueo, la aplicación puede fallar de manera catastrófica. Por ello, **es indispensable definir políticas de propiedad estables que impidan que un `mutex` sea accesible o destruido por partes no autorizadas del código**.

Asimismo, **los contenedores estándar y estructuras compartidas deben evitar almacenar punteros crudos a `mutex`**. En su lugar, debe preferirse la composición directa o el uso de punteros inteligentes siempre que sea compatible con el diseño. La razón de esta buena práctica es que un puntero crudo facilita que partes inesperadas del sistema eliminen el recurso sin sincronizarse previamente, introduciendo vulnerabilidades serias. Además, el uso de `shared_ptr` para `mutex` es particularmente delicado y suele desaconsejarse, ya que permite aliasing entre múltiples propietarios sin garantizar disciplina en el manejo de los bloqueos.

Otra consideración importante es evitar trasladar o mover objetos que contengan `mutex` una vez que están en uso. La biblioteca estándar prohíbe copiar `std::mutex`, pero permite mover ciertos objetos contenedores. Sin una comprensión profunda, mover un objeto con un `mutex` puede destruir implícitamente el original mientras sigue bloqueado, lo cual constituye un riesgo grave. Por ello, se recomienda que toda clase que incorpore un `mutex` declare constructores de movimiento eliminados para evitar estos escenarios peligrosos.

Riesgos asociados

- Condiciones de carrera (race conditions) que pueden conducir a comportamientos indefinidos.
- Bloqueo del programa (deadlock) si otros hilos esperan el `mutex` destruido.
- Accesos a memoria inválida, provocando fallos de segmentación (segmentation

faults).

- Corrupción de datos compartidos por falta de sincronización.
- Difícil detección y depuración de errores relacionados con concurrencia.
- Exposición a posibles vulnerabilidades de seguridad debido a la ejecución en condiciones inesperadas.
- Pérdida de estabilidad y confianza en la aplicación por comportamiento impredecible.

Buenas prácticas

- Desbloquear explícitamente el `mutex` antes de que su objeto sea destruido.

```
//-- Destrucción del mutex mientras permanece bloqueado
std::mutex* m = new std::mutex();
m->lock();
delete m;
```

```
//-- Liberación del bloqueo antes de destruir el mutex
std::mutex* m2 = new std::mutex();
m2->lock();
m2->unlock();
delete m2;
```

- Utilizar RAII (`std::lock_guard` o `std::unique_lock`) para garantizar la liberación automática antes de la destrucción.

```
//-- Mutex global destruido al finalizar el programa mientras sigue bloqueado
std::mutex m;
void f() {
  m.lock();
}
```

```
//-- Uso de lock_guard para asegurar liberación automática al salir del alcance
std::mutex m2;
void f2() {
  std::lock_guard lock(m2);
}
```

2.15.8.2 Estándares metodológicos

- **CERT/CPP-CON50**: Do not destroy a mutex while it is locked.

2.15.9 [cpp.sync.9] - Evitar desacoplar hilos mediante detach

ID	cpp.sync.9	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

El uso de `std::thread::detach` desacopla la ejecución del hilo y dificulta supervisar su finalización, manejar excepciones, asegurar la liberación correcta de recursos y detectar fallos en tiempo de ejecución. En entornos críticos o de alta seguridad, el uso de hilos desacoplados puede generar comportamientos impredecibles y condiciones de carrera difíciles de diagnosticar. Se exige emplear alternativas más seguras como mecanismos RAII, hilos unidos mediante `join` o pools de hilos controlados.

Codificación insegura

Programa que lanza un hilo para procesar datos y finaliza el flujo principal sin esperar su terminación. Se busca mostrar qué ocurre cuando el hilo se desacopla y continúa ejecutándose sin control sobre su ciclo de vida ni sobre los recursos que utiliza.

```
#include <iostream>
#include <thread>
#include <chrono>

void tarea(int& valor) {
  //-- Simula trabajo en segundo plano
  std::this_thread::sleep_for(std::chrono::seconds(2));
  //-- Accede a un recurso que puede no existir
  std::cout << "Valor procesado: " << valor << std::endl;
}

int main() {
  //-- Variable local compartida con el hilo
  int dato = 42;

  //-- Se crea el hilo y se desacopla
  std::thread hilo(tarea, std::ref(dato));
  hilo.detach();

  //-- El hilo principal termina sin esperar al secundario
  std::cout << "Fin del programa principal" << std::endl;
  return 0;
}
```

Codificación recomendada

Programa que lanza un hilo para procesar datos y asegura su finalización antes de salir del ámbito donde existen los recursos compartidos. Se pretende aprender a mantener

el control del ciclo de vida del hilo y evitar accesos a memoria liberada o estados inconsistentes.

```
#include <iostream>
#include <thread>
#include <chrono>

void tarea(int& valor) {
  //-- Simula trabajo en segundo plano
  std::this_thread::sleep_for(std::chrono::seconds(2));
  //-- Accede de forma segura al recurso mientras sigue vigente
  std::cout << "Valor procesado: " << valor << std::endl;
}

int main() {
  //-- Variable local compartida con el hilo
  int dato = 42;

  //-- Se crea el hilo y se mantiene el control sobre él
  std::thread hilo(tarea, std::ref(dato));

  //-- Se espera a que el hilo termine antes de salir del ámbito
  if (hilo.joinable()) {
    hilo.join();
  }

  //-- El recurso sigue siendo válido hasta finalizar el hilo
  std::cout << "Fin del programa principal" << std::endl;
  return 0;
}
```

2.15.9.1 Descripción

El uso de `std::thread::detach` se considera una práctica peligrosa en múltiples escenarios, ya que provoca la desvinculación total del hilo respecto a su creador, impidiendo conocer su estado, su terminación, así como verificar si se ha producido una excepción durante la ejecución. Esta pérdida de control conlleva riesgos significativos de seguridad y fiabilidad en el software.

Cuando un hilo se desacopla mediante `detach`, se ejecuta de forma independiente, sin ninguna obligación para el desarrollador de esperarlo o finalizarlo explícitamente. Esto origina varios problemas: el hilo podría seguir ejecutando código incluso después de que el contexto donde fue lanzado haya quedado inválido, podría producir accesos a memoria ya liberada, podría generar condiciones de carrera no detectadas o incluso podría aplicar operaciones críticas fuera del tiempo o contexto adecuado. **La ausencia de control directo sobre la vida del hilo se traduce en un comportamiento no determinista y, por tanto, inseguro.**

La recomendación del estándar moderno de C++ señala la importancia de diseñar hilos de forma que su ciclo de vida sea explícito, previsible y gestionado adecuadamente. Esto se logra preferentemente mediante `std::thread::join`, mediante el uso de envoltorios RAII como `std::jthread` (C++20) o mediante mecanismos de alto nivel como `std::async`, colas seguras o pools de hilos gestionados. Estos enfoques proporcionan un marco más robusto para el control de errores, la sincronización y la coordinación entre tareas.

Además de los problemas de control, un hilo desacoplado puede esconder excepciones que deben ser tratadas. Si un hilo lanza una excepción no capturada, el proceso puede abortar sin que el hilo principal tenga posibilidad de intervenir. Con hilos unidos o gestionados adecuadamente, las excepciones pueden detectarse, registrarse y tratarse, garantizando trazabilidad y trazabilidad adecuada para auditorías y análisis posteriores.

Otro riesgo relevante es la fuga de recursos. Un hilo que vive más allá del contexto que lo creó puede mantener referencias a objetos ya destruidos, punteros no válidos, manejadores de archivo abiertos o semáforos bloqueados. Estos escenarios pueden ocasionar degradaciones en el rendimiento, corrupción de memoria o bloqueos en tiempo de ejecución. **El patrón de diseño RAII evita este tipo de fallos al garantizar la liberación controlada de recursos y la finalización ordenada de los hilos asociados.**

Riesgos asociados

- Fugas de recursos: el hilo puede continuar ejecutándose después de que el hilo principal haya terminado, causando pérdida de memoria u otros recursos.
- Condiciones de carrera: es difícil sincronizar hilos desacoplados, lo que puede provocar accesos concurrentes inseguros a recursos compartidos.
- Comportamiento indefinido: sin join ni mecanismos de señalización, el hilo puede acceder a datos que ya han sido liberados o modificados por el hilo principal.
- Dificultad para el manejo de excepciones: las excepciones lanzadas en un hilo desacoplado no pueden ser capturadas, lo que puede llevar a fallos silenciosos o impredecibles.
- Pérdida de control del flujo de ejecución: el programa no puede determinar cuándo termina la ejecución de los hilos desacoplados, provocando posibles inconsistencias en el estado de la aplicación.
- Complejidad en la depuración y el mantenimiento: los errores relacionados con hilos desacoplados suelen ser intermitentes y difíciles de reproducir, complicando el diagnóstico y la corrección.

Buenas prácticas

- Gestionar la vida del hilo mediante RAII en lugar de usar `detach()`.

```
//-- Hilo desacoplado sin control de finalización
std::thread t([](){ procesarDatos(); });
t.detach();
```

```
//-- Hilo gestionado automáticamente con RAII
std::jthread t([](){ procesarDatos(); });
```

- Sincronizar explícitamente con `join()` para evitar ejecución fuera de contexto.

```
//-- Hilo desacoplado que puede acceder a recursos destruidos
void iniciar() {
  std::thread t([](){ usarRecursoGlobal(); });
  t.detach();
}
```

```
//-- Sincronización explícita antes de salir del alcance
void iniciar() {
  std::thread t([](){ usarRecursoGlobal(); });
  t.join();
}
```

- Evitar capturar referencias a variables locales en hilos desacoplados.

```
//-- Captura por referencia en hilo desacoplado
void ejecutar() {
  int valor = 10;
  std::thread t([&](){ procesar(valor); });
  t.detach();
}
```

```
//-- Captura por valor y sincronización controlada
void ejecutar() {
  int valor = 10;
  std::thread t([valor](){ procesar(valor); });
  t.join();
}
```

- Propagar errores y excepciones mediante mecanismos controlados en lugar de hilos desacoplados.

```
//-- Excepciones no controladas en hilo desacoplado
std::thread t([](){ lanzarExcepcion(); });
t.detach();
```

```
//-- Manejo controlado usando std::future
```

```
auto fut = std::async(std::launch::async, [](){ lanzarExcepcion(); });
fut.get();
```

- Definir una política clara de cancelación en vez de perder control con `detach()`.

```
//-- Hilo sin mecanismo de cancelación
std::thread t([](){ while(true){ trabajar(); } });
t.detach();
```

```
//-- Hilo con token de cancelación gestionado
std::jthread t([](std::stop_token st){
  while(!st.stop_requested()){ trabajar(); }
});
```

2.15.9.2 Estándares metodológicos

- **C++ Core-CP.26**: don't detach() a thread.

2.15.10 [cpp.sync.10] - Evitar una espera sin una condición asociada

ID	cpp.sync.10	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Disponibilidad	S-SDLC	Build

Se deben evitar esperas activas sin una condición asociada en código concurrente. La ausencia de una condición claramente definida puede generar consumo excesivo de CPU, comportamientos no deterministas, riesgos de bloqueo, condiciones de carrera y brechas de seguridad relacionadas con la disponibilidad y el control del flujo. Mediante el uso adecuado de primitivas de sincronización, como `std::condition_variable`, se garantiza un control seguro, eficiente y correctamente orquestado de los hilos.

Codificación insegura

Gestión de hilos donde uno espera a que otro complete una tarea. Se muestra una implementación que mantiene un hilo en espera activa hasta que una bandera cambia de estado, lo que permite entender el impacto en consumo de CPU y la falta de sincronización adecuada entre hilos.

```
#include <iostream>
#include <thread>
#include <atomic>
#include <chrono>

//-- Bandera global para indicar finalización
std::atomic<bool> listo(false);
```

```
void trabajador() {
  //-- Simula trabajo pesado
  std::this_thread::sleep_for(std::chrono::seconds(2));

  //-- Marca la tarea como completada
  listo.store(true);
}

int main() {
  std::thread t(trabajador);

  //-- Espera activa hasta que la bandera sea verdadera
  while (!listo.load()) {
    //-- Bucle vacío que consume CPU innecesariamente
  }

  std::cout << "Trabajo completado" << std::endl;

  t.join();

  return 0;
}
```

Codificación recomendada

Sincronización entre hilos utilizando una condición asociada a un mecanismo de espera bloqueante. Se emplea una variable de condición para suspender el hilo hasta que se cumpla un predicado específico, evitando consumo innecesario de CPU y asegurando una coordinación correcta.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>

//-- Recursos compartidos protegidos
std::mutex mtx;
std::condition_variable cv;
bool listo = false;

void trabajador() {
  //-- Simula trabajo pesado
  std::this_thread::sleep_for(std::chrono::seconds(2));
   {
    //-- Bloquea el mutex antes de modificar el estado
    std::lock_guard<std::mutex> lock(mtx);
```

```
    listo = true;
  }

  //-- Notifica al hilo en espera
  cv.notify_one();
}

int main() {
  std::thread t(trabajador);

  std::unique_lock<std::mutex> lock(mtx);

  //-- Espera bloqueante hasta que la condición sea verdadera
  cv.wait(lock, [] { return listo; });

  std::cout << "Trabajo completado" << std::endl;

  lock.unlock();
  t.join();

  return 0;
}
```

2.15.10.1 Descripción

El uso de la concurrencia exige una gestión rigurosa de las esperas entre hilos para evitar riesgos de seguridad y de disponibilidad. Una **espera sin condición asociada**, también llamada espera activa (busy waiting), se produce cuando un hilo permanece en un bucle comprobando repetidamente si se cumple un estado, sin emplear mecanismos seguros de sincronización que garanticen una transición controlada. Esta práctica no solo consume recursos innecesarios, sino que además abre la puerta a vulnerabilidades asociadas a la denegación de servicio interna, degradación del rendimiento, condiciones de carrera y estados inconsistentes derivados de accesos no coordinados a memoria compartida.

En sistemas concurrentes, los hilos comparten memoria y recursos. Sin una coordinación adecuada, pueden generarse comportamientos indeterminados difíciles de detectar en pruebas y potencialmente explotables. Una espera activa no valida ni sincroniza adecuadamente los eventos, y su ejecución puede entrar en conflicto con el planificador del sistema operativo, causando saturación de CPU y disminución de la disponibilidad del servicio. Asimismo, si la espera activa gira sobre una variable accesible desde múltiples hilos sin la protección adecuada, pueden aparecer condiciones de carrera que deriven en corrupción de memoria o fallos lógicos críticos.

Para evitar los riesgos asociados con este patrón inseguro, es fundamental emplear primitivas de sincronización estándar del lenguaje, como `std::condition_variable`, que permiten que un hilo suspenda su ejecución de forma segura hasta que una condición

concreta haya cambiado de estado. Con ello se evita el consumo innecesario de CPU y se garantiza que los hilos no continúen trabajando con estados intermedios o no consistentes. El uso de **bloqueos** correctamente estructurados, en combinación con variables condicionales, es clave para un flujo de ejecución seguro y controlado.

En sistemas embebidos, sistemas de alta disponibilidad o componentes críticos de infraestructura, el impacto de una espera activa puede ser aún mayor. Una sobrecarga de CPU mantiene los núcleos ocupados, lo que limita la capacidad del sistema para gestionar interrupciones, hilos prioritarios o tareas críticas. Además, puede generar un comportamiento cascada que derive en bloqueos, latencias crecientes o fallos generalizados en la plataforma.

Riesgos asociados

- Bucle infinito o situación de espera indefinida (deadlock), bloqueando hilos o procesos.
- Consumo excesivo de CPU por espera activa innecesaria, afectando el rendimiento.
- Vulnerabilidad ante condiciones de carrera por no verificar el estado compartido.
- Agotamiento de recursos del sistema debido al bloqueo prolongado.
- Imposibilidad de recuperación automática ante fallos o ausencia de la condición esperada.
- Difícil diagnóstico y depuración por falta de información sobre el estado esperado para continuar.
- Potenciales ataques de denegación de servicio si un atacante induce la espera infinita.

Buenas prácticas

- Utilizar variables de condición para evitar esperas activas (busy waiting).

```
//-- Espera activa consultando repetidamente una bandera

std::atomic<bool> listo(false);
while(!listo.load()) {
  // espera sin condición bloqueante
}
```

```
//-- Espera bloqueante usando std::condition_variable
std::mutex m;
std::condition_variable cv;
bool listo2 = false;
std::unique_lock<std::mutex> lock(m);
```

```
cv.wait(lock, []{ return listo2; });
```

- Asociar siempre la espera a una condición verificable mediante predicados.

```
//-- Espera sin verificar estado tras despertar
std::unique_lock<std::mutex> lock(m);
cv.wait(lock);
procesar();
```

```
//-- Espera con predicado que valida la condición esperada
std::unique_lock<std::mutex> lock2(m);
cv.wait(lock2, []{ return datosDisponibles; });
procesar();
```

- Utilizar temporizadores en esperas para evitar bloqueos indefinidos.

```
//-- Espera potencialmente indefinida ante fallo de notificación

std::unique_lock<std::mutex> lock(m);
cv.wait(lock, []{ return evento; });
```

```
//-- Espera con tiempo máximo definido para mitigar bloqueo permanente
std::unique_lock<std::mutex> lock2(m);
if(cv.wait_for(lock2, std::chrono::seconds(5), []{ return evento; })) {
  accion();
}
```

2.15.10.2 Estándares metodológicos

- **C++ Core-CP.24**: Think of a thread as a global container.

2.15.11 [cpp.sync.11] - Evitar implementar doble comprobación de bloqueo para inicialización

ID	**cpp.sync.11**	**Uso**	**Obligatorio**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Design**

Se debe evitar implementar el patrón conocido como **double-checked locking (DCL).** Históricamente ha producido condiciones de carrera, estados de inicialización inconsistentes y problemas de memoria no sincronizada. Aunque el lenguaje moderno introduce herramientas más seguras, este patrón sigue siendo propenso a errores incluso para desarrolladores experimentados. Su objetivo es asegurar la correcta inicialización de recursos compartidos mediante mecanismos seguros como

`std::call_once`, inicialización estática garantizada o primitivas de sincronización explícitas correctamente aplicadas.

Codificación insegura

Inicialización perezosa de un objeto compartido usando doble comprobación de bloqueo en un entorno multihilo. Se busca mostrar cómo una implementación manual sin garantías de sincronización adecuadas puede provocar condiciones de carrera y publicación de objetos parcialmente construidos.

```
#include <iostream>
#include <thread>
#include <mutex>

class Logger {
public:
  Logger() {
    //-- Simula inicialización costosa
    std::cout << "Logger inicializado" << std::endl;
  }

  void log(const std::string& msg) {
    std::cout << msg << std::endl;
  }
};

Logger* instance = nullptr;
std::mutex mtx;

Logger* getInstance() {
  //-- Primera comprobación sin sincronización
  if (instance == nullptr) {
    std::lock_guard<std::mutex> lock(mtx);
    //-- Segunda comprobación dentro del bloqueo
    if (instance == nullptr) {
      //-- Posible publicación de objeto parcialmente construido
      instance = new Logger();
    }
  }
  return instance;
}

void worker() {
  Logger* logger = getInstance();
  logger->log("Mensaje desde hilo");
}

int main() {
  std::thread t1(worker);
```

```
  std::thread t2(worker);
  t1.join();
  t2.join();
  return 0;
}
```

Codificación recomendada

Inicialización perezosa de un objeto compartido garantizando seguridad en entornos multihilo mediante mecanismos del estándar que aseguran sincronización correcta y evitan condiciones de carrera durante la construcción.

```
#include <iostream>
#include <thread>
#include <mutex>

class Logger {
public:
  Logger() {
    //-- Simula inicialización costosa
    std::cout << "Logger inicializado" << std::endl;
  }

  void log(const std::string& msg) {
    std::cout << msg << std::endl;
  }
};

Logger& getInstance() {
  //-- Inicialización thread-safe garantizada por el estándar (C++11+)
  static Logger instance;
  return instance;
}

void worker() {
  Logger& logger = getInstance();
  logger.log("Mensaje desde hilo");
}

int main() {
  std::thread t1(worker);
  std::thread t2(worker);
  t1.join();
  t2.join();
  return 0;
}
```

2.15.11.1 Descripción

El **double-checked locking (DCL)** es un patrón de sincronización usado para **inicializar un recurso compartido de forma perezosa (lazy initialization)** reduciendo el coste de usar un `mutex` en cada acceso.

La idea es **comprobar dos veces** si el objeto ya está inicializado:

1) **Primera comprobación sin bloqueo** (rápida).
2) Si el objeto no existe, **se adquiere un** `mutex`.
3) **Segunda comprobación dentro del bloqueo** para evitar que otro hilo lo haya creado mientras se esperaba el `mutex`.
4) Si sigue sin existir, se crea el objeto.

La práctica de implementar un patrón de **double-checked locking** para inicializar objetos de manera diferida ha sido históricamente considerada peligrosa debido a que genera múltiples escenarios de comportamiento indefinido y condiciones de carrera. La premisa del patrón es minimizar la necesidad de bloqueos verificando si un recurso está inicializado antes de adquirir el bloqueo y verificándolo nuevamente después de adquirirlo. Aunque esta técnica pretende optimizar el rendimiento, también se introducen problemas cuando se mezcla con la complejidad del modelo de memoria, la reordenación de instrucciones y la falta de garantías acerca de visibilidad y sincronización entre hilos.

En versiones previas al estándar C++11, no existía un modelo de memoria formalizado capaz de garantizar que la inicialización parcial de un objeto no pudiera ser observada por otros hilos. Incluso con C++11 en adelante, el uso de patrones incorrectos que dependen de la sincronización manual o de la suposición de que operaciones aparentemente atómicas lo son en realidad puede ocasionar estados intermedios visibles y datos compartidos no seguros. Como consecuencia, implementar doble comprobación mediante punteros sin aplicar reglas estrictas de gestión de memoria y sincronización exfolia la superficie de riesgo para vulnerabilidades como corrupción de memoria, condiciones de carrera lógicas y lectura de objetos no completamente construidos.

Para ilustrar el problema, el siguiente fragmento muestra un ejemplo típico del patrón inseguro:

```
if (instance == nullptr) {
    std::lock_guard lock(m);
    if (instance == nullptr) {
        instance = new Singleton();
    }
}
```

En este caso, aunque la segunda comprobación está protegida por un `std::mutex`, nada garantiza que un hilo que detecta `instance != nullptr` observe necesariamente un objeto totalmente construido. Dependiendo del compilador, la arquitectura o el orden de ejecución, la escritura del puntero podría hacerse visible antes que la construcción completa del objeto, permitiendo que otros hilos accedan a un estado inconsistente con consecuencias potencialmente catastróficas.

Un ejemplo de inicialización segura usando inicialización estática del estándar C++ moderno es el siguiente:

```
Singleton& get_instance() {
    static Singleton instance;
    return instance;
}
```

Este enfoque garantiza seguridad en la inicialización sin necesidad de bloqueos adicionales ni verificaciones dobles, ya que el estándar asegura que la inicialización del objeto ocurrirá solo una vez incluso en entornos concurrentes. La simplicidad del patrón evita confusiones y reduce la probabilidad de introducir condiciones de carrera involuntarias.

Otra alternativa ampliamente aceptada es el uso de `std::call_once` junto con un objeto `std::once_flag`. Esto permite que la inicialización de recursos pesados se realice de manera controlada sin permitir duplicación:

```
std::once_flag flag;
Singleton* instance = nullptr;

void init() {
    instance = new Singleton();
}

Singleton* get_instance() {
    std::call_once(flag, init);
    return instance;
}
```

Este patrón elimina la necesidad de comprobaciones múltiples y garantiza la seguridad y consistencia del proceso de inicialización, evitando el riesgo inherente de observar estados parciales de un objeto en construcción. Además, proporciona una semántica clara e inequívoca sobre cuándo y cómo se ejecuta el código crítico.

La razón principal para evitar la doble comprobación radica en que, al intentar optimizar microcostes de rendimiento, se sacrifica la robustez y predictibilidad del comportamiento concurrente. Las optimizaciones prematuras basadas en suposiciones incorrectas sobre el modelo de memoria pueden introducir vulnerabilidades difíciles de detectar, especialmente aquellas que solo se manifiestan bajo condiciones muy

específicas. Estas vulnerabilidades pueden transformarse en fallos de integridad de datos o en oportunidades de explotación cuando un atacante induce condiciones anómalas de ejecución.

En el ámbito de la codificación segura, es fundamental evitar construcciones cuya corrección dependa de detalles sutiles del compilador o del hardware. La doble comprobación de bloqueo es precisamente ese tipo de construcción: fácil de escribir, aparentemente lógica, pero intrínsecamente riesgosa si no se asegura completamente el orden de operaciones y la visibilidad entre hilos. Dado que C++ ofrece mecanismos estándar, eficientes y seguros, no existe justificación válida para continuar utilizando este patrón en código moderno orientado a la seguridad.

Riesgos asociados

- Condiciones de carrera (race conditions) durante la inicialización de variables compartidas.
- Inicialización parcial de objetos, lo que puede causar estados inconsistentes y errores de ejecución.
- Problemas de visibilidad de memoria debido a optimizaciones del compilador o de la CPU, resultando en la lectura de valores no inicializados.
- Accesos simultáneos inseguros a variables compartidas, que pueden derivar en corrupción de datos.
- Dificultad para detectar y depurar errores intermitentes (heisenbugs) causados por la sincronización incorrecta.
- Violación de la garantía de thread safety, generando fallos impredecibles en sistemas concurrentes.
- Dependencia de comportamientos no definidos o no garantizados por el estándar del lenguaje C++.

Buenas prácticas

- Utilizar inicialización estática local garantizada por el estándar (C++11 o superior) en lugar de doble comprobación de bloqueo manual.

```
//-- Inicialización con doble comprobación de bloqueo manual
Singleton* getInstance() {
  if (instance == nullptr) {
    std::lock_guard<std::mutex> lock(mtx);
    if (instance == nullptr) {
      instance = new Singleton();
    }
  }
  return instance;
```

```
}
```

```
//-- Inicialización estática local segura en entorno multihilo
Singleton& getInstance() {
  static Singleton instance;
  return instance;
}
```

- Emplear `std::call_once` y `std::once_flag` para inicialización única controlada en lugar de comprobaciones repetidas.

```
//-- Inicialización con comprobación externa y bloqueo condicional
Resource* getResource() {
  if (resource == nullptr) {
    std::lock_guard<std::mutex> lock(mtx);
    if (resource == nullptr) {
      resource = new Resource();
    }
  }
  return resource;
}
```

```
//-- Inicialización única mediante std::call_once
std::once_flag initFlag;
Resource* getResource() {
  std::call_once(initFlag, []() {
    resource = new Resource();
  });
  return resource;
}
```

- Evitar punteros globales con inicialización diferida y preferir objetos con duración estática o gestión automática de memoria.

```
//-- Uso de puntero global con inicialización diferida y bloqueo
Config* config = nullptr;
Config& getConfig() {
  if (!config) {
    std::lock_guard<std::mutex> lock(mtx);
    if (!config) {
      config = new Config();
    }
  }
  return *config;
}
```

```
//-- Objeto estático con duración automática del programa
Config& getConfig() {
  static Config configInstance;
  return configInstance;
}
```

2.15.11.2 Estándares metodológicos

- **C++ Core-CP.110**: Do not write your own double-checked locking for initialization.

2.15.12 [cpp.sync.12] - Usar variables de condición para preservar la seguridad y el avance del programa

ID	cpp.sync.12	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Disponibilidad	S-SDLC	Build

Se recomienda el uso adecuado de variables de condición para coordinar hilos de forma segura con el fin de prevenir condiciones de carrera, bloqueos indefinidos, esperas activas y estados inconsistentes durante la sincronización entre procesos o hilos. El control correcto basado en `std::condition_variable` y primitivas asociadas permite mantener la coherencia del estado compartido, evitando que un hilo avance sin que se cumplan las condiciones necesarias, y reduciendo riesgos que podrían derivar en vulnerabilidades explotables o fallos de disponibilidad en sistemas concurrentes.

Codificación insegura

Uso de una variable de condición para coordinar un hilo productor y un consumidor. Se muestra cómo una señalización incorrecta y la ausencia de verificación de la condición compartida pueden provocar bloqueos indefinidos o comportamientos inesperados, afectando la seguridad y el avance del programa.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void worker() {
  //-- El hilo espera a que ready sea true
  std::unique_lock<std::mutex> lock(mtx);
```

```
  cv.wait(lock);
  //-- No se verifica la condición tras despertar
  std::cout << "Procesando tarea..." << std::endl;
}

int main() {
  std::thread t(worker);

  //-- Simula trabajo previo
  std::this_thread::sleep_for(std::chrono::milliseconds(500));

  //-- Notifica sin proteger correctamente el estado compartido
  cv.notify_one();
  ready = true;

  t.join();
  return 0;
}
```

Codificación recomendada

Uso correcto de una variable de condición donde el estado compartido se protege con un `mutex` y se verifica mediante un predicado. Se garantiza que el hilo solo avance cuando la condición lógica se cumple, evitando despertares espurios y problemas de sincronización.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void worker() {
  //-- El hilo espera hasta que ready sea true usando un predicado
  std::unique_lock<std::mutex> lock(mtx);
  cv.wait(lock, [] { return ready; });
  //-- La condición se garantiza antes de continuar
  std::cout << "Procesando tarea..." << std::endl;
}

int main() {
  std::thread t(worker);

  //-- Simula trabajo previo
  std::this_thread::sleep_for(std::chrono::milliseconds(500));
```

```
    {
        //-- Protege el acceso al estado compartido
        std::lock_guard<std::mutex> lock(mtx);
        ready = true;
    }

    //-- Notifica después de actualizar el estado protegido
    cv.notify_one();

    t.join();
    return 0;
}
```

2.15.12.1 Descripción

El uso de `std::condition_variable` constituye una de las herramientas fundamentales para realizar sincronización segura entre hilos. Se utiliza habitualmente para coordinar operaciones que dependen de un estado compartido, garantizando que un hilo no continúe hasta que otro haya producido o actualizado la información necesaria. Emplear estas primitivas de forma incorrecta puede generar vulnerabilidades que afecten la disponibilidad, como bloqueos intermitentes, interbloqueos cíclicos, pérdidas de señal o condiciones de carrera que acaben corrompiendo el estado de los datos. Para mitigar estos riesgos, es importante aplicar patrones robustos de diseño, mantener la coherencia de los bloqueos y utilizar verificaciones explícitas del estado dentro de estructuras de control repetitivas.

Los conceptos más importantes para el uso seguro de variables de condición son:

- **Mutex asociado:** Toda variable de condición debe trabajar con un `std::mutex` o `std::unique_lock`.
- **Notificación controlada:** Usar `notify_one` o `notify_all` según el modelo de espera requerido.
- **Evitar esperas activas:** Las esperas deben gestionarse mediante `wait` o `wait_for`, no mediante sondeos continuos.
- **Verificación de predicados:** La espera siempre debe realizarse dentro de un bucle que verifique una condición lógica compartida.
- **Manejo de despertares espurios:** Las variables de condición pueden activarse sin causa, por lo que es necesario reconfirmar la condición.

El elemento crítico de seguridad consiste en garantizar que ningún hilo procese datos en un estado no válido. Las variables de condición deben ser parte de un mecanismo integral en el que el acceso a los datos compartidos esté completamente protegido por bloqueos coherentes. Asimismo, la lógica de actualización debe ser atómica desde la perspectiva del sistema concurrente.

Riesgos asociados

- Condiciones de carrera (race conditions) que pueden conducir a un comportamiento impredecible.
- Interbloqueos (deadlocks), provocando que hilos o procesos se queden esperando indefinidamente.
- Violaciones de seguridad por acceso concurrente no controlado a recursos compartidos.
- Corrupción de datos al modificarse variables compartidas sin la debida sincronización.
- Lectura de datos inconsistentes o en un estado intermedio.
- Fallos de lógica o avance incorrecto del programa debido a la ausencia de notificaciones entre hilos.
- Bloqueos indefinidos si los hilos esperan condiciones que nunca se cumplen.
- Difícil depuración y rastreo de errores relacionados con la concurrencia.

Buenas prácticas

- Utilizar `std::condition_variable` junto con `std::unique_lock` para garantizar espera segura y liberación automática del `mutex`.

```
//-- Espera activa sin sincronización adecuada
bool ready = false;
std::mutex m;
void wait(){
  while(!ready){}
}
```

```
//-- Espera bloqueante con liberación automática del mutex
bool ready = false;
std::mutex m;
std::condition_variable cv;
void wait(){
  std::unique_lock<std::mutex> lock(m);
  cv.wait(lock, []{ return ready; });
}
```

- Proteger siempre la condición compartida con el mismo `mutex` utilizado por la variable de condición.

```
//-- Modificación de la condición sin protección
bool ready = false;
std::condition_variable cv;
```

```
void signal(){
  ready = true;
  cv.notify_one();
}
```

```
//-- Modificación protegida por mutex consistente
bool ready = false;
std::mutex m;
std::condition_variable cv;
void signal(){
  std::lock_guard<std::mutex> lock(m);
  ready = true;
  cv.notify_one();
}
```

- Utilizar la versión de `wait` con predicado para evitar despertares espurios y condiciones de carrera.

```
//-- Espera sin verificación posterior al despertar
std::mutex m;
std::condition_variable cv;
bool ready = false;
void wait(){
  std::unique_lock<std::mutex> lock(m);
  cv.wait(lock);
  if(!ready){
    return;
  }
}
```

```
//-- Espera con predicado que valida la condición compartida
std::mutex m;
std::condition_variable cv;
bool ready = false;
void wait(){
  std::unique_lock<std::mutex> lock(m);
  cv.wait(lock, []{ return ready; });
}
```

- Usar notificaciones adecuadas (`notify_one` o `notify_all`) según el patrón de concurrencia para evitar bloqueos o inanición.

```
//-- Notificación única en escenario con múltiples hilos esperando
std::mutex m;
std::condition_variable cv;
int counter = 0;
void signal(){
```

```
  std::lock_guard<std::mutex> lock(m);
  counter++;
  cv.notify_one();
}
```

```
//-- Notificación global cuando múltiples hilos dependen de la condición
std::mutex m;
std::condition_variable cv;
int counter = 0;
void signal(){
  std::lock_guard<std::mutex> lock(m);
  counter++;
  cv.notify_all();
}
```

2.15.12.2 Estándares metodológicos

- **C++ Core-CP.110**: Do not write your own double-checked locking for initialization.

2.16 Números aleatorios

2.16.1 [cpp.rand.1] - Utilizar generadores criptográficamente seguros

ID	cpp.rand.1	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Confidencialidad	S-SDLC	Build

La generación de material criptográfico debe realizarse exclusivamente mediante generadores de **números aleatorios criptográficamente seguros** (CSPRNG). El uso de generadores pseudoaleatorios de propósito general, la utilización de semillas predecibles o la inicialización manual de mecanismos criptográficos pueden comprometer la confidencialidad, la integridad y la autenticidad de claves, nonces, vectores de inicialización, sales, tokens y cualquier otro valor crítico para la seguridad. Siempre que sea posible, debe delegarse la obtención de entropía, la inicialización y el resembrado en el **sistema operativo** o en **bibliotecas criptográficas especializadas**.

Codificación insegura

Generación de un token de autenticación mediante `std::mt19937`. Aunque el generador se inicialice con `std::random_device`, no proporciona seguridad criptográfica y no debe utilizarse para generar tokens, claves, nonces ni otros valores sensibles.

```
#include <array>
#include <cstdint>
```

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

int main() {
    //-- std::mt19937 no es un generador criptográficamente seguro
    std::random_device randomDevice;
    std::mt19937 generator(randomDevice());

    std::uniform_int_distribution<unsigned int> distribution(0U, 255U);

    //-- Genera un supuesto token de autenticación de 256 bits
    std::array<std::uint8_t, 32> token{};

    for (std::uint8_t& byte : token) {
        byte = static_cast<std::uint8_t>(distribution(generator));
    }

    //-- Muestra el token en formato hexadecimal
    for (const std::uint8_t byte : token) {
        std::cout << std::hex << std::setw(2) << std::setfill('0') <<
        static_cast<unsigned int>(byte);
    }

    std::cout << '\n';
    return 0;
}
```

`std::mt19937` está diseñado para simulaciones y otros usos no criptográficos. Su estado interno puede reconstruirse a partir de suficientes valores observados, lo que permitiría predecir futuros tokens.

Codificación recomendada

Generación de un token de autenticación mediante `RAND_bytes()`, que utiliza el generador criptográficamente seguro gestionado por OpenSSL. La aplicación comprueba el resultado antes de utilizar el token.

```
#include <array>
#include <iomanip>
#include <iostream>
#include <stdexcept>
#include <openssl/rand.h>

int main() {
    //-- Genera un token de autenticación de 256 bits
    std::array<unsigned char, 32> token{};
```

```
    const int result = RAND_bytes(token.data(), static_cast<int>(token.size()));

    //-- No se utiliza el token si la generación falla
    if (result != 1) {
        throw std::runtime_error("No se pudo generar el token de autenticacion");
    }

    //-- Muestra el token en formato hexadecimal
    for (const unsigned char byte : token) {
        std::cout << std::hex << std::setw(2) << std::setfill('0') <<
        static_cast<unsigned int>(byte);
    }

    std::cout << '\n';
    return 0;
}
```

`RAND_bytes()` utiliza un generador criptográficamente seguro, gestiona internamente la obtención de entropía y el resembrado, y permite detectar errores antes de utilizar el material generado.

2.16.1.1 Descripción

Numerosos mecanismos criptográficos dependen de la generación de valores impredecibles. Claves, nonces, vectores de inicialización (IV), semillas, sales, tokens de autenticación o identificadores de sesión deben generarse mediante mecanismos capaces de proporcionar una entropía suficiente y de impedir que un atacante pueda predecir o reconstruir los valores generados. La utilización de generadores pseudoaleatorios de propósito general o de mecanismos de inicialización inadecuados puede comprometer completamente la seguridad del sistema.

Los generadores pseudoaleatorios de propósito general, como `std::rand()` o los motores de la biblioteca `<random>` (`std::mt19937`, `std::ranlux`, etc.), están diseñados para simulaciones, pruebas o aplicaciones estadísticas, pero no ofrecen garantías de seguridad criptográfica. Asimismo, el estándar de C++ no garantiza que `std::random_device` proporcione una fuente de entropía criptográficamente segura en todas las implementaciones, por lo que no debe utilizarse como mecanismo para generar directamente material criptográfico ni asumirse como sustituto de un CSPRNG.

Para la generación de material criptográfico deben emplearse generadores criptográficamente seguros (CSPRNG) proporcionados por el sistema operativo o por bibliotecas criptográficas consolidadas. En plataformas Windows se recomienda utilizar `BCryptGenRandom()`; en sistemas Linux y otros entornos POSIX, `getrandom()` o las APIs equivalentes ofrecidas por bibliotecas como OpenSSL. Estas soluciones obtienen la entropía de fuentes apropiadas, gestionan internamente la inicialización y el

resembrado del generador y proporcionan un nivel de seguridad adecuado para aplicaciones criptográficas.

La aplicación de este requisito implica seleccionar mecanismos adecuados para la generación de valores aleatorios, evitar el uso de generadores pseudoaleatorios no criptográficos, no inicializar manualmente mecanismos criptográficos mediante semillas constantes o fácilmente predecibles y comprobar siempre el resultado de las funciones encargadas de generar el material aleatorio.

Riesgos asociados

- Generación de claves, nonces, vectores de inicialización (IV), semillas, sales o tokens predecibles.
- Compromiso de la confidencialidad e integridad de mecanismos y protocolos criptográficos.
- Reducción efectiva de la entropía del material criptográfico generado, facilitando ataques de predicción o fuerza bruta.
- Repetición de valores críticos utilizados en operaciones criptográficas.
- Incumplimiento de las recomendaciones y estándares de seguridad aplicables.

Buenas prácticas

- Utilizar exclusivamente generadores criptográficamente seguros (CSPRNG) para generar claves, nonces, vectores de inicialización (IV), semillas, sales, tokens y cualquier otro material criptográfico.

```
//-- Uso de std::mt19937 para generar una clave
std::random_device rd;
std::mt19937 generator(rd());
std::array key{};
std::uniform_int_distribution distribution(0U, 255U);

for (auto& byte : key) {
  byte = static_cast(distribution(generator));
}
```

```
//-- Uso de un CSPRNG proporcionado por una biblioteca criptográfica
std::array key{};
if (RAND_bytes(key.data(), static_cast(key.size())) != 1) {
    throw std::runtime_error("RAND_bytes failed");
}
```

- No inicializar manualmente mecanismos criptográficos mediante semillas constantes o fácilmente predecibles; delegar la obtención de entropía y la

inicialización en el sistema operativo o en una biblioteca criptográfica.

```
//-- Semilla basada en información predecible
std::mt19937 generator(static_cast(std::time(nullptr)));
```

```
//-- La inicialización y el resembrado son gestionados
//-- automáticamente por la biblioteca criptográfica
std::array nonce{};
if (RAND_bytes(nonce.data(), static_cast(nonce.size())) != 1) {
    throw std::runtime_error("RAND_bytes failed");
}
```

- Comprobar siempre el resultado de las funciones encargadas de generar material criptográfico y gestionar adecuadamente los posibles errores.

```
//-- Se ignora el resultado de la función
std::array key{};
RAND_bytes(key.data(), static_cast(key.size()));
```

```
//-- Se verifica el resultado antes de utilizar los datos generados
std::array key{};
if (RAND_bytes(key.data(), static_cast(key.size())) != 1) {
    throw std::runtime_error("RAND_bytes failed");
}
```

2.16.1.2 Estándares metodológicos

- **CERT/CPP-MSC50**: Do not use std::rand() for generating pseudorandom numbers.

2.17 Bibliotecas

2.17.1 [cpp.lib.1] - Sustituir funciones de libc por utilidades seguras de la stdlib en C++

ID	**cpp.lib.1**	**Uso**	**Recomendado**
CWSS Score	**90**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Design**

Se exige adoptar de forma prioritaria las utilidades modernas de la biblioteca estándar para reemplazar funciones heredadas de `libc` que suelen provocar fallos de integridad, errores de memoria o comportamientos indefinidos. El uso de contenedores, algoritmos genéricos y mecanismos automáticos de gestión de recursos proporciona un modelo más robusto y reduce la exposición a vulnerabilidades graves asociadas al manejo

manual de punteros y buffers. Al emplear estas abstracciones seguras se fortalece de manera significativa la estabilidad interna del software y se minimiza la superficie de ataque relacionada con operaciones peligrosas sobre memoria.

Codificación insegura

Lectura de una cadena introducida por el usuario y almacenamiento en memoria para su posterior uso. Se busca mostrar el impacto de utilizar funciones heredadas de `libc` que no controlan adecuadamente los límites del buffer y cómo esto puede provocar desbordamientos de memoria y comportamiento indefinido.

```
#include <iostream>
#include <cstring>

int main() {
  //-- Declaración de un buffer fijo en memoria
  char buffer[10];

  //-- Solicita al usuario una cadena sin validar longitud
  std::cout << "Introduce una palabra: ";

  //-- Uso de función insegura de libc que no valida límites
  std::cin >> buffer;

  //-- Copia adicional usando strcpy sin comprobar tamaño
  char destino[10];
  strcpy(destino, buffer);

  //-- Muestra el resultado almacenado
  std::cout << "Contenido: " << destino << std::endl;

  return 0;
}
```

Codificación recomendada

Lectura de una cadena introducida por el usuario utilizando utilidades propias de la biblioteca estándar de C++ que gestionan dinámicamente la memoria y evitan desbordamientos. Se pretende mostrar cómo reemplazar funciones de `libc` por alternativas seguras y modernas de la STL.

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

int main() {
  //-- Uso de std::string para gestión segura de memoria
  std::string buffer;
```

```
    //-- Solicita al usuario una cadena completa
    std::cout << "Introduce una palabra: ";

    //-- Lectura segura usando std::getline
    std::getline(std::cin, buffer);

    //-- Copia segura mediante asignación de std::string
    std::string destino = buffer;

    //-- Muestra el resultado almacenado
    std::cout << "Contenido: " << destino << std::endl;

    return 0;
}
```

2.17.1.1 Descripción

El uso preferente de las utilidades seguras de la biblioteca estándar frente a las funciones clásicas de `libc` constituye un elemento esencial para reforzar la integridad del software. Muchas de las funciones heredadas, como `strcpy`, `gets`, `scanf` o las operaciones de reserva de memoria mediante `malloc` y `calloc`, fueron diseñadas para entornos donde las garantías de seguridad eran muy limitadas. En consecuencia, delegan en el programador la responsabilidad de validar tamaños, gestionar manualmente offsets y garantizar la ausencia de desbordamientos. Cuando estas tareas no se realizan con precisión absoluta, aparecen vulnerabilidades críticas que comprometen la estabilidad del sistema y exponen datos a manipulaciones no autorizadas.

La biblioteca estándar introduce un conjunto de abstracciones que permiten reducir de manera significativa estas fuentes de riesgo. Entre las más relevantes destacan los contenedores como `std::string`, `std::vector` y `std::array`. Estas estructuras eliminan la necesidad de emplear buffers gestionados manualmente y automatizan operaciones como la ampliación de capacidad, la copia de elementos o la destrucción de recursos. Esta capacidad permite evitar errores frecuentes asociados al cálculo incorrecto de índices o al uso de punteros sin validez. Además, al emplear estas utilidades se refuerza la coherencia interna del programa y se disminuye la dependencia de operaciones propensas a fallos de integridad.

Un aspecto especialmente importante es la integración del modelo **RAII**, que garantiza que los recursos se adquieren y liberan siguiendo reglas estrictas de ciclo de vida. El uso de constructores y destructores automáticos evita fugas de memoria y elimina situaciones peligrosas como dobles liberaciones o accesos a memoria ya liberada. Este mecanismo es difícil de reproducir de forma segura con las funciones de `libc`, donde la responsabilidad completa recae en el programador. Para reforzar estas garantías destacan los punteros inteligentes, como `std::unique_ptr` y `std::shared_ptr`, que

establecen reglas claras de propiedad y proporcionan una gestión de memoria segura sin necesidad de llamar explícitamente a `free`.

Las operaciones de entrada y salida también se benefician de las abstracciones seguras disponibles en la biblioteca estándar. Mecanismos como `std::getline` o `std::stringstream` encapsulan la lectura y la transformación de datos bajo reglas con validaciones internas. En contraste, funciones como `fgets` o `scanf` requieren una atención detallada y múltiples comprobaciones adicionales para evitar lecturas fuera de rango o conversiones inseguras. La sustitución sistemática de estas funciones reduce de manera directa la posibilidad de introducir vulnerabilidades de corrupción de memoria o fallos de integridad durante el procesamiento de datos externos.

Los algoritmos genéricos de la biblioteca estándar constituyen otra pieza clave del modelo seguro. Operaciones como `std::sort`, `std::find` o `std::remove_if` encapsulan lógica compleja diseñada, probada y optimizada para garantizar un comportamiento estable. Al utilizarlos, el equipo de desarrollo evita implementar manualmente bucles basados en punteros o índices, una práctica que incrementa el riesgo de errores fuera de límites o corrupciones de estado interno. Esta arquitectura basada en algoritmos y contenedores reduce de forma considerable la superficie de ataque asociada a la manipulación directa de memoria.

La orientación a tipos fuertes también contribuye de forma significativa a la seguridad. Al incorporar tipos bien definidos y estructuras de datos con restricciones estrictas, se logra que muchos errores comunes sean detectados en tiempo de compilación. Utilidades como `std::array`, cuyo tamaño es conocido durante la compilación, evitan situaciones típicas donde el programador debe garantizar manualmente que un buffer tiene la longitud adecuada. De manera similar, el modelo de iteradores elimina la necesidad de trabajar con direcciones de memoria crudas, lo que reduce la probabilidad de accesos indebidos o cálculos de punteros incorrectos.

La gestión segura de estados y errores es otro ámbito donde la biblioteca estándar ofrece ventajas notables frente a `libc`. Mientras que muchas funciones heredadas retornan valores ambiguos o dependen del manejo de `errno`, C++ moderno proporciona mecanismos explícitos como `std::optional` y `std::variant`, que permiten modelar condiciones excepcionales sin recurrir a punteros nulos o estructuras inconsistentes. Asimismo, el sistema de excepciones del lenguaje ofrece una forma más segura de capturar y gestionar fallos sin dejar estructuras internas en estados inestables.

El uso sistemático de estas utilidades modernas requiere la definición de políticas internas que desaconsejen o prohíban el empleo de funciones heredadas de `libc` salvo en casos estrictamente justificados. Esta disciplina evita que el código incorpore patrones históricos asociados a errores de integridad y facilita auditorías técnicas, ya que el uso predominante de abstracciones seguras reduce la complejidad de análisis y minimiza la posibilidad de errores inadvertidos durante revisiones de seguridad o pruebas de estrés.

Riesgos asociados

- Desbordamientos de buffer (buffer overflow) al utilizar funciones inseguros como `strcpy`, `strcat` o `sprintf`.
- Condiciones de carrera (race conditions) por falta de manejo adecuado de la memoria o datos compartidos.
- Fugas de información sensible por impresión o manipulación insegura de buffers.
- Inyección de código o ejecución de código arbitrario debido a manipulación insegura de cadenas.
- Corrupción de memoria (memory corruption) por uso incorrecto de punteros y funciones que no validan tamaños.
- Fugas de memoria por mala gestión en funciones de asignación/liberación manual de memoria.
- Violación de los principios de portabilidad y mantenibilidad del código.
- Dificultad en la identificación y mitigación de vulnerabilidades conocidas por exploits existentes en funciones de `libc`.
- Mayor probabilidad de errores en el manejo de strings complejos, especialmente con datos de entrada del usuario.
- Incompatibilidad con futuras versiones o entornos donde la implementación de funciones de `libc` sea distinta o insegura.

Buenas prácticas

- Reemplazar funciones de copia de cadenas como `strcpy` por `std::string` para evitar desbordamientos de buffer.

```
//-- Copia directa sin control de tamaño
char destino[8];
strcpy(destino, "TextoMuyLargo");
```

```
//-- Uso de std::string con gestión automática de memoria
std::string destino = "TextoMuyLargo";
```

- Sustituir `sprintf` por `std::ostringstream` o `std::format` para formateo seguro.

```
//-- Formateo sin verificación de límites
char buffer[16];
sprintf(buffer, "%d %s", 10, "ejemplo");
```

```
//-- Formateo seguro con flujo de salida
std::ostringstream oss;
```

```
oss << 10 << " " << "ejemplo";
std::string resultado = oss.str();
```

- Evitar `gets` y usar `std::getline` para lectura segura de entrada.

```
//-- Lectura sin control de longitud
char nombre[32];
gets(nombre);
```

```
//-- Lectura segura con std::string
std::string nombre;
std::getline(std::cin, nombre);
```

- Reemplazar `atoi` por `std::stoi` para manejo seguro de conversiones numéricas.

```
//-- Conversión sin manejo de errores
char entrada[] = "123abc";
int valor = atoi(entrada);
```

```
//-- Conversión con validación mediante excepciones
std::string entrada = "123abc";
int valor = std::stoi(entrada);
```

- Sustituir `malloc/free` por contenedores o punteros inteligentes de la STL.

```
//-- Gestión manual de memoria
int* datos = (int*)malloc(10 * sizeof(int));
free(datos);
```

```
//-- Gestión automática con contenedor estándar
std::vector<int> datos(10);
```

- Reemplazar `fopen/fclose` por `std::ifstream` o `std::ofstream` para manejo seguro de archivos.

```
//-- Apertura manual de archivo
FILE* archivo = fopen("datos.txt", "r");
fclose(archivo);
```

```
//-- Uso de flujo con cierre automático
std::ifstream archivo("datos.txt");
```

2.17.1.2 Estándares metodológicos

- **C++ Core-ES.1**: Prefer the standard library to other libraries and to "handcrafted code".
- **C++ Core-R.1**: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization).
- **C++ Core-R.11**: Avoid calling new and delete explicitly.
- **C++ Core-R.12**: Immediately give the result of an explicit resource allocation to a manager object.
- **C++ Core-SL.1**: Use libraries wherever possible.
- **C++ Core-SL.2**: Prefer the standard library to other libraries.
- **C++ Core-SL.con.1**: Prefer using STL array or vector instead of a C array.
- **C++ Core-SL.con.2**: Prefer using STL vector by default unless you have a reason to use a different container.
- **C++ Core-SL.con.3**: Avoid bounds errors.
- **MISRA CPP-4.1.1**: A program shall conform to ISO/IEC 14882:2017 (C++17).

2.17.1.3 Referencias

Standard Library

https://en.cppreference.com/w/cpp/standard_library.html

3. Anexo: Metodología de cálculo del score CWSS

Para la priorización de reglas de codificación segura se ha adoptado un perfil simplificado de CWSS 1.0. Este perfil se utiliza únicamente con fines de comparación interna entre reglas y no pretende modelar un hallazgo desplegado en un entorno concreto.

CWSS define tres grupos de métricas cuyo score final es el producto de los tres subscores:

- Base Finding
- Attack Surface
- Environmental

Las métricas del grupo Attack Surface se fijan a valores de peso 1.0. Con esta parametrización, el subscore de Attack Surface es 1.0 y no pondera sobre Base Finding.

- Required Privilege=None
- Required Privilege Layer=Application
- Access Vector=Internet
- Authentication Strength=None
- Level of Interaction=Automated
- Deployment Scope=N/A.

Las métricas del grupo Environmental se fijan a Not Applicable (N/A), ya que describen factores dependientes del entorno operativo, del negocio o de controles externos, información que no forma parte de una guía genérica de implementación. En CWSS, el valor N/A equivale a peso 1.0 cuando la métrica se ignora intencionalmente, por lo que este grupo no modifica la puntuación.

Dentro de Base Finding, la métrica variable es Technical Impact (TI), al ser la única que puede evaluarse de forma estable y portable para reglas de implementación genéricas.

Su valor varía entre 0 y 100, y el literal que se muestra depende de este valor según la siguiente tabla:

Muy alto	>= 80
Alto	>= 60
Medio	>= 40
Bajo	>= 20
Muy bajo	> 0
Ninguno	0

Valores del Impacto Técnico a partir del CWSS Score

El resto de las métricas se fijan a los siguientes valores como convención para su perfil:

- Acquired Privilege=Unknown, porque, en una regla genérica, no siempre puede inferirse de forma consistente qué privilegios concretos se obtendrían en explotación.
- Acquired Privilege Layer=Application, por ser la capa natural de evaluación de la guía.
- Internal Control Effectiveness=None, para no introducir compensaciones propias de un producto concreto.
- Finding Confidence=Confirmed, porque el documento puntúa incumplimientos normativos de reglas, no hallazgos con incertidumbre analítica.

Con estas fijaciones, el score final siempre coincidirá con el subscore del Base Finding.

CWSS Score	**86.00**								
CWSS Vector	CWSS:1.0.1/TI:H,0.90/AP:UK,0.50/AL:A,1.00/IC:N,1.00/FC:T,1.00/RP:N,1.00/RL:A,1.00/AV:I,1.00/AS:N,1.00/IN:A,1.00/SC:A,1.00/BI:NA,1.00/DI:NA,1.00/EX:NA,1.00/EC:NA,1.00/P:NA,1.00								
Hallazgo Base	**86.000**								
Impacto Técnico	Crítico	**Alto**	Medio	Bajo	Ninguno	-	Por Defecto	Desconocido	N/A
Privilegios Adquiridos	Administrador	Usuario Parcialmente Privilegiado	Usuario Regular	Limitado / Invitado	Ninguno	-	Por Defecto	**Desconocido**	N/A
Capa de Privilegios Adquirida	**Aplicación**	Sistema	Red	Infraestructura Empresarial	-	-	Por Defecto	Desconocido	N/A
Eficacia de Controles Internos	**Ninguno**	Limitado	Moderado	Indirecto (Defensa en Profundidad)	Mejor Disponible	Completo	Por Defecto	Desconocido	N/A
Confianza del Hallazgo	**Confirmado**	Confirmado Localmente	Falso Positivo	-	-	-	Por Defecto	Desconocido	N/A
Superficie de Ataque	**1.000**								
Privilegios Requeridos	**Ninguno**	Limitado / Invitado	Usuario Regular	Usuario Parcialmente Privilegiado	Administrador	-	Por Defecto	Desconocido	N/A
Capa de Privilegios Requerida	**Aplicación**	Sistema	Red	Infraestructura Empresarial	-	-	Por Defecto	Desconocido	N/A
Vector de Acceso	**Internet**	Intranet	Red Privada	Red Adyacente	Local	Físico	Por Defecto	Desconocido	N/A
Fortaleza de Autenticación	**Ninguna**	Débil	Moderada	Fuerte	-	-	Por Defecto	Desconocido	N/A
Nivel de Interacción	**Automatizado**	Típica / Limitada	Moderada	Oportunista	Alta	Sin Interacción	Por Defecto	Desconocido	N/A
Alcance de Despliegue	**Todos**	Moderado	Raro	Potencialmente Alcanzable	-	-	Por Defecto	Desconocido	N/A
Entorno	**1.000**								
Impacto en el Negocio	Crítico	Alto	Medio	Bajo	Ninguno	-	Por Defecto	Desconocido	**N/A**
Probabilidad de Descubrimiento	Alta	Media	Baja	-	-	-	Por Defecto	Desconocido	**N/A**
Probabilidad de Explotación	Alta	Media	Baja	Ninguna	-	-	Por Defecto	Desconocido	**N/A**
Eficacia de Controles Externos	Ninguno	Limitado	Moderado	Indirecto (Defensa en Profundidad)	Mejor Disponible	Completo	Por Defecto	Desconocido	**N/A**
Prevalencia	Generalizada	Alta	Común	Limitada	-	-	Por Defecto	Desconocido	**N/A**

CWE and CWSS are trademarks of The MITRE Corporation.

Detalle de las métricas del CWSS con sus valores prefijados, salvo Impacto Técnico.

PUBLICACIONES DE DEFENSA

GOBIERNO DE ESPAÑA
MINISTERIO DE DEFENSA
SUBSECRETARÍA DE DEFENSA
SECRETARÍA GENERAL TÉCNICA
SUBDIRECCIÓN GENERAL DE PUBLICACIONES Y PATRIMONIO CULTURAL