centro criptológico nacional

Guía de Desarrollo Seguro
CCN-STIC 481 – Anexo E

Codificación Segura para Ada



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

CCN
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 11:35:53 +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 Guidelines for Safe and Secure Ada/SPARK 2
1.3.2 SPARK Ada for MISRA C Developer 2
1.4 ESTRUCTURA DE LA GUÍA 2

2. REGLAS DE CODIFICACIÓN SEGURA 5

2.1 CONVERSIONES 5
2.1.1 [ada.conv.1] - Restringir conversiones implícitas y explícitas que puedan violar invariantes de tipo 5
2.1.2 [ada.conv.2] - Limitar el aliasing de parámetros para evitar corrupción de estado 10
2.1.3 [ada.conv.3] - Reducir el uso de Unchecked_Conversion 14
2.1.4 [ada.conv.4] - Evitar el uso de System.Address_To_Access_Conversions 18
2.1.5 [ada.conv.5] - Evitar el uso del atributo Unchecked_Access 23
2.2 INICIALIZACIÓN 27
2.2.1 [ada.ini.1] - Prohibir lecturas de datos no inicializados o parcialmente inicializados 27
2.3 ESTRUCTURAS DE CONTROL Y FLUJO 31
2.3.1 [ada.flow.1] - Prohibir el uso de cláusulas others en manejadores de excepciones 31
2.3.2 [ada.flow.2] - Prohibir la propagación de excepciones fuera del ámbito visible del subprograma 35
2.3.3 [ada.flow.3] - Garantizar que no existan excepciones no manejadas en el sistema 39
2.3.4 [ada.flow.4] - Probar formalmente la ausencia de excepciones en tiempo de ejecución ... 44
2.4 FUNCIONES 48
2.4.1 [ada.func.1] - Prohibir efectos laterales en funciones 48
2.4.2 [ada.func.2] - Definir y verificar precondiciones y postcondiciones en subprogramas críticos 53
2.4.3 [ada.func.3] - Prohibir parámetros con modo out no asignados explícitamente 58
2.4.4 [ada.func.4] - Obligar al uso del valor de retorno de las funciones 62
2.5 GESTIÓN DE MEMORIA 65
2.5.1 [ada.mem.1] - Prohibir la asignación dinámica salvo que esté estrictamente controlada y acotada estáticamente 65
2.5.2 [ada.mem.2] - Prohibir el uso de allocators en tipos access no restringidos 70
2.5.3 [ada.mem.3] - Utilizar pools de almacenamiento definidos por el usuario cuando se permita memoria dinámica 73
2.5.4 [ada.mem.4] - Determinar y verificar estáticamente el uso máximo de pila 79
2.5.5 [ada.mem.5] - Prohibir múltiples liberaciones de una misma región de memoria 83
2.5.6 [ada.mem.6] - Liberar únicamente memoria previamente asignada y válida 87
2.5.7 [ada.mem.7] - Liberar memoria exclusivamente en el mismo pool donde fue asignada ... 91
2.5.8 [ada.mem.8] - Evitar el uso de Unchecked_Deallocation 97
2.5.9 [ada.mem.9] - Evitar usar pragma Suppress 101
2.6 CONCURRENCIA 105
2.6.1 [ada.sync.1] - Restringir el modelo de concurrencia a perfiles deterministas (Ravenscar o Jorvik) 105
2.6.2 [ada.sync.2] - Prohibir el uso de variables compartidas no protegidas para comunicación entre tareas 110
2.6.3 [ada.sync.3] - Prohibir la recursión para evitar desbordamientos de pila y pérdida de control temporal 115

2.6.4 [ada.sync.4] - Implementar cancelación cooperante de tareas.....120

3. ANEXO: METODOLOGÍA DE CÁLCULO DEL SCORE CWSS 126

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 Guidelines for Safe and Secure Ada/SPARK

https://learn.adacore.com/pdf_books/courses/Guidelines_for_Safe_and_Secure_Ada_SPARK.pdf

Es una guía de referencia para equipos que desarrollan software en Ada o SPARK bajo requisitos de seguridad funcional o ciberseguridad. Recoge un conjunto de reglas y recomendaciones de codificación orientadas a evitar construcciones del lenguaje propensas a errores, aprovechar el sistema de tipos de Ada para detectar defectos en compilación y aplicar análisis estático con SPARK cuando se requiere verificación formal. Resulta especialmente útil en proyectos que deben cumplir estándares como DO-178C, EN 50128 o IEC 61508.

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

- **AdaCore RCL.02**: Only Reclaim Allocated Storage.
- **AdaCore RCL.03**: Only Reclaim to the Same Pool.
- **AdaCore DYN.05**: User-Defined Storage Pools Policy.

1.3.2 SPARK Ada for MISRA C Developer

https://learn.adacore.com/pdf_books/courses/SPARK_for_the_MISRA_C_Developer.pdf

Es una guía práctica para desarrolladores familiarizados con las reglas MISRA C que quieran adoptar SPARK Ada como alternativa verificable para software de seguridad crítica. Compara ambos enfoques regla a regla, explicando cómo las garantías que MISRA C persigue mediante directrices de codificación se obtienen en SPARK Ada directamente a través del sistema de tipos y el análisis estático formal. Cubre contratos, flujo de datos, ausencia de excepciones en tiempo de ejecución y verificación con GNATprove.

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

- **AdaCore/MISRA 4.1.2**: Pointers Are Not References.
- **AdaCore/MISRA 6.1**: Preventing Undefined Behavior.
- **AdaCore/MISRA 7.1**: Preventing Undefined Behavior in SPARK.

1.4 Estructura de la guía

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

- Conversiones.
- Inicialización.

- Estructuras de control y flujo.
- Funciones.
- Gestión de memoria.
- Concurrencia.

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 Conversiones

2.1.1 [ada.conv.1] - Restringir conversiones implícitas y explícitas que puedan violar invariantes de tipo

ID	**ada.conv.1**	**Uso**	**Obligatorio**
CWSS Score	**90**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Las conversiones entre tipos pueden comprometer las **invariantes de tipo** cuando no se validan adecuadamente los rangos y restricciones. Las conversiones implícitas realizadas automáticamente por el compilador y las conversiones explícitas mediante `unchecked conversion` pueden introducir valores fuera del rango válido del tipo destino, causando comportamientos indefinidos o **corrupción de datos**.

Codificación insegura

Conversiones que ignoran las restricciones de un subtipo con rango definido, permitiendo crear valores que violan sus invariantes. Se busca mostrar cómo una conversión no controlada puede introducir datos inválidos en el sistema y comprometer la seguridad y la lógica del programa.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Conversion;

procedure Main is
  //-- Subtipo con invariante: solo valores entre 0 y 100
  subtype Positive_Score is Integer range 0 .. 100;

  //-- Instanciación de conversión no verificada
  function To_Positive_Score is new Ada.Unchecked_Conversion (Integer,
Positive_Score);

  //-- Valor fuera del rango permitido
  Raw_Value : Integer := 200;

  //-- Conversión que ignora el rango del subtipo
  Converted_Value : Positive_Score := To_Positive_Score (Raw_Value);

begin
  //-- Se imprime un valor que viola el invariante del subtipo
  Put_Line ("Valor convertido: " & Positive_Score'Image (Converted_Value));
end Main;
```

Codificación recomendada

Validación explícita antes de realizar la conversión a un subtipo con rango restringido. Se busca garantizar que los invariantes del tipo se mantengan, evitando conversiones inseguras y controlando los errores de forma clara.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  //-- Subtipo con invariante: solo valores entre 0 y 100
  subtype Positive_Score is Integer range 0 .. 100;

  //-- Función que valida antes de convertir
  function To_Positive_Score (Value : Integer) return Positive_Score is
  begin
    //-- Verificación explícita del rango permitido
    if Value in Positive_Score then
      return Positive_Score (Value);
    else
      raise Constraint_Error with "Valor fuera de rango permitido";
    end if;
  end To_Positive_Score;

  //-- Valor de entrada
  Raw_Value : Integer := 85;

  //-- Conversión segura mediante validación
  Converted_Value : Positive_Score := To_Positive_Score (Raw_Value);

begin
  //-- Se imprime un valor que respeta el invariante del subtipo
  Put_Line ("Valor convertido: " & Positive_Score'Image (Converted_Value));
end Main;
```

2.1.1.1 Descripción

Las conversiones de tipo representan uno de los puntos más vulnerables en la integridad de datos, especialmente cuando se realizan entre tipos con diferentes rangos, precisiones o representaciones internas. Estas operaciones pueden comprometer las invariantes que definen las propiedades fundamentales de cada tipo de dato.

Las **conversiones implícitas** ocurren automáticamente durante asignaciones, paso de parámetros o evaluación de expresiones cuando el compilador considera que existe compatibilidad entre tipos. Sin embargo, esta compatibilidad no garantiza que los valores resultantes mantengan su validez semántica o que se preserven dentro de los límites esperados del tipo destino.

Las **conversiones explícitas** mediante operadores de conversión o funciones de transformación otorgan mayor control al programador, pero también introducen responsabilidades adicionales para verificar que la operación sea segura y mantenga la coherencia de los datos.

Los riesgos asociados incluyen **desbordamientos numéricos**, pérdida de precisión en operaciones con punto flotante, truncamiento de valores, y violación de restricciones de rango definidas en tipos personalizados. Estos problemas pueden manifestarse como comportamientos impredecibles, corrupción silenciosa de datos o fallos en tiempo de ejecución.

La mitigación efectiva requiere establecer políticas restrictivas sobre qué conversiones están permitidas, implementar validaciones explícitas antes de realizar transformaciones de tipo, y utilizar mecanismos de verificación de rangos. Es fundamental evaluar cada conversión en el contexto específico de la aplicación y documentar claramente las asunciones sobre los rangos de valores esperados.

El diseño de tipos debe considerar las conversiones necesarias desde las fases iniciales, definiendo interfaces claras y funciones de conversión controladas que incorporen las validaciones apropiadas. Esta aproximación preventiva reduce significativamente la superficie de ataque y mejora la robustez del sistema.

Riesgos asociados

- Corrupción de datos al incumplir restricciones o invariantes del tipo destino.
- Comportamiento indefinido de la aplicación al operar con datos de tipos incompatibles.
- Exploit de escalada de privilegios o bypass de controles de acceso al cambiar intencionadamente el tipo de un objeto.
- Pérdida de robustez y fallos en tiempo de ejecución difíciles de depurar.
- Vulnerabilidad a ataques de desbordamiento o manipulación de memoria por valores fuera del rango esperado.
- Debilitamiento de los mecanismos de comprobación de tipo provistos por el compilador y el lenguaje.
- Reducción de la mantenibilidad y legibilidad del código, dificultando auditorías de seguridad.
- Riesgo de denegación de servicio si las conversiones erróneas afectan procesos críticos.
- Posibilidad de alteración de lógica de negocio y toma de decisiones incorrectas en la aplicación.

Buenas prácticas

- Evitar conversiones entre tipos numéricos distintos cuando representen conceptos diferentes, aunque tengan la misma representación interna.

```
//-- Conversión directa entre tipos conceptualmente distintos
type Edad is range 0 .. 120;
type Temperatura is range -50 .. 150;
E : Edad := 25;
T : Temperatura := Temperatura(E);
```

```
//-- Uso de subtipos o funciones de validación para preservar el invariante
subtype Edad_Segura is Integer range 0 .. 120;
subtype Temperatura_Segura is Integer range -50 .. 150;
E2 : Edad_Segura := 25;
function Convertir_Edad_A_Temperatura (V : Edad_Segura) return Temperatura_Segura
is
begin
  return Temperatura_Segura(V);
end Convertir_Edad_A_Temperatura;
```

- Evitar conversiones desde tipos derivados a su tipo base que eliminen restricciones definidas en subtipos.

```
//-- Eliminación de restricciones al convertir a tipo base
subtype Porcentaje is Integer range 0 .. 100;
P : Porcentaje := 50;
Valor : Integer := Integer(P);
Valor := 200;
```

```
//-- Mantener el uso del subtipo restringido para conservar validaciones
subtype Porcentaje_Seguro is Integer range 0 .. 100;
P2 : Porcentaje_Seguro := 50;
procedure Asignar (V : in out Porcentaje_Seguro; Nuevo : Integer) is
begin
  if Nuevo in Porcentaje_Seguro then
    V := Nuevo;
  end if;
end Asignar;
```

- No utilizar conversiones no verificadas (`Unchecked_Conversion`) sin validaciones adicionales.

```
//-- Conversión no verificada que ignora el modelo de tipos
type Bytes is array (1 .. 4) of Interfaces.Unsigned_8;
type Entero is new Integer;
```

```
function Convertir is new Ada.Unchecked_Conversion (Bytes, Entero);
```

```
//-- Validación explícita antes de reinterpretar los datos
function Convertir_Seguro (B : Bytes) return Entero is
  Resultado : Entero := 0;
begin
  for I in B'Range loop
    Resultado := Resultado * 256 + Entero(B(I));
  end loop;
  return Resultado;
end Convertir_Seguro;
```

- Evitar conversiones explícitas para forzar compatibilidad entre tipos privados, respetando la encapsulación.

```
//-- Conversión forzada que rompe encapsulación
package Cuentas is
  type Saldo is private;
private
  type Saldo is new Integer range 0 .. 1_000_000;
end Cuentas;
with Cuentas; use Cuentas;
S : Saldo := Saldo(500);
```

```
//-- Uso de función constructora que valida el invariante
package body Cuentas is
  function Crear (Valor : Integer) return Saldo is
  begin
    if Valor in 0 .. 1_000_000 then
      return Saldo(Valor);
    else
      raise Constraint_Error;
    end if;
  end Crear;
end Cuentas;
```

2.1.1.2 Estándares metodológicos

- **AdaCore/MISRA 4.2.2**: Restricting Explicit Conversions.
- **AdaCore/MISRA 4.2.3**: Restricting Implicit Conversions.

2.1.2 [ada.conv.2] - Limitar el aliasing de parámetros para evitar corrupción de estado

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

El aliasing de parámetros ocurre cuando dos o más parámetros de un subprograma referencian la misma ubicación de memoria, creando múltiples rutas de acceso al mismo dato. Esta situación puede provocar **comportamientos impredecibles** y **corrupción del estado** del programa, especialmente cuando se combinan parámetros de entrada y salida que apuntan al mismo objeto.

Codificación insegura

Manipulación de parámetros `in out` que pueden referenciar el mismo objeto, provocando efectos laterales no previstos y corrupción del estado lógico. Se busca mostrar cómo el aliasing permite que una misma variable sea modificada varias veces dentro de una operación que asume objetos distintos.

```
with Ada.Text_IO; use Ada.Text_IO;
with System;

//-- Demostración de aliasing no controlado en parámetros in out
procedure Main is
  procedure Actualizar (A, B : in out Integer) is
  begin
    //-- Se incrementa A
    A := A + 1;

    //-- Se calcula B en función del nuevo valor de A
    B := A + 1;
  end Actualizar;

  X : Integer := 5;
begin
  //-- Se pasan la misma variable en ambos parámetros
  Actualizar (X, X);

  //-- El resultado puede no ser el esperado debido al aliasing
  Put_Line ("Valor final de X: " & Integer'Image (X));
end Main;
```

Codificación recomendada

Validación explícita para evitar aliasing entre parámetros `in out`, comprobando si ambos hacen referencia al mismo objeto en memoria. Se pretende aprender a bloquear estados inconsistentes detectando la condición antes de modificar datos sensibles.

```
with Ada.Text_IO; use Ada.Text_IO;
with System;

//-- Prevención de aliasing verificando direcciones de memoria
procedure Main is
  procedure Actualizar_Seguro (A, B : in out Integer) is
  begin
    //-- Se valida que no apunten al mismo objeto
    if A'Address = B'Address then
      //-- Se bloquea la operación para evitar corrupción de estado
      raise Program_Error;
    end if;

    //-- Operaciones seguras al garantizar objetos distintos
    A := A + 1;
    B := A + 1;
  end Actualizar_Seguro;

  X : Integer := 5;
  Y : Integer := 5;
begin
  //-- Uso correcto con variables distintas
  Actualizar_Seguro (X, Y);

  Put_Line ("Valor final de X: " & Integer'Image (X));
  Put_Line ("Valor final de Y: " & Integer'Image (Y));
end Main;
```

2.1.2.1 Descripción

El aliasing de parámetros ocurre cuando dos o más parámetros de un subprograma hacen referencia a la misma ubicación de memoria durante la ejecución. Esta situación puede provocar comportamientos impredecibles y corrupción del estado del programa, especialmente cuando se combinan parámetros de entrada, salida y entrada-salida.

El problema surge principalmente con parámetros de modo `out` e `in out` que pueden modificar el contenido de variables. Cuando estos parámetros apuntan a la misma ubicación de memoria que otros parámetros del mismo subprograma, las modificaciones realizadas a través de un parámetro afectan inmediatamente al otro, creando dependencias no intencionadas entre las operaciones.

Los tipos de acceso, arrays y registros son particularmente susceptibles al aliasing, ya que pueden compartir componentes o referenciar las mismas estructuras de datos. Las variables globales también pueden crear situaciones de aliasing cuando se pasan como parámetros a subprogramas que también acceden a ellas directamente.

Los riesgos asociados incluyen la **corrupción silenciosa de datos**, donde las modificaciones no intencionadas pueden pasar desapercibidas durante las pruebas, pero manifestarse en condiciones específicas de ejecución. La **inconsistencia del estado** puede llevar a violaciones de invariantes y comportamientos erráticos del programa.

La mitigación efectiva requiere un diseño cuidadoso de las interfaces de subprogramas y el uso de técnicas de verificación estática. El compilador puede detectar algunos casos de aliasing, pero la responsabilidad principal recae en el diseño del código para evitar estas situaciones. La documentación clara de las precondiciones y el uso de tipos específicos pueden ayudar a prevenir el aliasing accidental.

Esta regla es especialmente relevante en sistemas críticos donde la integridad de los datos debe mantenerse en todo momento y donde los errores de aliasing pueden tener consecuencias graves para la seguridad y confiabilidad del sistema.

Riesgos asociados

- Corrupción inadvertida de datos compartidos debido a modificaciones simultáneas.
- Fallos inesperados en la lógica de la aplicación por efectos secundarios no controlados.
- Dificultad en el rastreo y corrección de errores debido a la falta de claridad sobre qué datos pueden verse afectados por cada operación.
- Vulnerabilidad a condiciones de carrera en sistemas concurrentes.
- Rotura de encapsulamiento y pérdida de invariantes de tipo o lógica.
- Posibilidad de explotación por parte de actores maliciosos mediante manipulación de referencias compartidas.
- Incompatibilidad con análisis formal y verificación automática del código.

Buenas prácticas

- Evitar pasar el mismo objeto como parámetro `in out` más de una vez en una llamada.

```
//-- Se pasa el mismo objeto a dos parámetros in out
  procedure Actualizar (A : in out Integer; B : in out Integer) is
  begin
    A := A + 1;
    B := B * 2;
```

```
  end Actualizar;
  X : Integer := 5;
  Actualizar (X, X);
```

```
//-- Se evita aliasing usando variable intermedia
  X : Integer := 5;
  Temp : Integer := X;
  Actualizar (X, Temp);
```

- Restringir el uso de parámetros `in out` cuando solo se requiere lectura.

```
//-- Parámetro declarado in out sin necesidad de modificación
  procedure Procesar (Dato : in out Integer) is
  begin
    Put_Line (Integer'Image (Dato));
  end Procesar;
```

```
//-- Parámetro declarado como in para evitar aliasing innecesario
  procedure Procesar (Dato : in Integer) is
  begin
    Put_Line (Integer'Image (Dato));
  end Procesar;
```

- Evitar aliasing mediante el uso de tipos protegidos para encapsular el estado compartido.

```
//-- Acceso directo a variable global compartida
  Contador : Integer := 0;
  procedure Incrementar (Valor : in out Integer) is
  begin
    Valor := Valor + 1;
  end Incrementar;
  Incrementar (Contador);
```

```
//-- Encapsulamiento del estado en tipo protegido
  protected type Contador_Seguro is
    procedure Incrementar;
  private
    Valor : Integer := 0;
  end Contador_Seguro;
  protected body Contador_Seguro is
    procedure Incrementar is
    begin
      Valor := Valor + 1;
    end Incrementar;
```

```
  end Contador_Seguro;
```

- Utilizar contratos (`Pre`/`Post`) para detectar posibles aliasing indebidos.

```
//-- Procedimiento sin restricción de aliasing
  procedure Copiar (Origen : in out Integer; Destino : in out Integer) is
  begin
    Destino := Origen;
    Origen := 0;
  end Copiar;
```

```
//-- Contrato que exige objetos distintos
  procedure Copiar (Origen : in out Integer; Destino : in out Integer)
    with Pre => Origen'Address /= Destino'Address is
  begin
    Destino := Origen;
    Origen := 0;
  end Copiar;
```

2.1.2.2 Estándares metodológicos

- **AdaCore/MISRA 6.1**: Preventing Undefined Behavior.
- **AdaCore RPP.08**: Limit Parameter Aliasing.

2.1.3 [ada.conv.3] - Reducir el uso de Unchecked_Conversion

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

La función `Unchecked_Conversion` permite convertir datos entre tipos incompatibles sin verificaciones de seguridad, lo que puede generar **comportamiento indefinido** y vulnerabilidades de seguridad. Su uso inadecuado puede provocar corrupción de memoria, acceso a datos no válidos y fallos del sistema difíciles de detectar durante las pruebas.

Codificación insegura

Conversión entre tipos incompatibles utilizando mecanismos de bajo nivel para forzar la reinterpretación de memoria. La intención es evidenciar los riesgos de seguridad y mantenibilidad cuando se evita el sistema de tipos del lenguaje, lo que puede producir resultados impredecibles y dependientes de la plataforma.

```
with Ada.Text_IO;
```

```
use Ada.Text_IO;
with Ada.Unchecked_Conversion;

procedure Main is
  //-- Declaración de una función de conversión no verificada
  function Int_To_Float is new Ada.Unchecked_Conversion (Integer, Float);

  I : Integer := 16#40490FDB#; -- Valor hexadecimal arbitrario

  F : Float;
begin
  //-- Conversión forzada reinterpretando los bits en memoria
  F := Int_To_Float (I);

  //-- Se imprime el resultado, que depende de la representación interna
  Put_Line ("Valor convertido: " & Float'Image (F));
end Main;
```

Codificación recomendada

Conversión entre tipos numéricos utilizando los mecanismos seguros proporcionados por el lenguaje. La intención es preservar la verificación de tipos en tiempo de compilación y ejecución, evitando reinterpretaciones de memoria que puedan introducir fallos sutiles o vulnerabilidades.

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure Main is
  //-- Declaración de variables con tipos definidos
  I : Integer := 42;
  F : Float;
begin
  //-- Conversión segura usando conversión explícita de tipo
  F := Float (I);

  //-- Se imprime el resultado de forma predecible y portable
  Put_Line ("Valor convertido: " & Float'Image (F));
end Main;
```

2.1.3.1 Descripción

La función `Unchecked_Conversion` permite realizar conversiones de tipo sin las verificaciones de seguridad habituales del sistema de tipos. Esta función omite las comprobaciones en tiempo de compilación y ejecución que normalmente garantizan la compatibilidad entre tipos de datos.

El uso de `Unchecked_Conversion` introduce **vulnerabilidades de seguridad** relacionadas con la corrupción de memoria, accesos no autorizados a datos y comportamientos indefinidos. La ausencia de verificaciones puede provocar interpretaciones incorrectas de los datos en memoria, especialmente cuando los tipos origen y destino tienen diferentes tamaños o representaciones internas.

Esta función resulta particularmente problemática cuando se aplica entre tipos con **diferentes alineaciones de memoria** o cuando involucra punteros y tipos de acceso. Las conversiones no verificadas pueden generar referencias a ubicaciones de memoria inválidas o permitir el acceso a datos fuera de los límites establecidos.

Los riesgos se incrementan significativamente en sistemas concurrentes, donde las conversiones no controladas pueden introducir **condiciones de carrera** y estados inconsistentes entre hilos de ejecución. La manipulación directa de representaciones de memoria sin supervisión del compilador compromete las garantías de seguridad del lenguaje.

La mitigación de estos riesgos requiere **limitar el uso** de `Unchecked_Conversion` a casos estrictamente necesarios y bien documentados. Se debe priorizar el uso de conversiones explícitas seguras y funciones de conversión que mantengan las verificaciones de tipos. Cuando su uso sea inevitable, es necesario implementar validaciones adicionales y controles de integridad de datos.

La **encapsulación** de las conversiones no verificadas en módulos específicos y la implementación de interfaces seguras ayudan a contener los riesgos asociados. El análisis estático del código y las revisiones exhaustivas constituyen medidas complementarias para identificar usos potencialmente peligrosos de esta función.

Riesgos asociados

- Violación de la seguridad de tipos, permitiendo operaciones ilegales y resultados no definidos.
- Acceso y escritura a memoria no intencionada, posibilitando vulnerabilidades explotables como buffer overflows.
- Introducción de errores difíciles de detectar y depurar al convertir datos entre tipos incompatibles.
- Comportamiento indefinido de la aplicación, produciendo fallos intermitentes o corrupción de datos.
- Pérdida de portabilidad, debido a dependencias en representaciones de datos específicas de la plataforma.
- Posible vulnerabilidad ante ataques que explotan conversiones de tipo inseguras para ejecutar código malicioso.
- Compromiso de la robustez y confiabilidad del software, especialmente en

sistemas críticos o embebidos.

- Dificultad en la verificación formal del sistema, disminuyendo la eficacia de técnicas automáticas de análisis.

Buenas prácticas

- Evitar el uso de `Unchecked_Conversion` cuando exista una conversión de tipos segura o un tipo derivado compatible.

```
//-- Uso de conversión no verificada entre tipos compatibles
function To_Int is new Ada.Unchecked_Conversion (Source => My_Int, Target =>
Integer);
Value : Integer := To_Int (X);
```

```
//-- Uso de conversión explícita basada en compatibilidad de tipos
subtype My_Int is Integer range 0 .. 100;
Value : Integer := Integer (X);
```

- Encapsular la conversión en un paquete privado para limitar su alcance y exposición.

```
//-- Conversión no verificada expuesta globalmente
function To_Bytes is new Ada.Unchecked_Conversion (Source => Integer, Target =>
Byte_Array);
Global_Data : Byte_Array := To_Bytes (Data);
```

```
//-- Conversión encapsulada en la parte privada del paquete
package body Conversions is
  function To_Bytes is new Ada.Unchecked_Conversion (Source => Integer, Target =>
Byte_Array);
end Conversions;
Result : Byte_Array := Conversions.To_Bytes (Data);
```

- Validar tamaño y representación de los tipos antes de aplicar `Unchecked_Conversion`.

```
//-- Conversión sin verificación de tamaño entre tipos
function To_Record is new Ada.Unchecked_Conversion (Source => Integer, Target =>
My_Record);
R : My_Record := To_Record (Raw_Data);
```

```
//-- Verificación previa de tamaño antes de convertir
pragma Assert (Integer'Size = My_Record'Size);
function To_Record is new Ada.Unchecked_Conversion (Source => Integer, Target =>
My_Record);
```

```
R : My_Record := To_Record (Raw_Data);
```

- Preferir representaciones explícitas o cláusulas de representación en lugar de reinterpretar memoria.

```
//-- Reinterpretación directa de memoria usando Unchecked_Conversion
function To_Float is new Ada.Unchecked_Conversion (Source => Integer, Target =>
Float);
F : Float := To_Float (Bits);
```

```
//-- Uso de cláusula de representación para controlar el mapeo de bits
type Int_Float is record
  case Boolean is
    when True  => I : Integer;
    when False => F : Float;
  end case;
end record;
pragma Unchecked_Union (Int_Float);
```

2.1.3.2 Estándares metodológicos

- **AdaCore/MISRA 4.2.2**: Restricting Explicit Conversions.
- **AdaCore/MISRA 4.2.3**: Restricting Implicit Conversions.
- **AdaCore RPP.13**: No Reuse of Standard Typemarks.
- **AdaCore SWE.01**: Use SPARK Extensively.

2.1.4 [ada.conv.4] - Evitar el uso de System.Address_To_Access_Conversions

ID	ada.conv.4	Uso	Obligatorio
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La conversión directa entre direcciones de sistema y punteros de acceso mediante `System.Address_To_Access_Conversions` introduce **vulnerabilidades de seguridad** al permitir el acceso no controlado a ubicaciones de memoria arbitrarias. Este mecanismo elude las verificaciones de tipos y los controles de acceso, creando oportunidades para **corrupción de memoria** y **accesos no autorizados** a datos sensibles.

Codificación insegura

Conversiones inseguras desde una dirección de memoria a un tipo `access` para manipular datos. Se busca mostrar cómo forzar una conversión desde `System.Address`

puede romper el tipado fuerte de Ada, generar aliasing no controlado y provocar errores difíciles de detectar en tiempo de ejecución.

```
with Ada.Text_IO; use Ada.Text_IO;
with System;
with System.Address_To_Access_Conversions;

procedure Main is
  //-- Declaración de una variable entera normal
  Value : Integer := 10;

  //-- Instanciación del paquete de conversión insegura
  package Conv is new System.Address_To_Access_Conversions (Integer);

  //-- Puntero obtenido a partir de la dirección de memoria
  Ptr : Conv.Object_Pointer;
begin
  //-- Conversión directa desde Address a access
  Ptr := Conv.To_Pointer (Value'Address);

  //-- Modificación indirecta del valor usando el puntero
  Ptr.all := 20;

  //-- Impresión del valor modificado
  Put_Line ("Valor: " & Integer'Image (Value));
end Main;
```

Codificación recomendada

Tipos `access` definidos de forma explícita y segura, aprovechando el tipado fuerte y las verificaciones del compilador. Se pretende mostrar cómo trabajar con referencias sin recurrir a conversiones desde direcciones de memoria, manteniendo la seguridad de tipos y evitando aliasing no controlado.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  //-- Definición de un tipo access seguro
  type Integer_Access is access all Integer;

  //-- Variable aliased para permitir 'Access
  Value : aliased Integer := 10;

  //-- Puntero tipado correctamente
  Ptr : Integer_Access;
begin
  //-- Obtención segura de la referencia
  Ptr := Value'Access;
```

```
   //-- Modificación del valor a través del access tipado
   Ptr.all := 20;

   //-- Impresión del valor modificado
   Put_Line ("Valor: " & Integer'Image (Value));
end Main;
```

2.1.4.1 Descripción

La conversión directa entre direcciones de sistema y punteros de acceso mediante `System.Address_To_Access_Conversions` introduce vulnerabilidades significativas en la gestión de memoria y puede comprometer la integridad del programa. Este mecanismo permite eludir las verificaciones de tipos y los controles de acceso a memoria que proporciona el sistema de tipos, creando puntos de fallo potenciales.

Las conversiones no controladas entre direcciones de memoria y referencias tipadas pueden generar **accesos a memoria no válida**, corrupción de datos y comportamientos impredecibles durante la ejecución. El uso de estas conversiones elimina las garantías de seguridad que ofrece el modelo de memoria, permitiendo que el programa acceda a regiones de memoria que pueden contener datos inconsistentes o estar fuera de los límites asignados.

La manipulación directa de direcciones de memoria sin las protecciones adecuadas puede resultar en **violaciones de segmento**, escrituras en ubicaciones críticas del sistema, o la exposición de información sensible almacenada en memoria. Estos problemas se agravan en entornos donde múltiples tareas acceden concurrentemente a los mismos recursos de memoria.

Para mitigar estos riesgos, se recomienda utilizar los mecanismos de acceso a memoria proporcionados por el lenguaje que mantienen la verificación de tipos y los controles de límites. Las alternativas incluyen el uso de **tipos de acceso controlados**, la implementación de interfaces específicas para la comunicación con hardware de bajo nivel, y la aplicación de técnicas de programación que preserven la integridad del modelo de memoria sin recurrir a conversiones directas no verificadas.

Riesgos asociados

- Desreferenciación de direcciones inválidas, lo que puede causar fallos de segmentación o corrupción de memoria.
- Introducción de vulnerabilidades de seguridad como ejecución de código arbitrario.
- Dificultad para realizar análisis estático y comprobación de seguridad en tiempo de compilación.

- Pérdida de portabilidad del código debido a dependencias de bajo nivel en la representación de direcciones en memoria.
- Posibilidad de exponer información confidencial mediante acceso indebido a datos en la memoria.
- Aumento del riesgo de fugas de memoria y referencias colgantes (dangling pointers).
- Complicación en la depuración y mantenimiento del sistema por comportamientos no definidos.
- Elusión de los mecanismos de seguridad provistos por el tipado estricto de Ada.

Buenas prácticas

- Utilizar tipos de acceso fuertemente tipados en lugar de conversiones desde `System.Address`.

```
//-- Conversión directa desde System.Address usando
SystemAddress_To_Access_Conversions
type Int_Ptr is access all Integer;
package Conv is new System.Address_To_Access_Conversions (Integer);
Ptr : Int_Ptr := Int_Ptr (Conv.To_Pointer (Some_Address));
```

```
//-- Uso de tipos de acceso definidos explícitamente y asignación segura
type Int_Ptr is access all Integer;
Value : aliased Integer := 10;
Ptr : Int_Ptr := Value'Access;
```

- Encapsular el acceso a memoria en abstracciones seguras en lugar de exponer direcciones del sistema.

```
//-- Exposición de dirección y conversión para manipulación externa
package Conv is new System.Address_To_Access_Conversions (Integer);
function Get return Integer is
  Ptr : constant access Integer := Conv.To_Pointer (External_Address);
begin
  return Ptr.all;
end Get;
```

```
//-- Encapsulamiento mediante interfaz que evita exposición de direcciones
package Safe_Data is
  function Get return Integer;
end Safe_Data;
package body Safe_Data is
  Data : aliased Integer := 0;
  function Get return Integer is
```

```
  begin
    return Data;
  end Get;
end Safe_Data;
```

- Emplear cláusulas `Representation` y aspectos `Address` solo cuando sea estrictamente necesario y controlado.

```
//-- Conversión genérica de direcciones sin restricciones de representación
package Conv is new System.Address_To_Access_Conversions (Float);
F_Ptr : access Float := Conv.To_Pointer (Raw_Address);
Value : Float := F_Ptr.all;
```

```
//-- Asociación controlada de dirección con objeto específico y documentado
Device_Register : aliased Float
  with Address => System'To_Address (16#4000_0000#);
Value : Float := Device_Register;
```

- Preferir bibliotecas estándar y mecanismos de interoperabilidad definidos antes que conversiones de direcciones manuales.

```
//-- Uso de System.Address para interoperar con código externo
package Conv is new System.Address_To_Access_Conversions (Integer);
External : System.Address := Get_External_Address;
Ptr : access Integer := Conv.To_Pointer (External);
Result : Integer := Ptr.all;
```

```
//-- Uso de Interfaces.C y tipos compatibles para interoperabilidad segura
with Interfaces.C;
use Interfaces;
C_Value : aliased C.int;
pragma Import (C, C_Value, "external_symbol");
Result : Integer := Integer (C_Value);
```

2.1.4.2 Estándares metodológicos

- **AdaCore/MISRA 4.1.1**: Pointers Are Not Addresses.
- **AdaCore/MISRA 4.1.4**: Pointers Should Be Typed.
- **AdaCore DYN.03**: Access Types Without Allocators Policy.
- **AdaCore RPP.08**: Limit Parameter Aliasing.

2.1.5 [ada.conv.5] - Evitar el uso del atributo Unchecked_Access

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

El atributo `Unchecked_Access` permite obtener punteros a objetos sin las verificaciones habituales de **tiempo de vida** y **accesibilidad**. Su uso introduce riesgos de **referencias colgantes** y accesos a memoria no válida, ya que elude las protecciones del sistema de tipos que normalmente previenen estos errores en tiempo de compilación.

Codificación insegura

Punteros obtenidos con el atributo `Unchecked_Access` para devolver la dirección de una variable local. Se busca mostrar cómo se puede generar una referencia colgante cuando el objeto referenciado deja de existir, provocando comportamiento indefinido y riesgos de seguridad.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Unchecked_Access_Bad is
  //-- Definición de un tipo de puntero a Integer
  type Int_Ptr is access all Integer;

  //-- Función que devuelve un puntero a una variable local
  function Get_Ptr return Int_Ptr is
    //-- Variable local aliased para permitir Unchecked_Access
    Local : aliased Integer := 10;
  begin
    //-- Se devuelve la dirección de una variable local (riesgo)
    return Local'Unchecked_Access;
  end Get_Ptr;

  Ptr : Int_Ptr;
begin
  //-- Se obtiene un puntero a memoria que ya no será válida
  Ptr := Get_Ptr;

  //-- Uso de un puntero colgante (comportamiento indefinido)
  Put_Line("Valor: " & Integer'Image(Ptr.all));
end Unchecked_Access_Bad;
```

Codificación recomendada

Gestión segura de memoria evitando `Unchecked_Access` y asegurando que el objeto referenciado tenga un tiempo de vida adecuado. Se utiliza memoria dinámica controlada y liberación explícita para evitar referencias colgantes.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Controlled_Access_Good is

   --  Definición de un tipo de puntero a Integer
   type Int_Ptr is access Integer;

   --  Función que reserva memoria dinámica en el heap
   function Get_Ptr return Int_Ptr is
   begin
      --  La memoria persiste más allá del scope de la función
      return new Integer'(10);
   end Get_Ptr;

   Ptr : Int_Ptr;

begin
   --  Se obtiene un puntero válido a memoria dinámica
   Ptr := Get_Ptr;

   --  Uso seguro del puntero
   Put_Line("Valor: " & Integer'Image(Ptr.all));

   --  La memoria es reclamada automáticamente por el runtime de Ada
   --  al finalizar el scope del tipo de acceso
end Controlled_Access_Good;
```

2.1.5.1 Descripción

El uso de `Unchecked_Access` introduce **riesgos de corrupción de memoria** y comportamiento indefinido en tiempo de ejecución. Los punteros creados mediante este atributo pueden referenciar ubicaciones de memoria inválidas, causando fallos de segmentación, accesos a datos corruptos o modificaciones no intencionadas de la memoria.

La verificación automática de **tiempo de vida de objetos** constituye uno de los mecanismos de protección más importantes del lenguaje. Al evitar `Unchecked_Access`, se mantiene la integridad de estas verificaciones y se previenen errores relacionados con referencias colgantes y accesos fuera de límites.

Para mitigar estos riesgos, se recomienda utilizar el atributo `'Access` estándar, que mantiene las verificaciones de seguridad del compilador. En casos donde sea necesario mayor flexibilidad, se pueden emplear **tipos de acceso anónimos** o reestructurar el diseño para evitar la necesidad de referencias no verificadas.

Cuando el uso de `Unchecked_Access` sea inevitable por requisitos específicos del sistema, debe limitarse a **contextos controlados** donde se pueda garantizar manualmente la

validez de las referencias, acompañado de documentación exhaustiva que justifique su necesidad y describa las precauciones tomadas.

Riesgos asociados

- Referencias colgantes (dangling references) que pueden apuntar a memoria liberada o reutilizada.
- Violaciones de la seguridad de tipo (type safety), permitiendo operaciones inseguras sobre memoria.
- Fugas de memoria (memory leaks) al perder el control sobre el ciclo de vida de los objetos.
- Corrupción de memoria (memory corruption) al acceder a objetos fuera de su tiempo de vida válido.
- Dificultades en el análisis estático, perdiendo garantías del compilador sobre la gestión de la memoria.
- Problemas de portabilidad entre compiladores o plataformas por diferencias en la administración de memoria.
- Aumento del riesgo de explotación de vulnerabilidades como uso después de liberar (use-after-free) y corrupción de heap.

Buenas prácticas

- Utilizar tipos de acceso seguros en lugar de aplicar `Unchecked_Access` sobre objetos locales.

```
//-- Obtención de referencia sin verificación de alcance
Obj : aliased Integer := 10;
Ptr : access Integer;
Ptr := Obj'Unchecked_Access;
```

```
//-- Uso de tipo de acceso declarado y reglas de accesibilidad
type Int_Access is access all Integer;
Obj : aliased Integer := 10;
Ptr : Int_Access;
Ptr := Obj'Access;
```

- Evitar devolver referencias `Unchecked_Access` a variables locales desde subprogramas.

```
//-- Retorno de referencia a objeto local
function Get_Ptr return access Integer is
  Local : aliased Integer := 5;
begin
```

```
  return Local'Unchecked_Access;
end Get_Ptr;
```

```
//-- Retorno de objeto gestionado con ámbito adecuado
type Int_Access is access all Integer;
function Get_Ptr return Int_Access is
  Global : aliased Integer := 5;
begin
  return Global'Access;
end Get_Ptr;
```

- Preferir el uso de parámetros `access` en subprogramas en lugar de convertir direcciones con `Unchecked_Access`.

```
//-- Conversión manual de dirección para modificar parámetro
procedure Update is
  Value : aliased Integer := 0;
  Ptr   : access Integer;
begin
  Ptr := Value'Unchecked_Access;
  Ptr.all := 20;
end Update;
```

```
//-- Uso de parámetro access con control de tipos
procedure Update (Ptr : access Integer) is
begin
  Ptr.all := 20;
end Update;
Value : aliased Integer := 0;
Update (Value'Access);
```

2.1.5.2 Estándares metodológicos

- **AdaCore/MISRA 4.1.2**: Pointers Are Not References.
- **AdaCore/MISRA 7.1**: Preventing Undefined Behavior in SPARK.
- **AdaCore DYN.03**: Access Types Without Allocators Policy.
- **AdaCore RPP.08**: Limit Parameter Aliasing.

2.2 Inicialización

2.2.1 [ada.ini.1] - Prohibir lecturas de datos no inicializados o parcialmente inicializados

ID	ada.ini.1	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

La lectura de variables no inicializadas o parcialmente inicializadas constituye una fuente común de **comportamiento indefinido** que puede provocar fallos impredecibles en tiempo de ejecución. Este problema surge cuando el código accede a datos cuyo valor no ha sido establecido explícitamente, resultando en contenido arbitrario de memoria que compromete la **fiabilidad del programa**.

Codificación insegura

Uso de una variable declarada sin valor inicial y posterior lectura antes de asignarle un valor válido. Se pretende mostrar cómo una variable puede contener datos indefinidos si no se inicializa explícitamente, generando comportamientos impredecibles y posibles fallos en ejecución.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  //-- Declaración de una variable sin inicializar
  Contador : Integer;
begin
  //-- Lectura del valor antes de haber sido inicializado
  Put_Line("Valor del contador: " & Integer'Image(Contador));

  //-- Asignación posterior (demasiado tarde para el primer uso)
  Contador := 10;

  //-- Segunda lectura, ahora con valor definido
  Put_Line("Nuevo valor del contador: " & Integer'Image(Contador));
end Main;
```

Codificación recomendada

Inicialización explícita de la variable en el momento de su declaración antes de cualquier lectura. Se busca garantizar que todos los datos tengan un estado válido desde el inicio, evitando valores indeterminados y asegurando un comportamiento predecible del programa.

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Main is
  //-- Declaración con inicialización explícita
  Contador : Integer := 0;
begin
  //-- Lectura segura tras inicialización
  Put_Line("Valor inicial del contador: " & Integer'Image(Contador));

  //-- Asignación controlada de un nuevo valor
  Contador := 10;

  //-- Lectura posterior con valor definido
  Put_Line("Nuevo valor del contador: " & Integer'Image(Contador));
end Main;
```

2.2.1.1 Descripción

Esta regla prohíbe el acceso a variables, campos de registro o elementos de un array que no han sido inicializados explícitamente o que contienen valores parcialmente definidos. El problema surge cuando el programa intenta leer datos cuyo contenido es indeterminado, lo que puede resultar en comportamiento impredecible.

Los datos no inicializados pueden contener valores residuales de operaciones previas en memoria, valores aleatorios del sistema operativo, o contenido parcialmente válido cuando solo algunos campos de una estructura han sido asignados. Esta situación genera **comportamiento indefinido** que compromete la fiabilidad y seguridad del programa.

El riesgo principal radica en que el programa puede funcionar aparentemente de forma correcta en ciertos entornos o condiciones, pero fallar de manera impredecible cuando cambian las circunstancias de ejecución. Esto incluye diferencias entre compiladores, versiones del sistema operativo, o estados previos de la memoria.

La detección de estas violaciones requiere análisis estático del flujo de control para identificar rutas de ejecución donde una variable puede ser leída antes de recibir un valor válido. Los compiladores modernos implementan advertencias para casos obvios, pero no pueden detectar todas las situaciones posibles, especialmente cuando intervienen punteros o estructuras de control complejas.

La mitigación efectiva incluye la inicialización explícita de todas las variables en el punto de declaración, el uso de valores por defecto apropiados para el contexto, y la implementación de verificaciones de estado antes del acceso a datos. También resulta útil establecer patrones de inicialización consistentes y utilizar herramientas de análisis estático que puedan identificar rutas de código problemáticas.

Esta regla es especialmente relevante en sistemas críticos donde el comportamiento determinista es obligatorio, y donde los fallos derivados de datos no inicializados pueden tener consecuencias graves para la seguridad o la integridad del sistema.

Riesgos asociados

- Divulgación de información sensible debido a la lectura de valores de memoria no inicializados.
- Comportamiento indefinido del programa, que puede resultar en fallas en tiempo de ejecución.
- Resultados incorrectos o impredecibles en cálculos y lógica del programa.
- Vulnerabilidad a ataques de seguridad, como explotación de datos residuales en memoria.
- Dificultad en el diagnóstico y la depuración de errores, aumentando los costos de mantenimiento.
- Posibilidad de generación de excepciones no controladas durante el uso de datos no inicializados.
- Corrupción de datos que puede propagarse a lo largo del ciclo de vida del programa.
- Pérdida de integridad en la información procesada por el sistema.

Buenas prácticas

- Inicializar explícitamente todas las variables en el momento de su declaración.

```
procedure Procesar is
  Contador : Integer;
  Total : Integer := 0;
begin
  //-- Uso de variable sin inicializar antes de asignación
  Total := Total + Contador;
end Procesar;
```

```
procedure Procesar is
  Contador : Integer := 0;
  Total : Integer := 0;
begin
  //-- Inicialización explícita antes de cualquier uso
  Total := Total + Contador;
end Procesar;
```

- Inicializar completamente los registros (`records`) usando agregados para evitar campos parcialmente definidos.

```
type Usuario is record
  Id : Integer;
```

```
  Activo : Boolean;
end record;

procedure Crear is
  U : Usuario;
begin
  //-- Asignación parcial dejando campos sin definir
  U.Id := 1001;
end Crear;
```

```
type Usuario is record
  Id : Integer;
  Activo : Boolean;
end record;

procedure Crear is
  U : Usuario := (Id => 1001, Activo => False);
begin
  //-- Inicialización completa del registro mediante agregado
  null;
end Crear;
```

- Inicializar todos los elementos de un array antes de acceder a cualquiera de sus posiciones.

```
procedure Calcular is
  type Vector is array (1 .. 3) of Integer;
  Datos : Vector;
  Suma : Integer := 0;
begin
  //-- Acceso a posiciones no inicializadas del array
  Suma := Datos(1) + Datos(2);
end Calcular;
```

```
procedure Calcular is
  type Vector is array (1 .. 3) of Integer;
  Datos : Vector := (others => 0);
  Suma : Integer := 0;
begin
  //-- Inicialización global del array antes de su uso
  Suma := Datos(1) + Datos(2);
end Calcular;
```

2.2.1.2 Estándares metodológicos

- **AdaCore/MISRA 5.1**: Detecting Reads of Uninitialized Data.

- **AdaCore/MISRA 5.2**: Detecting Partial or Redundant Initialization of Arrays and Structures.

2.3 Estructuras de control y flujo

2.3.1 [ada.flow.1] - Prohibir el uso de cláusulas others en manejadores de excepciones

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

Los manejadores de excepciones que utilizan la cláusula `others` capturan todas las excepciones no especificadas explícitamente, lo que puede **ocultar errores inesperados** y dificultar significativamente las tareas de depuración y mantenimiento del código. Esta práctica impide identificar qué excepciones específicas pueden producirse en cada contexto.

Codificación insegura

Gestión de excepciones donde se captura cualquier error usando una cláusula `others`, ocultando la causa real del fallo y dificultando el diagnóstico. Se pretende mostrar cómo esta práctica impide un tratamiento específico y puede enmascarar errores lógicos o de ejecución.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Division_Others is
  A : Integer := 10;
  B : Integer := 0;
  Result : Integer;
begin
  //-- Se intenta realizar una división que provocará Constraint_Error
  Result := A / B;
  Put_Line("Resultado: " & Integer'Image(Result));

exception
  //-- Captura genérica de cualquier excepción sin distinguir la causa
  when others =>
    Put_Line("Ha ocurrido un error.");
end Division_Others;
```

Codificación recomendada

Gestión de excepciones donde se capturan únicamente las excepciones esperadas de forma explícita, permitiendo un tratamiento claro y controlado de cada caso. Se busca demostrar cómo mejorar la trazabilidad del error y evitar ocultar fallos no previstos.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Division_Sin_Others is
  A : Integer := 10;
  B : Integer := 0;
  Result : Integer;
begin
  //-- Se intenta realizar una división que provocará Constraint_Error
  Result := A / B;
  Put_Line("Resultado: " & Integer'Image(Result));

exception
  //-- Captura explícita de la excepción esperada
  when Constraint_Error =>
    Put_Line("Error: división por cero detectada.");
end Division_Sin_Others;
```

2.3.1.1 Descripción

El uso indiscriminado de `others` en el manejo de excepciones puede enmascarar errores inesperados y dificultar la identificación de fallos durante el desarrollo y mantenimiento del software. Cuando se produce una excepción no contemplada, el manejador genérico puede ejecutar acciones inapropiadas para el tipo específico de error ocurrido.

Esta práctica compromete la **trazabilidad de errores** y puede generar comportamientos impredecibles en situaciones críticas. Los sistemas que requieren alta confiabilidad necesitan un manejo explícito y específico de cada tipo de excepción que puede producirse durante la ejecución.

La mitigación de estos riesgos requiere la implementación de manejadores específicos para cada tipo de excepción esperada. Cuando sea necesario capturar excepciones no previstas, debe limitarse el alcance del manejador genérico y asegurar que las acciones realizadas sean apropiadas para cualquier tipo de excepción que pueda producirse.

El registro detallado de información sobre excepciones capturadas mediante `others` facilita el análisis posterior y la identificación de casos no contemplados durante el diseño. Esta información resulta valiosa para mejorar la robustez del sistema en versiones futuras.

Riesgos asociados

- Pérdida de trazabilidad al no identificar la causa real de la excepción capturada.
- Dificultad para implementar medidas correctivas específicas, ya que no se diferencia entre tipos de excepciones.
- Encubrimiento de errores de programación graves o de lógica, al capturarse excepciones no anticipadas.
- Reducción de la robustez y confiabilidad del software por la manipulación genérica de errores.
- Incremento en la superficie de ataque, ya que un atacante podría desencadenar excepciones inesperadas que quedan ocultas.
- Obstaculización del análisis de seguridad y depuración del código.
- Posible omisión del registro de errores importantes, dificultando el monitoreo y la auditoría.
- Mayor probabilidad de introducir comportamientos inesperados o no seguros tras una excepción no identificada.

Buenas prácticas

- Capturar únicamente las excepciones específicas que se esperan y pueden gestionarse adecuadamente.

```
//-- Manejo genérico que oculta errores no previstos
begin
  Procesar_Datos;
exception
  when others =>
    Log_Error;
end;
```

```
//-- Manejo explícito de excepciones previstas
begin
  Procesar_Datos;
exception
  when Constraint_Error =>
    Log_Error;
  when Data_Error =>
    Recuperar_Datos;
end;
```

- Permitir que las excepciones no previstas se propaguen automáticamente en lugar de capturarlas de forma genérica.

```
//-- Captura global que impide la propagación natural
procedure Calcular is
begin
  Operacion_Critica;
exception
  when others =>
    null;
end Calcular;
```

```
//-- Sin captura genérica para permitir propagación controlada
procedure Calcular is
begin
  Operacion_Critica;
end Calcular;
```

- Definir y capturar excepciones propias cuando se requiera un tratamiento específico en lugar de utilizar `others`.

```
//-- Uso de captura genérica ante error funcional
procedure Validar (Edad : Integer) is
begin
  if Edad < 0 then
    raise Program_Error;
  end if;
exception
  when others =>
    Registrar_Fallo;
end Validar;
```

```
//-- Definición y captura explícita de excepción propia
Edad_Invalida : exception;
procedure Validar (Edad : Integer) is
begin
  if Edad < 0 then
    raise Edad_Invalida;
  end if;
exception
  when Edad_Invalida =>
    Registrar_Fallo;
end Validar;
```

2.3.1.2 Estándares metodológicos

- **AdaCore/MISRA 6.1**: Preventing Undefined Behavior.
- **AdaCore RPP.05**: No Use of "others" in Exception Handlers.

2.3.2 [ada.flow.2] - Prohibir la propagación de excepciones fuera del ámbito visible del subprograma

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

Las excepciones que se propagan fuera del ámbito donde fueron declaradas pueden generar comportamientos impredecibles y dificultar el mantenimiento del código. Cuando una excepción escapa del contexto en el que tiene significado semántico, los módulos que la reciben carecen de información suficiente para manejarla adecuadamente, lo que compromete la **robustez** y **predictibilidad** del sistema.

Codificación insegura

Subprograma que realiza una división donde puede producirse una excepción de restricción al dividir por cero. La intención es observar cómo una excepción no controlada dentro del subprograma se propaga hacia el exterior, superando su ámbito visible y dejando que el llamador reciba el fallo sin control explícito.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is

  function Dividir (A, B : Integer) return Integer is
  begin
    //-- Se realiza la división sin manejar posibles excepciones
    return A / B;
  end Dividir;

  Resultado : Integer;

begin
  //-- Se invoca la función con divisor cero, provocando Constraint_Error
  Resultado := Dividir (10, 0);
  Put_Line ("Resultado: " & Integer'Image (Resultado));
end Main;
```

Codificación recomendada

Misma operación de división, pero incorporando manejo explícito de excepciones dentro del subprograma. El objetivo es evitar que la excepción se propague fuera de su ámbito visible, encapsulando el error y devolviendo un valor controlado al llamador.

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is

  function Dividir (A, B : Integer) return Integer is
  begin
    //-- Se intenta realizar la división
    return A / B;
  exception
    when Constraint_Error =>
      //-- Se captura la excepción y se devuelve un valor seguro
      Put_Line ("Error: division por cero controlada");
      return 0;
  end Dividir;

  Resultado : Integer;

begin
  //-- Se invoca la función con divisor cero, pero la excepción no se propaga
  Resultado := Dividir (10, 0);
  Put_Line ("Resultado: " & Integer'Image (Resultado));
end Main;
```

2.3.2.1 Descripción

Esta regla establece que las excepciones generadas dentro de un subprograma no deben propagarse más allá de su ámbito de visibilidad sin un manejo explícito. Cuando una excepción se propaga fuera del contexto donde fue declarada o donde puede ser identificada apropiadamente, se convierte en una excepción anónima que compromete la capacidad del código llamador para responder de manera específica y controlada.

El problema surge cuando un subprograma permite que excepciones internas escapen hacia niveles superiores de la aplicación donde no pueden ser manejadas de forma precisa. Esto genera **comportamientos impredecibles** y dificulta el diagnóstico de errores, ya que el código receptor no tiene información suficiente sobre la naturaleza específica del fallo.

La propagación no controlada de excepciones puede provocar **terminaciones inesperadas del programa** o estados inconsistentes en la aplicación. Además, expone detalles internos de implementación que deberían permanecer encapsulados, violando los principios de abstracción y ocultación de información.

Para mitigar estos riesgos, los subprogramas deben implementar **bloques de manejo de excepciones** que capturen las excepciones internas y las transformen en respuestas apropiadas para el contexto del llamador. Esto puede incluir la conversión a excepciones del ámbito visible, el retorno de códigos de error específicos, o la ejecución de acciones de limpieza antes de propagar una excepción conocida.

La estrategia de mitigación también implica el **diseño cuidadoso de interfaces** que documenten explícitamente qué excepciones pueden ser generadas por cada subprograma, permitiendo que el código llamador implemente el manejo apropiado para cada caso específico.

Riesgos asociados

- Pérdida de control del flujo del programa, provocando estados inesperados o inestables.
- Exposición de detalles internos de la implementación a módulos externos, violando el encapsulamiento.
- Difícil diagnóstico de fallos, ya que los errores pueden ser capturados lejos de su origen, complicando la trazabilidad.
- Saltos incontrolados de ejecución, que pueden omitir rutinas críticas como liberación de recursos o acciones de limpieza.
- Posible denegación de servicio si las excepciones no se manejan y provocan la terminación prematura del programa.
- Vulnerabilidades de seguridad, incluyendo posibilidad de ataques de inyección si la excepción contiene información sensible o manipulable.
- Inconsistencias en los datos debido a interrupciones inesperadas en las transacciones o procesos críticos.
- Pérdida de información de contexto relevante, dificultando el manejo seguro o la recuperación tras un error.

Buenas prácticas

- Capturar y gestionar todas las excepciones dentro del subprograma antes de que alcancen el llamador.

```
//-- El subprograma no captura la excepción y esta se propaga al exterior
procedure Procesar_Dato (Valor : in Integer) is
begin
  Resultado := 100 / Valor;
end Procesar_Dato;
```

```
//-- El subprograma captura la excepción y gestiona el error localmente
procedure Procesar_Dato (Valor : in Integer) is
begin
  Resultado := 100 / Valor;
exception
  when Constraint_Error =>
    Resultado := 0;
    Log_Error ("División inválida");
```

```
end Procesar_Dato;
```

- Transformar excepciones internas en códigos de estado o tipos de resultado controlados.

```
//-- La excepción interna se propaga directamente al llamador
function Convertir (Texto : in String) return Integer is
begin
  return Integer'Value (Texto);
end Convertir;
```

```
//-- La excepción se captura y devuelve un estado controlado
function Convertir (Texto : in String) return Integer is
  Resultado : Integer := 0;
begin
  Resultado := Integer'Value (Texto);
  return Resultado;
exception
  when Constraint_Error =>
    Log_Error ("Conversión inválida");
    return 0;
end Convertir;
```

- Encapsular llamadas a subprogramas potencialmente peligrosos dentro de bloques locales con manejo explícito de excepciones.

```
//-- La llamada puede propagar excepciones fuera del subprograma
procedure Ejecutar is
begin
  Operacion_Critica;
end Ejecutar;
```

```
//-- La llamada se encapsula en un bloque con manejo local de excepciones
procedure Ejecutar is
begin
  begin
    Operacion_Critica;
  exception
    when others =>
      Log_Error ("Fallo en Operacion_Critica");
  end;
end Ejecutar;
```

- Definir excepciones propias y manejarlas internamente evitando exponer detalles de implementación.

```
//-- Se propaga una excepción predefinida que revela detalles internos
procedure Acceso_Recurso is
begin
  Open (Fichero, In_File, "datos.txt");
end Acceso_Recurso;
```

```
//-- Se captura la excepción interna y se maneja mediante una excepción definida
localmente
procedure Acceso_Recurso is
  Error_Recurso : exception;
begin
  begin
    Open (Fichero, In_File, "datos.txt");
  exception
    when Name_Error =>
      raise Error_Recurso;
  end;
exception
  when Error_Recurso =>
    Log_Error ("No se pudo acceder al recurso");
end Acceso_Recurso;
```

2.3.2.2 Estándares metodológicos

- **AdaCore EXU.03**: No Exception Propagation Beyond Name Visibility.

2.3.3 [ada.flow.3] - Garantizar que no existan excepciones no manejadas en el sistema

ID	**ada.flow.3**	**Uso**	**Obligatorio**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Las excepciones no capturadas pueden provocar terminaciones abruptas del programa, dejando recursos sin liberar y el sistema en estados inconsistentes. Cuando una excepción se propaga sin control hasta el nivel superior, el comportamiento resultante es impredecible y puede comprometer la estabilidad del sistema completo.

Codificación insegura

Realiza una división a partir de valores introducidos por el usuario. La intención es mostrar qué ocurre cuando se produce un error en tiempo de ejecución y no existe ningún mecanismo para capturar y tratar la excepción, provocando la finalización abrupta del sistema.

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Division_Sin_Manejo is
  Numerador   : Integer;
  Denominador : Integer;
  Resultado   : Integer;
begin
  //-- Se solicitan los datos al usuario
  Put("Ingrese el numerador: ");
  Get(Numerador);
  Put("Ingrese el denominador: ");
  Get(Denominador);

  //-- Posible division por cero sin manejo de excepcion
  Resultado := Numerador / Denominador;

  //-- Se muestra el resultado
  Put_Line("Resultado: " & Integer'Image(Resultado));
end Division_Sin_Manejo;
```

Codificación recomendada

Realiza la misma operación de división, incorporando validaciones y un bloque de manejo de excepciones para capturar errores como la división por cero o entradas inválidas. Se busca asegurar que cualquier situación inesperada sea controlada y que el sistema no termine de forma abrupta.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Division_Con_Manejo is
  Numerador   : Integer;
  Denominador : Integer;
  Resultado   : Integer;
begin
  //-- Se solicitan los datos al usuario
  Put("Ingrese el numerador: ");
  Get(Numerador);
  Put("Ingrese el denominador: ");
  Get(Denominador);

  //-- Validacion preventiva para evitar division por cero
  if Denominador = 0 then
    Put_Line("Error: el denominador no puede ser cero.");
    return;
  end if;

  //-- Operacion protegida dentro del flujo controlado
```

```
   Resultado := Numerador / Denominador;
   Put_Line("Resultado: " & Integer'Image(Resultado));

exception
   //-- Manejo explicito de division por cero
   when Constraint_Error =>
      Put_Line("Excepcion controlada: division por cero.");

   //-- Manejo generico para cualquier otra excepcion inesperada
   when others =>
      Put_Line("Excepcion no prevista controlada de forma segura.");
end Division_Con_Manejo;
```

2.3.3.1 Descripción

Las excepciones no manejadas representan una fuente significativa de vulnerabilidades de seguridad y fallos del sistema. Cuando una excepción se propaga sin control hasta el nivel superior del programa, puede provocar terminaciones abruptas, exposición de información sensible en mensajes de error, o dejar el sistema en un estado inconsistente.

El manejo inadecuado de excepciones puede crear vectores de ataque donde los atacantes explotan condiciones de error predecibles para comprometer la integridad del sistema. Los fallos no controlados pueden revelar detalles internos de la implementación, rutas de archivos, estructuras de datos, o información sobre el entorno de ejecución que facilite ataques posteriores.

La propagación descontrolada de excepciones también puede generar **condiciones de carrera** y **estados inconsistentes** en recursos compartidos. Cuando una operación crítica se interrumpe abruptamente, los recursos pueden quedar bloqueados, los datos parcialmente actualizados, o las estructuras de control en estados inválidos.

Para mitigar estos riesgos, es necesario implementar una estrategia de manejo de excepciones que cubra todos los niveles del programa. Esto incluye el uso de bloques `exception` apropiados, la definición de manejadores específicos para diferentes tipos de excepciones, y la implementación de mecanismos de recuperación que mantengan la estabilidad del sistema.

La estrategia debe contemplar tanto las excepciones predefinidas del lenguaje como las excepciones definidas por el usuario. Es importante establecer políticas claras sobre qué excepciones deben manejarse localmente y cuáles pueden propagarse a niveles superiores, siempre garantizando que exista un manejador final que evite la terminación abrupta del programa.

El registro adecuado de excepciones también forma parte de la mitigación, permitiendo el análisis posterior de fallos sin exponer información sensible a usuarios no autorizados.

Los mensajes de error deben ser informativos para los desarrolladores, pero genéricos para los usuarios finales.

Riesgos asociados

- Caída inesperada de la aplicación debido a excepciones no controladas.
- Pérdida de datos o corrupción de la información en ejecuciones parcialmente completadas.
- Exposición de información sensible a través de mensajes de error no gestionados.
- Estado inconsistente del sistema al no ejecutar tareas de limpieza o restauración.
- Vulnerabilidades de denegación de servicio (DoS) si el proceso termina abruptamente bajo ciertas circunstancias.
- Dificultad en el diagnóstico y análisis de fallos debido a la falta de mensajes de error claros y controlados.
- Propagación de errores hacia otras partes del sistema, afectando la confiabilidad global.
- Posibilidad de explotación por atacantes al forzar excepciones, generando comportamientos indeseados o inseguros.

Buenas prácticas

- Capturar explícitamente las excepciones esperadas y definir un manejador general como último recurso.

```
//-- Operación sin manejo explícito de excepciones
procedure Procesar_Dato is
  Valor : Integer := 0;
begin
  Valor := 100 / Valor;
end Procesar_Dato;
```

```
//-- Operación con manejo específico y general de excepciones
procedure Procesar_Dato is
  Valor : Integer := 0;
begin
  Valor := 100 / Valor;
exception
  when Constraint_Error =>
    Put_Line("Error aritmético controlado");
  when others =>
    Put_Line("Excepción no prevista registrada");
```

```
end Procesar_Dato;
```

- Evitar la propagación de excepciones en tareas concurrentes mediante bloques controlados.

```
//-- Tarea sin bloque de manejo interno
task body Trabajador is
begin
  raise Program_Error;
end Trabajador;
```

```
//-- Tarea con bloque de manejo interno de excepciones
task body Trabajador is
begin
  begin
    raise Program_Error;
  exception
    when Program_Error =>
      Put_Line("Fallo controlado en tarea");
    when others =>
      Put_Line("Error inesperado en tarea");
  end;
end Trabajador;
```

- Validar previamente las entradas y condiciones antes de ejecutar operaciones susceptibles de generar excepciones.

```
//-- Uso directo de conversión sin validación previa
procedure Convertir (Texto : String) is
  Numero : Integer;
begin
  Numero := Integer'Value(Texto);
end Convertir;
```

```
//-- Validación previa y manejo de posible excepción
procedure Convertir (Texto : String) is
  Numero : Integer;
begin
  if Texto'Length > 0 then
    begin
      Numero := Integer'Value(Texto);
    exception
      when Constraint_Error =>
        Put_Line("Formato numérico inválido");
    end;
  end if;
```

```
end Convertir;
```

2.3.3.2 Estándares metodológicos

- **AdaCore EXU.02**: No Unhandled Application-Defined Exceptions.

2.3.4 [ada.flow.4] - Probar formalmente la ausencia de excepciones en tiempo de ejecución

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

Las excepciones en tiempo de ejecución pueden interrumpir la ejecución normal del programa y comprometer la estabilidad del sistema. Errores como desbordamientos aritméticos, accesos fuera de límites de arrays o divisiones por cero generan **excepciones no controladas** que afectan la fiabilidad del software. La verificación formal permite demostrar matemáticamente que estas situaciones no ocurrirán durante la ejecución.

Codificación insegura

Realiza una división entre dos números enteros sin establecer precondiciones ni restricciones formales. La intención es mostrar cómo una operación aparentemente simple puede generar excepciones en tiempo de ejecución (como división por cero o desbordamiento) cuando no se definen contratos que permitan demostrar formalmente su seguridad.

```
with Ada.Text_IO; use Ada.Text_IO;

//-- Programa sin contratos formales ni restricciones de rango
procedure Division_Sin_Pruebas is
  function Dividir (A, B : Integer) return Integer is
  begin
    //-- Puede lanzar Constraint_Error si B = 0
    return A / B;
  end Dividir;

  Resultado : Integer;
begin
  //-- Caso que provocará excepción en tiempo de ejecución
  Resultado := Dividir (10, 0);
  Put_Line ("Resultado: " & Integer'Image (Resultado));
end Division_Sin_Pruebas;
```

Codificación recomendada

Define contratos formales y restricciones de tipo en modo SPARK para demostrar la ausencia de excepciones en tiempo de ejecución. Se emplean precondiciones que impiden la división por cero y se habilita el análisis formal para verificar que no se producirán errores en ejecución.

```
with Ada.Text_IO; use Ada.Text_IO;

//-- Activación del modo SPARK para análisis formal
pragma SPARK_Mode (On);

procedure Division_Con_Pruebas is
  //-- Subtipo que limita el rango para evitar desbordamientos
  subtype Rango_Seguro is Integer range -1_000 .. 1_000;

  //-- Función con precondición que evita división por cero
  function Dividir (A, B : Rango_Seguro) return Rango_Seguro
    with Pre => B /= 0;
  function Dividir (A, B : Rango_Seguro) return Rango_Seguro is
  begin
    //-- El analizador formal puede demostrar ausencia de excepciones
    return A / B;
  end Dividir;

  Resultado : Rango_Seguro;
begin
  //-- Llamada que cumple la precondición
  Resultado := Dividir (100, 5);
  Put_Line ("Resultado: " & Integer'Image (Resultado));
end Division_Con_Pruebas;
```

2.3.4.1 Descripción

La verificación formal de la ausencia de excepciones en tiempo de ejecución constituye un proceso de análisis estático que demuestra matemáticamente que un programa no generará errores durante su ejecución. Este enfoque utiliza técnicas de verificación formal para analizar el código fuente y garantizar que todas las operaciones se ejecuten dentro de los límites seguros definidos por el lenguaje.

Las excepciones en tiempo de ejecución representan una categoría de errores que incluye **desbordamientos aritméticos**, **accesos fuera de límites en arrays**, **desreferenciación de punteros nulos** y **violaciones de restricciones de tipos**. Estos errores pueden provocar comportamientos impredecibles, corrupción de datos, fallos del sistema o vulnerabilidades de seguridad explotables.

El proceso de prueba formal examina cada operación potencialmente peligrosa y verifica que las condiciones previas necesarias se cumplan en todos los caminos de ejecución

posibles. Esto incluye la verificación de que los índices de arrays permanezcan dentro de los rangos válidos, que las operaciones aritméticas no excedan los límites de los tipos de datos, y que todas las precondiciones de las subrutinas se satisfagan antes de su invocación.

La implementación de esta práctica requiere el uso de **herramientas de análisis estático especializadas** que pueden generar y verificar condiciones de verificación automáticamente. Estas herramientas analizan el flujo de control y los valores de las variables para determinar si existen caminos de ejecución que puedan llevar a excepciones.

La mitigación efectiva implica la combinación de **especificaciones formales precisas**, **uso de tipos con rangos restringidos**, **validación explícita de entradas** y **diseño de algoritmos que respeten invariantes bien definidos**. La verificación formal funciona mejor cuando se aplica desde las primeras fases del desarrollo, permitiendo que las pruebas matemáticas guíen el diseño del código hacia implementaciones inherentemente seguras.

Riesgos asociados

- Fallas inesperadas en producción debido a excepciones no controladas.
- Comportamientos indeterminados del sistema bajo condiciones de error.
- Pérdida de datos o corrupción de información crítica.
- Vulnerabilidades explotables por atacantes al provocar condiciones excepcionales.
- Interrupción del servicio o denegación de servicio (DoS).
- Dificultad para identificar y depurar fallos en entornos operativos.
- Incumplimiento de normas de seguridad y calidad del software.
- Reducción de la confiabilidad y robustez del sistema.
- Aumento del costo de mantenimiento y soporte técnico.
- Impacto negativo en la reputación y la confianza de los usuarios.

Buenas prácticas

- Definir rangos y subtipos restringidos para eliminar posibles desbordamientos y violaciones de rango.

```
//-- Uso de tipos sin restricciones que pueden provocar Constraint_Error
X : Integer := 1_000_000;
Y : Integer := 2_000_000;
Z : Integer;
```

```
Z := X * Y;
```

```
//-- Uso de subtipos con rango limitado que permiten verificación formal de límites
subtype Small_Int is Integer range -10_000 .. 10_000;
X : Small_Int := 1_000;
Y : Small_Int := 2_000;
Z : Small_Int;
Z := X * Y;
```

- Precondiciones explícitas en subprogramas para demostrar formalmente la validez de entradas.

```
//-- Procedimiento sin restricciones formales sobre los parámetros
procedure Dividir (A, B : in Integer; R : out Integer) is
begin
  R := A / B;
end Dividir;
```

```
//-- Procedimiento con precondición que evita división por cero
procedure Dividir (A, B : in Integer; R : out Integer)
  with Pre => B /= 0 is
begin
  R := A / B;
end Dividir;
```

- Inicializar explícitamente todas las variables para evitar el uso de valores indefinidos.

```
//-- Variable potencialmente no inicializada antes de su uso
Total : Integer;
Resultado : Integer;
Resultado := Total + 10;
```

```
//-- Variable inicializada explícitamente para permitir prueba de ausencia de uso
indebido
Total : Integer := 0;
Resultado : Integer := 0;
Resultado := Total + 10;
```

- Usar tipos derivados con restricciones en índices para evitar accesos fuera de rango en arrays.

```
//-- Acceso a array sin garantía formal de límites válidos
type Vector is array (Integer range 1 .. 10) of Integer;
V : Vector;
```

```
Index : Integer := 20;
Value : Integer := V(Index);
```

```
//-- Índice restringido mediante subtipo compatible con el array
subtype Index_Type is Integer range 1 .. 10;
type Vector is array (Index_Type) of Integer;
V : Vector;
Index : Index_Type := 5;
Value : Integer := V(Index);
```

2.3.4.2 Estándares metodológicos

- **AdaCore/MISRA 7.2**: Proof of Absence of Run-Time Errors in SPARK.
- **AdaCore EXU.04**: Prove Absence of Run-time Exceptions.

2.4 Funciones

2.4.1 [ada.func.1] - Prohibir efectos laterales en funciones

ID	ada.func.1	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Las funciones en Ada deben ser **libres de efectos laterales**, es decir, no deben modificar estados globales, parámetros externos ni realizar operaciones con impacto observable fuera de su valor de retorno.

Codificación insegura

Se usa una función que modifica un estado global para calcular un valor. Se busca mostrar cómo una función que altera variables externas introduce efectos laterales, dificulta el análisis del código y puede generar comportamientos inesperados cuando se reutiliza o se evalúa más de una vez.

```
with Ada.Text_IO; use Ada.Text_IO;

//-- Variable global que será modificada por la función
package Global_State is
  Counter : Integer := 0;
end Global_State;

with Global_State; use Global_State;

procedure Main is
```

```
  //-- Función con efecto lateral: modifica el estado global
  function Increment_And_Get return Integer is
  begin
    Counter := Counter + 1;
    return Counter;
  end Increment_And_Get;

  Result1 : Integer;
  Result2 : Integer;
begin
  //-- Cada llamada cambia el valor global de forma implícita
  Result1 := Increment_And_Get;
  Result2 := Increment_And_Get;

  Put_Line("Resultado 1: " & Integer'Image(Result1));
  Put_Line("Resultado 2: " & Integer'Image(Result2));
  Put_Line("Valor final del contador global: " & Integer'Image(Counter));
end Main;
```

Codificación recomendada

Cálculo del mismo valor sin modificar estado externo dentro de la función. Se muestra cómo una función debe limitarse a devolver un resultado basado únicamente en sus parámetros, delegando cualquier modificación de estado a un procedimiento explícito, lo que mejora la previsibilidad y facilita el mantenimiento.

```
with Ada.Text_IO; use Ada.Text_IO;

//-- Variable global gestionada de forma controlada
package Global_State is
  Counter : Integer := 0;
end Global_State;

with Global_State; use Global_State;

procedure Main is

  //-- Función sin efectos laterales: solo calcula y devuelve un valor
  function Next_Value(Current : Integer) return Integer is
  begin
    return Current + 1;
  end Next_Value;

  //-- Procedimiento explícito para modificar el estado global
  procedure Update_Counter(New_Value : Integer) is
  begin
    Counter := New_Value;
  end Update_Counter;
```

```
   Result1 : Integer;
   Result2 : Integer;
begin
   //-- La función no altera el estado global
   Result1 := Next_Value(Counter);
   Update_Counter(Result1);

   Result2 := Next_Value(Counter);
   Update_Counter(Result2);

   Put_Line("Resultado 1: " & Integer'Image(Result1));
   Put_Line("Resultado 2: " & Integer'Image(Result2));
   Put_Line("Valor final del contador global: " & Integer'Image(Counter));
end Main;
```

2.4.1.1 Descripción

Las funciones deben mantener un comportamiento predecible y determinista, limitándose exclusivamente a calcular y retornar valores sin modificar el estado del programa. Los **efectos laterales** en funciones incluyen la modificación de variables globales, parámetros pasados por referencia, operaciones de entrada/salida, asignaciones a variables externas al ámbito de la función, y cualquier acción que altere el estado del sistema más allá del cálculo del valor de retorno.

Esta restricción se fundamenta en el principio de **transparencia referencial**, donde una función debe producir siempre el mismo resultado cuando se invoca con los mismos argumentos. La presencia de efectos laterales rompe esta propiedad, introduciendo dependencias ocultas que dificultan el análisis estático del código y comprometen la verificación formal de programas.

Los riesgos asociados al incumplimiento de esta regla incluyen comportamientos impredecibles del programa, dificultades en la depuración y mantenimiento del código, problemas de concurrencia en entornos multihilo, y la introducción de errores sutiles que pueden manifestarse de forma intermitente. Además, las funciones con efectos laterales complican significativamente las tareas de testing y verificación formal.

La mitigación de estos riesgos requiere una separación clara entre **funciones puras** y **procedimientos**. Las funciones deben limitarse al cálculo de valores utilizando únicamente sus parámetros de entrada, mientras que los procedimientos deben encargarse de las operaciones que requieren modificar el estado del programa. Esta separación facilita el razonamiento sobre el código y mejora la capacidad de verificación del software.

El compilador puede detectar algunos tipos de efectos laterales mediante análisis estático, pero la responsabilidad principal recae en el diseño de la arquitectura del

software y en las prácticas de codificación que enfaticen la inmutabilidad y la programación funcional donde sea apropiado.

Riesgos asociados

- Generación de resultados inesperados debido a estados modificados fuera del ámbito de la función.
- Dificultad en la validación, trazabilidad y depuración del código por la introducción de comportamientos ocultos.
- Creación de dependencias implícitas entre diferentes partes del código, lo que incrementa el acoplamiento y reduce la mantenibilidad.
- Incremento en la probabilidad de condiciones de carrera y problemas de concurrencia, especialmente en sistemas paralelos o multitarea.
- Violación del principio de idempotencia: una misma llamada a la función puede producir resultados diferentes ante iguales entradas.
- Pérdida de confianza en la reutilización segura de funciones al no poder garantizar que no alteran el estado global o parámetros externos.
- Mayor riesgo de vulnerabilidades de seguridad asociadas a la manipulación inadvertida de datos compartidos o sensibles.
- Complicaciones en el análisis formal, verificación estática y pruebas unitarias de software crítico.

Buenas prácticas

- Evitar modificar variables globales dentro de funciones, limitando las funciones a cálculos puros.

```
//-- Función que altera un estado global
Global_Count : Integer := 0;
function Increment return Integer is
begin
  Global_Count := Global_Count + 1;
  return Global_Count;
end Increment;
```

```
//-- Función que calcula el valor sin modificar estado externo
function Increment(Value : Integer) return Integer is
begin
  return Value + 1;
end Increment;
```

- No utilizar parámetros `in out` o `out` en funciones; reservarlos exclusivamente

para procedimientos.

```
//-- Función que modifica un parámetro recibido
function Update(Value : in out Integer) return Integer is
begin
  Value := Value + 10;
  return Value;
end Update;
```

```
//-- Procedimiento para modificar y función separada para calcular
procedure Update(Value : in out Integer) is
begin
  Value := Value + 10;
end Update;
function Add_Ten(Value : Integer) return Integer is
begin
  return Value + 10;
end Add_Ten;
```

- Evitar operaciones de entrada/salida dentro de funciones para mantener determinismo y ausencia de efectos externos.

```
//-- Función que realiza salida por consola
function Get_Value return Integer is
  Input : Integer;
begin
  Ada.Text_IO.Get(Input);
  return Input;
end Get_Value;
```

```
//-- Procedimiento para entrada y función pura para procesar
procedure Read_Value(Result : out Integer) is
begin
  Ada.Text_IO.Get(Result);
end Read_Value;
function Process(Value : Integer) return Integer is
begin
  return Value * 2;
end Process;
```

- No invocar procedimientos con efectos laterales desde una función.

```
//-- Función que invoca procedimiento que altera estado global
procedure Log_Access is
begin
  Global_Count := Global_Count + 1;
end Log_Access;
```

```
function Compute(Value : Integer) return Integer is
begin
  Log_Access;
  return Value * Value;
end Compute;
```

```
//-- Separación de registro y cálculo
function Compute(Value : Integer) return Integer is
begin
  return Value * Value;
end Compute;
procedure Compute_With_Log(Value : Integer; Result : out Integer) is
begin
  Log_Access;
  Result := Value * Value;
end Compute_With_Log;
```

2.4.1.2 Estándares metodológicos

- **AdaCore/MISRA 6.1**: Preventing Undefined Behavior.
- **AdaCore RPP.06**: Avoid Function Side-Effects.

2.4.2 [ada.func.2] - Definir y verificar precondiciones y postcondiciones en subprogramas críticos

ID	ada.func.2	Uso	Recomendado
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

Definir y verificar **precondiciones** y **postcondiciones** en subprogramas críticos permite garantizar que los contratos de comportamiento se cumplen explícitamente, reduciendo errores lógicos, estados inconsistentes y vulnerabilidades derivadas de entradas o resultados no validados.

Codificación insegura

Subprograma crítico que realiza una división entera utilizada para calcular ratios financieros. Se busca mostrar la importancia de validar condiciones de entrada y salida cuando la operación puede provocar errores en tiempo de ejecución o resultados inconsistentes. El objetivo es observar qué ocurre cuando no se definen ni verifican condiciones formales.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Division_Sin_Contratos is
```

```
  //-- Función crítica que realiza una división sin validar entradas
  function Dividir (X : Integer; Y : Integer) return Integer is
  begin
    return X / Y;
  end Dividir;

  Resultado : Integer;
begin
  //-- Llamada con divisor en cero que provocará excepción en tiempo de ejecución
  Resultado := Dividir (10, 0);
  Put_Line ("Resultado: " & Integer'Image (Resultado));
end Division_Sin_Contratos;
```

Codificación recomendada

Subprograma crítico que realiza la misma división entera, incorporando precondiciones y postcondiciones mediante contratos de Ada 2012. Se pretende asegurar que el divisor nunca sea cero y que el resultado cumpla la propiedad matemática esperada, permitiendo detectar incumplimientos de forma explícita y controlada.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Division_Con_Contratos is
  pragma Assertion_Policy (Check);

  //-- Función crítica con precondición y postcondición definidas
  function Dividir (X : Integer; Y : Integer) return Integer
    with Pre  => Y /= 0,
       Post => Dividir'Result * Y = X is
  begin
    return X / Y;
  end Dividir;

  Resultado : Integer;
begin
  //-- Llamada válida que cumple la precondición
  Resultado := Dividir (10, 2);
  Put_Line ("Resultado: " & Integer'Image (Resultado));
end Division_Con_Contratos;
```

2.4.2.1 Descripción

Las precondiciones y postcondiciones constituyen contratos formales que especifican el comportamiento esperado de los subprogramas. Las precondiciones establecen las condiciones que deben cumplirse antes de la ejecución, mientras que las postcondiciones definen el estado garantizado tras la finalización exitosa del subprograma.

La implementación de estos contratos mediante `pragma Precondition` y `pragma Postcondition` permite detectar violaciones de las especificaciones durante la ejecución. Esta verificación automática identifica llamadas incorrectas a subprogramas y comportamientos inesperados que podrían comprometer la integridad del sistema.

Los subprogramas críticos requieren especial atención debido a su impacto directo en la **seguridad y fiabilidad** del sistema. La ausencia de contratos formales en estas funciones puede resultar en comportamientos indefinidos, corrupción de datos o fallos del sistema cuando se invocan con parámetros inválidos o en estados inconsistentes.

La verificación de contratos actúa como una **barrera de protección** que intercepta condiciones anómalas antes de que puedan propagarse y causar daños mayores. Esta detección temprana facilita la identificación de errores durante las fases de desarrollo y pruebas, reduciendo significativamente los costes de corrección.

La mitigación efectiva requiere la identificación sistemática de subprogramas críticos y la definición rigurosa de sus contratos. La activación de la verificación de contratos durante las fases de desarrollo y pruebas, combinada con el manejo apropiado de las excepciones generadas por violaciones, proporciona un mecanismo robusto de detección de errores.

Los contratos formales también mejoran la **documentación del código** al explicitar las expectativas y garantías de cada subprograma, facilitando el mantenimiento y la comprensión del comportamiento esperado por parte de otros desarrolladores.

Riesgos asociados

- Introducción de errores lógicos no detectados durante la ejecución del subprograma.
- Vulnerabilidad ante ataques por datos de entrada maliciosos o no validados.
- Dificultad para detectar y depurar fallos en la lógica del sistema.
- Comportamientos imprevistos que pueden comprometer la integridad de los datos.
- Anulación de mecanismos de seguridad debido a suposiciones incorrectas sobre el estado de entrada o salida.
- Fallo en el cumplimiento de los requisitos funcionales y de seguridad del sistema.
- Pérdida de confiabilidad y robustez en componentes críticos del software.
- Propagación de estados inválidos que pueden afectar a otros módulos o servicios dependientes.
- Incremento del riesgo de comprometer la disponibilidad y continuidad del servicio.

Buenas prácticas

- Definir precondiciones explícitas en subprogramas críticos para validar rangos y evitar estados inválidos antes de la ejecución.

```
//-- Subprograma sin validación explícita de entrada
procedure Dividir (A, B : in Integer; Resultado : out Integer) is
begin
  Resultado := A / B;
end Dividir;
```

```
//-- Subprograma con precondición que valida divisor distinto de cero
procedure Dividir (A, B : in Integer; Resultado : out Integer)
with
  Pre => B /= 0
is
begin
  Resultado := A / B;
end Dividir;
```

- Especificar postcondiciones para asegurar que el resultado cumple las propiedades esperadas tras la ejecución.

```
//-- Función sin verificación formal del resultado
function Incrementar (Valor : Integer) return Integer is
begin
  return Valor + 1;
end Incrementar;
```

```
//-- Función con postcondición que garantiza el valor esperado
function Incrementar (Valor : Integer) return Integer
with
  Post => Incrementar'Result = Valor + 1
is
begin
  return Valor + 1;
end Incrementar;
```

- Combinar precondiciones y postcondiciones en operaciones sensibles para reforzar la integridad del estado.

```
//-- Transferencia sin validaciones contractuales
procedure Transferir (Saldo_Origen  : in out Integer;
                      Saldo_Destino : in out Integer;
                      Monto         : in Integer) is
begin
  Saldo_Origen  := Saldo_Origen - Monto;
```

```
  Saldo_Destino := Saldo_Destino + Monto;
end Transferir;
```

```
//-- Transferencia con contrato que valida entrada y estado final
procedure Transferir (Saldo_Origen  : in out Integer;
                      Saldo_Destino : in out Integer;
                      Monto         : in Integer)
with
  Pre  => Monto > 0 and then Saldo_Origen >= Monto,
  Post => Saldo_Origen = Saldo_Origen'Old - Monto and then
          Saldo_Destino = Saldo_Destino'Old + Monto
is
begin
  Saldo_Origen  := Saldo_Origen - Monto;
  Saldo_Destino := Saldo_Destino + Monto;
end Transferir;
```

- Utilizar tipos con rango definido junto con precondiciones para reforzar restricciones de seguridad en parámetros críticos.

```
//-- Uso de tipo genérico sin restricciones explícitas
procedure Establecer_Edad (Edad : in Integer) is
begin
  null;
end Establecer_Edad;
```

```
//-- Uso de subtipo restringido y precondición adicional
subtype Edad_Valida is Integer range 0 .. 120;
procedure Establecer_Edad (Edad : in Edad_Valida)
with
  Pre => Edad >= 0
is
begin
  null;
end Establecer_Edad;
```

2.4.2.2 Estándares metodológicos

- **AdaCore RPP.09**: Use Precondition and Postcondition Contracts.

2.4.3 [ada.func.3] - Prohibir parámetros con modo out no asignados explícitamente

ID	ada.func.3	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Garantizar que todos los parámetros declarados con modo `out` sean asignados explícitamente antes de finalizar el subprograma, evitando el uso o propagación de valores indefinidos.

Codificación insegura

Uso de un parámetro `out` que no se asigna en todos los caminos de ejecución. Se busca mostrar cómo un flujo condicional incompleto puede dejar el parámetro sin inicializar, generando comportamiento impredecible y advertencias del compilador. La intención es evidenciar el riesgo de no garantizar una asignación explícita.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  //-- Procedimiento que puede no asignar el parámetro de salida
  procedure Calcular_Valor (Entrada : in Integer; Resultado : out Integer) is
  begin
    //-- Solo se asigna Resultado si la condición se cumple
    if Entrada > 0 then
      Resultado := Entrada * 2;
    end if;
    //-- Si Entrada es menor o igual a 0, Resultado queda sin asignar
  end Calcular_Valor;

  Valor : Integer := 0;
begin
  //-- Llamada con valor que no cumple la condición
  Calcular_Valor (Entrada => 0, Resultado => Valor);

  //-- Se imprime un valor que puede no haber sido inicializado correctamente
  Put_Line ("Valor obtenido: " & Integer'Image (Valor));
end Main;
```

Codificación recomendada

Uso de un parámetro `out` asegurando su asignación explícita en todos los caminos de ejecución. Se muestra cómo estructurar el flujo condicional para garantizar que el parámetro siempre tenga un valor definido antes de finalizar el procedimiento, eliminando ambigüedades y advertencias.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  //-- Procedimiento que garantiza la asignación del parámetro de salida
  procedure Calcular_Valor (Entrada : in Integer; Resultado : out Integer) is
  begin
    //-- Se cubren todos los caminos posibles
    if Entrada > 0 then
      Resultado := Entrada * 2;
    else
      //-- Asignación explícita cuando no se cumple la condición
      Resultado := 0;
    end if;
  end Calcular_Valor;

  Valor : Integer := 0;
begin
  //-- Llamada con valor que no cumple la condición
  Calcular_Valor (Entrada => 0, Resultado => Valor);

  //-- El valor siempre estará correctamente inicializado
  Put_Line ("Valor obtenido: " & Integer'Image (Valor));
end Main;
```

2.4.3.1 Descripción

Los parámetros con modo `out` requieren asignación explícita de valores antes del retorno de la subrutina. Cuando un parámetro `out` no recibe asignación, su valor permanece indefinido, lo que puede provocar comportamientos impredecibles en el código que invoca la subrutina.

El compilador no garantiza la inicialización automática de parámetros `out`, por lo que el programador debe asegurar que todos estos parámetros reciban valores válidos en todas las rutas de ejecución posibles. Esta responsabilidad incluye tanto las rutas normales como las excepcionales del flujo de control.

La ausencia de asignación en parámetros `out` introduce **vulnerabilidades de seguridad** relacionadas con el uso de datos no inicializados. Estos datos pueden contener información residual de la memoria, exponiendo potencialmente datos sensibles o causando fallos en la lógica del programa.

Los riesgos se incrementan cuando las subrutinas manejan múltiples rutas de ejecución, especialmente aquellas que incluyen manejo de excepciones o estructuras de control complejas. En estos escenarios, resulta más probable omitir la asignación en alguna rama del código.

La mitigación efectiva requiere **verificación sistemática** de todas las rutas de ejecución para confirmar que cada parámetro `out` recibe asignación. Las herramientas de análisis

estático pueden detectar automáticamente estas omisiones durante el desarrollo. Adicionalmente, el diseño de subrutinas debe minimizar la complejidad del flujo de control para reducir la probabilidad de omisiones inadvertidas.

La aplicación consistente de esta regla mejora la **predictibilidad del comportamiento** del programa y reduce significativamente los errores relacionados con datos no inicializados, contribuyendo a la robustez general del sistema.

Riesgos asociados

- Fugas de información no intencionada debido a que las variables `out` pueden contener datos residuales no inicializados.
- Comportamiento indefinido si el parámetro `out` se utiliza sin haber sido asignado, lo que puede provocar errores en tiempo de ejecución.
- Difícil trazabilidad y depuración de errores al no quedar claro si los parámetros han sido inicializados.
- Potenciales vulnerabilidades de seguridad por acceso a memoria con valores no controlados.
- Resultados incorrectos en los procesos que dependan del valor retornado por un parámetro `out` no asignado.
- Incompatibilidades o problemas de portabilidad al utilizar compiladores o plataformas con diferentes políticas de inicialización de memoria.
- Incremento en la dificultad del análisis estático al no poder determinar fácilmente el estado de los parámetros.

Buenas prácticas

- Asignar explícitamente todos los parámetros `out` en todos los caminos de ejecución.

```
//-- Parámetro out sin asignación en todos los caminos
procedure Calcular (Entrada : Integer; Resultado : out Integer) is
begin
  if Entrada > 0 then
    Resultado := Entrada * 2;
  end if;
end Calcular;
```

```
//-- Asignación explícita del parámetro out en todos los caminos
procedure Calcular (Entrada : Integer; Resultado : out Integer) is
begin
  if Entrada > 0 then
    Resultado := Entrada * 2;
  else
```

```
    Resultado := 0;
  end if;
end Calcular;
```

- Inicializar parámetros `out` al inicio del procedimiento para evitar estados indefinidos.

```
//-- Parámetro out condicionado a lógica posterior
procedure Obtener_Valor (Flag : Boolean; Dato : out Integer) is
begin
  if Flag then
    Dato := 10;
  end if;
end Obtener_Valor;
```

```
//-- Inicialización temprana del parámetro out
procedure Obtener_Valor (Flag : Boolean; Dato : out Integer) is
begin
  Dato := 0;
  if Flag then
    Dato := 10;
  end if;
end Obtener_Valor;
```

- Evitar salidas anticipadas sin asignar previamente el parámetro `out`.

```
//-- Retorno anticipado sin asignar parámetro out
procedure Procesar (Entrada : Integer; Salida : out Integer) is
begin
  if Entrada < 0 then
    return;
  end if;
  Salida := Entrada + 1;
end Procesar;
```

```
//-- Asignación antes de cualquier retorno anticipado
procedure Procesar (Entrada : Integer; Salida : out Integer) is
begin
  if Entrada < 0 then
    Salida := 0;
    return;
  end if;
  Salida := Entrada + 1;
end Procesar;
```

2.4.3.2 Estándares metodológicos

- **AdaCore RPP.04**: No Unassigned Mode-Out Procedure Parameters.

2.4.4 [ada.func.4] - Obligar al uso del valor de retorno de las funciones

ID	ada.func.4	Uso	Obligatorio
CWSS Score	74	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

Obligar al uso explícito del valor de retorno de las funciones para evitar ignorar resultados críticos que puedan indicar errores, fallos de validación o estados inseguros del sistema.

Codificación insegura

Validación de una contraseña mediante una función que devuelve un valor booleano. Se muestra cómo una función puede indicar éxito o fallo, y cómo ignorar ese valor provoca que la lógica de seguridad no se aplique realmente, dejando el sistema en un estado inseguro.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

  //-- Función que valida si la contraseña tiene al menos 8 caracteres
  function Validate_Password (Pwd : String) return Boolean is
  begin
    if Pwd'Length >= 8 then
      return True;
    else
      return False;
    end if;
  end Validate_Password;

  //-- Variable para almacenar el resultado, pero no se usará correctamente
  Is_Valid : Boolean;

begin
  //-- Se llama a la función, pero no se utiliza su valor para tomar decisiones
  Is_Valid := Validate_Password ("1234");

  //-- El programa continúa sin importar si la contraseña es válida o no
  Put_Line ("Acceso concedido");

end Main;
```

Codificación recomendada

Validación de una contraseña utilizando el valor de retorno de la función para decidir si se concede o no el acceso. Se fuerza el uso explícito del resultado, asegurando que la lógica de seguridad dependa directamente del valor devuelto por la función.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is

  //-- Función que valida si la contraseña tiene al menos 8 caracteres
  function Validate_Password (Pwd : String) return Boolean is
  begin
    if Pwd'Length >= 8 then
      return True;
    else
      return False;
    end if;
  end Validate_Password;

begin
  //-- Se utiliza directamente el valor de retorno para controlar el flujo
  if Validate_Password ("1234") then
    //-- Solo se concede acceso si la validación es correcta
    Put_Line ("Acceso concedido");
  else
    //-- Se bloquea el acceso si la validación falla
    Put_Line ("Acceso denegado");
  end if;

end Main;
```

2.4.4.1 Descripción

Las funciones que devuelven valores están diseñadas para comunicar información al código que las invoca. Ignorar estos valores de retorno puede provocar comportamientos impredecibles y errores difíciles de detectar durante la ejecución del programa.

Esta regla establece que **todo valor devuelto por una función debe ser utilizado o explícitamente descartado**. El compilador debe generar advertencias cuando se detecte que el valor de retorno de una función no se asigna a una variable, no se utiliza en una expresión o no se descarta de manera explícita.

Los valores de retorno suelen indicar el estado de la operación realizada, códigos de error, resultados de cálculos o información sobre el éxito o fallo de la función. Cuando estos valores se ignoran, el programa pierde información crítica que podría ser necesaria para tomar decisiones correctas en el flujo de ejecución.

Los riesgos asociados al incumplimiento de esta regla incluyen la propagación silenciosa de errores, estados inconsistentes en el programa y fallos en tiempo de ejecución que podrían haberse evitado mediante la verificación adecuada de los valores devueltos.

La mitigación de estos problemas se logra mediante la implementación de políticas de compilación que requieran el uso explícito de todos los valores de retorno. Cuando el valor no sea necesario para la lógica del programa, debe descartarse de forma explícita utilizando los mecanismos que proporciona el lenguaje para este propósito.

Esta práctica mejora la **trazabilidad del código** y facilita el mantenimiento, ya que hace visible la intención del programador respecto al manejo de los valores devueltos por las funciones.

Riesgos asociados

- Pérdida de información importante o resultados de operaciones críticas.
- Fallo silencioso en la ejecución debido a la omisión de errores o excepciones retornadas.
- Mayor probabilidad de que los errores pasen inadvertidos, dificultando la depuración.
- Comportamientos inesperados del sistema por asumir que la función se ejecutó correctamente.
- Exposición a vulnerabilidades de seguridad por ignorar estados o condiciones devueltos.
- Corrupción de datos o inconsistencias en el flujo del programa.
- Incremento de deuda técnica y baja mantenibilidad del código.

Buenas prácticas

- Asignar siempre el valor de retorno de una función a una variable o constante para asegurar su validación posterior.

```
//-- Llamada a función sin usar el valor devuelto
Calcular_Total (Importe);
```

```
//-- Asignación del valor devuelto a una variable para su validación
  Total : Float;
  Total := Calcular_Total (Importe);
  if Total > 0.0 then
    Procesar (Total);
  end if;
```

- Validar explícitamente el valor retornado antes de continuar con el flujo lógico.

```
//-- Uso del resultado sin comprobación previa
  Token : String := Obtener_Token (Usuario);
  Autorizar (Token);
```

```
//-- Comprobación del resultado antes de su uso
  Token : String := Obtener_Token (Usuario);
  if Token'Length > 0 then
    Autorizar (Token);
  else
    Registrar_Error;
  end if;
```

- Encapsular funciones cuyo valor de retorno sea crítico dentro de estructuras que obliguen a su verificación.

```
//-- Ignorar el indicador de éxito devuelto por la función
  Actualizar_Registro (Id, Datos);
```

```
//-- Evaluar el indicador de éxito antes de continuar
  Exito : Boolean;
  Exito := Actualizar_Registro (Id, Datos);
  if not Exito then
    Registrar_Error;
  end if;
```

2.4.4.2 Estándares metodológicos

- **AdaCore/MISRA 3.2.1**: Handling the Result of Function Calls.
- **AdaCore RPP.11**: Always Use the Result of Function Calls.

2.5 Gestión de memoria

2.5.1 [ada.mem.1] - Prohibir la asignación dinámica salvo que esté estrictamente controlada y acotada estáticamente

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

La asignación dinámica de memoria introduce riesgos significativos en sistemas críticos, incluyendo fragmentación del heap, agotamiento de memoria y comportamiento temporal impredecible. Los fallos en la gestión dinámica pueden provocar **memory**

leaks, corrupción de datos y vulnerabilidades de seguridad que comprometen la estabilidad del sistema.

Codificación insegura

Gestión de memoria mediante asignación dinámica sin límites ni control, mostrando cómo se reservan objetos en el heap dentro de un bucle sin una política de liberación ni una cota máxima definida. Se busca evidenciar los riesgos de consumo no controlado de memoria y pérdida de referencias.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  //-- Declaración de un tipo acceso a Integer
  type Int_Ptr is access all Integer;

  //-- Variable para almacenar la referencia dinámica
  Ptr : Int_Ptr;

begin
  //-- Bucle que realiza asignaciones dinámicas sin límite ni liberación
  for I in 1 .. 10 loop
    //-- Reserva de memoria en el heap
    Ptr := new Integer'(I);
    //-- Uso del valor reservado
    Put_Line ("Valor: " & Integer'Image (Ptr.all));
  end loop;

  //-- No se libera la memoria reservada
  Put_Line ("Fin del programa");
end Main;
```

Codificación recomendada

Uso de memoria estática mediante una estructura de datos acotada en tiempo de compilación, evitando la asignación dinámica. Se muestra cómo almacenar y procesar valores dentro de límites definidos, garantizando un consumo de memoria predecible y controlado.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  //-- Declaración de un array con tamaño fijo definido en compilación
  type Int_Array is array (1 .. 10) of Integer;

  //-- Instancia del array en memoria estática
  Data : Int_Array;
```

```
begin
  //-- Inicialización controlada dentro de los límites definidos
  for I in Data'Range loop
    Data(I) := I;
  end loop;

  //-- Uso seguro de los datos almacenados
  for I in Data'Range loop
    Put_Line ("Valor: " & Integer'Image (Data(I)));
  end loop;

  //-- No se realiza ninguna asignación dinámica
  Put_Line ("Fin del programa");
end Main;
```

2.5.1.1 Descripción

La asignación dinámica de memoria introduce incertidumbre en el comportamiento temporal y espacial de los programas. Esta práctica puede generar fragmentación del heap, fallos de asignación impredecibles y comportamiento no determinista en tiempo de ejecución.

El uso no controlado de operadores como `new` y procedimientos de asignación dinámica puede provocar **agotamiento de memoria**, **fugas de memoria** y **corrupción del heap**. Estos problemas son especialmente críticos en sistemas empotrados, de tiempo real o de alta disponibilidad donde los recursos son limitados y el comportamiento debe ser predecible.

La asignación dinámica sin restricciones dificulta el análisis estático del programa y complica la verificación de propiedades de seguridad. Los analizadores de código no pueden determinar con precisión los límites de memoria utilizados ni garantizar que las operaciones se completarán exitosamente.

Cuando la asignación dinámica sea inevitable, debe implementarse dentro de **límites estáticamente verificables**. Esto incluye el uso de pools de memoria preasignados, restricciones en el número máximo de asignaciones y mecanismos de liberación deterministas.

Las técnicas de mitigación incluyen el uso de tipos con tamaño conocido en tiempo de compilación, estructuras de datos estáticas dimensionadas para el caso peor, y la implementación de gestores de memoria personalizados con límites fijos. También resulta efectivo el empleo de técnicas de análisis de flujo para demostrar que todas las asignaciones tienen su correspondiente liberación.

El control estricto de la asignación dinámica permite mantener el **determinismo temporal** y facilita la certificación en dominios donde se requiere alta integridad del software.

Riesgos asociados

- Fugas de memoria (memory leaks) al no liberar recursos dinámicos correctamente.
- Fragmentación de memoria que puede degradar el rendimiento con el tiempo.
- Posibilidad de ataques de denegación de servicio (DoS) por consumo excesivo de memoria.
- Pérdida del control sobre los límites y la vida útil de los objetos en memoria.
- Introducción de vulnerabilidades de seguridad explotables por desbordamientos de memoria (heap overflow).
- Dificultad en la verificación formal y análisis estático del código debido a la dinámica impredecible.
- Condiciones de carrera y errores difíciles de depurar en sistemas concurrentes debido a la gestión inadecuada de la memoria compartida.
- Incremento de errores de punteros colgantes o dobles liberaciones (dangling pointers, double frees).
- Pérdida de determinismo al ejecutar en entornos de tiempo real o sistemas embebidos críticos.

Buenas prácticas

- Evitar el uso de `access types` generales y preferir objetos estáticos o con límites conocidos en tiempo de compilación.

```
//-- Uso de acceso dinámico sin límites claros
type Int_Ptr is access all Integer;
Ptr : Int_Ptr := new Integer;
```

```
//-- Uso de objeto estático con tamaño definido en compilación
Value : Integer := 0;
```

- Si la asignación dinámica es imprescindible, encapsularla en un paquete que limite y controle explícitamente el número máximo de instancias.

```
//-- Creación repetida de objetos sin control de cantidad
type Buffer is array (Positive range <>) of Integer;
type Buffer_Ptr is access Buffer;
B1 : Buffer_Ptr := new Buffer(1 .. 1024);
B2 : Buffer_Ptr := new Buffer(1 .. 2048);
```

```
//-- Encapsulación con límite máximo definido estáticamente
```

```
Max_Buffers : constant Positive := 2;
type Buffer_Array is array (1 .. Max_Buffers) of Buffer(1 .. 1024);
Buffers : Buffer_Array;
```

- Preferir estructuras de tamaño fijo o contenedores acotados en lugar de estructuras dinámicas no restringidas.

```
//-- Lista enlazada basada en reserva dinámica
type Node;
type Node_Ptr is access Node;
type Node is record
  Value : Integer;
  Next  : Node_Ptr;
end record;
Head : Node_Ptr := new Node'(Value => 1, Next => null);
```

```
//-- Array de tamaño fijo definido en compilación
type Node_Array is array (1 .. 10) of Integer;
Nodes : Node_Array := (others => 0);
```

- Evitar desasignaciones manuales dispersas y centralizar la gestión de memoria cuando esté estrictamente justificada.

```
//-- Liberación manual dispersa con riesgo de errores
type Int_Ptr is access Integer;
procedure Free is new Ada.Unchecked_Deallocation(Integer, Int_Ptr);
Ptr : Int_Ptr := new Integer'(10);
Free(Ptr);
```

```
//-- Uso de objeto controlado automáticamente sin liberación explícita
Value : Integer := 10;
```

2.5.1.2 Estándares metodológicos

- **AdaCore/MISRA 7.1**: Preventing Undefined Behavior in SPARK.
- **AdaCore DYN.02**: Traditional Static Allocation Policy.
- **AdaCore DYN.04**: Minimal Dynamic Allocation Policy.

2.5.2 [ada.mem.2] - Prohibir el uso de allocators en tipos access no restringidos

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

Los `allocators` en tipos `access` no restringidos pueden provocar **memory leaks** y **dangling pointers** cuando no se gestiona correctamente la liberación de memoria. La ausencia de restricciones permite que los punteros accedan a memoria ya liberada o que permanezcan referencias a objetos no alojados, comprometiendo la estabilidad del programa.

Codificación insegura

Gestión dinámica de memoria mediante un tipo `access` no restringido utilizando un `allocator`. Se observa cómo se reserva memoria en el heap con `new` y se usa el puntero resultante. El objetivo es entender el impacto del uso de `allocators` en tipos `access` generales y los riesgos asociados como fugas de memoria o pérdida de control del ciclo de vida.

```
with Ada.Text_IO; use Ada.Text_IO;

//-- Ejemplo que utiliza un allocator en un tipo access no restringido
procedure Main is
  //-- Declaración de un tipo access general (no restringido)
  type Int_Ptr is access Integer;

  //-- Variable de tipo access
  Ptr : Int_Ptr;
begin
  //-- Reserva dinámica de memoria en el heap usando allocator
  Ptr := new Integer'(10);

  //-- Uso del valor apuntado
  Put_Line("Valor: " & Integer'Image(Ptr.all));

  //-- No se libera explícitamente la memoria
  //-- En aplicaciones grandes esto puede provocar fugas
end Main;
```

Codificación recomendada

Uso de un objeto `aliased` con un tipo `access` sin recurrir a `allocators`. Se muestra cómo referenciar memoria existente sin reservar dinámicamente, manteniendo un control

claro del ciclo de vida del objeto y evitando riesgos asociados a la gestión manual del heap.

```
with Ada.Text_IO; use Ada.Text_IO;

//-- Ejemplo que evita el uso de allocators en tipos access
procedure Main is
  //-- Declaración de un tipo access general
  type Int_Ptr is access all Integer;

  //-- Objeto aliased con duración automática
  Value : aliased Integer := 10;

  //-- Puntero que referenciará un objeto existente
  Ptr : Int_Ptr;
begin
  //-- Asociación del puntero a un objeto ya existente
  Ptr := Value'Access;

  //-- Uso seguro del valor apuntado
  Put_Line("Valor: " & Integer'Image(Ptr.all));

  //-- No hay reserva dinámica, el ciclo de vida está controlado
end Main;
```

2.5.2.1 Descripción

Los `allocators` permiten la asignación dinámica de memoria durante la ejecución del programa. Cuando se utilizan con tipos `access` no restringidos, estos `allocators` pueden crear objetos en el heap sin limitaciones específicas sobre el tipo o subtipo del objeto referenciado.

El uso de `allocators` con tipos `access` no restringidos introduce **vulnerabilidades de gestión de memoria** que pueden comprometer la seguridad del sistema. La ausencia de restricciones permite la creación de referencias a objetos de diferentes subtipos, lo que puede generar inconsistencias en el acceso a memoria y comportamientos impredecibles.

Esta práctica puede provocar **corrupción de memoria** cuando el programa intenta acceder a objetos a través de referencias que no corresponden al tipo esperado. Los errores resultantes incluyen accesos fuera de límites, escrituras en ubicaciones de memoria incorrectas y violaciones de la integridad de los datos.

Los tipos `access` no restringidos también dificultan el **análisis estático del código**, ya que el compilador no puede verificar completamente la compatibilidad de tipos en tiempo de compilación. Esta limitación reduce la efectividad de las verificaciones automáticas de seguridad.

La mitigación de estos riesgos se logra mediante el uso de **tipos** `access` **restringidos** que especifican claramente el subtipo de los objetos que pueden referenciar. Esta restricción permite al compilador realizar verificaciones más estrictas y reduce las posibilidades de error.

Alternativamente, se puede evitar completamente la asignación dinámica utilizando **objetos con duración determinada** o estructuras de datos estáticas cuando el diseño del programa lo permita. Esta aproximación elimina los riesgos asociados con la gestión manual de memoria dinámica.

Riesgos asociados

- Fugas de memoria (memory leaks) debido a la imposibilidad de liberar la memoria asignada dinámicamente.
- Acceso a memoria liberada (dangling pointers), provocando comportamientos indefinidos o fallos en tiempo de ejecución.
- Incremento en la superficie de ataque para explotación de vulnerabilidades relacionadas con la manipulación no segura de punteros.
- Dificultad en el razonamiento sobre la propiedad de los objetos y el ciclo de vida de los recursos, lo que puede derivar en errores lógicos o de diseño.
- Posibilidad de corrupción de memoria o modificaciones no autorizadas de datos debido a accesos erróneos por punteros no controlados.
- Dificultades en la verificación formal y en el análisis estático del código, reduciendo la comprobabilidad de la seguridad y fiabilidad del software.
- Mayor complejidad para pruebas y depuración, incrementando los costos de mantenimiento y desarrollo seguro.

Buenas prácticas

- Utilizar objetos declarados estáticamente y el atributo `'Access` en lugar de `allocators` sobre tipos `access` no restringidos.

```
//-- Uso de allocator en tipo access no restringido
type Int_Ptr is access Integer;
Ptr : Int_Ptr := new Integer'(10);
```

```
//-- Uso de objeto aliased y atributo 'Access
type Int_Ptr is access all Integer;
Val : aliased Integer := 10;
Ptr : Int_Ptr := Val'Access;
```

- Definir tipos `access` como `access constant` para evitar modificaciones y reducir riesgos asociados a memoria dinámica.

```
//-- Tipo access no restringido con reserva dinámica
type Data_Ptr is access Integer;
D : Data_Ptr := new Integer;
D.all := 5;
```

```
//-- Tipo access constant asociado a objeto existente
type Data_Ptr is access constant Integer;
Val : aliased Integer := 5;
D : Data_Ptr := Val'Access;
```

- Emplear pools de almacenamiento restringidos cuando sea imprescindible usar `access types`.

```
//-- Allocator sin control de almacenamiento
type Buffer_Ptr is access String;
B : Buffer_Ptr := new String'(1 .. 10 => 'A');
```

```
//-- Access type asociado a Storage_Pool específico
for Buffer_Ptr'Storage_Pool use My_Pool;
type Buffer_Ptr is access all String;
B : Buffer_Ptr := My_Buffer'Access;
```

2.5.2.2 Estándares metodológicos

- **AdaCore/MISRA 4.1**: Enforcing Strong Typing for Pointers.
- **AdaCore DYN.03**: Access Types Without Allocators Policy.

2.5.3 [ada.mem.3] - Utilizar pools de almacenamiento definidos por el usuario cuando se permita memoria dinámica

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

La gestión de memoria dinámica mediante el mecanismo estándar puede introducir vulnerabilidades de seguridad y comportamientos impredecibles en aplicaciones críticas. El uso indiscriminado de `new` sin control específico sobre la asignación y liberación de memoria puede generar **fragmentación**, **agotamiento de recursos** y **fallos no deterministas** en tiempo de ejecución.

Codificación insegura

Gestión de memoria dinámica utilizando el `pool` de almacenamiento por defecto del compilador. Se reserva memoria en tiempo de ejecución para múltiples objetos sin

ningún control explícito sobre el mecanismo de asignación ni liberación, lo que impide auditar el uso de memoria y puede provocar agotamiento de recursos. El objetivo es mostrar cómo la memoria dinámica queda delegada completamente al entorno sin restricciones definidas por el desarrollador.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Deallocation;

procedure Default_Pool_Example is
  //-- Tipo de dato dinámico
  type Integer_Access is access all Integer;

  //-- Procedimiento para liberar memoria
  procedure Free is new Ada.Unchecked_Deallocation (Integer, Integer_Access);

  Ptr : Integer_Access;
begin
  //-- Reserva repetida de memoria usando el pool por defecto
  for I in 1 .. 5 loop
    Ptr := new Integer'(I);
    Put_Line ("Valor reservado: " & Integer'Image (Ptr.all));
  end loop;

  //-- Liberación manual solo del último elemento
  Free (Ptr);

  Put_Line ("Fin del programa");
end Default_Pool_Example;
```

Codificación recomendada

Gestión de memoria dinámica utilizando un `pool` de almacenamiento definido por el usuario. Se implementa un `pool` simple con un buffer estático y se asocia explícitamente al tipo de acceso. Esto permite controlar el tamaño máximo disponible y el mecanismo de asignación, limitando el consumo de memoria y haciendo el comportamiento más predecible y auditable.

```
with Ada.Text_IO; use Ada.Text_IO;
with System.Storage_Pools;
with System.Storage_Elements;

procedure Custom_Pool_Example is
  use System.Storage_Pools;
  use System.Storage_Elements;

  //-- Definición de un pool de almacenamiento simple
  type Simple_Pool is new Root_Storage_Pool with record
    Buffer : Storage_Array (1 .. 1024);
```

```
  Index  : Storage_Offset := 1;
end record;

overriding procedure Allocate
  (Pool  : in out Simple_Pool;
   Address : out System.Address;
   Size  : Storage_Count;
   Align : Storage_Count);

overriding procedure Deallocate
  (Pool  : in out Simple_Pool;
   Address : System.Address;
   Size  : Storage_Count;
   Align : Storage_Count);

overriding function Storage_Size (Pool : Simple_Pool)
  return Storage_Count;

overriding procedure Allocate
  (Pool  : in out Simple_Pool;
   Address : out System.Address;
   Size  : Storage_Count;
   Align : Storage_Count) is
begin
  //-- Asignación secuencial dentro del buffer
  Address := Pool.Buffer (Pool.Index)'Address;
  Pool.Index := Pool.Index + Storage_Offset (Size);
end Allocate;

overriding procedure Deallocate
  (Pool  : in out Simple_Pool;
   Address : System.Address;
   Size  : Storage_Count;
   Align : Storage_Count) is
begin
  //-- No se implementa liberación individual en este pool simple
  null;
end Deallocate;

overriding function Storage_Size (Pool : Simple_Pool)
  return Storage_Count is
begin
  return Pool.Buffer'Length;
end Storage_Size;

My_Pool : Simple_Pool;

//-- Tipo de acceso asociado explícitamente al pool personalizado
type Integer_Access is access all Integer;
```

```
   for Integer_Access'Storage_Pool use My_Pool;

   Ptr : Integer_Access;
begin
   //-- Reserva controlada dentro del pool definido por el usuario
   Ptr := new Integer'(42);
   Put_Line ("Valor reservado en pool personalizado: " & Integer'Image (Ptr.all));

   Put_Line ("Fin del programa");
end Custom_Pool_Example;
```

2.5.3.1 Descripción

Los `pools` de almacenamiento definidos por el usuario proporcionan un mecanismo de control explícito sobre la gestión de memoria dinámica. Esta práctica permite supervisar y limitar las operaciones de asignación y liberación de memoria mediante la implementación de tipos derivados de `Root_Storage_Pool`.

El uso de `pools` personalizados ofrece **visibilidad completa** sobre las operaciones de memoria que realiza el programa. A diferencia del `pool` de almacenamiento por defecto del sistema, los `pools` definidos por el usuario permiten implementar políticas específicas de asignación, establecer límites de memoria y registrar todas las operaciones de gestión de memoria.

Esta aproximación resulta especialmente relevante en **sistemas críticos** donde el comportamiento determinista de la memoria constituye un requisito. Los `pools` personalizados pueden implementar estrategias de asignación que eviten la fragmentación, establezcan límites máximos de memoria utilizada o proporcionen garantías temporales sobre las operaciones de asignación y liberación.

Los riesgos asociados al uso del `pool` de almacenamiento por defecto incluyen la **falta de control** sobre el comportamiento de la memoria, la imposibilidad de establecer límites y la ausencia de mecanismos de monitorización. Estos factores pueden derivar en comportamientos impredecibles del sistema, especialmente en entornos con restricciones de memoria o requisitos de tiempo real.

La mitigación de estos riesgos se logra mediante la implementación de `pools` que incorporen **mecanismos de control** específicos para el dominio de aplicación. Esto incluye la validación de parámetros de asignación, el establecimiento de límites de memoria y la implementación de políticas de recuperación ante condiciones de memoria insuficiente.

Los `pools` personalizados también facilitan la **detección temprana** de problemas relacionados con la gestión de memoria, como fugas de memoria o patrones de uso anómalos, mediante la incorporación de capacidades de registro y monitorización en las operaciones de asignación y liberación.

Riesgos asociados

- Fugas de memoria (memory leaks) por una gestión inadecuada de la memoria dinámica.
- Fragmentación de memoria, lo que puede degradar el rendimiento del sistema e incluso provocar fallos por agotamiento de memoria.
- Condiciones de carrera y corrupción de la memoria en sistemas concurrentes al no controlar adecuadamente el acceso a la memoria dinámica.
- Imposibilidad de cumplir requisitos de tiempo real o determinismo debido a la imprevisibilidad de la asignación y liberación de memoria.
- Dificultad para rastrear y depurar problemas relacionados con el uso de memoria dinámica sin una estructura clara de `pools` de almacenamiento.
- Exposición a ataques de denegación de servicio (DoS) por consumo excesivo e incontrolado de recursos de memoria.
- Incremento del riesgo de desreferenciación de punteros nulos o colgantes (dangling pointers).
- Pérdida de integridad o confidencialidad de la información almacenada en memoria dinámica no gestionada de forma segura.

Buenas prácticas

- Definir y asociar explícitamente un `Storage_Pool` personalizado para controlar y auditar las asignaciones dinámicas.

```
//-- Asignación dinámica usando el pool por defecto
type Data is record
  Value : Integer;
end record;

type Data_Access is access Data;

Obj : Data_Access := new Data;
```

```
//-- Asociación explícita de un pool de almacenamiento definido por el usuario
with System.Storage_Pools;

type My_Pool is new System.Storage_Pools.Root_Storage_Pool with null record;

overriding procedure Allocate
  (Pool : in out My_Pool; Address : out System.Address;
  Size : System.Storage_Elements.Storage_Count;
  Alignment : System.Storage_Elements.Storage_Count);
```

```
overriding procedure Deallocate
  (Pool : in out My_Pool; Address : System.Address;
  Size : System.Storage_Elements.Storage_Count;
  Alignment : System.Storage_Elements.Storage_Count);

My_Pool_Instance : My_Pool;

type Data_Access is access Data;
for Data_Access'Storage_Pool use My_Pool_Instance;

Obj : Data_Access := new Data;
```

- Restringir y validar el tamaño máximo de las asignaciones dentro del `Storage_Pool` para prevenir agotamiento de memoria.

```
//-- Pool sin validación de límites de memoria
overriding procedure Allocate
  (Pool : in out My_Pool; Address : out System.Address;
  Size : System.Storage_Elements.Storage_Count;
  Alignment : System.Storage_Elements.Storage_Count) is
begin
  Address := System.Memory.Alloc (Size);
end Allocate;
```

```
//-- Pool con validación de tamaño máximo permitido
Max_Size : constant System.Storage_Elements.Storage_Count := 1024;
overriding procedure Allocate
  (Pool : in out My_Pool; Address : out System.Address;
  Size : System.Storage_Elements.Storage_Count;
  Alignment : System.Storage_Elements.Storage_Count) is
begin
  if Size > Max_Size then
    raise Storage_Error;
  end if;
  Address := System.Memory.Alloc (Size);
end Allocate;
```

2.5.3.2 Estándares metodológicos

- **AdaCore/MISRA 6.1**: Preventing Undefined Behavior.
- **AdaCore DYN.05**: User-Defined Storage Pools Policy.

2.5.4 [ada.mem.4] - Determinar y verificar estáticamente el uso máximo de pila

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

El uso excesivo de memoria de pila puede provocar desbordamientos que comprometen la estabilidad y seguridad del programa. Los sistemas embebidos y de tiempo real requieren un control preciso del consumo de pila para garantizar un funcionamiento predecible. La recursión profunda, las variables locales de gran tamaño y las llamadas anidadas contribuyen al crecimiento del uso de pila de forma difícil de predecir en tiempo de ejecución.

Codificación insegura

Uso de recursividad sin restricciones para calcular un factorial. La intención es mostrar cómo el consumo de pila crece con cada llamada recursiva y cómo, sin límites explícitos ni verificación estática, el tamaño máximo de pila no puede determinarse fácilmente en tiempo de compilación.

```
with Ada.Text_IO;
  use Ada.Text_IO;

  procedure Main is

  //-- Función recursiva sin límite de profundidad
  function Factorial (N : Natural) return Natural is
  begin
    if N = 0 then
      return 1;
    else
      return N * Factorial (N - 1);
    end if;
  end Factorial;

    Result : Natural;
  begin
  //-- Llamada con valor potencialmente grande
    Result := Factorial (10);
    Put_Line ("Resultado: " & Natural'Image (Result));
  end Main;
```

Codificación recomendada

Uso de cálculo iterativo junto con una restricción de compilación que prohíbe la recursividad. La intención es permitir que el consumo máximo de pila sea determinable estáticamente, evitando crecimiento dinámico por llamadas anidadas y facilitando el análisis estático del uso de memoria.

```
pragma Restrictions (No_Recursion);
with Ada.Text_IO;
  use Ada.Text_IO;

  procedure Main is

  //-- Función iterativa con uso de pila constante
  function Factorial (N : Natural) return Natural is
    Result : Natural := 1;
  begin
    for I in 1 .. N loop
      Result := Result * I;
    end loop;
    return Result;
  end Factorial;

    Value  : constant Natural := 10;
    Result : Natural;
  begin
  //-- Cálculo con consumo de pila predecible
    Result := Factorial (Value);
    Put_Line ("Resultado: " & Natural'Image (Result));
  end Main;
```

2.5.4.1 Descripción

El uso de la pila durante la ejecución de un programa puede crecer de manera impredecible si no se controla adecuadamente. Esta situación genera riesgos de desbordamiento que pueden comprometer la estabilidad del sistema y crear vulnerabilidades de seguridad.

La determinación estática del **uso máximo de pila** requiere analizar todas las posibles rutas de ejecución del programa, incluyendo las llamadas a funciones anidadas, la asignación de variables locales y el paso de parámetros. Este análisis debe considerar tanto el caso promedio como el peor escenario posible.

La verificación estática implica establecer límites conocidos antes de la compilación y garantizar que el programa no los excederá durante su ejecución. Esto incluye el análisis de la **profundidad máxima de llamadas** y el tamaño de los marcos de pila individuales.

Los sistemas embebidos y de tiempo real son particularmente sensibles a este problema, ya que operan con recursos limitados y requieren comportamiento predecible. Un desbordamiento de pila puede causar corrupción de memoria, terminación inesperada del programa o ejecución de código malicioso.

La mitigación efectiva requiere herramientas de análisis estático que puedan calcular el uso máximo de pila en tiempo de compilación. También es necesario establecer **límites conservadores** basados en los recursos disponibles del sistema e implementar verificaciones que garanticen el cumplimiento de estos límites.

El análisis debe ser especialmente riguroso en presencia de recursión, llamadas indirectas a través de punteros a función, y en sistemas que utilizan múltiples tareas o hilos de ejecución concurrentes.

Riesgos asociados

- Desbordamiento de pila (stack overflow) durante la ejecución, causando fallos inesperados del programa.
- Interrupción prematura de tareas críticas debido a falta de espacio en la pila, lo que puede provocar condiciones de denegación de servicio.
- Vulnerabilidades explotables por atacantes, como ejecución de código arbitrario o corrupción de memoria.
- Comportamientos no deterministas y difícilmente reproducibles que complican la detección y corrección de errores.
- Pérdida de datos o corrupción de información sensible debido a interrupciones abruptas.
- Inestabilidad del sistema, especialmente en sistemas embebidos o tiempo real, donde la memoria es limitada o crítica.
- Dificultad para lograr la certificación de sistemas de seguridad si no se garantiza el uso controlado de recursos.

Buenas prácticas

- Definir límites explícitos de tamaño de pila mediante restricciones de tareas y análisis de profundidad de llamadas.

```
procedure Procesar is
  procedure Recursivo (N : Natural) is
  begin
    Recursivo (N - 1);
  end Recursivo;
begin
  Recursivo (1000);
```

```
end Procesar;
```

```
//-- Limitación explícita de profundidad y control iterativo
procedure Procesar is
  Max_Profundidad : constant Natural := 100;
begin
  for I in reverse 1 .. Max_Profundidad loop
    null;
  end loop;
end Procesar;
```

- Restringir el tamaño de objetos locales grandes evitando asignaciones automáticas en pila.

```
procedure Calculo is
  Buffer : array (1 .. 10_000) of Integer;
begin
  null;
end Calculo;
```

```
//-- Uso de almacenamiento dinámico controlado en lugar de pila
procedure Calculo is
  type Buffer_Type is array (1 .. 10_000) of Integer;
  type Buffer_Access is access Buffer_Type;
  Buffer : Buffer_Access := new Buffer_Type;
begin
  null;
end Calculo;
```

- Aplicar análisis estático y restricciones de configuración para tamaño máximo de pila en tareas concurrentes.

```
task body Tarea is
  Dato : String (1 .. 5000);
begin
  null;
end Tarea;
```

```
//-- Configuración explícita del tamaño de pila de la tarea
task type Tarea;
for Tarea'Storage_Size use 8192;
task body Tarea is
  Dato : String (1 .. 1000);
begin
  null;
```

```
end Tarea;
```

2.5.4.2 Estándares metodológicos

- **AdaCore/MISRA 7.2**: Proof of Absence of Run-Time Errors in SPARK.
- **AdaCore DYN.06**: Statically Determine Maximum Stack Requirements.

2.5.5 [ada.mem.5] - Prohibir múltiples liberaciones de una misma región de memoria

ID	ada.mem.5	Uso	Obligatorio
CWSS Score	90	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La liberación múltiple de una misma región de memoria constituye un error crítico que puede provocar **corrupción del heap**, comportamiento indefinido del programa y vulnerabilidades de seguridad explotables. Este problema surge cuando el código intenta liberar un puntero que ya ha sido previamente liberado, o cuando múltiples punteros apuntan a la misma dirección y se liberan de forma independiente.

Codificación insegura

Gestión manual de memoria dinámica utilizando `Ada.Unchecked_Deallocation` para liberar un objeto reservado en el heap. Se trabaja con un puntero a entero, mostrando cómo se reserva memoria y se libera. El objetivo es observar el comportamiento cuando se intenta liberar dos veces la misma región de memoria, lo que puede provocar comportamiento indefinido o errores en tiempo de ejecución.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Deallocation;

procedure Main is
  type Int_Ptr is access all Integer;
  procedure Free is new Ada.Unchecked_Deallocation(Integer, Int_Ptr);
  Ptr : Int_Ptr := new Integer'(10);
begin
  //-- Reserva memoria dinámica para un entero
  Put_Line("Valor inicial: " & Integer'Image(Ptr.all));

  //-- Primera liberación de la región de memoria
  Free(Ptr);

  //-- Segunda liberación sobre la misma dirección, comportamiento indefinido
  Free(Ptr);
```

```
  Put_Line("Fin del programa");
end Main;
```

Codificación recomendada

Gestión segura de memoria dinámica, incorporando validaciones antes de liberar la memoria y anulando explícitamente el puntero tras la liberación. El objetivo es evitar múltiples liberaciones de la misma región de memoria y reducir el riesgo de comportamiento indefinido.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Int_Ptr is access all Integer;

  procedure Safe_Release (Ptr : in out Int_Ptr) is
  begin
     if Ptr /= null then
        Ptr := null;
     end if;
  end Safe_Release;

  Ptr : Int_Ptr := new Integer'(10);
begin
  -- Reserva memoria dinámica para un entero
  Put_Line ("Valor inicial: " & Integer'Image (Ptr.all));

  -- Verifica que el puntero no sea nulo antes de invalidar la referencia
  Safe_Release (Ptr);

  -- Nueva verificación que evita reutilizar la misma referencia
  Safe_Release (Ptr);

  Put_Line ("Fin del programa");
end Main;
```

2.5.5.1 Descripción

La liberación múltiple de una misma región de memoria constituye un error crítico que puede comprometer tanto la estabilidad como la seguridad de las aplicaciones. Este problema surge cuando el código intenta desasignar memoria que ya ha sido previamente liberada, creando una condición de **double free** que puede tener consecuencias impredecibles en el comportamiento del programa.

En el contexto de la gestión manual de memoria, cada región asignada dinámicamente debe liberarse exactamente una vez durante el ciclo de vida del programa. La violación de este principio puede ocurrir por diversas causas: errores en la lógica de control de

flujo, manejo inadecuado de punteros compartidos entre diferentes partes del código, o falta de sincronización en entornos concurrentes.

Los riesgos asociados a esta práctica incluyen corrupción del heap de memoria, comportamiento indefinido del programa, y potenciales vulnerabilidades de seguridad que pueden ser explotadas por atacantes. El gestor de memoria del sistema puede quedar en un estado inconsistente, afectando no solo al programa actual sino potencialmente a otros procesos del sistema.

La mitigación de este problema requiere un diseño cuidadoso de la gestión de memoria que establezca **propietarios claros** para cada región asignada. Es necesario implementar patrones de programación que garanticen que cada asignación tenga una única liberación correspondiente, evitando situaciones donde múltiples rutas de ejecución puedan intentar desasignar la misma memoria.

Las estrategias de prevención incluyen el uso de técnicas de programación defensiva, como la inicialización de punteros a valores nulos después de la liberación, y la implementación de mecanismos de seguimiento que permitan verificar el estado de las regiones de memoria antes de su liberación. En arquitecturas más complejas, puede ser necesario establecer protocolos específicos para la transferencia de propiedad de memoria entre diferentes componentes del sistema.

Riesgos asociados

- Doble liberación de memoria (double free), lo que puede provocar corrupción en el heap.
- Posibilidad de ejecución de código arbitrario por un atacante que explote la corrupción del heap.
- Fallos inesperados de la aplicación tales como cuelgues o terminación abrupta.
- Aparición de vulnerabilidades de seguridad que pueden ser utilizadas en exploits como use-after-free.
- Comportamiento indefinido del programa, incrementando la dificultad para encontrar el origen de los errores.
- Pérdida de integridad en las estructuras de datos internas de control de memoria.
- Riesgo de denegación de servicio (DoS) debido a fallos en la gestión de recursos.
- Dificultad aumentada en las tareas de depuración y mantenimiento.

Buenas prácticas

- Asignar el puntero a `null` inmediatamente después de liberar memoria para evitar reutilización accidental.

```
//-- Liberación sin anular la referencia
procedure Free is new Ada.Unchecked_Deallocation(Integer, Int_Ptr);
Free(P);
Free(P);
```

```
declare
  -- Simula la liberación sin usar Unchecked_Deallocation.
  -- El objeto apuntado no se desasigna explícitamente.
  -- La seguridad se mantiene invalidando el acceso.
  procedure Release (X : in out Int_Ptr) is
  begin
    X := null;
  end Release;

begin
  Release (P);
  if P /= null then
    Release (P);
  end if;
end;
```

- Encapsular la liberación en un procedimiento seguro que verifique el estado del puntero antes de liberar.

```
//-- Liberación directa desde múltiples puntos del código
procedure Free is new Ada.Unchecked_Deallocation(Integer, Int_Ptr);
if Condition_A then
  Free(P);
end if;
if Condition_B then
  Free(P);
end if;
```

```
-- Procedimiento centralizado que valida y anula la referencia
procedure Safe_Release (Ptr : in out Int_Ptr) is
begin
   if Ptr /= null then
      Ptr := null;
   end if;
end Safe_Release;

Safe_Release (P);
Safe_Release (P);
```

- Limitar el uso de `Unchecked_Deallocation` mediante tipos controlados para gestionar automáticamente la liberación.

```
//-- Uso manual de liberación en distintos bloques
procedure Free is new Ada.Unchecked_Deallocation(Integer, Int_Ptr);
Free(P);
Free(P);
```

```
-- Tipo controlado que invalida la referencia una sola vez
type Managed_Ptr is new Ada.Finalization.Controlled with record
   Ptr : Int_Ptr := null;
end record;

overriding procedure Finalize (Obj : in out Managed_Ptr) is
begin
   if Obj.Ptr /= null then
      Obj.Ptr := null;
   end if;
end Finalize;
```

2.5.5.2 Estándares metodológicos

- **AdaCore RCL.01**: No Multiple Reclamations.

2.5.6 [ada.mem.6] - Liberar únicamente memoria previamente asignada y válida

ID	**ada.mem.6**	**Uso**	**Obligatorio**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

La liberación incorrecta de memoria constituye una fuente frecuente de errores críticos en aplicaciones. Intentar liberar memoria no asignada previamente, ya liberada, o con referencias inválidas puede provocar **comportamiento indefinido**, corrupción de datos y fallos del sistema. Estos errores son especialmente peligrosos porque pueden no manifestarse inmediatamente, dificultando su detección durante las pruebas.

Codificación insegura

Gestión manual de memoria dinámica utilizando `Unchecked_Deallocation` para liberar un objeto. Se muestra el uso de un puntero que puede quedar inválido y luego ser liberado nuevamente, lo que permite comprender el riesgo de liberar memoria que ya fue liberada o que no es válida.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Deallocation;

procedure Main is
```

```
  type Int_Ptr is access all Integer;
  procedure Free is new Ada.Unchecked_Deallocation (Integer, Int_Ptr);
  Ptr : Int_Ptr := null;
begin
  //-- Reserva dinámica de memoria
  Ptr := new Integer'(10);

  //-- Liberación inicial de memoria válida
  Free(Ptr);

  //-- Segundo intento de liberación sobre el mismo puntero (memoria ya liberada)
  Free(Ptr);

  Put_Line("Fin del programa");
end Main;
```

Codificación recomendada

Gestión segura de memoria dinámica asegurando que únicamente se libera memoria previamente asignada y válida. Se valida el puntero antes de liberarlo y se asigna explícitamente a `null` tras la liberación para evitar dobles liberaciones o uso de memoria inválida.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  type Int_Ptr is access all Integer;

  procedure Safe_Release (Ptr : in out Int_Ptr) is
  begin
     if Ptr /= null then
        Ptr := null;
     end if;
  end Safe_Release;

  Ptr : Int_Ptr := null;
begin
  -- Reserva dinámica de memoria
  Ptr := new Integer'(20);

  -- Verificación de puntero antes de invalidar la referencia
  Safe_Release (Ptr);

  -- Segundo intento protegido por validación
  Safe_Release (Ptr);

  Put_Line ("Fin del programa");
```

```
end Main;
```

2.5.6.1 Descripción

Esta regla establece que las operaciones de liberación de memoria deben aplicarse exclusivamente sobre bloques de memoria que han sido previamente asignados de forma válida y que no han sido liberados con anterioridad. La violación de este principio constituye una de las fuentes más comunes de errores de gestión de memoria en aplicaciones de sistemas.

El problema surge cuando se intenta liberar memoria que nunca fue asignada dinámicamente, que ya fue liberada previamente, o que apunta a direcciones de memoria no válidas. Estas situaciones pueden provocar **comportamiento indefinido** del programa, corrupción del heap de memoria, y fallos críticos del sistema en tiempo de ejecución.

Los riesgos asociados incluyen la corrupción de las estructuras internas del gestor de memoria, lo que puede comprometer la estabilidad de toda la aplicación. Además, estos errores pueden ser explotados por atacantes para ejecutar código malicioso o provocar denegaciones de servicio, especialmente en sistemas donde la gestión manual de memoria es frecuente.

La mitigación efectiva requiere implementar un control riguroso del ciclo de vida de los punteros y las referencias a memoria dinámica. Es necesario establecer patrones de programación que garanticen la correspondencia entre operaciones de asignación y liberación, así como la validación del estado de los punteros antes de su uso en operaciones de liberación.

El uso de herramientas de análisis estático y dinámico puede detectar estas violaciones durante el desarrollo y las pruebas. La aplicación de técnicas de programación defensiva, como la inicialización de punteros y la verificación de su validez, contribuye significativamente a prevenir estos errores de gestión de memoria.

Riesgos asociados

- Doble liberación de memoria (double free), que puede llevar a corrupción de memoria.
- Intento de liberar memoria no asignada, pudiendo causar fallos de ejecución y comportamientos indefinidos.
- Vulnerabilidad a ataques de escalada de privilegios si un atacante manipula punteros para liberar memoria arbitraria.
- Fugas de memoria por gestión incorrecta de la misma, disminuyendo la disponibilidad y eficiencia del sistema.
- Condiciones de carrera en entornos concurrentes al liberar memoria

incorrectamente compartida.

- Caídas inesperadas del programa (crash).
- Exposición a ataques de ejecución de código arbitrario por corrupción de los datos en la memoria liberada incorrectamente.
- Dificultad en la depuración y mantenimiento del código debido a comportamientos inesperados.

Buenas prácticas

- Verificar que el puntero no sea nulo y que haya sido inicializado antes de liberarlo.

```
type Int_Ptr is access Integer;
procedure Free is new Ada.Unchecked_Deallocation (Integer, Int_Ptr);
Ptr : Int_Ptr;
//-- Liberación sin verificar inicialización
Free (Ptr);
```

```
//-- Verificación previa de puntero válido
if Ptr /= null then
  Free (Ptr);
  Ptr := null;
end if;
```

- Evitar liberar memoria más de una vez estableciendo el puntero a `null` inmediatamente después de la liberación.

```
type Int_Ptr is access Integer;
procedure Free is new Ada.Unchecked_Deallocation (Integer, Int_Ptr);
Ptr : Int_Ptr := new Integer'(10);
//-- Doble liberación del mismo bloque de memoria
Free (Ptr);
Free (Ptr);
```

```
//-- Liberación controlada y anulación del puntero
Free (Ptr);
Ptr := null;
```

- Garantizar que únicamente se libere memoria obtenida dinámicamente y no objetos estáticos o automáticos.

```
type Int_Ptr is access Integer;
procedure Free is new Ada.Unchecked_Deallocation (Integer, Int_Ptr);
Value : aliased Integer := 5;
```

```
Ptr : Int_Ptr;
//-- Intento de liberar objeto no asignado dinámicamente
Ptr := Value'Access;
Free (Ptr);
```

```
//-- Liberación exclusiva de memoria creada con new
Ptr := new Integer'(5);
Free (Ptr);
Ptr := null;
```

2.5.6.2 Estándares metodológicos

- **AdaCore RCL.02**: Only Reclaim Allocated Storage.

2.5.7 [ada.mem.7] - Liberar memoria exclusivamente en el mismo pool donde fue asignada

ID	ada.mem.7	Uso	Obligatorio
CWSS Score	86	Impacto técnico	Muy alto
Dimensión	Integridad	S-SDLC	Build

La gestión incorrecta de pools de memoria puede provocar **corrupción de datos** y **comportamiento indefinido** del programa. Cuando la memoria se libera en un pool diferente al que fue asignada originalmente, se compromete la integridad del sistema de gestión de memoria, generando errores difíciles de detectar y depurar que pueden manifestarse como fallos intermitentes o degradación del rendimiento.

Codificación insegura

Gestión de memoria dinámica usando dos `Storage_Pool` diferentes. Se reserva memoria en un `pool` específico y posteriormente se intenta liberar desde un contexto distinto. El objetivo es observar qué ocurre cuando la liberación no se realiza en el mismo `pool` donde se asignó la memoria.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Storage_Pools;
with Ada.Unchecked_Deallocation;

procedure Main is
  //-- Definición de un pool simple derivado del pool raíz
  type Simple_Pool is new Ada.Storage_Pools.Root_Storage_Pool with null record;

  overriding procedure Allocate
    (Pool  : in out Simple_Pool;
     Storage_Address : out System.Address;
```

```
   Size_In_Storage_Elements : Ada.Storage_Elements.Storage_Count;
   Alignment : Ada.Storage_Elements.Storage_Count);

overriding procedure Deallocate
  (Pool  : in out Simple_Pool;
   Storage_Address : System.Address;
   Size_In_Storage_Elements : Ada.Storage_Elements.Storage_Count;
   Alignment : Ada.Storage_Elements.Storage_Count);

overriding function Storage_Size
  (Pool : Simple_Pool) return Ada.Storage_Elements.Storage_Count;

procedure Allocate
  (Pool  : in out Simple_Pool;
   Storage_Address : out System.Address;
   Size_In_Storage_Elements : Ada.Storage_Elements.Storage_Count;
   Alignment : Ada.Storage_Elements.Storage_Count) is
begin
  Storage_Address := System.Memory.Alloc (Integer (Size_In_Storage_Elements));
end Allocate;

procedure Deallocate
  (Pool  : in out Simple_Pool;
   Storage_Address : System.Address;
   Size_In_Storage_Elements : Ada.Storage_Elements.Storage_Count;
   Alignment : Ada.Storage_Elements.Storage_Count) is
begin
  System.Memory.Free (Storage_Address);
end Deallocate;

function Storage_Size
  (Pool : Simple_Pool) return Ada.Storage_Elements.Storage_Count is
begin
  return 1024;
end Storage_Size;

Pool_A : Simple_Pool;
Pool_B : Simple_Pool;

type Int_Access_A is access Integer;
for Int_Access_A'Storage_Pool use Pool_A;

type Int_Access_B is access Integer;
for Int_Access_B'Storage_Pool use Pool_B;

procedure Free_B is new Ada.Unchecked_Deallocation (Integer, Int_Access_B);

Ptr_A : Int_Access_A := new Integer'(10);
Ptr_B : Int_Access_B;
```

```
begin
  //-- Conversión forzada entre tipos de acceso con distinto pool
  Ptr_B := Int_Access_B (Ptr_A);

  //-- Liberación usando el pool incorrecto
  Free_B (Ptr_B);

  Put_Line ("Finalizado");
end Main;
```

Codificación recomendada

Gestión de memoria dinámica utilizando un único `Storage_Pool` coherente para la asignación y liberación. Se reserva memoria y se libera mediante el mismo tipo de acceso asociado al mismo pool. Se busca garantizar consistencia en la gestión de memoria y evitar errores en tiempo de ejecución.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Storage_Pools;
with Ada.Storage_Elements;
with System;

procedure Main is
  -- Definición de un pool simple derivado del pool raíz
  type Simple_Pool is new Ada.Storage_Pools.Root_Storage_Pool with null record;

  overriding procedure Allocate
    (Pool              : in out Simple_Pool;
    Storage_Address    : out System.Address;
    Size_In_Storage_Elements : Ada.Storage_Elements.Storage_Count;
    Alignment          : Ada.Storage_Elements.Storage_Count);

  overriding procedure Deallocate
    (Pool              : in out Simple_Pool;
    Storage_Address    : System.Address;
    Size_In_Storage_Elements : Ada.Storage_Elements.Storage_Count;
    Alignment          : Ada.Storage_Elements.Storage_Count);

  overriding function Storage_Size
    (Pool : Simple_Pool) return Ada.Storage_Elements.Storage_Count;

  procedure Allocate
    (Pool              : in out Simple_Pool;
    Storage_Address    : out System.Address;
    Size_In_Storage_Elements : Ada.Storage_Elements.Storage_Count;
    Alignment          : Ada.Storage_Elements.Storage_Count) is
    pragma Unreferenced (Pool, Alignment);
```

```
  begin
    Storage_Address := System.Memory.Alloc (Integer (Size_In_Storage_Elements));
  end Allocate;

  procedure Deallocate
    (Pool              : in out Simple_Pool;
    Storage_Address    : System.Address;
    Size_In_Storage_Elements : Ada.Storage_Elements.Storage_Count;
    Alignment          : Ada.Storage_Elements.Storage_Count) is
    pragma Unreferenced
      (Pool, Storage_Address, Size_In_Storage_Elements, Alignment);
  begin
    null;
  end Deallocate;

  function Storage_Size
    (Pool : Simple_Pool) return Ada.Storage_Elements.Storage_Count is
    pragma Unreferenced (Pool);
  begin
    return 1024;
  end Storage_Size;

  Pool_A : Simple_Pool;

  type Int_Access is access Integer;
  for Int_Access'Storage_Pool use Pool_A;

  procedure Safe_Release (Ptr : in out Int_Access) is
  begin
    if Ptr /= null then
      Ptr := null;
    end if;
  end Safe_Release;

  Ptr : Int_Access := new Integer'(20);

begin
  -- Uso normal del puntero
  Put_Line ("Valor: " & Integer'Image (Ptr.all));

  -- Invalidación segura de la referencia
  Safe_Release (Ptr);

  Put_Line ("Referencia invalidada correctamente");
end Main;
```

2.5.7.1 Descripción

La gestión de memoria mediante pools de almacenamiento requiere que cada bloque de memoria se libere únicamente en el pool donde fue originalmente asignado. Esta restricción surge de la arquitectura interna de los gestores de memoria, que mantienen estructuras de datos específicas para cada pool y no pueden procesar correctamente bloques procedentes de otros espacios de almacenamiento.

Cuando se intenta liberar memoria en un pool diferente al de origen, el gestor de memoria puede acceder a metadatos incorrectos o inexistentes, lo que genera comportamientos impredecibles. Estos errores son particularmente problemáticos porque pueden manifestarse de forma intermitente y ser difíciles de reproducir durante las pruebas.

Los riesgos asociados incluyen **corrupción de las estructuras internas** del gestor de memoria, **terminación anómala del programa** y **comportamiento indefinido** que puede comprometer la estabilidad del sistema. En entornos críticos, estos fallos pueden tener consecuencias graves para la seguridad y disponibilidad del servicio.

La mitigación efectiva de estos problemas requiere establecer **convenciones claras de asignación y liberación** que asocien cada bloque de memoria con su pool correspondiente. Las herramientas de análisis estático pueden detectar violaciones de esta regla durante el desarrollo, mientras que los sistemas de depuración especializados permiten identificar errores durante la ejecución.

El diseño de la arquitectura de memoria debe considerar la **trazabilidad de los bloques asignados** y establecer mecanismos que garanticen la correspondencia entre las operaciones de asignación y liberación. Esta aproximación sistemática reduce significativamente la probabilidad de errores relacionados con la gestión incorrecta de pools de memoria.

Riesgos asociados

- Corrupción de memoria del programa.
- Condiciones de carrera y comportamiento indefinido.
- Fugas de memoria (memory leaks) por una incorrecta gestión del ciclo de vida.
- Violación de los invariantes del pool de memoria, generando inestabilidad.
- Posibilidad de doble liberación (double free) al perder el control de los punteros.
- Inclusión de vulnerabilidades explotables para escalado de privilegios o ejecución de código arbitrario.
- Dificultad en el seguimiento y depuración de errores relacionados con la memoria.
- Pérdida de integridad y confidencialidad en la información almacenada

dinámicamente.

- Comportamiento inesperado de la aplicación en diferentes plataformas o compiladores.

Buenas prácticas

- Utilizar el mismo `Storage_Pool` para la liberación que el empleado durante la asignación del objeto.

```
//-- Liberación usando un pool diferente al de la asignación
type Mi_Pool is new Root_Storage_Pool with null record;
for Mi_Tipo'Storage_Pool use Pool_A;
Obj : access Mi_Tipo := new Mi_Tipo;
```

```
//-- Liberación asociada al mismo pool configurado para el tipo
for Mi_Tipo'Storage_Pool use Pool_A;
Obj : access Mi_Tipo := new Mi_Tipo;
Free (Obj);
```

- Evitar conversiones de tipos `access` que puedan provocar liberación en un `pool` distinto.

```
//-- Conversión entre tipos access con distinto Storage_Pool
type A_Access is access all Mi_Tipo;
for A_Access'Storage_Pool use Pool_A;
type B_Access is access all Mi_Tipo;
for B_Access'Storage_Pool use Pool_B;
Obj_A : A_Access := new Mi_Tipo;
Obj_B : B_Access := B_Access (Obj_A);
Free (Obj_B);
```

```
//-- Uso consistente del mismo tipo access y mismo Storage_Pool
type A_Access is access all Mi_Tipo;
for A_Access'Storage_Pool use Pool_A;
Obj_A : A_Access := new Mi_Tipo;
Free (Obj_A);
```

- Encapsular la asignación y liberación en un mismo paquete para controlar el `pool` utilizado.

```
//-- Asignación en un módulo y liberación externa sin control del pool
Obj : access Mi_Tipo := Crear_Objeto;
Free (Obj);
```

```
//-- Encapsulación de creación y destrucción dentro del mismo paquete
```

```
Obj : access Mi_Tipo := Paquete_Seguro.Crear;
Paquete_Seguro.Liberar (Obj);
```

- No mezclar pools personalizados con el `pool` por defecto para un mismo tipo.

```
//-- Asignación con pool por defecto y liberación esperando pool personalizado
Obj : access Mi_Tipo := new Mi_Tipo;
for Mi_Tipo'Storage_Pool use Pool_A;
Free (Obj);
```

```
//-- Definición explícita del Storage_Pool antes de cualquier asignación
for Mi_Tipo'Storage_Pool use Pool_A;
Obj : access Mi_Tipo := new Mi_Tipo;
Free (Obj);
```

2.5.7.2 Estándares metodológicos

- **AdaCore RCL.03**: Only Reclaim to the Same Pool.

2.5.8 [ada.mem.8] - Evitar el uso de Unchecked_Deallocation

ID	ada.mem.8	Uso	Recomendado
CWSS Score	23	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

El uso de `Ada.Unchecked_Deallocation` introduce riesgos significativos de **gestión manual de memoria** que pueden comprometer la seguridad del programa. Esta operación permite liberar memoria de forma explícita, creando potenciales **referencias colgantes**, **doble liberación de memoria** y **accesos a memoria no válida** que el sistema de tipos no puede detectar en tiempo de compilación.

Codificación insegura

Gestión manual de memoria dinámica usando acceso y liberación explícita. Se reserva memoria para un entero, se libera manualmente y luego se intenta seguir utilizando el puntero. El objetivo es mostrar los riesgos asociados a la liberación no controlada de memoria y los posibles fallos en tiempo de ejecución.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Unchecked_Deallocation;

procedure Main is
  //-- Definición de un tipo acceso a Integer
  type Int_Ptr is access all Integer;
```

```
  //-- Procedimiento para liberar memoria sin comprobaciones
  procedure Free is new Ada.Unchecked_Deallocation (Integer, Int_Ptr);

  P : Int_Ptr;
begin
  //-- Reserva dinámica de memoria
  P := new Integer'(10);
  Put_Line("Valor inicial: " & Integer'Image(P.all));

  //-- Liberación manual de memoria
  Free(P);

  //-- Uso posterior del puntero liberado (riesgo de error)
  if P /= null then
    Put_Line("Valor despues de liberar: " & Integer'Image(P.all));
  end if;

end Main;
```

Codificación recomendada

Uso de variables locales y gestión automática de memoria sin liberación explícita. Se utiliza un objeto en el ámbito local y un acceso seguro sin recurrir a liberación manual. El objetivo es mostrar cómo evitar riesgos de corrupción o uso indebido de memoria delegando la gestión al ciclo de vida del objeto.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  //-- Variable local gestionada automáticamente
  Value : aliased Integer := 10;

  //-- Tipo acceso sin necesidad de liberación manual
  type Int_Ptr is access all Integer;
  P : Int_Ptr := Value'Access;
begin
  //-- Uso seguro del puntero mientras el objeto está en alcance
  Put_Line("Valor seguro: " & Integer'Image(P.all));

  //-- No se realiza liberación manual; la memoria se gestiona por alcance
  Put_Line("Fin del programa sin liberacion manual");

end Main;
```

2.5.8.1 Descripción

La desasignación manual de memoria mediante `Ada.Unchecked_Deallocation` introduce vulnerabilidades significativas en el manejo de memoria. Esta operación permite liberar explícitamente bloques de memoria asignados dinámicamente, pero su uso inadecuado puede generar **referencias colgantes**, **doble liberación de memoria** y **corrupción del heap**.

El procedimiento `Unchecked_Deallocation` no verifica automáticamente si existen otras referencias al objeto que se está liberando, ni si la memoria ya ha sido previamente desasignada. Esta falta de verificación puede provocar comportamientos indefinidos durante la ejecución del programa, especialmente cuando múltiples punteros referencian el mismo bloque de memoria.

Los riesgos asociados incluyen la posibilidad de acceder a memoria ya liberada, lo que puede resultar en **lecturas de datos corruptos** o **escrituras en ubicaciones de memoria no válidas**. Estos problemas son particularmente peligrosos en sistemas críticos donde la integridad de los datos y la estabilidad del programa son requisitos fundamentales.

La mitigación de estos riesgos se basa en el aprovechamiento de los **mecanismos automáticos de gestión de memoria** que proporciona el lenguaje. El sistema de tipos y el manejo automático del ciclo de vida de los objetos ofrecen alternativas más seguras para la gestión de recursos dinámicos.

Cuando la desasignación explícita sea inevitable, es necesario implementar **protocolos estrictos de gestión de punteros** que garanticen que no existan referencias múltiples al mismo objeto y que se establezcan mecanismos de verificación antes de proceder con la liberación de memoria.

Riesgos asociados

- Pérdida o corrupción de memoria por desasignación incontrolada de objetos.
- Referencias colgantes (dangling references) por uso de punteros a memoria ya liberada.
- Fugas de memoria si la recolección de recursos no se gestiona adecuadamente tras la liberación manual.
- Condiciones de carrera y comportamientos no deterministas en aplicaciones concurrentes si la memoria liberada sigue siendo accesible por otros hilos.
- Inconsistencias de estado en estructuras de datos dinámicas por liberación anticipada o incorrecta de memoria.
- Incremento de la superficie de ataque para explotación de vulnerabilidades como use-after-free.
- Dificultad para el análisis y la verificación formal del código debido a la gestión manual de memoria.

- Comportamientos inesperados y fallos en tiempo de ejecución, difíciles de detectar y depurar.

Buenas prácticas

- Utilizar tipos controlados (`Ada.Finalization.Controlled`) para gestionar automáticamente la liberación de recursos en lugar de recurrir a desasignación manual.

```
//-- Liberación manual de memoria con Ada.Unchecked_Deallocation
procedure Free is new Ada.Unchecked_Deallocation(Object => Integer, Name =>
Integer_Access);
Ptr : Integer_Access := new Integer'(10);
Free(Ptr);
```

```
//-- Liberación automática mediante tipo controlado
type Recurso is new Ada.Finalization.Controlled with record
  Data : Integer := 0;
end record;
overriding procedure Finalize (Obj : in out Recurso) is
begin
  null;
end Finalize;
```

- Preferir el uso de objetos en pila (stack) y evitar el uso innecesario de tipos `access` que requieran desasignación explícita.

```
//-- Uso de tipo access con liberación explícita
type Integer_Access is access all Integer;
Ptr : Integer_Access := new Integer'(25);
```

```
//-- Uso de variable local sin asignación dinámica
Value : Integer := 25;
Value := Value + 1;
```

- Emplear contenedores estándar de Ada (`Ada.Containers`) que gestionan internamente la memoria en lugar de estructuras dinámicas manuales.

```
//-- Lista enlazada manual con desasignación explícita
type Node;
type Node_Access is access Node;
type Node is record
  Value : Integer;
  Next  : Node_Access;
end record;
```

```
//-- Uso de contenedor estándar con gestión automática
package Int_Vectors is new Ada.Containers.Vectors(Index_Type => Natural,
Element_Type => Integer);
V : Int_Vectors.Vector;
V.Append(10);
```

2.5.8.2 Estándares metodológicos

- **AdaCore/MISRA 7.1**: Preventing Undefined Behavior in SPARK.
- **AdaCore RCL.02**: Only Reclaim Allocated Storage.

2.5.9 [ada.mem.9] - Evitar usar pragma Suppress

ID	ada.mem.9	Uso	Obligatorio
CWSS Score	29	Impacto técnico	Alto
Dimensión	Integridad	S-SDLC	Build

El uso de `pragma Suppress` desactiva verificaciones de tiempo de ejecución que detectan errores críticos como desbordamientos de arrays, violaciones de rangos y condiciones de acceso nulo. Esta práctica elimina las **barreras de seguridad** incorporadas en el lenguaje, permitiendo que errores pasen desapercibidos y generen comportamientos impredecibles o vulnerabilidades de seguridad en el sistema.

Codificación insegura

Uso inseguro de `pragma Suppress` que desactiva comprobaciones de rango y permite accesos fuera de límites.

```
pragma Suppress(All_Checks);
procedure Unsafe_Access is
   type Int_Array is array (1 .. 3) of Integer;
   A : Int_Array := (1 => 10, 2 => 20, 3 => 30);
   X : Integer;
begin
   X := A(10); -- Acceso fuera de rango sin comprobación
end Unsafe_Access;
```

Codificación recomendada

Ejemplo seguro en Ada evitando el uso de `pragma Suppress`.

```
procedure Safe_Array_Access is
  subtype Index_Type is Integer range 1 .. 5;
  Data : array(Index_Type) of Integer := (1 => 10, 2 => 20, 3 => 30, 4 => 40, 5 =>
50);
```

```
   I : Integer := 3;
   Value : Integer;
begin
   -- Comprobación de rango implícita sin suprimir verificaciones
   if I in Index_Type then
      Value := Data(I);
   else
      Value := 0;
   end if;
end Safe_Array_Access;
```

2.5.9.1 Descripción

El `pragma Suppress` permite desactivar verificaciones de tiempo de ejecución específicas que el compilador incluye automáticamente para detectar errores como desbordamientos aritméticos, accesos fuera de límites de arrays o violaciones de restricciones de tipos. Su uso elimina las comprobaciones de seguridad que proporcionan detección temprana de errores críticos.

La desactivación de estas verificaciones introduce **vulnerabilidades de seguridad** al permitir que condiciones erróneas pasen desapercibidas durante la ejecución. Los errores no detectados pueden provocar comportamientos impredecibles, corrupción de memoria, accesos no autorizados a datos o fallos del sistema que comprometen tanto la integridad como la disponibilidad de la aplicación.

El `pragma` afecta directamente a la **robustez del software** al eliminar mecanismos de protección diseñados para mantener la consistencia del estado del programa. Sin estas verificaciones, los errores de programación que normalmente se detectarían y manejarían adecuadamente pueden propagarse silenciosamente a través del sistema.

La **mitigación principal** consiste en mantener activas todas las verificaciones de tiempo de ejecución durante el desarrollo y las pruebas. En casos excepcionales donde el rendimiento sea crítico, se debe realizar un análisis exhaustivo de riesgos y aplicar verificaciones manuales equivalentes. Las optimizaciones de rendimiento deben buscarse mediante técnicas de programación eficiente antes que mediante la eliminación de controles de seguridad.

Los entornos de producción requieren **evaluación cuidadosa** antes de considerar cualquier supresión de verificaciones. Cuando sea absolutamente necesario, la desactivación debe limitarse al ámbito más reducido posible y documentarse detalladamente, incluyendo justificación técnica y medidas compensatorias implementadas.

Riesgos asociados

- Omisión de comprobaciones de errores en tiempo de ejecución, lo que puede causar fallos silenciosos y comportamientos inesperados.

- Posibilidad de explotar vulnerabilidades como desbordamientos de buffer, referencias nulas o accesos fuera de rango debido a la falta de comprobaciones automáticas.
- Dificultades para depurar y detectar fallos potenciales, ya que los errores no se informan explícitamente.
- Incremento en la probabilidad de comprometer la integridad y confiabilidad del sistema.
- Riesgo de introducir vulnerabilidades de seguridad que pueden ser explotadas por actores maliciosos, especialmente en sistemas críticos.
- Reducción de la robustez del software ante condiciones inesperadas o entradas maliciosas.
- Compromiso de la portabilidad del código, pudiendo variar el comporamiento entre compiladores o versiones si las comprobaciones suprimidas son implementadas de forma diferente.
- Dificultad para garantizar el cumplimiento de estándares de seguridad y codificación en entornos regulados.

Buenas prácticas

- Utilizar `pragma Suppress` solo de forma localizada y justificada, nunca de manera global.

```
//-- Desactiva todas las comprobaciones de desbordamiento en el ámbito global
pragma Suppress (Overflow_Check);
```

```
//-- Aplica la supresión únicamente en un bloque específico y documentado
declare
  pragma Suppress (Overflow_Check);
  X : Integer := A + B;
begin
  null;
end;
```

- Priorizar el uso de tipos con rango definido para evitar depender de la supresión de comprobaciones.

```
//-- Utiliza un tipo sin restricciones y desactiva las comprobaciones de rango
pragma Suppress (Range_Check);
Value : Integer;
Value := Input;
```

```
//-- Define un subtipo con rango explícito que fuerza validación automática
```

```
subtype Safe_Range is Integer range 0 .. 100;
Value : Safe_Range;
Value := Input;
```

- Realizar validaciones explícitas antes de operaciones críticas en lugar de suprimir comprobaciones.

```
//-- Suprime la comprobación de división por cero
pragma Suppress (Division_Check);
Result := A / B;
```

```
//-- Verifica previamente el divisor antes de realizar la operación
if B /= 0 then
  Result := A / B;
else
  raise Constraint_Error;
end if;
```

- Evitar el uso de `pragma Suppress` en código de producción crítico para la seguridad.

```
//-- Desactiva todas las comprobaciones en una unidad crítica
pragma Suppress (All_Checks);
Balance := Balance - Amount;
```

```
//-- Mantiene las comprobaciones activas y maneja posibles excepciones
begin
  Balance := Balance - Amount;
exception
  when Constraint_Error =>
    Handle_Error;
end;
```

2.5.9.2 Estándares metodológicos

- **AdaCore/MISRA 7.2**: Proof of Absence of Run-Time Errors in SPARK.
- **AdaCore EXU.04**: Prove Absence of Run-time Exceptions.

2.6 Concurrencia

2.6.1 [ada.sync.1] - Restringir el modelo de concurrencia a perfiles deterministas (Ravenscar o Jorvik)

ID	**ada.sync.1**	**Uso**	**Recomendado**
CWSS Score	**86**	**Impacto técnico**	**Muy alto**
Dimensión	**Integridad**	**S-SDLC**	**Build**

Los modelos de concurrencia no restringidos pueden introducir comportamientos **no deterministas** que comprometen la predictibilidad y verificabilidad del sistema. El uso de características avanzadas de concurrencia como `select` statements, `abort` statements o tareas dinámicas genera condiciones de carrera y estados impredecibles que dificultan el análisis temporal y la certificación en sistemas críticos.

Codificación insegura

Múltiples tareas que utilizan entradas selectivas, retrasos dinámicos y cambios implícitos en planificación. Se busca mostrar un modelo de concurrencia flexible, pero no determinista, donde el orden de ejecución y los tiempos pueden variar entre ejecuciones, dificultando el análisis temporal y la verificación en sistemas críticos.

```
with Ada.Text_IO; use Ada.Text_IO;

//-- Programa sin perfil determinista
procedure Main is

  //-- Declaración de una tarea con entrada
  task type Worker is
    entry Start;
  end Worker;

  task body Worker is
  begin
    //-- Espera no determinista de activación
    accept Start;

    //-- Retardo dependiente del planificador
    delay 0.5;

    Put_Line("Tarea ejecutada");
  end Worker;

  //-- Instancias de tareas
  W1, W2 : Worker;

begin
```

```
  //-- Activación de tareas en orden no determinista
  W1.Start;
  W2.Start;

  delay 1.0;

  Put_Line("Fin del programa");
end Main;
```

Codificación recomendada

Programa concurrente restringido mediante el perfil `Ravenscar`, eliminando características dinámicas como entradas o retrasos relativos no controlados. Se pretende mostrar un modelo determinista, apto para sistemas de tiempo real críticos, donde la concurrencia es estática y analizable.

```
pragma Profile (Ravenscar);

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Real_Time; use Ada.Real_Time;

//-- Programa con perfil Ravenscar
procedure Main is

  //-- Periodo fijo de ejecución
  Period : constant Time_Span := Milliseconds(500);

  //-- Tarea periódica simple
  task Worker;

  task body Worker is
    Next_Release : Time := Clock;
  begin
    loop
      //-- Espera hasta el siguiente instante planificado
      Next_Release := Next_Release + Period;
      delay until Next_Release;

      Put_Line("Tarea periódica ejecutada");
    end loop;
  end Worker;

begin
  //-- El hilo principal permanece activo
  delay 2.0;
  Put_Line("Fin del programa");
end Main;
```

2.6.1.1 Descripción

Los perfiles de concurrencia `Ravenscar` y `Jorvik` definen subconjuntos restringidos del modelo completo de concurrencia, eliminando construcciones que pueden introducir comportamientos no deterministas o difíciles de analizar. Estos perfiles establecen limitaciones específicas sobre el uso de tareas, objetos protegidos, y primitivas de sincronización.

El perfil `Ravenscar` constituye el más restrictivo de ambos, permitiendo únicamente patrones de concurrencia que garantizan **determinismo temporal** y facilitan el análisis estático. Prohíbe construcciones como `select` statements, `delay` statements dentro de objetos protegidos, y múltiples puntos de entrada en tareas. El perfil `Jorvik` extiende `Ravenscar` con funcionalidades adicionales manteniendo las garantías de determinismo.

La aplicación de estos perfiles reduce significativamente los riesgos asociados con **condiciones de carrera**, interbloqueos, y comportamientos impredecibles en sistemas concurrentes. Los perfiles eliminan fuentes comunes de no determinismo que pueden manifestarse como fallos intermitentes o comportamientos dependientes del timing del sistema.

La mitigación se logra mediante la adopción de uno de estos perfiles durante la fase de diseño del sistema, configurando el compilador para rechazar construcciones no permitidas. Esta aproximación requiere reestructurar algoritmos concurrentes para ajustarse a los patrones permitidos, pero proporciona **garantías formales** sobre el comportamiento temporal del sistema.

Los perfiles resultan especialmente apropiados para sistemas de tiempo real donde la predictibilidad temporal es un requisito crítico, y donde la verificación formal del comportamiento concurrente debe ser factible mediante herramientas de análisis estático.

Riesgos asociados

- Condiciones de carrera debido a la ejecución no determinista de tareas.
- Interbloqueos (deadlocks) por una mala gestión de prioridades y recursos compartidos.
- Dificultad para realizar pruebas y depuración por la imprevisibilidad en la planificación de tareas.
- Vulnerabilidades de seguridad por exposición de datos sensibles a través de estados compartidos no protegidos.
- Fallo en sistemas críticos en tiempo real debido a la imposibilidad de garantizar tiempos máximos de respuesta.
- Comportamientos inesperados en la aplicación por la interacción inesperada de múltiples hilos.

- Imposibilidad de cumplir requisitos de certificación para aplicaciones de misión crítica (ej. aeroespacial, ferroviario, médico).
- Incremento de la complejidad del código, dificultando el mantenimiento y la comprensión del sistema.
- Introducción de errores sutiles que pueden pasar desapercibidos en revisiones estáticas o pruebas funcionales.
- Exposición a ataques de denegación de servicio (DoS) internos por saturación de recursos concurrentes no controlados.

Buenas prácticas

- Configurar explícitamente el perfil `Ravenscar` o `Jorvik` en el proyecto para garantizar restricciones deterministas de concurrencia.

```
//-- Uso de tareas sin perfil determinista
task type Worker is
  entry Start;
end Worker;
```

```
//-- Activación explícita del perfil Ravenscar
pragma Profile (Ravenscar);
task type Worker is
  entry Start;
end Worker;
```

- Evitar construcciones no permitidas en perfiles deterministas, como `select` dinámicos o `delay` relativos no acotados.

```
//-- Uso de select con alternativas no deterministas
task body Worker is
begin
  select
    accept Start;
  or
    delay 10.0;
  end select;
end Worker;
```

```
//-- Sustitución por punto de entrada único y temporización controlada
pragma Profile (Ravenscar);
task body Worker is
begin
  accept Start;
end Worker;
```

- Utilizar objetos protegidos en lugar de variables compartidas para sincronización bajo perfiles deterministas.

```
//-- Variable global compartida sin protección
Shared_Data : Integer := 0;
task body Worker is
begin
  Shared_Data := Shared_Data + 1;
end Worker;
```

```
//-- Uso de objeto protegido compatible con Ravenscar
protected Shared_Object is
  procedure Increment;
private
  Value : Integer := 0;
end Shared_Object;
protected body Shared_Object is
  procedure Increment is
  begin
    Value := Value + 1;
  end Increment;
end Shared_Object;
```

- Definir prioridades fijas y políticas de planificación deterministas para todas las tareas concurrentes.

```
//-- Tarea sin prioridad explícita
task type Worker;
task body Worker is
begin
  null;
end Worker;
```

```
//-- Tarea con prioridad fija bajo política FIFO_Within_Priorities
pragma Profile (Ravenscar);
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
task type Worker;
pragma Priority (System.Priority'Last - 1);
task body Worker is
begin
  null;
end Worker;
```

2.6.1.2 Estándares metodológicos

- **AdaCore CON.01**: Use the Ravenscar Profile.

- **AdaCore CON.02**: Use the Jorvik Profile.

2.6.2 [ada.sync.2] - Prohibir el uso de variables compartidas no protegidas para comunicación entre tareas

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

La comunicación entre tareas mediante variables compartidas sin mecanismos de protección genera **condiciones de carrera** y comportamientos impredecibles. El acceso concurrente a datos compartidos puede provocar **corrupción de información**, estados inconsistentes y fallos intermitentes difíciles de detectar durante las pruebas, pero críticos en producción.

Codificación insegura

Concurrencia insegura por uso de variable global compartida sin sincronización.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  -- Variable global compartida sin protección
  Shared_Counter : Integer := 0;

  -- Tarea que incrementa el contador
  task type Increment_Task;

  task body Increment_Task is
  begin
    for I in 1 .. 1000 loop
      Shared_Counter := Shared_Counter + 1;
    end loop;
  exception
    when others => null;
  end Increment_Task;

  T1 : Increment_Task;
  T2 : Increment_Task;

begin
  null;
  Put_Line("Valor final del contador: " & Integer'Image(Shared_Counter));
end Main;
```

Codificación recomendada

Concurrencia segura usando un `protected` object.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  protected Safe_Counter is
    procedure Increment;
    function Get_Value return Integer;
  private
    Value : Integer := 0;
  end Safe_Counter;

  protected body Safe_Counter is
    procedure Increment is
    begin
      Value := Value + 1;
    end Increment;

    function Get_Value return Integer is
    begin
      return Value;
    end Get_Value;
  end Safe_Counter;

  task type Increment_Task;

  task body Increment_Task is
  begin
    for I in 1 .. 1000 loop
      Safe_Counter.Increment;
    end loop;
  exception
    when others => null;
  end Increment_Task;

  T1 : Increment_Task;
  T2 : Increment_Task;

begin
  null;
  Put_Line("Valor final del contador: " & Integer'Image(Safe_Counter.Get_Value));
end Main;
```

2.6.2.1 Descripción

Las variables compartidas entre tareas requieren mecanismos de sincronización explícitos para garantizar el acceso coherente a los datos. El acceso concurrente sin

protección a estas variables genera condiciones de carrera que comprometen la integridad de los datos y la predictibilidad del comportamiento del programa.

Cuando múltiples tareas acceden simultáneamente a una variable sin sincronización, las operaciones de lectura y escritura pueden intercalarse de forma impredecible. Esta situación produce **estados inconsistentes** donde el valor de la variable no refleja ninguna de las operaciones completas realizadas por las tareas concurrentes.

El problema se agrava en arquitecturas multiprocesador donde cada procesador mantiene cachés locales. Las modificaciones realizadas por una tarea pueden no ser inmediatamente visibles para otras tareas ejecutándose en procesadores diferentes, creando **inconsistencias de memoria** que persisten hasta que se fuerzan las sincronizaciones de caché.

Los riesgos asociados incluyen corrupción de datos, comportamiento no determinista del programa, y fallos intermitentes difíciles de reproducir y diagnosticar. Estos problemas se manifiestan especialmente en sistemas de tiempo real donde la predictibilidad temporal es crítica.

La mitigación efectiva requiere el uso de **objetos protegidos** que encapsulan las variables compartidas junto con las operaciones que las manipulan. Estos objetos garantizan acceso mutuamente exclusivo y proporcionan las barreras de memoria necesarias para mantener la coherencia entre procesadores.

Alternativamente, la comunicación entre tareas puede realizarse mediante **paso de mensajes** a través de tipos de entrada, eliminando completamente la necesidad de variables compartidas. Esta aproximación reduce la complejidad de sincronización y mejora la modularidad del diseño.

Para casos específicos donde el rendimiento es crítico, pueden emplearse variables marcadas como `Atomic` o `Volatile`, aunque estas opciones requieren un análisis cuidadoso de los requisitos de sincronización y las garantías proporcionadas por la implementación específica.

Riesgos asociados

- Condiciones de carrera (race conditions) que resultan en comportamiento no determinista.
- Corrupción de datos debido a accesos simultáneos no controlados.
- Dificultad para detectar y depurar errores relacionados con la concurrencia.
- Posibles bloqueos (deadlocks) o interbloqueos indeseados.
- Pérdida de integridad y consistencia en la información compartida.
- Exposición a vulnerabilidades explotables en sistemas críticos.
- Fallos de seguridad por manipulación indebida de datos.

- Resultados impredecibles en la lógica del sistema y mal funcionamiento general.
- Incremento en la complejidad del mantenimiento y la evolución del código.
- Fugas de información sensible a través de comportamientos inesperados.

Buenas prácticas

- Utilizar objetos protegidos para sincronizar el acceso a datos compartidos entre tareas.

```
//-- Variable compartida accedida directamente por múltiples tareas
Contador : Integer := 0;
//-- Tarea que incrementa sin sincronización
task body Tarea_A is
begin
  Contador := Contador + 1;
end Tarea_A;
```

```
//-- Objeto protegido que encapsula el acceso al recurso compartido
protected Contador_Seguro is
  procedure Incrementar;
  function Obtener return Integer;
private
  Valor : Integer := 0;
end Contador_Seguro;
protected body Contador_Seguro is
  procedure Incrementar is
  begin
    Valor := Valor + 1;
  end Incrementar;
  function Obtener return Integer is
  begin
    return Valor;
  end Obtener;
end Contador_Seguro;
```

- Emplear mecanismos de comunicación por entrada (`entry`) en lugar de variables globales compartidas.

```
//-- Variable global usada para pasar datos entre tareas
Dato_Compartido : Integer;
task body Productor is
begin
  Dato_Compartido := 10;
end Productor;
task body Consumidor is
begin
  Put_Line(Integer'Image(Dato_Compartido));
```

```
end Consumidor;
```

```
//-- Tarea con entry para transferencia controlada de datos
task Coordinador is
  entry Enviar(Dato : in Integer);
end Coordinador;
task body Coordinador is
  Valor : Integer;
begin
  accept Enviar(Dato : in Integer) do
    Valor := Dato;
  end Enviar;
  Put_Line(Integer'Image(Valor));
end Coordinador;
```

- Limitar el alcance de los datos compartidos y evitar variables globales visibles por múltiples unidades.

```
//-- Variable global visible en todo el paquete
package Datos is
  Estado : Boolean := False;
end Datos;
task body Tarea_Modifica is
begin
  Datos.Estado := True;
end Tarea_Modifica;
```

```
//-- Encapsulamiento del estado dentro de un objeto protegido en el paquete
package Datos_Seguros is
  protected Estado_Controlado is
    procedure Activar;
    function Esta_Activo return Boolean;
  private
    Estado : Boolean := False;
  end Estado_Controlado;
end Datos_Seguros;
```

- Evitar el uso de `pragma Atomic` o `Volatile` como sustituto de sincronización estructurada.

```
//-- Uso de pragma Atomic sin exclusión mutua
Bandera : Boolean;
pragma Atomic(Bandera);
task body Tarea_1 is
begin
  Bandera := True;
```

```
end Tarea_1;
```

```
//-- Uso de objeto protegido para gestionar la bandera
protected Bandera_Segura is
  procedure Establecer;
  function Consultar return Boolean;
private
  Valor : Boolean := False;
end Bandera_Segura;
protected body Bandera_Segura is
  procedure Establecer is
  begin
    Valor := True;
  end Establecer;
  function Consultar return Boolean is
  begin
    return Valor;
  end Consultar;
end Bandera_Segura;
```

2.6.2.2 Estándares metodológicos

- **AdaCore/MISRA 6.1**: Preventing Undefined Behavior.
- **AdaCore CON.03**: Avoid Shared Variables for Inter-task Communication.

2.6.3 [ada.sync.3] - Prohibir la recursión para evitar desbordamientos de pila y pérdida de control temporal

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

La recursión puede provocar **desbordamientos de pila** cuando las llamadas anidadas superan el espacio disponible en memoria. Este problema resulta especialmente crítico en sistemas con recursos limitados o requisitos de tiempo real, donde el agotamiento del stack puede causar fallos del sistema y comportamientos impredecibles.

Codificación insegura

Se calcula el factorial de un número utilizando una llamada recursiva. La intención es ilustrar cómo la recursión puede generar un consumo no controlado de pila cuando el valor de entrada crece, afectando la disponibilidad y el comportamiento temporal del sistema.

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Main is
  //-- Función recursiva para calcular el factorial
  function Factorial (N : Natural) return Natural is
  begin
    //-- Caso base de la recursión
    if N = 0 then
      return 1;
    else
      //-- Llamada recursiva que incrementa el uso de pila
      return N * Factorial (N - 1);
    end if;
  end Factorial;

  Result : Natural;
begin
  //-- Se usa un valor que puede provocar gran profundidad de llamadas
  Result := Factorial (20);
  Put_Line ("Resultado: " & Natural'Image (Result));
end Main;
```

Codificación recomendada

Se calcula el factorial de un número mediante una estructura iterativa. El objetivo es demostrar cómo evitar la recursión usando un bucle controlado, reduciendo el riesgo de desbordamiento de pila y mejorando la previsibilidad del consumo de recursos.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  //-- Función iterativa para calcular el factorial sin recursión
  function Factorial (N : Natural) return Natural is
    Result : Natural := 1;
  begin
    //-- Bucle que realiza el cálculo de forma controlada
    for I in 1 .. N loop
      Result := Result * I;
    end loop;
    return Result;
  end Factorial;

  Value  : Natural := 20;
  Result : Natural;
begin
  //-- Cálculo sin llamadas anidadas en la pila
  Result := Factorial (Value);
  Put_Line ("Resultado: " & Natural'Image (Result));
```

```
end Main;
```

2.6.3.1 Descripción

La recursión presenta riesgos significativos en sistemas donde el control de recursos y la predictibilidad temporal son requisitos críticos. El uso de funciones y procedimientos recursivos puede provocar un consumo descontrolado de memoria de pila, especialmente cuando la profundidad de recursión depende de datos de entrada variables o cuando no existen mecanismos adecuados de terminación.

Los **desbordamientos de pila** constituyen una amenaza directa para la estabilidad del sistema, ya que pueden causar fallos catastróficos en tiempo de ejecución. Estos fallos son particularmente problemáticos en sistemas embebidos o de tiempo real, donde la disponibilidad continua del servicio es esencial.

La recursión también introduce **impredecibilidad temporal** en la ejecución del programa. El tiempo de procesamiento y el consumo de memoria varían según la profundidad de recursión alcanzada, lo que dificulta el análisis estático de rendimiento y la garantía de cumplimiento de restricciones temporales.

En sistemas de **tiempo real**, esta impredecibilidad puede provocar el incumplimiento de plazos críticos, afectando la funcionalidad del sistema completo. La variabilidad en los tiempos de respuesta hace que sea prácticamente imposible certificar el comportamiento temporal del software.

La mitigación de estos riesgos se logra mediante la **eliminación completa de construcciones recursivas** del código fuente. Las soluciones alternativas incluyen la implementación de algoritmos iterativos equivalentes y el uso de estructuras de datos explícitas como pilas o colas para simular el comportamiento recursivo de manera controlada.

Los **algoritmos iterativos** ofrecen un consumo de memoria predecible y constante, facilitando el análisis estático de recursos. El uso de estructuras de datos auxiliares permite mantener un control preciso sobre el consumo de memoria y establecer límites máximos de procesamiento.

Riesgos asociados

- Desbordamiento de pila (stack overflow) por llamadas recursivas no controladas.
- Condiciones de denegación de servicio (DoS) cuando la recursión consume todos los recursos de ejecución.
- Pérdida de control sobre el tiempo de ejecución, dificultando la predictibilidad y el análisis temporal (importante en sistemas en tiempo real).
- Dificultad en la detección y recuperación de errores debido a cadenas profundas de llamadas recursivas.

- Aumento en la complejidad de pruebas y verificación formal del software debido a la naturaleza potencialmente infinita de las llamadas recursivas.
- Vulnerabilidad a ataques de sobrecarga intencionada si un atacante puede provocar una recursión excesiva (por ejemplo, a través de entradas maliciosas).
- Imposibilidad de cumplir con restricciones de recursos críticos (CPU, memoria) en sistemas embebidos o críticos.

Buenas prácticas

- Sustituir funciones recursivas por estructuras iterativas controladas.

```
//-- Cálculo recursivo sin control de profundidad
function Factorial (N : Natural) return Natural is
begin
  if N = 0 then
    return 1;
  else
    return N * Factorial(N - 1);
  end if;
end Factorial;
```

```
//-- Cálculo iterativo con control explícito
function Factorial (N : Natural) return Natural is
  Result : Natural := 1;
begin
  for I in 1 .. N loop
    Result := Result * I;
  end loop;
  return Result;
end Factorial;
```

- Reemplazar recorridos recursivos de estructuras por bucles explícitos con estructuras auxiliares controladas.

```
//-- Recorrido recursivo de lista enlazada
procedure Print_List (Node : List_Node_Access) is
begin
  if Node /= null then
    Put_Line(Node.Value);
    Print_List(Node.Next);
  end if;
end Print_List;
```

```
//-- Recorrido iterativo con control explícito del puntero
procedure Print_List (Node : List_Node_Access) is
  Current : List_Node_Access := Node;
```

```
begin
  while Current /= null loop
    Put_Line(Current.Value);
    Current := Current.Next;
  end loop;
end Print_List;
```

- Evitar recursión indirecta entre subprogramas sustituyéndola por una máquina de estados o bucle controlado.

```
//-- Llamadas recursivas indirectas sin límite definido
procedure A (N : Natural);
procedure B (N : Natural);
procedure A (N : Natural) is
begin
  if N > 0 then
    B(N - 1);
  end if;
end A;
procedure B (N : Natural) is
begin
  if N > 0 then
    A(N - 1);
  end if;
end B;
```

```
//-- Sustitución por bucle con estado explícito
procedure Process (N : Natural) is
  Counter : Natural := N;
begin
  while Counter > 0 loop
    Counter := Counter - 1;
  end loop;
end Process;
```

- Definir límites máximos explícitos en algoritmos potencialmente profundos para garantizar consumo de pila constante.

```
//-- Búsqueda recursiva sin control de profundidad
function Search (Arr : Int_Array; Index : Natural) return Boolean is
begin
  if Index > Arr'Last then
    return False;
  elsif Arr(Index) = 0 then
    return True;
  else
    return Search(Arr, Index + 1);
```

```
  end if;
end Search;
```

```
//-- Búsqueda iterativa con límite basado en rango del array
function Search (Arr : Int_Array) return Boolean is
begin
  for Index in Arr'Range loop
    if Arr(Index) = 0 then
      return True;
    end if;
  end loop;
  return False;
end Search;
```

2.6.3.2 Estándares metodológicos

- **AdaCore RPP.12**: No Recursion.

2.6.4 [ada.sync.4] - Implementar cancelación cooperante de tareas

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

La cancelación abrupta de tareas puede provocar **inconsistencias en el estado del programa** y **corrupción de datos compartidos**. Las tareas que terminan inesperadamente pueden dejar recursos sin liberar, estructuras de datos en estados intermedios o bloqueos activos que afecten a otras tareas del sistema.

Codificación insegura

Una tarea que realiza trabajo en bucle y es detenida mediante la sentencia `abort`. Se observa cómo la cancelación forzada interrumpe la ejecución sin permitir que la tarea libere recursos ni finalice de forma controlada, lo que puede generar estados inconsistentes y problemas de mantenimiento.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Cancelacion_Forzada is

  task Worker;

  task body Worker is
  begin
    //-- La tarea entra en un bucle infinito simulando trabajo
    loop
```

```
      Put_Line("Trabajando...");
      delay 1.0;
    end loop;
  end Worker;

begin
  //-- El programa principal espera unos segundos
  delay 3.0;

  //-- Se fuerza la cancelación de la tarea
  abort Worker;

  Put_Line("Tarea abortada");
end Cancelacion_Forzada;
```

Codificación recomendada

Una tarea que realiza trabajo en bucle y verifica periódicamente una condición de parada compartida mediante un objeto protegido. La tarea finaliza de forma controlada cuando se solicita la cancelación, permitiendo una terminación limpia y predecible.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Cancelacion_Cooperativa is

  protected Control is
    procedure Solicitar_Parada;
    function Parar return Boolean;
  private
    Detener : Boolean := False;
  end Control;

  protected body Control is
    procedure Solicitar_Parada is
    begin
      Detener := True;
    end Solicitar_Parada;

    function Parar return Boolean is
    begin
      return Detener;
    end Parar;
  end Control;

  task Worker;

  task body Worker is
  begin
```

```
    //-- La tarea comprueba periódicamente si debe finalizar
    loop
      exit when Control.Parar;
      Put_Line("Trabajando...");
      delay 1.0;
    end loop;

    //-- Finalización controlada de la tarea
    Put_Line("Tarea finalizada correctamente");
  end Worker;

begin
  //-- El programa principal espera unos segundos
  delay 3.0;

  //-- Se solicita la parada de forma cooperativa
  Control.Solicitar_Parada;

  delay 2.0;
  Put_Line("Programa finalizado");
end Cancelacion_Cooperativa;
```

2.6.4.1 Descripción

La cancelación cooperante de tareas requiere que las tareas verifiquen periódicamente si han sido marcadas para su finalización y respondan de manera controlada a estas solicitudes. Este mecanismo permite que las tareas finalicen su ejecución de forma ordenada, liberando recursos y completando operaciones críticas antes de terminar.

El modelo de cancelación cooperante se basa en que la tarea objetivo debe **verificar activamente** su estado de cancelación mediante llamadas explícitas a primitivas del sistema. A diferencia de la terminación forzosa, este enfoque otorga control a la tarea para decidir cuándo y cómo responder a la solicitud de cancelación.

La implementación inadecuada de este mecanismo puede generar **condiciones de carrera**, **bloqueos indefinidos** y **corrupción de datos compartidos**. Las tareas que no verifican su estado de cancelación con suficiente frecuencia pueden continuar ejecutándose indefinidamente, consumiendo recursos del sistema y potencialmente interfiriendo con otras operaciones.

Los riesgos se incrementan cuando las tareas manejan recursos compartidos o realizan operaciones de entrada/salida prolongadas. La falta de puntos de cancelación apropiados puede resultar en **recursos no liberados**, **archivos abiertos** o **conexiones de red activas** que persisten más allá del ciclo de vida esperado de la tarea.

La mitigación efectiva requiere la inserción de **puntos de cancelación regulares** en el código de la tarea, especialmente en bucles de larga duración y antes de operaciones bloqueantes. Las tareas deben implementar rutinas de limpieza que se ejecutan cuando

se detecta una solicitud de cancelación, asegurando que los recursos se liberan correctamente.

La sincronización adecuada entre la tarea que solicita la cancelación y la tarea objetivo es esencial para evitar condiciones de carrera. El uso de **variables de estado compartidas** protegidas por mecanismos de sincronización apropiados facilita la comunicación segura entre tareas durante el proceso de cancelación.

Riesgos asociados

- Bloqueo de recursos o deadlocks debido a tareas que nunca liberan recursos al no ser canceladas correctamente.
- Condiciones de carrera y corrupción de datos por finalizaciones abruptas o no controladas de tareas.
- Pérdida de datos o de estado intermedio si una tarea se cancela inesperadamente sin guardar la información necesaria.
- Consumo excesivo de recursos del sistema (memoria, CPU) por tareas huérfanas que siguen ejecutándose sin posibilidad de ser detenidas eficientemente.
- Mayor dificultad para el mantenimiento del código y la depuración, ya que los flujos de control pueden volverse impredecibles.
- Escalada de fallas: la cancelación abrupta puede dejar al sistema en un estado inconsistente, propagando errores a otras tareas o partes del programa.
- Vulnerabilidad a denegación de servicio (DoS) si tareas maliciosas o defectuosas impiden el correcto funcionamiento del sistema al no cooperar en su finalización.

Buenas prácticas

- Utilizar una variable compartida protegida para señalizar la cancelación en lugar de abortar la tarea abruptamente.

```
//-- Cancelación abrupta mediante abort
abort Worker_Task;
```

```
//-- Señalización cooperativa mediante objeto protegido
protected Cancel_Flag is
  procedure Request_Cancel;
  function Is_Cancelled return Boolean;
private
  Cancelled : Boolean := False;
end Cancel_Flag;
protected body Cancel_Flag is
  procedure Request_Cancel is
  begin
    Cancelled := True;
```

```
  end Request_Cancel;
  function Is_Cancelled return Boolean is
  begin
    return Cancelled;
  end Is_Cancelled;
end Cancel_Flag;
```

- Comprobar periódicamente la condición de cancelación dentro del bucle principal de la tarea.

```
//-- Bucle sin verificación de cancelación
task body Worker_Task is
  begin
    loop
      Process_Data;
    end loop;
end Worker_Task;
```

```
//-- Bucle con verificación periódica de cancelación
task body Worker_Task is
  begin
    loop
      exit when Cancel_Flag.Is_Cancelled;
      Process_Data;
    end loop;
end Worker_Task;
```

- Realizar limpieza controlada de recursos antes de finalizar la tarea tras recibir la señal de cancelación.

```
//-- Finalización inmediata sin liberar recursos
if Cancel_Flag.Is_Cancelled then
  return;
end if;
```

```
//-- Liberación explícita de recursos antes de salir
if Cancel_Flag.Is_Cancelled then
  Close_File (Data_File);
  Release_Lock (Shared_Resource);
  return;
end if;
```

- Utilizar entradas selectivas con alternativa `terminate` para permitir finalización ordenada cuando no haya más trabajo.

```
//-- Tarea bloqueada indefinidamente en accept
```

```
task body Server_Task is
  begin
    accept Request (Data : in Integer) do
      Handle (Data);
    end Request;
end Server_Task;
```

```
//-- Uso de select con alternativa terminate
task body Server_Task is
  begin
    loop
      select
        accept Request (Data : in Integer) do
          Handle (Data);
        end Request;
      or
        terminate;
      end select;
    end loop;
end Server_Task;
```

2.6.4.2 Estándares metodológicos

- **AdaCore CON.01**: Use the Ravenscar Profile.
- **AdaCore CON.03**: Avoid Shared Variables for Inter-task Communication.

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