Conceptos de Paradigmas y Lenguajes de Programación

Resumen teorías - EMT 2

Clase 4 - Unidades de programa

Unidades

Los lenguajes de programación permiten que un programa esté compuesto por unidades

Una unidad es la abstracción de una acción. En general se las llama rutinas (procedimientos/funciones)

Nombre

String de caracteres que se usa para invocar a la rutina (identificador)

El nombre de la rutina se introduce en su declaración y es lo que se usa para invocarlas

Alcance

Rango de instrucciones donde se conoce su nombre

El alcance se extiende desde el punto de su declaración hasta algún constructor de cierre

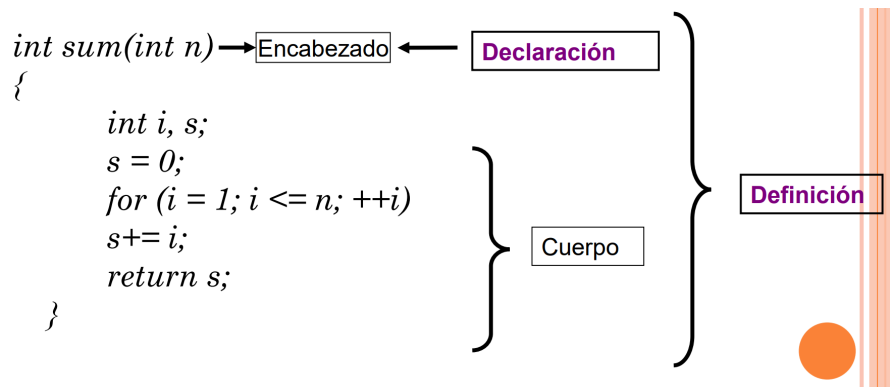
Según el lenguaje puede ser estático o dinámico

Activación: la llamada sólo puede estar dentro del alcance de la rutina

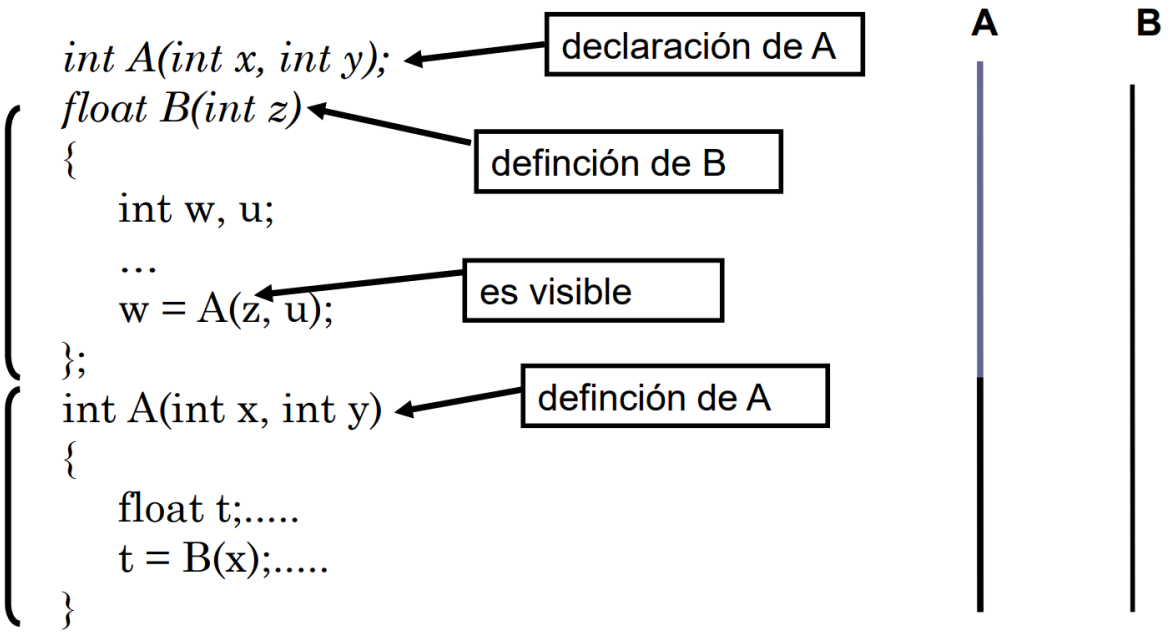
Definición vs declaración

Algunos lenguajes (C, C++, Ada, etc) hacen distinción entre definición y declaración de rutinas

Definición = declaración/encabezado + cuerpo



La separación de la declaración de la definición implica que se pueden hacer rutinas mutuamente recursivas:



Tipo

El encabezado de la rutina define el tipo de los parámetros y el tipo del valor de retorno (si lo hay)

Signatura: permite especificar el tipo de una rutina Una rutina fun que tiene como entrada parámetros de tipo T1, T2, Tn y devuelve un valor de tipo R, puede especificarse con la siguiente signatura fun: T1xT2x....Tn → R

Un llamado a una rutina es correcto si está de acuerdo al tipo de la rutina

La conformidad requiere la correspondencia de tipos entre parámetros formales y reales

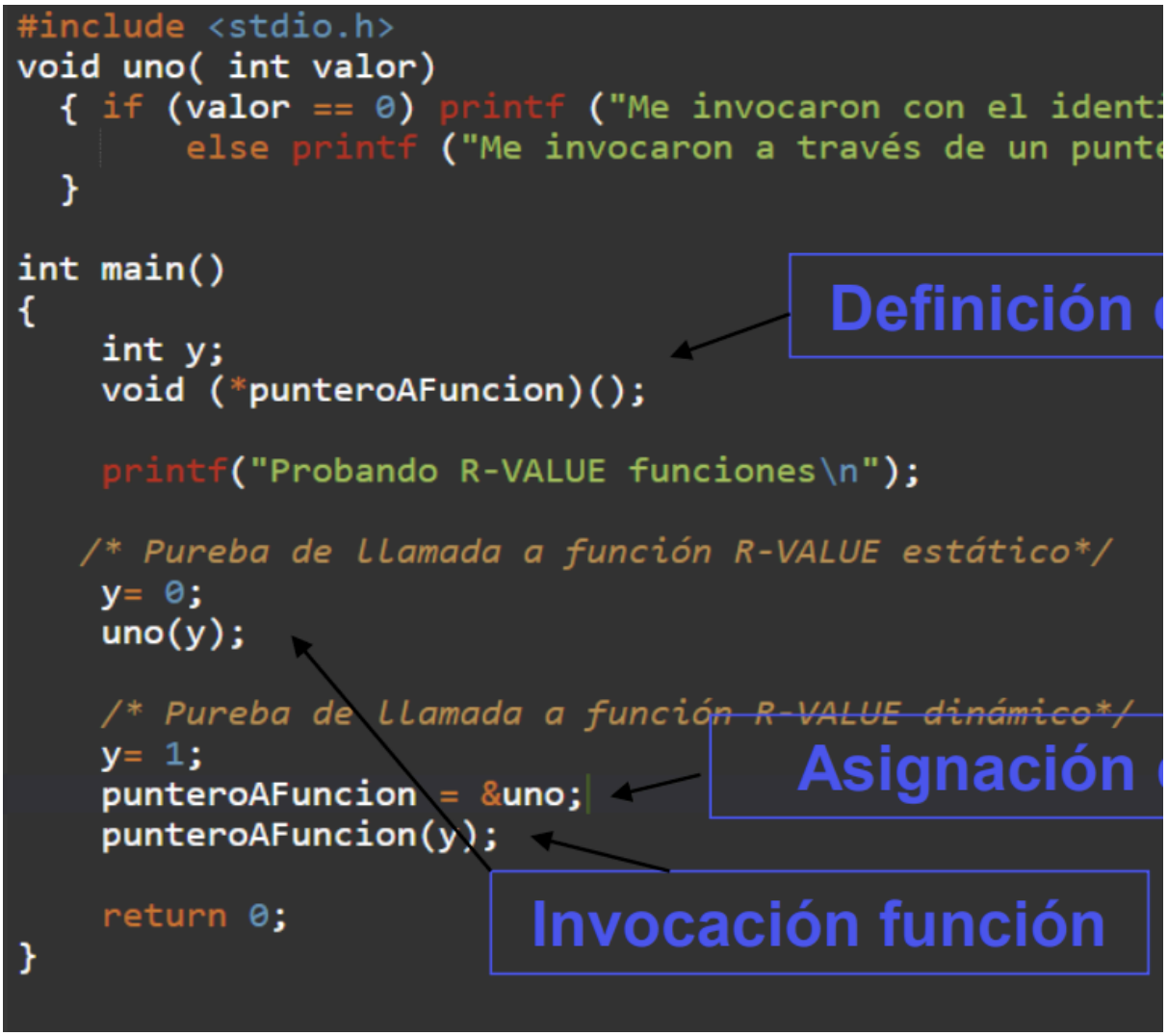
L-valor

Es el lugar de memoria en el que se almacena el cuerpo de la rutina

R-valor

La llamada a la rutina causa la ejecución de su código, eso constituye su r-valor

1. Estático: el caso más usual
2. Dinámica: variables de tipo rutina (se implementan a través de punteros a rutinas)



Comunicación entre rutinas

Parámetros formales: los que aparecen en la definición de la rutina

Parámetros reales: los que aparecen en la invocación de la rutina (dato o rutina)

Ligadura entre parámetros formales y reales

Método posicional

Se ligan uno a uno:

Rutina S (F1,F2,……,Fn) → Definición

S(A1, A2,….. An) → Llamado

Fi se liga a Ai para i de 1 a n. Deben conocerse las posiciones

Valores por defecto

C++: int distancia (int a = 0, int b = 0)

- distancia() → distancia(0, 0)
- distancia(10) → distancia(10, 0)
- distancia(10, 1) → distancia(10, 1)

Método por nombre

Se ligan por el nombre. Deben conocerse los nombres de los parámetros formales

Ada: procedure Ejem (A:T1; B: T2:= W; C:T3);

Si X, Y y Z son de tipo T1, T2 y T3

- Ejem (X,Y,Z) → asociación posicional
- Ejem (X, C => Z) → X se liga a A por posición, B toma el valor por defecto W C se liga a Z por nombre
- Ejem (C =>Z, A=>X, B=>Y) → se ligan todos por nombre

Representación en ejecución

La definición de la rutina especifica un proceso de cómputo

Cuando se invoca una rutina se ejecuta una instancia del proceso con los particulares valores de los parámetros

Instancia de la unidad: es la representación de la rutina en ejecución. Tiene dos zonas:

- Segmento de código: instrucciones de la unidad se almacenan en la memoria de instrucción (código: Contenido fijo). Siempre es el mismo en una misma ejecución del programa
- Registro de activación: datos locales de la unidad se almacenan en la memoria de datos (datos: Contenido cambiante). Se crea uno, por cada llamada a la rutina (con los datos particulares de esa llamada)

SIMPLESEM

- Memoria de Código: C(y) valor almacenado en la y-ésima celda de la memoria de código. Comienza en cero
- Memoria de Datos: D(y) valor almacenado en la y-ésima celda de la memoria de datos. Comienza en cero y representa el l-valor, D(y) o C(y) su r-valor
- Ip: puntero a la instrucción que se está ejecutando:
    - Se inicializa en cero en cada ejecución y se actualiza cuando se ejecuta cada instrucción
    - Direcciones de C

La ejecución de una rutina consiste en:

1. Obtener la instrucción actual para ser ejecutada (C[ip])
2. Incrementar ip
3. Ejecutar la instrucción actual
4. Repetir

Instrucciones

SET

Setea valores en la memoria de datos

set target, source: copia el valor representado por source en la dirección representada por target

set 10, D[20]: copia el valor almacenado en la posición 20 en la posición 10

E/S

read y write permiten la comunicación con el exterior

set 15, read el valor leído se almacena en la dirección 15

set write, D[50] se transfiere el valor almacenado en la posición 50

JUMP

Bifurcación incondicional

jump 47: la próxima instrucción a ejecutarse será la que esté almacenada en la dirección 47 de C

Direccionamiento indirecto:

jump D[30]

El ip va a a pasar a tener el valor que estaba en la posición 30 de la memoria de datos

JUMPT

Bifurcación condicional, bifurca si la expresión se evalúa como verdadera

jumpt 47, D[13]>D[8]: bifurca si el valor almacenado en la celda 13 es mayor que el almacenado en la celda 8

Punto de retorno

Es una pieza cambiante de información que debe ser salvada en el registro de activación de la unidad llamada

Ambiente de referencia

- Ambiente local: variables locales, ligadas a los objetos almacenados en su registro de activación
- Ambiente no local: variables no locales, ligadas a objetos almacenados en los registros de activación de otras unidades

Estructura de ejecución de lenguajes de programación

Estático

- Espacio fijo

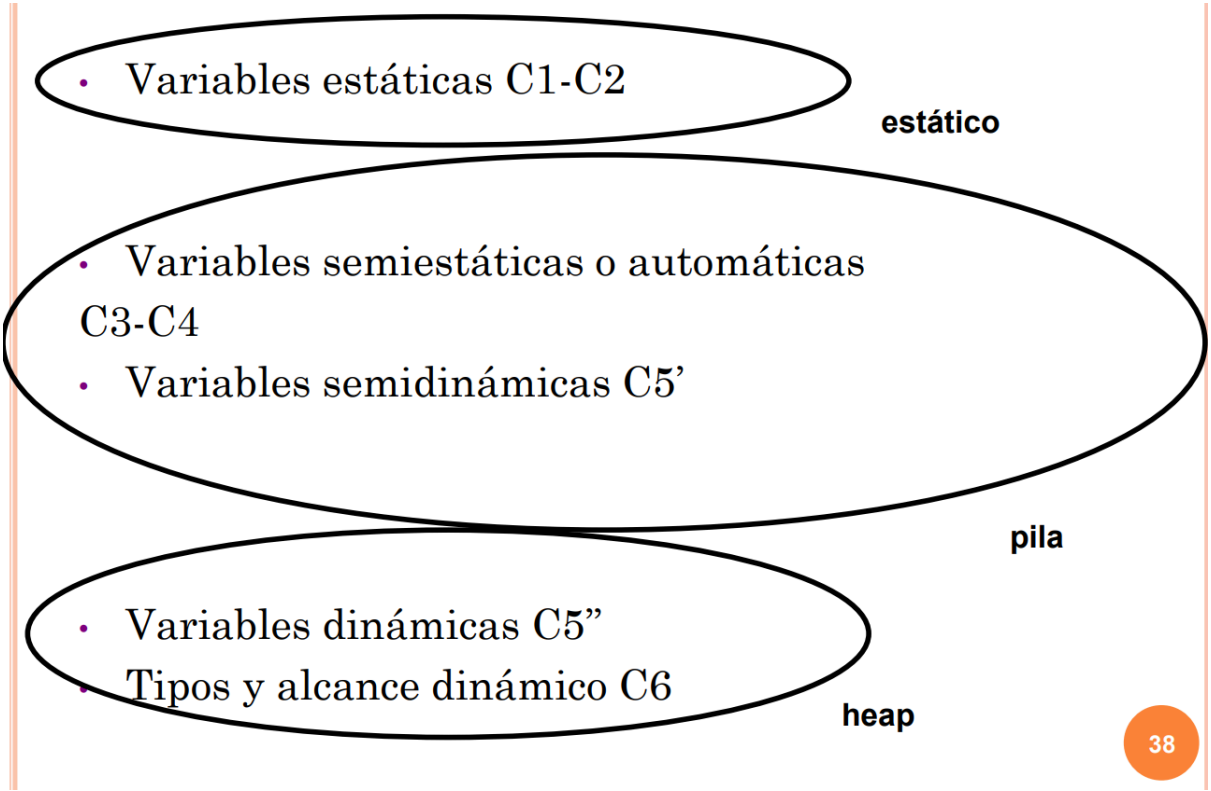
- El espacio necesario para la ejecución se deduce del código
- Todo los requerimientos de memoria necesarios se conocen antes de la ejecución
- La alocación puede hacerse estáticamente
- No puede haber recursión
- Todas las variables van a ser estáticas (en cuanto a tiempo de vida)

Basado en pila

- Espacio predecible (el registro de activación de cada unidad será de tamaño fijo y conocido, pero no se sabrá cuántas instancias de cada unidad se necesitarán durante la ejecución)
- El espacio se deduce del código (Algol-60, Pascal, Java)
- Programas más potentes cuyos requerimientos de memoria no pueden calcularse en traducción
- La memoria a utilizarse es predecible y sigue una disciplina last-in-first-out
- Las variables se alocan automáticamente y se desalocan cuando el alcance se termina
- Se utiliza una estructura de pila para modelizarlo
- Una pila no es parte de la semántica del lenguaje, es parte de nuestro modelo semántico

Dinámico

- Espacio impredecible
- Lenguajes con impredecible uso de memoria (Ruby, php, Python)
- Los datos son alocados dinámicamente sólo cuando se los necesita durante la ejecución
- No pueden modelizarse con una pila, el programador puede crear objetos de datos en cualquier punto arbitrario durante la ejecución del programa
- Los datos se alocan en la zona de memoria heap


Variables estáticas C1-C2
estático
Variables semiestáticas o automáticas
C3-C4
Variables semidinámicas C5'
pila
Variables dinámicas C5"
Tipos y alcance dinámico C6
heap
38

Clase 5 - Esquemas de ejecución

C1

- Sentencias simples
- Tipos simples:
    - Enteros
    - Reales
    - Arreglos
    - Estructuras
- Sin funciones
- Datos estáticos de tamaño fijo
- un programa = una rutina main()
    - Declaraciones
    - Sentencias
- E/S: read/write

Para este tipo de esquema se necesitan los siguientes elementos de ejecución:

- **IP**

Cada unidad de programa, en su registro de activación, va a tener solo sus datos locales

C2

- Definición de rutinas internas al main
- Programa:
    - Datos globales
    - Declaraciones de rutinas
    - Rutina principal
        - Datos locales
        - Se invoca automáticamente en ejecución
- Rutinas internas:
    - Disjuntas: no pueden estar anidadas
    - No son recursivas

    - Ambiente de las rutinas internas:
        - Datos locales
        - Datos globales

Para este tipo de esquema se necesitan los siguientes elementos de ejecución:

- IP
- **Punto de retorno**

Cada unidad de programa, en su registro de activación, va a tener sus datos locales y el punto de retorno

C3

- Esquema basado en pila
- Rutinas con capacidad de llamarse a sí mismas (recursión directa) o de llamar a otra rutina en forma recursiva (recursión indirecta)
- Rutinas con la capacidad de devolver valores, es decir, funciones

El registro de activación de cada unidad será de tamaño fijo y conocido, pero no se sabrá cuántas instancias de cada unidad se necesitarán durante la ejecución

Igual que en C2 el compilador puede ligar cada variable con su desplazamiento dentro del correspondiente registro de activación. El desplazamiento es estático

La dirección donde se cargará el registro de activación, es dinámica, por lo tanto, la ligadura con la direcciones absolutas en la zona de datos de la memoria, solo puede hacerse en ejecución

Cada nueva invocación aloca un nuevo registro de activación y se establecen las nuevas ligaduras entre el segmento de código y el nuevo registro de activación

Para este tipo de esquema se necesitan los siguientes elementos de ejecución:

- IP
- **Current**
- **Free**
- Punto de retorno
- **Valor de retorno**
- **Link dinámico**

Cada unidad de programa, en su registro de activación, va a tener sus datos locales, el punto de retorno, el valor de retorno y el link dinámico

C4

- Esquema basado en pila
- Controla el alcance de las variables
- Define el tiempo de vida de las variables
- Divide el programa en unidades más pequeñas
- Los bloques pueden ser:
    - Disjuntos (no tiene porción común)
    - Anidados (un bloque está completamente contenido en otro
- Puede haber sentencias compuestas, que contengan bloques

C4''

- Permite rutinas anidadas

Para este tipo de esquema se necesitan los siguientes elementos de ejecución:

- IP
- Current
- Free
- Punto de retorno
- Valor de retorno
- Link dinámico
- **Link estático**

Cada unidad de programa, en su registro de activación, va a tener sus datos locales, el punto de retorno, el valor de retorno, el link dinámico y el link estático

C5'

- Registro de activación cuyo tamaño se conoce cuando se activa la unidad
- Datos semidinámicos

type VECTOR is array (INTEGER range <>);

Define un arreglo con índice irrestricto:

- A: VECTOR (0..N);
- B: VECTOR(1..M);

N y M deben ligarse a algún valor entero para que A y B puedan alocarse en ejecución (referencia al ambiente no local o parámetros)

Compilación: se reserva lugar en el registro de activación para los descriptores de los arreglos dinámicos

Todos los accesos al arreglo dinámico son traducidos como referencias indirectas a través del puntero en el descriptor, cuyo desplazamiento se determina estáticamente

C5''

- Los datos pueden alocarse durante la ejecución
- Datos dinámicos: se alocan explícitamente durante la ejecución mediante instrucciones de alocación
- Los datos dinámicos se alocan en la heap

C6

- Lenguajes dinámicos
- Se trata de aquellos lenguajes que adoptan más reglas dinámicas que estáticas
- Usan tipado dinámico y reglas de alcance dinámicas
- Se podrían tener reglas de tipado dinámicas y de alcance estático, pero en la práctica las propiedades dinámicas se adoptan juntas
- Una propiedad dinámica significa que las ligaduras correspondientes se llevan a cabo en ejecución y no en compilación

Clase 6 - Parámetros

Rutinas

- También llamadas subprogramas
- Son una unidad de programa (función, procedimiento)
- Están formadas por un conjunto de sentencias que representan una acción abstracta
- Permiten al programador definir una nueva operación a semejanza de las operaciones primarias ya integradas en el lenguaje
- Permiten ampliar a los lenguajes, dan modularidad, claridad y buen diseño
- Se lanzan con una llamada explícita (se invocan por su nombre) y luego retornan a algún punto de la ejecución (responden al esquema call/return)
- Los subprogramas son el ejemplo más usual y útil presente desde los primeros lenguajes ensambladores

  Tienen dos formas:

  1. Procedimientos (ejecutan una acción)
  2. Funciones (devuelven un valor)

Procedimientos

- Un procedimiento es una construcción que permite dar nombre a un conjunto de sentencias y declaraciones asociadas que se usarán para resolver un subproblema dado
- Brindará una solución de código más corta, comprensible y fácilmente modificable
- Permiten al programador definir y crear nuevas acciones/sentencias
- El programador las invocará
- Pueden no recibir ni devolver ningún valor
- Los resultados los produce en variables no locales o en parámetros que cambian su valor

Funciones

- Mientras que un procedimiento ejecuta un grupo de sentencias, una función además devuelve un valor al punto donde se llamó

- El valor que recibe la función se usa para calcular el valor total de la expresión y devolver algún valor
- Permite al programador crear nuevas operaciones
- Similar a las funciones matemáticas ya que hacen algo y luego devuelven un valor y no producen efectos colaterales
- Se las invoca dentro de expresiones y lo que calcula reemplaza a la invocación dentro de la expresión
- Siempre deben retornar un valor

Conclusiones

- Cuando se diseña un subprograma el programador se concentra en el cómo trabajará dicho subprograma
- Cuando se usa un subprograma se ignorará el cómo. Sólo interesará el qué me permite hacer (la implementación permanece oculta (abstracción))
- Con una sola definición se pueden crear muchas activaciones. La definición de un subprograma es un patrón para crear activaciones durante la ejecución
- Un subprograma es la implementación de una acción abstracta y su invocación representa el uso de dicha abstracción
- Codificar un subprograma es como si hubiéramos incorporado una nueva sentencia a nuestro lenguaje

Formas de comunicar rutinas

Si utilizan variables locales no hay problema

Si no utilizan variables locales hay 2 formas

Acceso al ambiente no local

Se comparten variables que son de otra unidad con un acceso no local, no es muy claro, y es bastante propenso a errores

Se usa cadena estática o dinámica

Ambiente no local implícito

- Es automático
- Utiliza alguna de las 2 reglas:
    - Regla de alcance dinámico (quién me llamó y busco el identificador)
    - Regla de alcance estático (dónde está contenido y busco el identificador)

Ambiente común explícito

- Permite definir áreas comunes de código
- El programador debe especificar que es lo comparte
- Cada lenguaje tiene su forma de realizarlo

A través de parámetros

- Parámetro Real (Argumento):
    - Es un valor u otra entidad que se pasa a un procedimiento o función
    - Están colocados en la parte de la invocación de la rutina
- Parámetro Formal (Parámetro):
    - Es una variable utilizada para recibir valores de entrada en una rutina, subrutina, etc.
    - Están colocados en la parte de la declaración de la rutina

El pasaje de parámetros es mejor, ya que el uso intensivo de accesos al ambiente no local puede provocar alguna pérdida de control, y provocar que las variables terminen siendo visibles donde no es necesario y llevar a errores

Ventajas

- Brinda distintas posibilidades de compartir cosas (que veremos más adelante)
- Permite enviar distintos parámetros en distintas invocaciones a las rutinas
- Más flexibilidad, se pueden transferir más datos y de diferente tipo en cada llamada
- Permite compartir en forma más abstracta sólo especificamos el nombre y tipo a argumentos y parámetros

- Protección: el uso intensivo de accesos al ambiente no local decrementa la seguridad de las soluciones ya que las variables terminan siendo visibles aun donde no es necesario o donde no debería
- Legilibilidad:
    - Permite al programador encontrar más fácilmente los errores. (Ej. Si transformo en rutinas o funciones a los que les paso valores, es más fácil depurar y encontrar errores y no chequear cada repetición en el código). Esto le da modificabilidad, si hay errores uno se focaliza en qué cosas estoy compartiendo, qué argumentos y parámetros estoy utilizando y su tipo

Parámetros

Los parámetros formales son variables locales a su entorno

Se declaran con una sintaxis particular a cada lenguaje

Sirven para intercambiar información entre la función/rutina que hace la llamada y la que la recibe

Momento de vinculación de parámetros reales y formales

Comprende la evaluación de los parámetros reales y la ligadura con los parámetros formales

- Evaluación:
    - En general antes de la invocación primero se evalúan los parámetros reales, y luego se hace la ligadura
    - Se verifica que todo esté bien antes de transferir el control a la unidad llamada
- Ligadura:
    - Por posición:
        - Se corresponden con la posición que ocupan en la lista de parámetros
        - Van en el mismo orden
    - Por nombre o palabra clave: se corresponden con el nombre por lo tanto pueden estar colocados en distinto orden en la lista de parámetros

Datos como parámetros

In

El parámetro formal recibe el dato desde el parámetro real
La conexión es al inicio, se copia y se corta la vinculación

Por valor

El valor del parámetro real se usa para inicializar el correspondiente parámetro formal al invocar la unidad

Se transfiere el dato real y se copia

En este caso el parámetro formal actúa como una variable local de la unidad llamada, y crea otra variable

La conexión es al inicio para pasar el valor y se corta la vinculación

Es el mecanismo por default y el más usado

Desventajas:

- Consume tiempo para hacer la copia de cada parámetro
- Consume almacenamiento para duplicar cada dato (pensar grandes volúmenes)

Ventajas:

- Protege los datos de la unidad llamadora, el parámetro real no se modifica
- No hay efectos negativos o colaterales

Por valor constante

Se envía un valor, pero la rutina receptora no puede modificarlo, es decir queda con un valor fijo que no se puede cambiar

No indica si se realiza o no la copia (dependerá del lenguaje)

La implementación debe verificar que el parámetro real no sea modificado

No todos los lenguajes permiten el modo IN con pasaje por valor constante

Desventajas:

- Requiere realizar más trabajo para implementar los controles

Ventajas:

- Protege los datos de la unidad llamadora
- El parámetro real no se modifica

Out

Se invoca la rutina y cuando esta termina devuelve el parámetro formal al parámetro real

La conexión es al final

Por resultado

El valor del parámetro formal de la rutina se copia al parámetro real al terminar de ejecutarse la unidad que fue llamada

El parámetro formal es una variable local del entorno de la rutina

El parámetro formal es una variable sin valor inicial porque no recibe nada

Desventajas:

- Consume tiempo y espacio porque hace copia al final
- Debemos inicializar la variable en la unidad llamada de alguna forma (si el lenguaje no lo hace por defecto)

Ventajas:

- Protege los datos de la unidad llamadora, el parámetro real no se modifica en la ejecución de la unidad llamada

Por resultado de funciones

Es el resultado que me devuelven las funciones

Reemplaza la invocación en la expresión que contiene el llamado

Puede devolverse de distintas formas según lenguaje:

- Return como en Python, C, etc.
- Nombre de la función (último valor asignado) que se considera como una variable local (como en Pascal)

In/Out

El parámetro formal recibe el dato del parámetro real y el parámetro formal le envía el dato al parámetro real al finalizar la rutina

La conexión es al inicio y al final

Valor/Resultado

El parámetro formal es una variable local que recibe una copia a la entrada del contenido del parámetro real y a la salida el parámetro real recibe una copia de lo que tiene el parámetro formal

Cuando se invoca la rutina, el parámetro real le da valor al parámetro formal (se genera copia) y se desliga en ese momento

La rutina trabaja sobre ese parámetro formal pero no afecta al parámetro real porque trabaja sobre su copia

Cada referencia al parámetro formal es una referencia local

Una vez que termina de ejecutar el parámetro formal le devuelve un valor al parámetro real y lo copia

Se dice que hay una ligadura y una conexión entre parámetro real y el formal cuando se inicia la ejecución de la rutina y cuando se termina, pero no en el medio

Tiene las desventajas y las ventajas de ambos:

Desventajas:

- Consume tiempo para hacer la copia de cada parámetro
- Consume almacenamiento para duplicar cada dato (pensar grandes volúmenes)
- Consume tiempo y espacio porque hace copia al final

Ventajas:

- Protege los datos de la unidad llamadora, el parámetro real no se modifica
- No hay efectos negativos o colaterales

Por referencia

También llamada por "variable"

No es copia por valor es por referencia

Se asocia la dirección (l-valor) del PR al PF

La conexión es al inicio y permanece hasta el final

El PF será una variable local a su entorno que contiene la dirección al PR de la unidad llamadora que estará entonces en un ambiente no local. Así se extiende el alcance de la rutina (aliasing situation)

Cada referencia al PF será a un ambiente no local, entonces cualquier cambio que se realice en el PF dentro del cuerpo del subprograma quedará registrado en el PR. El cambio será automático

El PR queda compartido por la unidad llamadora y llamada. Será bidireccional

Desventajas:

- Se puede llegar a modificar el PR inadvertidamente. Es el peor problema
- El acceso al dato es más lento por la indirección a resolver cada vez que se invoque
- Se pueden generar alias cuando dos variables o referencias diferentes se asignen a la misma dirección de memoria
- Estas cosas afectan la legibilidad y por lo tanto la confiabilidad, se hace muy difícil la verificación de programas y depuración de errores

Ventajas:

- Ya que no se realizan copias de los datos: será eficiente en espacio y tiempo sobre todo en grandes volúmenes de datos
- La indirección es de bajo costo de implementación por muchas arquitecturas

Por nombre

No se verá en detalle en la materia

El parámetro formal es sustituido "textualmente" por una expresión del parámetro real, más un puntero al entorno del parámetro real (expresión textual, entorno real)

Se utiliza una estructura aparte que resuelve esto

Se establece la ligadura entre parámetro formal y parámetro real en el momento de la invocación, pero la "ligadura de valor" se difiere hasta el momento en que se lo utiliza (la dirección se resuelve en ejecución)

Distinto semánticamente a por referencia

El objetivo es otorgar evolución de valor diferida

No es de los más utilizados por los lenguajes

Thunks: es una unidad pequeña de código (función) que encapsula y representa a una expresión que pospone su evaluación hasta que sea necesario. Es un concepto clave en la evaluación perezosa/diferida

Para implementar el pasaje por nombre se utilizan:

- Los thunks que son procedimientos sin nombre
- Cada aparición del parámetro formal se reemplaza en el cuerpo de la unidad llamada por una invocación a un thunks
- En el momento de la ejecución activará al procedimiento que evaluará el parámetro real en el ambiente apropiado

Desventajas:

- Es un método que extiende el alcance del parámetro real, pero esto mismo puede llevar a errores
- Posee evaluación diferida al ejecutar
- Es más lento ya que debe evaluarse cada vez que se lo usa (ej.: si es un loop se evalúa cada vez)
- Es difícil de implementar y genera soluciones confusas para el lector y el escritor

Subprogramas como parámetros

En algunas situaciones es conveniente o necesario poder usar nombres de subprogramas como parámetro para ejecutar alguna acción

En general se usa con funciones matemáticas

No lo incorporan todos los lenguajes

Algunas cosas pueden ser confusas de resolver (chequeo de tipos de subprogramas, subprogramas anidados, etc.)

Ambiente de referencia para las referencias no locales dentro del cuerpo del subprograma pasado como parámetro

Debe determinarse cuál es el ambiente de referencia no local correcto para un subprograma que se ha invocado y que ha sido pasado como parámetro

Cuando un procedimiento que se pasó como parámetro se ejecuta, puede tener problemas con referencias que hay dentro del procedimiento

Si hay una variable que no pertenece a él y no es local entonces surge el problema a dónde va a buscarla porque este es un procedimiento que se mandó como parámetro

Si el lenguaje sigue la cadena estática para buscar su referencia se pregunta dónde está contenido este procedimiento, ahí no hay problema

Si el lenguaje sigue la cadena dinámica entonces se pregunta quién lo llamó

Hay varias opciones para resolverlo:

- Ligadura shallow o superficial: el ambiente de referencia, es el del subprograma que tiene declarado el parámetro formal del subprograma. Ejemplo: SNOBOL (usaremos este en la práctica)
- Ligadura deep o profunda: el ambiente es el del subprograma donde está declarado el subprograma usado como parámetro real. Se utiliza en los lenguajes con alcance estático y estructura de bloque
- Ligadura ad hoc: El ambiente del subprograma donde se encuentra el llamado a la unidad que tiene un parámetro subprograma. Menos fiable (poco natural)

Unidades genéricas

Son una opción para lenguajes que no permiten pasar parámetros de tipo subprogramas

Una unidad genérica es una unidad que puede instanciarse con parámetros formales de distinto tipo. Por ejemplo, una unidad genérica que ordena elementos de un arreglo podrá instanciarse para que ordene elementos enteros, flotantes, etc.

Permiten compartir cosas, procesos, procedimientos o lo que se defina, y reutilizar así código

Las unidades genéricas no se usan directamente, sino sólo para crear instancias no genéricas a partir de ellas

Las unidades genéricas representan una plantilla mediante la cual se indica al compilador cómo crear unidades no genéricas

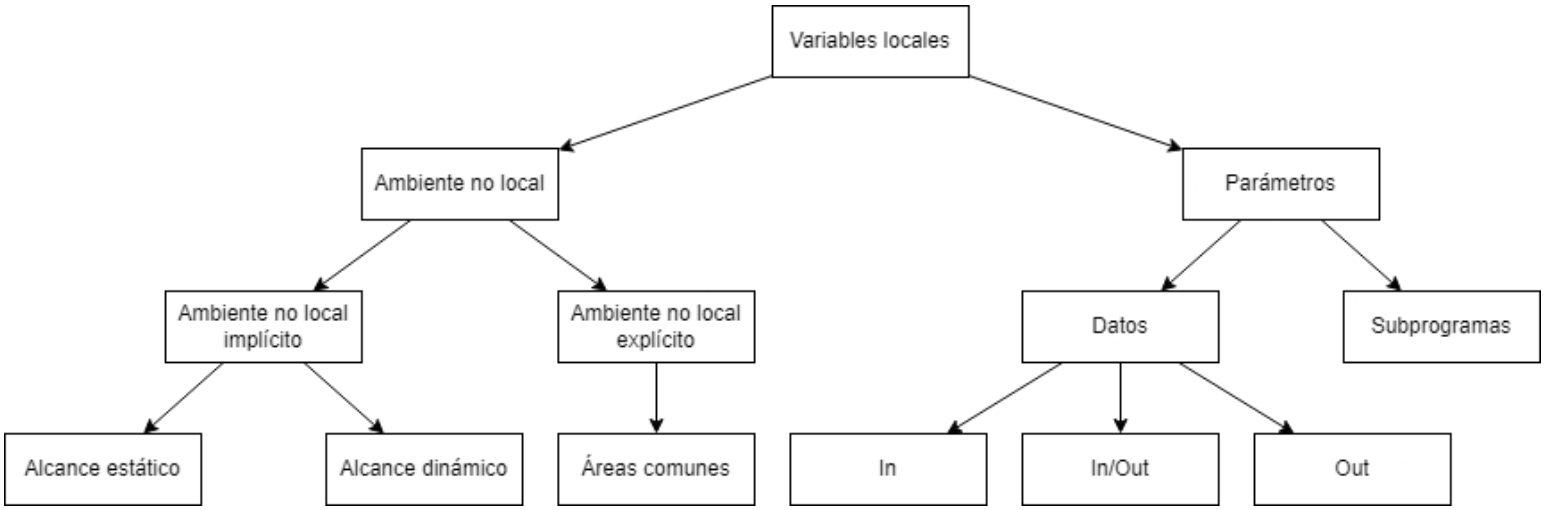
Como pueden usarse diferentes instancias con diferentes subprogramas, proveen la funcionalidad del parámetro subprograma

Ada permite crear unidades genéricas, con parámetros que se pueden concretar para diferentes instancias de la unidad, consiguiendo de esta manera la reutilización de código

Se debe anteceder la declaración de la unidad correspondiente con la palabra "generic"

Los parámetros formales de la unidad se sitúan en la zona comprendida entre la palabra "generic" y el comienzo de la declaración de la unidad

Para crear una instancia de una unidad genérica hay que especificar un nombre para la unidad no genérica que el compilador va a construir y los parámetros reales genéricos a utilizar en el lugar de los formales



Clase 7 - Tipos de datos

Concepto

Son más que un conjunto de datos, ya que tienen un comportamiento semántico o sentido

Podemos definir a un tipo como un conjunto de valores y un conjunto de operaciones que se pueden utilizar para manipularlos

Historia

Los primeros lenguajes tenían el inconveniente de que las estructuras de datos podían ser modeladas sólo con los pocos tipos de datos básicos definidos por el lenguaje

Se empieza a ver una clara intención de soportar varios y distintos tipos de datos con el objeto brindar un mayor apoyo al desarrollo de una amplia variedad de aplicaciones

Tomando el concepto de tipo de dato definido por el usuario arribamos al concepto de tipo de dato abstracto. Separa la representación y conjunto de operaciones (invisibles al usuario)

La evolución del TAD es el concepto de Clase

Tipo de dato

	Elementales/escalares	Compuestos
Predefinidos	• Enteros • Reales • Caracteres • Booleanos	String
Definidos por el usuario	Enumerados	• Arreglos • Registros • Listas

Su dominio es el conjunto de valores posibles

Cualquier lenguaje de programación está equipado con un conjunto finito de tipos predefinidos (built-in / primitivos), que normalmente reflejan el comportamiento del hardware subyacente

A nivel de hardware, los valores pertenecen al dominio sin tipo, lo que constituye el dominio universal, estos son interpretados de manera diferente, según los diferentes tipos que se utilicen

Los lenguajes de programación deben permitir al programador especificar agrupaciones de objetos de datos elementales (o tipos predefinidos)

Estas definiciones de tipos de datos en función de otros denominan tipos de datos definidos por el usuario

Lo puede realizar de forma recursiva, o mediante agregaciones de agregados o uniones

Esto se logra mediante la prestación de una serie de constructores que permiten definir a estos tipos de datos definidos por el usuario

Entre los tipos compuestos definidos por el usuario se encuentran los tipos estructurados pues se utilizan estructuras de datos para su composición

Tipos predefinidos

Reflejan el comportamiento del hardware subyacente y son una abstracción de él

Las ventajas de los tipos predefinidos son:

- Invisibilidad de la representación
- Verificación estática
- Desambiguar operadores
- Control de precisión

Que un conjunto de valores de un tipo sea definido por la implementación del lenguaje significa que será seleccionado por el compilador. Mientras que si el tipo es definido por el lenguaje será definido en su definición

Tipos definidos por el usuario

Los lenguajes de programación permiten al programador especificar agrupaciones de objetos de datos elementales (o tipos predefinidos) y, de forma recursiva, agregaciones

de agregados. Esto se logra mediante la prestación de una serie de constructores que permiten definir lo que denominamos tipo de dato definido por el usuario

Separan la especificación de la implementación. Se definen los tipos que el problema necesita

Permiten:

- Instanciar objetos de las agregaciones
- Definir nuevos tipos de dichas agregaciones
- Hacer un chequeo de consistencia

Ventajas:

- Legibilidad: se puede elegir el nombre más apropiado para el nuevo tipo
- Estructura jerárquica de las definiciones de tipos
- Modificabilidad: solo se cambia en la definición
- Factorización: se usa la cantidad de veces necesarias
- La instanciación de los objetos en un tipo dado implica una descripción abstracta de sus valores. Los detalles de la implementación solo quedan en la definición del tipo

Constructores de tipos compuestos

Producto cartesiano

Es un registro

Por ej.: un registro de un polígono representa el producto cartesiano entre el conjunto de los enteros (para el número de lados) y el conjunto de los reales (para el tamaño de dichos lados)

Correspondencia finita

Es un arreglo (cada número dentro de la dimensión del arreglo se corresponde a un valor al que se accede mediante un subíndice)

Mapea valores de un tipo de dato hacia otros, a través del cumplimiento de una relación

Ej.Arreglos, vectores, matrices, listas de Python

Están indexados, ordenados. Algunos usan [], pero ADA usa () para acceder por posición

En APL, Algol 68, Ada, Python se pueden indexar por más de un elemento en el rango (slicing)

Los lenguajes interpretados permiten tener elementos de distinto tipo

Unión

Permite manipular diferentes tipos en distinto momento de la ejecución

En la unión y en la unión discriminada el chequeo de tipos debe hacerse en ejecución

La declaración es muy similar a la del producto cartesiano. La diferencia es que sus campos son mutuamente excluyentes

Unión discriminada:

Agrega un discriminante para indicar la opción elegida

Si tenemos la unión discriminada entre dos conjuntos S y T, y aplicamos el discriminante a un elemento **e** perteneciente a la unión discriminada, devolverá S o T

El elemento **e** debe manipularse de acuerdo al valor del discriminante

La unión discriminada se puede manejar en forma segura consultando el discriminante antes de utilizar el valor del elemento

Las uniones discriminadas son un poco más seguras pues permiten al programador manejar la situación a través del discriminante

Algunos lenguajes como Pascal, soportan unión discriminada a través de la definición de variantes:

```
type natural = 0. .maxint;
address_type = (absolute, offset);
safe_address = record
        case kind: address_type of
                absolute: (abs_addr: natural)
                offset: (off_addr: integer)
        end;
```

Problemas:

- El chequeo debe realizarse en ejecución. No se puede asegurar en compilación qué tipo o variante adquiere una variable

- El discriminante y las variantes pueden manejarse independientemente uno de otros
- La implementación del lenguaje puede ignorar los chequeos
- Puede omitirse el discriminante, con lo cual aunque se quisiera no se puede chequear

Recursión

Un tipo de dato recursivo T se define como una estructura que puede contener componentes del tipo T

Define datos agrupados:

- Cuyo tamaño puede crecer arbitrariamente
- Cuya estructura puede ser arbitrariamente compleja

Los lenguajes de programación convencionales soportan la implementación de los tipos de datos recursivos a través de los punteros

Un puntero es una referencia a un objeto

Una variable puntero es una variable cuyo r-valor es una referencia a un objeto (el l-valor del otro objeto)

Punteros

Estructuras de tamaño arbitrario

Número de ítems no determinado: los punteros permiten conectar juntos muchos items sin tener un nombre explícito para todos ellos (recursión)

Relaciones múltiples entre los ítems: los punteros permiten que el dato sea puesto en varias estructuras sin necesidad de duplicarlo

Acceso a bajo nivel: los punteros están cerca de la máquina

Valores:

- Direcciones de memoria
- Valor nulo (no asignado) dirección no válida

Operaciones:

- Asignación de valor: generalmente asociado a la alocación de la variable apuntada
- Referencias: a su valor (como dirección), operaciones entre punteros al valor de la variable apuntada: desreferenciación implícita

Los punteros son un mecanismo muy potente para definir estructuras de datos recursivas

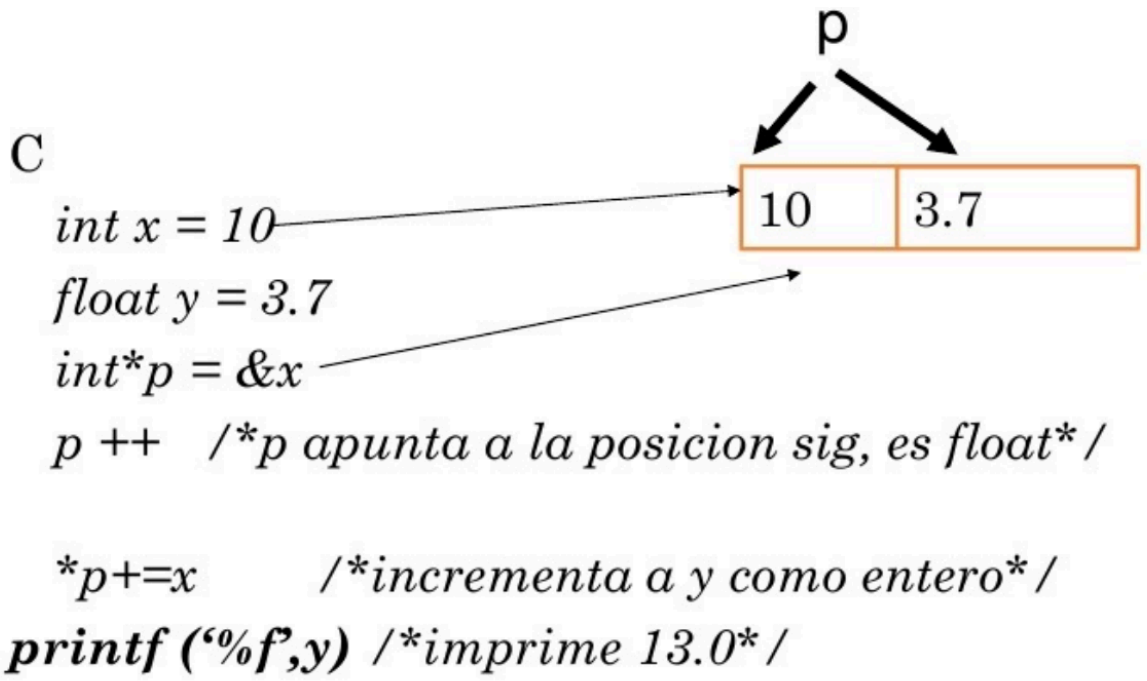
Por acceder a bajo nivel, pueden oscurecer o hacer inseguros a los programas que los usan

Se compara a los punteros con los go to:

- Los go to amplían el rango de sentencias que pueden ejecutar
- Los punteros amplían el rango de las celdas de memoria que pueden ser referenciadas por una variable y también amplían el tipo de los valores que puede contener un objeto

Inseguridades

Violación de tipos



Referencias sueltas (referencias dangling)

Si este objeto no está alocado se dice que el puntero es peligroso (dangling)

Una referencia suelta o dangling es un puntero que contiene una dirección de una variable dinámica que fue desalocada

Si luego se usa el puntero producirá error

Punteros no inicializados

Peligro de acceso descontrolado a posiciones de memoria

Verificación dinámica de la inicialización

Solución: valor especial nulo (nil/void/null)

Punteros y uniones discriminadas

```
union ojo{
    int int_var,
    int* int_ref
}
```

En el caso de C, este es el mismo efecto que causa la aritmética de punteros

Alias

```
int* p1;
int* p2;
int x;
p1 = &x;
p2 = &x;
```

p1 y p2 son punteros

p1 y x son alias

p2 y x también lo son

Se da si hay variables comparten un objeto en el mismo entorno de referencia, y sus caminos de acceso conducen al mismo objeto

El objeto compartido modificado vía un camino se modifica para todos los caminos

Ventaja: compartir objetos se utiliza para mejorar la eficiencia

Desventajas:

- Genera programas que sean difíciles de leer
- Puede generar errores porque el valor de una variable se puede modificar incluso cuando no se utiliza su nombre

Liberación de memoria: objetos perdidos

Las variables puntero se alocan como cualquier otra variable en la pila de registros de activación

Los objetos (apuntados) que se alocan a través de la primitiva new son alocados en la heap

La memoria disponible (heap) podría rápidamente agotarse a menos que de alguna forma se devuelva el almacenamiento alocado liberado

Manejo de memoria

Si los objetos en el heap dejan de ser accesibles esa memoria podría liberarse

Un objeto se dice accesible si alguna variable en la pila lo apunta directa o indirectamente

Un objeto es basura si no es accesible

La liberación de espacio en la heap puede ser explícita (dispose()/delete()) o implícita (garbage collector)

El reconocimiento de la basura recae en el programador, quien notifica al sistema cuando un objeto ya no se usa

No garantiza que no haya otro puntero que apunte a esta dirección definida como basura (si eso pasa, este otro puntero se transforma en dangling)

Este error es difícil de chequear y la mayoría de los lenguajes no lo implementan por que es costoso

En caso de implementar un garbage collector el mismo debe ser muy eficiente y ejecutar en paralelo para no bajar el rendimiento del lenguaje