

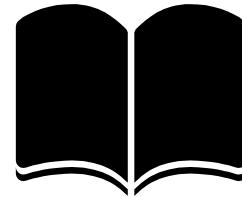
# Lenguajes de Programación I



Página web: <https://lengprg1.alumnos.exa.unicen.edu.ar/>

**Docentes:** José María Massa, Marcela Castro, Hugo Manterola, José Fernández León - Camila García, Luciano Giménez

# Organización de la materia



## Organización del curso

- Clases Teóricas (martes 13 a 16 hs)
- Clases Prácticas (jueves 14 a 17 hs)
  - TP Especiales (laboratorio)
- Clases Teórico-Prácticas
- Material (Clases, Trabajos Prácticos, Resoluciones y apuntes): Página de la materia
- Novedades y autoevaluación: Página / Moodle

## Régimen de aprobación

- Cursada: Parcial, Recuperatorio y Prefinal
- Final: Escrito o Mixto

## Bibliografía de base:

- Cox, R., Griesemer, R., Pike, R., Taylor, I. L., & Thompson, K. (2022). *The Go programming language and environment*. Communications of the ACM, 65(5), 70-78.
- Klabnik, S., & Nichols, C. (2023). *The Rust programming language*. No Starch Press.
- Concepts of Programming Languages, R. Sebesta, Addison-Wesley, 2016
- Programming Languages: Principles and Practices, K. Louden, Course Technology, 2011
- Programming Language Concepts, C. Ghezzi, M. Jazayeri, Wiley, J & Sons, 1997 EN BIBLIOTECA UNICEN
- Programming Languages: Design and Implementation, V. Pratt, T. W., M., Zelkowitz, M. V. Zelkowitz, Prentice Hall, 2000 EN BIBLIOTECA UNICEN
- Programming Languages: Principles and Paradigms, M. Gabrielli, S. Martíni, Springer, 2010
- Fundamentos de los Lenguajes de Programación, E. Horowitz, Springer Verlag, 1985

# Objetivos



## Principales objetivos de la materia

- Disminuir la curva de aprendizaje de un nuevo lenguaje, reconociendo sus características semánticas esenciales.
- Identificar los principales aspectos semánticos, sintácticos y pragmáticos de los lenguajes de programación.
- Identificar las principales limitaciones de un lenguaje y mitigar las mismas con un adecuado diseño de la aplicación.
- Seleccionar, diseñar o modificar un lenguaje para un problema determinado.

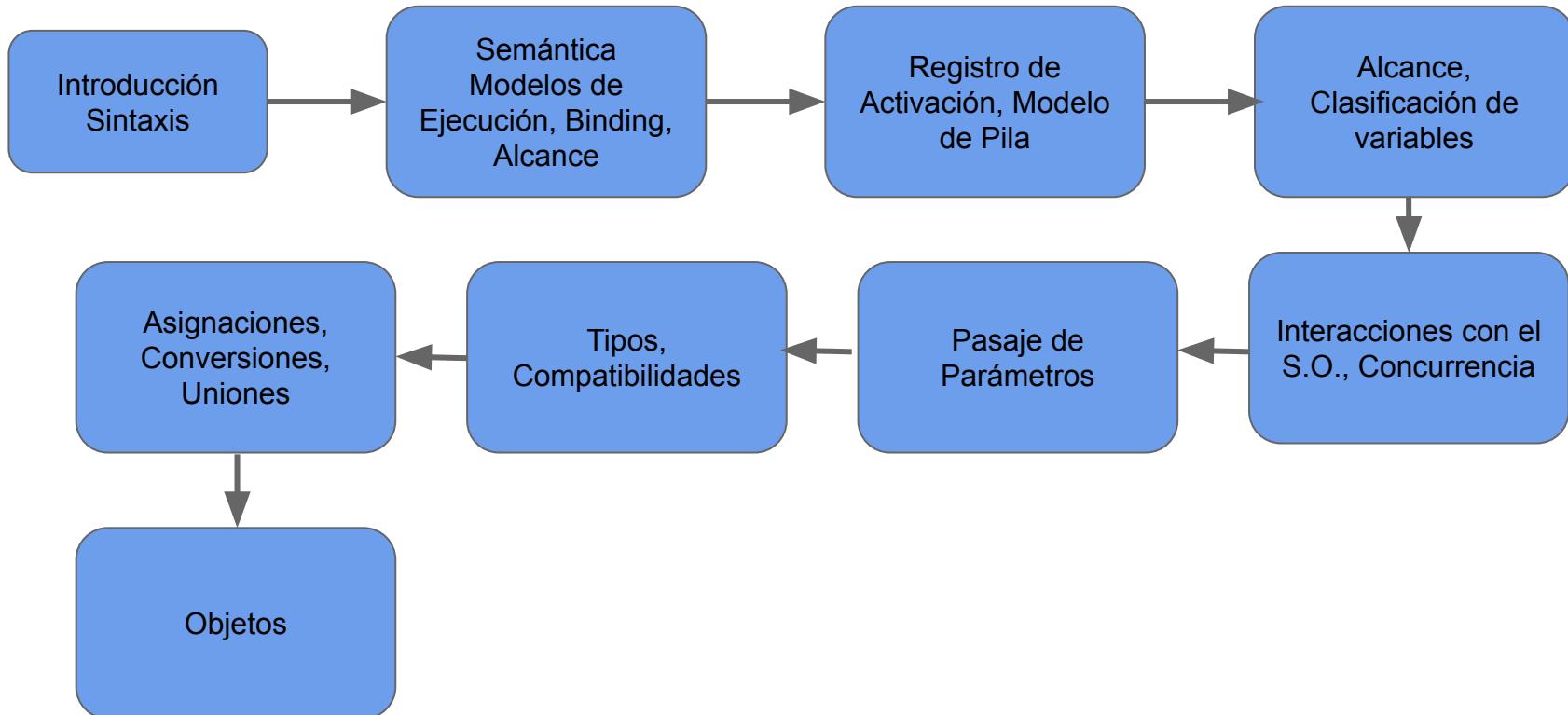
## Qué se va a ver

- Conceptos esenciales de los lenguajes de programación.
- Efectos que estos conceptos tienen en los paradigmas de solución de problemas.
- Programas mínimos para ejercitarse con conceptos.

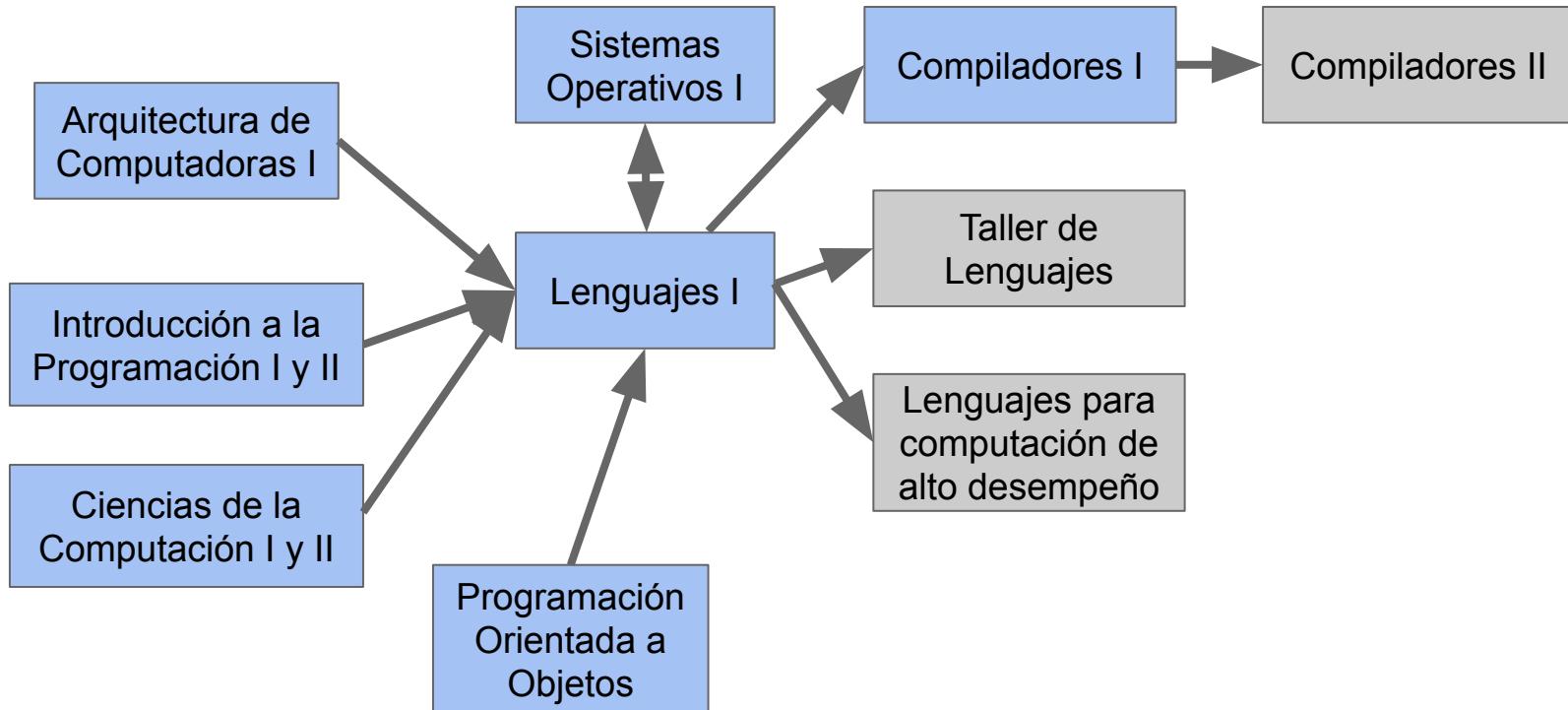
## Qué no se va a ver

- Estudio detallado de lenguajes en particular.
- Problemas de Programación.
- Programas grandes y complejos.

# Programa de la materia



# Encuadre de la Materia en la carrera

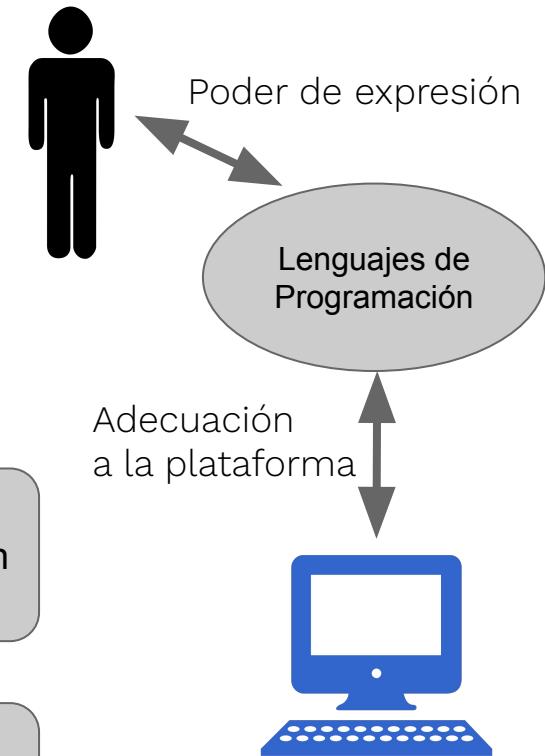
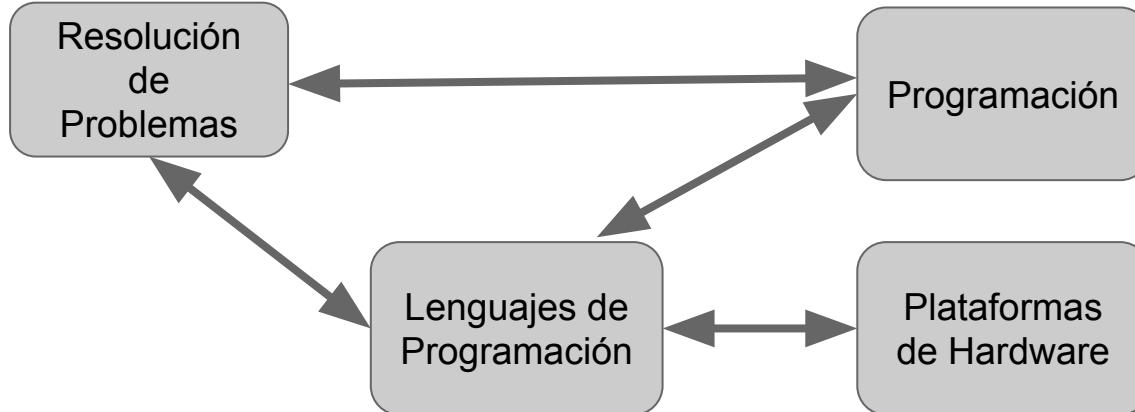


# Introducción

## ¿Qué son los lenguajes de Programación?

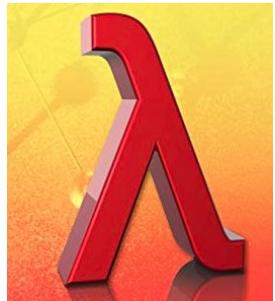
- Lenguajes para comunicación entre una computadora y un humano.
  - Comunicación entre seres humanos.

## ¿Cómo surgen los lenguajes de programación?



# Poder de expresión y adecuación al hardware

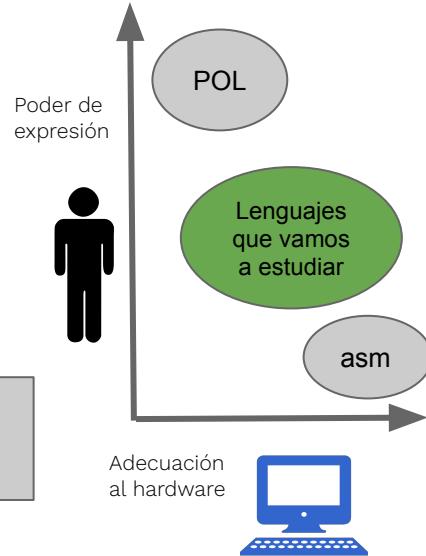
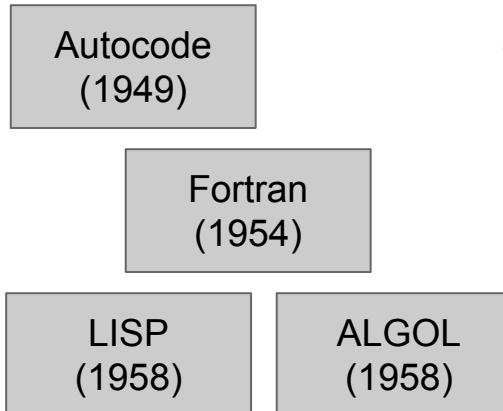
## Evolución histórica



Lambda Calculus  
(Alonzo Church, 1936)



Plankalkül (Konrad  
Zuse, 1942)



Vamos a estudiar los lenguajes de  
programación de alto nivel

¿Por qué hoy siguen existiendo  
lenguajes de bajo nivel (assembler,  
web assembly, bytecodes, etc.) ?

# ¿Cómo vamos a estudiar los lenguajes?

## Existe un gran conjunto de lenguajes de programación

### The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software Issues*, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework. This framework dictates the rules about the uses of user-coined names, and the conventions about characterizing functional relationships. Within this framework the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a nonprocedural (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side

differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have fielded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to

- History of Programming Languages:  
<https://hopl.info/> (8945 lenguajes)

## Es imposible estudiarlos por extensión

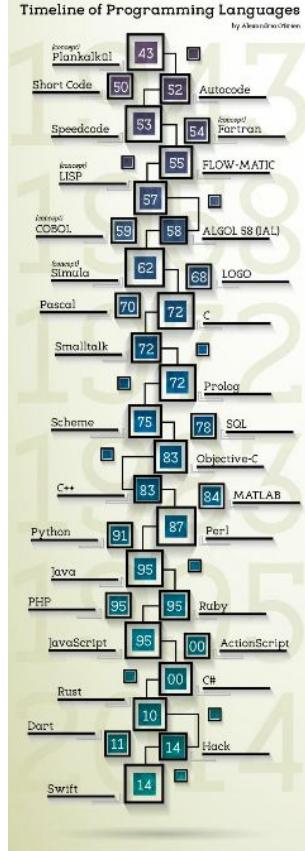
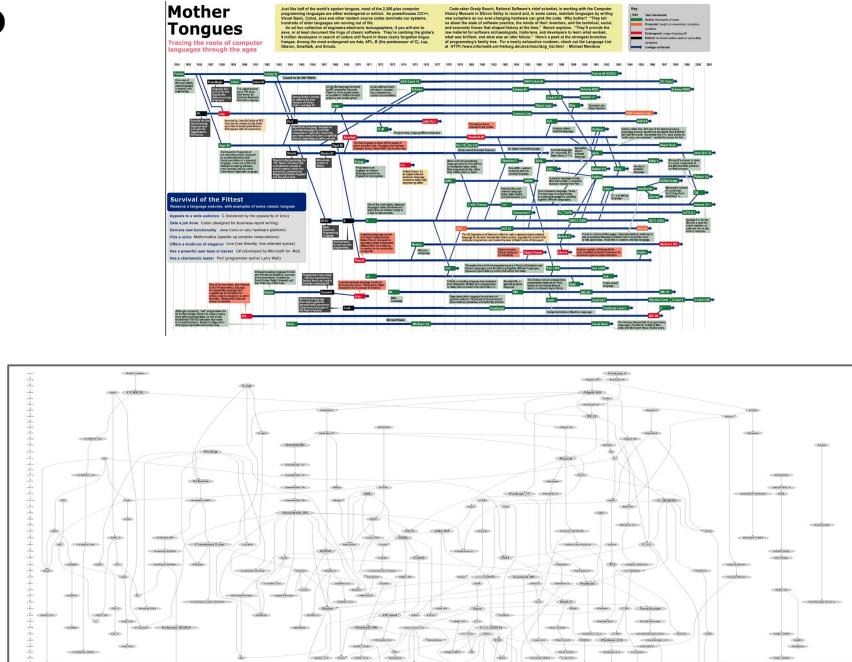
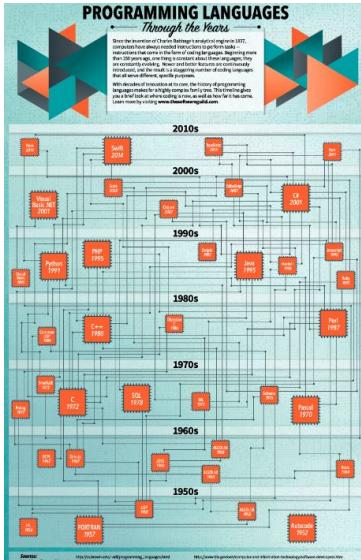
### ¿Cómo los vamos a estudiar?

## Vamos a estudiarlos por comprensión

### ¿Cómo vamos a definir las categorías?

# ¿Cómo vamos a estudiar los lenguajes?

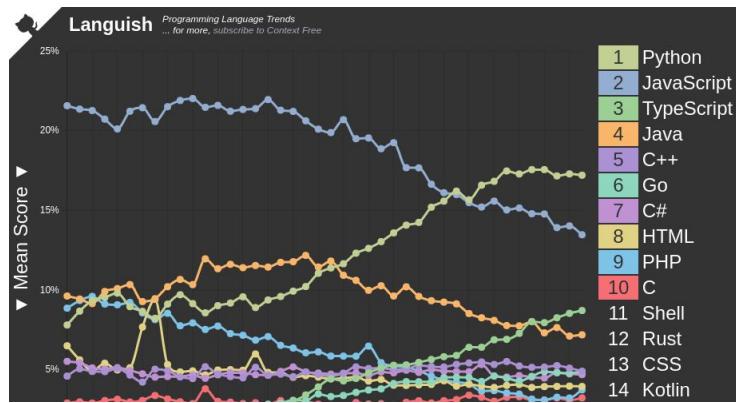
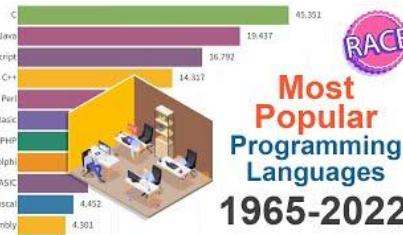
## Estudio cronológico



[Línea de tiempo en tamaño completo](#)

# ¿Cómo vamos a estudiar los lenguajes?

## Popularidad



- <https://www.youtube.com/c/CodingTech>
- <https://contextfree.info/>

Worldwide, Aug 2023 :

| Rank | Change | Language    | Share   | 1-year trend |
|------|--------|-------------|---------|--------------|
| 1    |        | Python      | 28.04 % | +0.3 %       |
| 2    |        | Java        | 15.78 % | -1.3 %       |
| 3    |        | JavaScript  | 9.27 %  | -0.2 %       |
| 4    |        | C#          | 6.77 %  | -0.2 %       |
| 5    |        | C/C++       | 6.59 %  | +0.4 %       |
| 6    |        | PHP         | 5.01 %  | -0.4 %       |
| 7    |        | R           | 4.35 %  | +0.0 %       |
| 8    |        | TypeScript  | 3.09 %  | +0.3 %       |
| 9    | ↑↑     | Swift       | 2.54 %  | +0.5 %       |
| 10   |        | Objective-C | 2.15 %  | +0.1 %       |
| 11   | ↑↑     | Rust        | 2.14 %  | +0.5 %       |
| 12   | ↓↓↓    | Go          | 1.93 %  | -0.2 %       |
| 13   | ↓      | Kotlin      | 1.77 %  | -0.0 %       |
| 14   |        | Matlab      | 1.63 %  | +0.1 %       |
| 15   | ↑↑↑↑   | Ada         | 1.08 %  | +0.3 %       |
| 16   | ↓      | Ruby        | 1.06 %  | -0.1 %       |

<http://pypl.github.io/PYPL.html>

<https://www.tiobe.com/tiobe-index/>

# ¿Cómo vamos a estudiar los lenguajes?

## Por paradigma

- Celdas de memoria
- Secuencia de instrucciones
- Instrucciones leen y modifican las celdas de memoria

**Imperativo**

- Afirmaciones lógicas
- Reglas
- Preguntas

**Lógico**

- Definición de funciones
- Aplicación de funciones

**Funcional**

# Ejemplos de Paradigmas

## Imperativo:

```
int x = 3;  
int sum = 0;  
for (int i=x;i>0;i=i-1) {  
    sum = sum + i;  
}
```



Variables (celdas de memoria)

Secuencia de instrucciones

Leen y modifican las celdas de memoria

## Lógico:

Afirmación lógica

```
sum(0,0).  
sum(X, S):-  
    Aux is X - 1,  
    sum(Aux,R),  
    S is X + R.  
?-sum(3,Z).
```

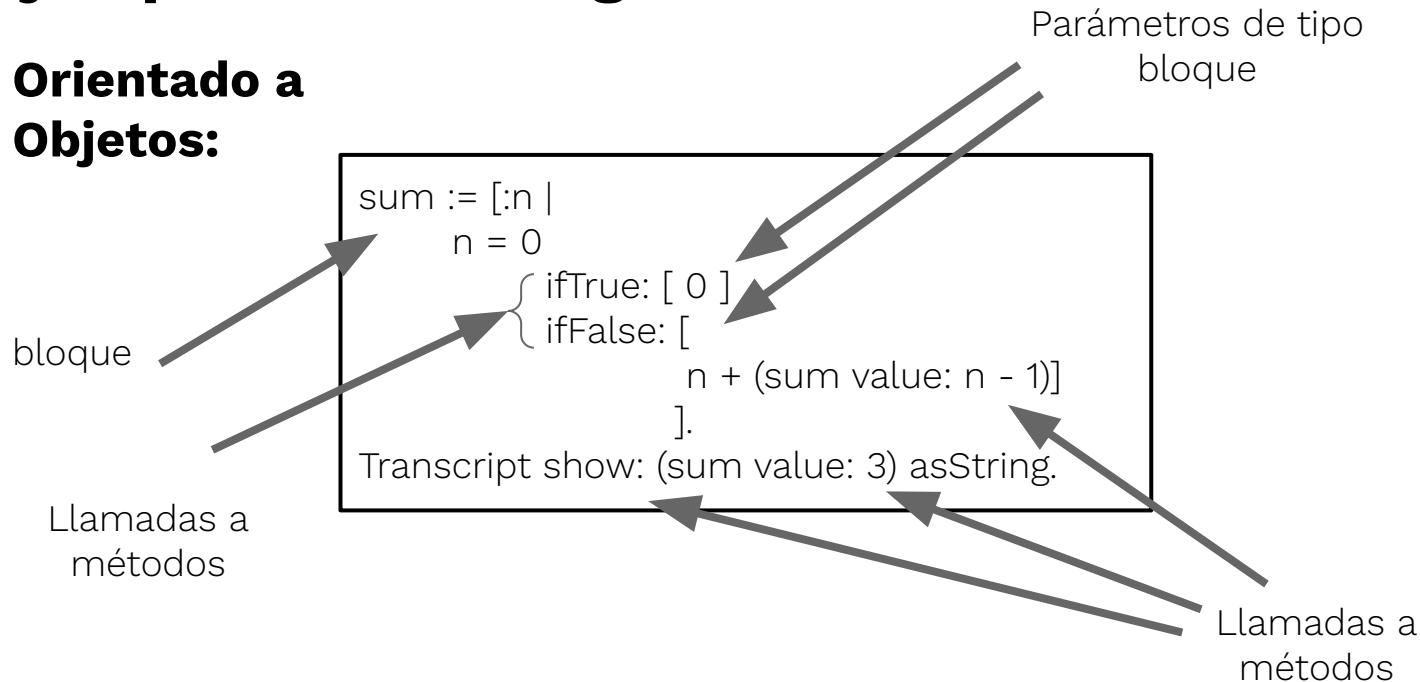
Regla de inferencia

Se puede escribir como:  
sum(0, 0) :- true.

Preguntas

# Ejemplos de Paradigmas

## Orientado a Objetos:



# Ejemplos de Paradigmas

## Funcional:

```
(define (sum x)
  (if (= x 0) 0
      (+ x (sum (- x 1)))))

(display (sum 3))
```

Definición de funciones

Aplicación de funciones

Lenguajes Funcionales  
(puros e impuros):

- Familia de Lisp
- Scheme
- Haskell
- Scala
- Ocam
- Clojure
- Racket
- ...

¿Qué sucede con la “inexistencia” de variables?

En muchos de estos lenguajes, las variables no existen... para el programador.

¿Dónde se almacenan los valores intermedios que devuelve una función?

# Lenguajes Funcionales

## Problema de la decidibilidad



## Cálculo Lambda

$\lambda x.M$

- $x$  es una variable
- $M$  es una expresión

$(\lambda x.M) N$

- $N$  es una expresión
- Reducción  $\beta$ : se reemplaza en  $M$ ,  $x$  por  $N$

En  $M$ :

- $x$ : variable ligada
- otras variables: variables libres

¿Qué se puede lograr con esto?

## Definir funciones:

$(\lambda x. 1) x$

Concatena 1 con  $x$

Las funciones no tienen nombre

Aplicar funciones:  $(\lambda x. 1) 2$

1 2

Reducción  $\beta$  se reemplaza  $x$  por 2

¿A qué se parece esto en algún lenguaje conocido?

-> Church, A., "An unsolvable problem of elementary number theory", American journal of mathematics, Vol 58, Nro. 2, 1936, pág. 345-363.

-> Turing, A. M., "On computable numbers, with an application to the Entscheidungsproblem", Proceedings of the London mathematical society, Vol 2, Nro. 1, 1937, pág. 230-265.

# Lenguajes Funcionales

**Enviar funciones como parámetro:**

$$(\lambda x. 1(x\ 3))\ (\lambda y. 2\ y)$$

Función que recibe el parámetro

$$1((\lambda y. 2\ y)\ 3) \quad \text{Reducción } \beta$$

Función enviada como parámetro

$$1(2\ 3) \quad \text{Reducción } \beta$$

**Retornar funciones como resultado:**

$$(\lambda x. (\lambda y. (3\ x\ y)))\ 2\ 1$$

Función que retorna la función

$$(\lambda y. 3\ 2\ y)\ 1 \quad \text{Reducción } \beta$$

Función retornada a ser ejecutada en el contexto externo con el parámetro 1

$$(3\ 2)\ 1 \quad \text{Reducción } \beta$$

## Funciones:

- Pueden invocarse
  - Pueden ser enviadas como parámetros.
  - Pueden ser devueltas como resultado de una función.
  - Pueden asignarse a variables.
  - Pueden formar parte de tipos de datos estructurados.
- 3ra.  
2da.  
1ra.

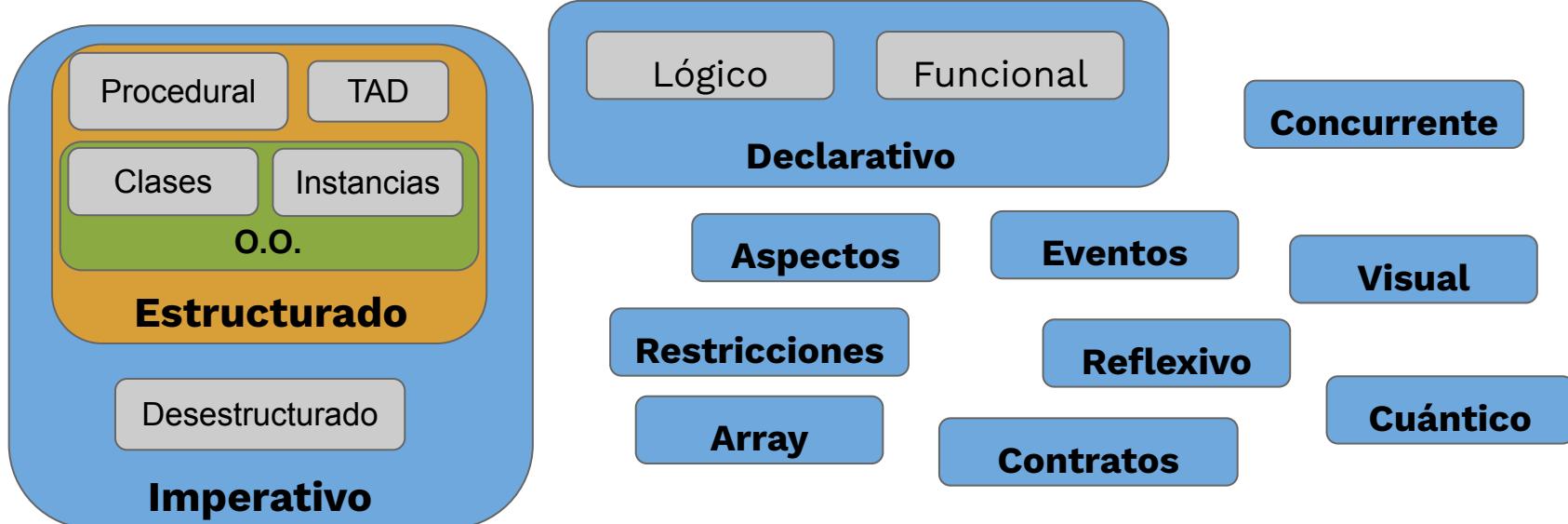
**Lenguajes funcionales “puros”:** LISP I, Scheme, Clojure, Racket, Haskell, F#, etc.

**Lenguajes que incorporaron estas características:** Java (2006), PHP (2009), Python 2.7 (2010), C++ (2011), C# (2017), etc.

**Lenguajes que se crearon incorporando estas características:** Dart (2009), Rust (2010), Go (2012), Kotlin (2016), etc.

# ¿Cómo vamos a estudiar los lenguajes?

Por categoría/ paradigma

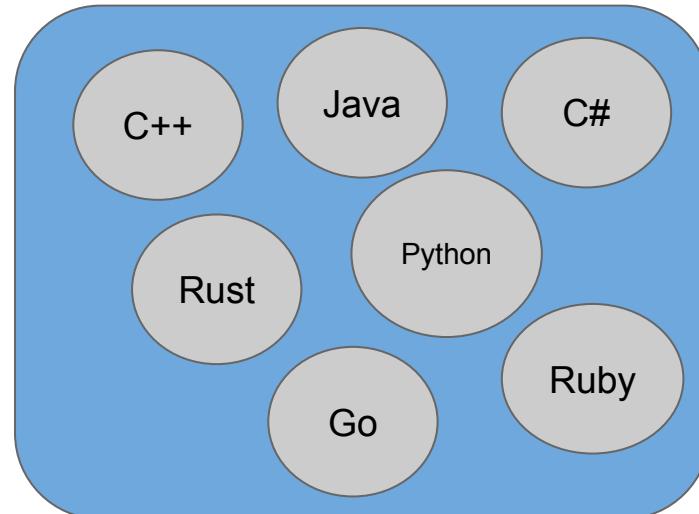


- Cada paradigma se basa en un conjunto de conceptos
- Cada paradigma tiene sub-paradigmas
- Muchos lenguajes tienen cierto grado de soporte a varios paradigmas

# ¿Cómo vamos a estudiar los lenguajes?

## Por categoría/ paradigma

- **Lenguajes Imperativos**
  - **Objetos**
  - **Características funcionales**
  - **Aspectos**
  - **Eventos**
  - **Concurrencia**
  - **Visual**
  - **Características Declarativas**
  - **etc.**



**Multiparadigma**

[https://en.wikipedia.org/wiki/Comparison\\_of\\_multi-paradigm\\_programming\\_languages](https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages)

# ¿Qué sucede hoy en día con los paradigmas?

```
class A {  
    constructor(x) {  
        this.x = x;  
    }  
    calcular() {  
        const sumacuadrados = [1, 2, 3, 4]  
            .filter(num => num % 2 === 0)  
            .map(even => even * even)  
            .reduce((x, y) => x + y);  
        console.log(sumacuadrados);  
    }  
}  
  
let a1 = new A(3);  
a1.calcular();
```

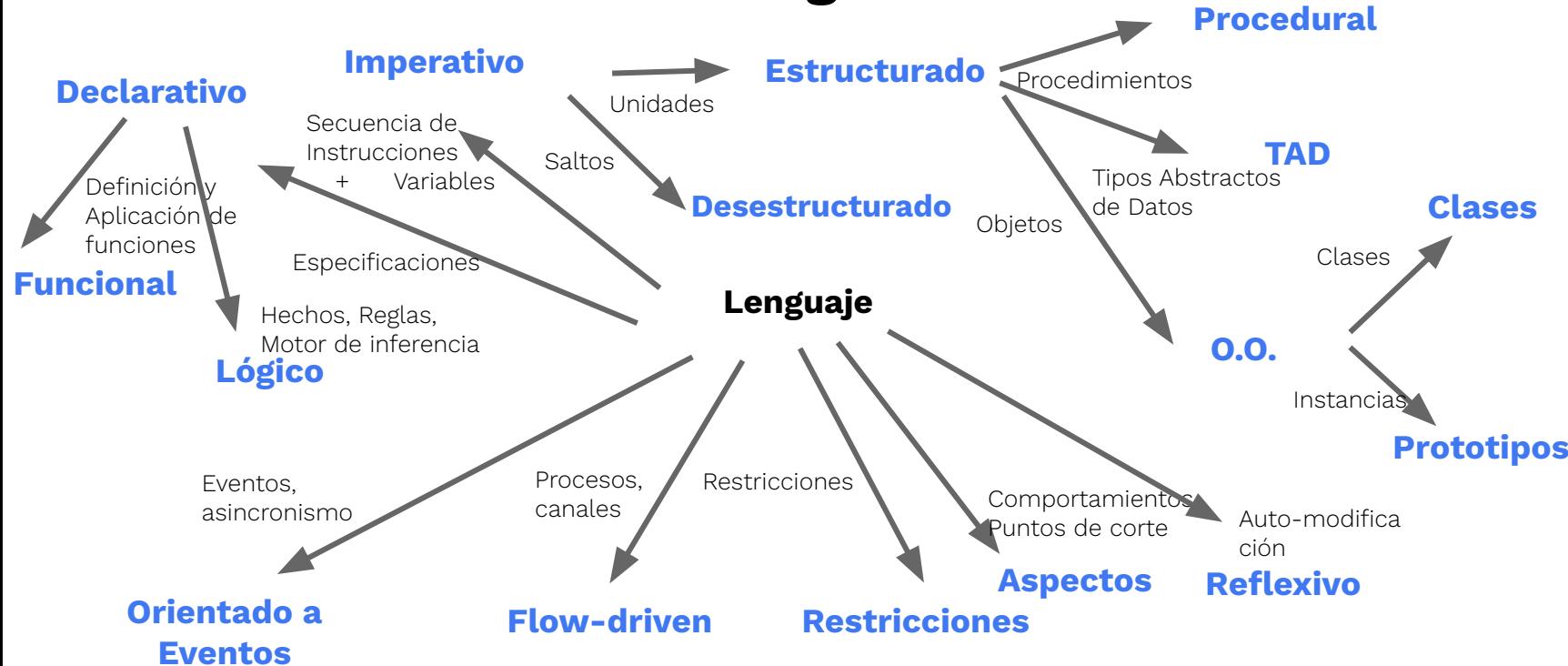
Javascript

Orientación a  
Objetos  
(instancias)

Funcional

Procedural

# Características de Paradigmas



# ¿Cómo vamos a estudiar los lenguajes?

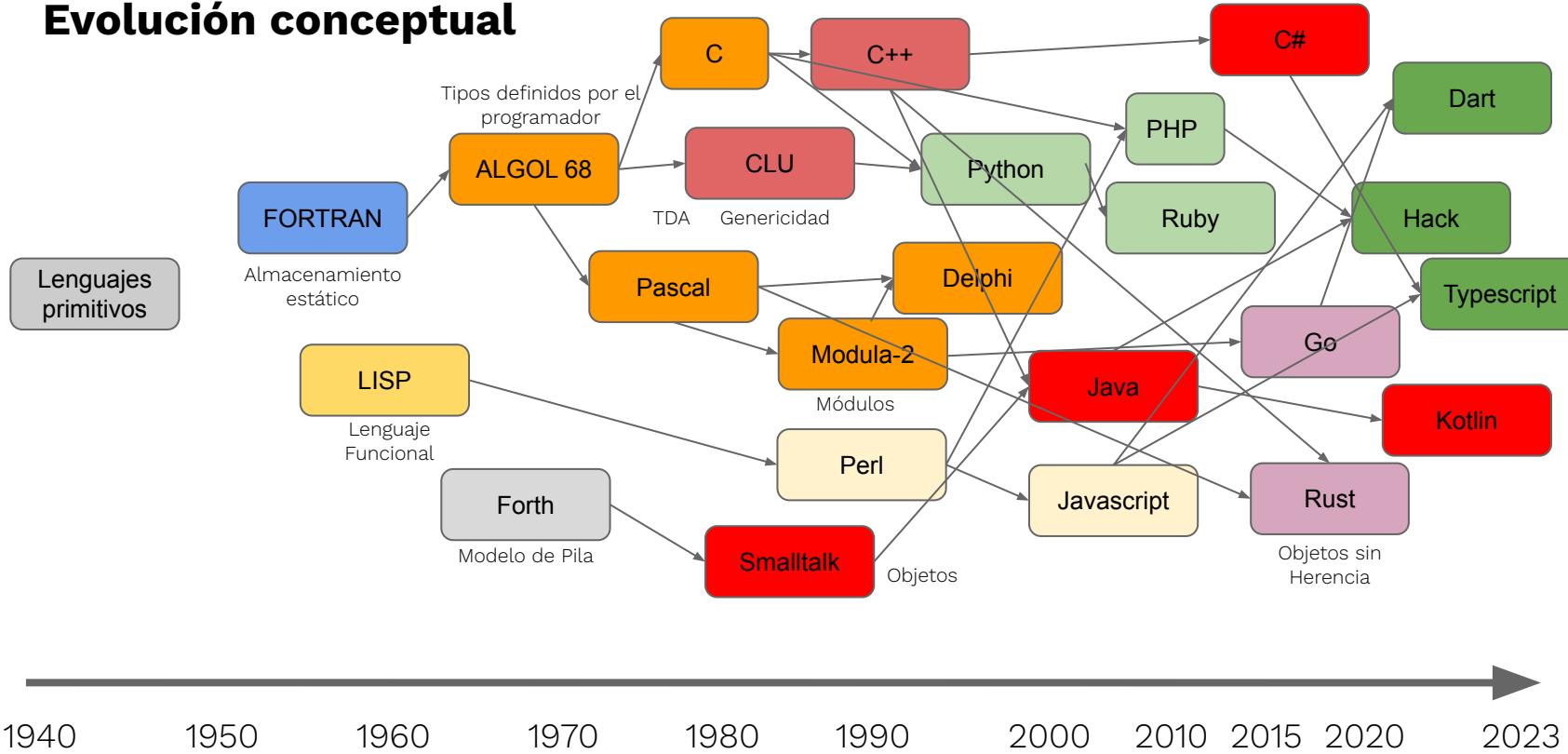
## Por modelo de ejecución



- Lenguajes con versiones compiladas e interpretadas (C y Basic)
- Lenguajes dinámicos que se compilan a una representación intermedia (Python, Smalltalk)
- Lenguajes híbridos que son primero compilados y luego interpretados (Java, Kotlin)
- Lenguajes con pila de ejecución en el heap (Python, etc.)

# ¿Cómo vamos a estudiar los lenguajes?

## Evolución conceptual



# **¿Cómo vamos a estudiar los lenguajes?**



## **Permanecen en el tiempo**

- Closures de hoy en día (Java, Python, C++, C#, Swift, Go, Rust, Javascript, Typescript, Perl, Ruby, etc. ) tienen su origen en las expresiones lambda de LISP de 1958
  - GoRoutines de Go tienen su origen en las co-rutinas de FORTRAN de 1954

## **Identificar aspectos**

- Sintácticos
  - Semánticos
  - Pragmáticos

## **Patrones que se repiten en varios lenguajes**

- Tipos
  - Alcance/Ámbito
  - Almacenamiento
  - Concurrencia
  - Nombre
  - Tiempo de vida
  - Valor
  - Pasaje de parámetros
  - ...



# Aspectos de los lenguajes

## ¿Cómo se escriben los diferentes elementos del lenguaje?

- En Java, los identificadores son sensibles a las mayúsculas.
- En C++, los modificadores de acceso para los atributos de las clases se colocan delante de su declaración. Ej. public int c;
- En Python se utiliza la indentación para definir ámbitos.
- En Typescript las variables se declaran con la palabra reservada let, por ejemplo let a = 3;

## Sintaxis

## ¿Qué significado tienen las sentencias?

- En PHP cuando se hace la asignación "\$var1 = &\$var2;" la misma, es por referencia, es decir la dirección de \$var2 es copiada en \$var1
- En Java la palabra reservada static para los atributos indica que estos se almacenan de forma estática.
- En Python 3 la palabra reservada nonlocal aplicada a una variable indica que la variable está declarada en un ámbito no local..

## Semántica

## ¿Cómo se hace para que las sentencias logren el efecto deseado?

- En C++, la inicialización de las variables puede variar de un compilador a otro.
- Si en C++ se crean muchos hilos de ejecución (threads) se aumenta la probabilidad de conflictos de memoria.
- En Java, la configuración del garbage collector puede provocar que el programa se detenga muy frecuentemente por ciertos períodos de tiempo.

## Pragmática

La precedencia es un aspecto semántico pero puede lograrse con la definición de la sintaxis.

No son siempre independientes

Es posible describir una semántica a través del resultado de las operaciones (influenciado por la pragmática).

# Sintaxis

**Especifica la forma en la que se deben escribir los programas**

Elementos del lenguaje:

- Constantes
- Variables
- Palabras clave
- Parámetros
- Tipos
- Anotaciones
- etc.

Cumplen las funciones de las palabras del lenguaje natural

## Regla léxica

- Los identificadores comienzan con una letra, seguida de letras o números y tienen una longitud de 10 caracteres.

## Reglas léxicas

Especifican cómo se escribe cada elemento

## Reglas sintácticas

Especifican cómo se combinan los elementos para formar programas

## Regla sintáctica

- Una definición de función comienza con el tipo de valor de retorno, un identificador, a continuación una lista de parámetros formales y finalmente posee un cuerpo.

# Sintaxis

## Sistemas de Formalización:

- BNF (Backus Naur Form)
- Extended BNF
- Augmented BNF
- ANTLR
- BNF con extensiones regulares
- Diagramas sintácticos

Java1.8:

<https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html>

Python 2.7:

<https://docs.python.org/2/reference/grammar.html>

Python 3.6:

<https://docs.python.org/3/reference/grammar.html>

Kotlin: <https://kotlinlang.org/docs/reference/grammar.html>

C++: <https://www.bogotobogo.org/hch/>

<https://golang.org/ref/spec>

Objeto de la  
gramática

- No Terminal ::= expansion
- ::= significa se define como
- No Terminales entre <>
- Terminales entre “”

## RE-BNF

arith\_expr: term (( '+' | '-' ) term)\*  
decorators: decorator+

- Repetición 0 o más veces: \* a la derecha
- Repetición 1 o más veces: + a la derecha
- Opcional 0 o 1 vez: ? a la derecha

## BNF

<asig> ::= <id> "=" <expr>  
<expr> ::= <exp> "+" <term>  
<expr> ::= <exp> "-" <term>  
<expr> ::= <term>  
<term> ::= <term> "\*" <factor>  
| <term> "/" <factor>  
<term> ::= <factor>  
<factor> ::= <var> | <cte>

Recursividad  
declarativa

## EBNF

<term> ::= [ "-"] <factor>  
<args> ::= <arg> { "," <arg>}  
<expr> ::= <term> ("+" | "-") <expr>

- Opción: elementos entre [ ]
- Repetición: elementos entre { }
- Agrupamiento: elementos entre ( )

Cada lenguaje elige un sistema particular

# Gramática de un lenguaje

Vamos a definir  
un lenguaje  
completo

Se podría  
reducir  
aún más

Muchos lenguajes  
poseen gramáticas  
pequeñas

- C
- Python

Reglas  
sintácticas

Reglas  
léxicas

|   |
|---|
| 1) <programa> --> <sentencia>                                   |
| 2) <programa> --> <programa> <sentencia>                        |
| 3) <sentencia> --> <seleccion>                                  |
| 4) <sentencia> --> <iteracion>                                  |
| 5) <sentencia> --> <asignacion>                                 |
| 6) <seleccion> -> if <condicion> then <bloque>                  |
| 7) <seleccion> -> if <condicion> then <bloque> else <bloque>    |
| 8) <condicion> --> <comparacion>                                |
| 9) <condicion> --> <condicion> and <comparacion>                |
| 10) <condicion> --> <condicion> or <comparacion>                |
| 11) <iteracion> --> while <condicion> do <bloque>               |
| 12) <asignacion> --> <var> := <expresion>                       |
| 13) <bloque> --> begin <grupo de sentencias> end                |
| 14) <grupo de sentencias> --> <sentencia>                       |
| 15) <grupo de sentencias> --> <grupo de sentencias> <sentencia> |
| 16) <comparacion> --> <expresion> <comparador> <expresion>      |
| 17) <comparador> --> >  |
| 18) <comparador> --> >=   |
| 19) <comparador> --> ==   |
| 20) <comparador> --> <  |
| 21) <comparador> --> <  |
| 22) <comparador> --> <=   |
| 23) <expresion> --> <expresion> + <termino>                     |
| 24) <expresion> --> <expresion> - <termino>                     |
| 25) <expresion> --> <termino>                                   |
| 26) <termino> --> <termino> * <factor>                          |
| 27) <termino> --> <termino> / <factor>                          |
| 28) <termino> --> <factor>                                      |
| 29) <factor> --> <var>  |
| 30) <factor> --> <cte>  |
| 31) <var> --> <letra>   |
| 32) <var> --> <var> <letra>                                     |
| 33) <var> --> <var> <dígito>                                    |
| 34) <cte> --> <dígito>  |
| 35) <cte> --> <cte> <dígito>                                    |
| 36 a 61) <letra> --> A   B   ...   Z                            |
| 62 a 71) <dígito> --> 0   1   ...   9                           |

# Limitaciones de las especificaciones gramaticales

## Limitaciones prácticas:

- Longitud de identificadores
- Limitaciones de rangos en constantes
- Ubicaciones dentro del código fuente
- Formateo de texto
- Aplicabilidad de operaciones a constantes y variables de diferentes tipos
- etc.

## ¿Es posible desde el BNF hacer lo siguiente?

Los operadores "+", "-", "\*" y "/" pueden aplicarse a los tipos numéricos. El tipo boolean acepta solo "+" y "-". El tipo String acepta solo "+".

**Ejemplo: Lenguaje que pide que los identificadores tengan como máximo 8 caracteres:**

1) <ident> --> <letra>  
2 a 6) <ident> --> <letra> <c1> | <letra> <c2> | <letra> <c3> | <letra> <c4> | <letra> <c5> | <letra> <c6> | <letra> <c7>  
7 y 8) <c1> --> <letra> | <dígito>  
  
9) <c2> --> <c1> <c1>  
10) <c3> --> <c2> <c1>  
11) <c4> --> <c3> <c1>  
12) <c5> --> <c4> <c1>  
11) <c6> --> <c5> <c1>  
11) <c7> --> <c6> <c1>  
  
12 a 40) <letra> --> a | b | c | d | e | f | g | h | i | j | k | l | k | m | n | o | p | q | r | s | t | u | v | w | x | y | z  
41 a 50) <dígito> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

# Limitaciones de las especificaciones gramaticales

Los operadores "+", "-", "\*" y "/" pueden aplicarse a los tipos numéricos. El tipo boolean acepta solo "+" y "-". El tipo String acepta solo "+".

¿Se pueden utilizar estas definiciones de Expresiones, Términos, Factores y Variables?

|                                 |
|---------------------------------|
| 1) ASIG --> variable := EXP     |
| 2) EXP --> EXP + TERMINO        |
| 3) EXP --> EXP - TERMINO        |
| 4) EXP --> TERMINO              |
| 5) TERMINO --> TERMINO * FACTOR |
| 6) TERMINO --> TERMINO / FACTOR |
| 7) TERMINO --> FACTOR           |
| 8) FACTOR --> variable          |
| 9) FACTOR --> cte               |

Debemos definir nuevos No Terminales para cada tipo que se deba diferenciar

## Operaciones con tipos numéricos

```
<EXPRNUM> --> <EXPRNUM>+<TERMINONUM>
<EXPRNUM> --> <EXPRNUM>-<TERMINONUM>
<TERMINONUM> --> <TERMINONUM>*<FACTORTNUM>
<TERMINONUM> --> <TERMINONUM>/<FACTORTNUM>
<TERMINONUM> --> <FACTORTNUM>
<FACTORTNUM> --> <VAR>
<FACTORTNUM> --> <CTE>
```

## Operaciones con tipos booleanos

```
<EXPRBOOL> --> <EXPRBOOL>+<TERMINOBOOL>
<EXPRBOOL> --> <EXPRBOOL>-<TERMINOBOOL>
<TERMINOBOOL> --> <FACTORTBOOL>
<FACTORTBOOL> --> <VAR>
<FACTORTBOOL> --> <CTE>
```

¿Se solucionó el problema?

# Limitaciones de las especificaciones gramaticales

## Operaciones con tipos numéricos

```
<EXPRNUM> -->  
<EXPRNUM>+<TERMINONUM>  
<EXPRNUM> -->  
<EXPRNUM>+<TERMINONUM>  
<TERMINONUM> -->  
<TERMINONUM>*<FACTORMUM>  
<TERMINONUM> -->  
<TERMINONUM>/<FACTORMUM>  
<TERMINONUM> --> <FACTORMUM>  
<FACTORMUM> --> <VAR>  
<FACTORMUM> --> <CTE>
```

```
<VARNUM> --> N_<letra> | <VARNUM><letra> | <VARNUM><DIGITO>  
<VARBOOL> --> B_<letra> | <VARBOOL><letra> | <VARBOOL><DIGITO>
```

## Operaciones con tipos booleanos

```
<EXPRBOOL> -->  
<EXPRBOOL>+<TERMINOBOOL>  
<EXPRBOOL> -->  
<EXPRBOOL>-<TERMINOBOOL>  
<TERMINOBOOL> --> <FACTORBOOL>  
<FACTORBOOL> --> <VAR>  
<FACTORBOOL> --> <CTE>
```

```
int x,y;  
bool a,b;  
x = x*y;  
b = b*a;
```

Los nombres "a" y "b" son reducidos al no terminal <VAR>, por lo tanto b\*a es válido

**Es necesario diferenciar sintácticamente las variables de los tipos int y bool**

**Reemplazar <VAR> por <VARNUM> Y <VARBOOL>**

**Este tipo de soluciones obliga a ir a una definición por extensión, en lugar de por comprension**

```
int N_x,N_y;  
bool B_a,B_b;  
N_x = N_x*N_y;  
N_b = B_b*B_a;
```

## Lenguajes con características similares según el comienzo de las variables:

- FORTRAN: letras de i a k -> son de tipo entero
- PERL: con \$ -> son de estructura escalar, con % -> son tablas de hash, con @ son de estructura tipo arreglo.
- RUBY: con una letra -> locales, con @ -> variables de instancia, con @@ variables de clase, con \$ variables globales.

# Limitaciones esenciales de las gramáticas

¿Es posible limitar ciertas operaciones sobre variables de tipos definidos por el usuario?

**Sistemas como BNF y similares no pueden contemplar elementos no definidos en el momento de definición del lenguaje**

¿Es posible realizar el control si una variable fue declarada?

**Sistemas como BNF y similares definen gramáticas libres de contexto. Es necesario realizar una verificación semántica.**

Algunas herramientas que utilizan mecanismos de definición de sintaxis:



<https://www.antlr.org/>



**YACC & BISON**

<http://dinosaur.compilertools.net/>



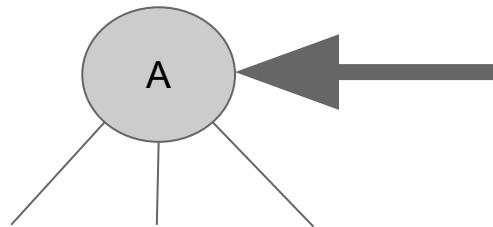
**LLVM**

<https://llvm.org/>

# Árbol de Parsing

**Lo construye el compilador para:**

- Verificar que el programa esté correctamente escrito
- Generar el código intermedio/ejecutable a partir del árbol



Objeto de la  
gramática

**Dos enfoques principales:**

- **Parsing Descendente**
- **Parsing Ascendente**

... ... ... ... ... ...



Fragmento  
del  
programa

**En Lenguajes lo usamos para  
verificar las gramáticas**

# Árbol de Parsing

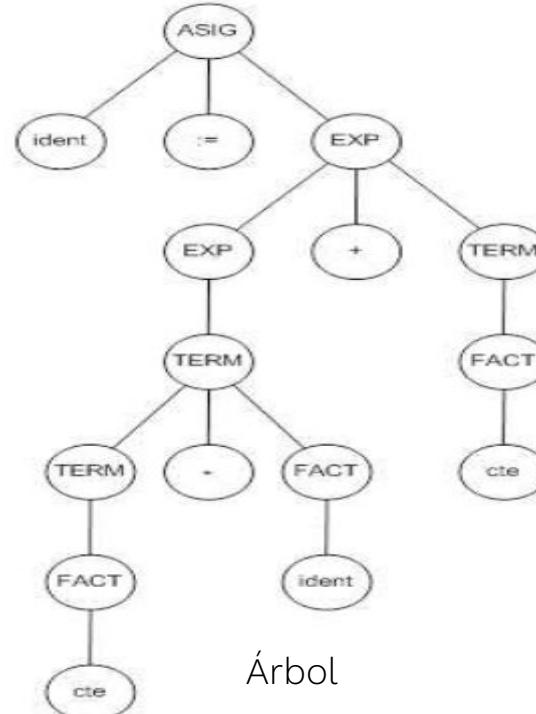
## Ejemplo:

|                                 |
|---------------------------------|
| 1) ASIG --> variable := EXP     |
| 2) EXP --> EXP + TERMINO        |
| 3) EXP --> EXP - TERMINO        |
| 4) EXP --> TERMINO              |
| 5) TERMINO --> TERMINO * FACTOR |
| 6) TERMINO --> TERMINO / FACTOR |
| 7) TERMINO --> FACTOR           |
| 8) FACTOR --> variable          |
| 9) FACTOR --> cte               |

Gramática

id = cte \* id + cte

Programa



Árbol

# Parsing descendente

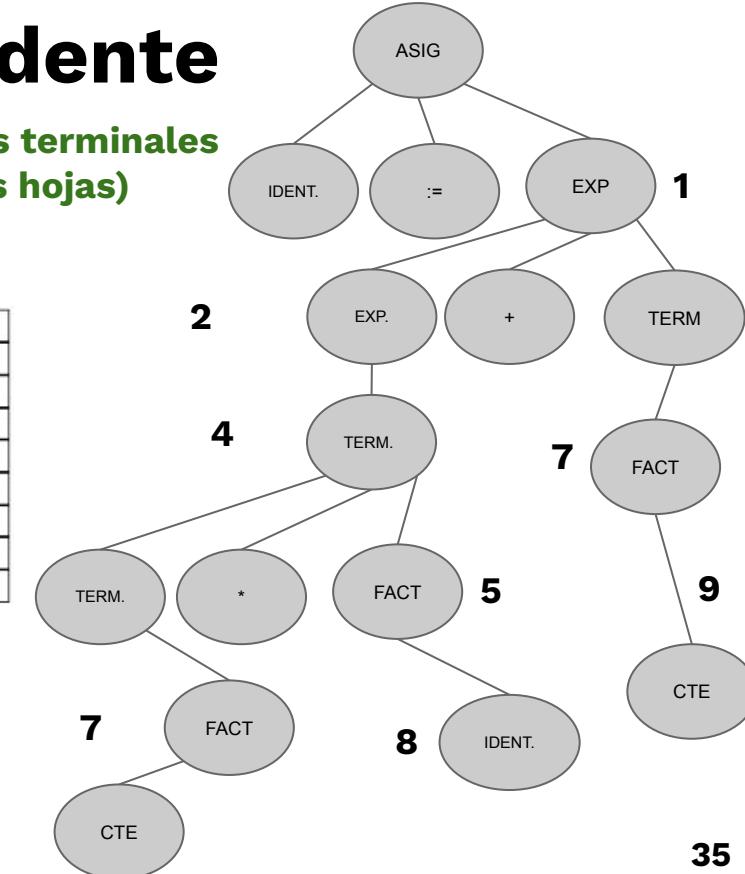
Desde la hipótesis hasta los terminales  
(desde la raíz hasta las hojas)

## Algoritmo:

1. Tomar la hipótesis
2. Buscar una regla
3. Reemplazar la hipótesis por la definición
4. Identificar los nuevos no terminales
5. Volver al paso 2

|                          |
|--------------------------|
| 1) ASIG --> ident := EXP |
| 2) EXP --> EXP + TERM    |
| 3) EXP --> EXP - TERM    |
| 4) EXP --> TERM          |
| 5) TERM --> TERM * FACT  |
| 6) TERM --> TERM / FACT  |
| 7) TERM --> FACT         |
| 8) FACT --> ident        |
| 9) FACT --> cte          |

Id := cte \* id + cte



# Parsing Ascendente

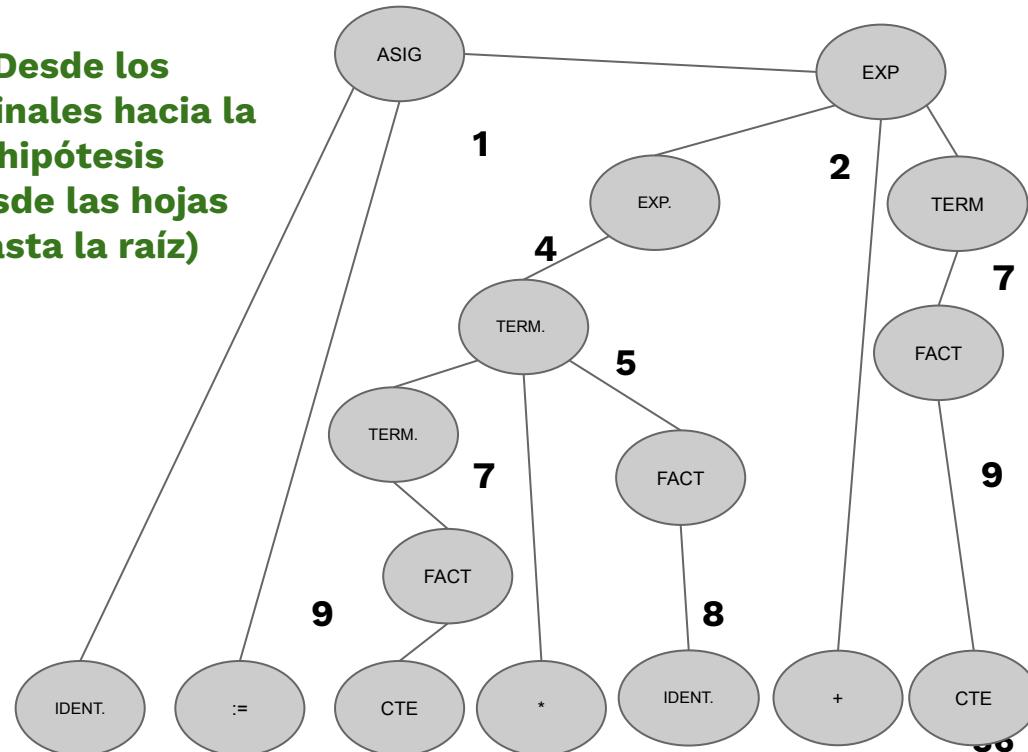
## Algoritmo:

1. Seleccionar un terminal
2. Encontrar una regla donde esté definido
3. Reemplazar el terminal por el no terminal
4. Seleccionar un conjunto de terminales, no terminales o ambos
5. Volver al paso 2

- |                          |
|--------------------------|
| 1) ASIG --> ident := EXP |
| 2) EXP --> EXP + TERM    |
| 3) EXP --> EXP - TERM    |
| 4) EXP --> TERM          |
| 5) TERM --> TERM * FACT  |
| 6) TERM --> TERM / FACT  |
| 7) TERM --> FACT         |
| 8) FACT --> ident        |
| 9) FACT --> cte          |

**Desde los terminales hacia la hipótesis  
(desde las hojas hasta la raíz)**

Id := cte\*id + cte



# Árbol de Parsing

- **Se puede construir al menos un árbol** → **El programa está bien escrito según la gramática**
- **No se puede construir al menos un árbol** → **El programa no está bien escrito según la gramática**
- **Existe más de un árbol de parsing** → **La gramática es ambigua**

El compilador/intérprete explora todos los caminos posibles para construir el árbol

El compilador/intérprete generalmente utiliza el árbol de parsing para generar variables auxiliares.

# Precedencia

Órden en el cual se realizan las operaciones

a := b \* c + d / e

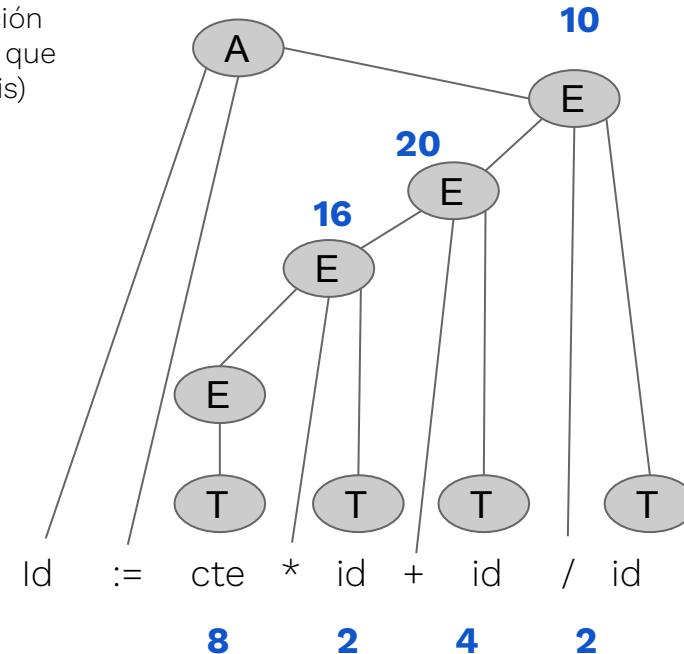
Se quiere que la multiplicación y la división se realice antes que la suma (sin usar paréntesis)

1. A → <id> :=<E>
2. E → <E>+<T>
3. E → <E>-<T>
4. E → <E>\*<T>
5. E → <E>/<T>
6. E → <T>
7. T → <id>
8. T → <cte>
  
- ...

¿Esta gramática permite hacerlo?

Esta gramática  
no tiene orden de  
precedencia de  
operadores

¿Por qué?



# Precedencia

Órden en el cual se realizan las operaciones

a := b \* c + d / e

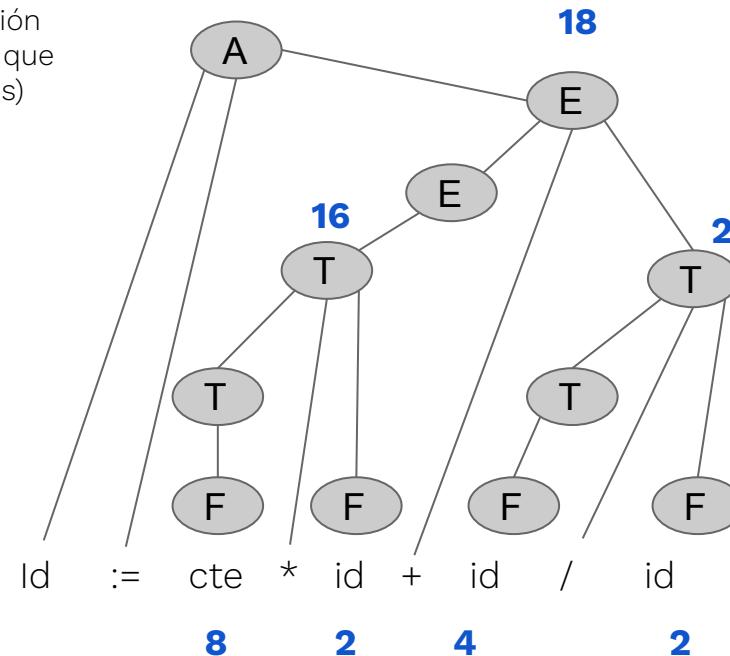
1. A  $\rightarrow$  <id> :=<E>
2. E  $\rightarrow$  <E> + <T>
3. E  $\rightarrow$  <E> - <T>
4. E  $\rightarrow$  <T>
5. T  $\rightarrow$  <T>\*<F>
6. T  $\rightarrow$  <T>/<F>
7. T  $\rightarrow$  <F>
8. F  $\rightarrow$  <id>
9. F  $\rightarrow$  <cte>
- ...

¿Esta gramática permite hacerlo?

Se quiere que la multiplicación y la división se realice antes que la suma (sin usar paréntesis)

Esta gramática tiene orden de precedencia de operadores

¿Por qué?



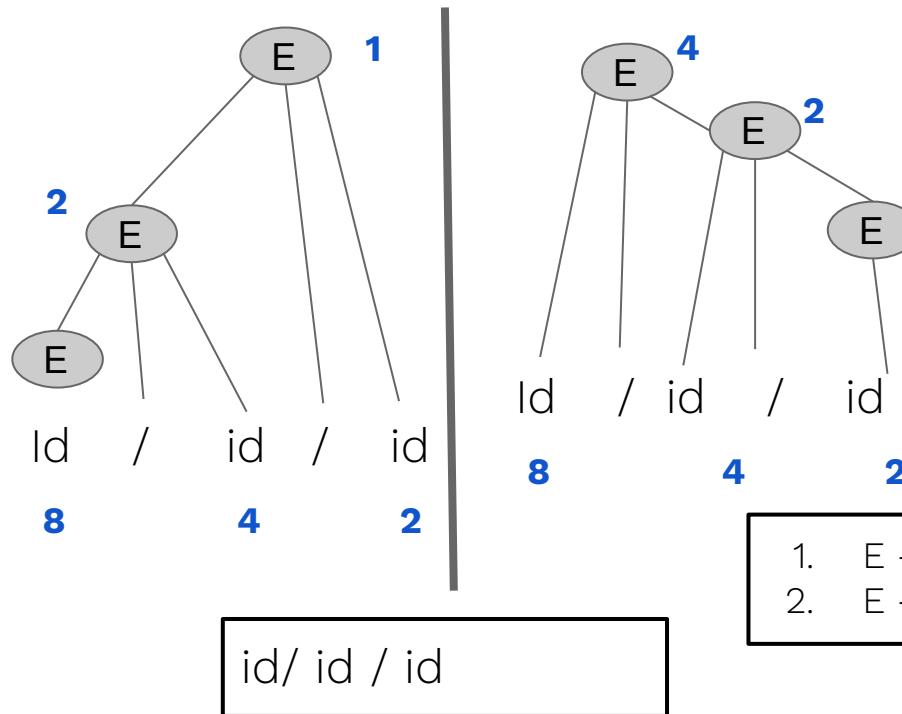
# Recursividad

## Efectos de la recursividad de la gramática

1.  $E \rightarrow E / id$
2.  $E \rightarrow id$

¿La recursividad afecta la asociatividad?

- Recursiva a derecha -> operaciones de der. a izq.
- Recursiva a izquierda -> operaciones de izq. a der.

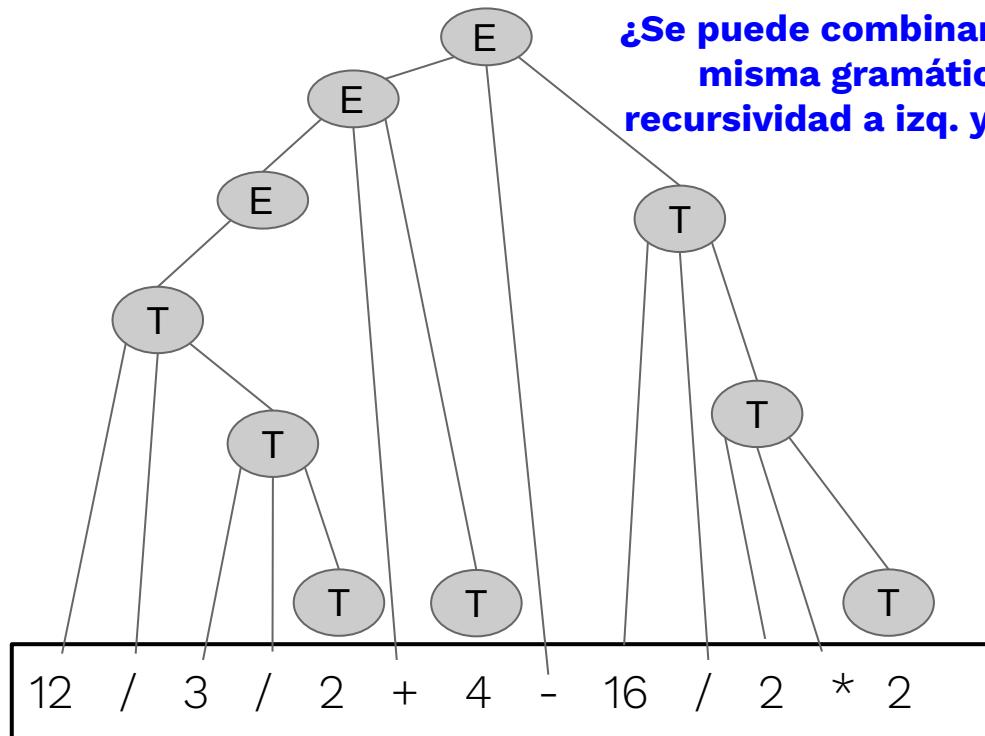


1.  $E \rightarrow id / E$
2.  $E \rightarrow id$

# Recursividad Combinada

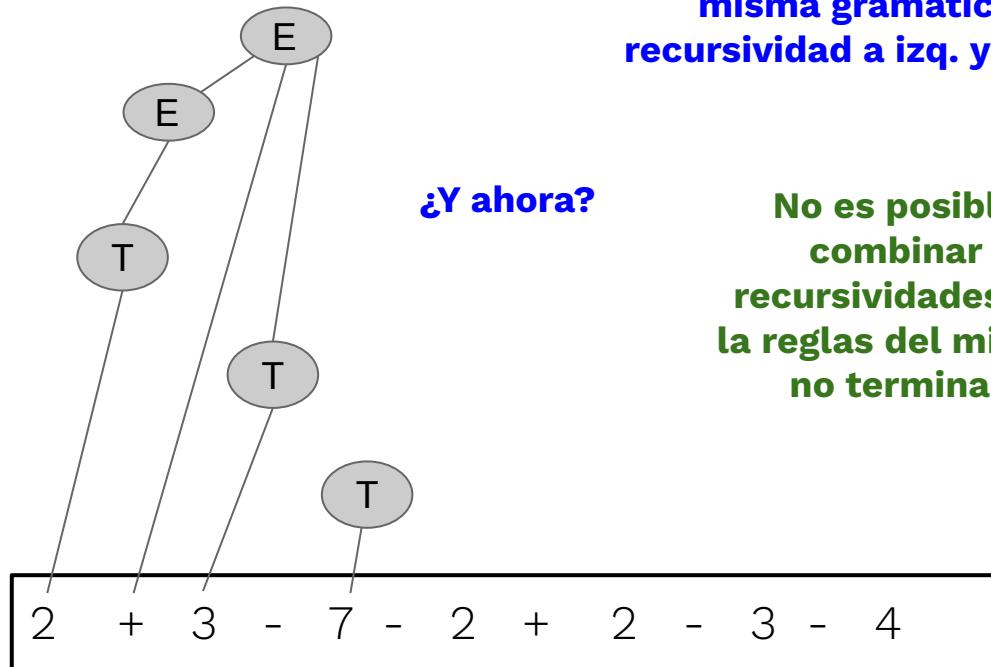
1.  $E \rightarrow E + T$
2.  $E \rightarrow E - T$
3.  $E \rightarrow T$
4.  $T \rightarrow \text{cte} * T$
5.  $T \rightarrow \text{cte} / T$
6.  $T \rightarrow \text{cte}$

- Recursiva a derecha -> operaciones de der. a izq.
- Recursiva a izquierda -> operaciones de izq. a der.



# Recursividad Combinada

- 1.  $E \rightarrow E + T$
- 2.  $E \rightarrow T - E$
- 3.  $E \rightarrow T$
- 4.  $T \rightarrow T * \text{cte}$
- 5.  $T \rightarrow T / \text{cte}$
- 6.  $T \rightarrow \text{cte}$



¿Se puede combinar en la misma gramática recursividad a izq. y der.?

¿Y ahora?

No es posible combinar recursividades en la reglas del mismo no terminal

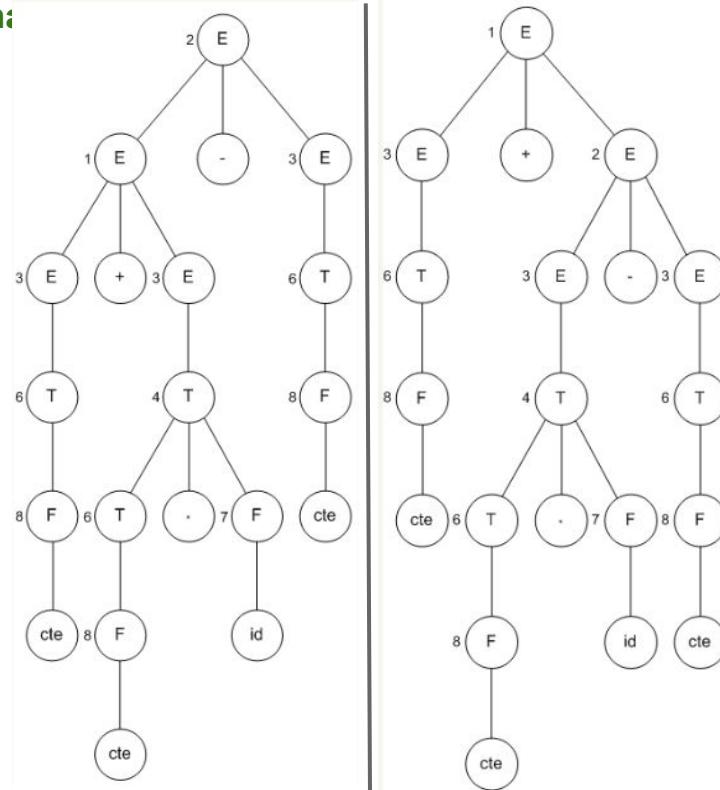
# Ambigüedad

Una gramática es ambigua si es posible construir más de un árbol de parsing para un programa

1.  $E \rightarrow E + E$
2.  $E \rightarrow E - E$
3.  $E \rightarrow T$
4.  $T \rightarrow T * F$
5.  $T \rightarrow T / F$
6.  $T \rightarrow F$
7.  $F \rightarrow id$
8.  $F \rightarrow cte$

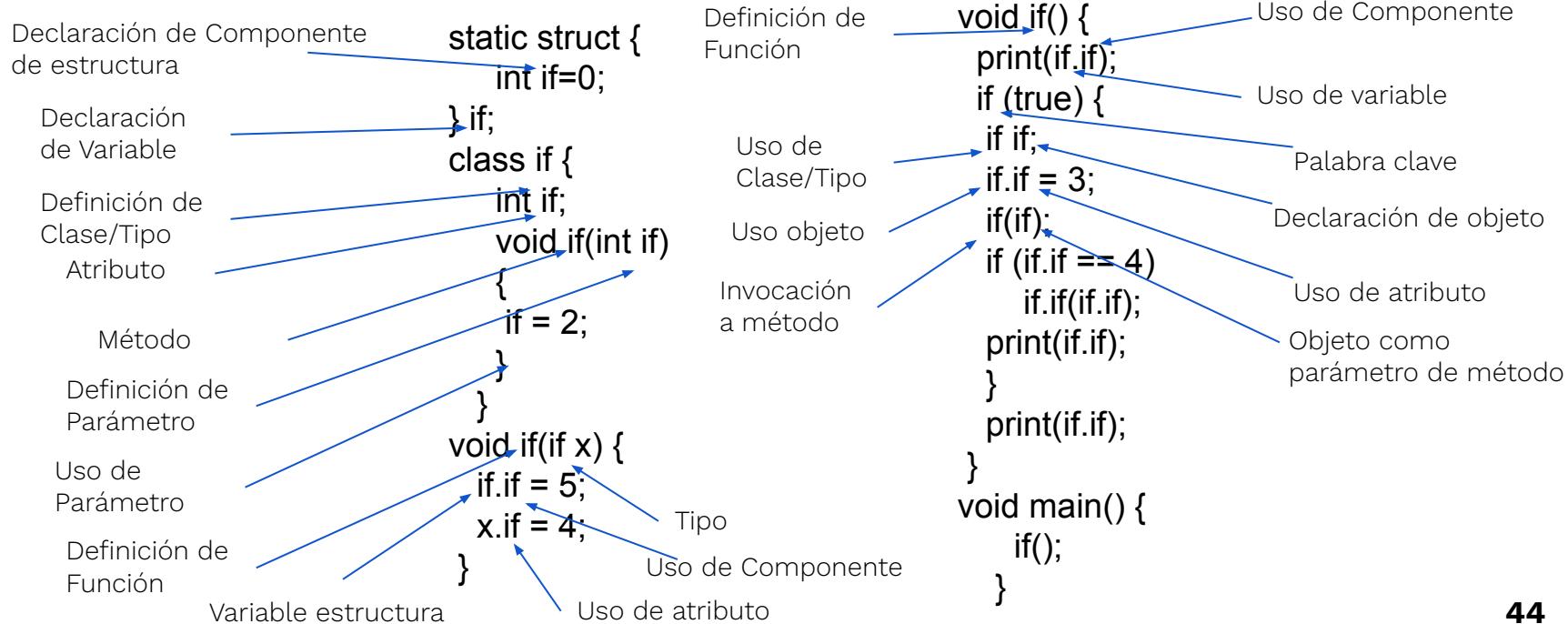
id + cte \* id - id

- No es posible afirmar si un programa está bien escrito.
- Un compilador produciría dos programas diferentes para el mismo código !
- Los resultados de cada programa serían diferentes para los mismos datos!



# Ejercicios

Clasificar cada ocurrencia de la cadena “if” según a cuál tipo de elemento del lenguaje corresponda: tipo, variable, componente de estructura, atributo, nombre de función, nombre de método, palabra clave, constante, etc



# Ejercicios

```
1.start: programa
2.programa: instrucion
   | bloque
3.instrucciones: instrucion | instrucciones instrucion
4.bloque: "_comienza_bloque_" instrucciones "_termina_bloque_"
5.instrucion: declaracion ";"
   | invocacion ";"
   | asignacion ";"
   | seleccion ";"
   | iteracion ";"
6.declaracion: "_se_declaran_con_el_tipo_" tipo "_las_variables_"
listavariables
7.tipo: "entero" | "largo" | "flotante" | "doble" | "cadena"
8.listavariables: NAME | listavariables NAME
9.invocacion: "_se_llama_a" NAME "_con_parametros_"
"_abre_parentesis_" expresion "_cierra_parentesis_"
10.asignacion: NAME "_se_vuelve_" expresion
11.seleccion: "_si_" comparacion "_entonces_" bloque ";" "_sino_"
bloque
12.iteracion: "_mientras_que_" comparacion bloque
13.expresion: expresionsimple | comparacion
14.comparacion: expresionsimple comparador expresionsimple
15.comparador: "_es_menor_que_" | "_es_mayor_que_" |
"_es igual que_" | "_es_mayor_o_igual_que_" |
"_es_menor_o_igual_que_" | "_es_distinto_"
16.expresionsimple: termino
```

Dada la siguiente gramática y el programa correspondiente, indicar para cada línea marcada, la última regla que se aplica para aceptar la instrucción.

```
17.termino: factor
   | termino "_mas_" factor
   | termino "_menos_" factor

18.factor: atomo
   | factor "_por_" atomo
   | factor "_dividido_" atomo
   | invocacion

?atomo: NUMBER
   | "_menos_" atomo
   | NAME
```

# Ejercicios

```
_comienza_bloque_
    _se_declaran_con_el_tipo_ entero _las_variables_ a b ; // 1
    _se_declaran_con_el_tipo_ largo _las_variables_ c d ;
        a _se_vuelve_ b ;                                // 2
        b _se_vuelve_ c _mas_ 1 ;                      // 3
        d _se_vuelve_ _se_llama_a_ f _con_parametros_ _abre_parentesis_ a
            _cierra_parentesis_ ;                         // 4
    _si_ a _es igual que_ 2 _entonces_
        _comienza_bloque_
            a _se_vuelve_ 1 ;
        _termina_bloque_ ;
        _sino_
            _comienza_bloque
                _ a _se_vuelve_ 2 ;
            _termina_bloque_ ;
    _mientras_ que_ a _es mayor o igual que_ 1           // 6
        _comienza_bloque_
            d _se_vuelve_ d _mas_ 3 ;
        _termina_bloque_ ;
_termina_bloque_
```

# Semántica

**¿Qué es la semántica?**

**El estudio del significado de las sentencias**

**¿Cómo se describe?**

**Manuales de referencia  
Traductores  
Definiciones formales**

**¿Qué vamos a hacer?**

**Recurrir a la pragmática, reconociendo la influencia de la pragmática en la semántica**

**Compiladores, Intérpretes, Entornos de Ejecución (S.O., Arquitectura), etc.**

# Ligadura

¿Qué es el binding o ligadura?

“El estudio del momento preciso en el que una propiedad de un elemento del lenguaje es conocida

- Valor
- Almacenamiento
- Tipo
- Alcance
- Nombre
- Estructura
- ....
- Variable
- Función
- Método
- Parámetro
- Bloque
- Tipo
- ...

¿Cuáles son los momentos posibles?

Ejemplos:

- Binding / Ligadura del tipo de las variables es el momento en el que se conoce el tipo de la variable
- Binding / Ligadura de almacenamiento de las variables es el momento en el que se conoce la dirección de la celda donde se almacenan las variables

# Ligadura

## Estático

%i BASIC  
i FORTRAN  
@a Perl

- En BASIC\* las variables seguidas con % son de tipo entero.
- En FORTRAN\* las variables i, j, k, etc. son enteros.
- En Perl las variables que comienzan con @ tienen estructura de tipo arreglo.

## Tiempo de definición del lenguaje

## Estático

i : integer; Pascal  
int i C/C++, Java

- En Pascal, C, C++, Java, etc. el tipo de i es entero

i := 123 en Go  
let a = 1; en Rust

- **¿Qué sucede en Go, Rust y otros lenguajes?**  
[\(ver ejemplos\)](#)

## Tiempo de escritura del programa

## Dinámico

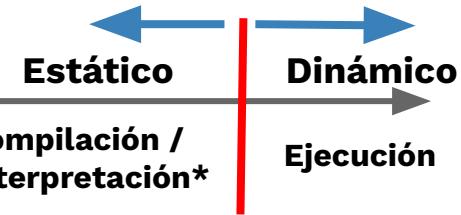
```
if (a==1): Python
    b = "Hola"
else:
    b = 2.0
print (b)
```

```
if. T do. i=: 3 J
else. i=: a end.
```

Hay que esperar a que se ejecuten las instrucciones para conocer el tipo de las variables

## Tiempo de ejecución

# Línea de Tiempo



| Diseño / Definición del lenguaje   | Construcción del compilador / intérprete  | Escritura del programa   | Compilación / Interpretación*   | Ejecución   |
|--|---|--|---|---|
| <p>En Pascal, el operador de suma se hace con el símbolo “+”.</p> <p>La suma tiene una precedencia menor que la multiplicación</p> | <p>El compilador de Borland C++ admite que la función main devuelva el tipo void mientras que Dev C++ solo admite el tipo int</p> | <p>En ADA se puede definir con una palabra clave el formato de representación interna de las variables numéricas</p> | <p>Algunos compiladores en general almacenan los valores de las constantes en celdas de memoria y otros lo introducen dentro del ejecutable</p> | <p>La ubicación en memoria de las variables no estáticas.</p> |

## Sintaxis

Una situación puede implicar varias etapas y tener varios aspectos al mismo tiempo.

## Semántica

¿Puede definirse un aspecto sintáctico en ejecución?  
¿Puede definirse un aspecto pragmático en el momento de definición del lenguaje?

# Propiedades de las Variables

## ¿Qué es una variable?

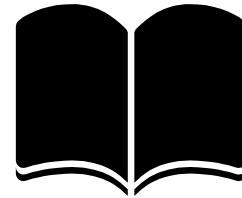
Lenguajes imperativos: **Variable**: <name, scope, type, l\_value, r\_value> [1]  
Lenguajes funcionales: **Variable**: <type, atomics, symbol\_value>

| Propiedad      | Estático | Dinámico |
|----------------|----------|----------|
| Valor          |          |          |
| Tipo           |          |          |
| Alcance        |          |          |
| Almacenamiento |          |          |

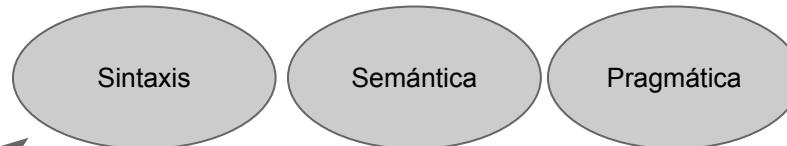
[1] Oren Patz, Michael J. Main, "Programming language concepts", Addison Wesley Longman, New York, 1998, página 69. Consultado el 16/01/2015, al 1998.

# Lenguajes de Programación I -

## Clase 2: Semántica - Binding



¿Qué vimos hasta ahora?



- Aspecto sintáctico de los lenguajes de programación.
- No por extensión, sino por comprensión.
- Cómo se define formalmente la manera de escribir los elementos del lenguaje y las sentencias que componen un programa

- Limitaciones prácticas y esenciales
- Precedencia
- Asociatividad
- Recursividad
- Ambigüedad

**Vamos a continuar con la semántica**

# Propiedades de las Variables

## ¿Qué es una variable?

Lenguajes imperativos: **Variable**: <name, scope, type, l\_value, r\_value> [1]  
Lenguajes funcionales: **Variable**: <type, atomics, symbol\_value>

| Propiedad      | Estático | Dinámico |
|----------------|----------|----------|
| Valor          |          |          |
| Tipo           |          |          |
| Alcance        |          |          |
| Almacenamiento |          |          |

[1] Oren Patz, Michael J. Main, "Programming language concepts", Addison Wesley, New York, 1998, página 69. Consultado el 16/01/2015, al 1998.

# Ligadura de Valor

El patrón de bits almacenado en la celda asociada a la variable

Las variables son  
“variables”

Hay algunos lenguajes en  
donde existen  
restricciones a estos  
cambios

→ Su valor cambia en  
ejecución →

¿Tiene binding estático de  
valor el lenguaje?

Binding dinámico

```
float x;  
x = 1.0;  
input(x);  
x = rand();
```

Constantes

Constantes literales

```
int a = 123;
```

Constante  
literal

Constantes simbólicas

```
#define b 789  
#define pi 3.141592
```

Constantes  
propriamente dichas

```
const d = 234;
```

(ver ejemplos en C)

## Lenguajes de Programación I - Clase 2: Valor

### Inmutabilidad

```
fn main() {  
    let x = 5; Rust  
    ...  
    x = 7;  
}  
//Error de compilación
```

```
let mut x = 5;
```

Esto se relaciona además con Ownership y Borrowing

¿Qué pasa en este caso?

Cuasi-constantes o  
pseudo-constantes

Por defecto todas las variables tienen binding estático de valor.  
Para que sean “variables” deben ser declaradas como mutables.  
En los punteros es posible indicar la mutabilidad de lo apuntado.

```
let x = 1; Rust  
let mut y = 2;  
let p1 = &x;  
let p2 = &y;  
let mut p3 = &x;  
let mut p4 = &y;
```

```
let p5 = &mut x; X Rust  
let mut p6 = &mut x; X  
let p7 = &mut y;  
let mut p8 = &mut y;  
*p7 = 5; *p3 = 3; X  
*p8 = 6; *p4 = 4; X
```

```
X : INTEGER  
...  
X := ...  
Y: constant FLOAT := 2.0 * X;
```

# Ligadura de Valor

## Cuasi-Constantes

```
class P {  
    public readonly int x = 3;          C# 7.0  
    public P(int p) {  
        x = p;  
    }  
    public m(int p) {  
        x = p;   
    }  
    ...  
    P p1 = new P(4);  
    ...
```

¿Qué tipo de binding de valor se tiene en estos casos?

Binding dinámico

```
class P {  
    public final int x;  
    public P(int xi) {  
        x = xi;  
    }  
    public void m1() {  
        x = 3;   
    }  
    ...  
    P p1 = new P(2);  
    ...
```

Java 8

Varios lenguajes con cláusulas de tipo final o readonly.  
Generalmente los orientados a objetos

# Ligadura de Valor : Inmutabilidad

```
a: int;      CLU  
a=2;  
a=3;
```

Lenguajes Funcionales → Inmutabilidad

Esta instrucción no cambia el valor de a.

Se crea una nueva celda de memoria, se almacena un 3 y la celda donde se almacenó a cambia a la nueva (\*)

```
s1 = "A" Python  
s2 = s1  
s1 = "a" (*)  
print (s1)  
s1 = s1 + "B" (*)  
print (s1)  
print (s2)
```

```
class C: Python  
    i=1  
    c1 = C  
    print (c1.i)  
    c2 = c1;  
    c1.i = 3  
    print (c2.i)  
    print (c1.i)
```

```
String s1 = "A"; Java  
String s2 = s1;  
System.out.println(s1);  
s1 = s1 + " B"; (*)  
System.out.println(s1);  
System.out.println(s2);  
String s3 = "C";  
s3.concat(" B"); (*)  
System.out.println(s3);
```

¿Qué pasa en este caso?

Muchos lenguajes tienen tipos cuyas variables son inmutables (Por ej. en Java las denominadas wrapper classes: Integer, Float, String, etc.)

## Ligadura de Valor

| Propiedad      | Estático                              | Dinámico            |
|----------------|---------------------------------------|---------------------|
| Valor          | Muy pocos lenguajes funcionales puros | Practicamente todos |
| Tipo           |                                       |                     |
| Alcance        |                                       |                     |
| Almacenamiento |                                       |                     |

Para que un lenguaje tenga una propiedad estática o dinámica, la misma debe poder aplicarse a todos los elementos

# Ligadura de Tipo

El conjunto de valores posibles que puede adquirir la variable

A\$ BASIC\*

int i = 3; C/C++

i FORTRAN\*

ClaseA a1 = new A(); Java  
a1 = b1; a1 = new B();

double j = 2.3;  
int i = j;  
C/C++

double j = 2.3; Java  
int i = j; 

let v = new Persona("Pepe");  
v = new Comprobante(123);  
v = w; Javascript

m = "A" Ruby  
m = n

\$t = true; PHP  
\$t = \$p;

Pragmática: Algunos compiladores dan error y otros hacen una conversión.

Mirando el programa y conociendo como funciona el lenguaje se pueden saber los tipos de las variables

**Estático**

Mirando el programa no se puede determinar los tipos de las variables

**Dinámico**

# Ligadura de Tipo: Inferencia

Mecanismo para obviar el nombre del tipo en algunas instrucciones

```
auto x = 3; C/C++
```

```
auto j = 2.3;  
auto i = j;  
C/C++
```

```
let x; Rust  
let y = 1;  
x = y;
```

```
var i int Go  
var j = i  
k := j
```

```
var x = "Hola"; C#  
int i = j;
```

¿A cuáles tipos pertenecen las variables del lado izquierdo?

Evita el uso de var para variables con inicialización

El compilador infiere el tipo de las variables

Se reserva el nombre x para este ámbito

Mirando el programa y conociendo como funciona el lenguaje se pueden saber los tipos de las variables

Estático

¿Existe la inferencia de tipos en un lenguaje dinámico?

```
a = 1 Python  
a = 2.3  
a = "Hola"  
class C:  
    x = 1  
a = C()
```

El lenguaje debe ir registrando los cambios de tipos de a

```
let x : number = 1; Typescript  
let y : string = 'Hola';  
let z = x * y;
```

Inferencia

Dinámico

## Ligadura de Tipo

| Propiedad      | Estático   | Dinámico  |
|----------------|--|---|
| Valor          | Muy pocos  | Practicamente todos   |
| Tipo           | <b>Tipo Algol:</b> Pascal, C, C++, Java, Swift, Kotlin, Rust, Go, etc. | <b>APL, J, LISP, Python, PHP, Perl, Dart, Hack, Javascript, Typescript*</b> |
| Alcance        |  |   |
| Almacenamiento |  |   |

Para que un lenguaje tenga una propiedad estática o dinámica, la misma debe poder aplicarse a todos los elementos

# Ligadura de Alcance

Cuáles instrucciones pueden usar una variable

PASCAL

```
Procedure p;  
var x : Integer;  
begin  
(*instrucciones que  
pueden usar x*)  
end;
```

```
class A { Java  
private int x = 1;  
public m() {  
int z = 2; }  
}
```

Dentro  
de m

Estático

w = 4 Python 3

```
def f():
```

```
    x = 1
```

```
    def g():
```

```
        z = 2
```

```
        z = x
```

```
        z = w
```

```
g()
```

```
f()
```

w = 4 Python 3

```
def f():
```

```
    x = 1
```

```
    def g():
```

```
        z = 2
```

```
        x = z
```

```
        w = z
```

```
g()
```

```
f()
```

¿Cuáles variables x  
y w se utilizan?

w = 4 Python 3

```
def f():
```

```
    x = 1
```

```
    def g():
```

nonlocal x  
global w

```
    z = 2
```

```
    x = z
```

```
    w = z
```

```
g()
```

```
f()
```

¿Qué pasa si quiero usar  
x y w del lado izq.?

¿Y ahora?

Mirando el programa y conociendo  
como funciona el lenguaje se puede  
saber el alcance de las variables

# Lenguajes de Programación I - Clase 2: Alcance

## Cuáles instrucciones pueden usar una variable

```
10 INPUT A BASIC
20 B = 5
30 IF (A>4) C=3
40 D=B+C
```

La variable C se crea solo si se cumple la condición

No se puede saber si C está al alcance o no

```
f1(); Perl
print $x;
sub f1 {
    if (...) $x = 3;
    print $x;
}
```

¿Qué imprime este programa?

¿Donde se crea \$x?  
¿y ahora?

Mirando el programa no se puede saber el alcance de las variables

```
f1(); Perl
print $x;
sub f1 {
    local $x = 3;
    print $x;
}
```

¿Qué sucede ahora?

```
f1(); Perl
print $x;
sub f1 {
    my local $x = 3;
    print $x;
    f2();
}
sub f2 {print $x;}
```

¿Qué sucede al cambiar local por my?

**La secuencia de invocación**

**Dinámico**

¿Qué se puede deducir ejecutando este programa?

## Lenguajes de Programación I - Clase 2: Alcance

```
Perl  
if (rand()>0.5)  
    f1();  
else f2();  
print $y;  
  
sub f1 { $y = 3;  
}  
  
sub f2 { my $y = 4;  
}
```

Perl

¿Cuál  
variable y se  
utiliza?

- Es posible cambiar el ámbito en cualquier momento
- Es posible crear ámbitos nuevos en tiempo de ejecución

### Casos de estudio

J

...  
cocurrent'A' NB. Creo un ámbito A  
x=:3  
y=:5  
cocurrent'B' NB. Creo un ámbito B  
x=:10  
y=:20  
if. y <: z do. cocurrent'A' end.

¿Cuáles  
variables x e  
y se utilizan?

Cambia el  
ámbito  
actual a 'A'

# Ligadura de Alcance

Closures: Unidades que se ejecutan fuera del lugar donde fueron definidas

Python

```
def a():
    x = 2
    def b():
        y = 3
        print x+y
    return b
z=a()
z()
```

b es un closure

x = 2  
def b()  
y = 3  
print x+y  
return b

z=a()

**b es un closure**

¿Cambia el alcance en el caso de los closures?

¿Cuáles variables se utilizan?

Pero la variable x desaparece cuando desaparece a

El closure se asigna a una variable

Se está ejecutando el closure fuera de a (a través de z)

¿Cómo hacen los lenguajes para almacenar las variables que deberían desaparecer?

Esta función es un closure

Perl

```
sub a() {
    $y = 2;
    my $x = 1;
    return sub {
        print $x + $y;
    }
}
$z = a();
&$z();
$y = 3;
&$z();
```

# Ligadura de Alcance

## ¿Para qué sirven los closures?

- Implementar funciones que cumplan con la definición de función matemática
- Pasarle al Sistema Operativo una función que atienda una señal
- Retornar una función desde una función o método
- Utilizar funciones de alto orden (funciones que reciben y utilizan funciones)
- Preservar el estado de la ejecución en un momento determinado

**Prácticamente todos los lenguajes poseen closures**

Lisp, C++, Perl, Java, Python, Ruby, Lua, PHP, JavaScript, Delphi, C#, Go, Dart, Rust, Swift, Hack, etc.

**Implicancias de los closures**

**Sensibilidad a la historia**

Si pueden alterar las variables capturadas

**Efectos Laterales**

Si pueden alterar variables del entorno donde se ejecutan

## Ligadura de Alcance

| Propiedad             | Estático   | Dinámico   |
|-----------------------|--|--|
| <b>Valor</b>          | Muy pocos  | Practicamente todos  |
| <b>Tipo</b>           | Tipo Algol: Pascal, C, C++, Java, Swift, Kotlin, Rust, Go, etc.                | APL, J, LISP, Python, PHP, Perl, Dart, Hack, Javascript, Typescript* |
| <b>Alcance</b>        | <b>Tipo Algol: Pascal, C, C++, Java, Swift, Kotlin, Rust, Go, Python, etc.</b> | <b>APL, J, BASIC, Perl</b>   |
| <b>Almacenamiento</b> |  |  |

Para que un lenguaje tenga una propiedad estática o dinámica, la misma debe poder aplicarse a todos los elementos

# Ligadura de Almacenamiento

Almacenamiento: El lugar de memoria donde está la variable

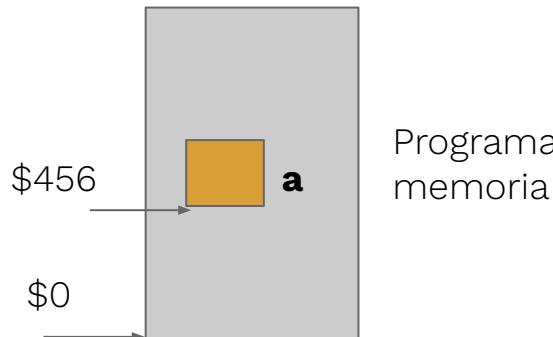
**FORTRAN**  
10 INTEGER A  
20 A = 7

**ASSEMBLER**  
MOV AX, 7  
MOV \$F356,AX

**C/C++**  
void main() {  
 static int a;  
 a = 7;  
}

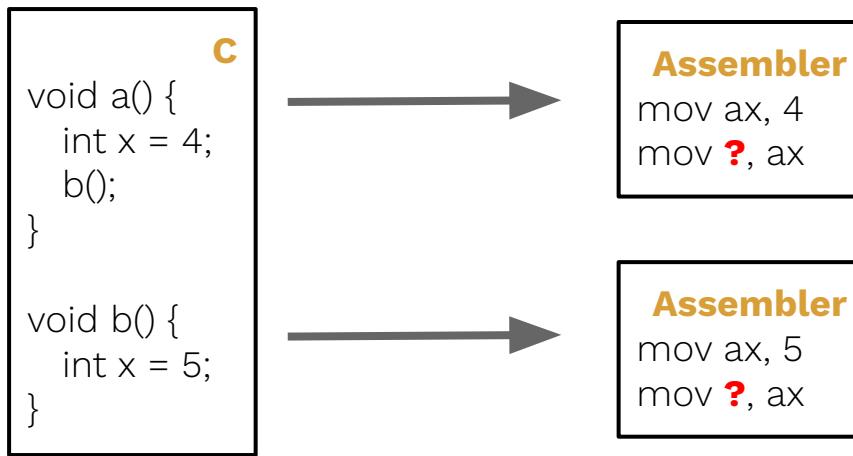
**Java**  
class C {  
 static int a;  
 void m() {  
 a = 9;  
 }  
}

- El lenguaje asoció la variable a una dirección
- Se usa la variable a través de su dirección
- La variable no se va a mover durante toda la ejecución del programa



En FORTRAN todas las variables son de este tipo, en C/C++, Java, etc. solo las indicadas por el programador sintácticamente

# Ligadura de Almacenamiento



La mayoría de los lenguajes funcionan de esta manera: C, C++, Java, Pascal, Delphi, Rust, Go, Kotlin, C#, etc.

- **Las variables aparecen cuando la unidad que las contiene se empieza a ejecutar**
- **Las variables desaparecen cuando la unidad que las contiene termina de ejecutarse (\*)**

Además, puede haber más de una copia de la variable al mismo tiempo

# Ligadura de Almacenamiento

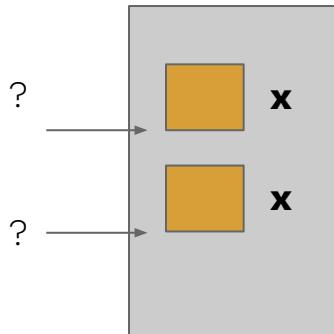
## Recursividad

```
void recursiva() {  
    int x = 4;  
    ...  
    recursiva();  
}
```

Assembler  
mov ax, 4  
mov ?, ax

Debe haber dos o más  
juegos de variables

¿Esto se puede implementar  
en lenguajes estáticos?



Programa en memoria

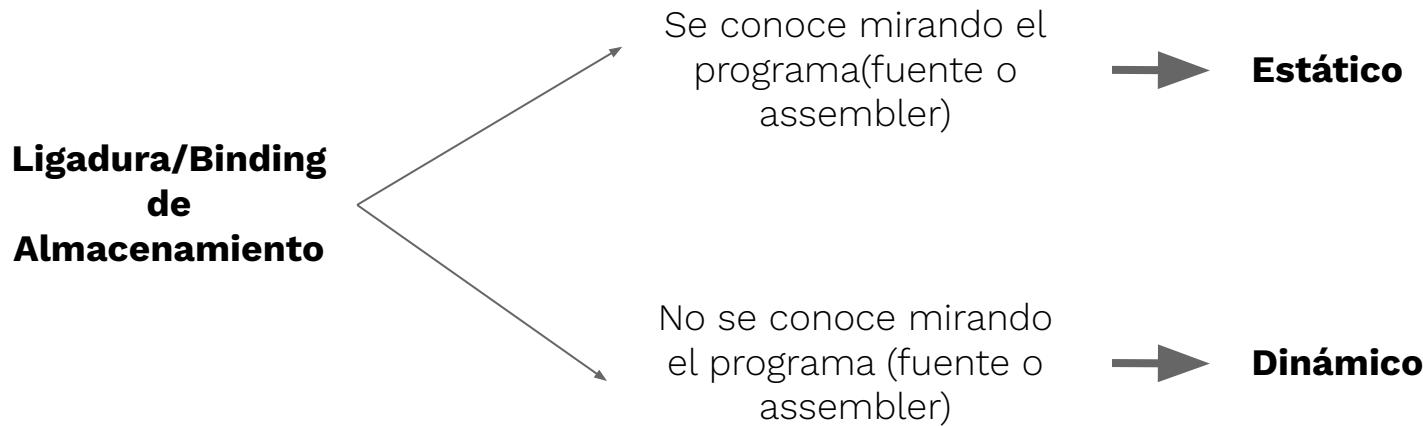
## Problemas a resolver

- ¿Cómo se administran los diferentes juegos de variables?
- ¿Cómo se hace para que las variables aparezcan y desaparezcan de memoria

Se Necesita algo más complejo que direcciones fijas

# Ligadura de Almacenamiento

**Almacenamiento:** El lugar de memoria donde está la variable



## Ligadura de Almacenamiento

| Propiedad             | Estático  | Dinámico   |
|-----------------------|---|--|
| <b>Valor</b>          | Muy pocos   | Practicamente todos  |
| <b>Tipo</b>           | Tipo Algol: Pascal, C, C++, Java, Swift, Kotlin, Rust, Go, etc.         | APL, J, LISP, Python, PHP, Perl, Dart, Hack, Javascript, Typescript* |
| <b>Alcance</b>        | Tipo Algol: Pascal, C, C++, Java, Swift, Kotlin, Rust, Go, Python, etc. | APL, J, BASIC, Perl  |
| <b>Almacenamiento</b> | <b>COBOL, FORTRAN</b>   | <b>Mayoría</b>   |

Para que un lenguaje tenga una propiedad estática o dinámica, la misma debe poder aplicarse a todos los elementos

## Otros aspectos de una variable

|                       |  |
|-----------------------|--|
| <b>Nombre</b>         | Identificador que utiliza el programador para referirse a la celda de memoria de la variable |
| <b>Tiempo de vida</b> | Tiempo en el que la variable tiene asignada una celda de memoria (para el lenguaje)          |

**¿Tienen ligadura estática o dinámica?**

Existen aspectos que son consecuencia de otros  
El Tiempo de vida es consecuencia del almacenamiento y el alcance

```
{  
    a1 = new A();  
    ....  
    if (...) delete a1;  
    ...  
}
```

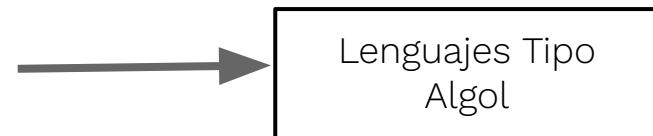
# Clasificación de Lenguajes

- **Almacenamiento estático**



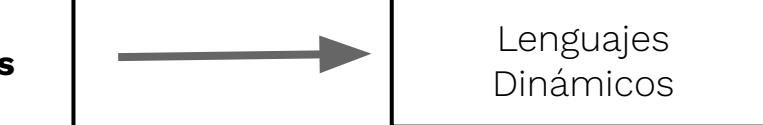
FORTRAN, Cobol y derivados

- **Almacenamiento dinámico**
- **Tipos estáticos**
- **Alcance estático**



Algol, C, C++, C#, ADA, Java, Kotlin, Pascal, Delphi, Go, Rust, Swift, etc.

- **Almacenamiento dinámico**
- **Tipos dinámicos o Alcance dinámico**
  - **Alcance dinámico y tipos dinámicos**
  - **Alcance dinámico y tipos estáticos**
  - **Alcance estático y tipos dinámicos**
  - **Alcance estático y tipos dinámicos con verificación de tipos**



Previa a la ejecución Javascript, Python,  
Durante la ejecución PHP, Ruby, Perl, R, Smalltalk, etc.

## Variedad de Lenguajes Dinámicos

- **Alcance dinámico y Tipos dinámicos:** Lisp, J, APL, Perl
- **Alcance dinámico y Tipos estáticos:** Algunas versiones de Lisp
- **Alcance estático y Tipos dinámicos:** Algunas versiones de Lisp, Scheme, Python, Ruby, JavaScript, Lua, PHP, etc.
- **Alcance estático y Tipos dinámicos con verificador de tipos previo a la ejecución:** Dart, Hack, BeanShell y otros
- **Alcance estático y Tipos dinámicos con verificador de tipos durante la ejecución:** Typescript, ML, Haskell, Scala, Typed LUA, Python con anotaciones, etc.

# Unidades de ejecución

- Conjunto de sentencias de un programa
- Variables locales invisibles desde el exterior

```
{  
    int funcion1()      C  
    {  
        int x;  
        x=3+b;  
    }  
    int main()  
    {  
        int y;  
        y = funcion1();  
    }  
}
```

```
{  
    Program Programa;  
    procedure B;  
    var x: integer;  
    begin  
        ...  
    end;  
    begin  
        B();  
    end.  
}
```

Pascal

¿Por qué la clase  
no es una unidad?

```
{  
    public class  
    Cliente{  
        public int codigo;  
        ...  
        public Cliente(){  
            int temp;  
            ...  
            codigo=0;  
        }  
    }  
}
```

Java

Todos estos lenguajes tienen unidades

# Unidades de ejecución

## Unidades:

- Procedimientos
- Funciones
- Métodos
- Unidades anónimas
- Closures o funciones lambda
- Operaciones
- ...

¿Cuál es la diferencia entre las unidades en estos dos lenguajes?

¿Cuáles variables puede usar la unidad f1 en cada caso?

The diagram illustrates nested code blocks. On the left, a brace groups two code snippets: one for C and one for Pascal. The C code defines a function f1() that returns an integer x, which is then used in the main() function. The Pascal code defines a procedure f1() that returns a variable z, which is then used in the main() function. A large brace on the right groups both the C and Pascal code blocks.

```
int f1()
{
    int x;
    x=3+b;
}
int main()
{
    int y;
    y = f1();
}
```

C

Unidades no anidadas

The diagram illustrates nested code blocks. On the left, a brace groups two code snippets: one for C and one for Pascal. The C code defines a function f1() that returns an integer x, which is then used in the main() function. The Pascal code defines a procedure f1() that returns a variable z, which is then used in the main() function. A large brace on the right groups both the C and Pascal code blocks.

```
Program Programa;
var z: integer;
procedure f1;
var x: integer;
begin
...
end;
begin
f1();
end.
```

Pascal

Unidades anidadas

# Unidades de ejecución en memoria

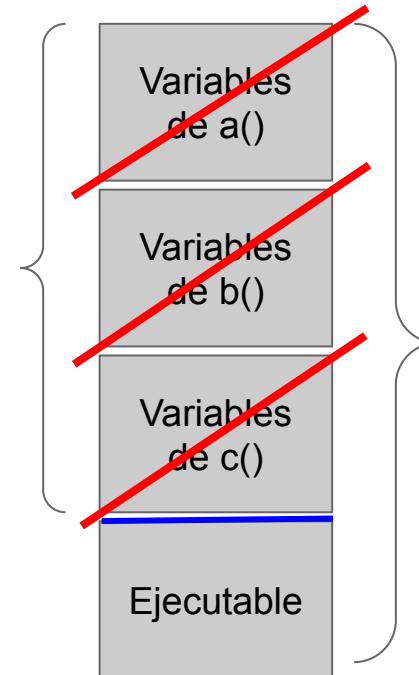
```
void a()
{
    int x;
    x=5;
}
void b()
{
    int x,y;
    a();
}
void c()
{
    int y;
    b();
    y=3;
...
}
```

Secuencia de invocación

~~c() -> b() -> a()~~

**Pila de ejecución**

**No es necesario mantener en memoria las variables de las funciones que ya finalizaron**



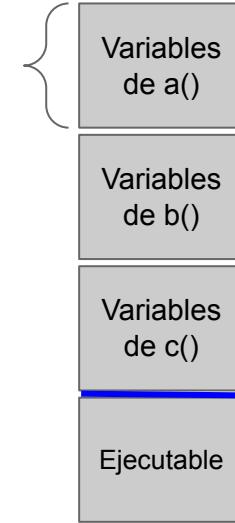
**Espacio asignado por el S.O.**

```
static int z;  
void a()  
{  
    int x;  
    x=5;  
}  
void b()  
{  
    int x,y;  
    a();  
}  
void c()  
{  
    int y;  
    b();  
    y=3;  
    ...  
}
```

## Unidades de ejecución en memoria

c() -> b() -> a()

**Registro de  
activación**



¿Cómo se crean los espacios para las variables de a(), b() y c()?

¿Cómo se hace para ir al espacio de una función?

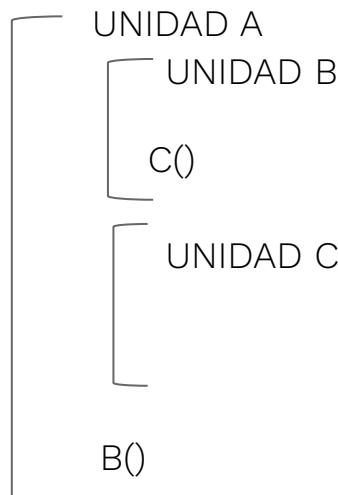
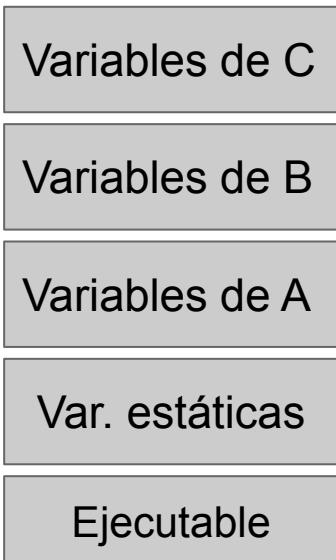
¿Cómo se hace para volver al espacio de la función anterior?

¿Dónde están las variables estáticas y globales?

¿Dónde está el código de las funciones?

# Registro de Activación

Contiene la información **NUEVA** necesaria para que la unidad actual pueda ser ejecutada



La unidad B puede necesitar variables de A y la unidad C variables de A

¿Cómo se accede a las variables?

¿Lo hace el programador o el lenguaje?

## Ejemplos de Alcance

### Pascal

```
program A;
var x : integer;
procedure M;
var p : integer;
procedure S;
var q : integer;
begin
  q := p + x;
end;
begin
  p := x;
  S();
end;
begin
  M();
end;
```

Se están usando desde S variables que están en M y en A sin indicarlo

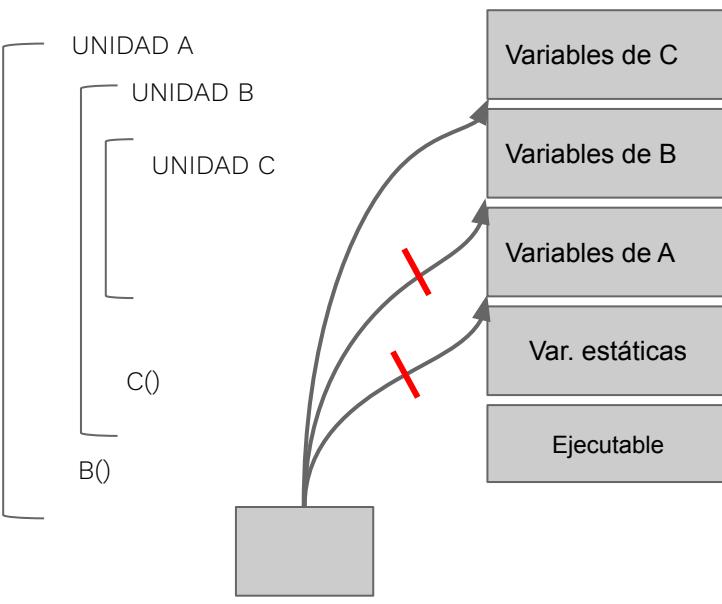
Se están usando desde g variables que están en f y en A sin indicarlo

**Esto mismo sucede en la mayoría de los lenguajes con anidamiento de unidades**

### C#

```
class A {
    public void m() {
        int x;
        void f() {
            int y;
            void g() {
                x = x + y;
            };
            y = 1;
            g();
        };
        ...
        f();
    }
}
```

## Acceso a las variables locales



¿Cómo hace el lenguaje para saber dónde está el registro de activación de la unidad actual?

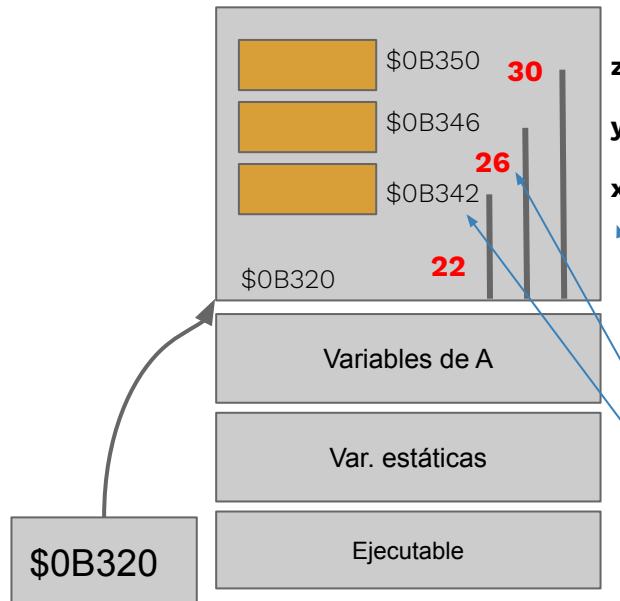
El puntero de pila contiene la dirección del registro de activación de la unidad actual

¿Dónde puede estar ubicado el puntero de pila?

- Suele estar en un registro SP (ESP, XESP) de los procesadores X86, ARM y otras arquitecturas.
- Otros registros pueden apuntar a secciones internas del registro de activación

# Indexado de variables

¿Cómo se accede a las variables locales?



programa fuente

```
void b() {  
    int z,x,y;  
    ....  
    z = x + y;  
}  
  
void a() {  
    b();  
}
```

assembler

```
mov ax, ?  
add ax, ?  
mov ?, ax
```

```
mov ax, [SP] 22  
add ax, [SP] 26  
mov [SP] 30, ax
```

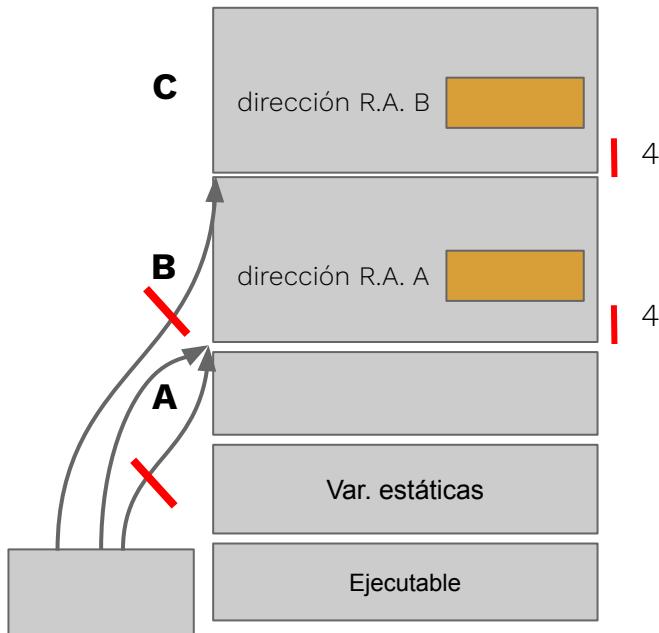
**absoluta      relativa**

|                                |    |
|--------------------------------|----|
| x está en la dirección \$0B342 | 22 |
| y está en la dirección \$0B346 | 26 |
| z está en la dirección \$0B350 | 30 |

No importa dónde está el registro de activación en la pila

¿Es necesario almacenar nombres, direcciones y desplazamientos?

¿Qué pasa cuando se cambia la unidad en ejecución?



Puntero de pila

## Puntero de pila

**A → B → C**

¿Cómo se cambia al registro de activación de la unidad invocada?

¿Cómo se vuelve al registro de activación de la unidad invocadora?

¿Dónde se almacenan las direcciones?

Para volver

assembler

```
ADD SP, Tamaño de B
```

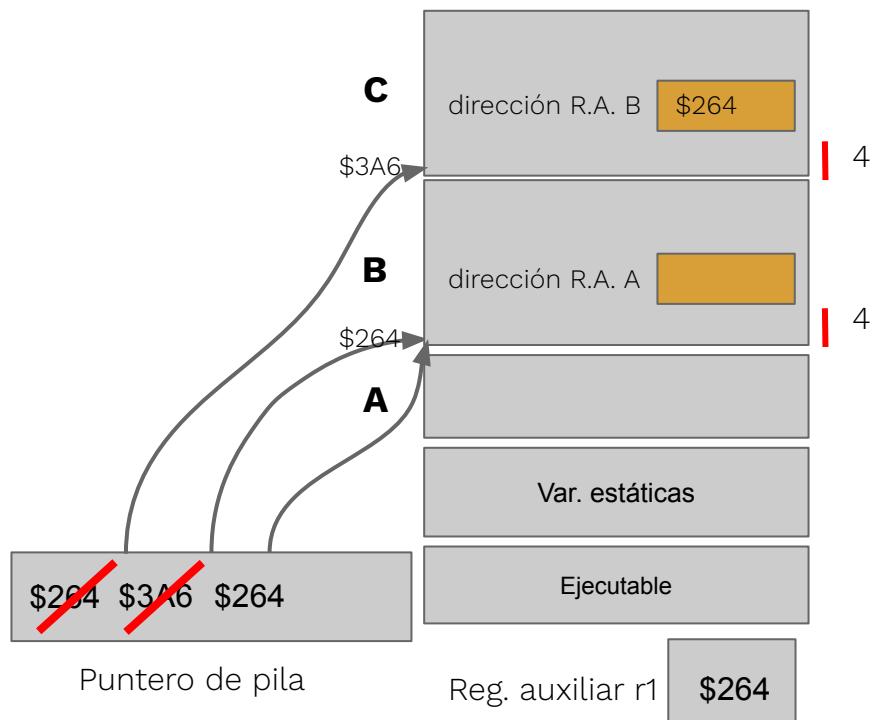
Es necesario almacenar las direcciones de los R.A. de las unidades anteriores

En los mismos registros de activación

```
MOV SP, [SP] 4
```

# Creación del puntero

¿Cuándo y cómo se crea el puntero?



¿Cómo se va a C?

assembler

```
MOV R1, SP  
ADD SP, TAMAÑO B  
MOV [SP] 4, R1  
MOV SP, [SP] 4
```

Resguarda valor SP  
SP APUNTA A C  
SP[4] APUNTA A B

Para volver

¿Cuándo se generan estas instrucciones?

¿Cuándo se ejecutan estas instrucciones?

# Punteros a la base y al tope

Código fuente

```
void f() {  
    var x=1;  
}  
  
void g() {  
    f();  
}
```

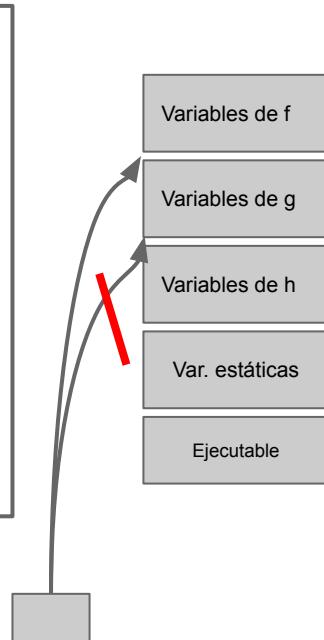
Assembler

```
f:  
MOV SP[12], $1  
  
g:  
MOV R1, SP  
ADD SP, TAMAÑO g  
MOV [SP] 4, R1  
CALL f:  
MOV SP, [SP] 4
```

Puntero a la base del R.A.

Puntero de pila

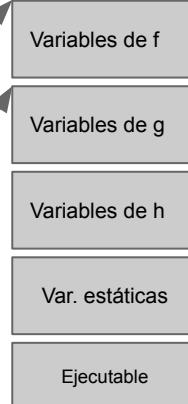
A → B → C



Assembler

```
f:  
MOV R1, SP  
ADD SP, TAMAÑO f  
MOV [SP] 4, R1  
MOV SP[12], $1  
MOV SP, [SP] 4
```

```
g:  
CALL f:
```



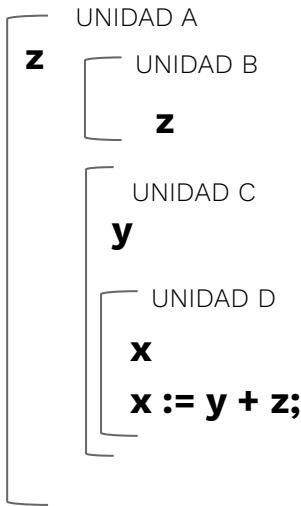
¿Cuál solución es más eficiente?

¿Quién hace el trabajo de crear el R.A.?

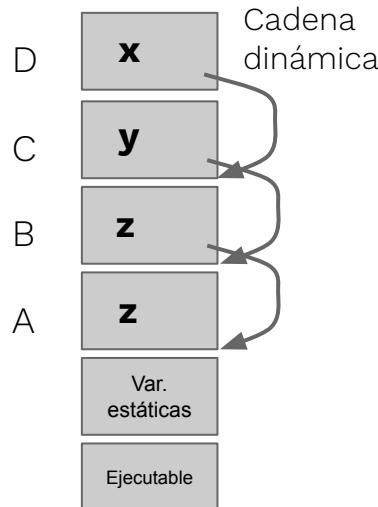
¿Cómo se accede a las variables en el caso de puntero al tope?

## Acceso a variables no locales

¿Cómo se accede a las variables no locales a la unidad?



A -> B -> C -> D



¿Cómo se hace para acceder a las variables y, z desde la unidad D?

¿Alcanza con la cadena dinámica para acceder a las variables no locales?

En un lenguaje Tipo Algol no se almacenan nombres de variables!

¿Cuál variable z se debe utilizar en un lenguaje con alcance estático dado por el anidamiento?

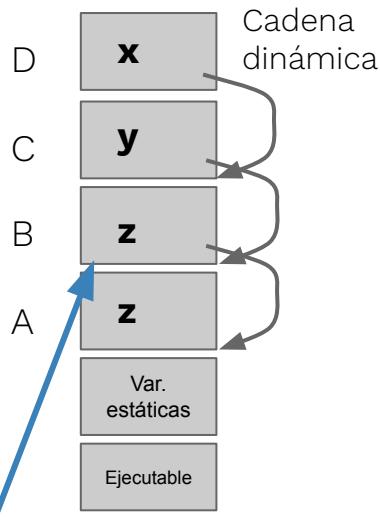
¿Qué sucede en un lenguaje dinámico con pila?

### python

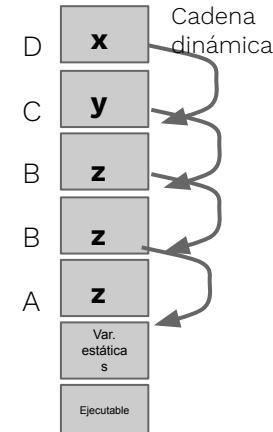
```
def a:  
    z = 1  
def b:  
    z = 3  
    c()  
def c:  
    y = 2  
    def d:  
        x = y + z  
    d()  
b()
```

## Acceso a variables no locales

Es necesario otro mecanismo



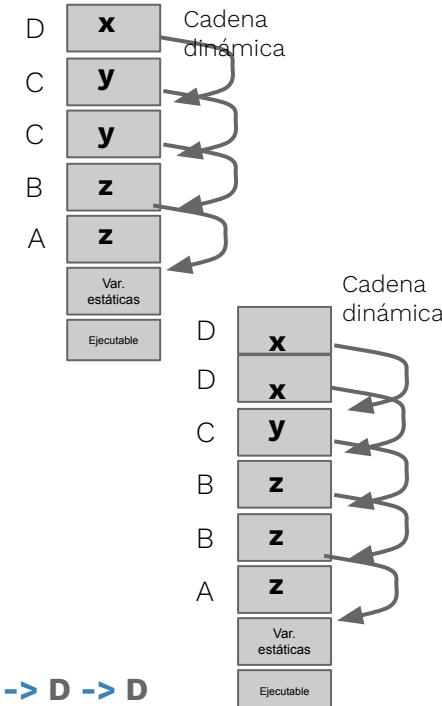
En un lenguaje dinámico con pila encontraría la variable de B antes que la de A!!



A → B → B → C → D

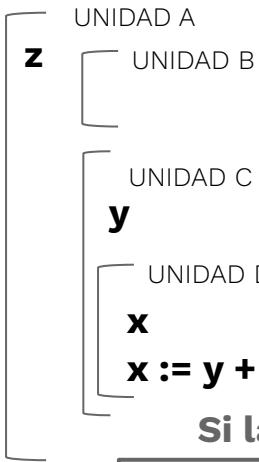
A → B → B → C → D → D

A → B → C → C → D



## Acceso a variables no locales

A → B → C → D



Si las variables fuesen locales a D

```
MOV AX, [SP] 48 ;MOVER Y AAX  
ADD AX, [SP] 22 ;SE SUMA Z CON AX  
MOV [SP]16,AX ;SE MUEVE AX A X
```

assembler

Faltan instrucciones para ir al R.A. correspondiente

Independientemente de las llamadas:

A → B → B → C → D

A → B → C → C → D

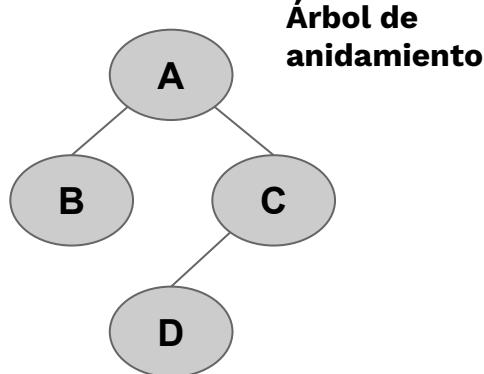
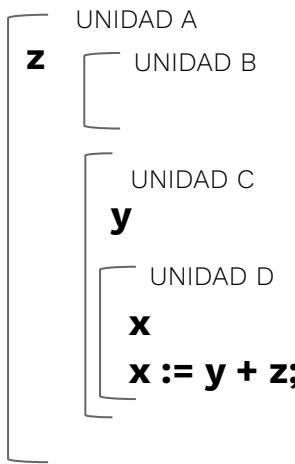
A → B → B → C → D → D

etc.

La estructura de anidamiento se mantiene

|   |    |   |
|---|----|---|
| x | 16 | 0 |
| y | 48 | 1 |
| z | 22 | 2 |

## Acceso a variables no locales

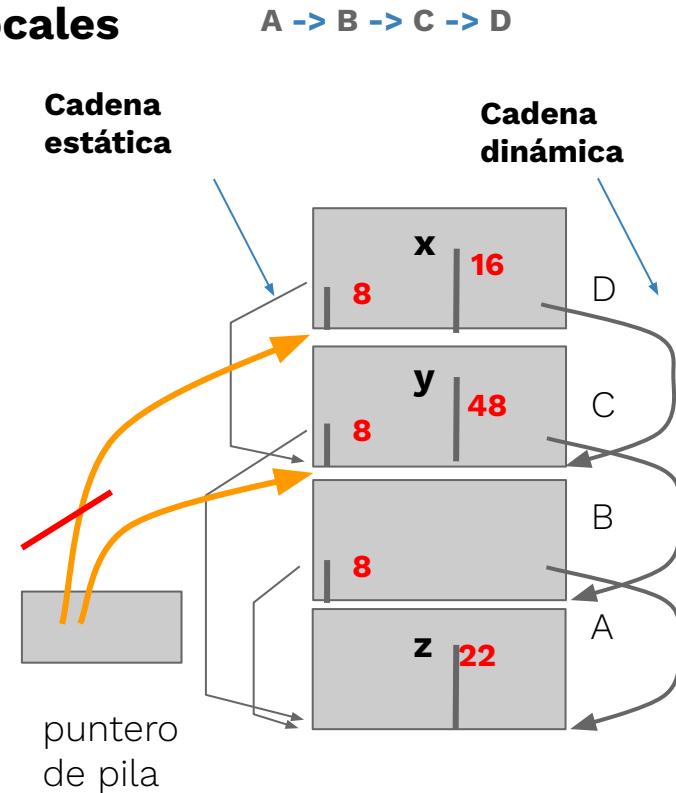


Para llegar al R.A. de C desde D

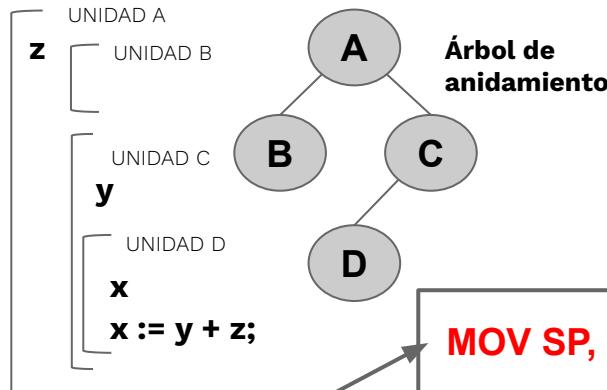
MOV SP, [SP] 8

Para llegar al R.A. de A desde D

MOV SP, [SP] 8  
MOV SP, [SP] 8



## Acceso a variables no locales



Va al R.A. de C

Van al R.A. de A

assembler

**MOV SP, [SP] 8**  
MOV AX, [SP] 48 ;MOVER Y A AX

**MOV SP, [SP] 8**

**MOV SP, [SP] 8**

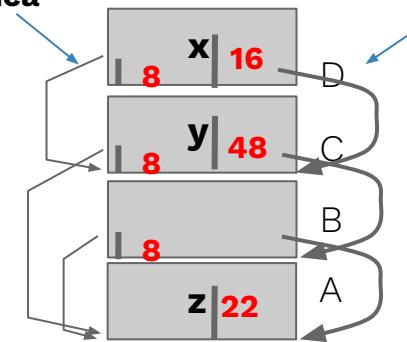
ADD AX, [SP] 22 ;SE SUMA Z CON AX

MOV [SP]16,AX ;SE MUEVE AX A X

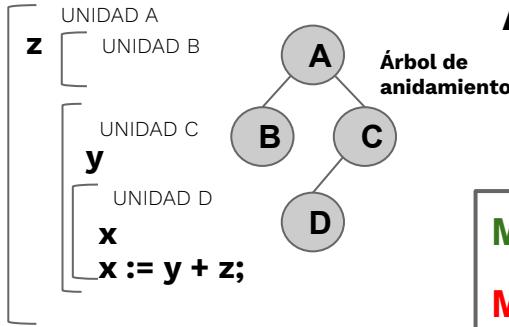
A → B → C → D

Cadena  
estática

Cadena  
dinámica



¿Cuáles instrucciones se necesitan?



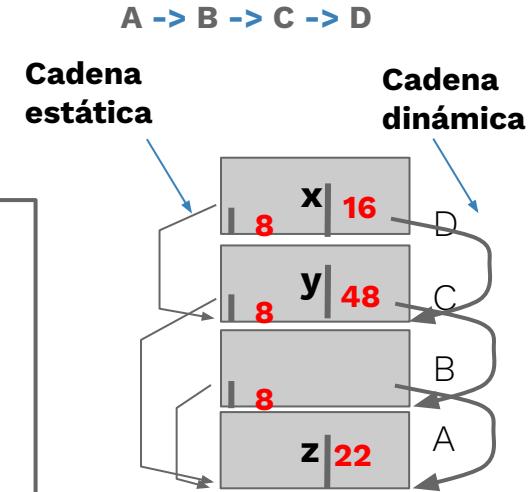
## Acceso a variables no locales

assembler

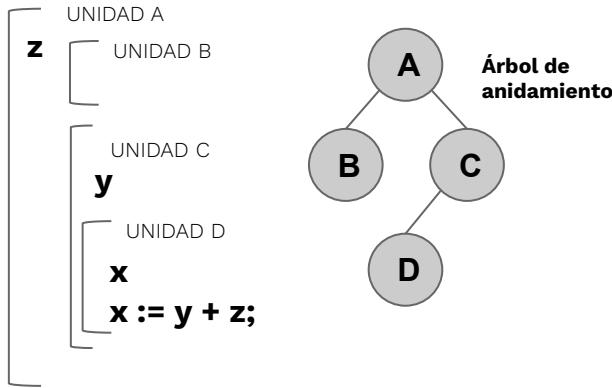
Restauran el SP  
Se mueven por la C.E.  
Hacén las operaciones

```

MOV R1, SP
MOV SP, [SP] 8
MOV AX, [SP] 48 ;MOVER Y A AX
MOV SP, R1
MOV SP, [SP] 8
MOV SP, [SP] 8
ADD AX, [SP] 22 ;SE SUMA Z CON AX
MOV SP, R1
MOV [SP]16,AX ;SE MUEVE AX A X
  
```



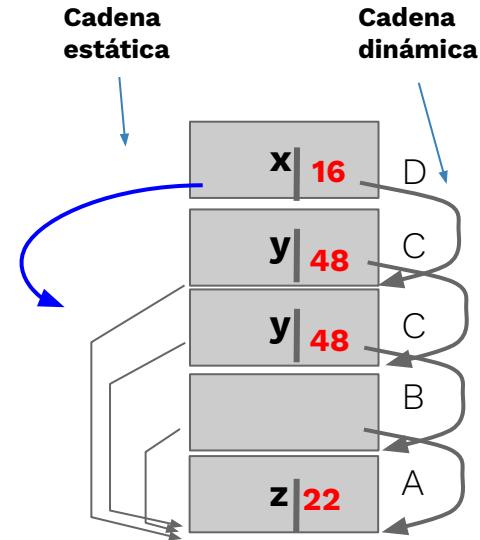
## **Invocaciones Recursivas**



**A → B → C → C → D**

## ¿Cuál variable “y” debe usar la unidad “D”?

## ¿Cuál variable “y” usa efectivamente la unidad “D”?



**Se determina según el mecanismo de construcción de la cadena estática**

**Que la unidad “D” use la variable “y” más reciente ¿se determina en tiempo de compilación o en tiempo de ejecución?**

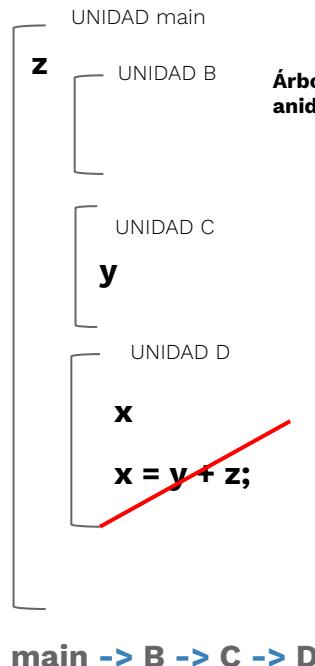
```

int B() {
    C();
}
int C() {
    int y;
    D();
}
int D() {
    int x;
    x = y + z;
}

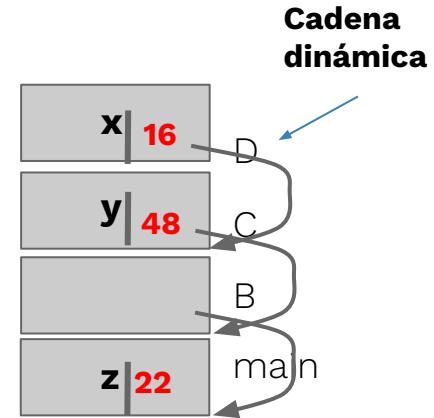
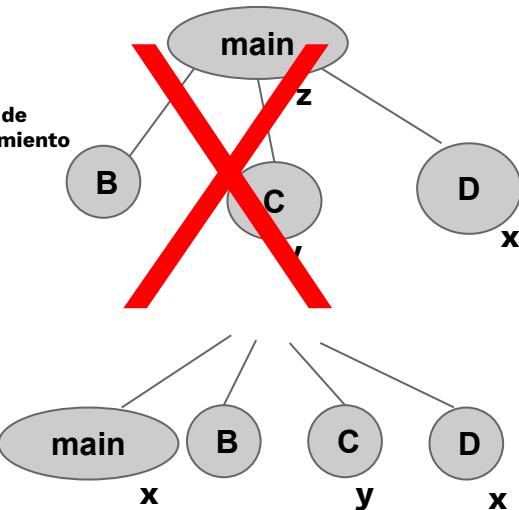
int main()
{
    int z;
    B();
}

```

C



## Unidades disjuntas



¿Compila el programa?

¿Es necesaria la cadena estática?

¿Es necesaria la cadena dinámica?

```

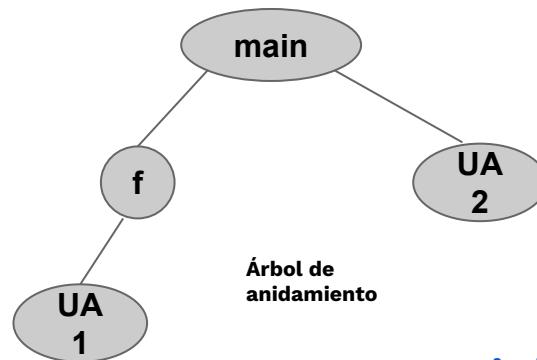
class Program    C#
{
    static void Main() {
        int w=123;
        void f() {
            int x=234;
            if (x==2) {
                int z;
                z = x + w;
            }
        }
        while (w<5) {
            int y = w;
        };
    }
};

```

## Unidades anónimas

Bloques sin nombre de sentencias que permiten definir variables locales

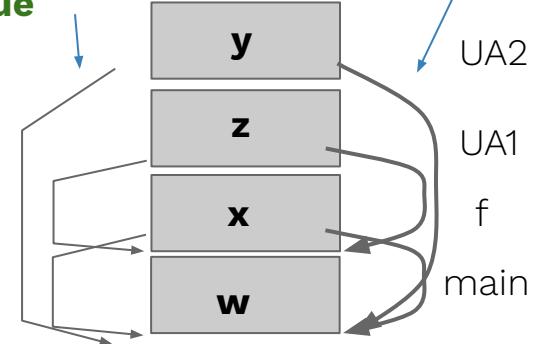
¿Quienes las pueden llamar?



¿Es necesaria la cadena estática?

¿Es necesaria la cadena dinámica?

Cadena  
estática



Cadena  
dinámica

UA2

UA1

f

main

# Unidades anónimas

¿Compilan estos programas?

**C** X

```
int main() {  
    int z;  
    if (...) {  
        int x;  
    }  
    z = x;  
};
```

**Java** X

```
public class A{  
    public static  
        void main(String []args){  
            int x=1;  
            if (x==2) {  
                int z = 3;  
            }  
            x = z;  
        }  
}
```

**C** ✓

```
int main() {  
    int x;  
    if (...) {  
        int x;  
    }  
};
```

**Java** X

```
public class A{  
    public static  
        void main(String []args){  
            int x=1;  
            if (x==1) {  
                int x = 3;  
            }  
        }  
}
```

¿Cuáles programas sirven para saber si el lenguaje tiene unidades anónimas?

**Pascal** X

```
IF (...) BEGIN  
    VAR A : INTEGER;  
    END;
```

# Unidades anónimas e iteraciones

## Rust

```
fn main() {  
    for i in 0..5 {  
        let a = 1;  
    }  
}
```

## Zig

```
while (true) : (x = x + 1) {  
    if (x < 2) {  
        ...  
        continue;  
    }  
    break;  
}
```

## Go

```
package main  
import "fmt"  
func main() {  
    var i int = 1  
    for i:=0; i<3; i++ {  
        var i int = 2  
        fmt.Println(i)  
    }  
    fmt.Println(i)  
}
```

¿En una iteración quien llama a la siguiente U.A. la unidad que la contiene o la U.A. de la iteración actual?

¿Cuántas unidades anónimas existen al mismo tiempo durante la ejecución?

Continúa con la siguiente iteración  
Sale del while

# Expresiones Lambda

```
package main          Go
import "fmt"

var a func()

func f() {
    a()
}

func main() {
    var x = 1
    func() { fmt.Println(x) }()
    a = func() { fmt.Println(x) }
    a()
    f()
}
```

Unidades anónimas

**Cuando se define una expresión lambda y se ejecuta es similar a ejecutar una unidad anónima**

**Cuando se asigna una expresión lambda a una variable, se crea un closure que se asigna a la variable**

**¿Las expresiones lambda permiten reducir o aumentar el alcance de la unidad anónima definida en la expresión lambda?**

**Si la expresión lambda utiliza variables de su ámbito, estas variables se almacenan en un closure**

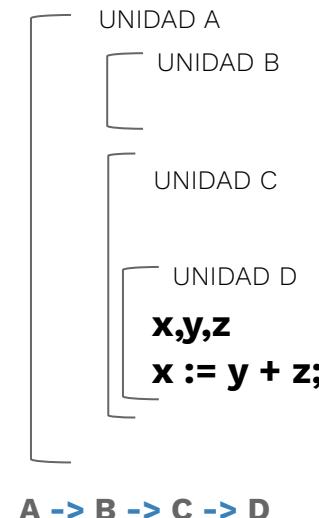
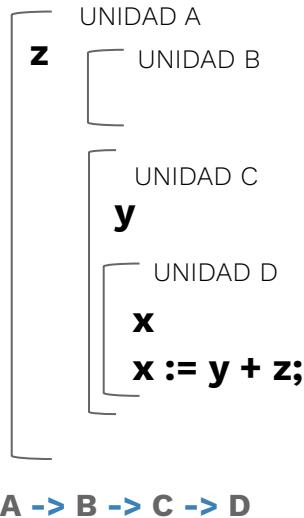
# Cadena estática y eficiencia

$x := y + z$   
si  $z,y$  son no  
locales a D



## Assembler

```
MOV R1,SP  
MOV SP,[SP]8  
MOV AX,[SP]48  
MOV SP, R1  
MOV SP,[SP]8  
MOV SP,[SP]8  
ADD AX,[SP]22  
MOV SP, R1  
MOV [SP]16,AX
```



$x := y + z$   
si son locales



## Assembler

```
MOV AX,[SP]48  
ADD AX,[SP]22  
MOV [SP]16,AX
```

**3 veces menos instrucciones con variables locales !!**  
**La cantidad de instrucciones depende de la distancia a la  
variable en el árbol de anidamiento**

# Construcción de la cadena estática

¿Cuáles unidades pueden invocar a una unidad?

```
fn main() {  
    let mut x = 5;  
    if x == 5 {  
        let x = 6;  
    }  
}
```

**Unidades  
anónimas**

invocada  
por

**Unidad  
contenedora**

```
func main() {  
    func f() {  
        func g() {  
        }  
    }  
    func h() {  
    }  
}
```

**Unidades con  
nombre**

invocada  
por

**Las unidades  
que la tengan al  
alcance**

**Hay que conocer el alcance de las unidades!**

## Alcance de unidades

¿El alcance de las unidades es igual que el alcance de las variables?

El alcance de las unidades no es el mismo que el de las variables

El nombre de la unidad debe ser global a la unidad misma

El nombre de la unidad es local a la unidad que la contiene

En el ejemplo:

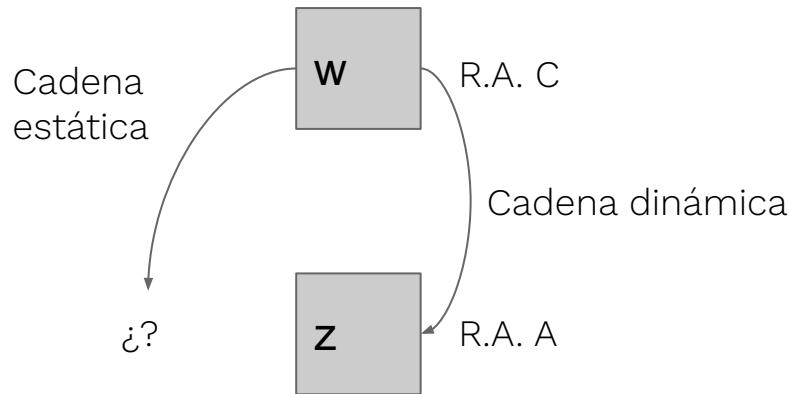
- B es local a A
- B tiene el mismo alcance que y

Pascal  
program A;  
var y : integer;  
procedure B;  
var x : integer;  
begin  
 x := 3;  
 ...  
 B;  
end;  
begin  
 x := 4; X  
 B; ✓  
end;

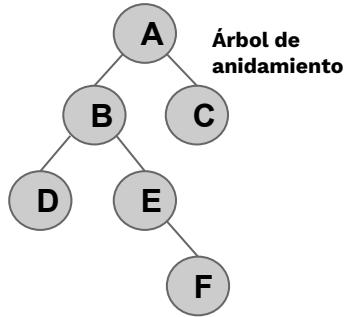
¿Por qué una se acepta y la otra no?

## Caso de análisis

```
procedure A;  
var z : integer;  
procedure B;  
var x : integer;  
procedure C;  
var w : integer;  
begin  
... z := x + w;  
end;  
begin (*de B*)  
...  
end;  
begin (*de A*)  
C;  
end;
```



- **C puede necesitar variables de B**
- **La secuencia de invocaciones que lleva a la unidad C debe contener la unidad B**



L: Llamada local

R: Llamada Recursiva

G1: Llamada global (distancia 1)

G2: Llamada global (distancia 2)

G3: Llamada global (distancia 3)

G4: Llamada global (distancia 4)

## Matriz de invocaciones

|   | A  | B  | C  | D  | E  | F |
|---|----|----|----|----|----|---|
| A | R  | L  | L  | X  | X  | X |
| B | G2 | R  | G1 | L  | L  | X |
| C | G2 | G1 | R  | X  | X  | X |
| D | G3 | G2 | G2 | R  | G1 | X |
| E | G3 | G2 | G2 | G1 | R  | L |
| F | G4 | G3 | G3 | G2 | G2 | R |

¿Las llamadas recursivas son locales o globales?

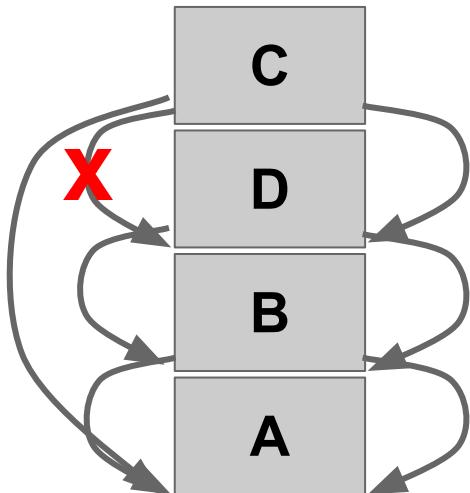
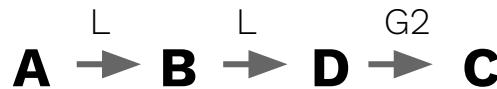
¿Cuál es la unidad con mayor alcance?

¿Cuál es la unidad con menor alcance?

¿Pueden existir unidades con el mismo nombre?

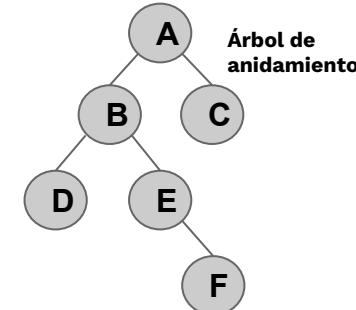
# Método de construcción de la cadena estática

Secuencia de invocaciones:



Algoritmo:

1. Copiar el puntero de la cadena dinámica en la estática
2. Seguir la cadena estática tantas veces como la distancia entre el llamador y el padre del llamado (globalidad)

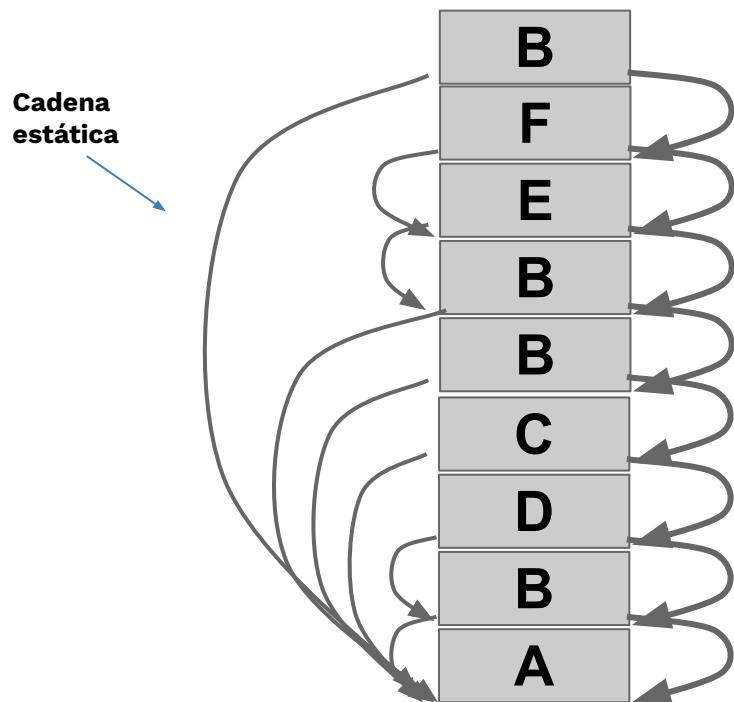


Assembler:

|                    |                               |
|--------------------|-------------------------------|
| MOV R1, SP         | Se resguarda el SP en R1      |
| MOV [SP] 8, [SP] 4 | Se copia C.D. en C.E.         |
| MOV SP, [SP] 8     | Se sigue la C.E. según paso 2 |
| ...                |                               |
| MOV R2, SP         | Se resguarda el SP en R2      |
| MOV SP, R1         | Se restaura el SP             |
| MOV SP[8], R2      | Se copia la C.E. en R.A. de C |

## Ejemplo de Cadena Estática

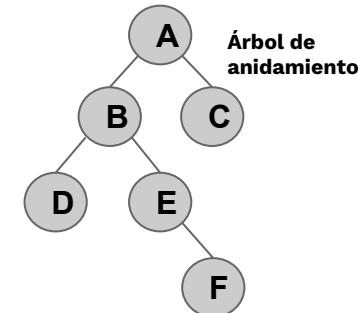
A → B → D → C → B → B → E → F → B



### Algoritmo:

1. Copiar el puntero de la cadena dinámica en la estática
2. Seguir la cadena estática tantas veces como la distancia entre el llamador y el parent del llamado (globalidad)

¿Cómo sería la cadena si hay un llamado de C a F?



# Preguntas

**Inciso de pregunta de Final:  
¿Es posible el siguiente programa?**

```
Program Programa(output);
Procedure A;
  Procedure A;
    begin
      writeln('A de adentro');
      A;
    end;
  begin
    writeln('A de afuera');
    A;
  end;
begin
  A;
end.
```

**¿Para qué sirve la cadena dinámica?**

**¿Para qué sirve la cadena estática?**

**¿Cuándo se generan las instrucciones para  
crear la cadena estática?**

**¿Cuándo se generan las instrucciones para  
crear la cadena dinámica?**

# Clasificación de Variables

Respecto de su almacenamiento, hasta ahora se vieron 2 clases de variables:

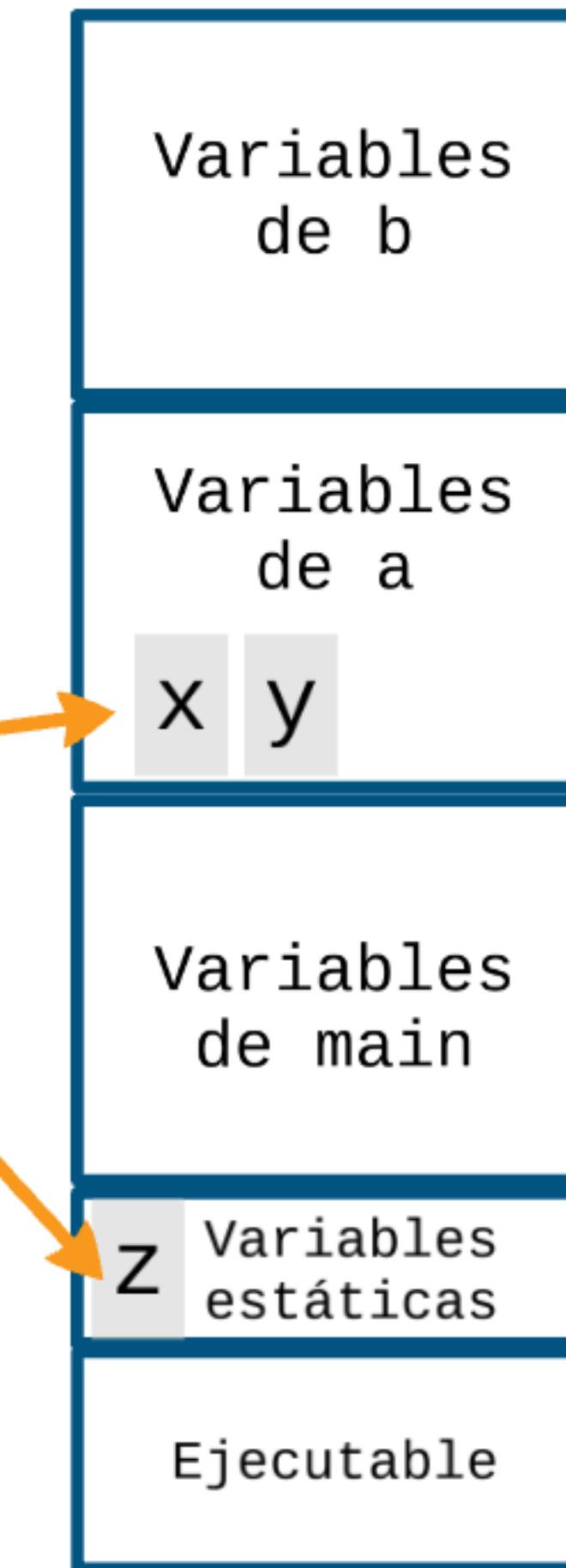
## Estáticas

- Tamaño fijo
- Están en una dirección fija

## Semi-estáticas

- Tamaño fijo
- Pueden estar en el registro de activación

```
static int z;  
void b()  
{...}  
void a()  
{  
    int x,y;  
    b();  
}  
void main()  
{  
    a();  
}
```



# ¿Qué ocurre en estos ejemplos?

```
static int x;  
  
void a()  
{  
    static int y;  
}  
void main()  
{  
    a();  
}
```

```
static int x;  
  
void a()  
{  
    static int x;  
}  
void main()  
{  
    a();  
}
```

```
static int x=1;  
void main()  
{  
    static int x=2;  
    printf("%d",x);  
}
```

```
void a() {  
    printf("%d",x);  
}  
  
void main()  
{  
    static int x=2;  
    printf("%d",x);  
}
```



# Variables Semi-estáticas

## Pascal

```
procedure A;
procedure B;
procedure C;
var w: array [4..10,3..15] of integer;
begin
  w[3,4] := 5;
end;
begin
  C;
end;
begin
  B;
end;
```

¿Dónde se almacena la variable w?

- En el registro de activación de la unidad C

¿Qué tamaño tiene la variable w?

- El tamaño necesario para almacenar el arreglo completo

¿Cuándo se conoce el tamaño y los límites del arreglo w?

En tiempo de compilación

w es una variable semi-estática

# Clasificación de Variables

Existen otro tipo de variables

```
void a(int tam)
{
    int x[tam];
}
void main()
{
    a(rand()*10);
}
```

¿Qué clase de variable es x?

No se pueden conocer en tiempo de compilación ni sus límites ni su tamaño

No es estática ni semi-estática

x es semi-dinámica

Clasificación de variables por almacenamiento

# Clasificación de variables por almacenamiento

- Estáticas

- Tamaño fijo
- Están en una dirección fija de la pila de ejecución

- Semi-estáticas

- Tamaño fijo
- Se pueden poner en el registro de activación

- Semi-dinámicas

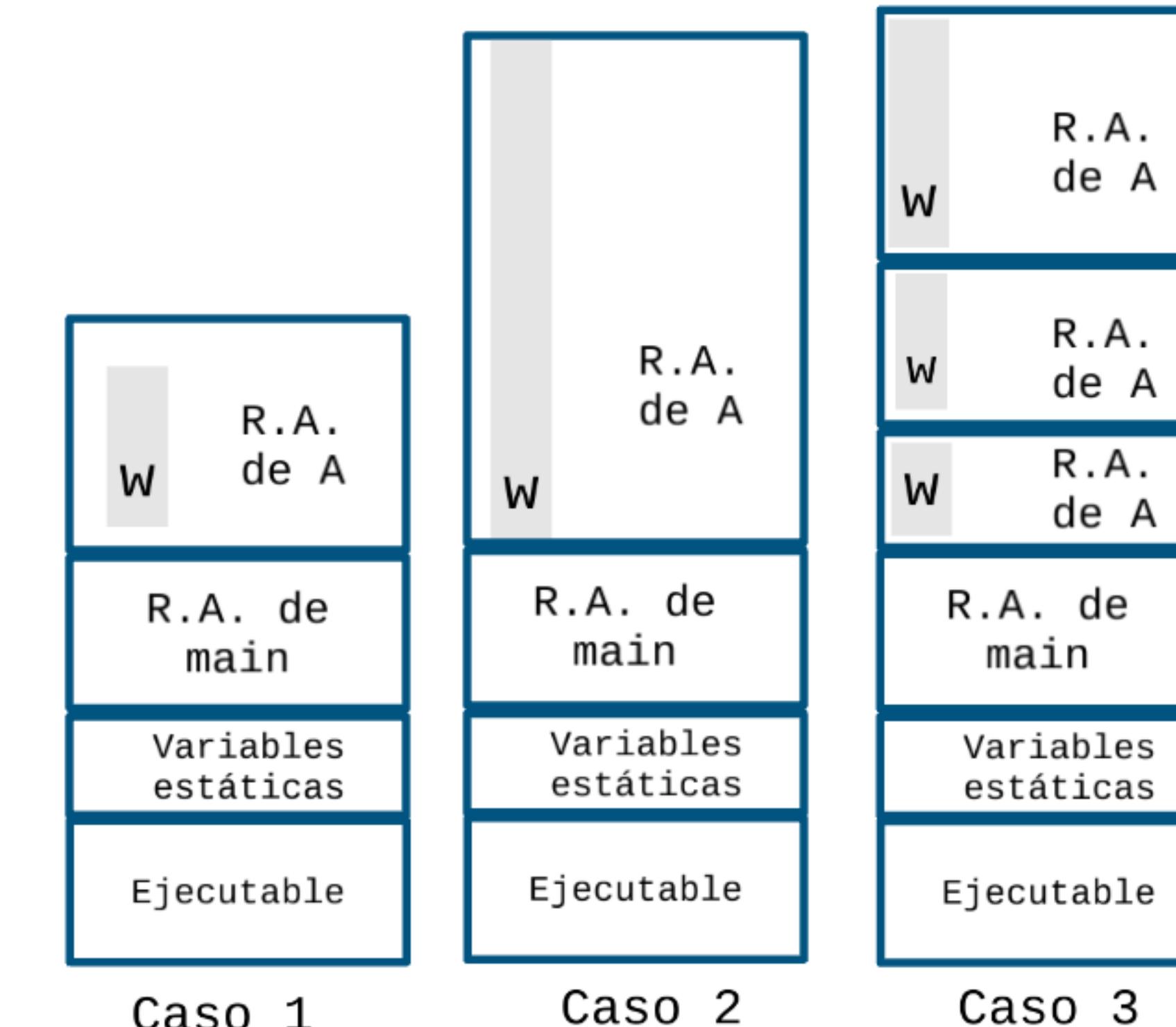
- Tamaño que puede cambiar SOLO entre diferentes R.A. (invocaciones) de la unidad.
- Se pueden poner en el registro de activación

# Variables semi-dinámicas

¿Qué consecuencias traen estas variables en los registros de activación?

```
void A(int tam)
{
    int w[tam];
    if (tam<5)
        A(tam*2)
}
void main()
{
    A(5); // Caso 1
    A(15); //Caso 2
    A(2); //Caso 3
}
```

Sucesivas  
invocaciones a la  
unidad a(), provocan  
que el R.A. tenga  
tamaños diferentes



¿Dónde se debe ubicar el elemento que define  
el tamaño de los arreglos?

Al alcance de unidad donde está el arreglo.

# Ubicación del elemento que determina el tamaño de las variables semi-dinámicas

- Variable no local o global  
¿Hay alguna diferencia si la variable global es estática o está almacenada en el R.A. de otra unidad?
- Parámetro
- Variable local  
(Hay algunos compiladores que no lo permiten)

¿Qué diferencia respecto del alcance y almacenamiento hay entre un parámetro y una variable local?

# Tamaño de variables semi-dinámicas

## ¿Dónde están las variables que definen el tamaño de las variables semi-dinámicas?

Deben estar al alcance de la unidad: respecto del almacenamiento pueden ser locales, estáticas o estar en unidades superiores en el árbol de anidamiento (globales)

```
static int t;  
void B()  
{  
    int w[t];  
    if (t>1) {  
        t--;  
        B();}  
}  
void A()  
{  
    t=3;  
    B();  
}
```

Los tamaños de los R.A. de invocaciones anteriores de B() no pueden cambiar de tamaño



Debe existir una copia de la variable t para cada invocación de B()



Variables semi-dinámicas

- El valor de la variable t "original" puede cambiar
- Los "valores" de t',t" y t''' se mantienen constantes durante las invocaciones de B()

# Variables semi-dinámicas

¿Qué sucede en este caso?

c/c++

Depende del compilador

```
void A()
{
    int j;
    int x[j];
}
```

Algunos dan error y otros no

¿Los errores son de  
compilación o de ejecución?

En los que no dan error,  
¿qué sucede en este caso con el  
tamaño del arreglo x?

```
void A()
{
    int j;
    int x[j];
    j++;
}
```

# VARIABLES SEMI-DINÁMICAS EN EL R.A.

¿Cómo afectan a la ubicación del resto de las variables?

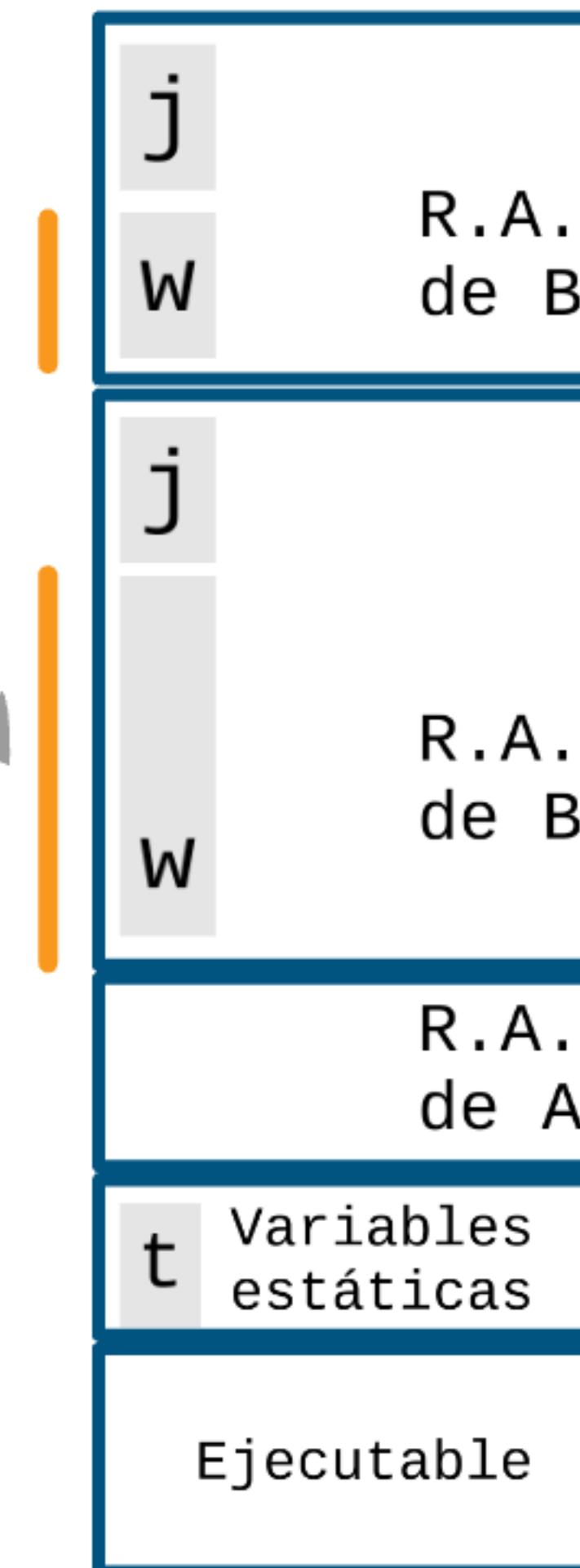
```
static int t;  
void B()  
{  
    int w[t];  
    int j;  
    ...  
    B();  
}  
void A()  
{  
    t=3;  
    B();  
}
```

Distintos  
offsets  
para la  
variable j

El modelo de Base +  
offset no funciona

¿Es cierto?

Variables semi-estáticas y variables semi-dinámicas en el RA

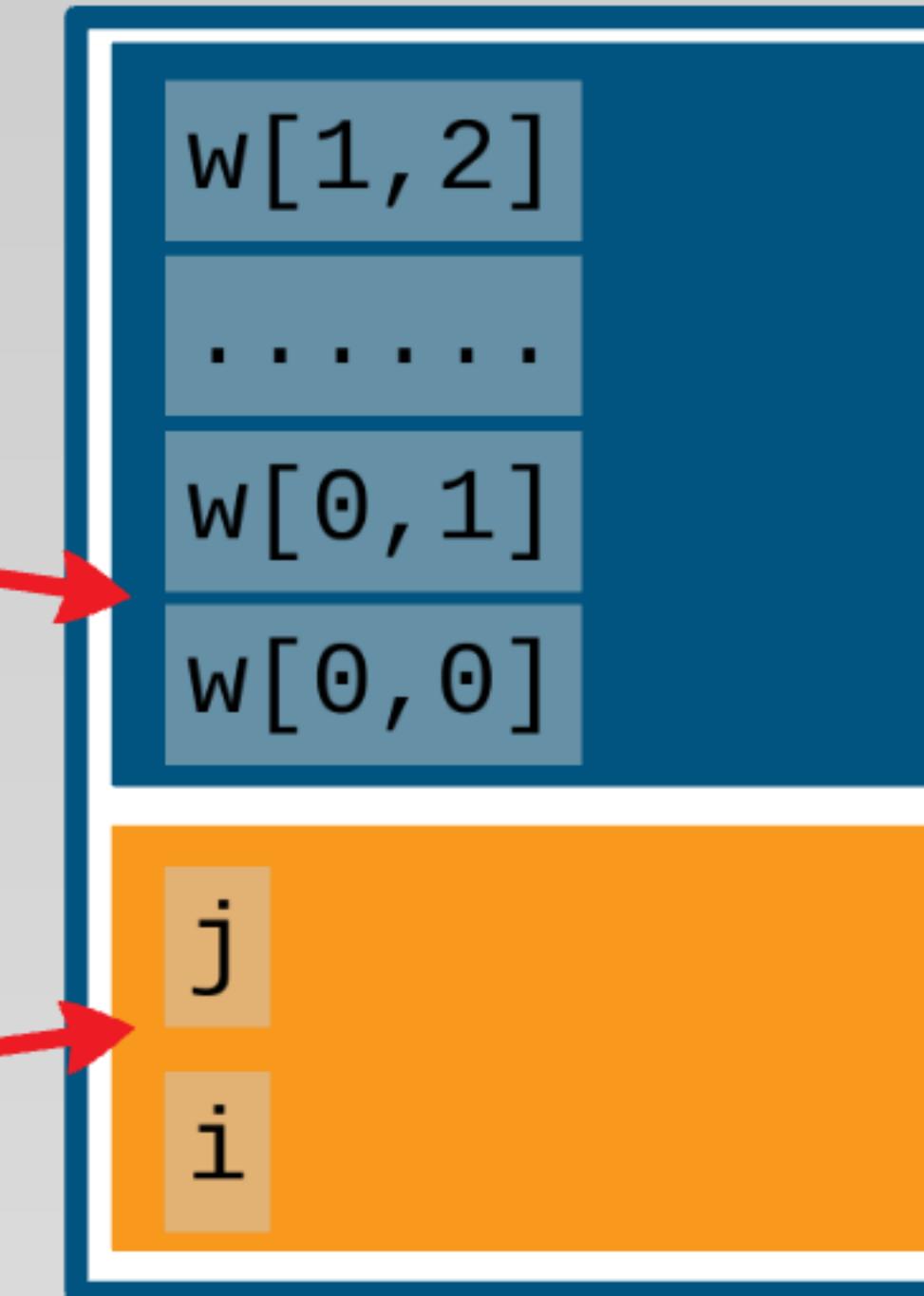


# Variables semi-estáticas y variables semi-dinámicas en el R.A

Variables  
semi-  
dinámicas

Variables semi-  
estáticas

En esta parte  
funciona el modelo  
base + offset



R.A.

```
static int x=2;
static int y=3;
void A()
{
    int w[x][y];
    int i, j
}
```

¿Cómo se accede a las variables semi-dinámicas?

Si los límites pueden variar, se necesita algo más que los datos

# Límites de los arreglos

## ¿Dónde están los límites de los arreglos?

### Ejemplo en Delphi

```
procedure A;
procedure B;
procedure C;
var w: array [4..10,3..15] of integer;
begin
w[3,4] := 5;
end;
begin
  C;
end;
begin
  B;
end;
```

[Error] Unit1.pas(31): Constant expression violates subrange bounds  
[Fatal Error] Project1.dpr(5): Could not compile used unit 'Unit1.pas'

### Ejemplo en C

```
void A()
{
    int M[3][5];
    M[-4][-2] = 6;
}
```

Compila bien

Hay lenguajes que comprueban límites y otros que no.

Si en tiempo de ejecución, el lenguaje comprueba los límites, tienen que estar en tiempo de ejecución.

Si el lenguaje no comprueba límites, ¿es cierto que los límites no están en memoria?

# Tamaño de variables semi-dinámicas

¿Cómo se ubican los arreglos en memoria?

```
procedure A(x: integer);  
// Ej. Programa con sintaxis de Pascal  
var w: array [4..x,3..15] of integer;  
begin  
  w[5,4] := 0;  
end;  
..  
A(10);
```

• Por filas

Dirección de  $w[i,j] = \text{dir } w[4,3] +$

$\{[(i-4)*(15-3+1)+j-3]*\text{Tamaño(int)}$

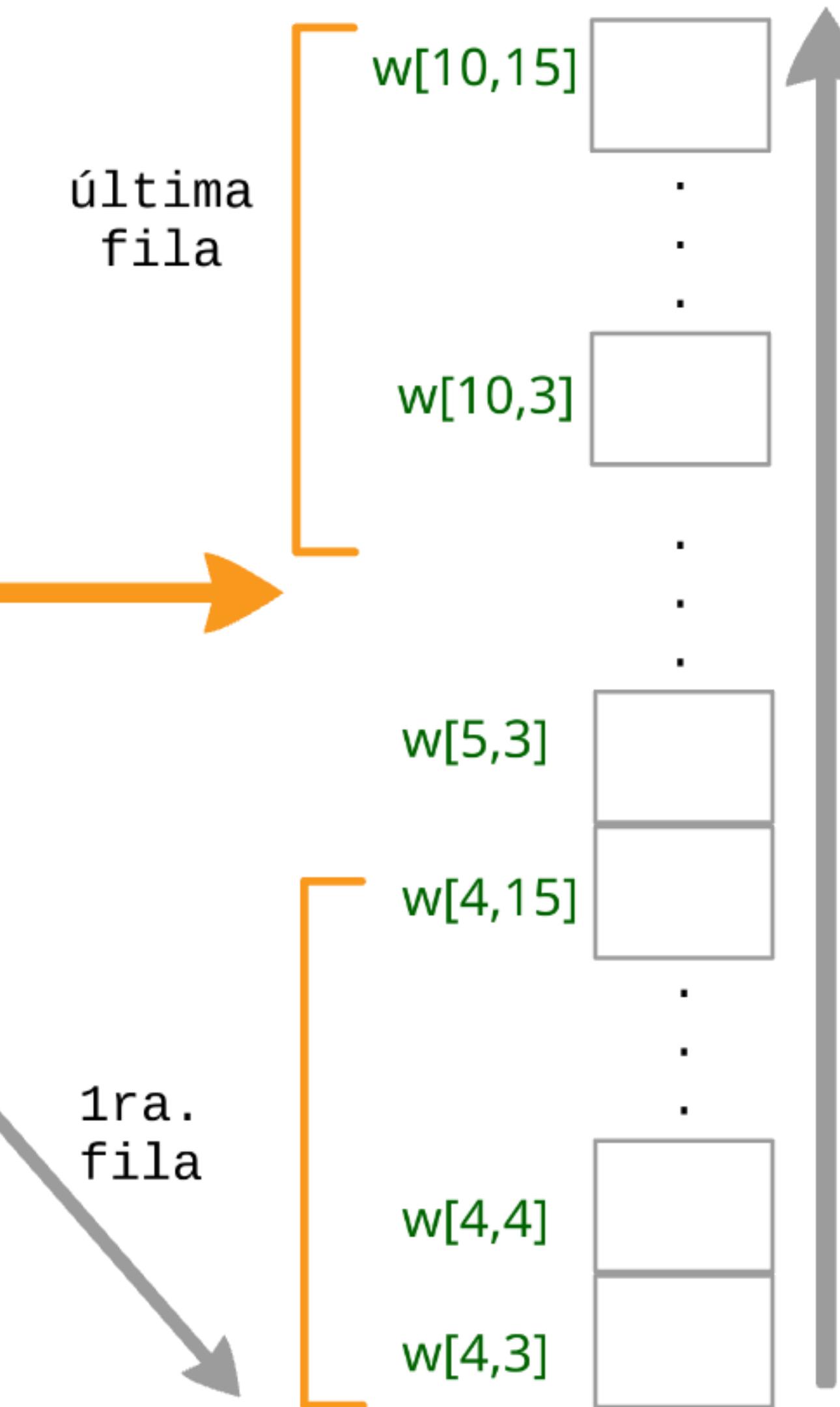
# filas antes  
de la que  
estamos  
buscando

# elementos  
por filas

# columnas  
hasta la que  
estamos  
buscando

¿Dónde están los números 4, 3 y 15?

Si son constantes, están en el código ejecutable



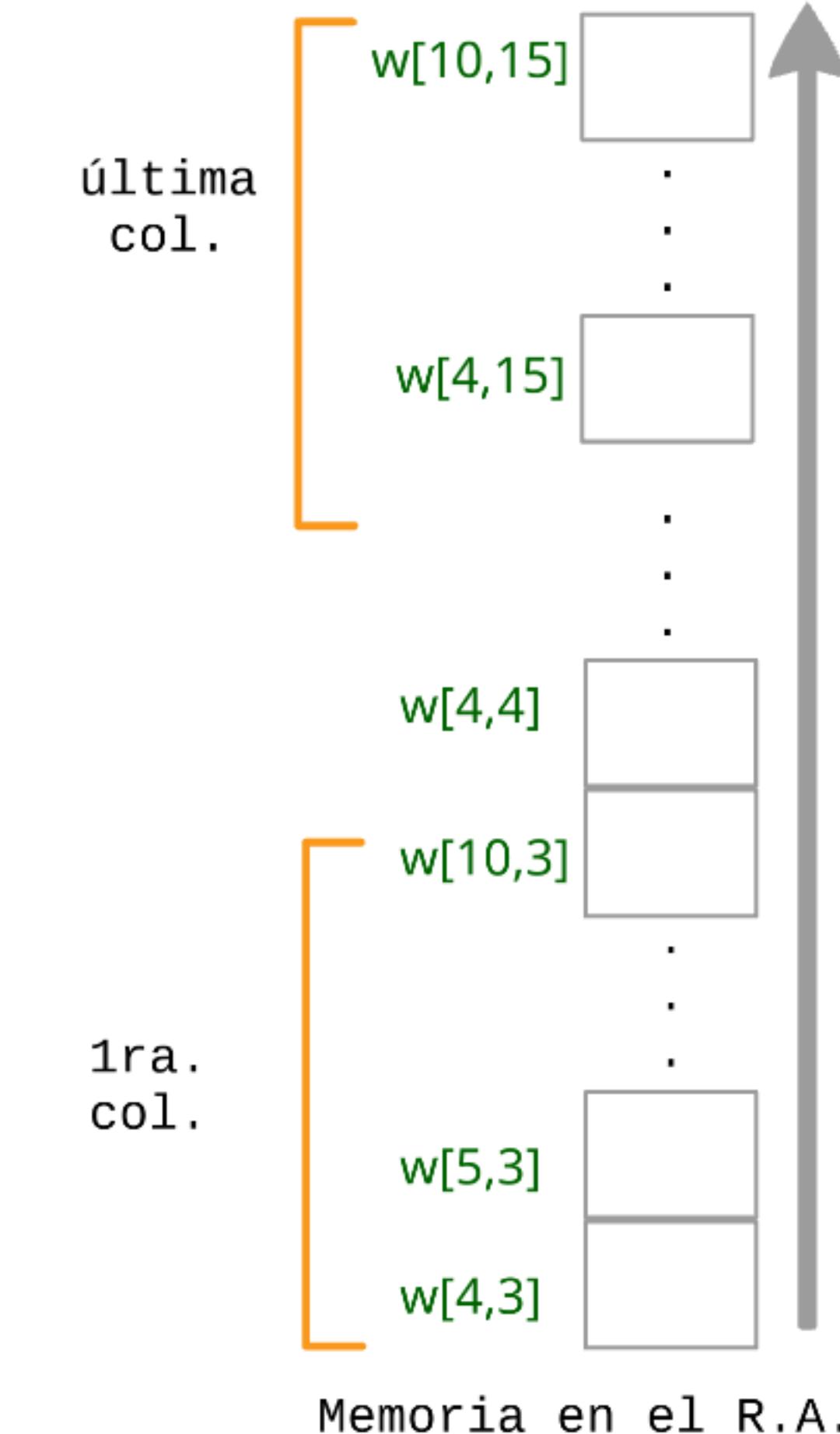
Memoria en el R.A.

# Por columnas

Dirección de  $w[i,j] = \text{dir } w[4,3] +$   
 $\{[(j-3)*(10-4+1)+i-4]*\text{Tamaño}(\text{int})$

# col antes de la que estamos buscando    # elementos por columnas    # filas hasta la que estamos buscando

```
procedure A(x : integer);
var w: array [4..x,3..15] of integer;
begin
  w[5,4] := 0;
end;
..
A(10);
```



# Acceso a las variables semi-dinámicas

¿Dónde están los límites si son variables?

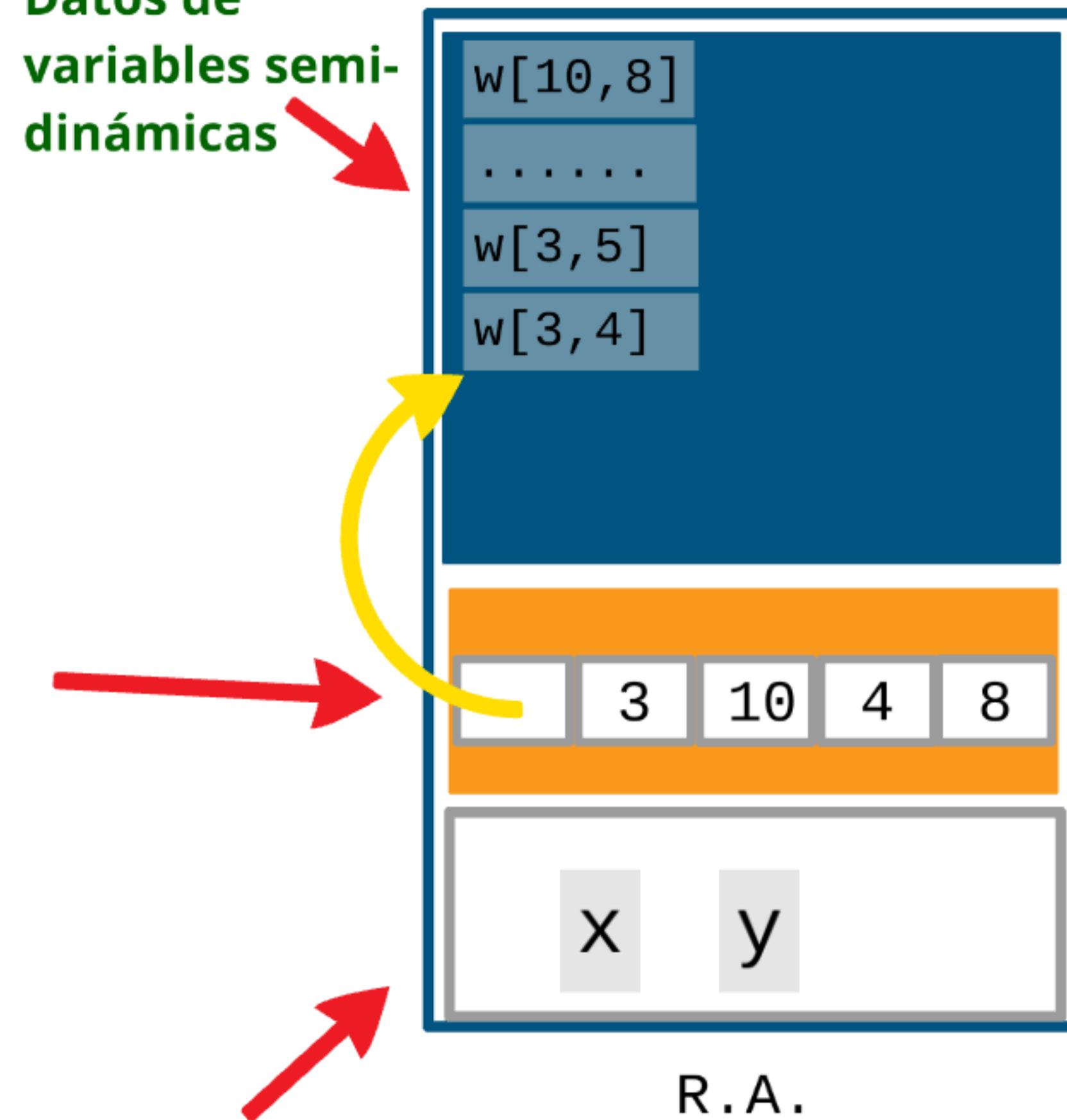
Si son variables no pueden estar en el código

```
procedure B;  
var i,j,k,l : integer;  
procedure A;  
x,y : integer;  
var w: array [i..j,k..l] of integer;  
begin  
    w[x,y] := 0;  
end;  
begin  
    i := 3;  
    j := 10;  
    k := 4;  
    l := 8;  
    A();  
end
```

Descriptores de Variables semi-dinámicas

- Los descriptores de las var. S.D. contienen los límites de las variables y un puntero a donde comienzan sus datos en el R.A..
- Si el lenguaje verifica límites necesita los 4, si no los verifica, con 3 límites es suficiente.

Datos de variables semi-dinámicas



Variables semi-estáticas

R.A.  
A()

# Verificación de límites en variables S.D.

```
procedure B;  
var j,l : integer;  
procedure A;  
x,y : integer;  
var w: array [3..j,4..l] of integer;  
begin  
    w[x,y] := 0;   
end;  
begin  
    j := 10;  
    l := 8;  
end  
  
if x<3 then "error"  
if x>10 then "error"  
if y<4 then "error"  
if y>8 then "error"  
// Instrucciones para llegar al  
descriptor con base +  
desplazamiento  
// Instrucciones para calcular  
la posición
```

El lenguaje inserta:

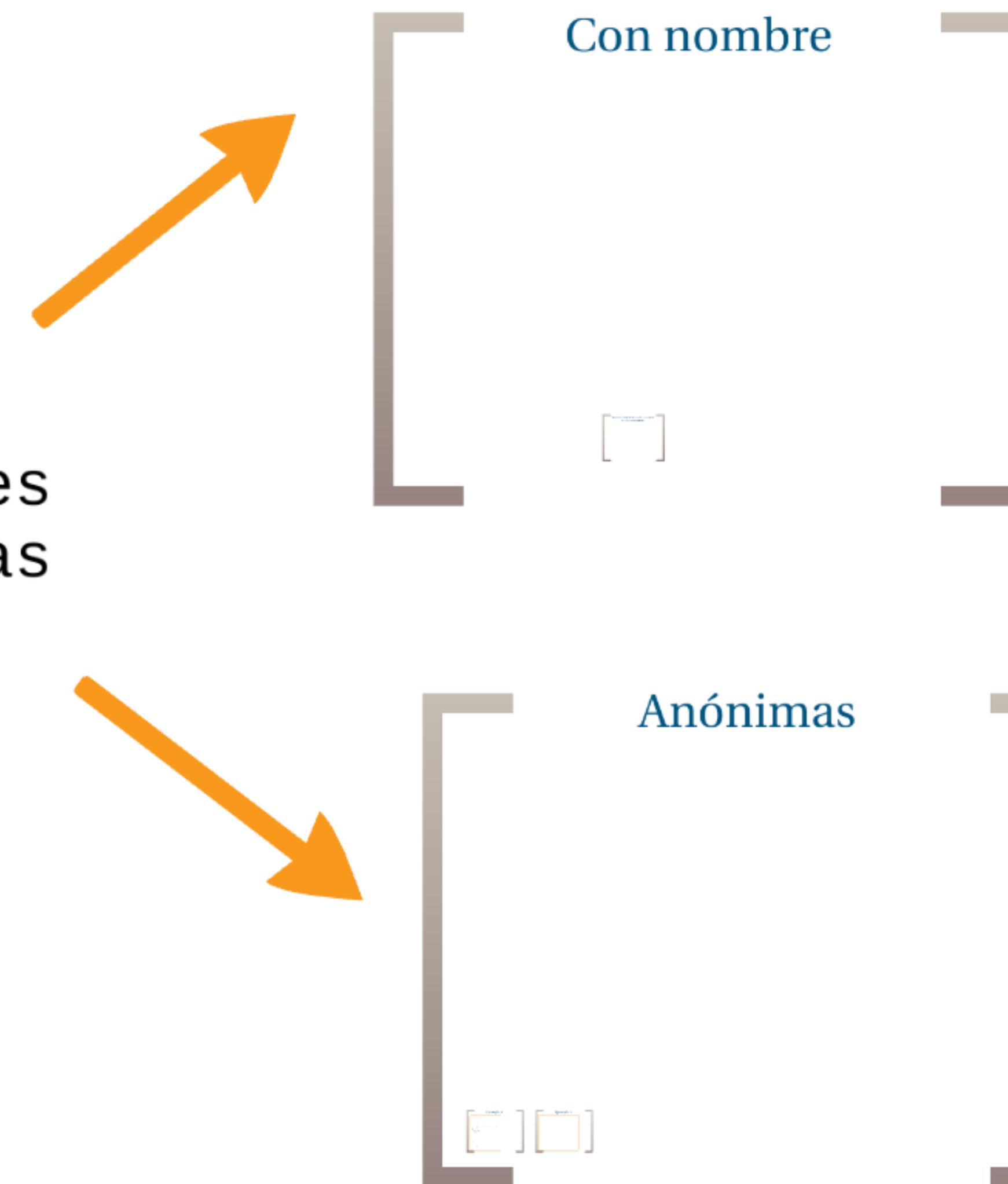
```
MOV R1, offset de x  
CMP R1, 3  
BLT ERROR  
MOV R2, Offset de j'  
CMP R1, R2  
BGT ERROR  
MOV R1, offset de y  
CMP R1, 4  
BLT ERROR  
MOV R2, Offset de l'  
CMP R1, R2 ; Comparar con el límite  
superior  
BGT ERROR  
// Instrucciones para acceder al  
elemento del arreglo
```

- Si los límites son constantes están en el código ejecutable.
- Si los límites son variables globales, sus valores son variables semi-estáticas

# Variables dinámicas

Variables que pueden cambiar de tamaño en cualquier momento

Variables  
dinámicas



# Con nombre

Ejemplo en Algol:

```
flex int a [1 : 0];  
a := (4, 1, 7, 8, -5, 3);  
a := (0,1);  
a := b;
```

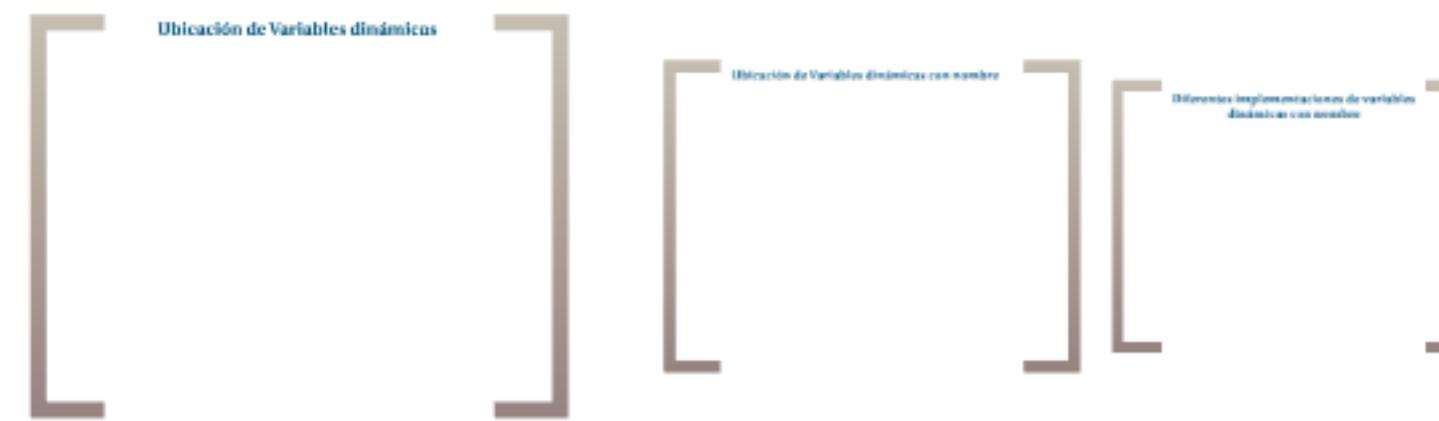
- flex declara un arreglo flexible.
- Los límites incongruentes significan que el arreglo es dinámico y se va a inicializar con la próxima asignación

El arreglo a tiene un tamaño de 6

Ahora el arreglo a tiene un tamaño de 2

Ahora tiene el tamaño del valor de b

¿Dónde está ubicada la variable a?



# Arreglos dinámicos con nombre

- Clase vector en C++, Rust
- Clase ArrayList en Java
- Clase ArrayList o List en .NET
- Clase OrderedCollection en Smalltalk
- Delphi, D y ADA como tipos básicos
- Perl y Ruby como tipo básico

## Ejemplo en Delphi:

```
procedure P;
var x : array of integer;
begin
  setlength(x,10);
  x[3] := 4;
  setlength(x,14);
  x[12] := 5;
end;
```

## Ejemplo en Java:

```
Vector v1 = new Vector();
Vector v2 = new Vector();
v1.add(0, 1);
v1.add(1, 2);
v1.add(2, v2);
```

¿Qué inconveniente tienen los lenguajes con tipos dinámicos para soportar arreglos dinámicos?

Ninguno. Son lenguajes que están preparados para que las variables cambien de tamaño

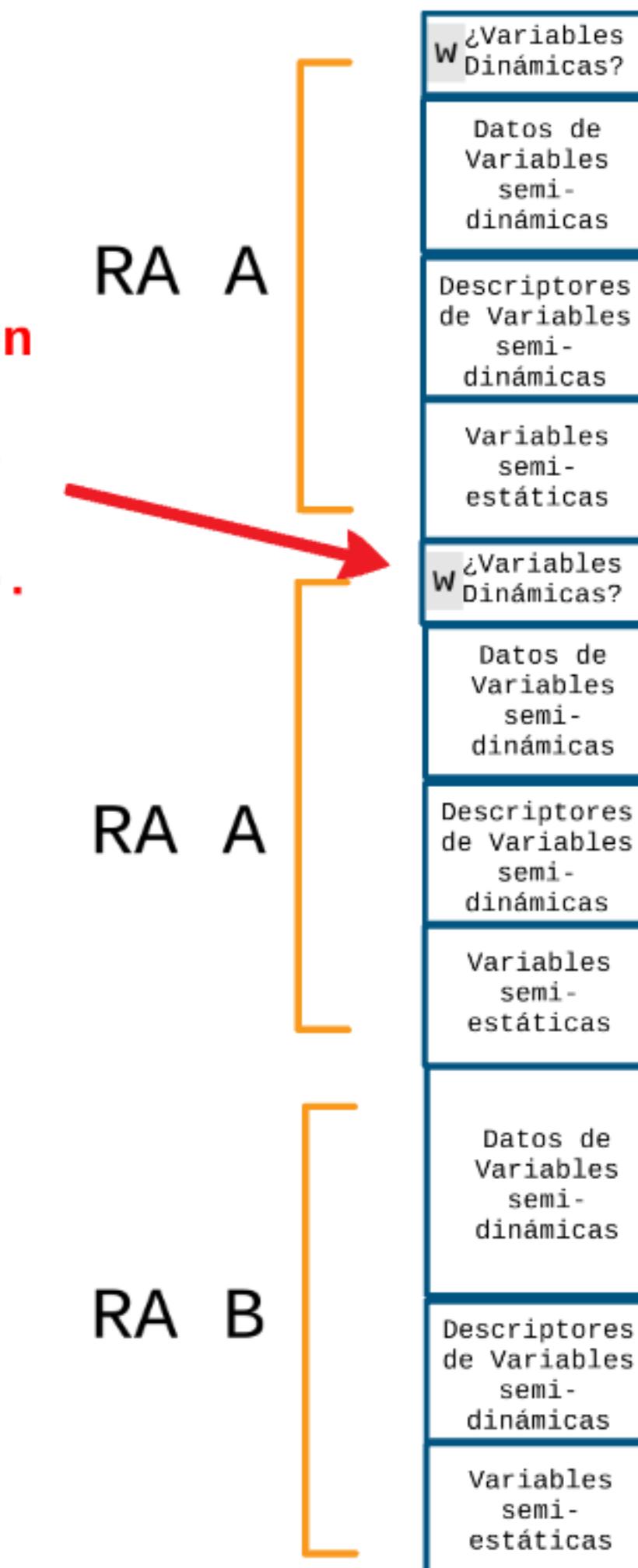
# Ubicación de Variables dinámicas

¿Pueden estar en los registros de activación?

## En lenguajes Tipo-Algol

```
procedure B;  
procedure A;  
var w : array of integer  
begin  
  w := [1,2,3];  
  ...  
  w := [1,2,3,4,5];  
  ...  
  A;  
end;  
begin  
  A;  
end
```

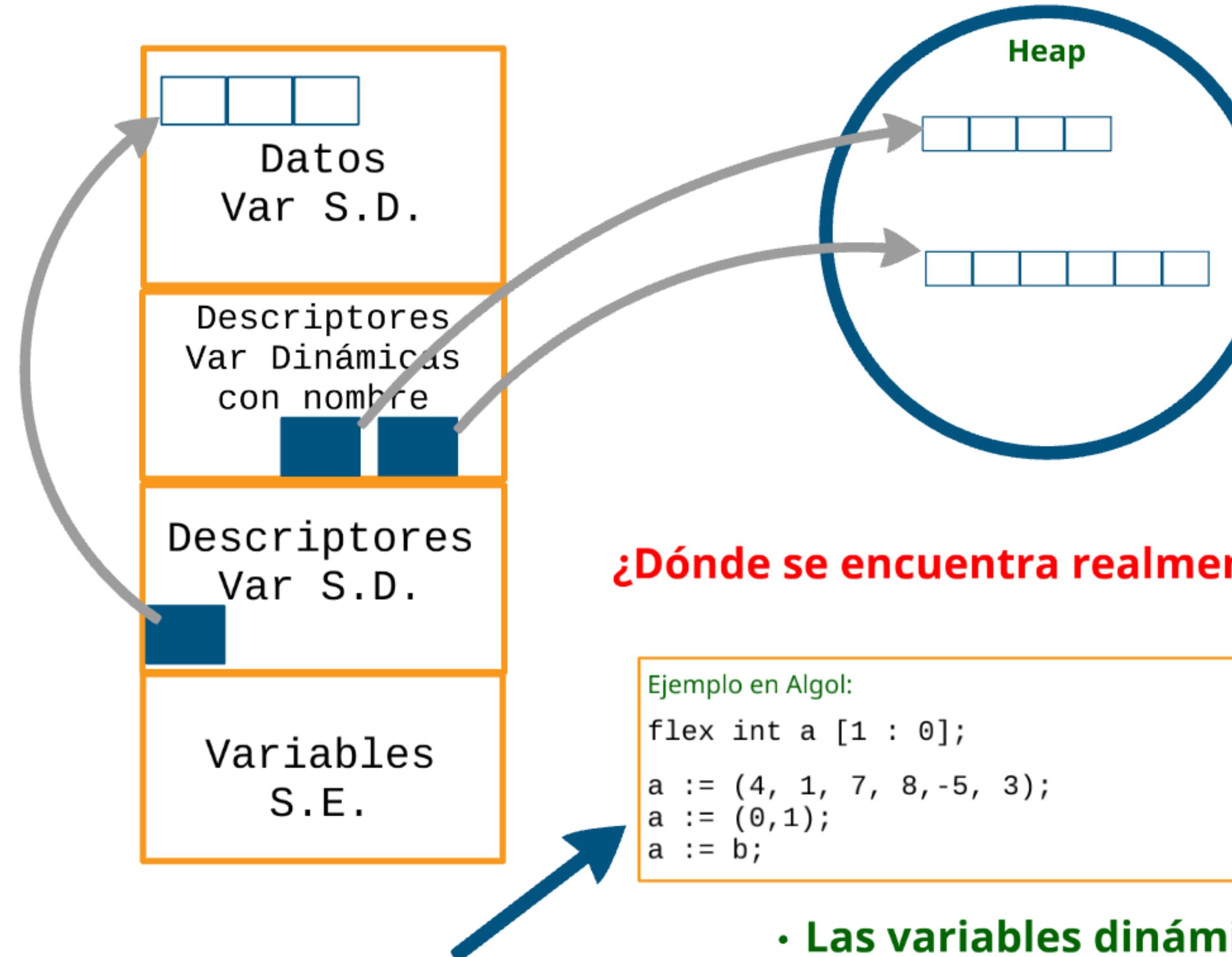
No pueden estar en la pila porque pueden cambiar de tamaño en cualquier momento.



Tienen que estar en otro lugar fuera de la pila de ejecución!

# Ubicación de Variables dinámicas con nombre

Se ubican en otra área de memoria llamada heap



El programador no hace  
nada para pedir memoria

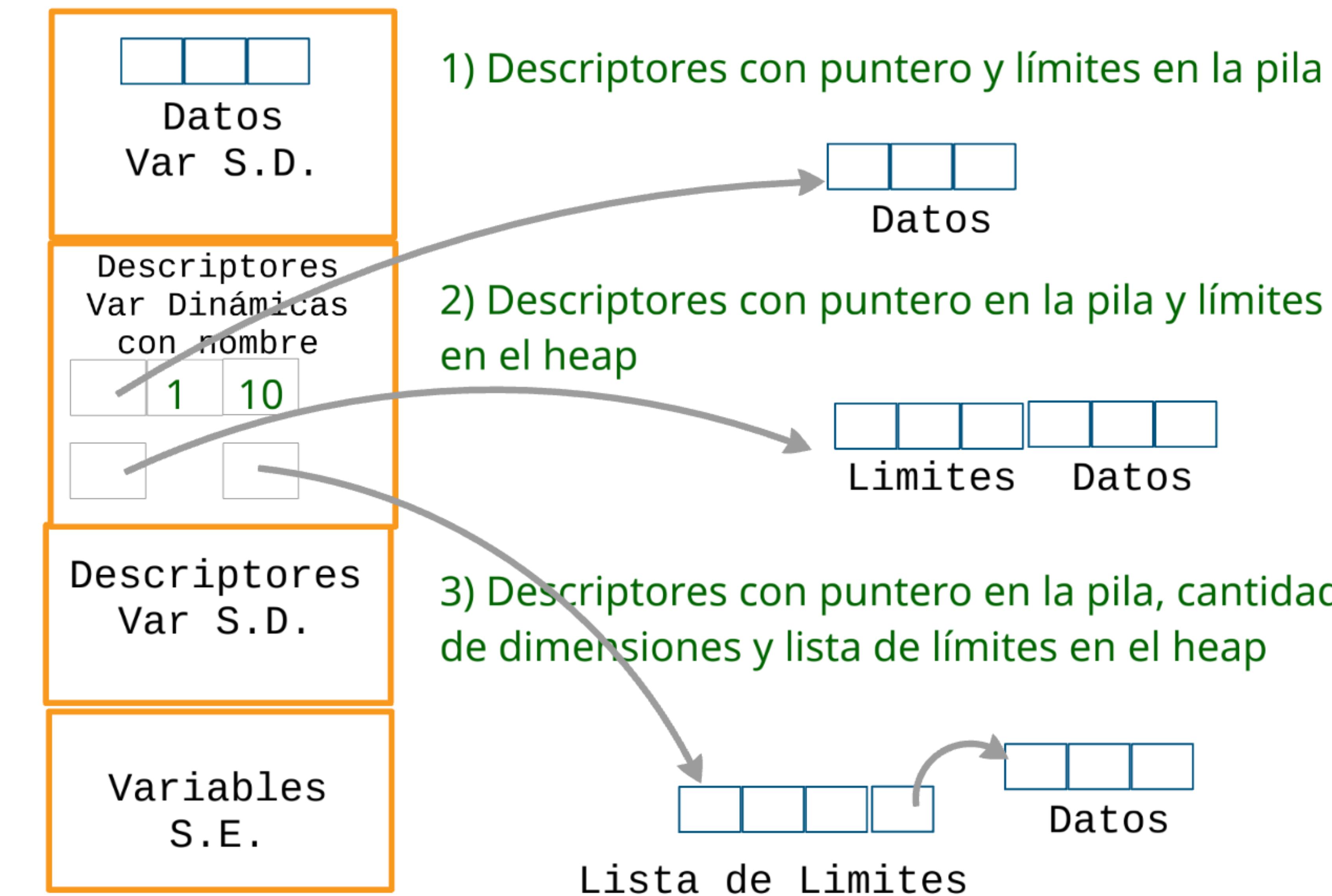
¿Dónde se encuentra realmente el heap?

Ejemplo en Algol:

```
flex int a [1 : 0];  
a := (4, 1, 7, 8, -5, 3);  
a := (0,1);  
a := b;
```

- Las variables dinámicas con nombre las administra el lenguaje

# Diferentes implementaciones de variables dinámicas con nombre



# Variables dinámicas

Variables que pueden cambiar de tamaño en cualquier momento

Clasificación de Variables

Variables  
dinámicas



Con nombre

Ejemplo en Algol:  
flex int a [1 : 0];  
a := (4, 1, 7, 8, -5, 3);  
a := (0,1);  
a := b;  
¿Dónde está ubicada la variable a?

- flex declara un arreglo flexible.
- Los límites incongruentes significan que el arreglo es dinámico y se va a inicializar con la próxima asignación

El arreglo a tiene un tamaño de 6  
Ahora el arreglo a tiene un tamaño de 2  
Ahora tiene el tamaño del valor de b

Arreglos dinámicos con nombre

- Clase vector en C++/Bust
- Clase ArrayList en Java
- Clase ArrayList<T> en .NET
- Clase ArrayList<T> en Smalltalk
- Delphi, D y ADA como tipos básicos
- Perl y Ruby como tipo báscio



Introducción a cómo se administra el heap



Anónimas



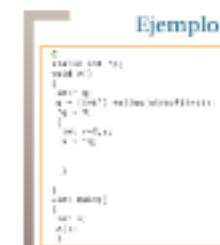
# Anónimas

## Pascal:

```
Procedure D;  
var p : ^integer;  
begin  
  new(p);  
  p^:= 3;  
  dispose(p);  
end.
```

## C:

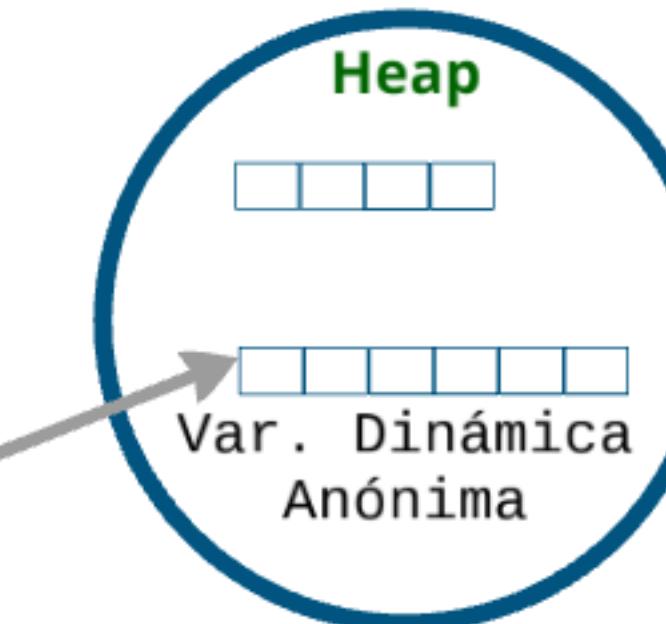
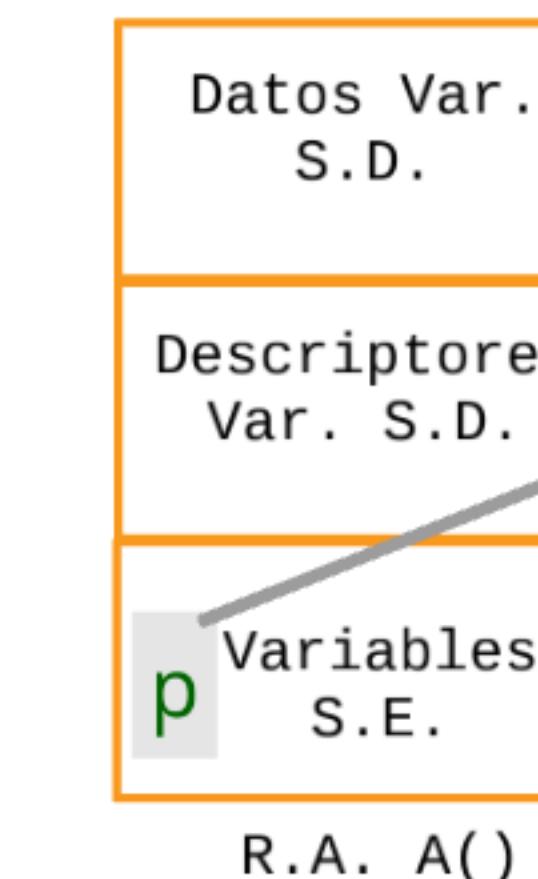
```
void A()  
{  
  int* p;  
  p = malloc(sizeof(int));  
  *p = 3;  
  free(p);  
}
```



`p=malloc(sizeof(int))`

¿Cuántas variables participan en esta sentencia?

- Una variable semi-estática p
- Una variable dinámica anónima que está en el heap



¿Dónde se encuentra realmente el heap?

¿Quién administra las variables dinámicas anónimas?

El programador

# Ejemplo 1

C:

```
static int *p;
void A()
{
    int* q;
    q = (int*) malloc(sizeof(int));
    *q = 3;
    {
        int r=2,s;
        s = *q;
        free(q);
        p= &r;
    }

}
int main()
{
    int x;
    A();
}
```

¿Cuántas variables hay?

¿Cómo se ordenan las variables por su alcance?

¿Cómo se ordenan las variables por su tiempo de vida?

¿Hay casos de variables en las que el alcance no coincide con su tiempo de vida?

# Ejemplo 2

C:

```
int main()
{
    int *q;
    q = (int*) malloc(sizeof(int));
    *q = 3;
    printf("q %d\n", q);
    printf("*q: %d\n", *q);

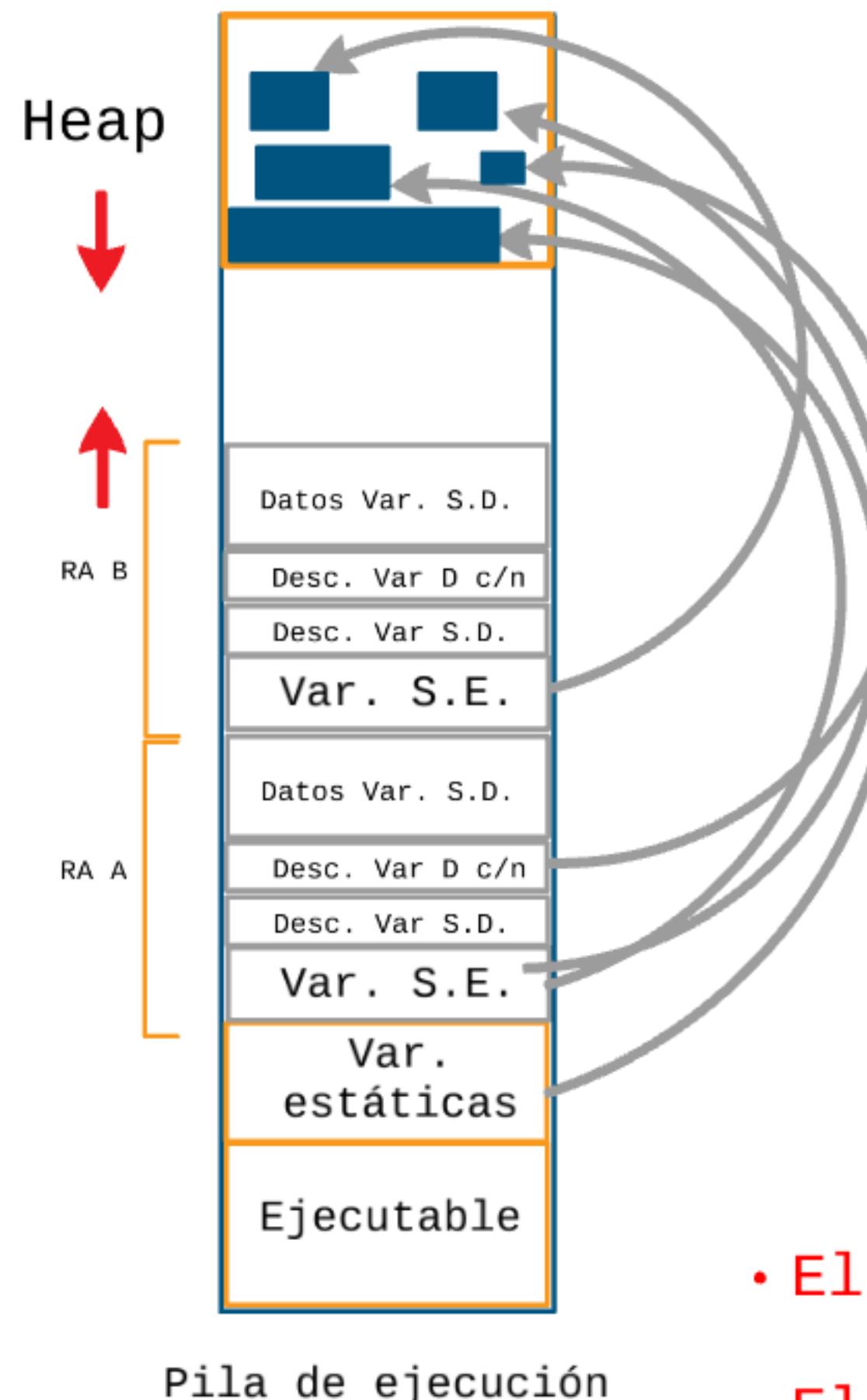
    int **p;
    p = (int**) malloc(sizeof(int*));
    printf("p %d\n", p);
    printf("*p: %d\n", *p);

    *p = 5; // ¿Qué significa esto?
    *p = (int*) malloc(sizeof(int));
    **p = 6;
    printf("*p %d\n", *p);
    printf("**p: %d\n", **p);
    return 0;
}
```

# Introducción a cómo se administra el heap

El heap en realidad no está fuera del área de memoria asignada por el S.O.

## ¿Qué sucede con esta solución?



- La pila de R.A. crece y decrece de forma controlada con la ejecución de las unidades.
- El heap tiende a crecer de forma descontrolada por su propia desorganización.

```
void A()
{
    int* p;
    p = malloc(sizeof(int)*100);
    ...
    free(p);
    p = malloc(sizeof(int)*89);
    ...
    free(p);
}
```

- El heap puede chocar contra el R.A. actual
- El nuevo R.A. puede chocar con el heap

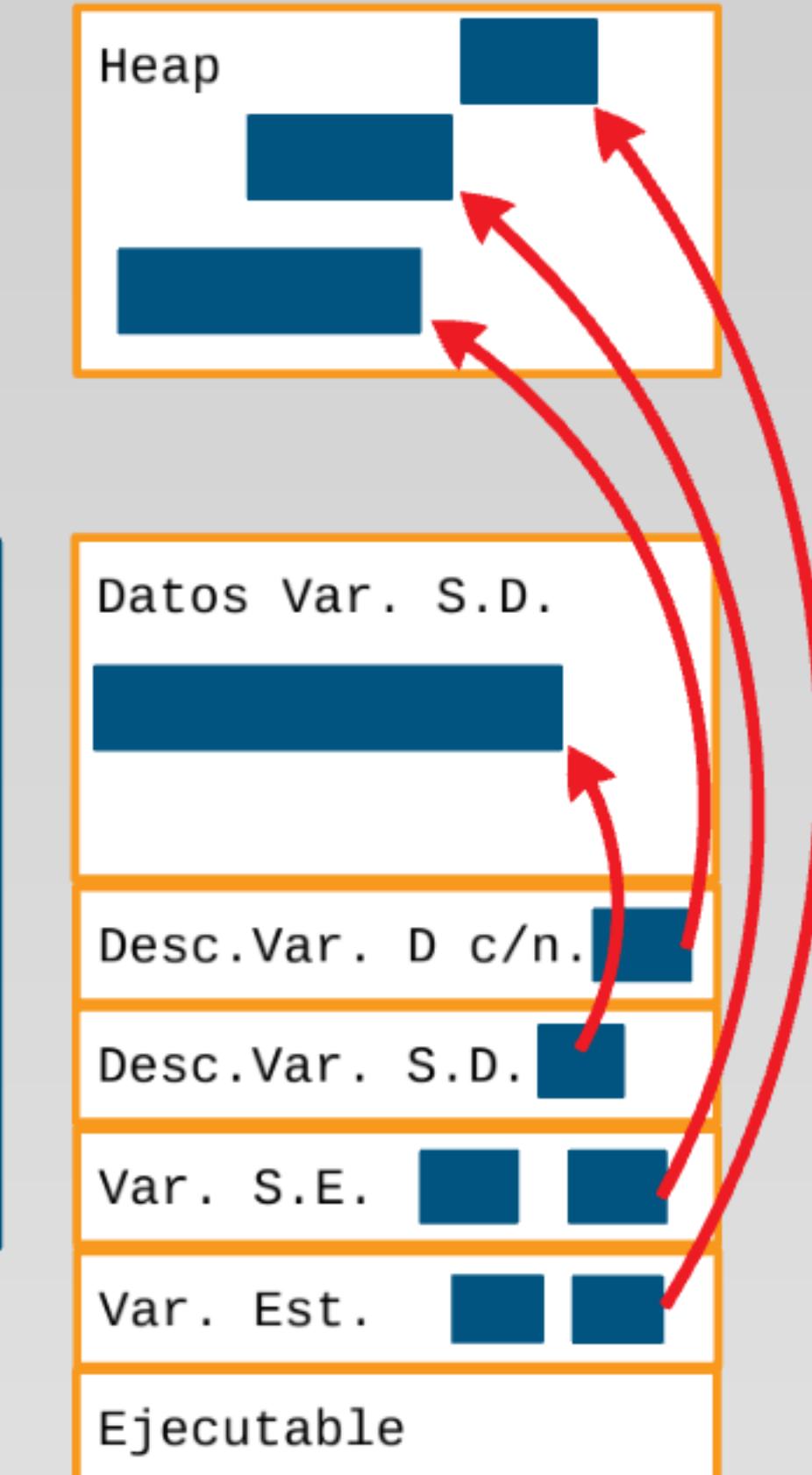
# Clasificación de Variables

- Estáticas
- Semi-estáticas
- Semi-dinámicas
- Dinámicas
  - Pueden cambiar de tamaño con cualquier

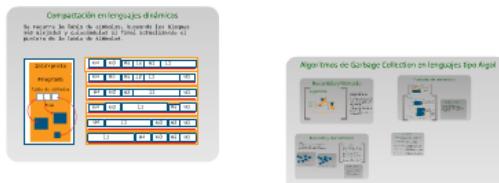
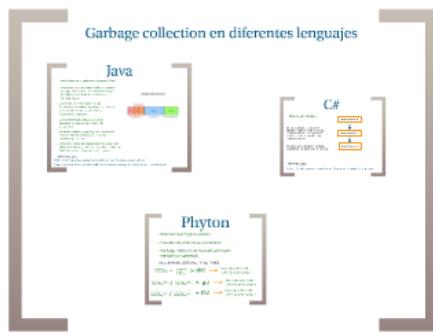
- Tamaño fijo
- Están en una dirección fija en la pila de ejecución
- Tamaño fijo
- Pueden estar en el registro de activación
- ¿Dónde más pueden estar?

- Tamaño que puede cambiar SOLO entre diferentes R.A. (invocaciones) de la unidad.
- Están en el registro de activación

- Con nombre
  - Administradas por el lenguaje
  - Descriptor en el R.A. o en el heap
- Anónimas
  - Administradas por el programador



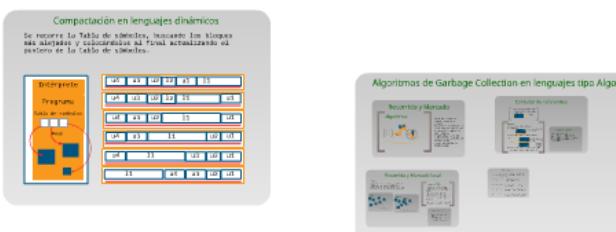
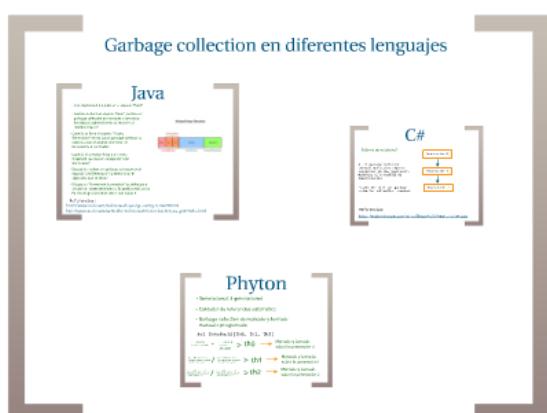
- Administración de memoria
- Garbage collection
- Fragmentación



# Lenguajes de Programación

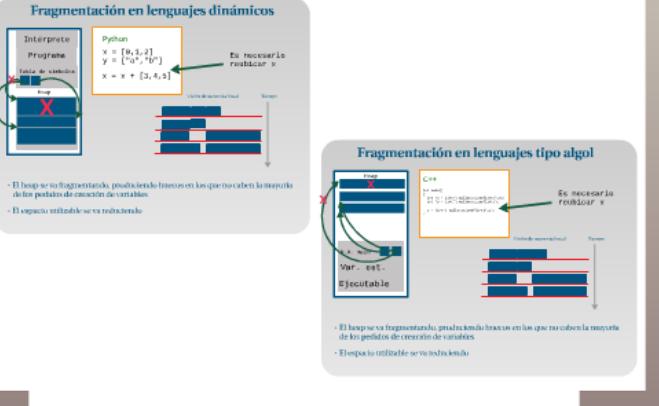
- Introducción
- Sintáxis
- Semántica - Binding, Tipos de lenguaje
- Alcance dinámico y Alcance estático
- Pila de ejecución de lenguajes tipo Algol
- Clasificación de variables en lenguajes de pila
  - Manejo dinámico de memoria - Garbage Collection
  - Interacción con el Sistema Operativo - Conurrencia
  - Pasaje de parámetros
  - Asignaciones
  - Tipos: coerciones y conversiones
  - Objetos

- Administración de memoria
- Garbage collection
- Fragmentación



# Problemas de asignación dinámica de memoria

## Fragmentación



## Lenguajes tipo Algol

### Problemas con punteros

- Un bloque es borrado y quedan punteros a él.
- Un bloque marcado como usado ya no es referenciado por ningún puntero.

#### Fabricar un puntero colgado

```
int* p;
int* q;
p=(int*) malloc(100*sizeof(int));
q = p;
free(p);
```

Ahora q es un puntero colgado

#### Fabricar Garbage 1

```
int* p;
int* q;
p=(int*) malloc(100*sizeof(int));
q=(int*) malloc(100*sizeof(int));
q = p;
q = p;
```

El bloque anterior apuntado por q es garbage

#### Fabricar Garbage 2

```
m1 a = new m1();
m2 b = new m2();
a = new m3();
/* el bloque apuntado inicialmente por a ya no es accesible */
```

Algunos lenguajes tienen errores de compilación que detectan este problema.

## Lenguajes tipo Algol

### Punteros colgados

Usuario

### Solución

Usuario

### Garbage

Usuario

Usuario  
(a veces el lenguaje)

### Fragmentación

Usuario / Lenguaje

Usuario  
(a veces el lenguaje)

**Lenguajes tipo Algol**  
Tipos de errores y soluciones comunes

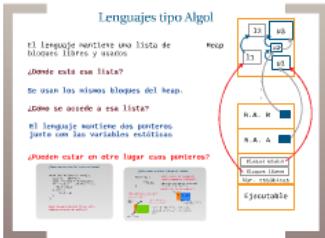
```
Clase a = new Clase(); // en Java
int* p = malloc(sizeof(Clase)); // en C
printf("p = %d\n", p); // en C
void p(); // en Pascal
new (p); // en C/C++
```

Algunos lenguajes de programación tienen errores de compilación que detectan este problema.

**Lenguajes tipo Algol**  
Tipos de errores y soluciones comunes

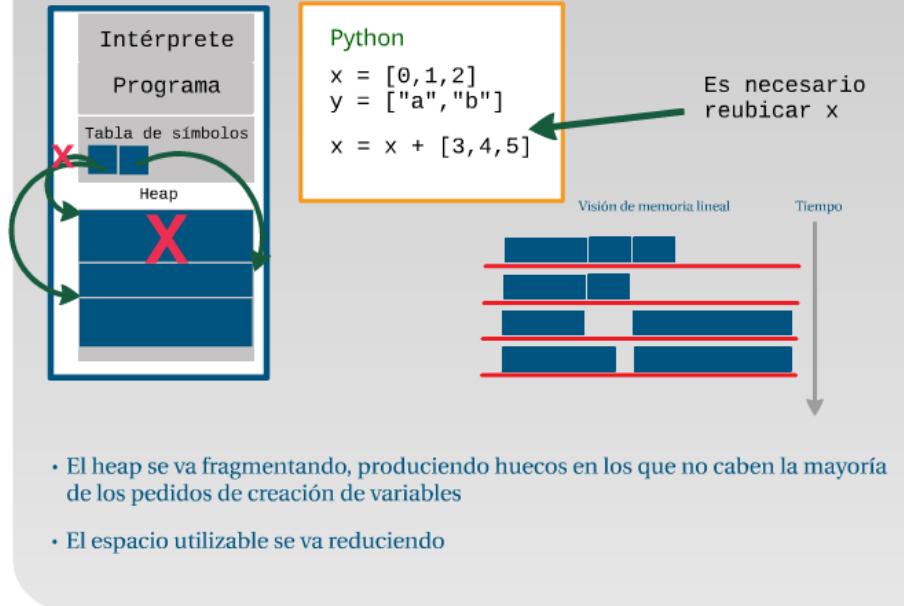
```
new(x); // en C
delete(x); // en C++
delete[] (x); // en Pascal
```

Algunos lenguajes de programación tienen errores de compilación que detectan este problema.

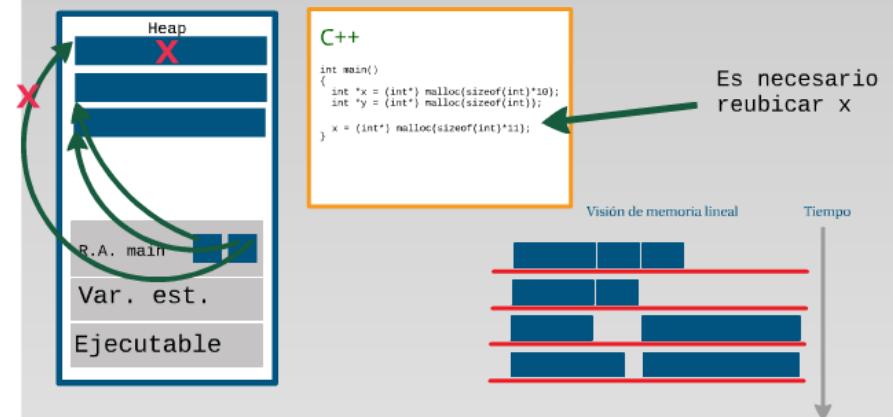


# Fragmentación

## Fragmentación en lenguajes dinámicos

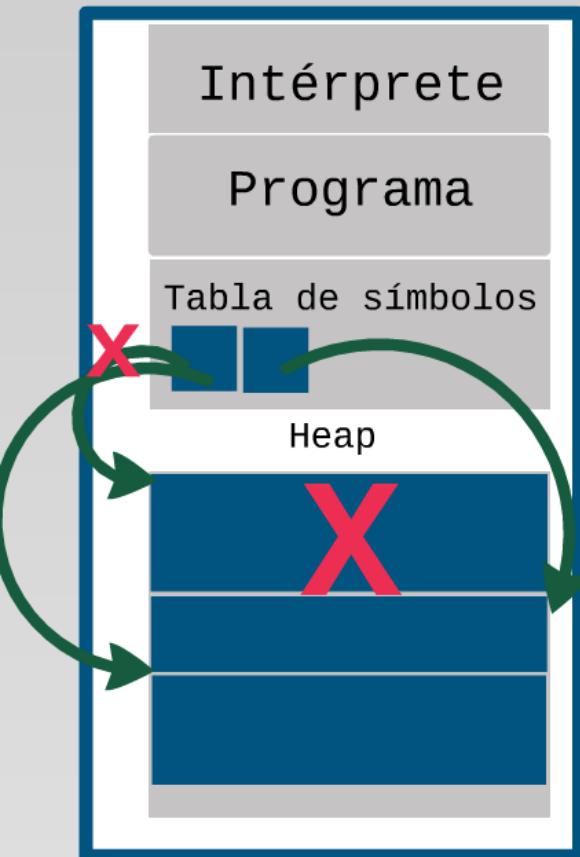


## Fragmentación en lenguajes tipo algol



- El heap se va fragmentando, produciendo huecos en los que no caben la mayoría de los pedidos de creación de variables
- El espacio utilizable se va reduciendo

# Fragmentación en lenguajes dinámicos



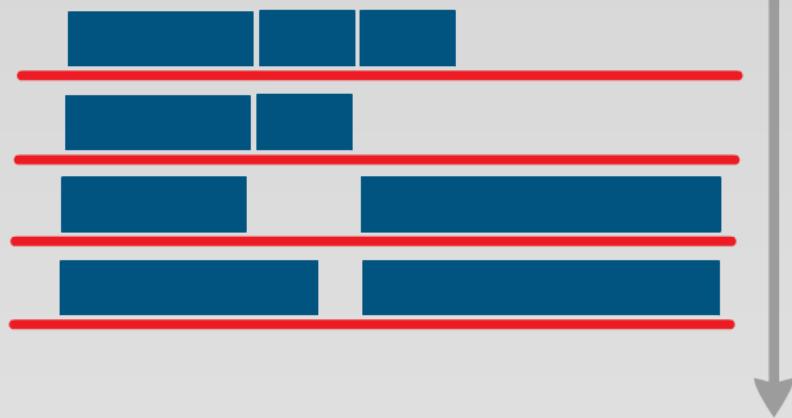
Python

```
x = [0, 1, 2]
y = ["a", "b"]
x = x + [3, 4, 5]
```

Es necesario  
reubicar x

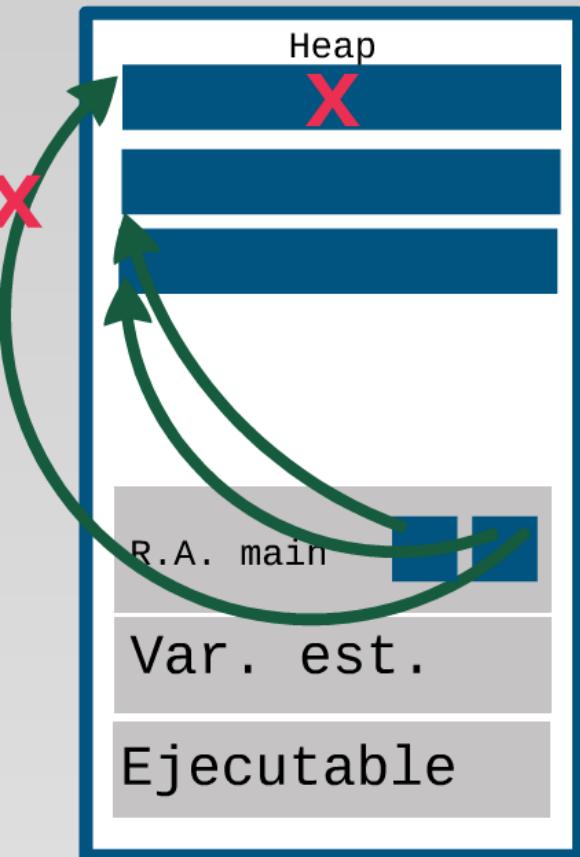
Visión de memoria lineal

Tiempo



- El heap se va fragmentando, produciendo huecos en los que no caben la mayoría de los pedidos de creación de variables
- El espacio utilizable se va reduciendo

# Fragmentación en lenguajes tipo algol



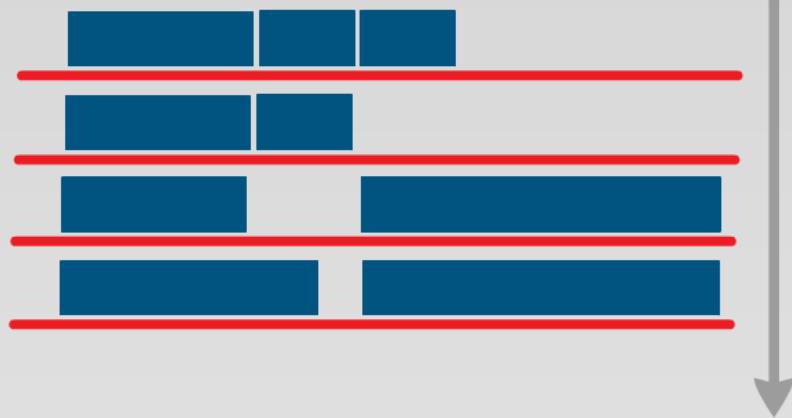
C++

```
int main()
{
    int *x = (int*) malloc(sizeof(int)*10);
    int *y = (int*) malloc(sizeof(int));
    x = (int*) malloc(sizeof(int)*11);
}
```

Es necesario  
reubicar x

Visión de memoria lineal

Tiempo



- El heap se va fragmentando, produciendo huecos en los que no caben la mayoría de los pedidos de creación de variables
- El espacio utilizable se va reduciendo

## Lenguajes Tipo-Algol

### Ejemplos de pedido y asignación de memoria

```
Clase a = new Clase(); // en Java
```

```
int* p = malloc(sizeof(int)); // En C
```

```
float* p = malloc(100*sizeof(float)); // En C
```

```
var p : integer; {En Pascal}  
new (p);
```

¿Quién se encarga de encontrar un bloque libre en el heap y hacer que el puntero apunte a ese bloque?

- De la asignación al puntero se encarga el programador.
- De encontrar un bloque libre se encarga el lenguaje (a pedido del programador)

## Lenguajes Tipo-Algol

### Ejemplos de pedido y liberación de memoria

free(p) // En C

delete(p) // En C++

Dispose(p); {En Pascal}

¿Quién se encarga de hacer que el bloque del heap apuntado por esas variables vuelva a estar disponible?

- De solicitarlo, el programador.
- De realmente "marcar" ese bloque como libre, el lenguaje

# Lenguajes tipo Algol

El lenguaje mantiene una lista de bloques libres y usados

**¿Dónde está esa lista?**

Se usan los mismos bloques del heap.

**¿Cómo se accede a esa lista?**

El lenguaje mantiene dos punteros junto con las variables estáticas

**¿Pueden estar en otro lugar esos punteros?**

¿Cómo funcionan malloc() o new realmente?

```
void *malloc(size_t size) {
    void *p = sbrk(0);
    void *request = sbrk(size);
    if (request == (void*) -1) {
        return NULL;
    } else {
        return p;
    }
}
```

¿Qué inconvenientes tiene esta implementación de malloc?

¿Qué sucede con el free(), dispose() o delete?

```
void f() {
    ...
    free(p);
}
```

¿Cómo sabe el lenguaje cuanta memoria liberar?

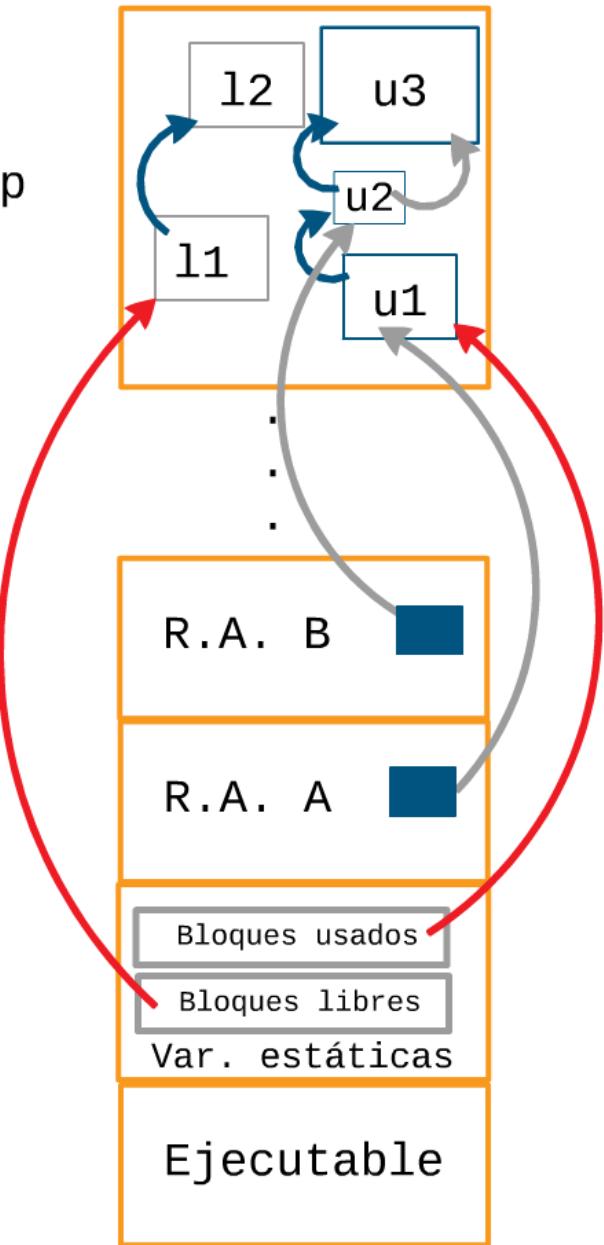
Almacena información adicional por cada bloque pedido  
¿dónde está esa información?

¿Qué sucede con el espacio liberado? ¿Se reutiliza?  
¿Cómo se hace?



¿Qué sucede con nuestro malloc?  
Ahora debe buscar un bloque en la lista de libres y si no lo encuentra debe pedirle al S.O. más espacio de heap

Heap



## ¿Cómo funcionan malloc() o new realmente?

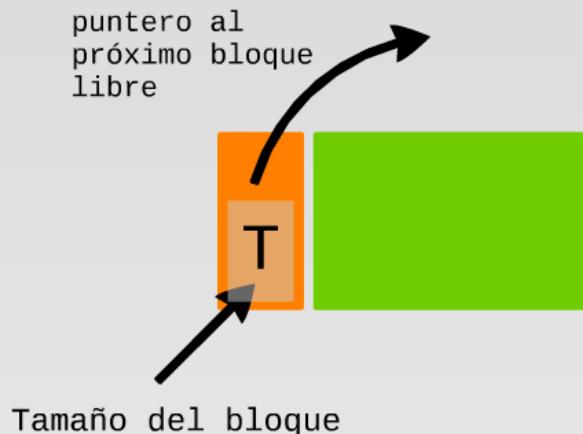
```
void *malloc(size_t size) {  
    void *p = sbrk(0);  
    void *request = sbrk(size);  
    if (request == (void*) -1) {  
        return NULL;  
    } else {  
        return p;  
    }  
}
```

¿Qué inconvenientes tiene esta implementación de malloc?

## ¿Qué sucede con el free(), dispose() o delete?

```
void f() {  
    ...  
    free(p);  
}
```

¿Qué sucede con el espacio liberado? ¿Se reutiliza?  
¿Cómo se hace?



¿Cómo sabe el lenguaje cuanta memoria liberar?

Almacena información adicional por cada bloque pedido

¿dónde está esa información?



¿Qué sucede con nuestro malloc?

Ahora debe buscar un bloque en la lista de libres y si no lo encuentra debe pedirle al S.O. más espacio de heap

# Lenguajes tipo Algol

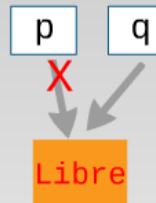
## Problemas con punteros

- Un bloque es borrado y quedan Punteros colgados
- Un bloque marcado como usado ya no es referenciado por ningún puntero. Garbage

### Fabricar un puntero colgado

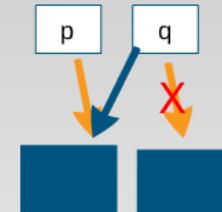
```
int* p;
int* q;
p=(int*) malloc(100*sizeof(int))
q = p;
free(p);
```

Ahora q es un puntero colgado



### Fabricar Garbage 1

```
int* p;
int* q;
p=(int*) malloc(100*sizeof(int));
q=(int*) malloc(100*sizeof(int));
q = p;
```

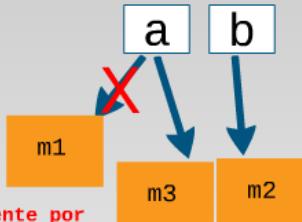


El bloque antes apuntado por q es garbage

### Fabricar Garbage 2

```
m1 a = new m1();
m2 b = new m2();
a = new m3();
```

/\* el bloque apuntado inicialmente por a ya no es accesible \*/



### Fabricar (potencial) Garbage 3

```
Object a = new m1();
Object b = new m2();
...
if (a.chequeo())
    a=b;
```

/\* dependiendo del valor que retorna a.chequeo() el bloque apuntado por a puede (o no) convertirse en garbage\*/

### Fabricar Garbage 4

¿Qué imprime este programa en Java?

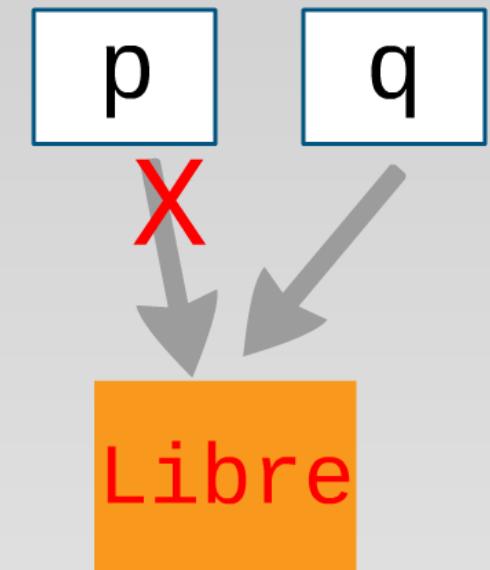
```
String s1 = "A";
String s2 = s1;
s1 = s1 + "B";
System.out.println(s1);
System.out.println(s2);
```

Al ejecutar s1=s1+"B" se crea un nuevo objeto y se hace que s1 lo apunte

es referenciado por ni

## Fabricar un puntero colgado

```
int* p;  
int* q;  
p=(int*) malloc(100*sizeof(int))  
q = p;  
free(p);
```



Ahora q es un puntero colgado

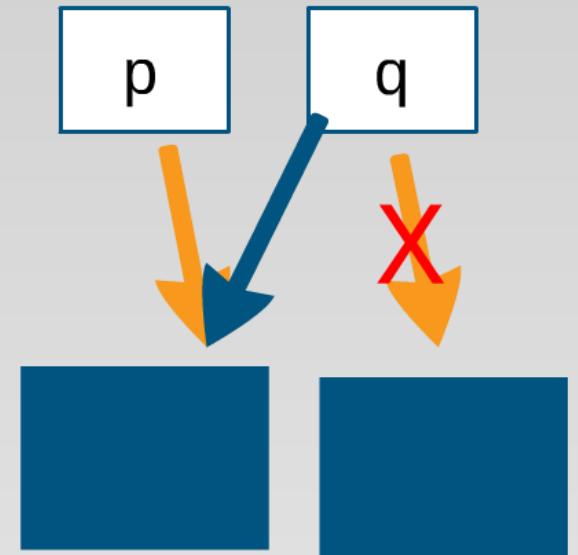
# ntero.



# Garbage

## Fabricar Garbage 1

```
int* p;  
  
int* q;  
p=(int*) malloc(100*sizeof(int));  
  
q=(int*) malloc(100*sizeof(int));  
  
q = p;
```

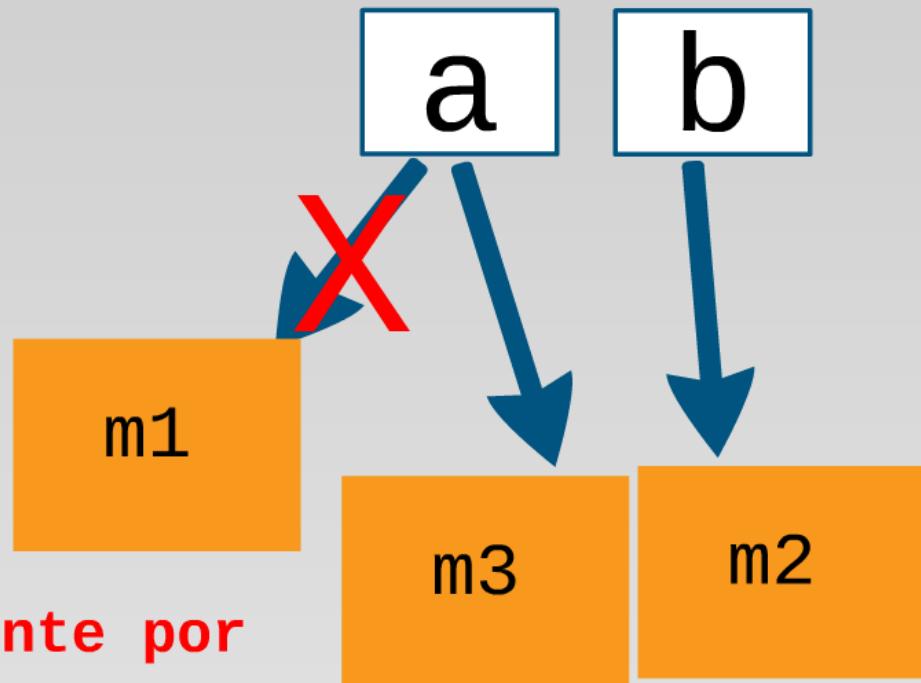


El bloque antes  
apuntado por q es  
garbage

## Fabricar Garbage 2

```
m1 a = new m1();  
m2 b = new m2();  
  
a = new m3();
```

/\* el bloque apuntado inicialmente por  
a ya no es accesible \*/



## Fabricar (potencial) Garbage 3

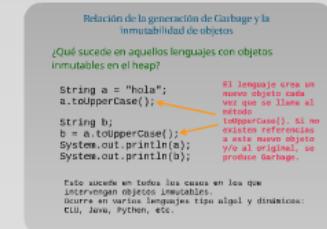
```
Object a = new m1();
Object b = new m2();
...
if (a.chequeo( ))
    a=b;

/* dependiendo del valor que retorna
a.chequeo() el bloque apuntado por a
puede (o no) convertirse en garbage*/
```

# Fabricar Garbage 4

¿Qué imprime éste programa en Java?

```
String s1 = "A";
String s2 = s1;
s1 = s1 + " B";
System.out.println(s1);
System.out.println(s2);
```



Al ejecutar `s1=s1+"B"` se crea un nuevo objeto y se hace que `s1` lo apunte

## Relación de la generación de Garbage y la inmutabilidad de objetos

¿Qué sucede en aquellos lenguajes con objetos inmutables en el heap?

```
String a = "hola";  
a.toUpperCase();
```

```
String b;  
b = a.toUpperCase();  
System.out.println(a);  
System.out.println(b);
```

El lenguaje crea un nuevo objeto cada vez que se llama al método `toUpperCase()`. Si no existen referencias a este nuevo objeto y/o al original, se produce Garbage.

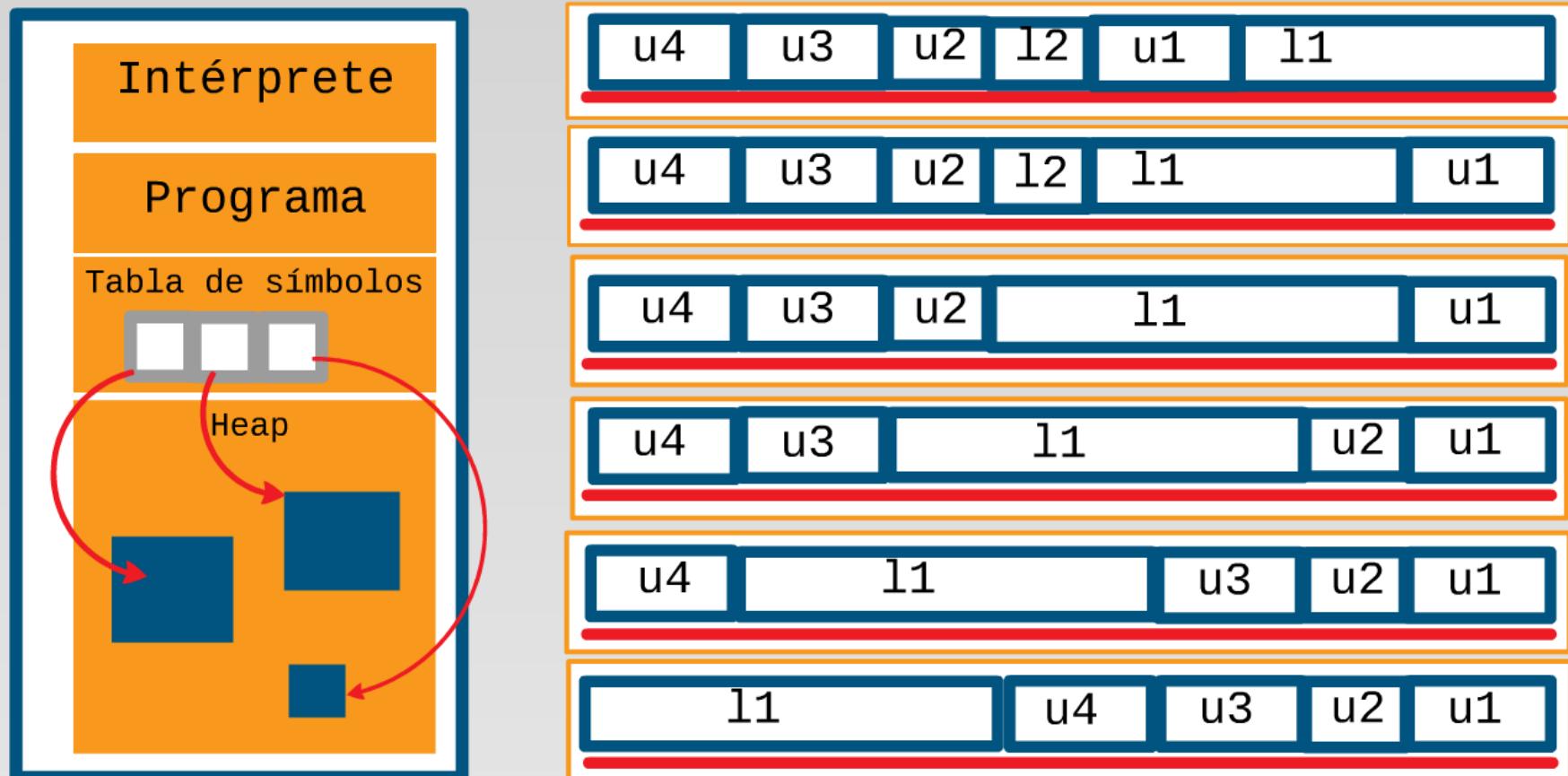
Esto sucede en todos los casos en los que intervengan objetos inmutables.  
Ocurre en varios lenguajes tipo algol y dinámicos:  
CLU, Java, Python, etc.

## Lenguajes tipo Algol

|                   | Problema           | Solución                         |
|-------------------|--------------------|----------------------------------|
| Punteros colgados | Usuario            | Usuario                          |
| Garbage           | Usuario            | Usuario<br>(a veces el lenguaje) |
| Fragmentación     | Usuario / Lenguaje | Usuario<br>(a veces el lenguaje) |

# Compactación en lenguajes dinámicos

Se recorre la Tabla de símbolos, buscando los bloques más alejados y colocándolos al final actualizando el puntero de la tabla de símbolos.



# Algoritmos de Garbage Collection en lenguajes tipo Algol

## Recorrido y Marcado

### Algoritmo:



- Garbage Collector concentrado en el tiempo
- Se activa al chocar la pila con el heap o por invocación explícita del usuario.
- Inadmissible para un contexto de sistema en tiempo real
- Pocos lenguajes lo implementan

## Recorrido y Marcado local

### Idea:

Algoritmo basado en contador de referencias pero soluciona el problema de estructuras cíclicas



### Algoritmo:

- A partir del puntero borrado:
  - Paso 1: Recorrer los bloques decrementando los contadores y marcando los nodos como posible basura.
  - Paso 2: Se recorre nuevamente buscando nodos que no tienen ningún hijo diferente de cero, se desmarcan y se actualizan los contadores de sus hijos.
  - Paso 3: Todos los bloques marcados se mueven a la lista de libres.

### Consideraciones:

• Solución al problema de las estructuras cíclicas.  
• Garbage Collector necesita de revisar todo el espacio de memoria cada vez que el contador de referencia de un determinado objeto se vuelve cero.

## Contador de referencias

- Algoritmo distribuido en el tiempo
- A cada bloque se le asigna un campo entero para almacenar la cantidad de referencias



### Algoritmo:

- Cada vez que se crea un nuevo bloque:
  - p = malloc(100);
  - en la lista de bloques
  - se incrementa el contador de alta un nodo con la cantidad de referencias inicializada en 1
- Cada vez que se realiza una asignación del tipo:
  - p = q;
  - se decrementa la apuntada por p y se incrementa la apuntada por q
- Si el contador se vuelve 0 se convierte en basura

### Consideraciones:

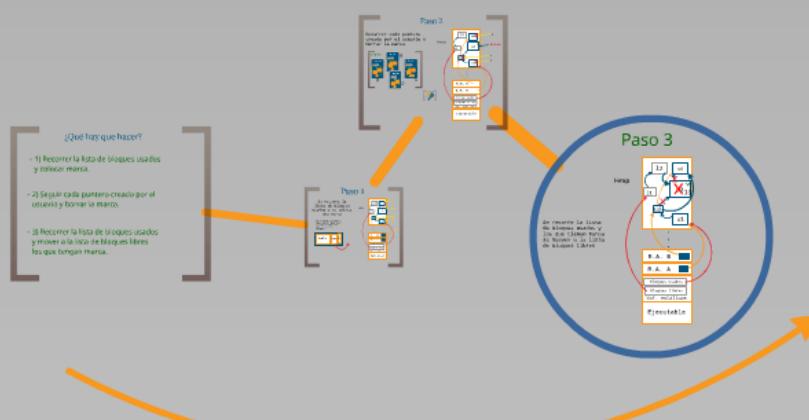
- Simple de implementar
- No es muy costoso temporalmente
- Ejecución dispersa en el tiempo
- No funciona para estructuras cíclicas
- No soluciona el problema de la fragmentación

## Otros métodos:

- Copiado: se mantienen dos espacios de memoria y se realiza GC sobre uno de ellos mientras se usa el otro
- Generacionales: se dividen los objetos según su tiempo de vida
- Incrementales: se realiza el GC por pasos, en cada uno se recorre el estado anterior
- Concurrentes: se ejecuta el GC en otro hilo de ejecución diferente al del programa
- Distribuidos: se realizan diferentes espacios de memoria se realizan diferentes tipos de GC

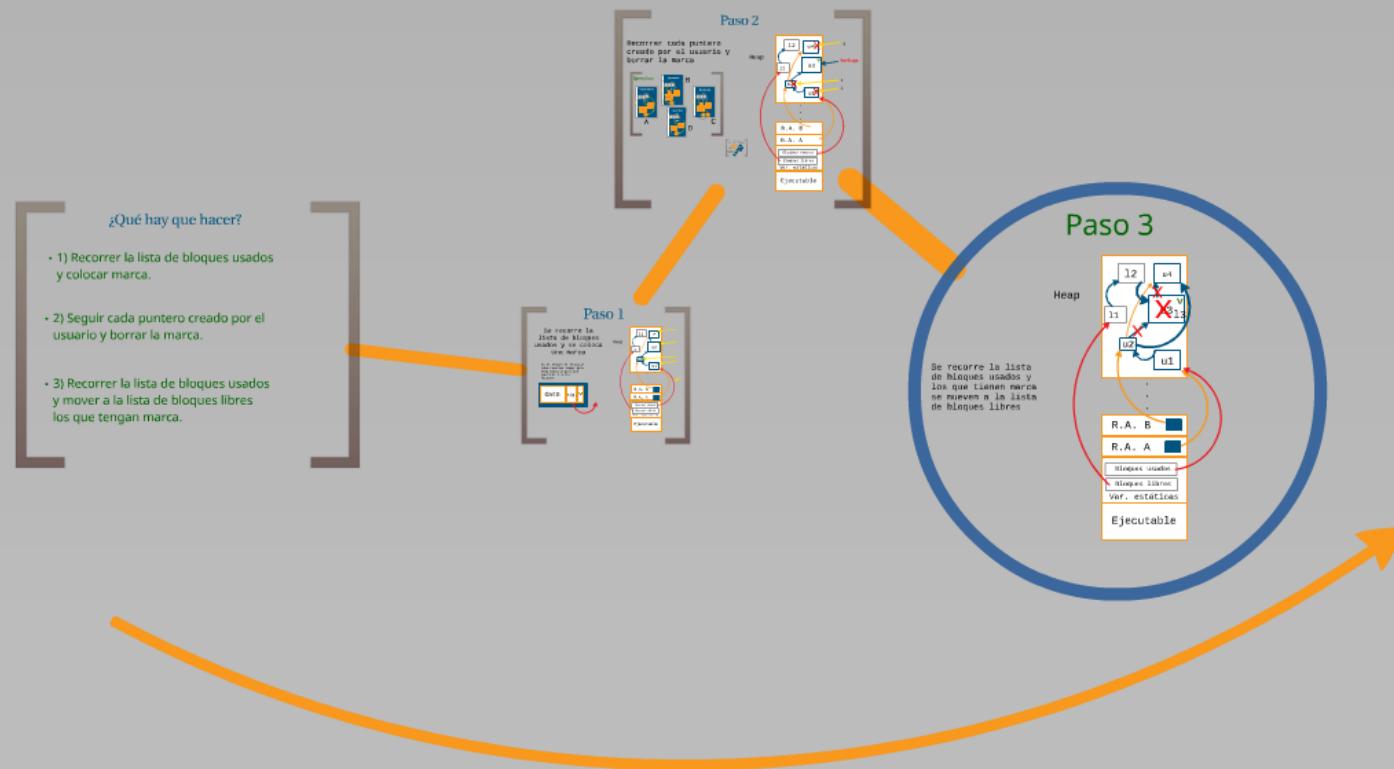
# Recorrido y Marcado

## Algoritmo:



- Garbage Collector concentrado en el tiempo
- Se activa al chocar la pila con el heap o por invocación explícita del usuario.
- Inadmissible para un contexto de sistema en tiempo real
- Pocos lenguajes lo implementan

# Algoritmo:



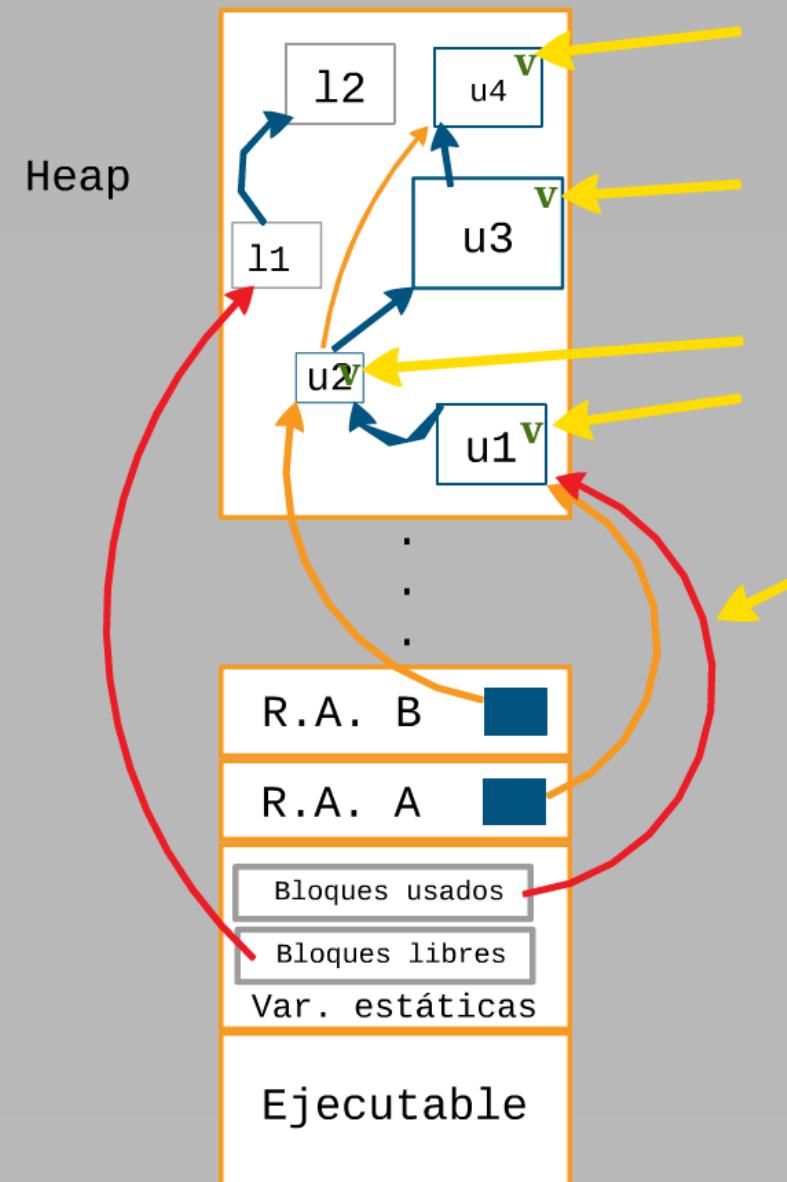
# ¿Qué hay que hacer?

- 1) Recorrer la lista de bloques usados y colocar marca.
- 2) Seguir cada puntero creado por el usuario y borrar la marca.
- 3) Recorrer la lista de bloques usados y mover a la lista de bloques libres los que tengan marca.

# Paso 1

Se recorre la lista de bloques usados y se coloca una marca

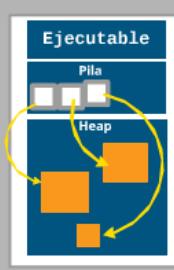
En el bloque el lenguaje debe reservar lugar para esta marca y para los punteros a otros bloques.



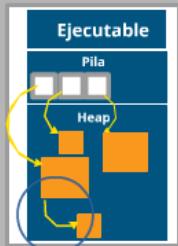
# Paso 2

Recorrer cada puntero  
creado por el usuario y  
borrar la marca

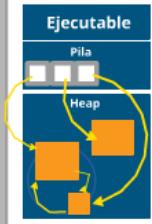
Ejemplos:



A

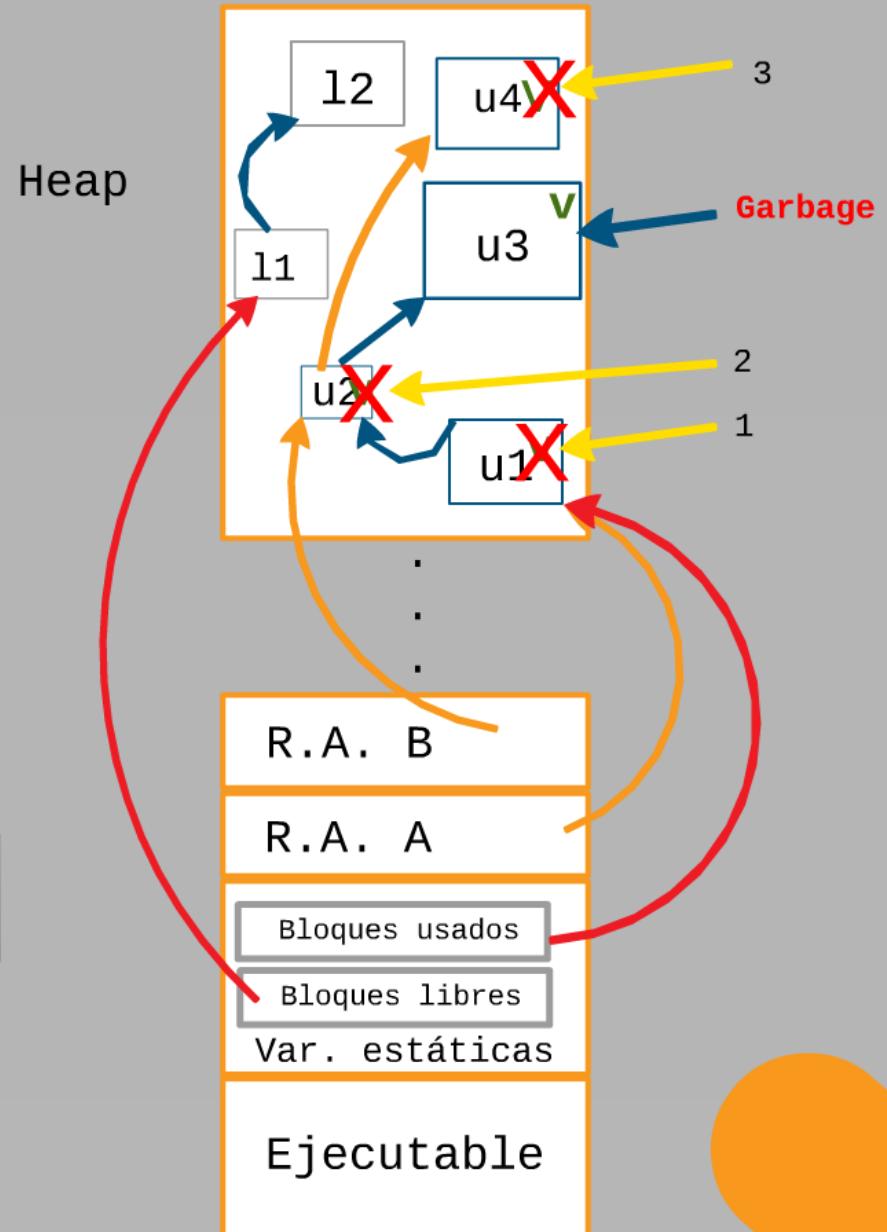


B

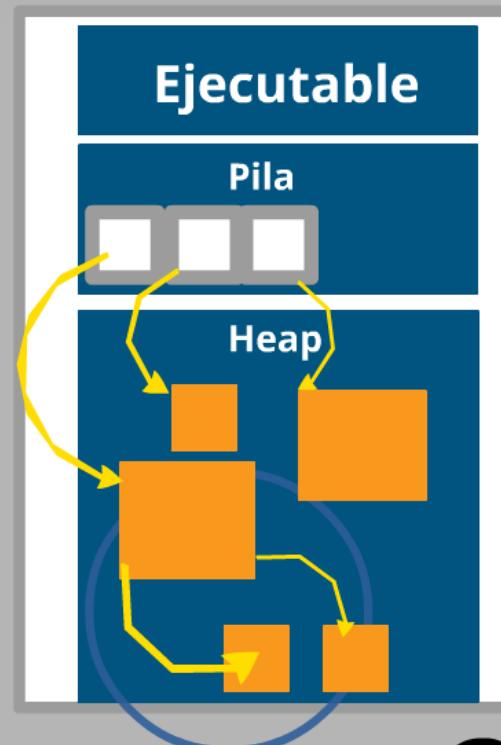
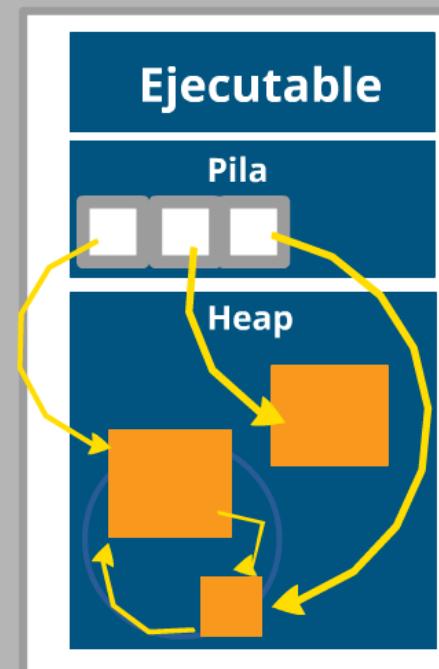
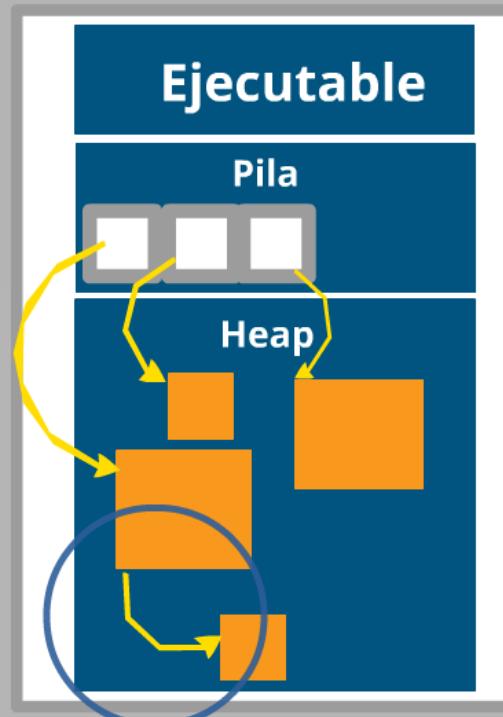
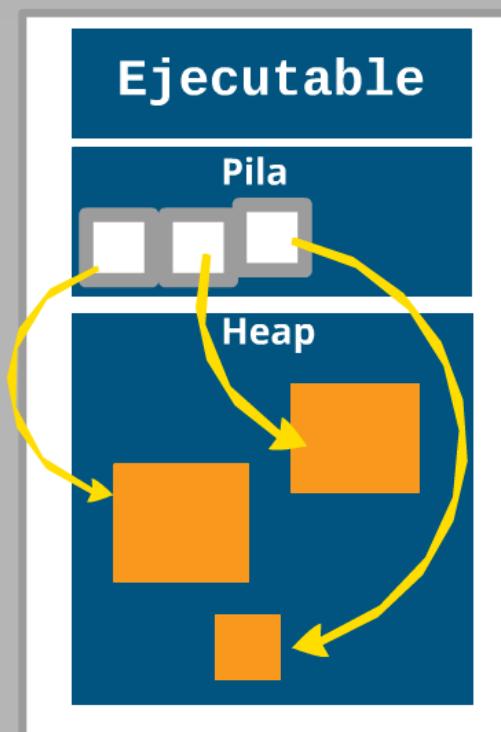


C

D



# Ejemplos:

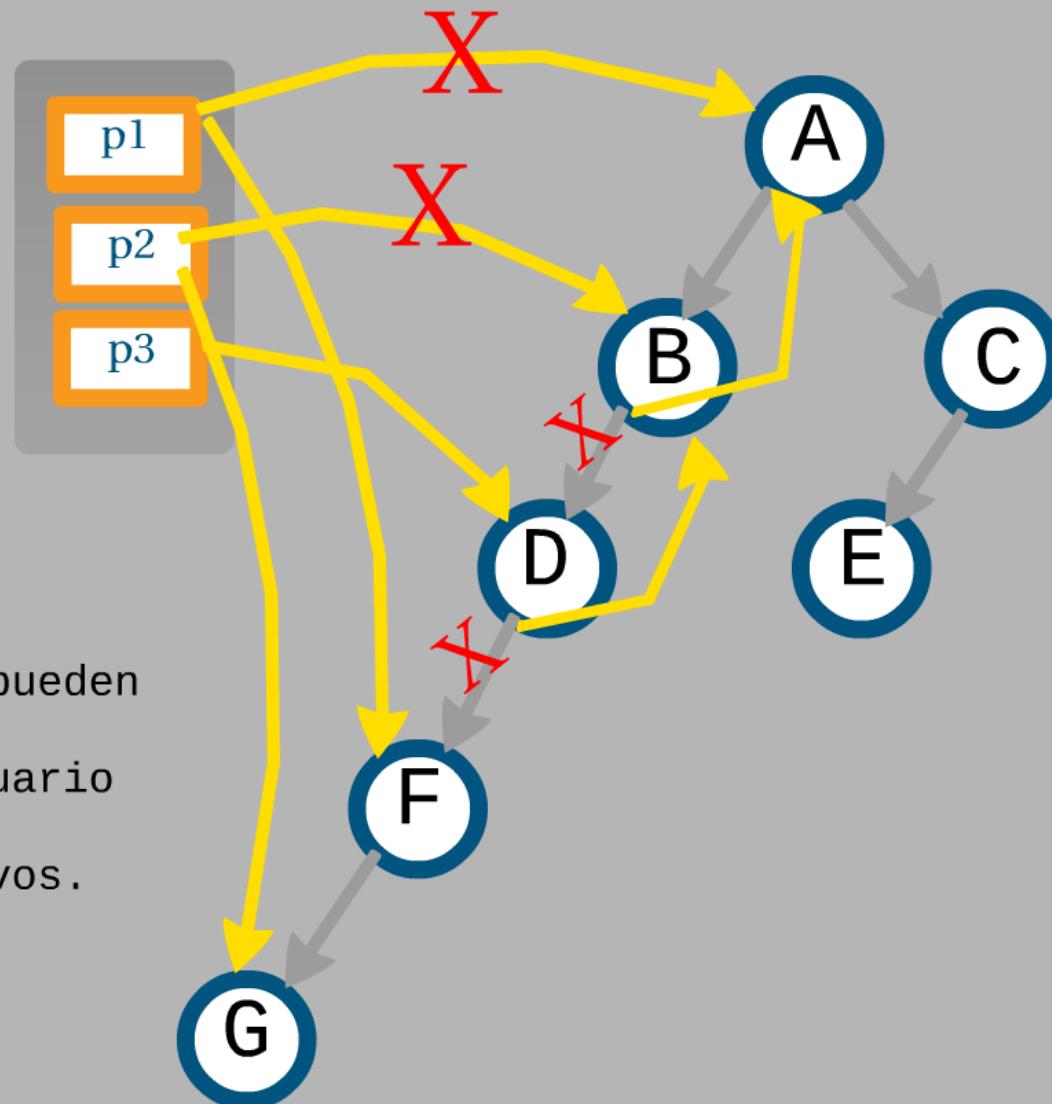


# Recorrer estructuras del usuario en un contexto de poca memoria

¿Cómo se recorren las estructuras si la pila ya chocó con el heap?

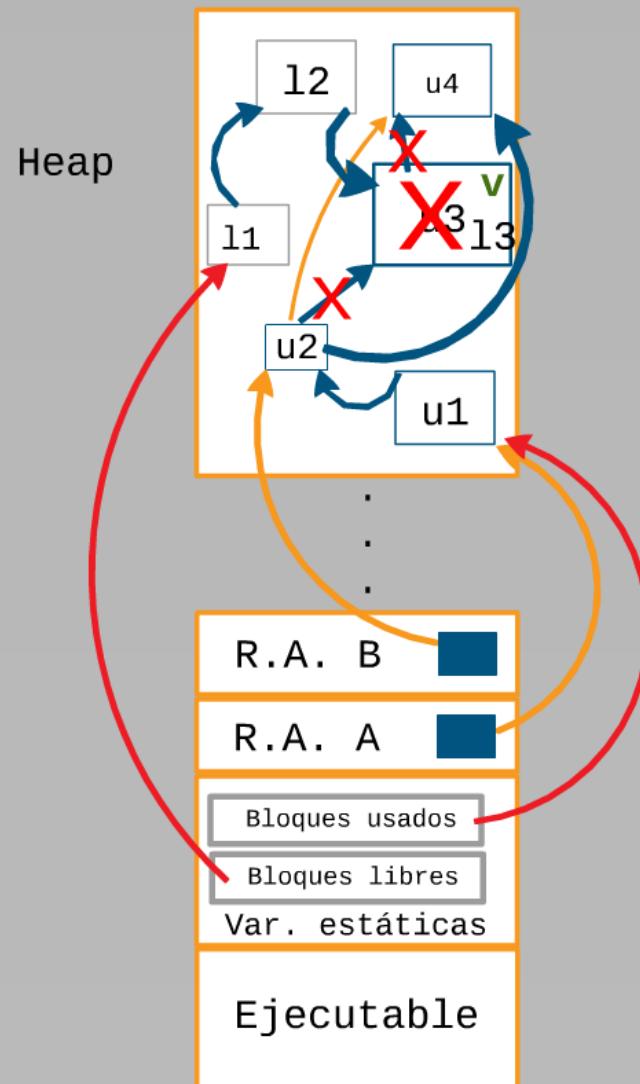
Variables  
estáticas

De esta manera se pueden  
ir recorriendo las  
estructuras del usuario  
sin necesidad de  
algoritmos recursivos.



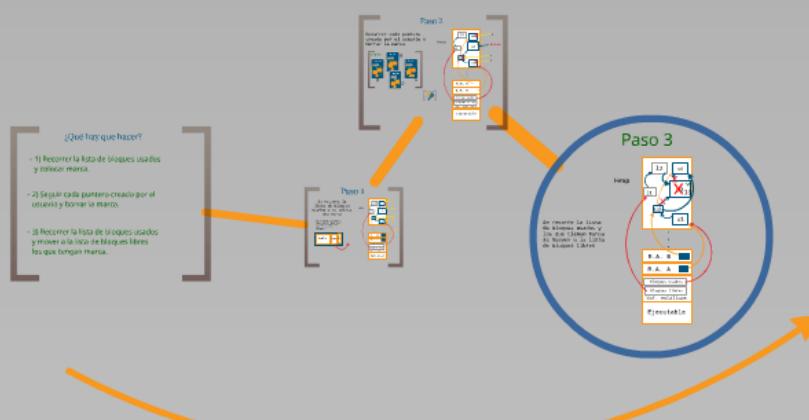
# Paso 3

Se recorre la lista de bloques usados y los que tienen marca se mueven a la lista de bloques libres



# Recorrido y Marcado

## Algoritmo:



- Garbage Collector concentrado en el tiempo
- Se activa al chocar la pila con el heap o por invocación explícita del usuario.
- Inadmissible para un contexto de sistema en tiempo real
- Pocos lenguajes lo implementan

# Contador de referencias

- Algoritmo distribuido en el tiempo
- A cada bloque se le agrega un campo entero para almacenar la cantidad de referencias



## Algoritmo:

- Cada vez que se crea un nuevo bloque

```
p = malloc(100);
```

en la lista de bloques usados se da de alta un nodo con la cuenta de referencias inicializada en 1

A diagram of a memory block. It consists of two adjacent blue rectangles. The left rectangle is labeled 'datos'. The right rectangle contains the number '1'. An arrow points from the text 'inicializada en 1' to the number '1'.

- Cada vez que se realiza una asignación del tipo:

```
p = q
```

se decrementa lo apuntado por p y se incrementa lo apuntado por q

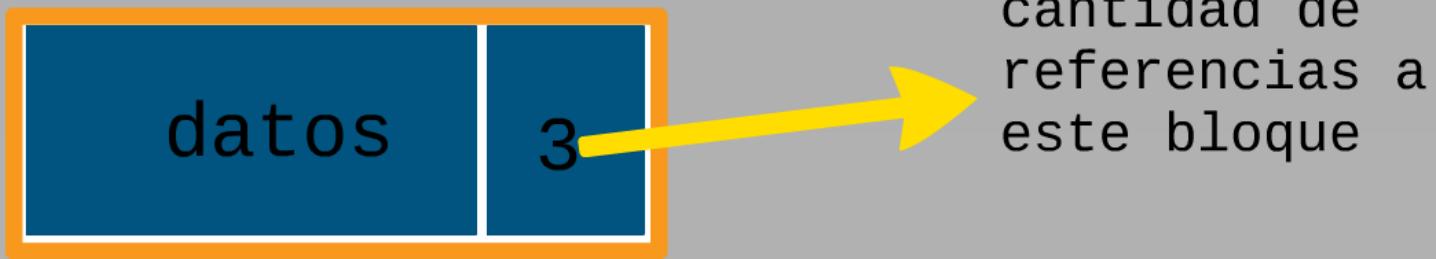
A diagram illustrating pointer assignment. It shows two memory blocks. The top block has 'datos' and 'n-1'. The bottom block has 'datos' and 'n+1'. A curved arrow labeled 'antes de la asignación' points from the 'p' pointer to the 'n-1' block. Another curved arrow points from the 'q' pointer to the 'n+1' block.

- Si el contador se vuelve 0 se convierte en garbage  
¿Qué pasa con las estructuras del usuario?

## Consideraciones:

- Simple de implementar
- No es muy costoso temporalmente
- Ejecución dispersa en el tiempo
- No funciona para estructuras cíclicas
- No soluciona el problema de la fragmentación

- Algoritmo distribuido en el tiempo
- A cada bloque se le agrega un campo entero para almacenar la cantidad de referencias



goritmo:

## Algoritmo:

- Cada vez que se crea un nuevo bloque

```
p = malloc(100);
```

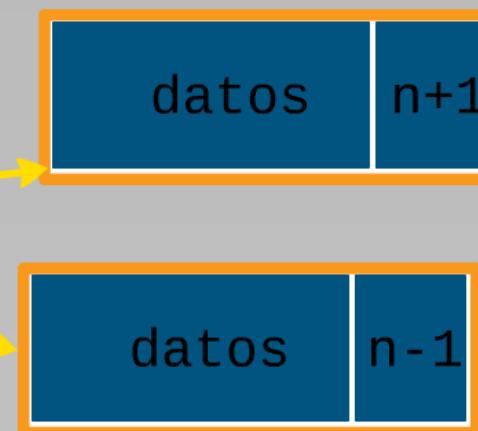


en la lista de bloques usados se da de alta un nodo con la cuenta de referencias inicializada en 1

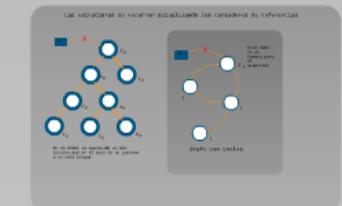
- Cada vez que se realiza una asignación del tipo:

$p = q$

*antes de la asignación*



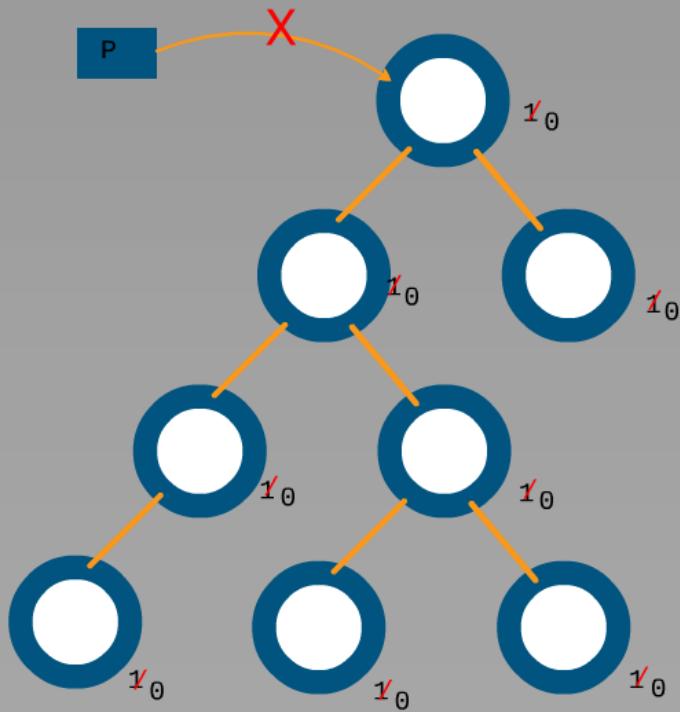
se decrementa lo apuntado por p y se incrementa lo apuntado por q



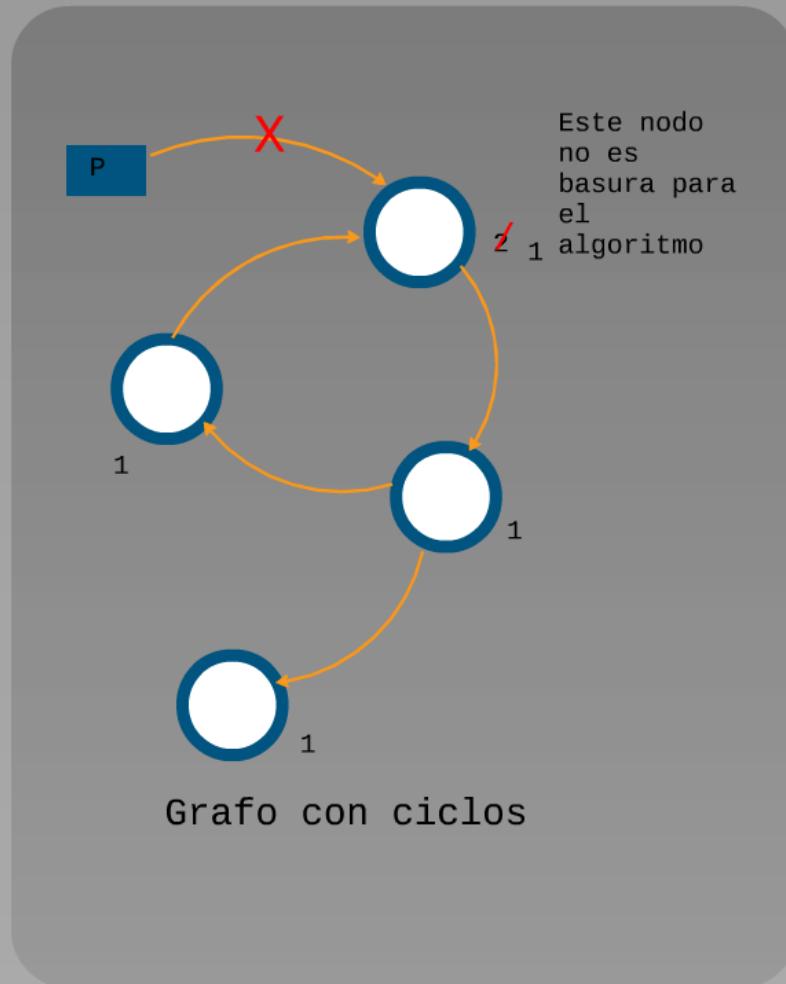
- Si el contador se vuelve 0 se convierte en garbage

¿Qué pasa con las estructuras del usuario?

Las estructuras se recorren actualizando los contadores de referencias



En un árbol la operación es más costosa que en el caso de un puntero a un solo bloque



# Consideraciones:

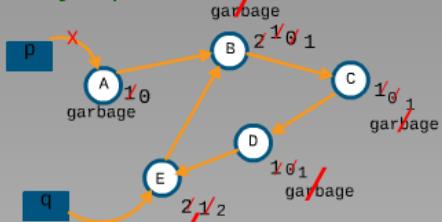
- Simple de implementar
- No es muy costoso temporalmente
- Ejecución dispersa en el tiempo
- No funciona para estructuras cíclicas
- No soluciona el problema de la fragmentación

# Recorrido y Marcado local

## Idea:

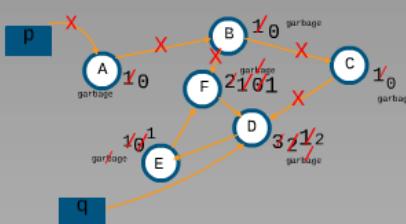
Algoritmo basado en contador de referencias pero soluciona el problema de estructuras cíclicas

### Ejemplo 1:



Solo el nodo A es garbage

### Ejemplo 2:



Los nodos A, B, C son garbage

## Algoritmo:

### A partir del puntero borrado:

- Paso 1: Recorrer los bloques decrementando los contadores y marcando los nodos como posible garbage.
- Paso 2: Se recorre nuevamente buscando nodos con cuenta diferente de cero, se desmarcan y se actualizan los contadores de sus hijos.
- Paso 3: Todos los bloques marcados se mueven a la lista de libres.

## Consideraciones:

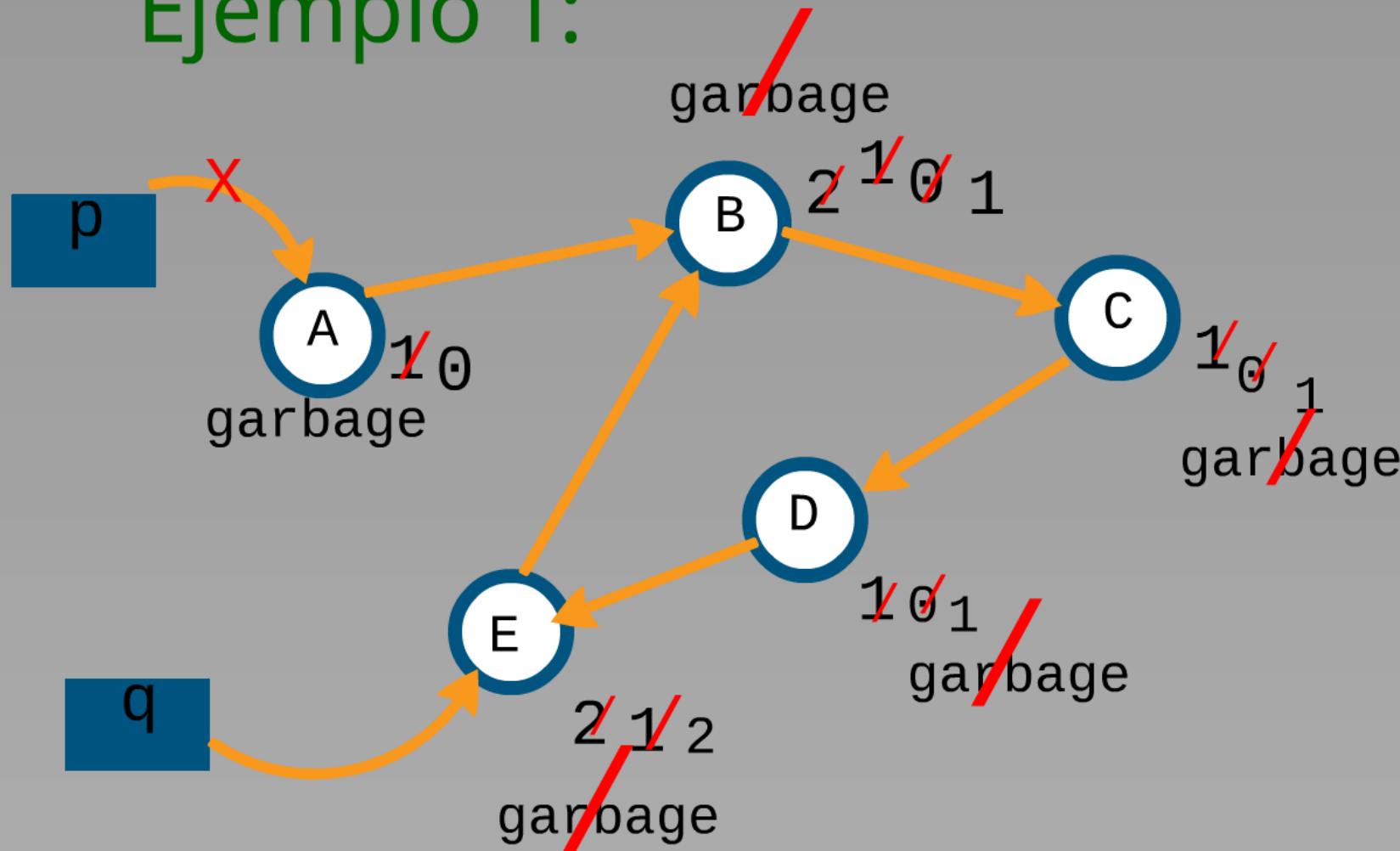
- Soluciona el problema de las estructuras cíclicas
- Complejidad temporal moderada al realizar una búsqueda local
- Tiene los beneficios del contador de referencias y del algoritmo de búsqueda y marcado sin demasiados perjuicios.

# Algoritmo:

## A partir del puntero borrado:

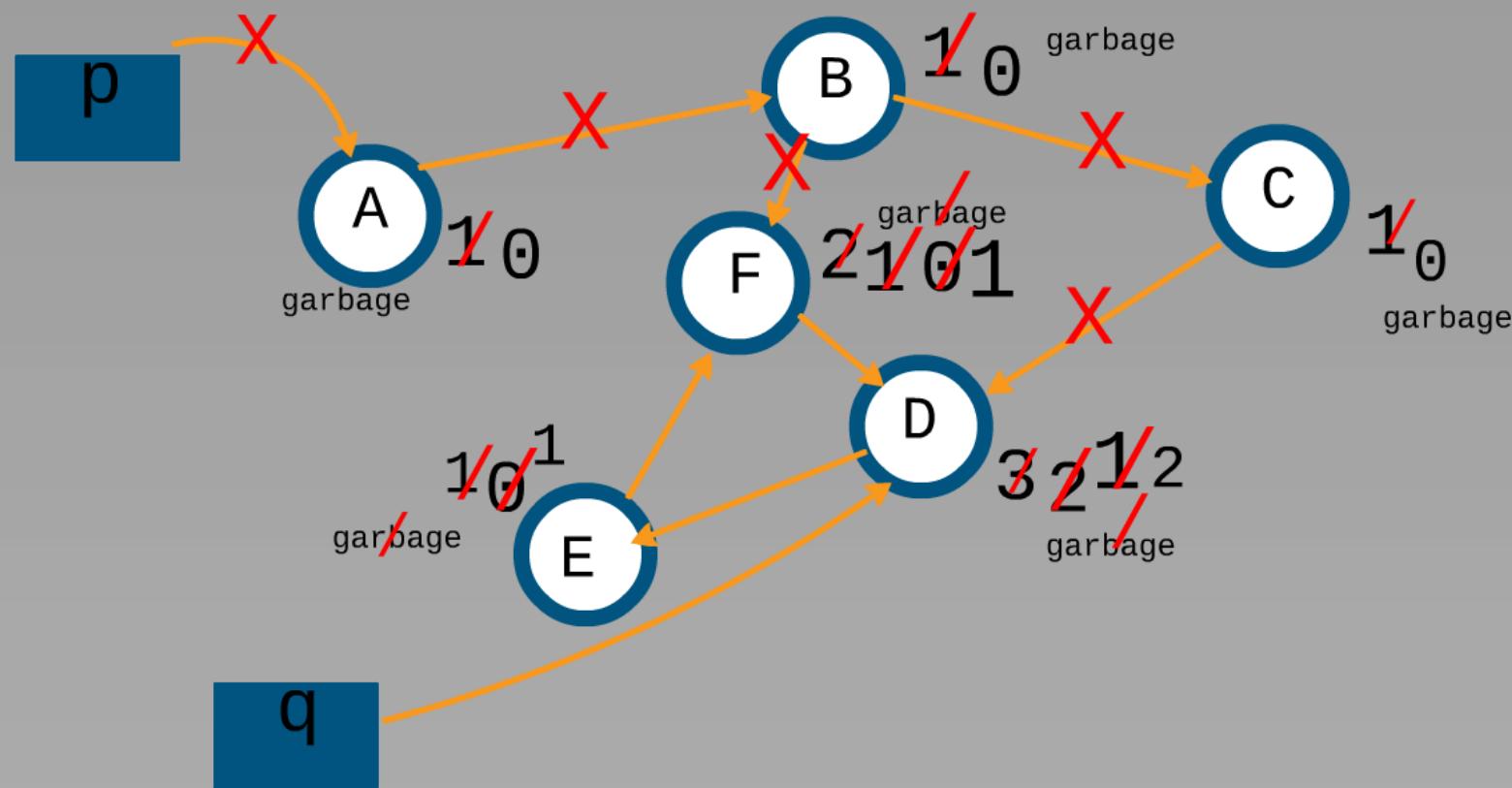
- Paso 1: Recorrer los bloques decrementando los contadores y marcando los nodos como posible garbage.
- Paso 2: Se recorre nuevamente buscando nodos con cuenta diferente de cero, se desmarcan y se actualizan los contadores de sus hijos.
- Paso 3: Todos los bloques marcados se mueven a la lista de libres.

# Ejemplo 1:



Solo el nodo A es garbage

## Ejemplo 2:



Los nodos A, B, C son garbage

# Consideraciones:

- Soluciona el problema de las estructuras cíclicas
- Complejidad temporal moderada al realizar una búsqueda local
- Tiene los beneficios del contador de referencias y del algoritmo de búsqueda y marcado sin demasiados perjuicios.

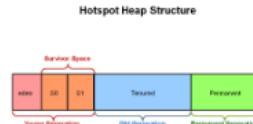
## Otros métodos:

- **Copiado:** Se mantienen dos espacios de heap en espejo y se realiza GC sobre uno de ellos mientras se usa el otro
- **Generacionales:** Se dividen los objetos según su tiempo de vida
- **Incrementales:** Se realiza el GC por pasos, en cada uno se retoma el estado anterior.
- **Concurrentes:** Se ejecuta el GC en otro hilo de ejecución diferente al del programa
- **Distribuidos:** Sobre diferentes espacios de memoria se realizan diferentes tipos de GC

# Garbage collection en diferentes lenguajes

## Java

- Los objetos son alojados en el espacio "Edén"
- Cuando se llena el espacio "Edén", se hace un garbage collection de marcado y borrado y los objetos sobrevivientes se mueven al "Survivor Space"
- Cuando se llena el espacio "Young Generation" se realiza un garbage collector, y cada vez que un objeto sobrevive, se incrementa un contador.
- Cuando el contador llega a un cierto threshold, se mueven al espacio "Old Generation"
- Es posible realizar un garbage collection en el espacio "Old Generation" y seleccionar el algoritmo que se deseé
- El espacio "Permanent Generation" se utiliza para almacenar clases del sistema. Es posible realizar un Full Garbage Collection sobre este espacio



### Referencias:

<http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html>  
<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

## C#

### Sistema generacional

Si el garbage collector detecta que muchos objetos sobreviven en una generación, modifica el threshold de supervivencia



Es posible utilizar garbage collector automático o manual

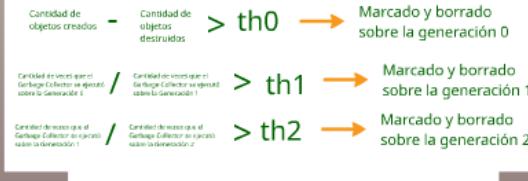
### Referencias:

[https://msdn.microsoft.com/en-us/library/ee787088\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx)

## Phyton

- Generacional: 3 generaciones
- Contador de referencias automático
- Garbage collection de marcado y borrado manual o programado

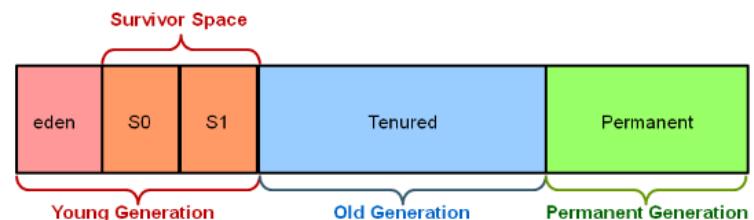
`set_threshold(th0, th1, th2)`



# Java

- Los objetos son alojados en el espacio "Edén"
- Cuando se llena el espacio "Edén", se hace un garbage collection de marcado y borrado y los objetos sobrevivientes se mueven al "Survivor Space"
- Cuando se llena el espacio "Young Generation" se realiza un garbage collector, y cada vez que un objeto sobrevive, se incrementa un contador.
- Cuando el contador llega a un cierto threshold, se mueven al espacio "Old Generation"
- Es posible realizar un garbage collection en el espacio "Old Generation" y seleccionar el algoritmo que se deseé
- El espacio "Permanent Generation" se utiliza para almacenar clases del sistema. Es posible realizar un Full Garbage Collection sobre este espacio

Hotspot Heap Structure



## Referencias:

<http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html>

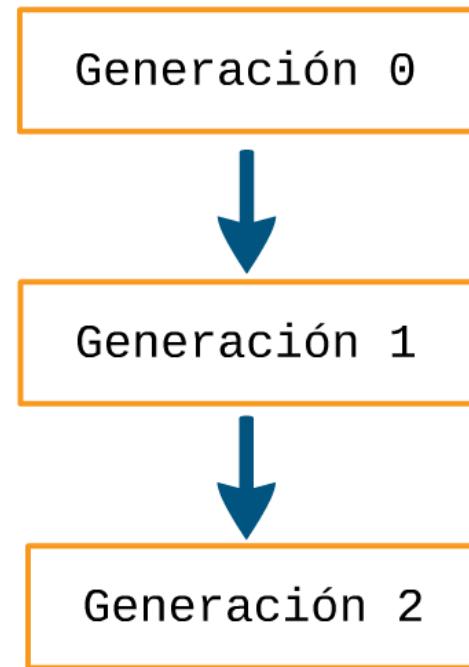
<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

# C#

## Sistema generacional

Si el garbage collector detecta que muchos objetos sobreviven en una generación, modifica el threshold de supervivencia

Es posible utilizar garbage collector automático o manual



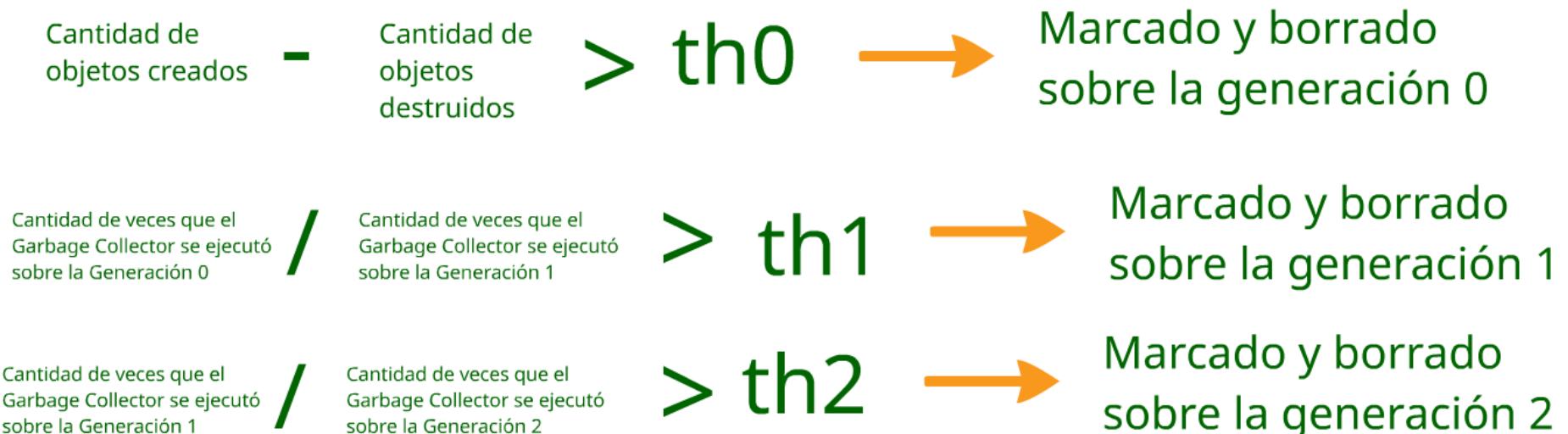
## Referencias:

[https://msdn.microsoft.com/en-us/library/ee787088\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx)

# Phyton

- Generacional: 3 generaciones
- Contador de referencias automático
- Garbage collection de marcado y borrado manual o programado

`set_threshold(th0, th1, th2)`



# Interacciones con el Sistema Operativo

- Manejo de memoria
- Concurrency

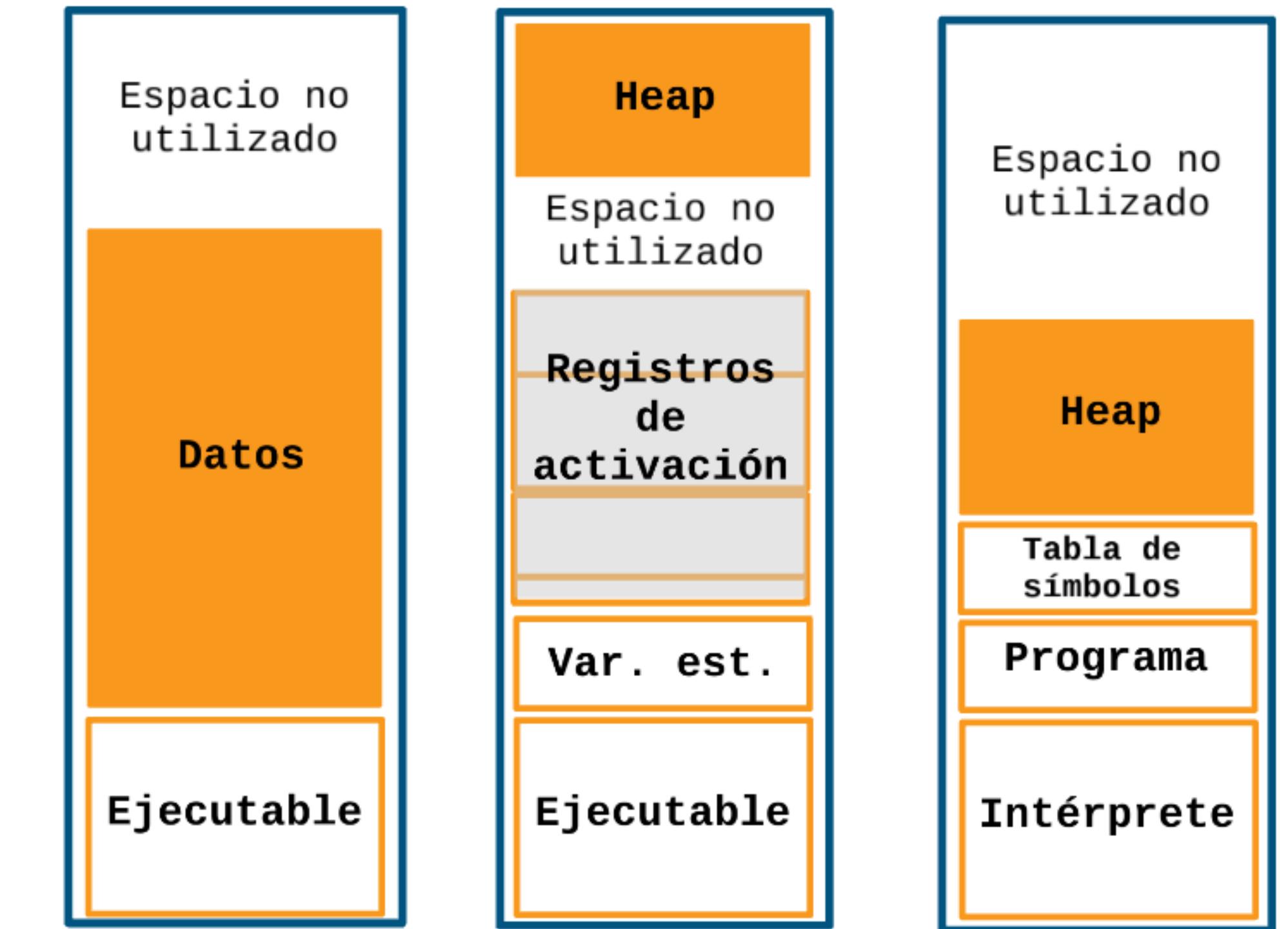
Lenguajes de Programación I  
Facultad de Cs. Exactas  
UNICEN  
José M. Massa

# Interacciones con el S.O. para la administración de la memoria

¿Qué tenemos hasta ahora?

Sistemas Operativos con Particiones fijas (única partición)

El programa entra o no entra en la partición



Estático

Tipo-Algol

Dinámico

- Si la pila de R.A. con el heap, chocan se hace Garbage Collection
- Puede ser que un programa no se pueda ejecutar en la partición

- El heap puede crecer hasta ocupar toda la partición
- La tabla de símbolos también crece

# Interacciones con el S.O. para la administración de la memoria

Única partición



Varias particiones

S.O. muy primitivos

El S.O. tiene varias particiones fijas y le asigna una de ellas al lenguaje para ejecutar el programa

¿Qué ocurre con varias particiones?

El S.O. puede asignarle aquella que tenga el tamaño más adecuado, pero no soluciona el problema del choque de la pila con el heap

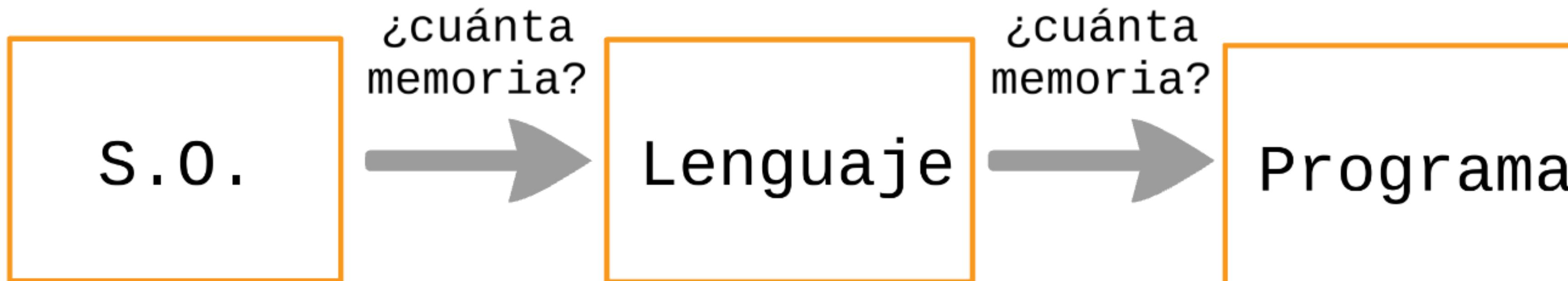
# Asignación de memoria en S.O. con particiones fijas

¿Qué es lo que ocurre con las particiones fijas?

- Se llevan mal con los lenguajes

¿Cómo hace el S.O. para darle al Lenguaje una partición adecuada? [ ]

- El S.O. le pregunta al lenguaje cuánta memoria necesita  
El lenguaje no sabe cuánta memoria va a necesitar el programa
- El lenguaje le pregunta al programa cuánta memoria necesita  
El programa no sabe cuánta memoria va a necesitar realmente
- El programador debe especificar una cantidad de memoria que espera que use el programa



¿Cómo hace el programador para especificar cuánta memoria necesita su programa?

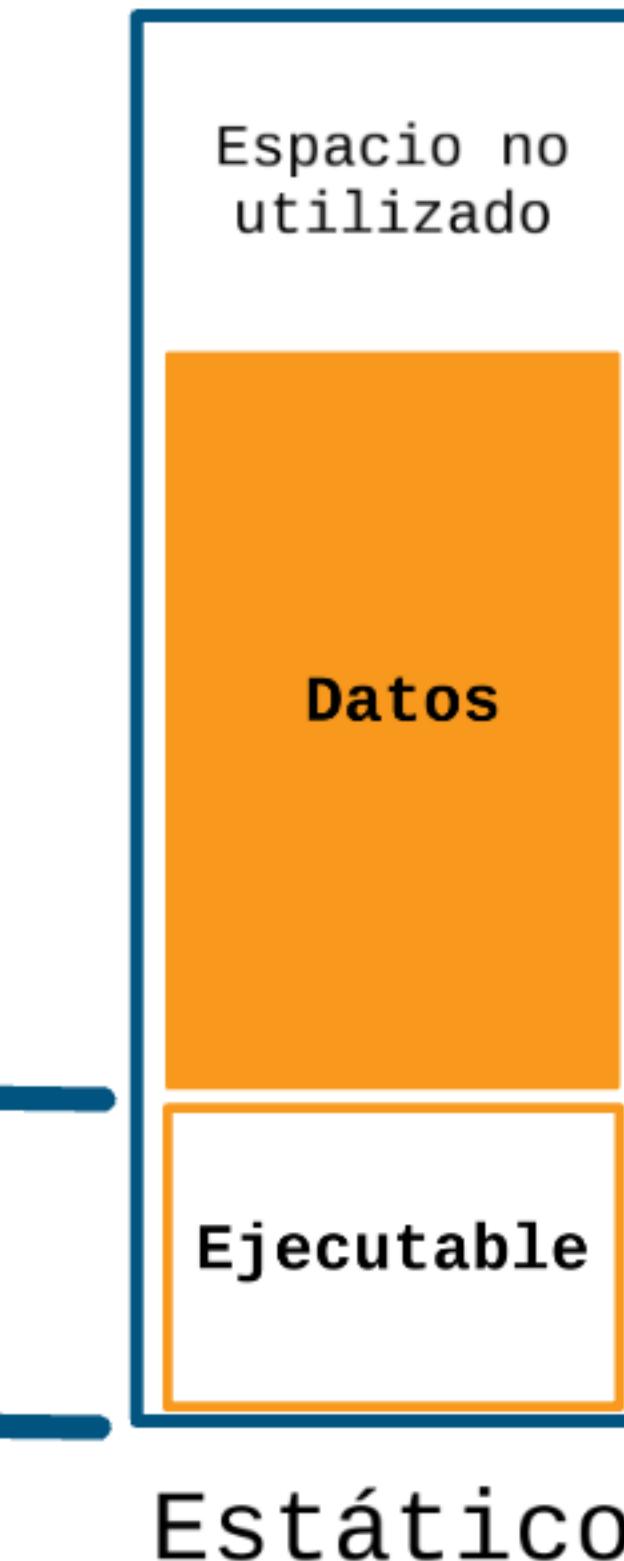
# ¿Qué es lo que ve el S.O.?



Lenguaje  
Tipo-Algol

No puede calcular cuánto tamaño va a usar el programa

Al comienzo sólo ve el ejecutable



Estático



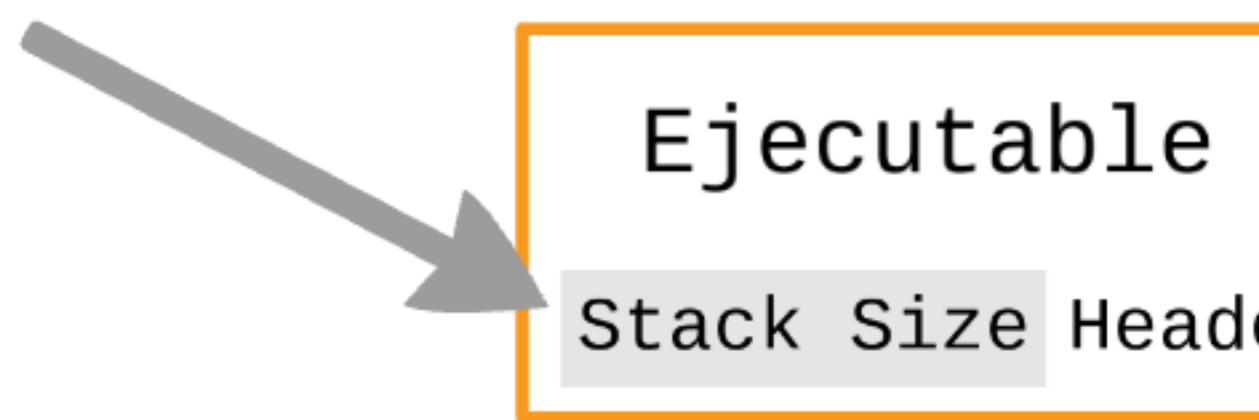
Dinámico

En los lenguajes dinámicos ve el intérprete y el programa

## Asignación de memoria en S.O. con particiones fijas

El lenguaje tiene un valor por defecto para la cantidad de memoria que va a necesitar el programa

El lenguaje coloca ese dato en el ejecutable



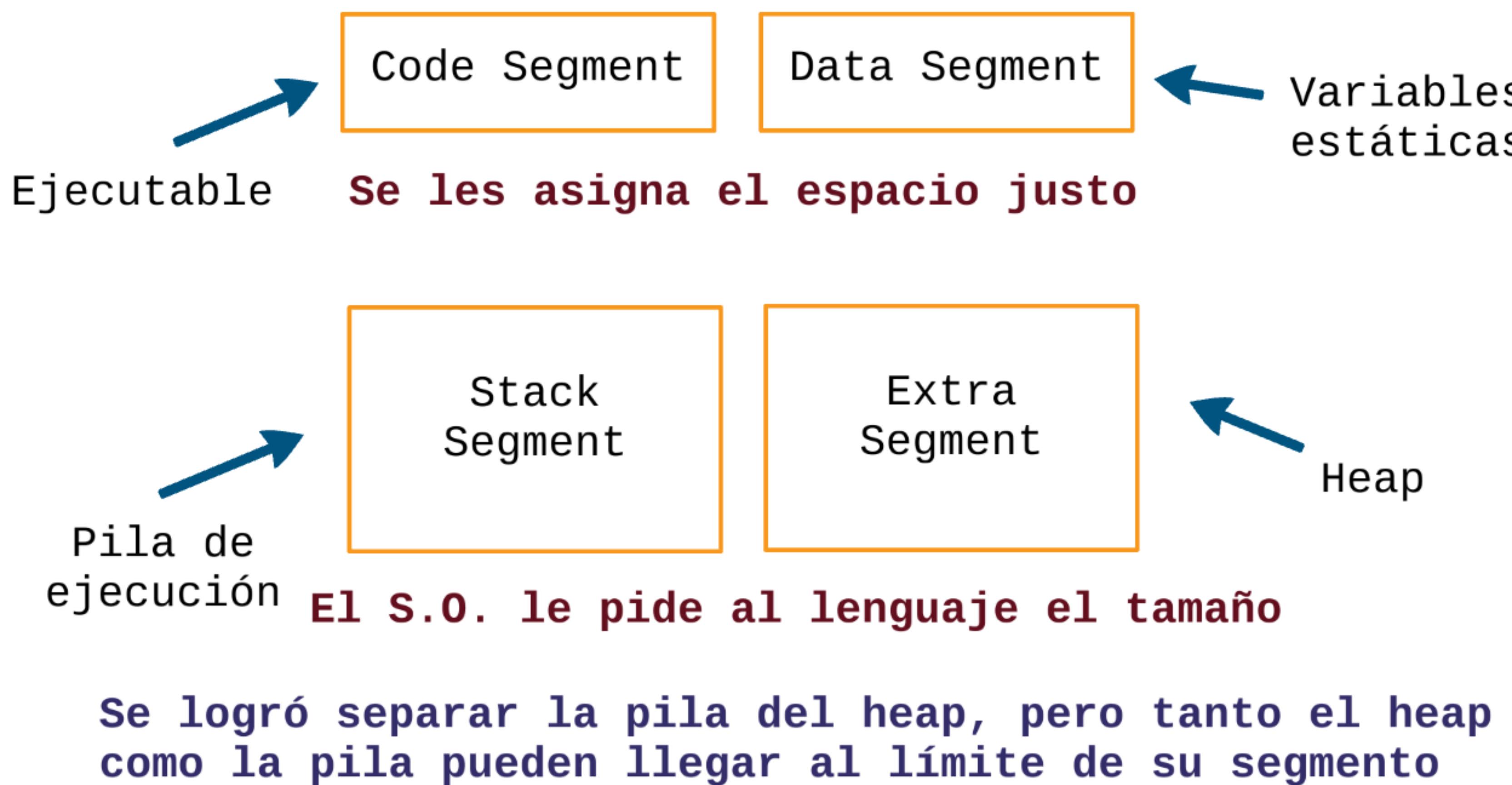
**¿Cómo hace el programador para cambiar ese dato?**

- Variables de entorno  
(ej. en SUN/SOLARIS setenv STACKSIZE 32768)
- Opciones de compilación  
(ej. gcc -Wl, --stack=xxxxx)
- Pragmas  
(#pragma dataseg(...))

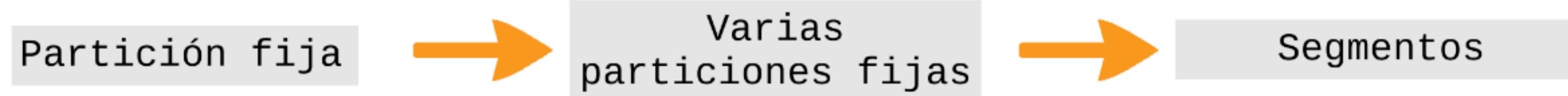
## Asignación de memoria en S.O. con segmentos

El S.O. permite la creación de diferentes espacios  
de memoria de tamaño personalizado

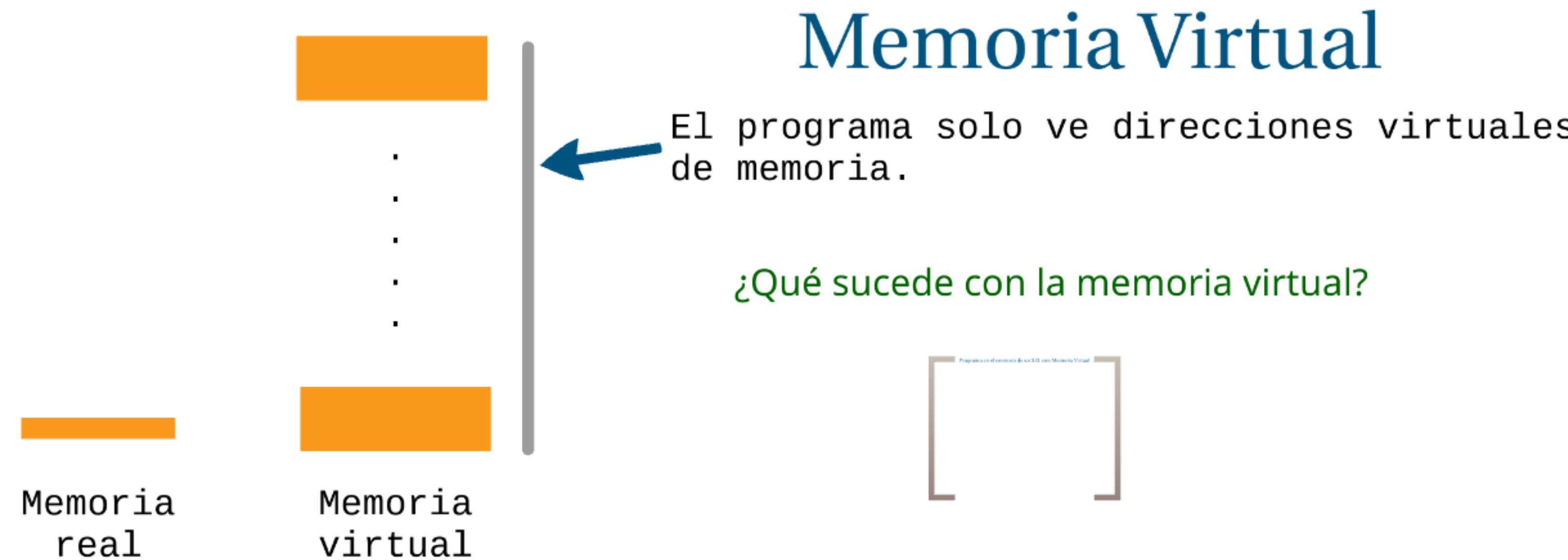
Los S.O. basados en segmentos están muy relacionados con  
la arquitectura de hardware (En Intel: CS, DS, SS, ES)



# Asignación de memoria en S.O. con memoria virtual / paginada



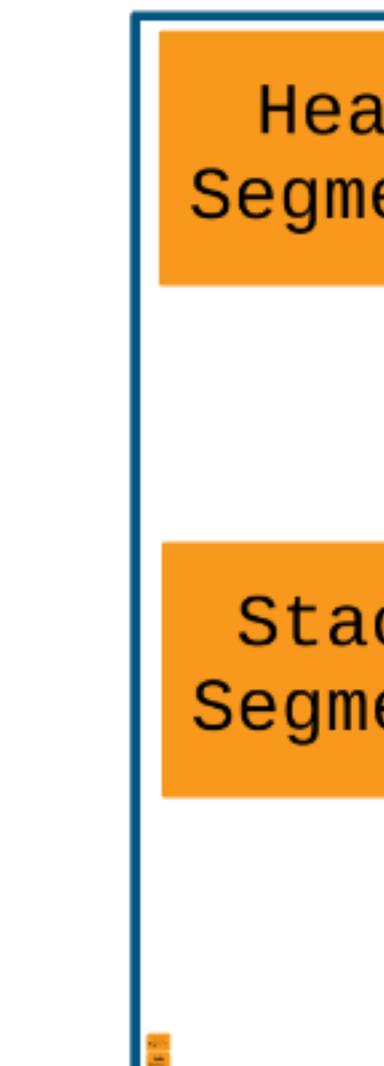
- En ninguno de estos casos se evitan los conflictos de memoria
- El Garbage Collector sigue siendo necesario



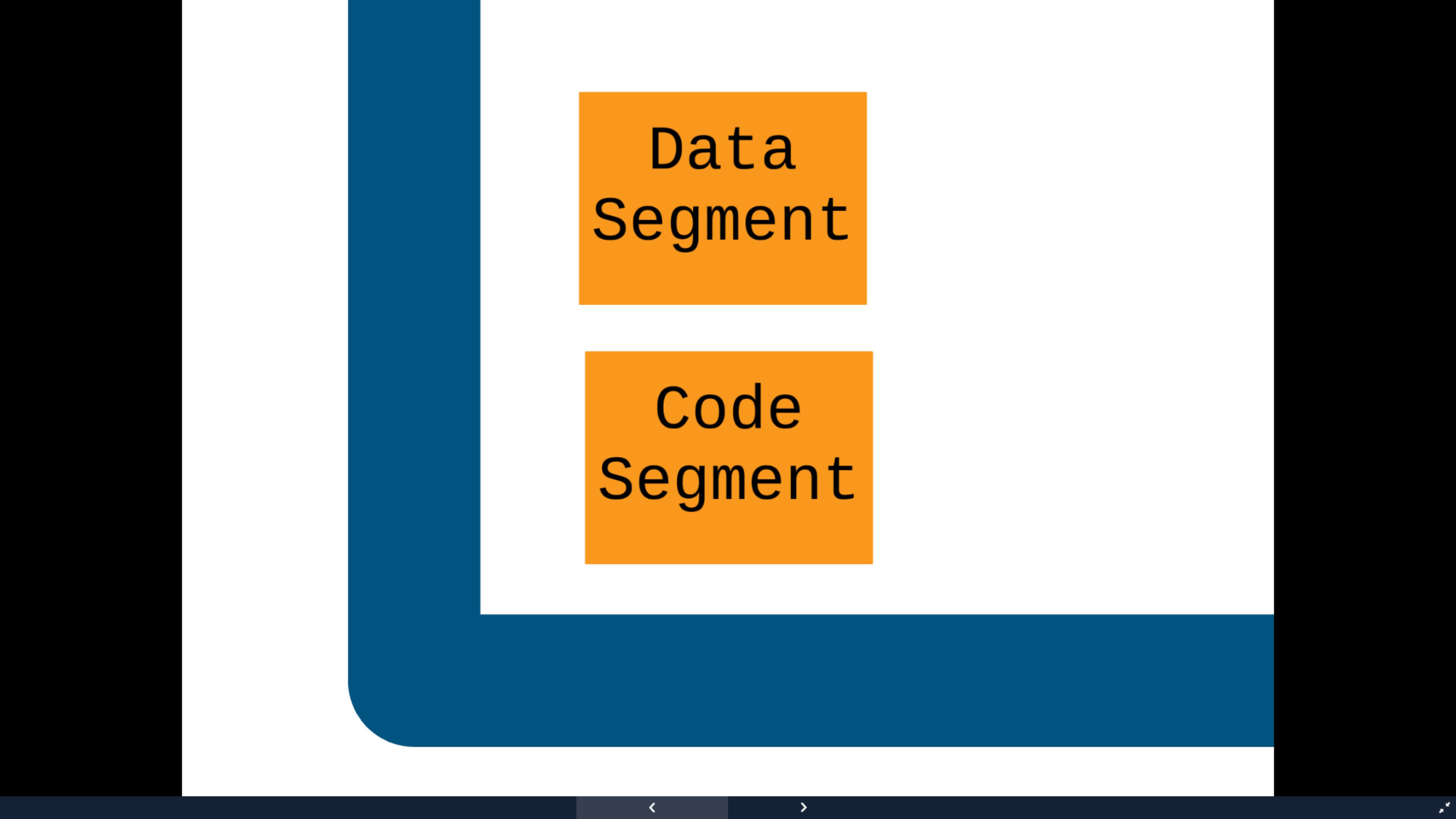
# Programa en el contexto de un S.O. con Memoria Virtual



- La posibilidad de choque real del heap con la pila de R.A. se reduce al mínimo
- ¿Qué pasa con un programa funcionando las 24 hs. de los 7 días de la semana?
  - A fuerza de generar garbage en algún momento van a chocar
  - Se puede llenar el segmento del heap o el de la pila



S.O. con segmentos y memoria virtual



The diagram illustrates the memory structure of a process. It features a vertical blue bar on the left representing the stack, which has a white top section and a blue bottom section. To the right of the stack is a white area containing two orange rectangular boxes. The top box is labeled "Data Segment" and the bottom box is labeled "Code Segment".

Data  
Segment

Code  
Segment

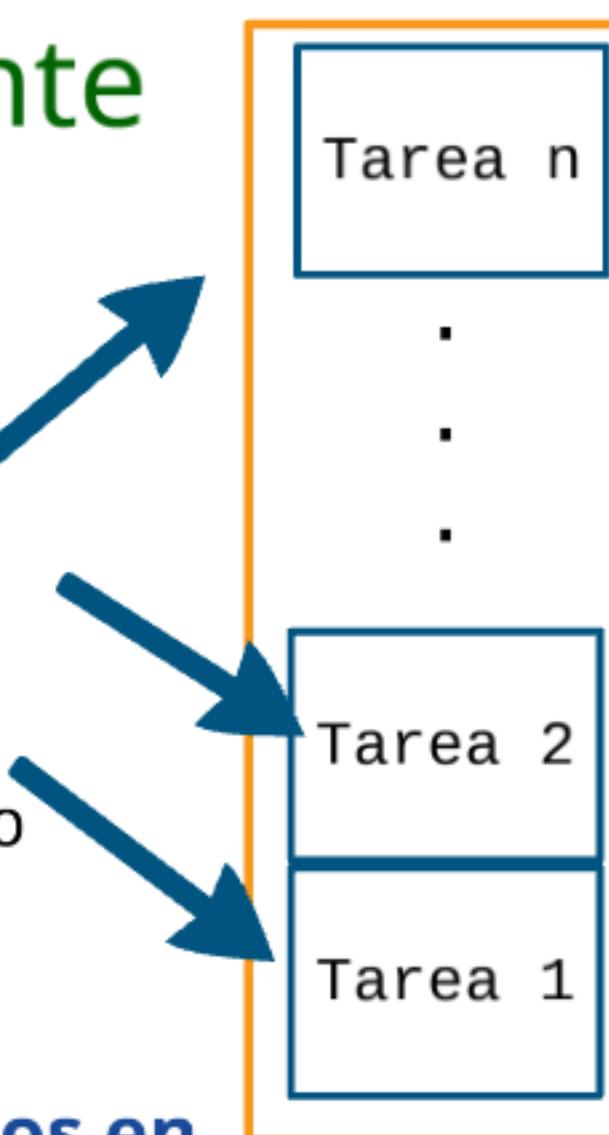
# Concurrencia de Lenguajes/Sistemas Operativos

Concurrencia: Simultaneidad en la ejecución de tareas (procesos, hilos)

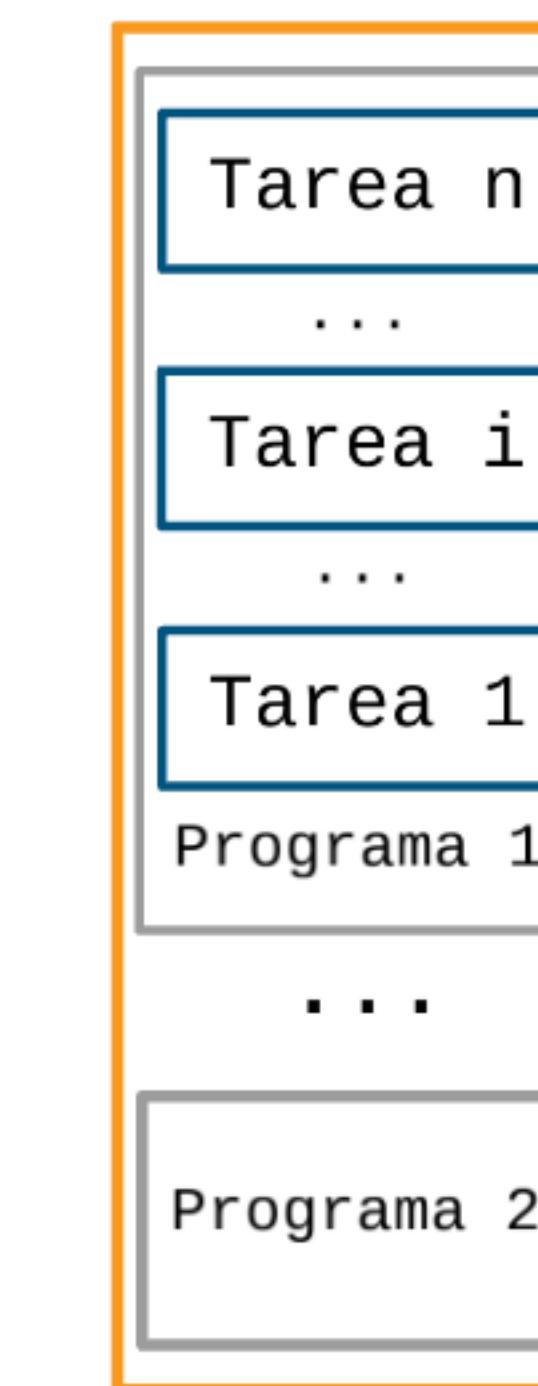
Se pueden ejecutar en un solo procesador, en varios núcleos de un procesador o en diferentes computadoras

## S.O. Concurrente

- Pueden estar escritas en diferentes lenguajes
- Diferentes espacios de direccionamiento propio
- **Ejecuta varios procesos en forma simultánea usando context switching**



Los primeros lenguajes concurrentes no funcionaban sobre S.O. Concurrentes. Ej. Pascal Concurrente sobre DOS (Monotarea)



- Están escritos en un solo lenguaje.
- La concurrencia es entre partes del programa dentro del mismo espacio de direccionamiento

**¿Qué sucede dentro del programa?**

**¿Cómo se administra la concurrencia?**

## Lenguaje Concurrente

# Lenguajes Concurrentes

- Pascal Concurrente, ADA  
GO, C++ (2020), Ruby  
Julia, etc.



No necesitan  
S.O. Concurrente

¿Dónde está la concurrencia?

Entre las diferentes funciones o procedimientos

¿Quién administra la concurrencia?

El Run Time System

¿Qué pasa con los Registros de Activación de las funciones concurrentes?

El RTS administra su funcionamiento

Heap

Otros Registros de activación

R.A. Función concurrente 2

R.A. Función concurrente 1

Código función concurrente 1

Código función concurrente 2

Run Time System

Ejecutable

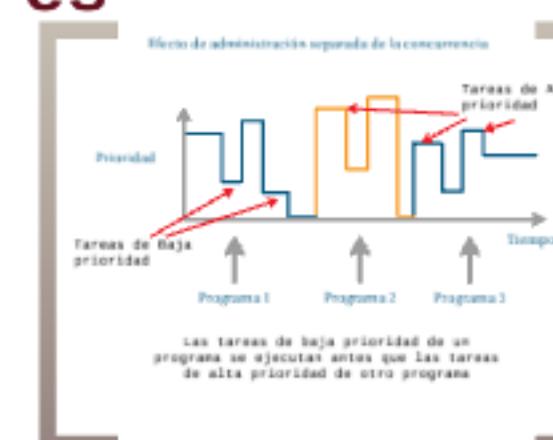
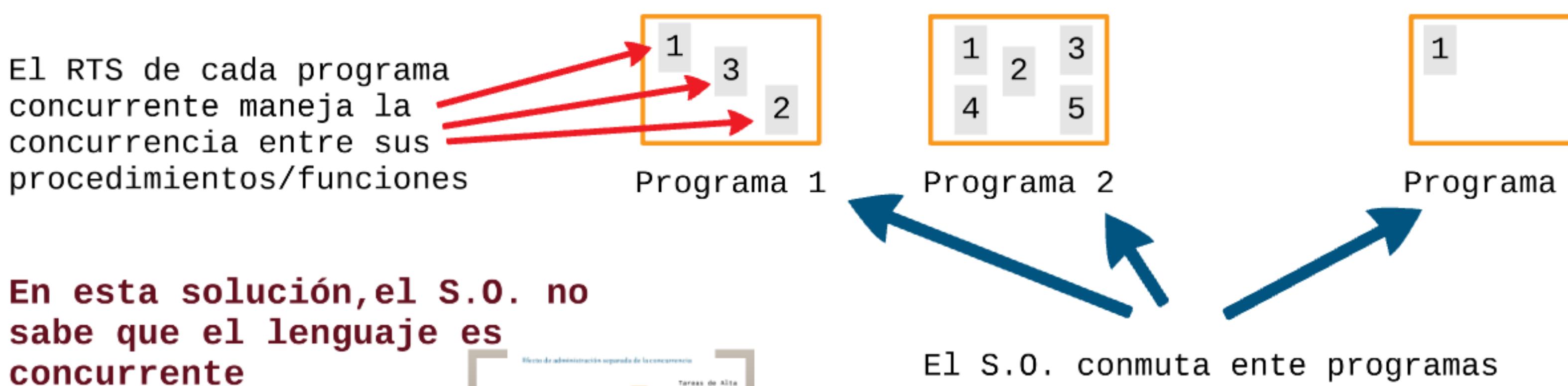
# Combinación Lenguajes/Sistemas Operativos concurrentes

Solo S.O.  
Concurrente → Cola de CPU administrada por S.O.

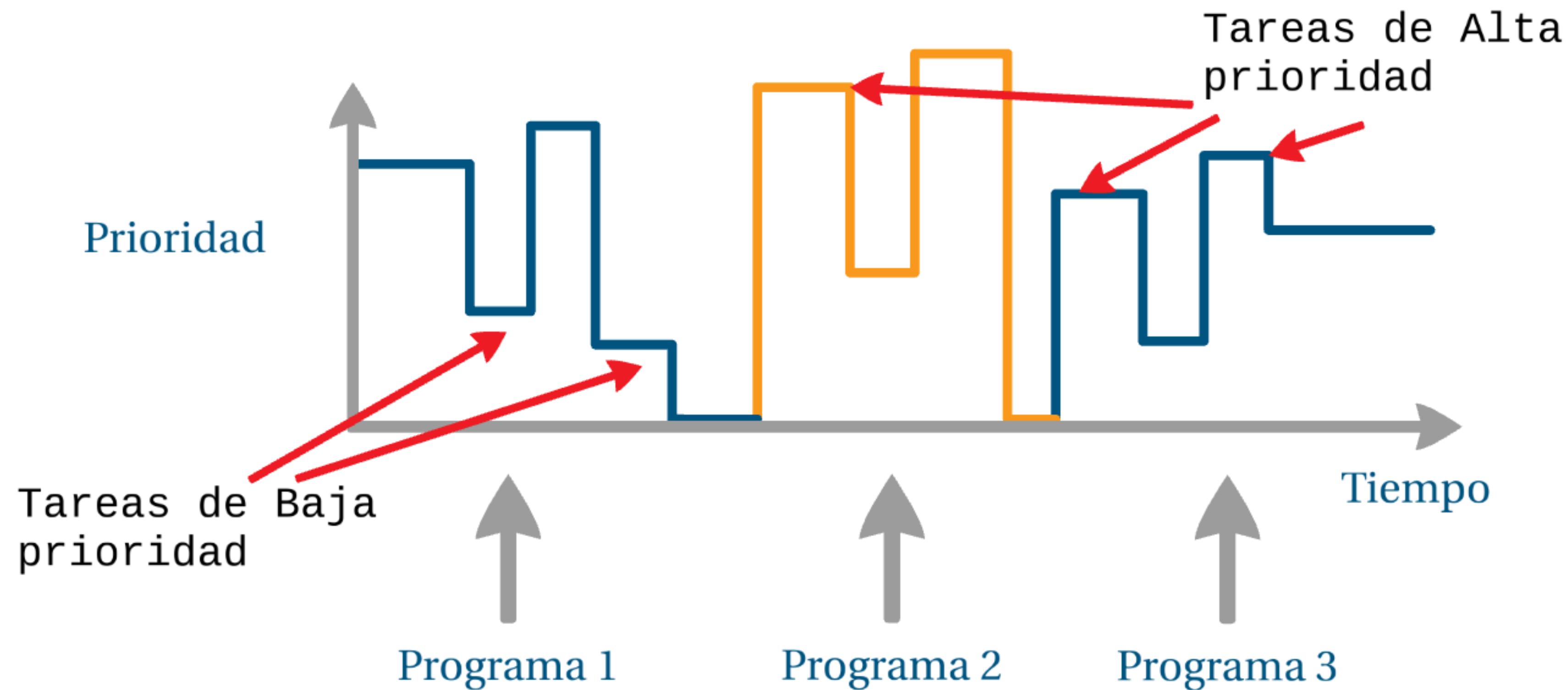
Solo Lenguaje  
Concurrente → Cola de CPU administrada por RTS

## ¿Qué ocurre si la concurrencia es combinada?

Parte de la concurrencia la maneja el  
S.O. y parte la maneja el Lenguaje



## Efecto de administración separada de la concurrencia

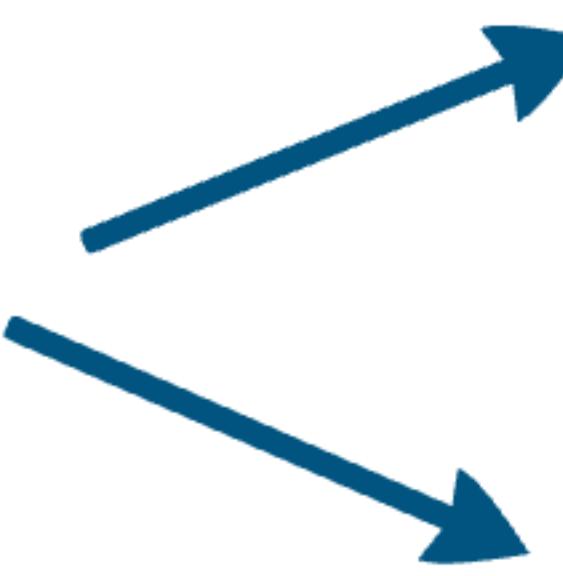


Las tareas de baja prioridad de un programa se ejecutan antes que las tareas de alta prioridad de otro programa

# Concurrencia planificada en conjunto entre S.O. y Lenguaje

¿Qué ocurre si el S.O. "sabe" que el lenguaje es concurrente?

Aspectos de la concurrencia



Administración de prioridades  
A quién le toca el CPU



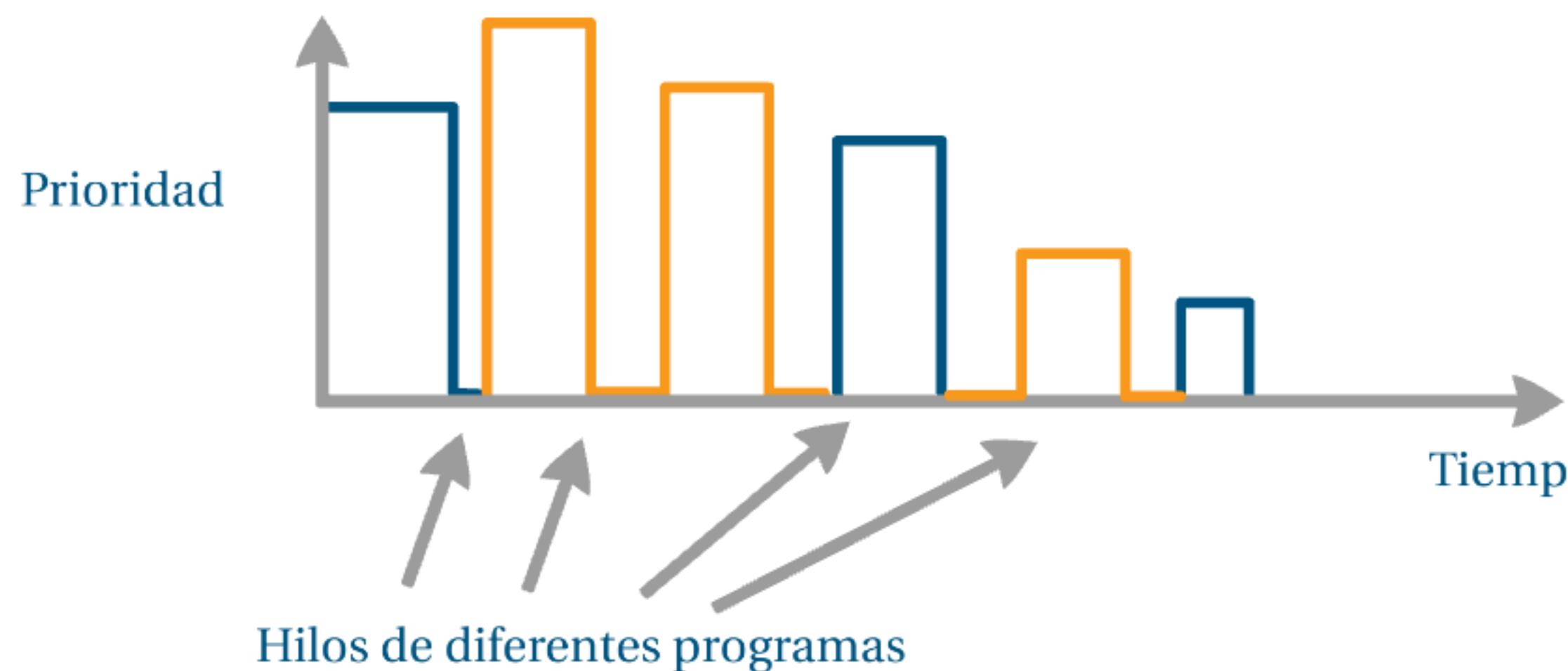
Administración de memoria

Dónde se ubican los datos de cada parte del programa



# El S.O. administra los hilos de los programas

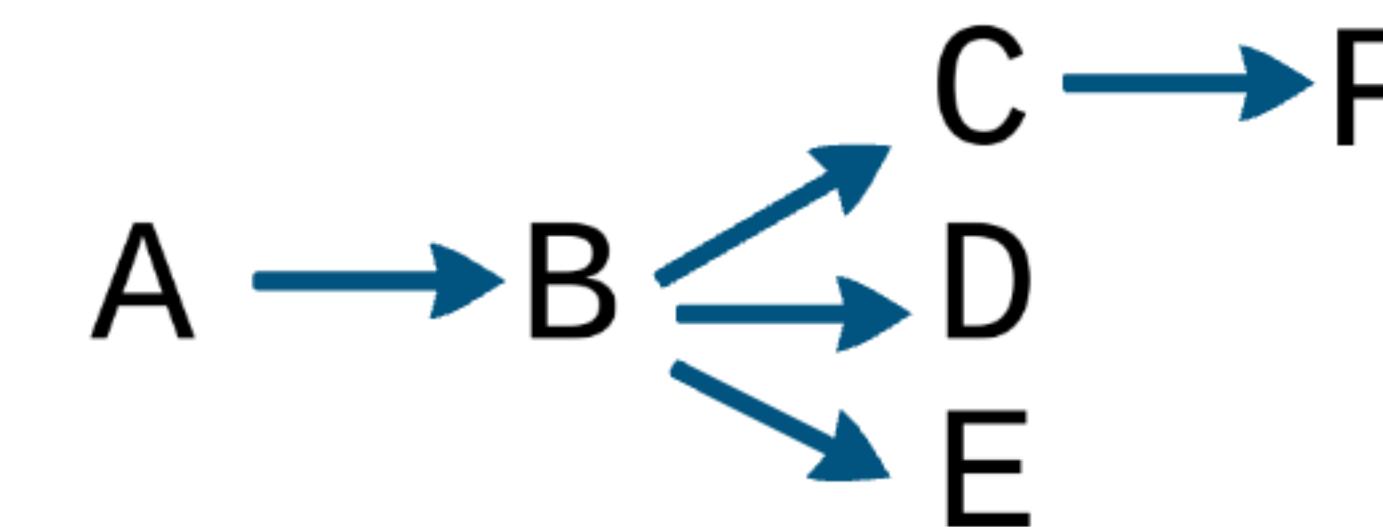
Hilo (S.O.): es la tarea más pequeña que puede administrar un S.O.



El S.O. intenta ejecutar los hilos de mayor prioridad aunque pertenezcan a diferentes programas

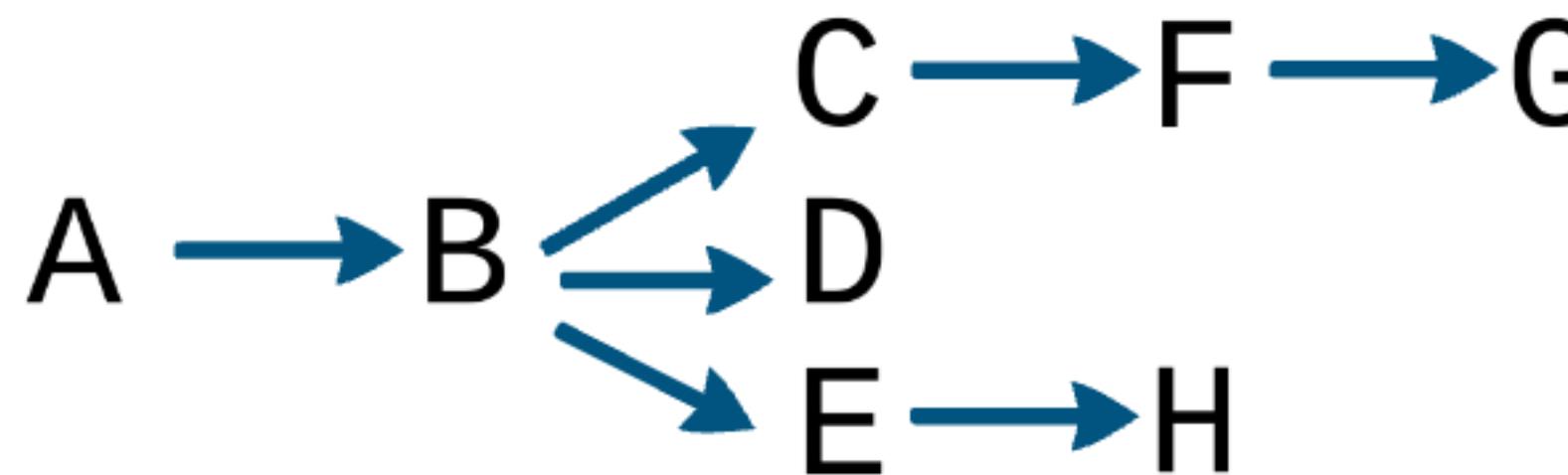
¿Qué ocurre con las pilas de ejecución?

Un programa ahora puede tener la siguiente secuencia de invocación:



# Administración de memoria en Lenguajes/S.O. Concurrentes

Teniendo las invocaciones:

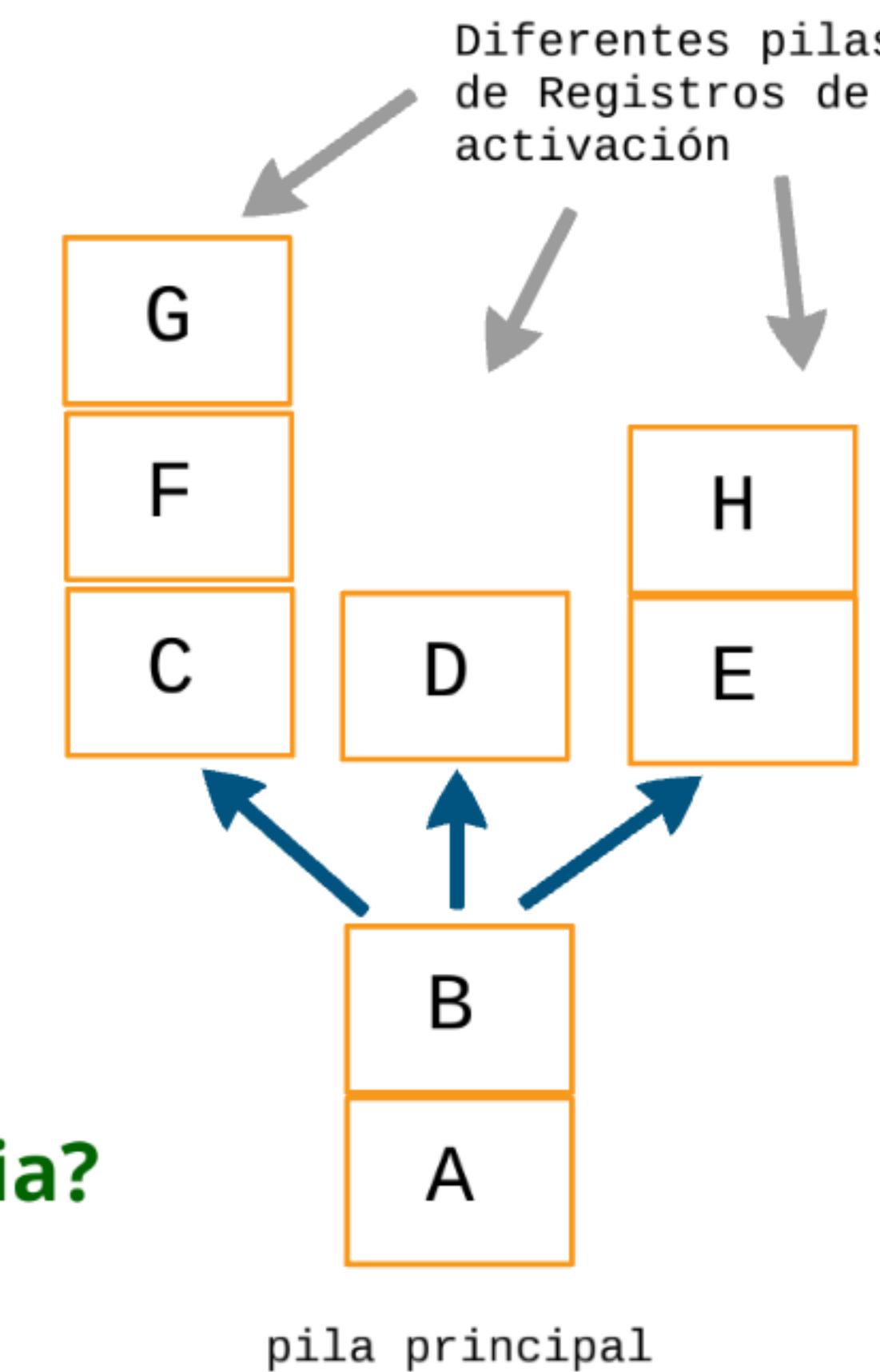


- Se van a crear diferentes pilas para cada hilo de cada programa concurrente

**¿Cómo se distribuyen las pilas en la memoria?**

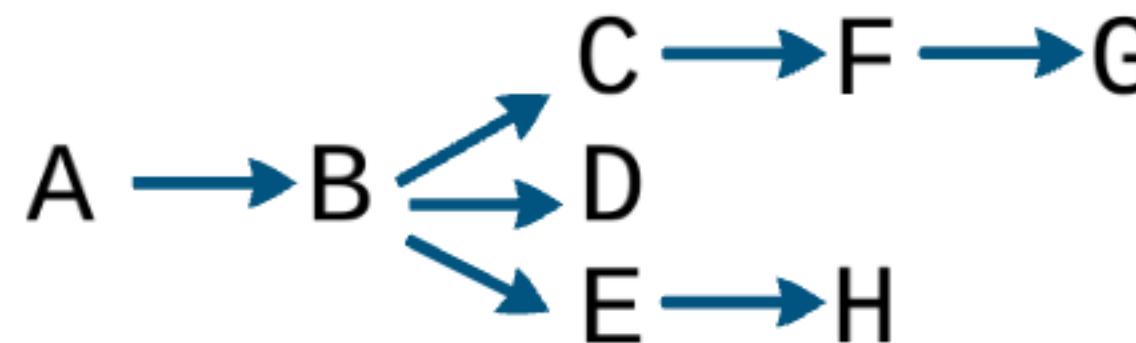
**¿Qué sucede con el heap?**

**¿Qué sucede con las variables estáticas?**



# Pila de ejecución en un contexto de concurrencia planificada en conjunto entre S.O. y Lenguaje

## ¿Cómo se ubican las pilas?

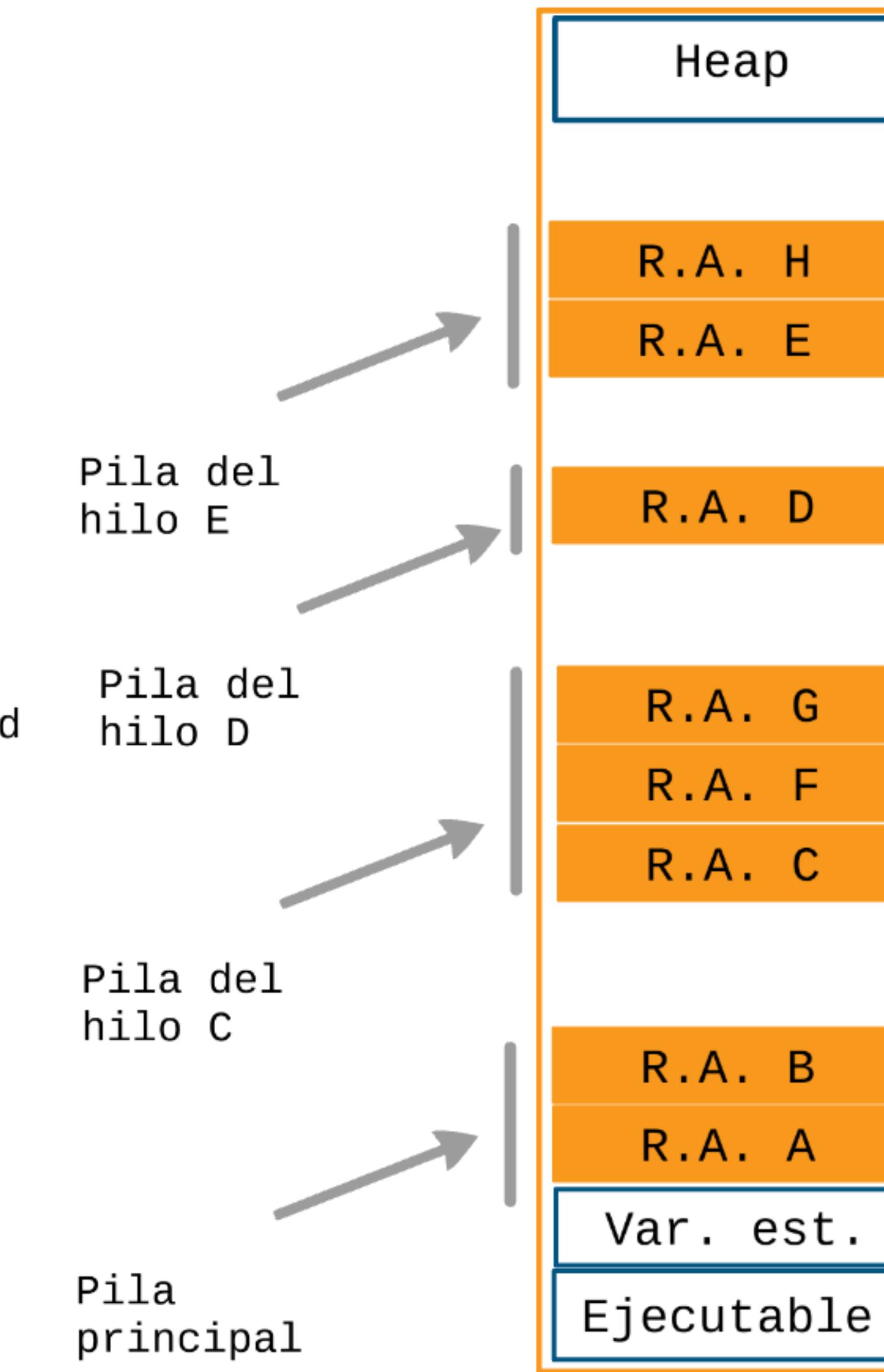


El tamaño y la ubicación de la pila de cada hilo puede ser definida por:

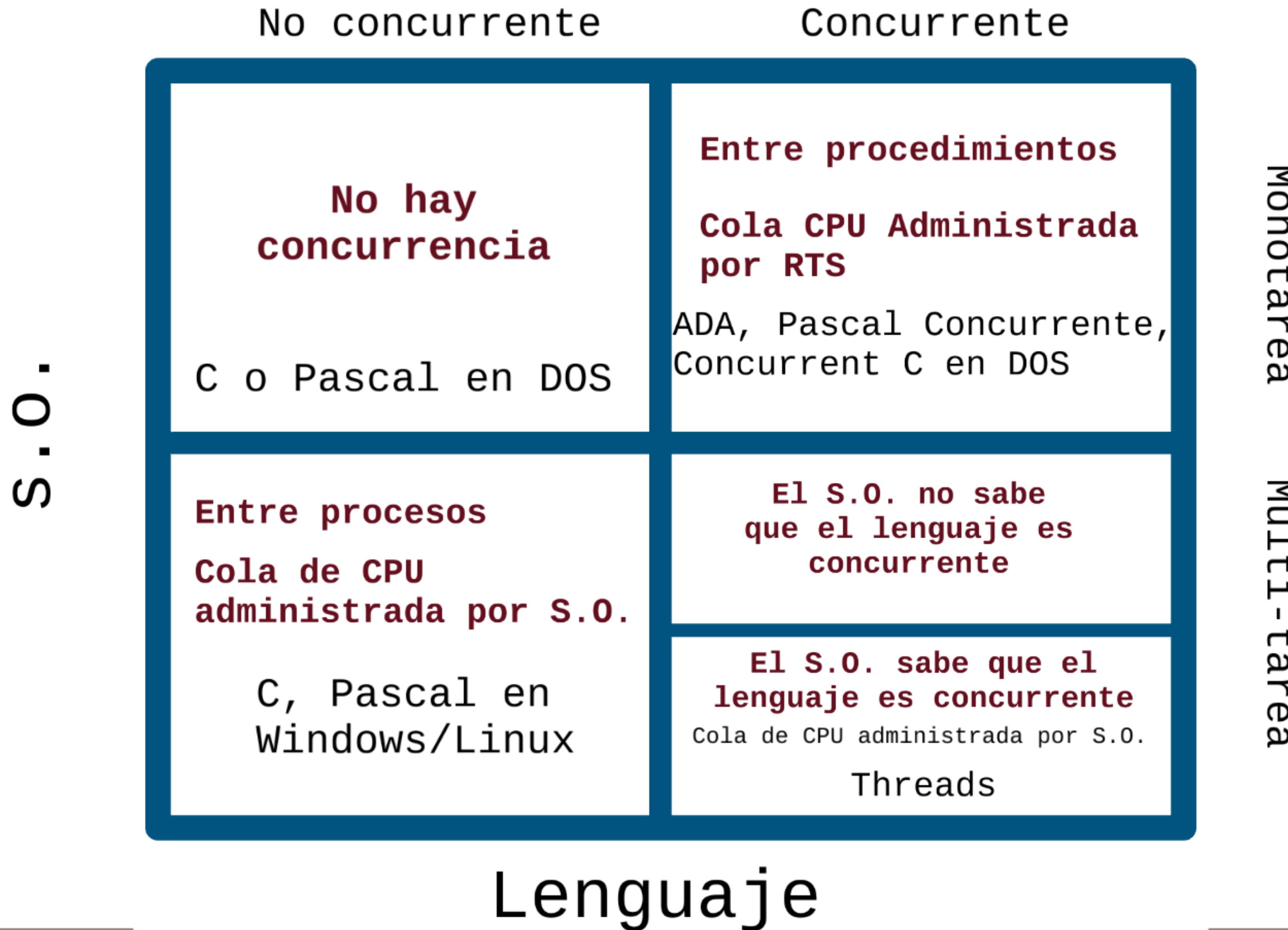
- El S.O. (sólo controla la cantidad de hilos)
- El lenguaje (Ubica las pilas en memoria)
- El programador

¿Cuáles son los problemas potenciales que aparecieron?

Posibilidad de choque entre las pilas de los hilos



# Resumen Lenguajes/Sistemas Operativos concurrentes



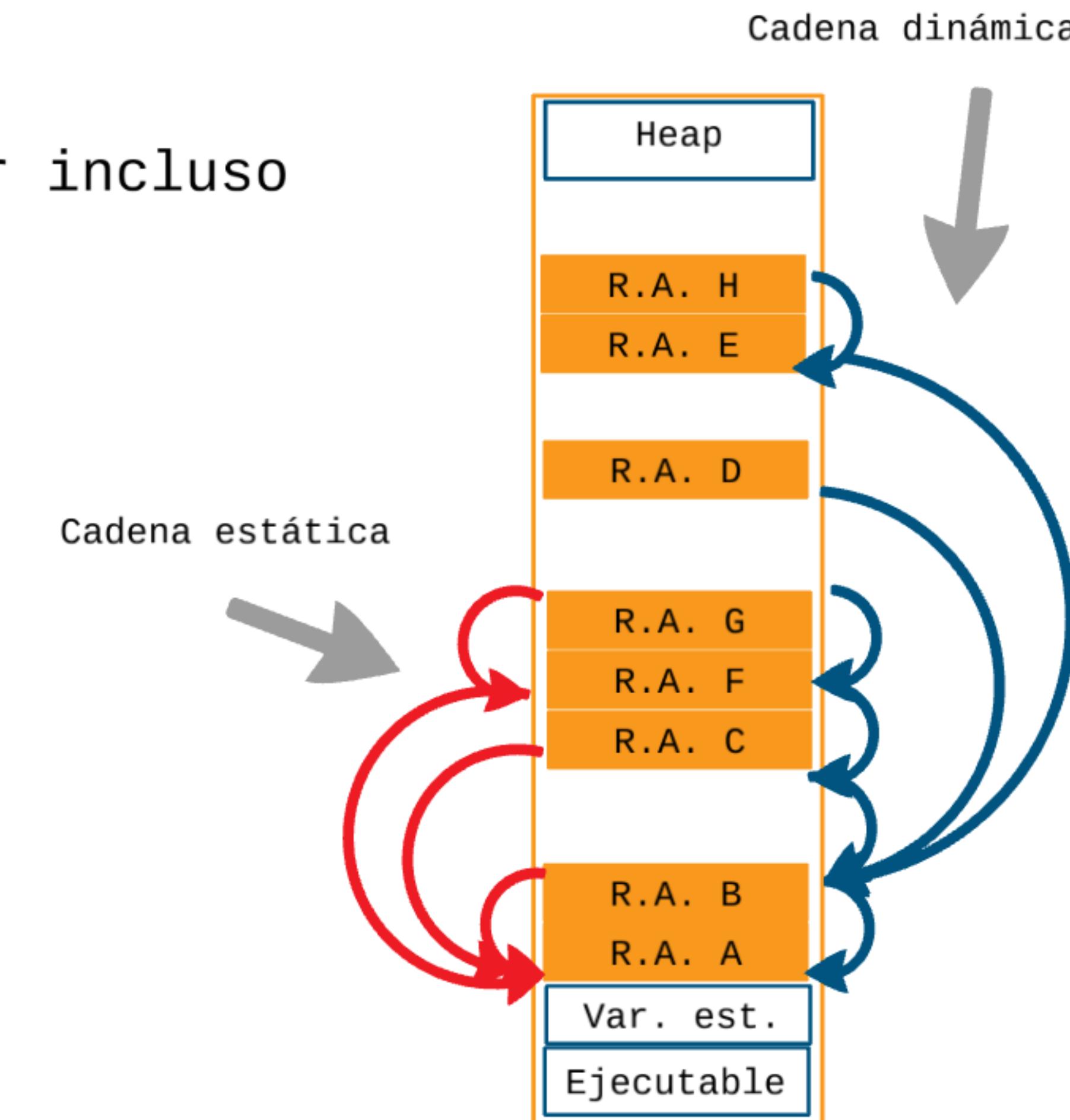
# Pila de ejecución en un contexto de concurrencia planificada en conjunto entre S.O. y Lenguaje

¿Qué sucede con los hilos de ejecución?

- Una pila para cada hilo
- Las pilas se pueden chocar incluso teniendo memoria virtual

¿Qué sucede con las cadenas estáticas y dinámicas?

Siguen siendo válidas



# Pila de ejecución en un contexto de concurrencia planificada en conjunto entre S.O. y Lenguaje

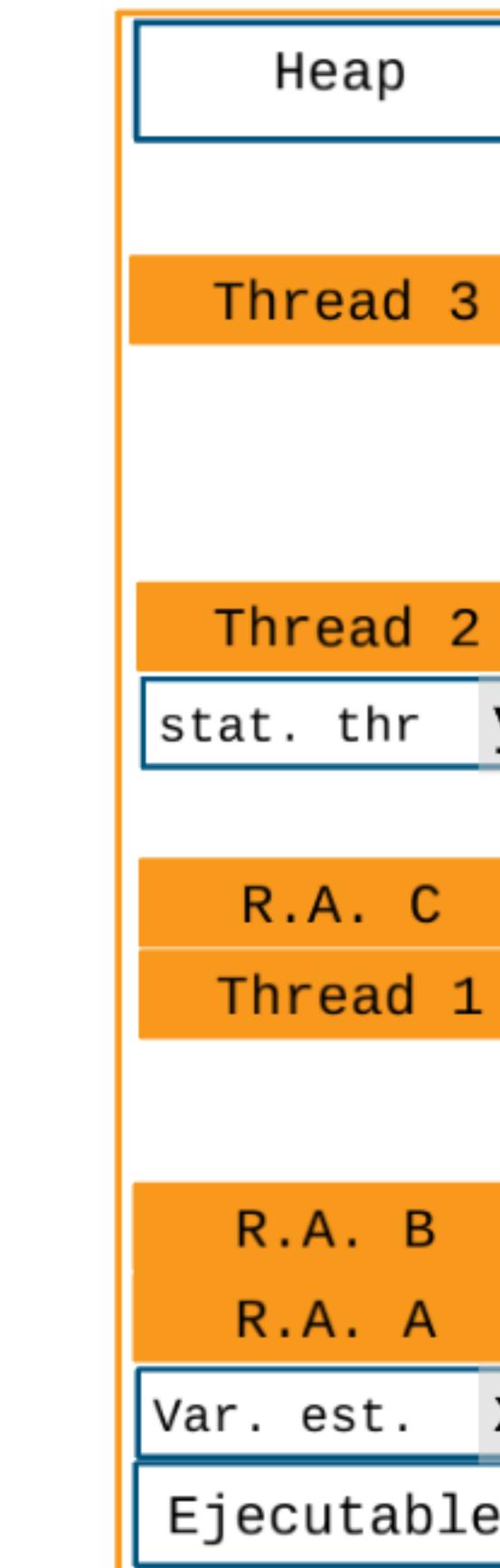
Aparece un concepto nuevo de almacenamiento y alcance

Además de las variables estáticas ya conocidas, aparecen las variables estáticas de cada thread globales al mismo.

`static int x;` Es global a todo el programa

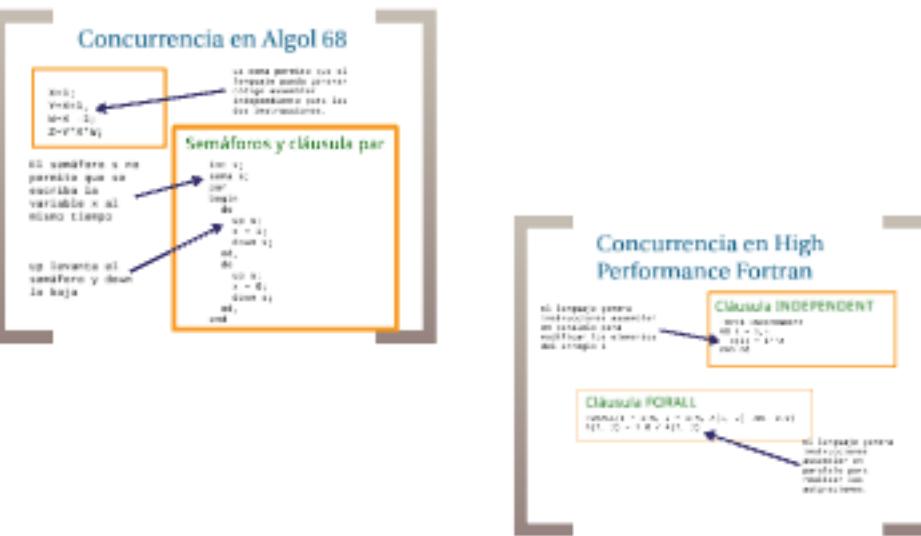
`static __thread int y;` Es global a cada thread, pero estática

Los threads además pueden tener sus variables locales como cualquier unidad



# Concurrencia y Paralelismo en Lenguajes de Programación

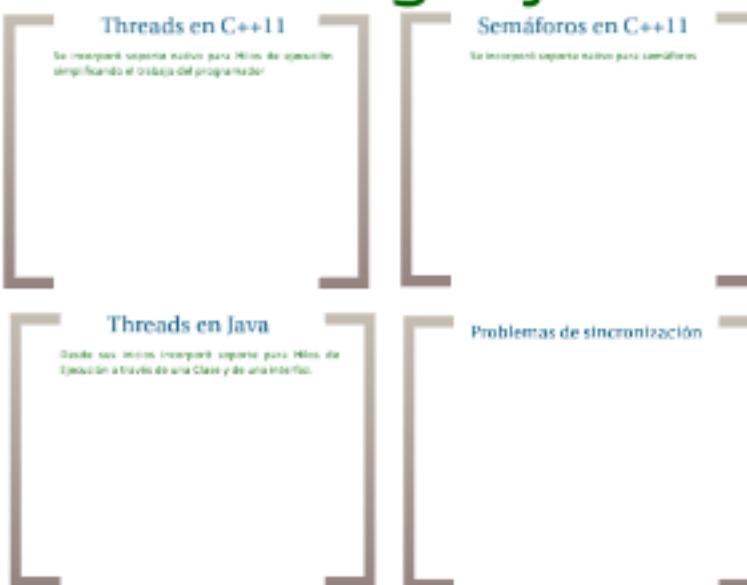
## Comienzos:



## Bibliotecas y Frameworks



## Soporte nativo en el Lenguaje



# Comienzos:

## Concurrencia en Algol 68

```
X=1;  
Y=X+1,  
W=X -1;  
Z=Y*X*W;
```

La coma permite que el lenguaje pueda generar código assembler independiente para las dos instrucciones.

El semáforo s no permite que se escriba la variable x al mismo tiempo

up levanta el semáforo y down lo baja

```
int x;  
sema s;  
par  
begin  
do  
    up s;  
    x = 1;  
    down s;  
od,  
do  
    up s;  
    x = 0;  
    down s;  
od,  
end
```

## Concurrencia en High Performance Fortran

El lenguaje genera instrucciones assembler en paralelo para modificar los elementos del arreglo i

```
!HPF$ INDEPENDENT  
DO i = 1,n  
    x(i) = i**2  
END DO
```

## Cláusula FORALL

```
FORALL(I = 1:N, J = 1:N, A(I, J) .NE. 0.0)  
    B(I, J) = 1.0 / A(I, J)
```

El lenguaje genera instrucciones assembler en paralelo para realizar las asignaciones.

# Concurrencia en Algol 68

```
X=1;  
Y=X+1,  
W=X -1;  
Z=Y*X*W;
```

La coma permite que el lenguaje pueda generar código assembler independiente para las dos instrucciones.

## Semáforos y cláusula par

El semáforo s no permite que se escriba la variable x al mismo tiempo

up levanta el semáforo y down lo baja

```
int x;  
sema s;  
par  
begin  
do  
    up s;  
    x = 1;  
    down s;  
od,  
do  
    up s;  
    x = 0;  
    down s;  
od,  
end
```

# Concurrencia en High Performance Fortran

El lenguaje genera instrucciones assembler en paralelo para modificar los elementos del arreglo i

## Cláusula INDEPENDENT

```
! HPF$ INDEPENDENT  
DO i = 1,n  
    x(i) = i**2  
END DO
```

## Cláusula FORALL

```
FORALL(I = 1:N, J = 1:N, A(I, J) .NE. 0.0)  
    B(I, J) = 1.0 / A(I, J)
```

El lenguaje genera instrucciones assembler en paralelo para realizar las asignaciones.

# POSIX Threads

Es una biblioteca de Linux, escrita en C, para la creación de Hilos de Ejecución desde cualquier lenguaje.

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const
pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

```
void f(int i) {...}

pthread_t th1;
th1 = thread_create(&th1, NULL, f, "1");
pthread_exit(NULL);
```

# Soporte nativo en el Lenguaje

## Threads en C++11

Se incorporó soporte nativo para Hilos de ejecución simplificando el trabajo del programador

## Semáforos en C++11

Se incorporó soporte nativo para semáforos

## Threads en Java

Desde sus inicios incorporó soporte para Hilos de Ejecución a través de una Clase y de una interfaz.

## Problemas de sincronización

# Threads en C++11

Se incorporó soporte nativo para Hilos de ejecución simplificando el trabajo del programador

```
#include <iostream>
#include <thread>
void f() {
    std::cout << "Hola" << std::endl;
}

int main() {
    std::thread t1(f)
    t1.join();
    return 0;
}
```

Se crea un Thread con la función f

Se asocia el Thread t1 con la función main()

# Semáforos en C++11

Se incorporó soporte nativo para semáforos

```
std::mutex mtx;
int global = 0;

void imprime () {
    mtx.lock();
    global=global+1;
    for(int i=0;i<100000;i++);
    std::cout << global << std::endl;
    mtx.unlock();
}
int main ()
{
for(int i=0;i<10;i++)
{
    std::thread th1 (imprime);
    std::thread th2 (imprime);

    th1.join();
    th2.join();

};
    return 0;
}
```

Se bloquea el semáforo

Se libera el semáforo

# Threads en Java

Desde sus inicios incorporó soporte para Hilos de Ejecución a través de una Clase y de una interfaz.

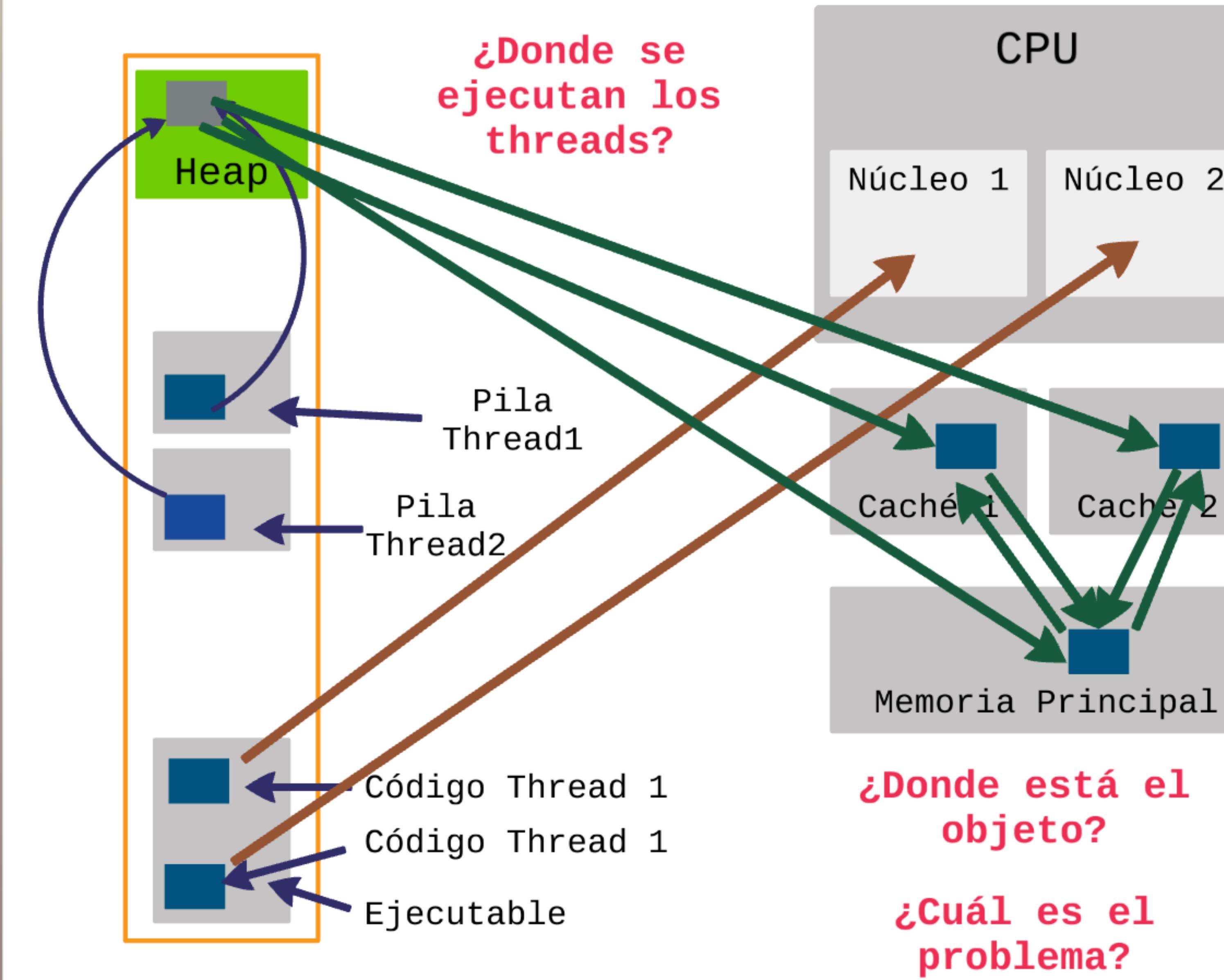
```
public class HolaThread implements  
Runnable {  
  
    public void run() {  
        System.out.println("Hola!");  
    }  
  
    public static void main(String args[])  
    { (new Thread(new HolaThread())).start();  
    }  
}
```

Runnable es una interfaz de Java

Se crea un objeto de la clase HolaThread

Se crea un objeto de la clase Thread

# Problemas de sincronización



# Soluciones de sincronización

- Declaración con modificador "volatile"

La variable siempre se lee desde memoria principal

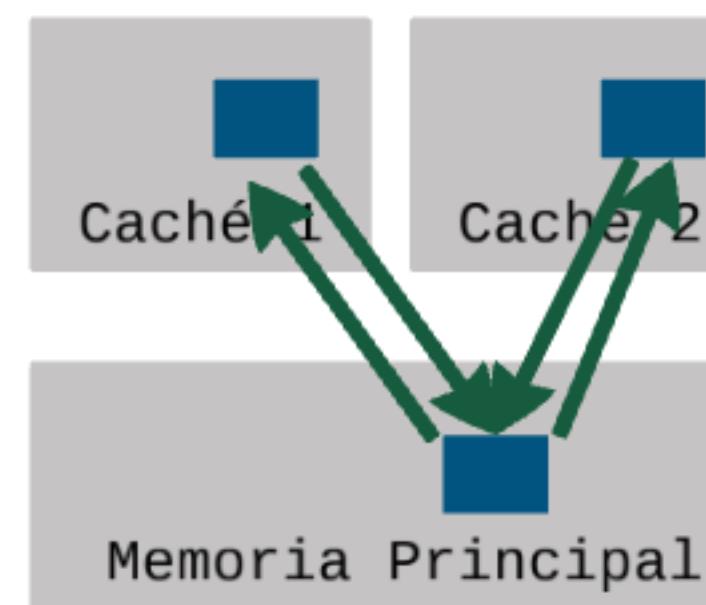
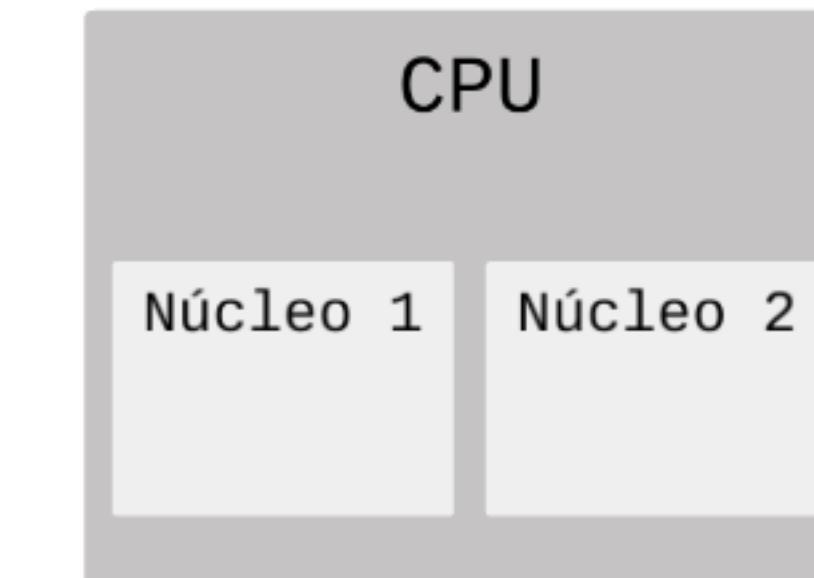
**¿Eso soluciona todos los problemas?**

**¿Qué sucede con las "condiciones de carrera?"**

Dos códigos pueden competir igualmente por el mismo espacio de memoria principal

Es necesario otro mecanismo

Los bloques de sincronización resuelven el problema de la consistencia



**Java**

```
public  
synchronized void  
f(int v){  
    this.x = v;  
}
```

# Pasaje de Parámetros

- Sintaxis
- Semántica

Lenguajes de Programación I

UNICEN

José M. Massa

# Pasaje de Parámetros

```
void F(int x, float y);  
{...};  
  
int a;  
float b;  
F(3,7.4);  
F(a,b)
```

```
PROCEDURE P (x: INTEGER);  
BEGIN  
  x := 3;  
  ...  
END;  
...  
var a : integer;  
...  
P(a)
```

```
public class C{  
  public M1(String n){...}  
};  
...  
C a;  
String b;  
a.M1(b);  
a.M1("Hola")
```

- **Parámetros formales**

Se definen en el encabezado de la función/procedimiento, método

- **Parámetros reales**

Son los que se utilizan en la invocación

- **Asociación entre parámetros reales y formales**

# Aspectos sintácticos y semánticos

## Aspecto sintáctico

- Cómo se asocian sintácticamente los parámetros reales y formales

Posición

Explícito

Anónimo

## Aspecto semántico

- Qué es lo que ocurre cuando se pasan parámetros

Referencia

Nombre

Copia

# Posición

Asociación por la posición  
de los parámetros en la  
definición (formales) y la  
invocación (reales)

```
PROCEDURE P (x: INTEGER;y : FLOAT;z : INTEGER);  
BEGIN  
    x := 3;  
    Y := 5.7;  
    ...  
END;  
...  
var a,b : integer;  
    c : float;  
...  
P(a,c,b);
```

¿Dónde pueden estar los parámetros reales?

¿Tienen que estar todos los parámetros reales?

En FORTRAN si. Muy útil para asegurar  
la consistencia del cálculo de  
fórmulas matemáticas, pero tedioso  
para la programación en otras áreas de  
aplicación

# ¿Qué sucede en este caso?

```
SUBROUTINE S(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10)
  INTEGER a1,a2,a3,a4,a5,a6,a7,a8,a9,a10
  SUMA=a1+a2+a3+a4+a5+a6+a7+a8+a9+a10
  RETURN
END

CALL S(b1,b2,b3,0,b5,0,b7,0,0,b10)
```

Esta solución obliga a que el llamador ponga un valor especial para los parámetros que no necesita pasar

**Las primeras versiones de FORTRAN y otros lenguajes no admitían parámetros faltantes**

Luego aparecieron lenguajes que soportaban parámetros faltantes

# Parámetros faltantes

Se define la función:      float ff(int a;int b);

Se invoca como:      • ff(z) ←      ¿Cuál es el parámetro faltante?

## Parámetros Faltantes al final

Sólo pueden estar al final de los  
parámetros de la función

No es posible que un parámetro  
del medio pueda faltar

C++:

```
void ff(int a;float b=3.2;int c = 4);
...
ff(5);
ff(5,3.6);
```

# Parámetros Faltantes en cualquier lugar

C#:

```
public void Metodo1(int r, string o = "Valor por defecto", float q = 1.3)  
Metodo1(5, q : 3.2);
```

De ésta manera se indica que falta el parámetro o, pero no el parámetro q

Python:

```
def ff(str1, a=1, str2='No'):  
ff('Hola')           ← Estas llamadas son válidas  
ff('Hola',2)  
ff('Hola',2, 'que tal')  
ff('Hola',a=3)       ← Estas también son válidas  
ff('Hola',str2='¿cómo andas?')
```

¿Qué otra diferencia existe entre el ejemplo de C# y Python?

C# es un lenguaje tipo-Algol y Phyton es dinámico, sin embargo comparten características sintácticas del pasaje de parámetros

# Parámetros faltantes

Se define la función: float ff(int a;int b);

Se invoca como: • ff(z) ← ¿Cuál es el parámetro faltante?

## Parámetros Faltantes al final

Sólo pueden estar al final de los parámetros de la función

No es posible que un parámetro del medio pueda faltar

C++:

```
void ff(int a;float b=3.2;int c = 4);
...
ff(5);
ff(5,3.6);
```

Parámetros Faltantes en cualquier lugar

```
Java:
public void Metodo1(int r, string s = "Valor por defecto", float q = 1.3)
Metodo1(5,q : 3.2);
```

De esta manera se indica que falta el parámetro s, pero no el parámetro q.

```
Python:
def ff(str1, a=1, str2='Hola'):
ff('Hola')
ff('Hola',2)
ff('Hola',2, 'que tal')      Estas llamadas son válidas
ff('Hola',a=3)              Estas también son válidas
ff('Hola',str2='como andas?')
```

¿Qué otra diferencia existe entre el ejemplo de C# y Python?  
C# es un lenguaje tipo Algodón y Python es dinámico, sin embargo comparten características sintácticas del pasaje de parámetros

Caso ridículo:

```
void ff(int a;int b;float c;char d;double e;int f;float g;int h;int i);
• ff(,,,,,5,,);
```

# Explícito

Los casos de C# y Phyton indican una situación nueva respecto del pasaje de parámetros por posición

C#:

```
public void Metodo1(int r, string o = "Valor por defecto", float q = 1.3)  
Metodo1(5, q : 3.2);
```

Esto indica que el llamador debe conocer los nombres de los parámetros formales

Esto ya había sido implementado antes en el lenguaje ADA

ADA:

```
function Ecuacion (a,b,c,d: Integer) return Integer is  
begin  
...  
end Ecuacion;
```

```
Ecuacion(i=>a, j=>b, k=>c, l=>d);
```

- Esto indica que el llamador debe conocer los nombres de los parámetros formales
- Dificulta la mantenibilidad de los programas

- Es útil para cuando existen muchos parámetros opcionales o faltantes y se quieren pasar pocos

**¿Qué sucede con la idea de que en los lenguajes Tipo-Algol los nombres de las variables no están en el programa?**

# Explícito

Los casos de C# y Phyton indican una situación nueva respecto del pasaje de parámetros por posición

C#:

```
public void Metodo1(int r, string o = "Valor por defecto", float q = 1.3)  
Metodo1(5, q : 3.2);
```

Esto indica que el llamador debe conocer los nombres de los parámetros formales

Esto ya había sido implementado antes en el lenguaje ADA

ADA:

```
function Ecuacion (a,b,c,d: Integer) return Integer is  
begin  
...  
end Ecuacion;
```

```
Ecuacion(i=>a, j=>b, k=>c, l=>d);
```

- Esto indica que el llamador debe conocer los nombres de los parámetros formales
- Dificulta la mantenibilidad de los programas

- Es útil para cuando existen muchos parámetros opcionales o faltantes y se quieren pasar pocos

**¿Qué sucede con la idea de que en los lenguajes Tipo-Algol los nombres de las variables no están en el programa?**

**Cuando se compila, se transforma la notación explícita a posicional**

# Anónimo

## Cantidad variable de parámetros

```
C  
void ff(int a,...);  
...  
int x,y;  
ff(x,y,3);
```

- Cantidad variable de parámetros
- El invocado no conoce la cantidad ni el tipo de los parámetros

Convención: El primer parámetro indica la cantidad y el tipo

```
C  
int printf(const char *format, ... );  
printf("%c,%d,%4.2f", 'a', 34, 3.65);
```

Cantidad de caracteres %: cantidad de parámetros

A continuación, tipo y formato de los parámetros

C

```
void imprimir(const char *fmt, ...)  
{  
    const char *p;  
    for(p = fmt; *p != '\0'; p++)  
    {  
        switch(*++p)  
        {  
            case 'c':  
                imprimir caracter  
                break;  
            case 'd':  
                imprimir entero  
                break;  
        ...  
    }
```

Itera sobre la lista de parámetros

- Cantidad variable de parámetros
- El primer parámetro dice la cantidad y el tipo

Los parámetros se pasan como una lista de enteros

Java

```
static void f(int... x)  
{  
    System.out.println(x.length);  
    System.out.println();  
    for (int i : x)  
        System.out.println(i);  
}
```

# Aspectos sintácticos y semánticos

## Aspecto sintáctico

- Cómo se asocian sintácticamente los parámetros reales y formales

### Posición

Asociación por la posición de los parámetros en la definición (formales) y la invocación (reales)

¿Dónde pueden estar los parámetros reales?  
¿Tienen que estar todos los parámetros reales?

En FORTRAN sí. Muy útil para asegurar la consistencia del cálculo de fórmulas matemáticas, pero tedioso para la programación en otras áreas de aplicación

### Explícito

los lenguajes C y Python indican una situación nueva respecto del paso de parámetros por posición

Este ya había sido implementado anteriormente en el lenguaje ADA.

```
ADA
function relacion (A,B,C:D; Integer) return Integer;
begin
  end;
end relacion;
```

En C++ se usa cuando necesitas muchos parámetros posicionales o fallarías y no quieras usar posiciones de los parámetros formales

¿Qué sucede con las ideas de que en los lenguajes Tigo-Algoel los nombres de las variables no están en el programa?

Cuando se compila, se transforma la notación explícita a posicional

### Anónimo

Cantidad variable de parámetros

•Cantidad variable de parámetros  
•El invocado no conoce la cantidad ni el tipo de los parámetros

Convenção: El primer parámetro indica la cantidad y el tipo

```
C
void ff(int a,...);
...
int x,y;
ff(x,y,3);
```

```
C
int printf(const char *format, ... );
printf("%c,%d,%4.2f", 'a',34,3.65);
```

Cantidad de caracteres N: cantidad de parámetros  
A continuación, tipo y formato de los parámetros

## Aspecto semántico

- Qué es lo que ocurre cuando se pasan parámetros

### Referencia

### Nombre

### Copia

# Referencia

```
procedure A;  
var x,y : integer;  
procedure B(l: integer;m:integer);  
begin  
    l := 5;  
    m := 9;  
end;  
begin  
    x :=3;  
    y := 4;  
    B(x,y);  
end;
```

Se crea un "alias" entre cada parámetro real y su correspondiente parámetro formal



Referencian la misma celda de memoria



Las modificaciones sobre el parámetro formal se reflejan en el parámetro real

| B |   |     |
|---|---|-----|
| A | y | 4 9 |
| A | x | 3 5 |

¿En qué otro lugar pueden estar x e y?

Deben ser locales de A, no locales a A o globales

¿Dónde están l y m ?

En ningun lado, son "alias" de x e y

# Pasaje de parámetros por referencia

- Fue el primer mecanismo de pasaje de parámetros
- Implementado por FORTRAN

```
procedure A;  
var x,y : integer;  
procedure B(i: integer);  
var z : integer;  
begin  
    z := i;  
    i := 3;  
end;  
begin  
XB(x+y);  
B(3);  
end;
```

## Problemas con:

- Constantes
- Expresiones

¿Por qué no se puede hacer esto?

x+y y 3 no son variables  
referenciables

# Pasaje de parámetros por referencia en C++

```
void ff(int& i)
{
    i = 3;
    printf("%d\n", &i);
}

int main(int argc, char* argv[])
{
    int a=4;
    ff(a);
    printf("%d\n", a);
    printf("%d\n", &a);
}
```

Se indica que el pasaje de parámetros es por referencia

# Nombre

- Se reemplaza el nombre del parámetro formal por el del real
- Se recompila el programa

```
procedure A;  
var x,y : integer;  
procedure B(l: integer;m:integer);  
begin  
    l := 5; → x := 5;  
    m := 9; → y := 9;  
end;  
begin  
    x :=3;  
    y := 4;  
    B(x,y);  
end;
```

```
procedure A;  
var x,y : integer;  
procedure B(l: integer;m:integer);  
begin  
    l := 5; → x+y := 5;  
    m := 9; → 3 := 9; X  
end;  
begin  
    x :=3;  
    y := 4;  
    B(x+y,3);  
end;
```

Estos errores se atrapan en las primeras fases de la compilación (luego del reemplazo)

- Mejoró respecto al pasaje de parámetros por referencia en que ahora los errores se detectan en las primeras fases tiempo de compilación (análisis sintáctico)
- Ineficiente, habría tantas copias de la unidad llamada como llamadores diferentes existan

El caso de C

# Macros de C

```
#define VALOR_MACRO 100
```

Se reemplaza VALOR\_MACRO por 100

C

```
#define x a+b
int ff(x,y)
{
    int c=2;
    return 8*x+c;
}
...
a = 3; b = 4;
d = ff(a+b,4);
```

Se espera un valor  
de 58 para d



C

```
#define x a+b
int ff(x,y)
{
    int c=2;
    return 8*a+b+c;
}
...
a = 3; b = 4;
d = ff(a+b,4);
```

8\*3+4+2

Se obtiene un valor  
de 30 para d

La asociatividad se perdió

Esto en Algol no pasa porque se re-compila

# Ejemplo

```
void swap(x,y)
    temp = x;
    x = y;
    y = temp;
end;

i=2;a[i]=5;

swap(i,a[i]);
```

**Se está modificando otra celda**



```
void swap(i,a[i])
    temp = i;
    i = a[i];
    a[i] = temp;
end;
```



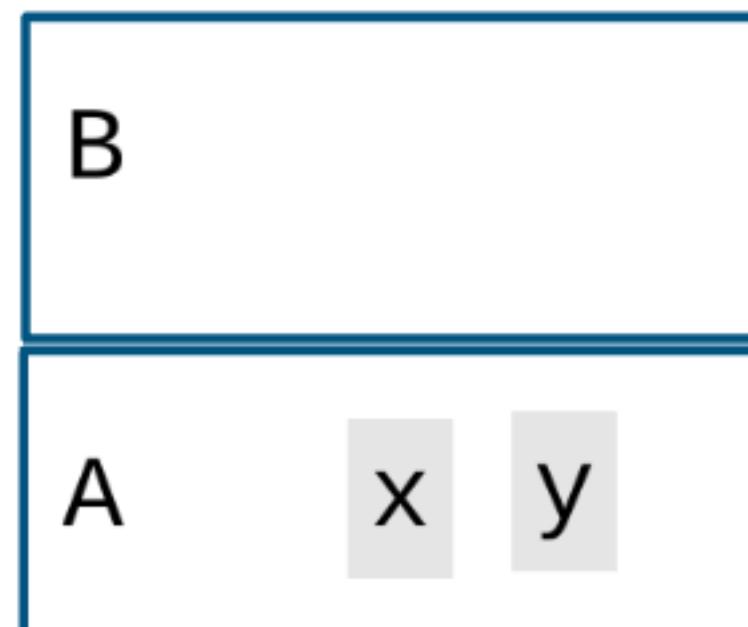
```
void swap(i,a[i])
    temp = 2;
    i = a[2];
    a[5] = temp;
end;
```

# Implicancias del pasaje de parámetros por nombre

- Mejoró respecto del pasaje de parámetros por referencias
- Su implementación efectiva resulta ineficiente
- Ocasiona problemas al reemplazar los nombres de los parámetros formales por lo de los reales

**En tiempo de ejecución ¿dónde se encuentran los parámetros formales?**

- En ningún lugar
- Se usan los parámetros reales



```
procedure A;  
procedure B(k : integer;l: integer);  
begin...end;  
begin  
  B(x,y);  
end
```

# Pasaje de Parámetros por Nombre

Reemplazar el nombre del parámetro formal por el real

## Problemas

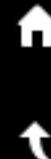
```
void g() {  
    void f(int a) {  
        int x;  
        x = a;}  
    int y,z;  
    f(y);  
    f(y+z);  
    f(3);  
}
```

Problemas en expresiones, constantes  
y parámetros faltantes

Problemas de alcance

Problemas de redeclaraciones

## Implementaciones



# Problemas

```
void g() {  
  
    void f(int a) {  
        int x;  
        x = a;}  
  
    int y,z;  
  
    f(y); → CALL f'  
  
    f(y+z); → CALL f''  
  
    f(3); → CALL f'''  
}
```

```
void g() {  
  
    void f'(int a) {  
        int x;  
        x = y;}  
  
    void f''(int a) {  
        int x;  
        x = y+z;}  
  
    void f'''(int a) {  
        int x;  
        x = 3;}
```

¿Qué sucede con los parámetros formales?

El nombre del parámetro formal no interesa luego de la re-compilación

Es necesario compilar tantas copias de la función llamada como llamados diferentes haya

# Problemas de alcance

```
void f(int a) {  
    int x;  
    x = a;}
```

```
void g() {  
    int z;  
    f(z);  
}
```



```
void f(int a) {
```

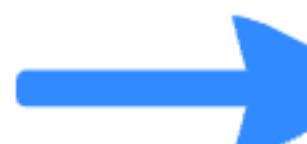
```
    int x;  
    x = z;}
```

```
void g() {  
    int z;  
    f(z);  
}
```

¿Se puede  
usar la  
variable z?

```
int z;  
void f(int a) {  
    int x;  
    x = a;}
```

```
void g() {  
    f(z);  
}
```



```
int z;  
void f(int a) {  
    int x;  
    x = z;}
```

```
void g() {  
    f(z);  
}
```

Puede suceder que en la función llamada, las variables del parámetro real no estén al alcance.

# Problemas de redeclaraciones

```
void f(int a) {  
    int z;  
    x = a;}  
  
void g() {  
    int z;  
    f(z);  
}
```



```
void f(int a) {  
    int z;  
    x = z;}  
  
void g() {  
    int z;  
    f(z);  
}
```

¿Cuál  
variable z  
se utiliza?

Si el lenguaje no admite re-uso de nombre de variables (shadowing), se produce un error de compilación por redeclaración de variables

Si el lenguaje admite re-uso de nombre de variables (shadowing), se utiliza la última declarada.

# Problemas en expresiones, constantes y parámetros faltantes

```
void f(int a)
{
    int x;
    a = x; }

void g() {
    int y,z;
    h(y+z);
}
```



```
void f(int a) {
    int x;
    y+z = x; }

void g() {
    int y,z;
    h(y+z);
}
```

```
void f(int a) {
    int x;
    a = x; }

void g() {
    h(3);
}
```



```
void f(int a) {
    int x;
    3 = x; }

void g() {
    h(3);
}
```

Pueden surgir errores al intentar usar el parámetro formal del lado izquierdo

**¿Son errores de compilación o de ejecución?**

# Implementaciones

- Nombre propiamente dicho



- Se compilan tantas funciones llamadas como diferentes invocaciones haya (Algol 68)

- Macros (C/C++)



- Se reemplaza el parámetro formal por el parámetro real especificado en la cláusula de definición (C, C++)

- Dirección del parámetro real



- Se compilan tantas funciones llamadas como invocaciones diferentes haya

- Funciones Thunk

Funciones Thunk



- Se utiliza la dirección del parámetro real directamente en la función llamada cada vez que se usa

- Se generan funciones que resuelven el parámetro real

(Pascal y otros que admiten funciones thunk como bibliotecas)

# Funciones Thunk

Son funciones que se encargan de evaluar los parámetros reales. El lenguaje reemplaza los parámetros formales por las llamadas a estas funciones.

```
void f(int a) {  
    int x;  
    x = a;}  
  
void g() {  
    int z;  
    f(z_thunk());  
}
```

```
void f(int a_thunk())  
{  
    int x;  
    x = a_thunk();  
}
```

```
void z_thunk() {  
    return z;  
}
```

El programador o el  
lenguaje inserta este  
llamado

Esta función puede ser  
escrita por el  
programador o el  
lenguaje

**¿Cómo se resuelve el ámbito?**  
**¿Cómo hace la función thunk() para acceder a la variable z?**

La función thunk se ejecuta como un closure

El lenguaje construye la función thunk del llamador y este se la pasa al llamado para que el mismo pueda acceder a los parámetros reales

Por valor

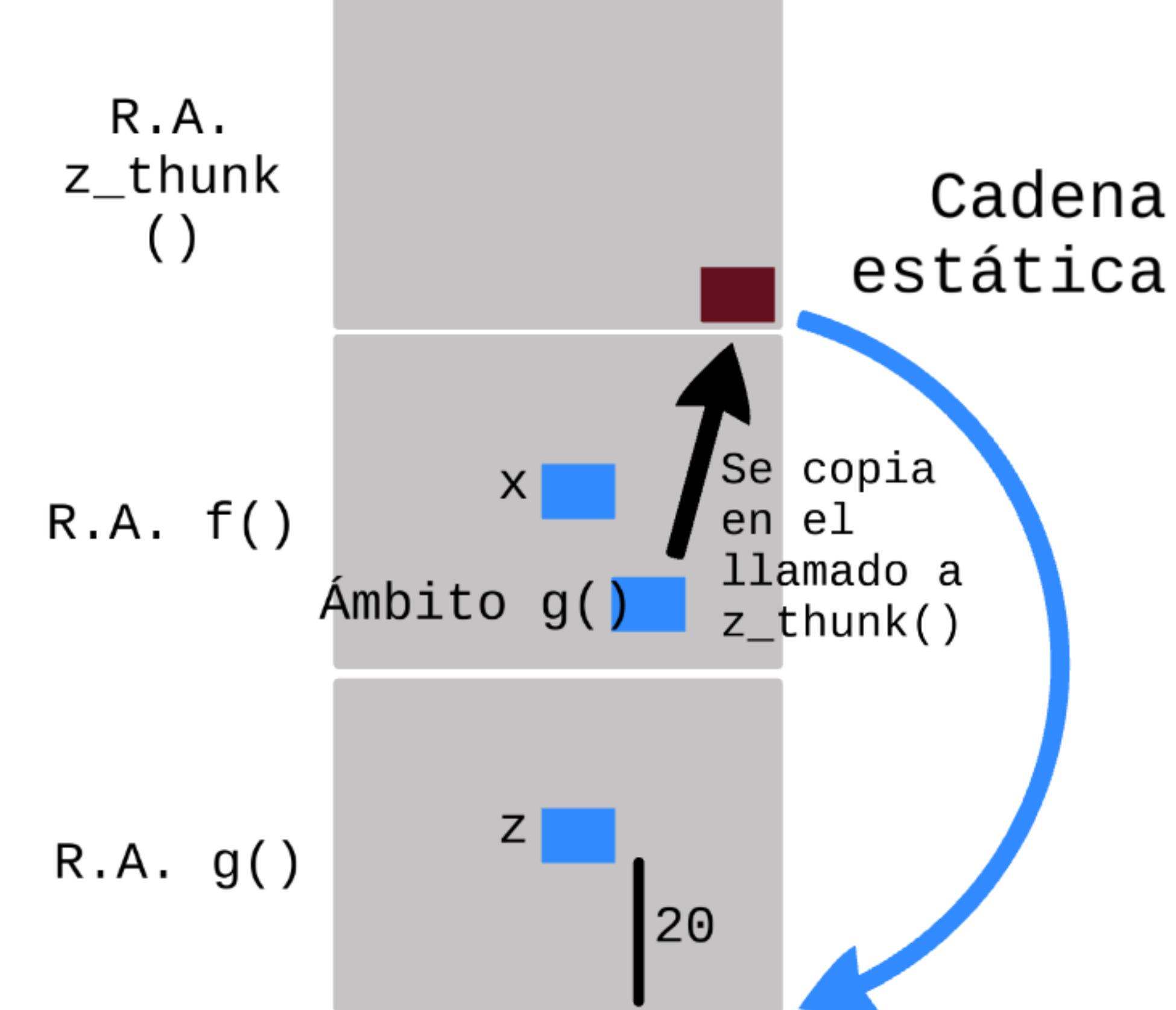
Por referencia

# Por valor

```
void f(int a) {  
    int x;  
    x = a;}  
  
void g() {  
    int z;  
    f(z_thunk());  
}
```

```
void z_thunk() {  
    return z;  
}
```

MOV SP, SP[8]  
RET SP[20]



La función Thunk puede acceder al parámetro real a través de la cadena estática

# Por referencia

```
void f(int a) {  
    int x;  
    x = a;}  
  
void g() {  
    int z;  
    f(z_thunk());  
}
```

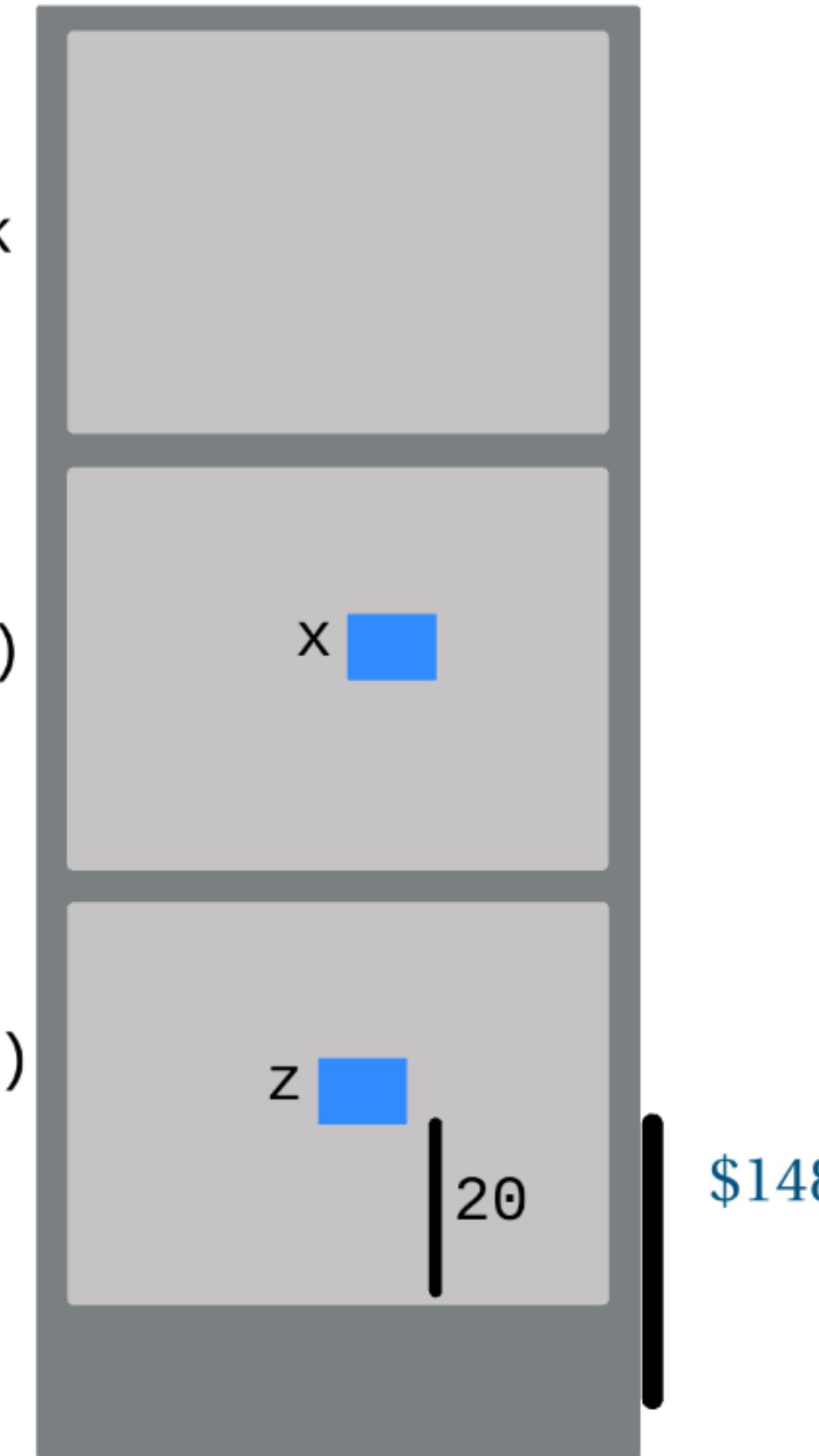
```
void z_thunk() {  
    return *z;  
}
```

RET [\$148]

R.A.  
z\_thunk  
( )

R.A. f()

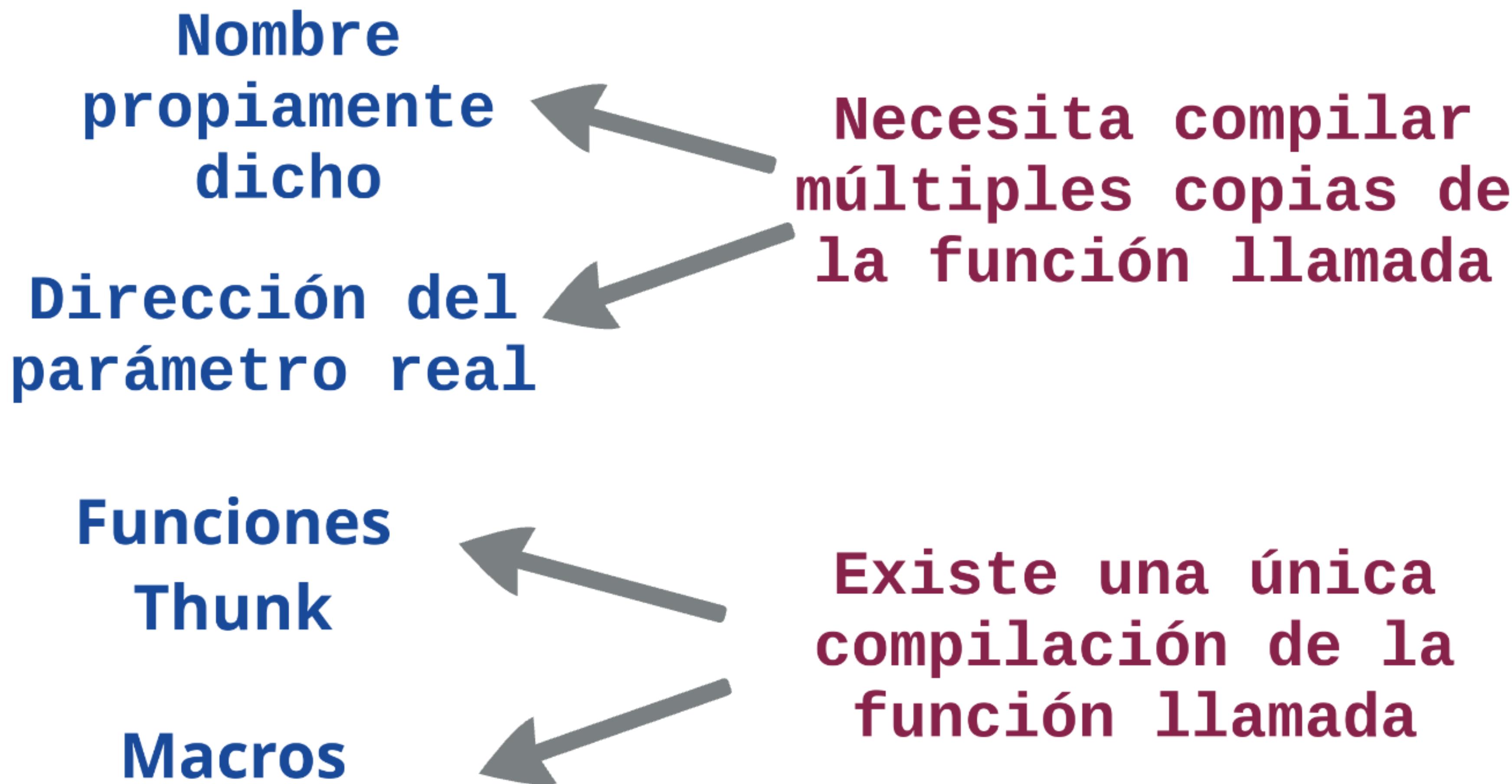
R.A. g()



La función Thunk accede al parámetro real a través de su dirección

# Conclusiones

Implementaciones de pasaje de parámetros por nombre



# Copia

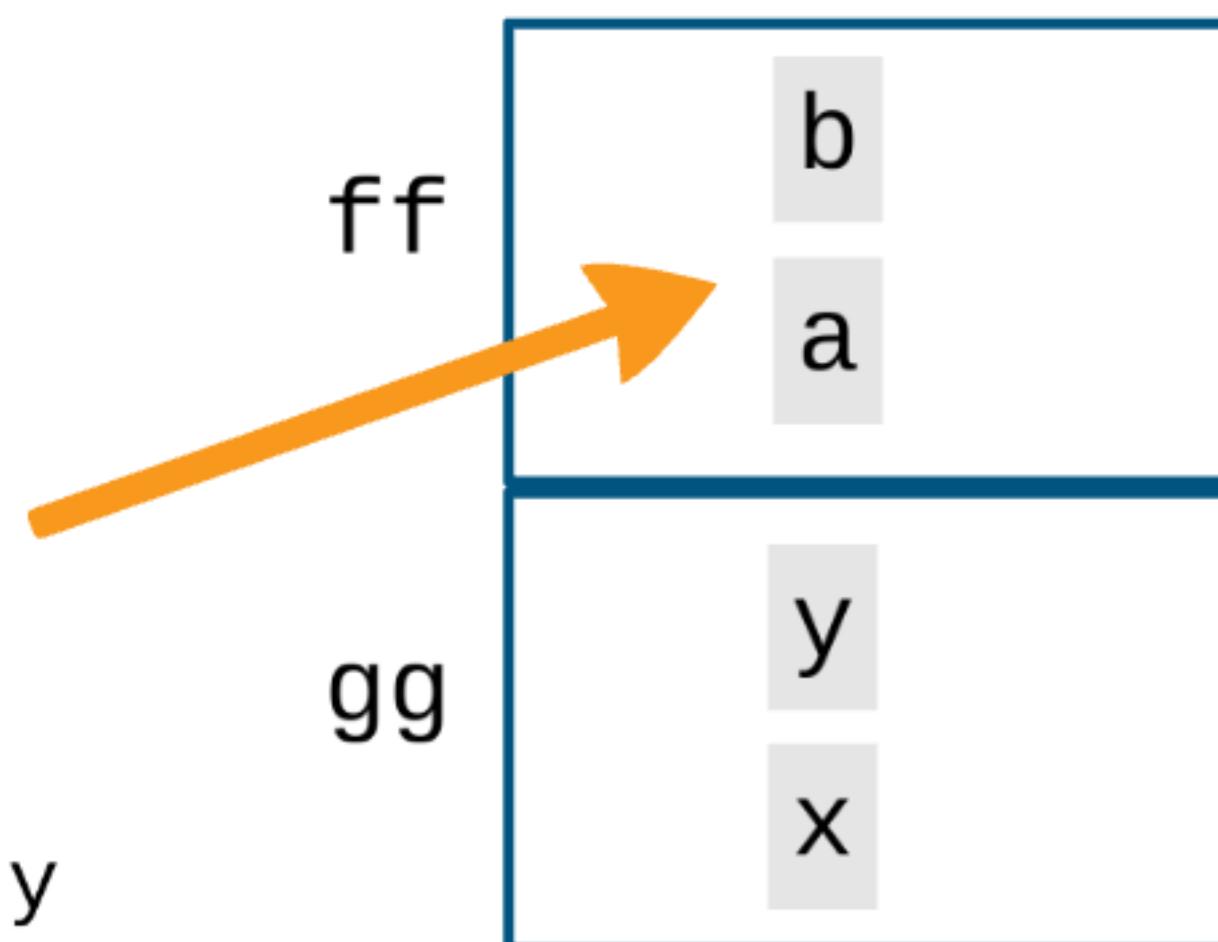
Se copian los valores de los parámetros reales en los parámetros formales

Los parámetros reales y formales se desacoplan

```
int ff(int a,int b);  
{  
}  
void gg;  
{  
    int x=3;  
    int y=4;  
    ff(x,y);  
}
```

Copia valor

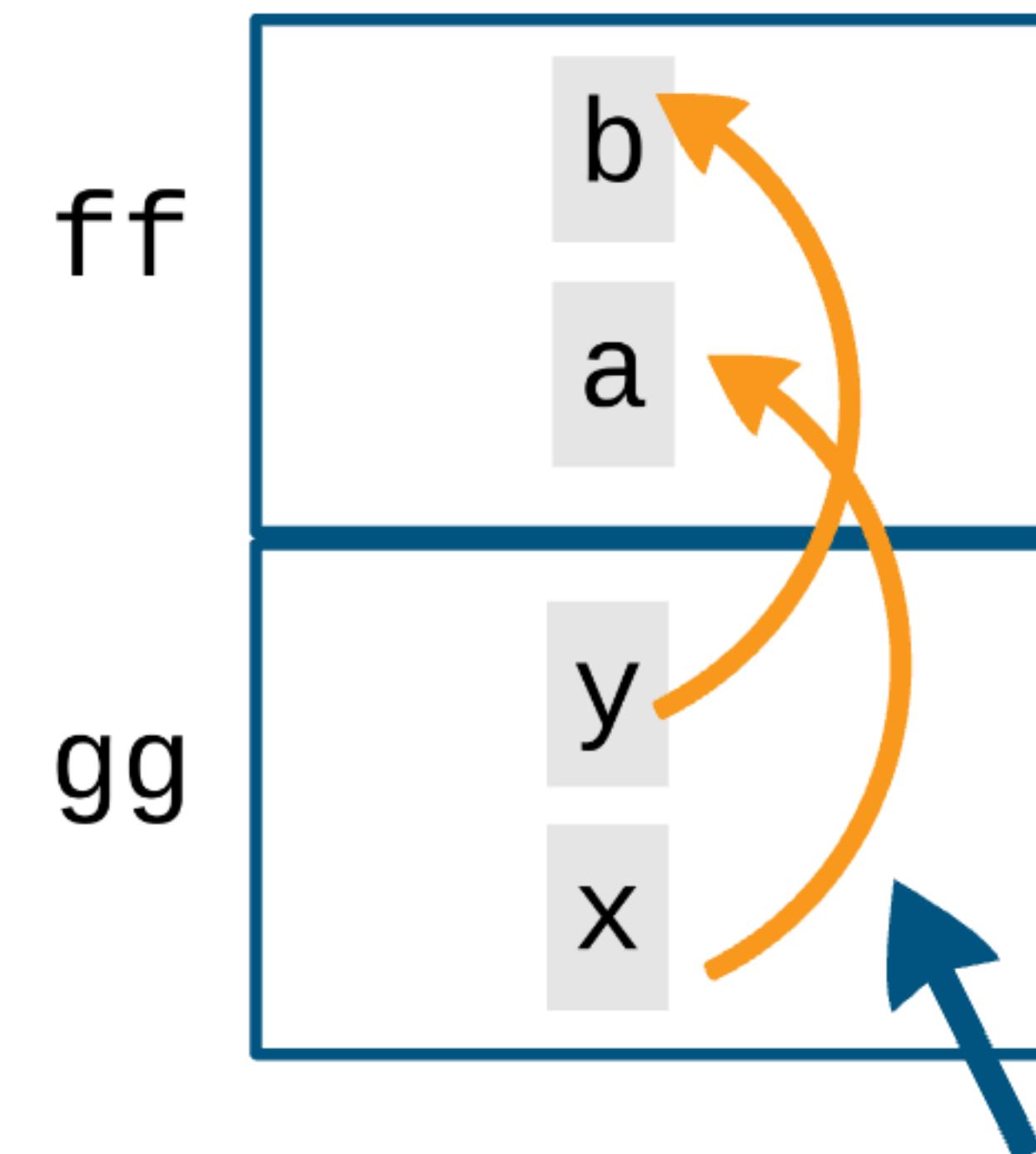
La función ff tiene en su registro de activación una copia de los parámetros x e y



# Copia valor

Se copia el parámetro real sobre el formal en el momento de la invocación

```
int ff(int a,int b);  
{  
    int j = a;  
    b = j; ← Esta instrucción  
no modifica el  
valor de y  
}  
void gg;  
{  
    int x=3;  
    int y=4;  
    ff(x,y);  
}
```



¿Cómo se indica que se quiere utilizar este mecanismo?

**Ojo!, estas flechas  
no indican punteros**

# Sintaxis para pasaje de parámetros de copia valor

## Implícito en muchos lenguajes

```
class Ejemplo
{
    static void MCopiaValor(int i)
    {
        int a = i;
    }
    static void Main()
    {
        int x=3;
        MCopiaValor(x);
    }
}
```

Al no colocar nada, el mecanismo por defecto es por copia valor

- Existen muy pocos lenguajes que admiten un modificador de parámetro "in"
- Se utiliza generalmente para mejorar la legibilidad del programa

- En muy pocos casos se utiliza para prohibir la asignación a un parámetro que es por copia valor (Ej. Oracle PL/SQL)

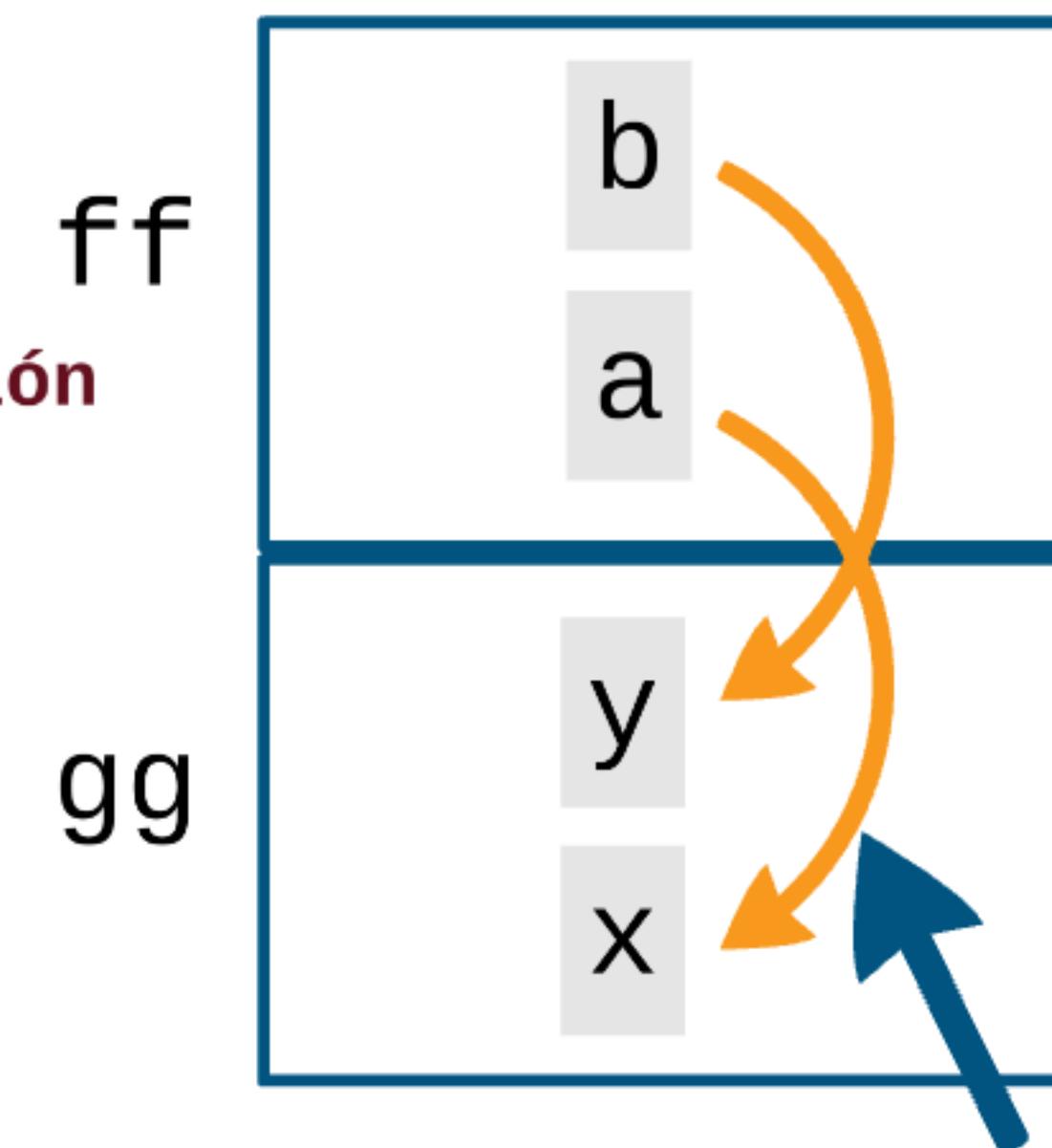
# Copia resultado

Se copia el parámetro formal sobre el real al finalizar la invocación

El retorno de la función suele ser un parámetro de copia resultado

```
int ff(int a,int b);
{
    ...
    b = 3;
}
void gg;
{
    int x=3;
    int y=4;
    ff(x,y);
}
Ahora y vale 3
```

Esta instrucción modifica el valor de y



¿Cómo se indica que se quiere utilizar este mecanismo?

Ojo!, estas flechas no indican punteros

# Sintaxis para pasaje de parámetros de copia resultado

Explícito en la mayoría de los lenguajes que implementan copia-resultado

## ADA

```
procedure F is
  X : Integer := 4;
  procedure G (A : out Integer)
  is
    begin
      A := 3;
    end G;
  begin
    G(X);
    ahora X tiene el valor 3
  end F;
```

¿Qué diferencias tiene con el pasaje por referencia?

Otros lenguajes que lo implementan:

- Swift (Apple)
- Oracle PL/SQL

# Copia valor-resultado

- Se copia el parámetro real sobre el formal en la invocación
- Se copia el parámetro formal sobre el real al finalizar la invocación

## ADA

```
void outer (void)
{
    int a = 5, b = 7;
    void inner (IN OUT int c; REF int d)
    {
        printf ("a: %d b: %d c: %d d: %d\n", a,b,c,d);
        a = 0; b = 9; c = 4; d = 6;
        printf ("a: %d b: %d c: %d d: %d\n", a,b,c,d);
    }

    inner(a,b);
    printf ("a: %d b: %d\n", a,b);
}
```

Extraido de: <http://c2.com/cgi/wiki?CallByValueResult>

a: 5 b: 7 c: 5 d: 7

a: 0 b: 6 c: 4 d: 6

a: 4 b: 6

- El parámetro formal c se copió sobre el parámetro real a
- el parámetro real b se pasó por referencia

# Problema

C

```
void ff2(int *p)
{
    *p = 2;
}

int main()
{
    int a=3;
    int *pa;
    pa = &a;
    ff2(pa);
    printf("%d",a);
}
```

¿Qué tipo de pasaje de parámetros es éste?

- La función ff2 consigue tener el valor del parámetro
- La función invocadora (main) recibe de vuelta el valor modificado por la invocada (ff2)

¿Es realmente un pasaje de parámetros por copia valor-resultado?

- NO. Porque no se COPIA el parámetro formal en el real.
- Se pasa el puntero por copia valor y el resultado se obtiene accediendo a la variable apuntada

# Otro problema

Java

```
public class B{public int z;}  
public class A{  
    public int M(B b1) { b1.z = 3};  
    public main()  
    {  
        B b0 = new(B);  
        b0.z = 4;  
        this.M(b0);  
    }  
}
```

¿Qué tipo de pasaje de  
parámetros es éste?

# Relación entre sintaxis y semántica de pasaje de parámetros

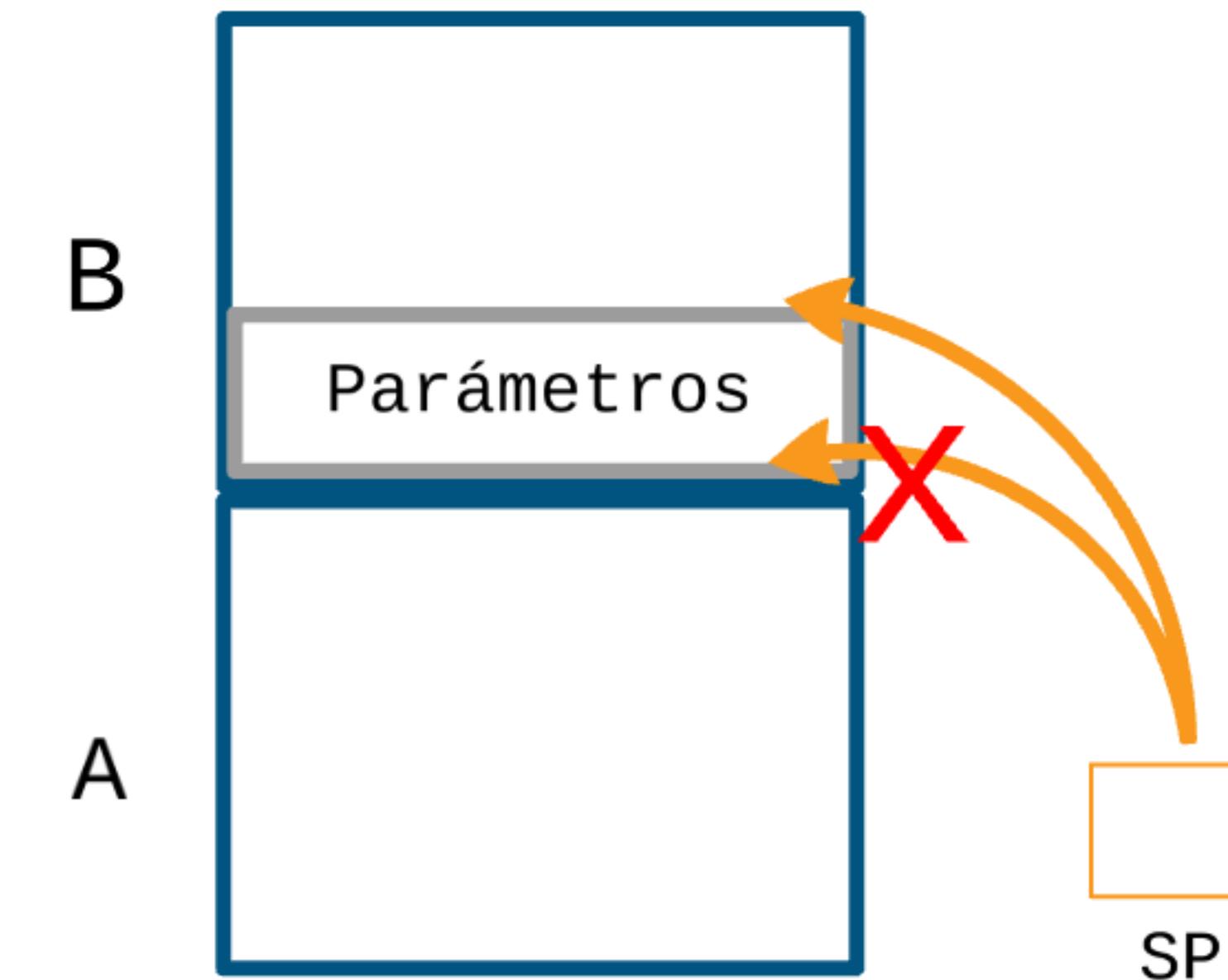
| Sem\Sin         | Posición | Faltantes         | Explícitos | Anónimos |
|-----------------|----------|-------------------|------------|----------|
| Referencia      | ✓        | ✗                 | ✓          | ✗        |
| Nombre          | ✓        | ✗                 | ✓          | ✗        |
| Valor           | ✓        | Valor por defecto | ✓          | ✓        |
| Resultado       | ✓        |                   | ✓          | ✗        |
| Valor-resultado | ✓        | ✗                 | ✓          | ✗        |

C# combina parte de la semántica de valor, resultado y valor-resultado con referencia

# Pasaje de parámetros anónimos en el registro de activación

¿Cómo se hace si tenemos punteros a la base?

- Los parámetros pueden ocupar diferente tamaño en diferentes invocaciones
- la solución es que el puntero de pila apunte a la interfaz entre los parámetros y la unidad invocada.



SP[Offset] Para ubicar las variables

SP[-offset] Para ubicar los parámetros

# Casos problemáticos en el pasaje de parámetros

|                 | Expresiones                 | Constantes                             |
|-----------------|-----------------------------|--|
| Referencia      | ✗                           | ✗                                      |
| Nombre          | Funciona pero con problemas | Solo si la constante está a la derecha |
| Valor           | ✓                           | ✓                                      |
| Resultado       | ✗                           | ✗                                      |
| Valor-Resultado | ✗                           | ✗                                      |

Variantes  
in y out

# Pasaje de Parámetros

Press Esc to exit full screen

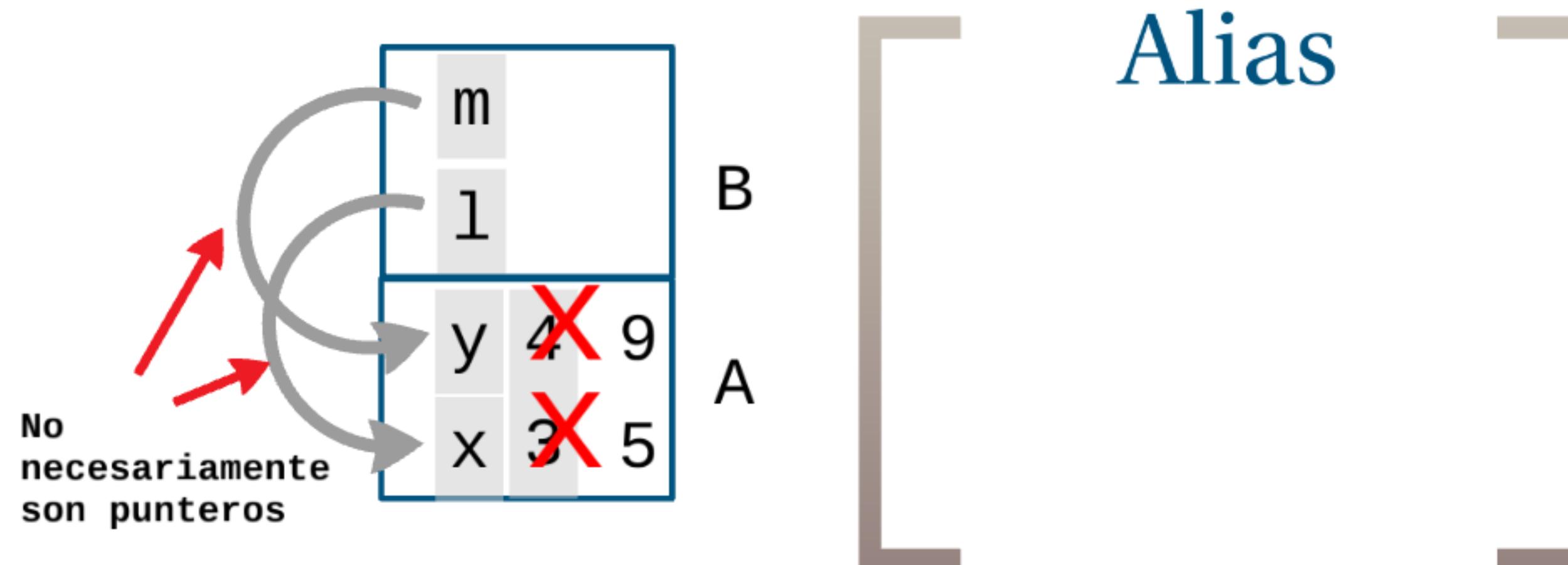
- Problemas de pasaje de parámetros como referencia
- Pasaje de arreglos como parámetros
- Parámetros en lenguajes dinámicos
- Pasaje de funciones como parámetros
- Closures

Lenguajes de Programación I  
UNICEN

# Referencia

```
procedure A;  
var x,y : integer;  
procedure B(l: integer;m:integer);  
begin  
    l := 5;  
    m := 9;  
end;  
begin  
    x :=3;  
    y := 4;  
    B(x,y);  
end;
```

- Referencian la misma celda de memoria
- Las modificaciones sobre el parámetro formal se reflejan en el parámetro real



## Punteros

- Algunos lenguajes implementan el pasaje de parámetros por referencia como punteros ocultos al programador

# Alias

```
void f(int &x, int &y)
{
    x = y;
}
f(a,a);
```

```
void f(int &x, int &y)
{
    x++;
    ...
}
f (1,l[1]);
```

X se transformó en un alias de y

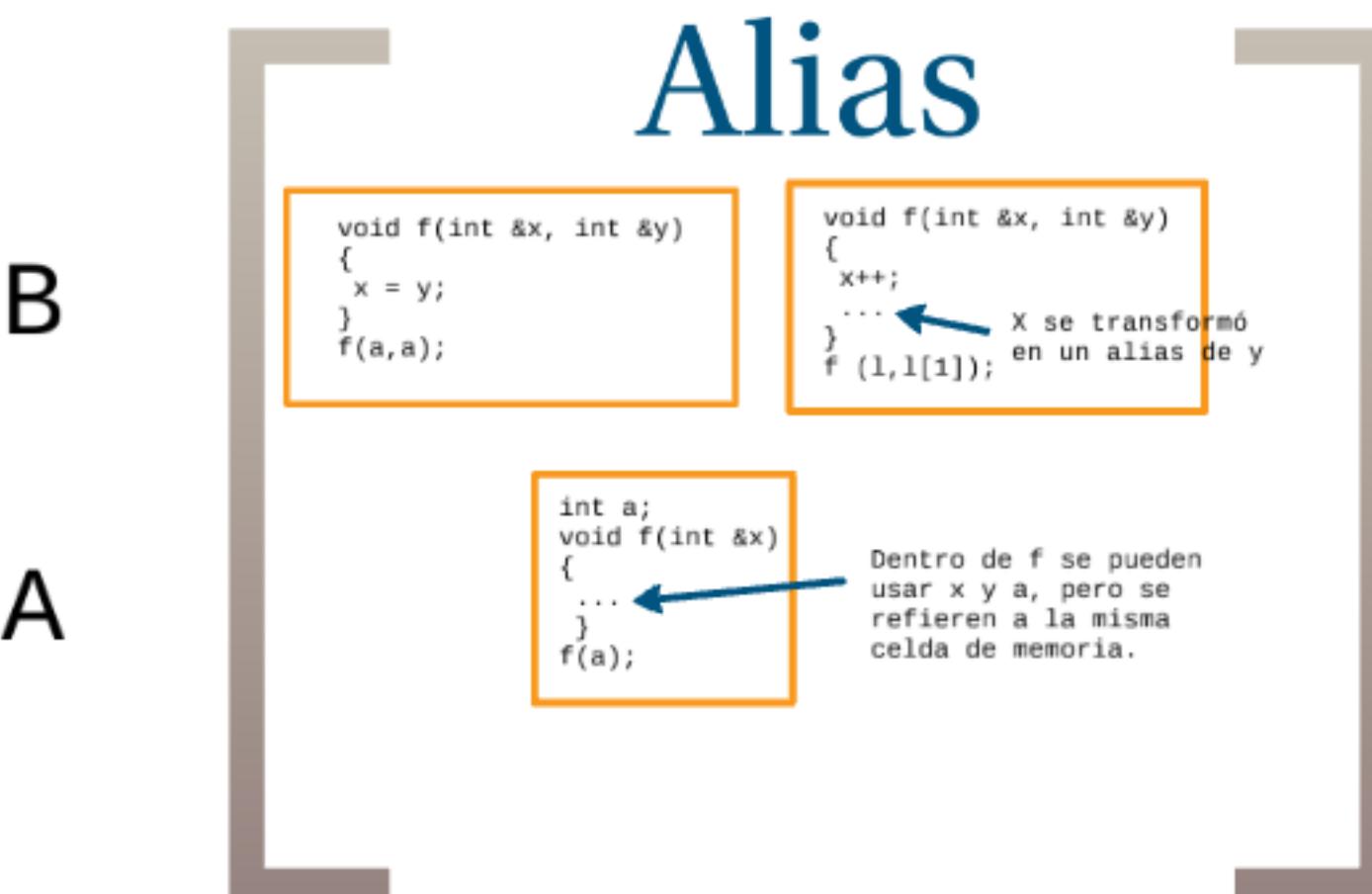
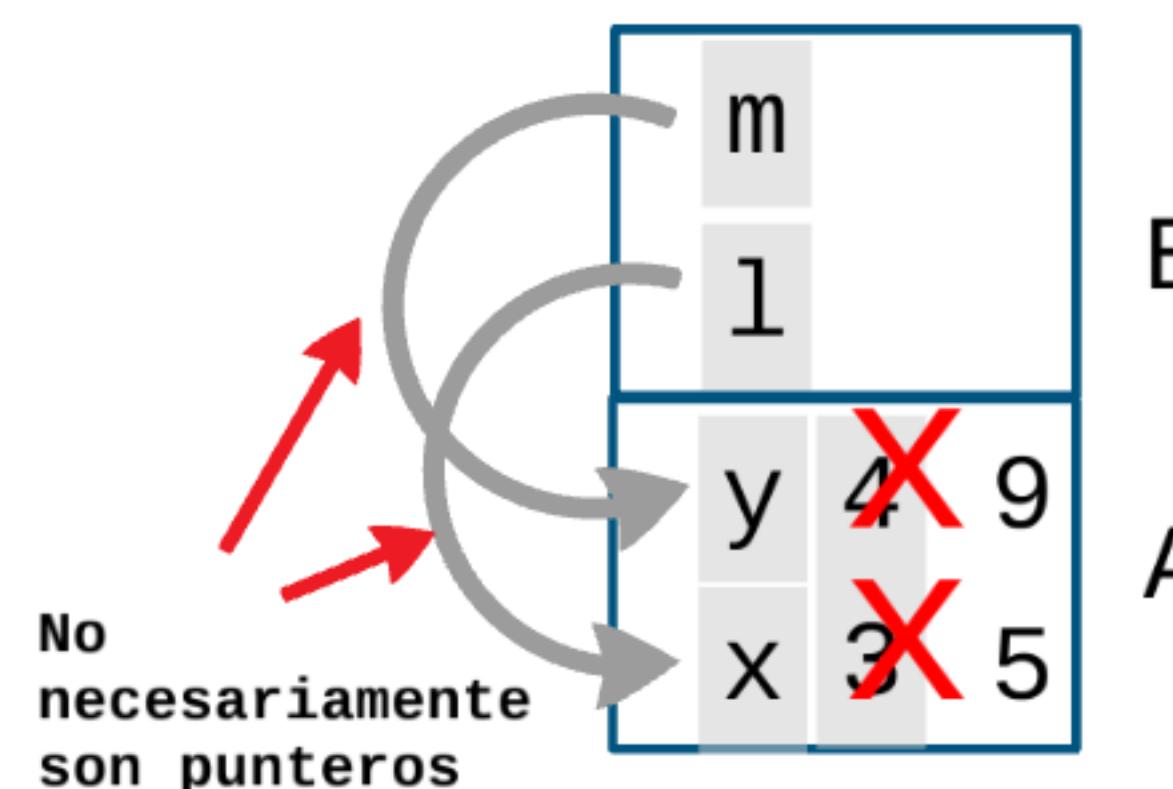
```
int a;
void f(int &x)
{
    ...
}
f(a);
```

Dentro de f se pueden usar x y a, pero se refieren a la misma celda de memoria.

# Referencia

```
procedure A;  
var x,y : integer;  
procedure B(l: integer;m:integer);  
begin  
    l := 5;  
    m := 9;  
end;  
begin  
    x :=3;  
    y := 4;  
    B(x,y);  
end;
```

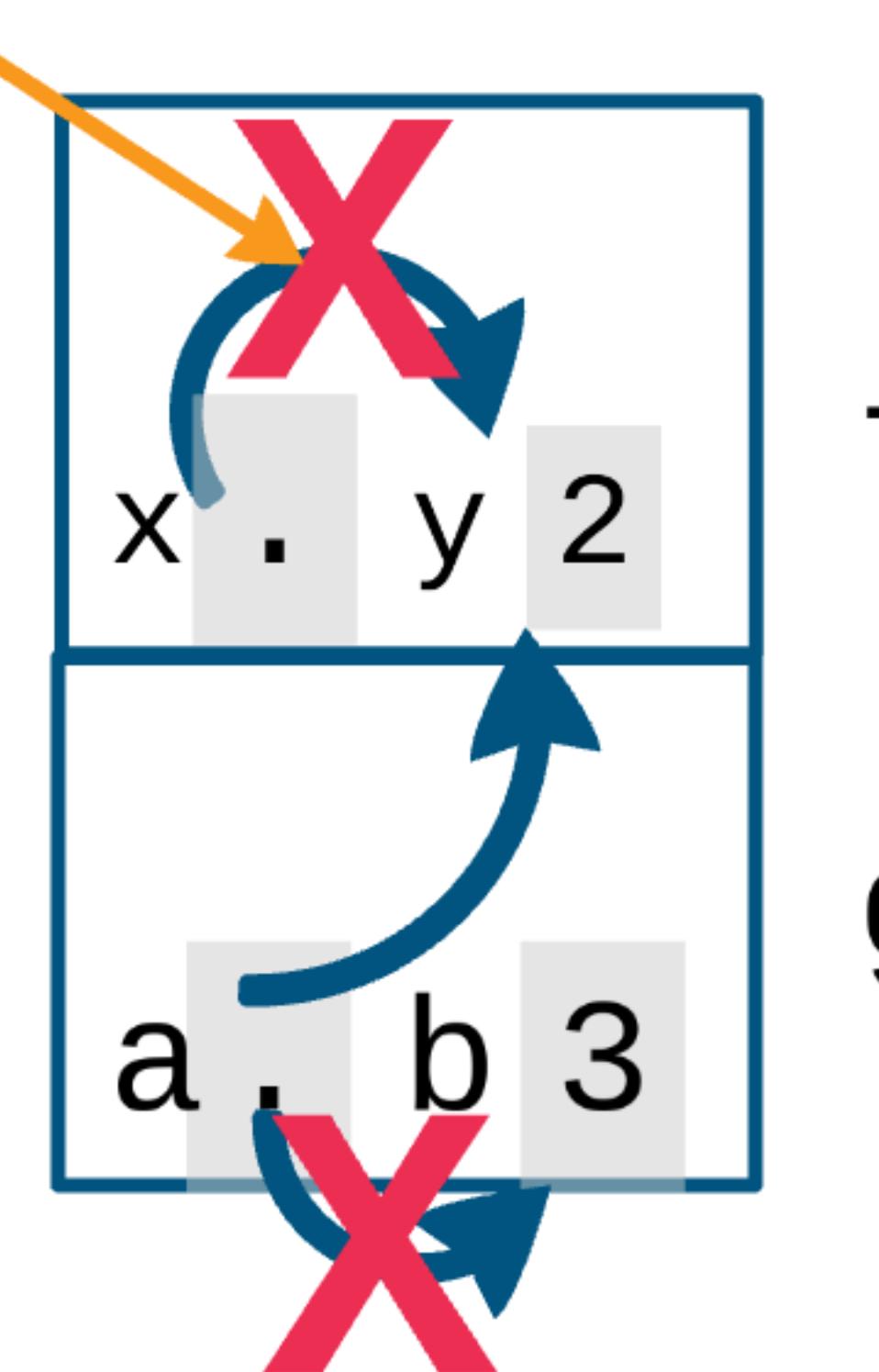
- Referencian la misma celda de memoria
- Las modificaciones sobre el parámetro formal se reflejan en el parámetro real



# Problemas de pasaje de parámetros como referencia

## Alcance y tiempo de vida de una variable

Esto no ocurre porque x es un alias de a



```
void f(int* &x)
{
    int y=2;
    x=&y;
}
void g()
{
    int b;
    int *a=&b;
    *a = 3;
    f(a);
    ...
    printf("%d", *a)
}
```

# Semántica de pasaje de parámetros por lenguaje

|                 | FORTRAN | C | C++ | Java    | ADA | C#   |
|-----------------|---------|---|-----|---------|-----|------|
| Referencia      | ✓       | ✗ | ✓   | objetos | ✓   | ✓    |
| Nombre          | ✗       | ✗ | ✗   | ✗       | ✗   | ✗    |
| Valor           | ✗       | ✓ | ✓   | ✓       | ✓   | Ref. |
| Resultado       | ✗       | ✗ | ✗   | ✗       | ✓   | Ref. |
| Valor-resultado | ✗       | ✗ | ✗   | ✗       | ✓   | ✗    |

# Contratos

Se utilizan para especificar lo que se espera o que se requiera que ocurra al ejecutar una unidad

Las semánticas de pasaje de parámetro son formas simples de contratos

Se materializan como expresiones de comparación de las cuales depende la ejecución de una unidad o bloque de código

Soporte nativo Eiffel, Kotlin, ADA 2012, C++ 2020, etc.

Algunas formas de contratos

- Pre-Condiciones
- Post-Condiciones
- Aserciones

¿Es posible verificar todos los contratos en tiempo de compilación?

¿Qué sucede con la ejecución de la función que no cumple el contrato?

ADA

with Ada.Text\_IO; use Ada.Text\_IO;

procedure F is

type Int\_8 is range -10..10;

function Suma1 (X : Int\_8) return Int\_8 is (X+1)

with

Pre => (X<9);

begin

Put\_Line ("Suma1 " & Int\_8'Image (Suma1 (1)));

Put\_Line ("Suma1 " & Int\_8'Image (Suma1 (9)));

end F;

Se pide una condición mas estricta que el tipo de X

# Pasaje de arreglos como parámetros

```
C  
f(int x[][]8) {  
    x[1][1]=0;  
}  
  
g(){  
int m[5][8];  
f(m)  
}
```

¿Cómo hace la función f para acceder a los elementos de la matriz x?

Dirección Base + i \*columnas + j

Se agrega el número de columnas como parámetro formal

Se puede implementar con aritmética de punteros, pasando las dimensiones como parámetros adicionales

Esto es desde la programación, no desde el lenguaje

Otra solución:

Las dimensiones son parte del tipo

ADA

```
type Mat_Type is array (Integer range <>,  
Integer range <>) of Float;  
Mat_1 : Mat_Type(1..100, 1..20);  
function Sumer(Mat : in Mat_Type) return  
Float is  
Sum : Float := 0.0;  
begin  
for Row in Mat'range(1) loop  
for Col in Mat'range(2) loop  
Sum := Sum + Mat(Row, Col);  
end loop; -- for Col . . .  
end loop; -- for Row . . .  
return Sum;  
end Sumer;
```

Extraido de pag. 412 Louden

Algo similar ocurre con Java y C#, en los que los arreglos son objetos y los límites son atributos

# Pasaje de funciones como parámetro

Solución para implementar procedimientos genéricos

```
Procedure CalculaIntegral(Procedure p)
begin
  end;
Procedure Calcular()
  Procedure ICirculo()
    begin
      ...
    end;
  Procedure IRectangulo()
    begin
      ...
    end
begin
  ...
  ReadLn(f : char);
  if f = 'c' then
  begin
    CalculaIntegral(ICirculoI())
  end
  if f = 'r' then
  begin
    CalculaIntegral(IRectangulo())
  end
end;
```

Recibe un procedimiento como parámetro

Estos procedimientos están definidos en el ámbito de Calcular()

Se pasan funciones como parámetro

# Pasaje de funciones como parámetro

```
int Q(int i,function R(int j))  
{int x;  
    x=4;  
    i=R(i);  
}  
void P()  
{int i;  
    int S(int b){ x=x+b;  
        return i+b; }  
    i=2;  
    Q(x,S);  
}  
x=7;  
P;
```

¿Cómo hace S para acceder a las variables no locales a ella?

¿Qué sucede con el chequeo del tipo de los parámetros?

# Pasaje de funciones como parámetro

## Chequeo de tipos de parámetros

C

```
double g ( double x) {...} ←  
double f ( double (*z)(double x), int j)  
{  
    ...  
}  
  
main()  
{  
    double s;  
    s = f(g,1); ←  
}
```

Se conocen los parámetros de g

Se define cantidad y tipo de parámetros

No hace falta especificar parámetros para g, porque ya se conocen

En Fortran 95, se especifican los tipos de los parámetros de funciones que son pasadas como parámetros

# Pasaje de funciones como parámetro

## Ámbito de la función pasada como parámetro

```
function sub1() {  
    var x;  
    function sub2() {  
        print(x);  
    };  
    function sub3() {  
        var x;  
        x = 3;  
        sub4(sub2);  
    };  
    function sub4(subx) {  
        var x;  
        x = 4;  
        subx();  
    };  
    x = 1;  
    sub3();  
};
```

¿Qué sucede si sub2() está anidada dentro de sub3()?

### ¿Qué imprime éste programa?

- 4 ámbito shallow
- 1 ámbito profundo
- 3 ámbito ad-hoc

### ¿Cuál variable x utiliza sub2()?

Ámbito shallow (superficial)

El ámbito es el del llamador  
(sub4)

Ámbito profundo

El ámbito es el de padre **Lenguajes con Alcance estático**  
(sub1)

Ámbito Ad-hoc

El ámbito es el de sub3

Sin ámbito global

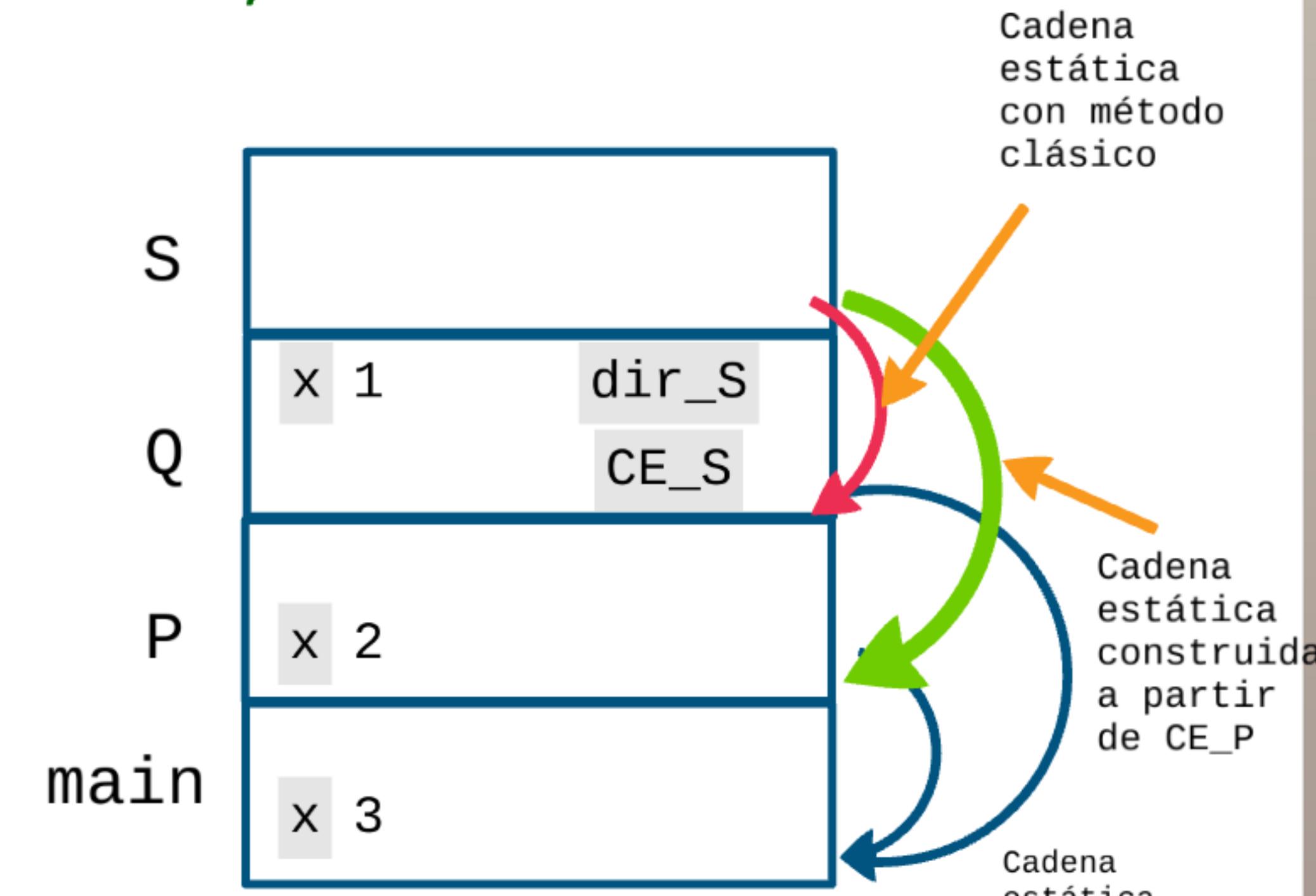
**Lenguajes con Alcance dinámico**

**Se usa en algunos lenguajes en ciertas situaciones**

# Pasaje de funciones como parámetro

## Registro de activación, ámbito Profundo

```
void main{
int Q(function R()){
    int x=1;
    R();
}
void P()
{
    int x=2;
    void S(){ x=4; }
    Q(S);
}
int x=3;
P;
}
```



Para pasar una función como parámetro, es necesario que la función llamadora pase el puntero de la dirección en el código ejecutable donde comienza la función pasada como parámetro (`dir_S`)

Para construir la cadena estática de S, es necesario que P le pase a Q su dirección (`CE_S`) y se coloque ese puntero como cadena estática en el R.A. de S (en el caso de puntero a la base del R.A.)

# Funciones como parámetro

R

```
dividir <- function(x, y, FUN =  
round, ...){  
  dividir <- FUN(x / y, ...)  
}  
  
dividir(3,4, FUN = floor, digits = 3)
```

Python

```
def x(a):  
    print a  
def y(z,t):  
    z(t)  
y(x,1)
```

C++

```
// Código extraido de https://www.quantstart.com/articles/Function-Objects-Functors-in-C-Part-1  
  
double add(double left, double right) {  
    return left + right;  
}  
double multiply(double left, double right) {  
    return left * right;  
}  
double binary_op(double left, double right, double (*f)(double, double)) {  
    return (*f)(left, right);  
}  
int main( ) {  
    double a = 5.0;  
    double b = 10.0;  
    std::cout << "Add: " << binary_op(a, b, add) << std::endl;  
    std::cout << "Multiply: " << binary_op(a, b, multiply) << std::endl;  
    return 0;  
}
```

GO

```
package main  
  
import "fmt"  
  
type fn func(int)  
var j int = 1  
  
func f1(i int) {  
    fmt.Printf("\ni %v", i)  
    fmt.Printf("\nj %v", j)  
}  
  
func test(f fn, val int) {  
    var j int = 5  
    fmt.Printf("\nj %v", j)  
    f(val)  
}  
  
func main() {  
    var j int = 4  
    fmt.Printf("\nj %v", j)  
    test(f1,1)  
}
```

¿Qué tipo de ámbito está utilizando este lenguaje para el pasaje de funciones como parámetro?

# Closures

## ¿Por qué estudiar los closures?

- Aportan características funcionales al lenguaje
- Permite igualar las funciones con las variables (first class citizens: asignación, pasaje como parámetro, atributo de un objeto o componente de estructura, valor de retorno de una función o método)
- Proporcionan mayor flexibilidad a las funciones
- Permiten que un objeto retorne un método sin necesidad de dar acceso a sus propios atributos
- Permite implementar funciones generadoras de funciones

## Lenguajes que tienen closures

ML, Lisp, Perl, Ruby, Python, Javascript, PHP, C++14, Java8+, C#, Rust, Go, entre otros.

# Closures

- Función evaluada o ejecutada en un ámbito diferente al cual fue definida.
- Puede hacer uso de variables no presentes en su ámbito de evaluación.
- El lenguaje debe proveer un mecanismo para que la función acceda a las variables del ámbito correspondiente.

```
fn function f1() {  
    int x;  
    fn function f2(){  
        print x;  
    }  
    return f2;  
};  
int main() {  
z=f1();  
z();  
}
```

Si no existiese este mecanismo,  
las variables no estarían en  
memoria!

¿Cómo hace la función f2  
para acceder a la variable  
x cuando se ejecute sin que  
el R.A. de f1 esté en  
memoria?

Desde main se está invocando  
indirectamente a f2.

# Closures

R

```
stat <- function(stat_name){  
  function(x){  
    stat_name(x)  
  }  
}  
  
mean_of_x <- stat(mean)  
sd_of_x <- stat(sd)  
mean_of_x(x)  
sd_of_x(x)
```

Rust

```
fn main() {  
let mut num = 5;  
{  
  let mut add_num = move |x: i32| num+= x;  
  
  add_num(5);  
}  
println!("num: {}", num);  
}
```

Javascript

```
function fn() {  
var x = 3;  
var lambdaFun = () => x+1;  
x++;  
console.log(lambdaFun()); // Imprime 5  
}
```

GO

```
package main  
import "fmt"  
var k int = 1  
  
func f() func() {  
  var i int = 3  
  return func() {  
    var j int = 5  
    fmt.Printf("", i+j+k);  
    i = i + 1;  
    j = j + 1;  
    k = k + 1;  
  }  
}  
  
func main() {  
  a := f();  
  a();  
  a();  
  b := f();  
  a();  
  b();  
}
```

¿Qué imprime este programa?

# Closures

¿Cómo se resuelve el ámbito de los closures en lenguajes de tipo algol?

- Por copia
- Por referencia
- En tiempo de compilación el lenguaje resuelve la captura de variables

```
fn function f1() {  
    int x;  
    fn function f2(){  
        print x;  
    }  
    return f2;  
};  
int main() {  
z=f1();  
z();  
}
```

¿Qué sucede cuando se capturan por referencia?

¿Dónde pueden estar las copias de las variables capturadas?

- R.A. Función llamadora del closure
- Heap
- Persistencia del R.A. original

# Ubicación de variables en Closures

## Rust

```
fn main(){
let mut a = 5;

{
    let mut closure1 = move |x: i32| {
        a += x;
        println!("direccion de a adentro de
closure: {:?}", &a);
    };
    closure1(5);
}
println!("{}", a);
println!("direccion de a en main: {:?}", &a);
}
```

Especificación  
de captura de  
variables

## C++

```
#include <stdio.h>
// -std=c++14
static int y=1;
auto creadora()
{
    int x=2;
    printf("&x en creadora: %p \n", &x);
    auto func = [=] () {
        int z= 3;
        printf("x+y+z: %d \n", x+y+z);
        printf("x en closure: %d \n", x);
        printf("y en closure: %d \n", y);
        printf("z en closure: %d \n", z);
        printf("&x en closure: %p \n", &x);
        printf("&y en closure: %p \n", &y);
        printf("&z en closure: %p \n", &z);
    };
    // func();
    return func;
}

int main()
{
// y=8;
    int w = 4;
    auto z=creadora();
    z();
    printf("&y en main: %p \n", &y);
}
```

Especificación  
de captura de  
variables

¿Qué sucede si el closure  
captura sus variables por  
referencia?

# Tratamiento de funciones

Tercera clase:

Las funciones  
pueden invocarse  
pero no pasarse  
ni retornarse  
como parámetros

+

Segunda clase:

Las funciones  
pueden pasarse  
como parámetros

+

Primera clase:

Las funciones  
pueden asignarse a  
una variable y  
retornarse como  
parámetros

# Problemas a resolver en el tratamiento de funciones

Tercera clase:

Invocación y  
Pasaje de  
parámetros

+

Segunda clase:

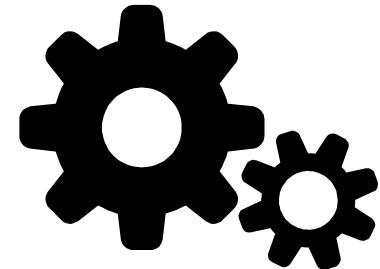
Pasar como parámetro  
el ámbito original de  
la función

+

Primera clase:

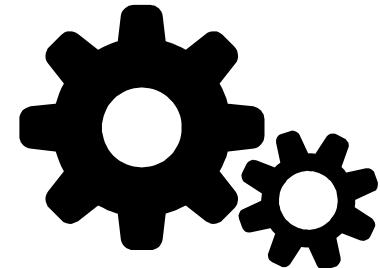
Capturar el ámbito que  
necesita la función luego  
de su retorno

**Lenguajes de Programación I**  
**Facultad de Ciencias Exactas**  
**UNICEN**



# **Clase Teórico - Práctica**

# **Concurrencia**



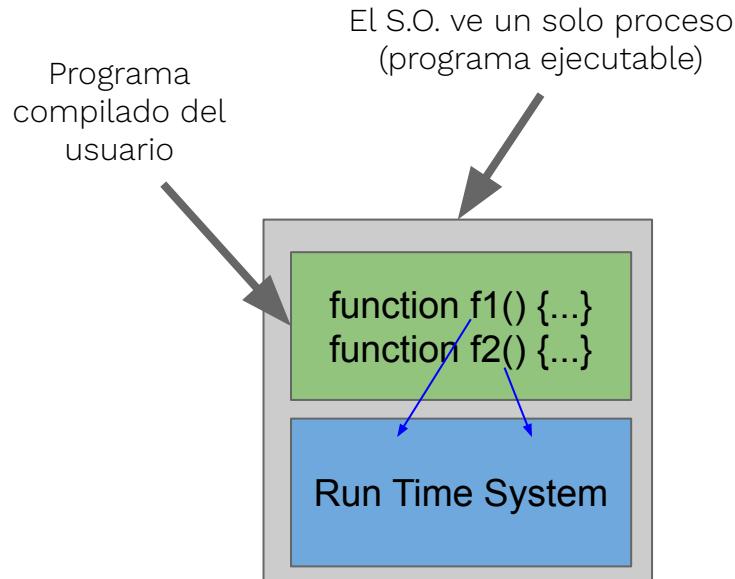
- **Concurrencia en Lenguajes dinámicos**
- **Problemas de Concurrencia**
- **Concurrencia interna en Lenguajes tipo Algol (Go)**
- **Hilos de ejecución en Lenguajes tipo Algol (C++ y Java)**
- **Semáforos (C++)**
- **Canales (Go)**

# Concurrencia interna

El S.O. no sabe que existe esta concurrencia (Unidades concurrentes: Co-Rutinas de FORTRAN, Funciones concurrentes de Concurrent-C, Pascal Concurrente y otros, Go-Rutinas de GO, etc.)

## Lenguajes Compilados

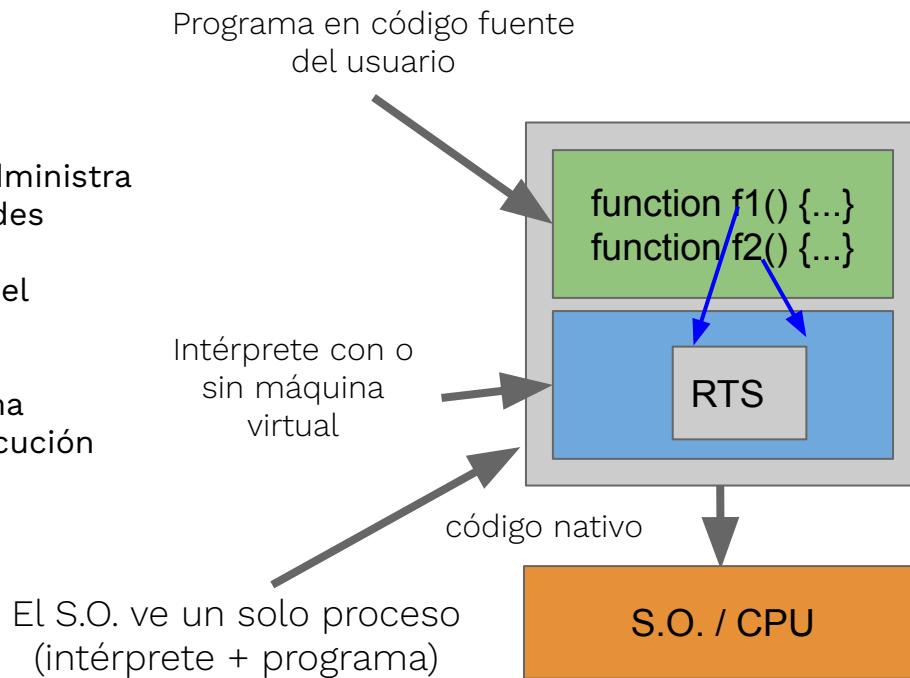
- El Run Time System del Lenguaje (RTS) administra las prioridades y la memoria de las unidades concurrentes.
- El RTS se encuentra compilado junto con el programa ejecutable del usuario.
- Para el S.O. hay un único proceso.
- En el tiempo que el S.O. asigna a nuestro programa, el RTS deberá planificar la ejecución de las unidades concurrentes.



# Concurrencia interna

## Lenguajes Interpretados

- El Run Time System del Lenguaje (RTS) administra las prioridades y la memoria de las unidades concurrentes.
- El RTS se encuentra compilado junto con el intérprete.
- Para el S.O. hay un único proceso.
- En el tiempo que el S.O. asigna al programa ejecutable, el RTS deberá planificar la ejecución de las unidades concurrentes.



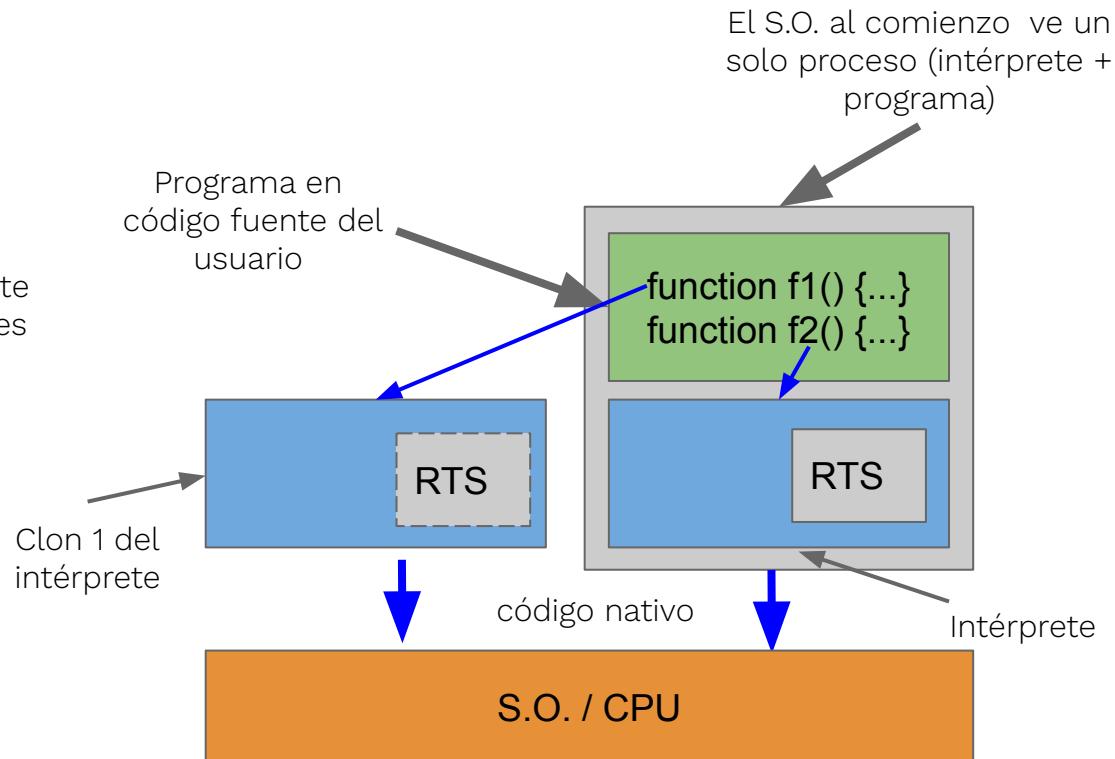
# Concurrencia conjunta en lenguajes interpretados

## Lenguajes Interpretados sin Máquina Virtual

(El intérprete se ejecuta directamente sobre el S.O., y genera instrucciones específicas para la plataforma)

Una parte del intérprete se clona y convierte en un hilo de ejecución

El S.O. ve varios hilos de ejecución (clones del intérprete)



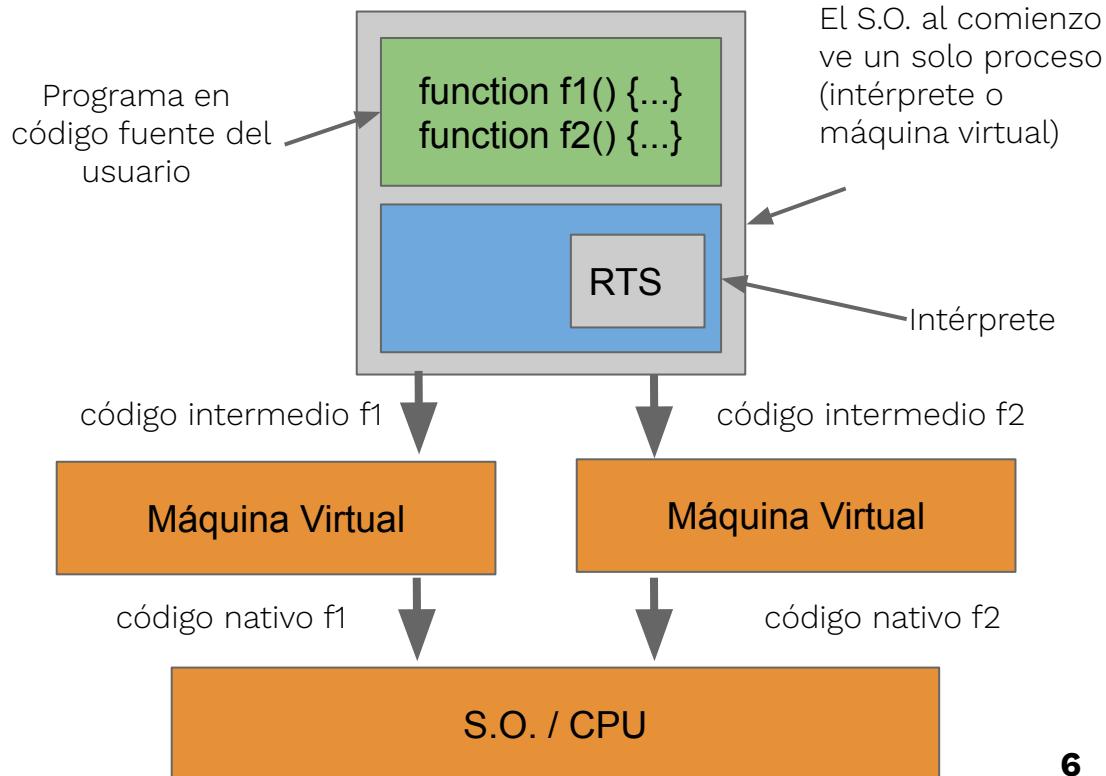
# Concurrencia conjunta en lenguajes interpretados

## Lenguajes Interpretados con Máquina Virtual

(El intérprete genera instrucciones para una máquina virtual y ésta se ejecuta sobre el S.O.)

Una parte de la máquina virtual se clona y se convierte en un hilo.

El S.O. ve varios hilos de ejecución (construidos por la máquina virtual a partir del código intermedio)



# Manejo de Memoria en concurrencia conjunta en lenguajes interpretados

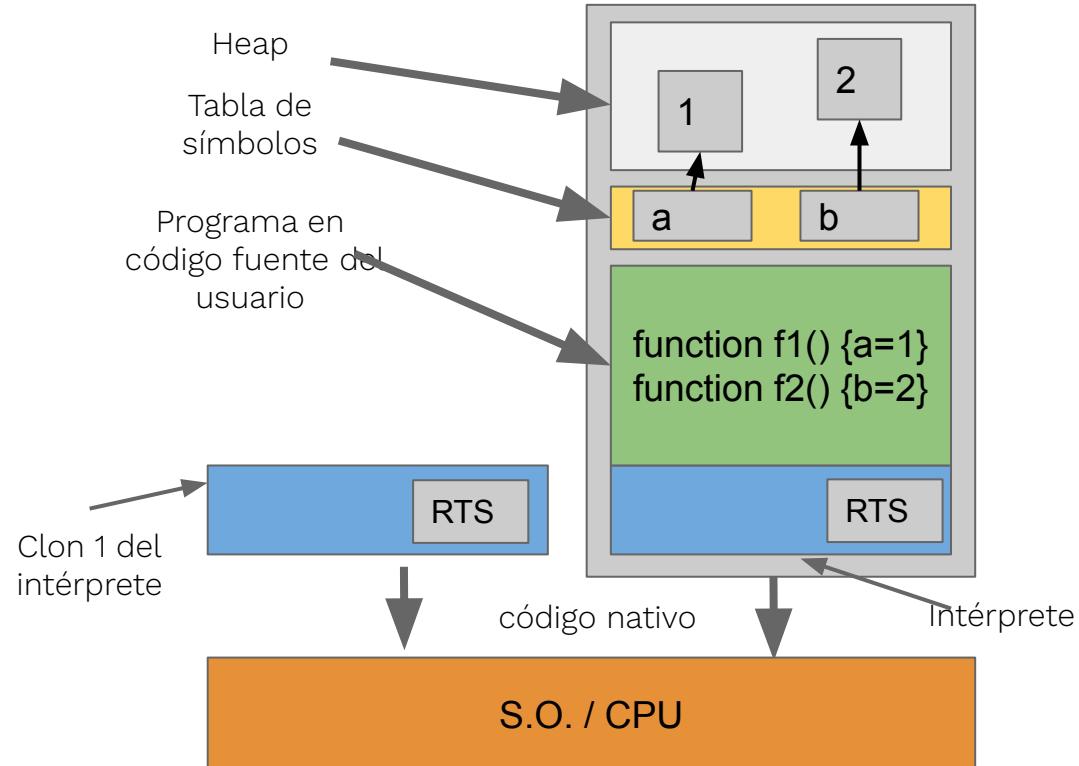
## Con/sin Máquina Virtual

Concurrencia interna o conjunta

La Tabla de símbolos y el Heap se comparten entre todos los hilos

El intérprete/MV administra el acceso a la memoria y/o ofrece mecanismos al programador

Similitud con los lenguajes compilados



# Concurrencia en Python

## Python

```
import threading  
import time  
import random  
a = 0  
def f(id):  
    global a  
    for i in range(1, 10):  
        time.sleep(random.random()*4)  
        print ("id",id)  
        print (a)  
        a=a+1  
t1 = threading.Thread(target=f, args = (1,))  
t2 = threading.Thread(target=f, args = (2,))  
t1.start()  
t2.start()  
t1.join()  
t2.join()
```

variable global a

función que se ejecuta concurrentemente

Creación de hilos desde el programa principal

Ejecución de hilos desde el programa principal

Vinculación de cada hilo al hilo principal

**¿Cuántos hilos ve el Sistema Operativo?**

# Semáforos en Python

```
import threading  
x = "NO"  
CANT = 1000
```

```
def f1(s):  
    global x, CANT  
    for i in range(CANT):  
        s.acquire()  
        x = "SI"  
        print (x)  
        print ("f1")  
        s.release()  
...
```

Se bloquea el semáforo

Se libera el semáforo

```
...  
def f2(s):  
    global x, CANT  
    for i in range(CANT):  
        s.acquire()  
        x = "NO"  
        print (x)  
        print ("f2")  
        s.release()  
  
s = threading.Lock()  
t1 = threading.Thread(target=f1, args = (s,))  
t2 = threading.Thread(target=f2, args = (s,))  
t1.start()  
t2.start()  
t1.join()  
t2.join()
```

# Concurrencia en Perl

## Perl

```
#!/usr/bin/perl
use threads;
use threads::shared;
#my $a : shared = 0;
my $a = 0;
sub sub1 {
    for ($i=0;$i<100;$i++)
    {
        $a = $a + 1;
        sleep(rand()*0.2);
        printf("Hilo: %2d,a= %2d\n",threads->self->tid,$a);
    }
}
$th1 = threads->new(\&sub1);
$th2 = threads->new(\&sub1);
print "Hilo principal\n";
$th1->join();
$th2->join();
```

variable global a

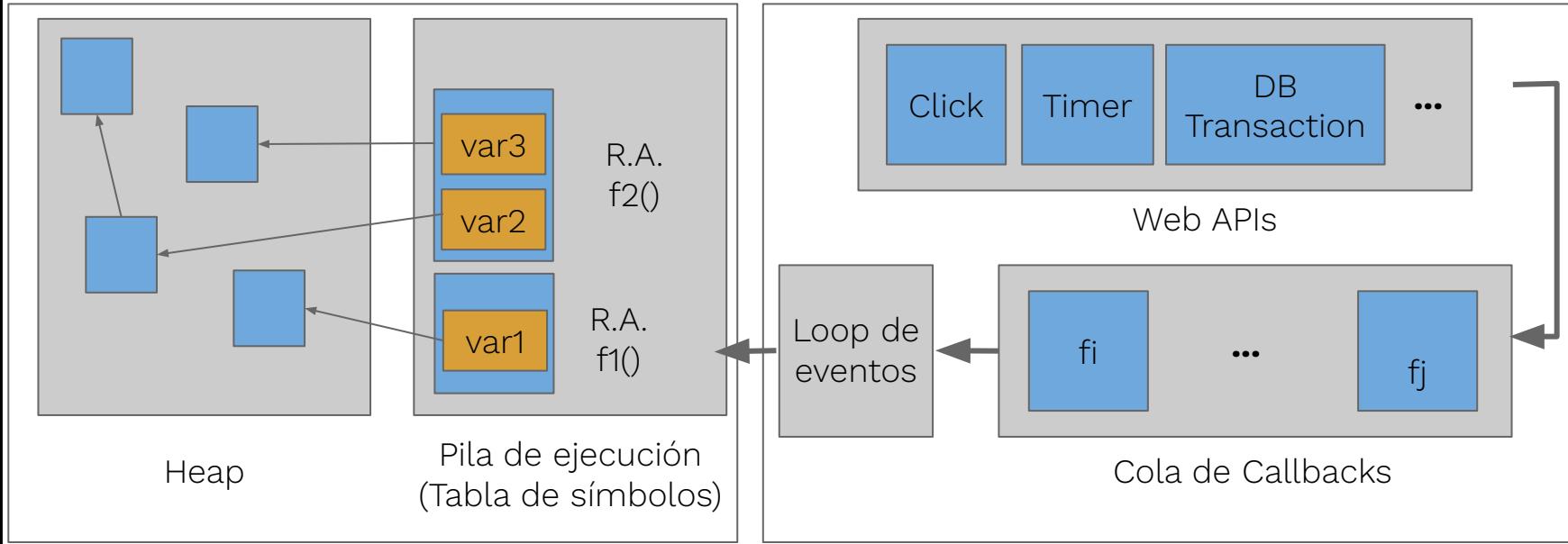
función que se ejecuta concurrentemente

Creación y ejecución de hilos desde el  
programa principal

Vinculación de cada hilo al hilo principal

**¿Cuántos hilos ve el Sistema Operativo?**

# Concurrencia en Javascript



**El intérprete se ejecuta en un solo hilo!**

# Concurrencia en Javascript

```
$ .on('button', 'click', function onClick() {  
    setTimeout(function timer() {  
        console.log('Usted hizo click en el  
botón');  
    }, 2000);  
});  
  
console.log("Hola");  
  
setTimeout(function timeout() {  
    console.log("Por favor, haga click en el  
botón");  
}, 5000);  
  
console.log("Bienvenido");
```

Funciones Callbacks

Función enviada como parámetro

- La Concurrencia se realiza de forma conjunta.
- El navegador organiza la ejecución de las funciones asíncronas a través de la cola de callbacks.
- El Loop de eventos administra el momento exacto de su ejecución

# Concurrencia en Javascript

```
let Promise1 = new Promise((exito) =>
{
    setTimeout(function() {
        exito("Se cumplió el
temporizador");
    }, 1000);
});

for(i=0;i<100000;i++);
Promise1.then((msj) => {
    console.log("Bienvenido, " + msj);
});
console.log("Hola");
```

Función que será llamada en caso de cumplirse la Promesa

Variable/Objeto Promise

Llamada a Web API enviando una función anónima que contiene una llamada a la función exito

Se invoca a la promesa enviando una función anónima correspondiente al parámetro exito

Además Javascript provee construcciones **async, await, etc.**

# Seguridad en concurrencia

- Thread-Safety=Cada hilo se ejecuta e interactúa con los demás sin la posibilidad que se produzcan resultados inesperados en la ejecución.

**Problema:** Cada hilo puede poseer su propia pila o Tabla de Símbolos pero estos datos pueden ser accedidos por otros hilos en virtud de punteros, anidamiento, etc, además, comparte el mismo espacio de direccionamiento que los demás para el heap y variables estáticas.

## Algunas acciones que pueden provocar problemas:

- Lecturas concurrentes
- Escrituras concurrentes
- Condiciones de carrera

**Terminología:** Memory Safety, Thread-safety, Conditionally Thread-Safety, Not Thread-Safe, etc.

## ¿Qué ofrecen los lenguajes para minimizar resultados inesperados?

- Re-entrancy, Local Thread Storage, Immutable variables, Mutual Exclusion, Atomic Operations, etc..

# **Global Interpreter Lock**

Mecanismo de Exclusión Mútua que evita la creación de hilos para funciones concurrentes.

## **Causas:**

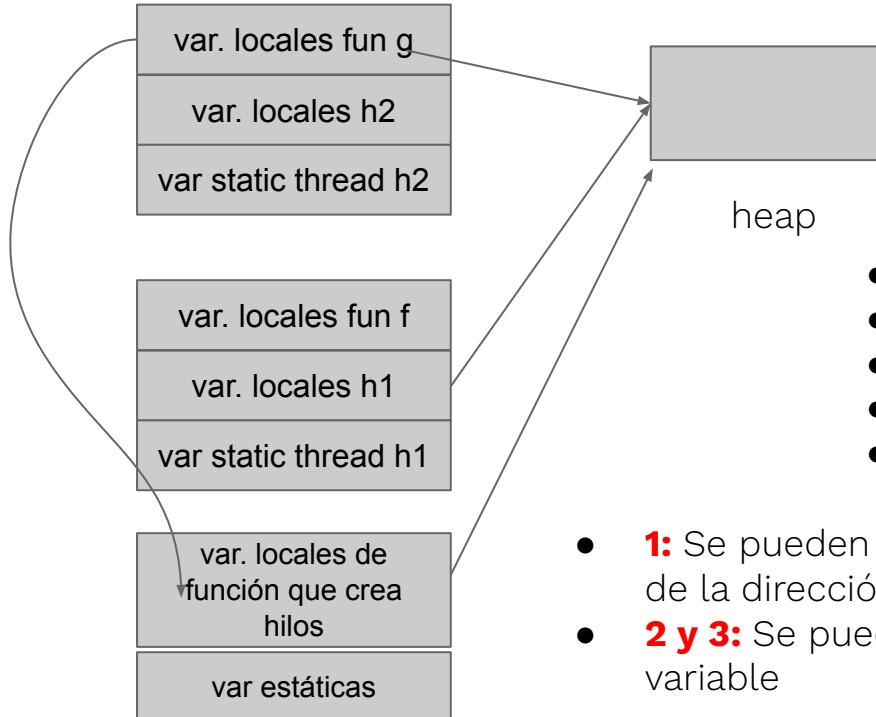
- En un lenguaje interpretado las variables no referencian directamente a una celda de memoria, sino que se accede a la información a través de su símbolo. Es posible acceder a una variable global desde dos funciones concurrentes.
- El intérprete no puede asegurar que una variable pueda ser accedida sólo por un hilo de ejecución.

## **Intérpretes con GIL**

- CPython (intérprete y máquina virtual oficial de Python)
- MRI (Intérprete y máquina virtual oficial de Ruby)
- V8 (Intérprete y máquina virtual de JavaScript)

## **Existen otros intérpretes sin GIL**

# Problemas de Conurrencia en Lenguajes Tipo Algol



Los hilos pueden acceder simultáneamente a variables compartidas

## ¿Cuáles son las variables problemáticas?

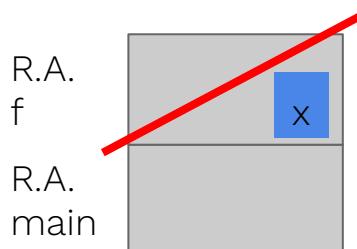
- Variables globales **\*1**
  - Variables locales de la pila del hilo
  - Variables no locales de la pila del hilo
  - Variables ubicadas en el heap **\*2**
  - Variables de la pila del que creó el hilo **\*3**
- 
- **1:** Se pueden acceder por el alcance del lenguaje (a través de la dirección)
  - **2 y 3:** Se pueden acceder si el hilo tiene un puntero a la variable

# Repaso de Pila - C++

C++

## ¿Qué imprime éste programa?

```
#include <iostream>
using namespace std;
void f() {
    int x;
    cout << &x << endl;
}
int main() {
    for (int i=0;i<10;i++)
        f();
}
```

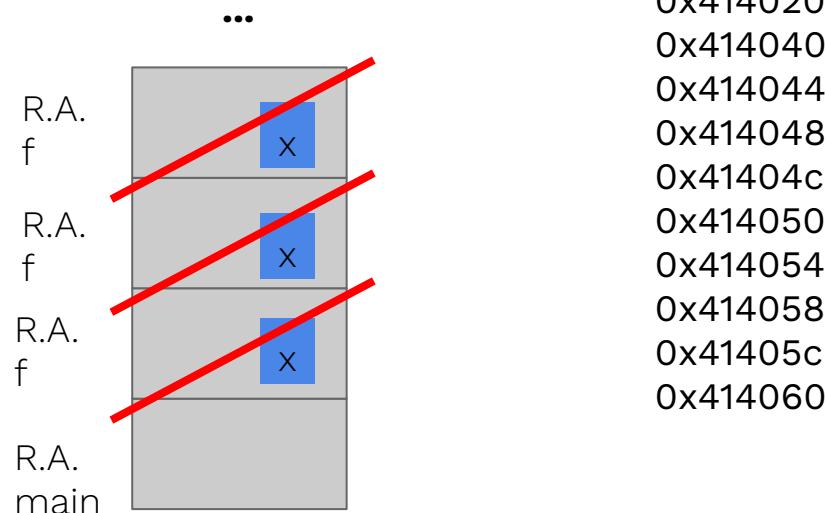


# Concurrencia en GO

```
package main
import "fmt"

func f() {
    var x int
    x = 3
    fmt.Println(&x)
}

func main() {
    var i int
    i = 0
    for i < 10 {
        f()
        i = i + 1
    }
}
```



# Concurrencia en GO

## ¿Por qué se justifica un modelo de pila de ese tipo?

- En GO no es necesario disponer de soluciones adicionales para la concurrencia o para la captura de variables en los closures. Las variables permanecen en los registros de activación indefinidamente o hasta que una función necesita memoria y el lenguaje le asigna la misma porción.
- **Go Routines:**
  - son funciones convencionales ejecutadas asincrónicamente,
  - son tareas al nivel del lenguaje. El S.O. no sabe que existen

# Concurrencia en GO

## ■ **Ventajas sobre los Threads:**

- No es necesario realizar context switch,
- no existe un límite impuesto por el S.O.

## ■ **Desventajas sobre los Threads:**

- A nivel S.O. es posible ejecutar tareas de baja prioridad por sobre las de alta prioridad.

# GO Routines

```
package main
import "fmt"
import "time"

func f() {
    var x int
    x = 3
    fmt.Println(&x)

}

func main() {
    var i int
    i = 0
    for i < 10 {
        go f()
        i = i + 1
    }
    time.Sleep(5 * time.Second)
}
```



# GO Canales

- Facilitan la comunicación entre las Go Routines
- Proporcionan comunicación segura sin pérdida de información (buffers)
- Mientras una goroutine utiliza el canal
  - El Run Time System de Go bloquea la ejecución del resto de las goroutines que utilizan el mismo canal.

# GO Canales

- Esto sucede de forma automática sin necesidad de comunicarle al sistema operativo la existencia de un recurso compartido a diferencia de Java, C++ y otros.
- Se puede dar una situación de deadlock si una go routine entra en un loop y el resto de las goroutines utilizan el mismo canal.

# GO Canales

```
package main
import "fmt"

func f(c chan int) {
    c <- 1
}
func g(c chan int) {
    c <- 2
}
func main() {
    canal := make(chan int)
    var i int
    i = 0
    for i<10 {
        go f(canal)
        go g(canal)
        i = i + 1
        valor := <-canal
        fmt.Println(valor)
    }
}
```

# **Ejecución de Ejercicios y Ejemplos**

Prueba de Ejercicios

# Para Probar los Ejercicios

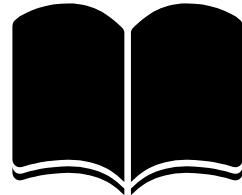
- Emplear Compiladores e intérpretes en línea o Nativos.
- Compilador de GO muy sencillo de instalar
- Intérpretes de Python y Perl disponibles en Linux.
- G++



# **Asignaciones**

**Lenguajes de Programación I**

# Asignaciones



- Variables
- Sintaxis
- Semántica
- Copia y Referencia
- Ejemplos
- Asignaciones unarias
- Asignaciones múltiples
- Semejanzas con pasaje de parámetros

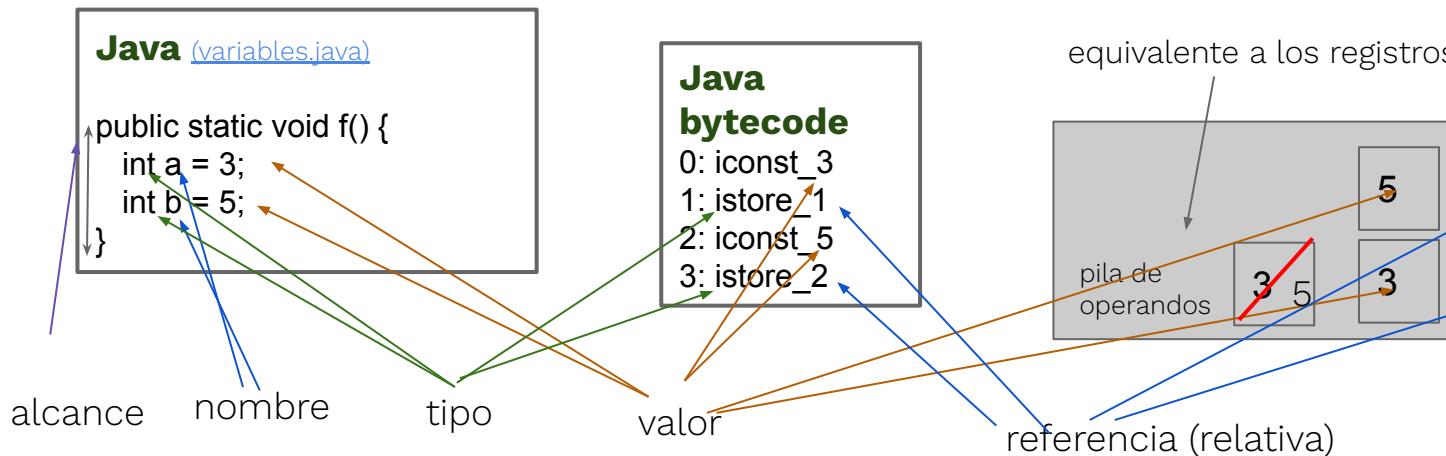
# Variables en lenguajes imperativos

## Tupla de propiedades

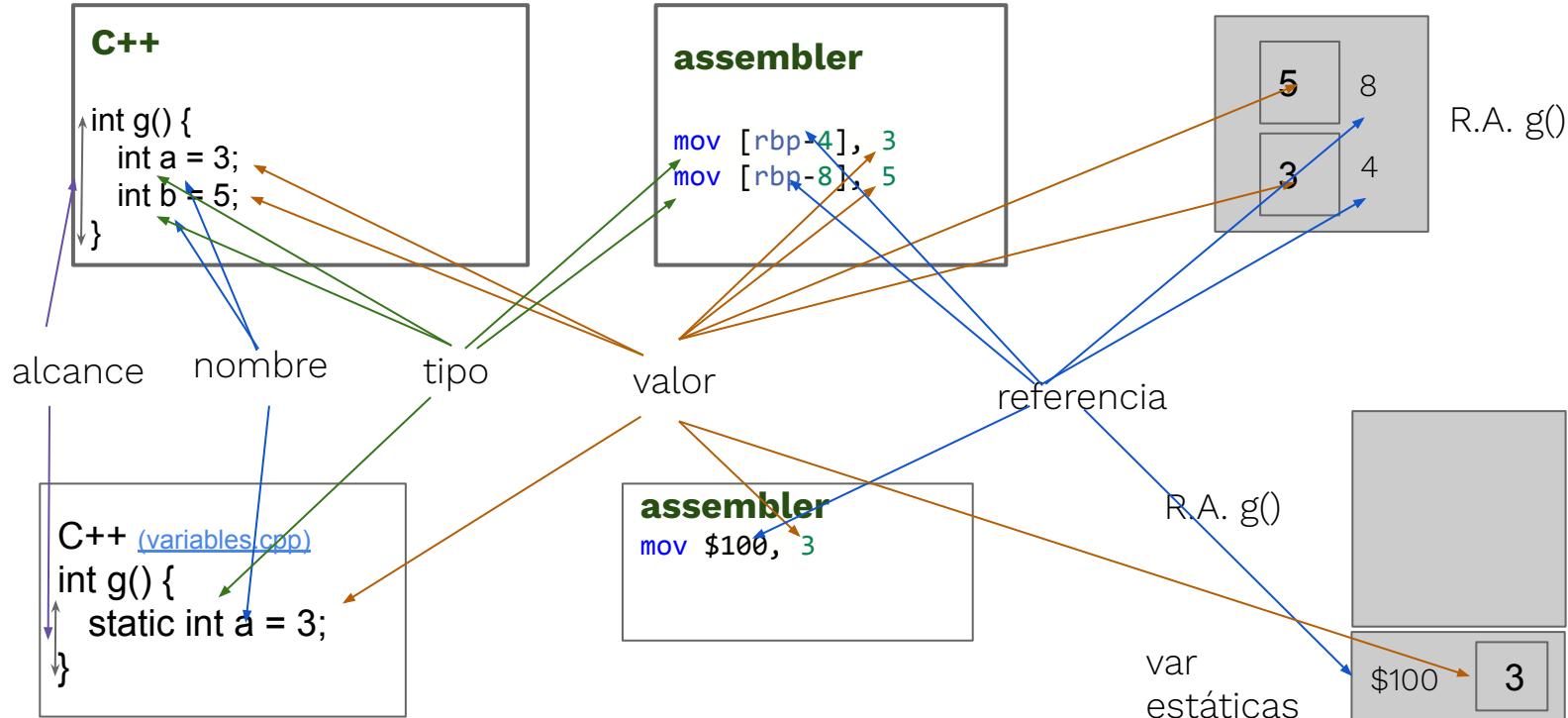
- Valor (r-value)
- Referencia (l-value)
- Nombre
- Tipo
- Alcance

Cada uno tiene su ligadura  
(estática o dinámica)

- ¿Tienen que existir todos?
- ¿Pueden cambiar antes de la ejecución?
- ¿Pueden cambiar durante la ejecución?



# Variables en lenguajes imperativos



# Sintaxis de asignaciones

| Sintaxis           | Lenguajes   |
|--------------------|---|
| <b>a = b</b>       | Java, C, C++, Perl, FORTRAN, Go, Swift, Rust, Dart, PHP, Hack, etc. |
| <b>a := b</b>      | Pascal, Delphi, Ada, ALGOL, Go (inferencia)                         |
| <b>a &lt;- b</b>   | APL, Smalltalk  |
| <b>(setq a b)</b>  | Lisp  |
| <b>MOVE B TO A</b> | COBOL   |
| <b>set a b</b>     | Tcl   |
| <b>a =: b</b>      | J   |
| <b>LET a = b</b>   | Basic   |

# Tipos de asignaciones

## Tupla de propiedades

- Valor (r-value)
- Referencia (l-value)
- Nombre
- Tipo
- Alcance

## Asignación de Valor

Se accede a la celda de memoria que referencia esa variable (l-value)

$$a = b$$

**Una asignación significa transferir al menos una de las cinco propiedades de una variable a otra**

Si el lado derecho está asociado a una celda de memoria

→ **Se accede a la celda de memoria y se extrae el valor de la variable (dereferencing)**

Si el lado derecho no está asociado a una celda de memoria (constantes literales)

$$a = 3$$

→ **Se utiliza el valor**

# Asignación por valor

## Tupla de propiedades

- Valor (r-value)
- Referencia (l-value)
- Nombre
- Tipo
- Alcance

- variables (tipos básicos, escalares o compuestos, estructuras, objetos)
- constantes (~~literales~~ o simbólicas)
- ~~expresiones~~
- ~~llamadas a funciones~~
- ~~dirección de memoria~~
- parámetros formales
- closures
- ~~expresiones lambda~~
- etc.

**a = b** →

- variables (tipos básicos, escalares o compuestos, estructuras, objetos)
- constantes (literales o simbólicas)
- expresiones
- llamadas a funciones
- dirección de memoria
- parámetros formales
- closures
- expresiones lambda
- etc.

↑  
Todo lo que tenga valor

↓  
Todo lo que tenga referencia

# Asignación por referencia

## Tupla de propiedades

- Valor (r-value)
- Referencia (l-value)
- Nombre
- Tipo
- Alcance

- variables (tipos básicos, escalares o compuestos, estructuras, objetos)
- constantes (~~literales~~ o simbólicas)
- ~~expresiones~~
- ~~llamadas a funciones~~
- parámetros formales (\*)
- closure
- ~~expresiones lambda~~
- etc.

**a = b** →

- variables (tipos básicos, escalares o compuestos, estructuras, objetos)
- constantes (~~literales~~ o simbólicas)
- ~~expresiones~~
- ~~llamadas a funciones~~
- parámetros formales (\*)
- closure
- ~~expresiones lambda~~
- etc.



Todo lo que tenga referencia



Todo lo que tenga referencia

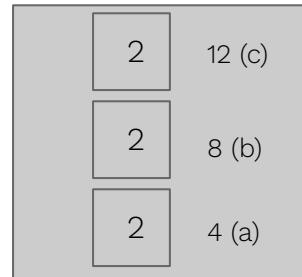
# Efectos de la asignación por valor

## Go ([asignacion.go](#))

```
func main() {  
    var a int = 2  
    var b int  
    b = a  
    c := a  
}
```

Las variables se mantienen independientes

reg.  
activación  
main



Se declara implicitamente y se infiere el tipo de c por el de a

**Prácticamente todos los lenguajes tienen asignación por valor para las variables de tipos básicos.**

**¿Qué sucede con las variables de tipos estructurados (arreglos, estructuras, registros)?**

[asignacionarrays.cpp](#)

**¿Qué sucede con las variables cuyo tipo es una clase?**

# Efectos de la asignación por valor

## Java ([asignacion.java](#))

```
public static void main(String[] args)
{
    int a = 3;
    int b = a;
    a = 4;
    System.out.println(b);
}
```

Reg. activación main()



## Java bytecode

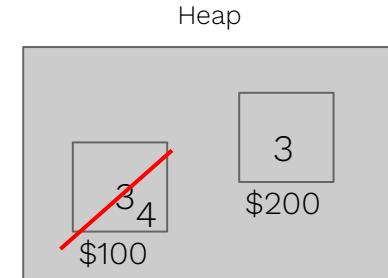
```
0:  iconst_3
1:  istore_1
2:  iload_1
3:  istore_2
```

Las dos variables se  
mantienen  
independientes

# Asignaciones en Java entre objetos

## Java [\(asignacionobj.java\)](#)

```
class Entero {  
    public int valor=3;  
}  
  
public class asignacionobj  
{  
    public static void main(String[] args)  
    {  
        Entero a = new Entero();  
        Entero b = new Entero();  
        b = a;  
        a.valor = 4;  
        System.out.println(b.valor);  
    }  
}
```



Reg. activación main()



Las dos variables  
referencian a la misma celda  
de memoria

## Java bytecode

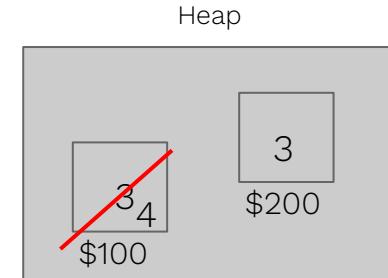
```
0: new      #2  
3: dup  
4: invokespecial #3  
7: astore_1  
8: new      #2  
11: dup  
12: invokespecial #3  
15: astore_2  
16: aload_1  
17: astore_2  
18: aload_1  
19: iconst_4  
20: putfield  #4
```

La asignación del objeto es  
por referencia

# Asignaciones en Java entre objetos

## Java ([asignacionobj.java](#))

```
class Entero {  
    public int valor=3;  
}  
  
public class asignacionobj  
{  
    public static void main(String[] args)  
    {  
        Entero a = new Entero();  
        Entero b = new Entero();  
        b = a;  
        a.valor = 4;  
        System.out.println(b.valor);  
    }  
}
```



Reg. activación main()



Las dos variables  
referencian a la misma celda  
de memoria

- Si consideramos la celda asociada a la variable, la asignación es por copia
- Si consideramos la celda apuntada por la variable (objeto), la asignación es por referencia.

¿Cuál es la diferencia con el primer ejemplo en java?

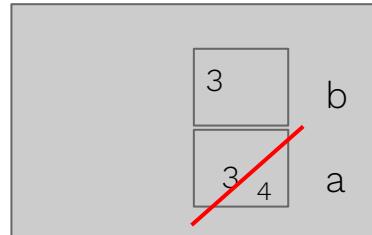
¿Qué es lo que está  
ocultando la sintaxis?

# Asignaciones en C++ entre objetos

C++ ([asignacionobj.cpp](#))

```
class Entero {  
    public: int valor=3;  
}  
  
int main() {  
    Entero a;  
    Entero b;  
    b = a;  
    a.valor = 4;  
    printf("%d.%d",b.valor);  
}
```

Reg. activación main()



Las dos variables  
(objetos) se mantienen  
independientes

- El objeto está en la pila
- Se realiza una extracción de valor de un objeto y se copia en el otro
- La asignación es por copia.

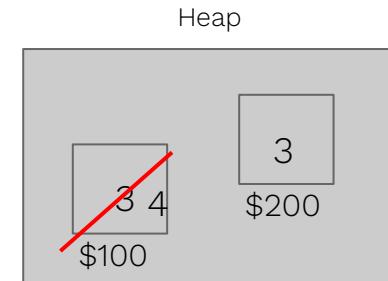
¿Por qué en uno de los lenguajes la asignación es por copia mientras que en el otro es por referencia?

¿Qué se podría hacer en C++ para emular el funcionamiento de Java?

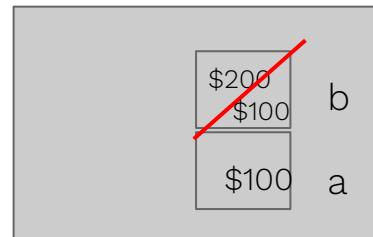
# Asignaciones en C++ entre punteros a objetos

## C++ ([asignacionobjptr.cpp](#))

```
class Entero {  
    public: int valor=3;  
}  
  
public class asignacionobj  
{  
    public static void main(String[] args)  
    {  
        Entero *a = new Entero();  
        Entero *b = new Entero();  
        b = a;  
        a->valor = 4;  
        printf("%d->valor",b->valor);  
    }  
}
```



Reg. activación main()



Las dos variables  
referencian a la misma celda  
de memoria

- La asignación de las variables semi-estáticas es por copia.
- La asignación de las variables dinámicas anónimas es por referencia.

¿Cuál es la diferencia con el ejemplo anterior en java?

- En C++ no hay ambigüedad porque el programador sabe explícitamente que “a” y “b” son punteros.
- En Java semánticamente son punteros pero sintácticamente no.

# Asignación por copia y por referencia

## Copia

- Se realiza una extracción de valor de la celda de memoria del lado derecho.
- Ese valor extraído es copiado en la celda de memoria referenciada por el lado izquierdo.

## Referencia

- Se realiza una supresión de la extracción del valor del lado derecho y se obtiene una dirección.
- Esa dirección es copiada en la celda de memoria referenciada por el lado izquierdo.

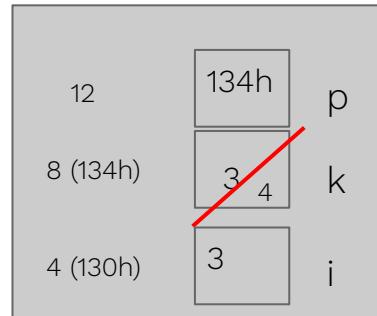
¿Cuál tipo de asignación contribuye a generar mayor garbage?

# Asignaciones en C++ utilizando referencias

C++

```
int main() {  
    int i,k;  
    int * p;  
    p = &k;  
    k = 3;  
    i = k;  
    i = *p;  
    k++;  
}
```

Reg. activación main()



¿Cuál es el r-value en cada instrucción?

- Las variables `i` y `k` se mantienen independientes
- Las variables `*p` y `k` son la misma

¿Qué resultados arroja este programa?

- El operador `&` significa “supresión de extracción de valor”
- No es un “extractor de dirección”

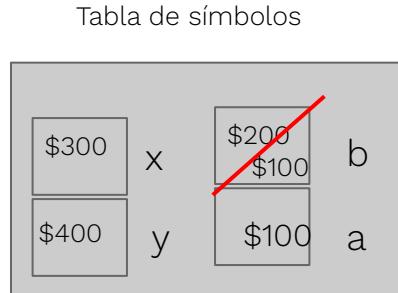
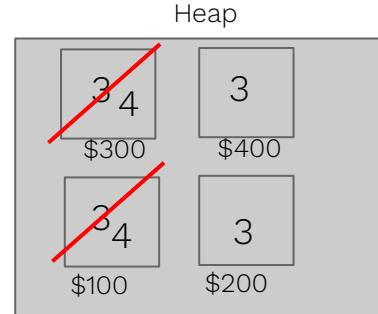
C++

```
int main() {  
    int x=1, y=2; int r;  
    int *p;  
    int **q;  
    p = &x;  
    q = &p;  
    y = **q;  
    r = &(&x);  
    printf("%d",x);  
}
```

# ¿Qué sucede en Python?

## Python 3 ([asignacion.py](#))

```
class Entero:  
    valor = 3  
a = Entero()  
b = Entero()  
b = a  
a.valor = 4  
print (a.valor)  
print (b.valor)  
x = 3  
y = x  
x = 4  
print (y)
```



- La asignación de las variables de tipos básicos es por copia.
- La asignación de las variables de tipos definidos por el usuario son por referencia

¿A cuál de los ejemplos (C++ o Java) es parecido sintácticamente?

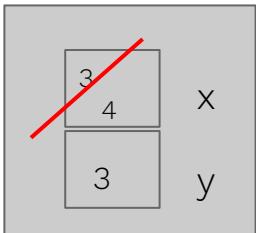
¿A cuál de los ejemplos (C++ o Java) es parecido semánticamente?

# ¿Qué sucede en C#?

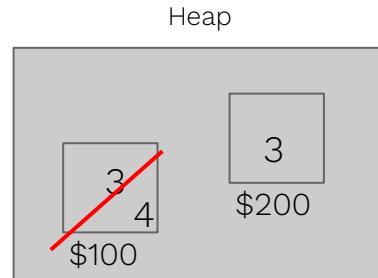
## C# ([asignacion.cs](#))

```
public static void Main(string[] args)
{
    int x = 3;
    int y = x;
    x = 4;
    Console.WriteLine(y);
    Console.WriteLine("\n");
}
```

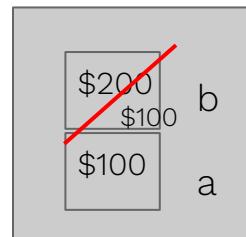
Registro de Activación main()



La asignación de las variables de tipos básicos es por copia.



Reg. activación main()



## C# ([asignacion.cs](#))

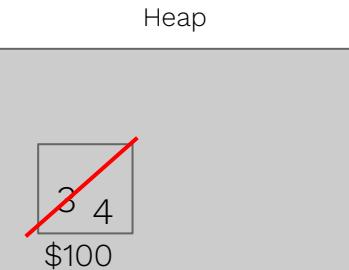
```
class EnteroClase {
    public int valor=3;
}
public static void Main(string[] args)
{
    EnteroClase a = new EnteroClase();
    EnteroClase b = new EnteroClase();
    b=a;
    a.valor = 4;
    Console.WriteLine(b.valor);
}
```

¿A cuál de los ejemplos (C++ o Java) es parecido sintácticamente y semánticamente?

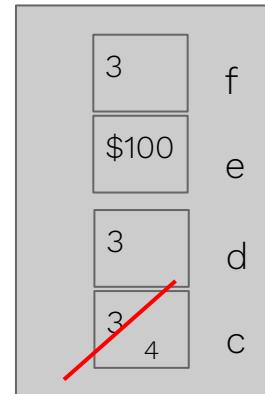
# ¿Qué sucede en C#?

## C# ([asignacion.cs](#))

```
struct EnteroStruct {  
    public int valor;  
}  
public static void Main(string[] args)  
{  
    EnteroStruct c;  
    c.valor = 3;  
    EnteroStruct d = c;  
    c.valor = 4;  
    Console.WriteLine(d.valor);  
    ...  
}
```



Reg. activación main()



```
...  
EnteroStruct e = new EnteroStruct();  
e.valor = 3;  
EnteroStruct f = e;  
e.valor = 4;  
Console.WriteLine(f.valor);  
}
```

- **Las variables de los tipos básicos se asignan por copia**

- **Las variables cuyo tipo se declaró como struct se asignan por copia**
- **Las variables cuyo tipo se declaró como clase se asignan por referencia**

# Asignaciones unarias

Asignaciones implícitas en otras instrucciones

Presentes en varios lenguajes: C++, Perl, Java, etc

**Unaria:**

```
cuenta++;
```

~

```
cuenta = cuenta + 1;
```

**Prefijo:**

```
suma = ++ cuenta;
```

~

```
cuenta = cuenta + 1;  
suma = cuenta;
```

**Postfijo:**

```
suma = cuenta++;
```

~

```
suma = cuenta;  
cuenta = cuenta + 1;
```

**Precedencia:**

```
-cuenta++;
```

~

```
-(cuenta++);
```

# Asignaciones en expresiones y constructores

Presentes en varios lenguajes: C++, Perl, Java, etc

```
while ((ch=getchar()) != EOF) {...}
```

Se asigna el valor de retorno de getchar() a ch y este valor es utilizado en la comparación

```
a = b + (c=d/b)-1;
```

Se admiten por cuestiones de legibilidad

~

```
c=d/b;  
temp = b + c;  
a = temp - 1;
```

```
x = while(...) {  
...break 2;  
} else 5;
```

```
sum = count = 0;
```

Las asignaciones devuelven valor

```
sum = (count := 0) (Python 3.8+)
```

Iteraciones como expresiones

```
a = b.set(valor)
```

Métodos pueden realizar asignaciones

```
int x;
```

Declaración con asignación por defecto del lenguaje

```
Clase1 a1 = new Clase1();
```

Invocaciones a constructores

```
Clase1 a1;
```

Invocación implícita a constructor

# Asignaciones múltiples

Presentes en varios lenguajes: Perl, Ruby, Python, Lua, Go, etc.

## Python ([asignacionmultiple.py](#))

```
x = 3  
y = 4  
x,y = y,x+y
```

## Python Bytecode

```
LOAD_NAME 6 (y)  
LOAD_NAME 5 (x)  
LOAD_NAME 6 (y)  
BINARY_ADD  
ROT_TWO  
STORE_NAME 5 (x)  
STORE_NAME 6 (y)
```

se preserva el valor anterior de y

se intercambian los valores de la pila de operandos

## Go ([asignacionmultiple.go](#))

```
func main() {  
    var x int = 354  
    var y int = 453  
    x,y = y,x+y  
}
```

## Go Assembler

```
mov QWORD PTR [rbp-56], 354  
mov QWORD PTR [rbp-64], 453  
mov rcx, QWORD PTR [rbp-64]  
mov rdi, QWORD PTR [rbp-56]  
mov rsi, QWORD PTR [rbp-64]  
add rsi, rdi  
mov QWORD PTR [rbp-56], rcx  
mov QWORD PTR [rbp-64], rsi
```

se preserva el valor anterior de y en el registro rcx

En los dos casos se crean variables auxiliares para realizar la asignación

Se asigna a x el valor anterior de y

# Semejanzas con pasaje de parámetros

## Asignación por Copia

- Se realiza una extracción de valor de la celda de memoria del lado derecho.
- Ese valor extraído es copiado en la celda de memoria referenciada por el lado izquierdo.

## Pasaje de parámetros por Copia valor

- Se realiza una extracción de valor de la celda de memoria del parámetro real.
- Ese valor extraído es copiado en la celda de memoria referenciada por el parámetro formal.

## Asignación por Referencia

- Se realiza una supresión de la extracción del valor del lado derecho y se obtiene una dirección.
- Esa dirección es copiada en la celda de memoria referenciada por el lado izquierdo.

## Pasaje de parámetros por Referencia

- Se realiza una supresión de la extracción del valor del parámetro real y se obtiene una dirección.
- Esa dirección es pasada como parámetro formal (alias o puntero).

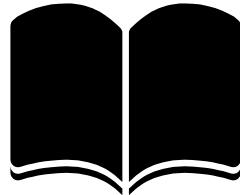
¿Puede existir una asignación por nombre?



# **Tipos: Compatibilidades y Conversiones**

**Lenguajes de Programación I**

# **Tipos: Compatibilidades y conversiones**



- Sistemas de Tipos
- Conversiones
  - Implícitas
  - Explícitas
- Compatibilidades
  - Nombre
  - Estructura
- Representación de la información

# Tipos de datos

## Tipos básicos o escalares

Para el Lenguaje contienen un único valor (pueden estar compuestos internamente por varios valores)

- Booleanos
- Carácter
- Enteros
- Punto fijo
- Punto flotante (real, float, double, etc.)
- Nulos
- Enumeraciones
- Intervalos
- Tipos ordenados
- etc.

## ¿Qué son los tipos de datos?

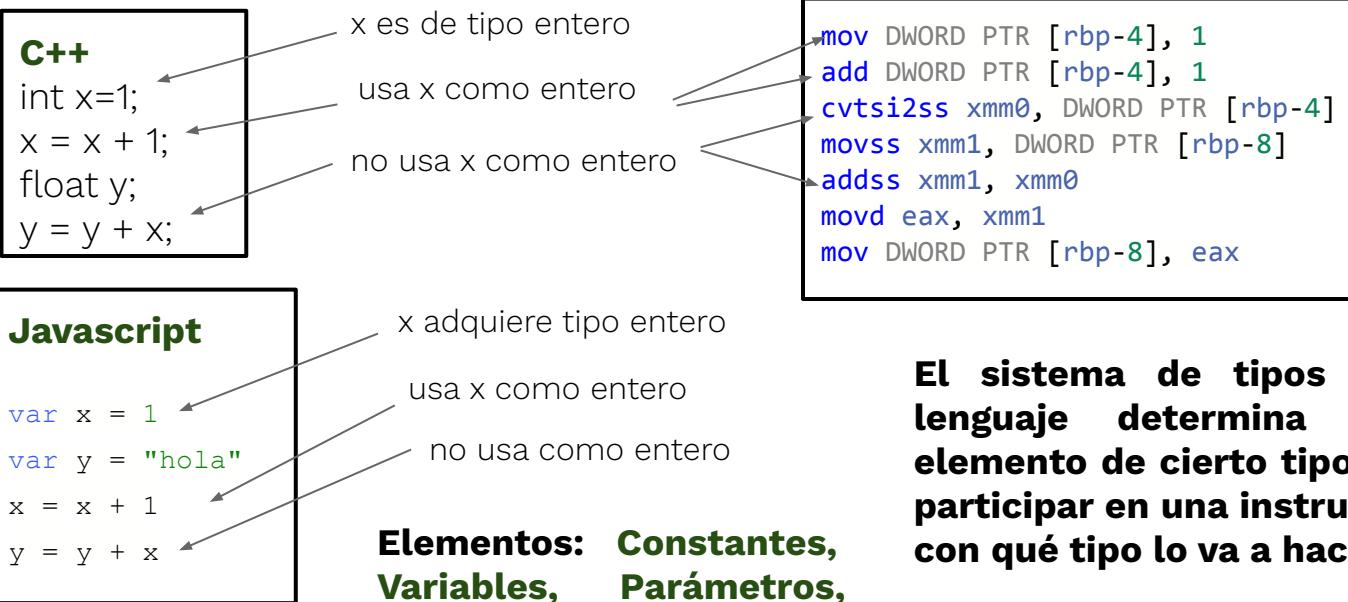
Un conjunto de valores posibles, con un conjunto de operaciones asociadas a esos valores

## Tipos Agregados o compuestos

Para el Lenguaje contienen más de un valor

- Repetición de valores del mismo tipo
  - Arreglos
  - Listas
  - Vectores
  - Conjuntos
- Valores de distinto tipo
  - Estructuras/Registros
  - Tipos recursivos
  - Tipos abstractos de datos
  - Módulos
  - Clases

# Sistemas de Tipos de datos



**Elementos: Constantes, Variables, Parámetros, Retorno de funciones, Expresiones, Funciones, etc.**

**El sistema de tipos de un lenguaje determina si un elemento de cierto tipo puede participar en una instrucción y con qué tipo lo va a hacer.**

# Clasificación de los sistemas de tipos

Momento en que se conoce el tipo

**Static typing**

**Dynamic typing**

Las operaciones aplicables respetan siempre los términos del tipo

**Strong Typing**

**Weakly (Soft) Typing**

Los elementos siempre tienen un valor admisible para el tipo que poseen

**Type Safety**

**Type Unsafety**

**Type Soundness**

[tipos.cpp](#)  
[tipos.ts](#)  
[tipos.pl](#)  
[tipos.php](#)

# Clasificación de los sistemas de tipos

## Compatibilidad

¿Qué sucede si se quiere hacer una operación con dos o más variables de distinto tipo?

## Conversiones

```
void f() {  
    int a,b,c;  
    c = a + b;  
}
```

| Tabla de símbolos |      |        |
|-------------------|------|--------|
| nombre            | tipo | ámbito |
| a                 | int  | f      |
| b                 | int  | f      |
| c                 | int  | f      |

Mismo tipo



Se generan las instrucciones

Distintos tipos totalmente compatibles



Se generan las instrucciones

Distintos tipos parcialmente compatibles



Se realizan “conversiones” y se generan las instrucciones

Distintos tipos no compatibles



Error

**Todo esto puede ocurrir en tiempo de compilación o en tiempo de ejecución dependiendo del tipo de lenguaje**

# Compatibilidad de tipos por nombre

## ADA

```
type enteros is new Integer;  
a : Integer;  
b : Enteros;  
a := a + b;
```

Error de incompatibilidad de tipos

## ¿Qué ventajas tiene esta forma de compatibilidad?

- Mayor seguridad en las operaciones
- Mayor legibilidad de los programas

## ¿Qué significa este error?

ADA tiene compatibilidad de tipos por nombre

Los elementos son compatibles si los nombres de los tipos son iguales

## ¿Cómo se hace para poder operar entre variables de distinto tipo?

## ADA

```
a := a + Integer(b);
```

Las escribe el programador, las provee el lenguaje o el programador

## Conversiones explícitas

# Compatibilidad de tipos por estructura

C

```
typedef int entero;  
int a;  
entero b;  
a = a + b;
```

No da ningún tipo de error

## ¿Qué significa esto?

C tiene una forma de compatibilidad por estructura

Los elementos son compatibles si sus representaciones en memoria (estructuras) son compatibles

¿Cómo se hace para poder operar entre variables de distinto tipo?



El lenguaje puede realizar conversiones implícitas

¿Qué ventajas tiene esta forma de compatibilidad?

- Mayor flexibilidad para operar entre tipos diferentes
- Mayor facilidad de escritura de los programas

# Compatibilidades combinadas

## C++ / Java

```
int x=3  
long y=5;  
long z = x + y;
```

Compatibilidad por estructura

## ¿Cuales son los beneficios de la compatibilidad combinada?

- **Tipos básicos:** Mayor flexibilidad para operar entre tipos diferentes y mayor facilidad de escritura de los programas
- **Tipos definidos por el usuario:** Mayor seguridad en las operaciones y mayor legibilidad de los programas

Compatibilidad por nombre

## Java ([compatibilidadnombre.java](#))

```
Class Clase1 {int x;}  
Class Clase2 {int x;}  
...  
Clase1 a = new Clase1();  
Clase2 b = new Clase2();  
a = b;
```

## C++ ([compatibilidadnombre.cpp](#))

```
Class Clase1 {int x;}  
Class Clase2 {int x;}  
...  
Clase1 a,*c;  
Clase2 b,*d;  
a = b;  
c = d;
```

# Relación Compatibilidad y conversiones

Compatibilidad por nombre



**Conversiones explícitas**

Compatibilidad por estructura



**Conversiones implícitas  
o explícitas**

C++ ([compatibilidadnombre2.cpp](#))  
`int *x;  
float *y;  
*y = *x;  
y = x, (float*) x;`

Incompatibles por estructura (compatibilidad parcial, conversiones implícitas)

- Compatibles por estructura (compatibilidad parcial, conversiones implícitas)
- Incompatibles por nombre (conversión explícita provista por el lenguaje)

# Relación Compatibilidad y conversiones

## Python 3

```
x = 1.0  
y = "2"  
x = x + y  
x = x + float(y)
```

Se resuelve y se hace efectiva durante la ejecución.

## Tabla de símbolos

| nombre | tipo   | referencia |
|--------|--------|------------|
| x      | int    | \$100      |
| y      | string | \$200      |

## Tabla de compatibilidad de tipos

| Tipo   | float | string |
|--------|-------|--------|
| float  | si    | no     |
| string | no    | si     |

Se resuelve durante la compilación y se hace efectiva durante la ejecución.

## Java

```
int x=1;  
double y = 2.0;  
y = x;  
x = y; (int) y;
```

Ambas tablas existen en tiempo de ejecución

## Tabla de símbolos

| nombre | tipo   | referencia |
|--------|--------|------------|
| x      | int    | Offset 1   |
| y      | double | Offset 2   |

## Tabla de compatibilidad de tipos

| Tipo   | int | double |
|--------|-----|--------|
| int    | si  | si     |
| double | no  | si     |

Ambas tablas existen sólamente en tiempo de compilación

# Relación Compatibilidad y conversiones

Se comprueba que se pueda realizar la asignación

$x = a + b * c;$

| Tabla de símbolos |           |            |
|-------------------|-----------|------------|
| nombre            | tipo      | referencia |
| x                 | Tipo de x | ...        |
| a                 | Tipo de a | ...        |
| b                 | Tipo de b | ...        |
| c                 | Tipo de c | ...        |

| Tipo   | int | float | string |
|--------|-----|-------|--------|
| int    |     |       |        |
| float  |     |       |        |
| double |     |       |        |

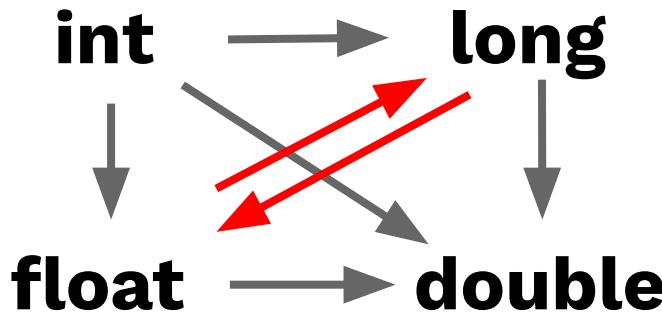
Se verifican los tipos de las variables involucradas y si es necesario se “convierten”

El proceso del lado derecho es el mismo independientemente si el lenguaje tiene tipos estáticos o dinámicos.

En lenguajes dinámicos puede implicar que el lado izquierdo adopte el tipo del lado derecho.

# Criterios de conversiones implícitas

Se intenta convertir hacia un tipo más abarcativo (agrandar la representación)



**C++** ([conversiones.cpp](#))

```
int i; long l; float f; double d;  
d = i*l+i*f+f*l+i*d+f*d;
```

Considerando int de 2 bytes, long y float de 4 bytes y double de 8 bytes.

float y long suelen tener el mismo tamaño en bytes pero diferente representación interna.

No se puede saber cuál va a contener el valor mayor

Ambos tipos se convierten a double

**¿Cómo se hacen las conversiones?**

# Conversiones implícitas

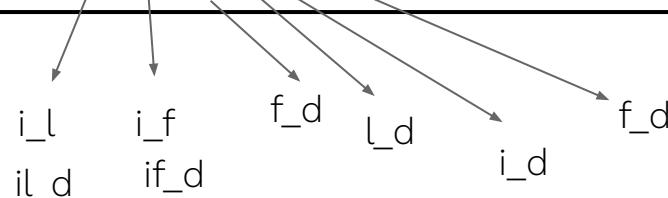
```
float x,z;  
long y;  
x = y + z;
```

¿Cambia realmente el tipo de y a double?  
¿Cambia realmente el tipo de z a double?  
¿Cambia el tipo de x a double?

¿Qué hace el lenguaje cuando se encuentra con esto?

C++ ([conversiones.cpp](#))

```
int i; long l; float f; double d;  
d = i*l+i*f+f*l+i*d+f*d;
```



En sentido estricto  
nunca cambian los tipos  
de las variables del lado  
derecho.  
En un lenguaje con tipos  
dinámicos, podría  
cambiar el tipo de la  
variable del lado  
izquierdo

Crea variables auxiliares invisibles  
para el programador

¿Dónde pueden estar esas variables?

- Variables estáticas
- Pila o Tabla de símbolos
- Registros o pila de operandos del CPU

# Conversiones en asignaciones

```
x = 1  
y = 2.0  
x = y
```

Cambia el tipo de x a punto flotante

Se emite un error por incompatibilidad

Se “convierte” el valor del lado derecho para realizar la asignación

**C**  
double d=2.3;  
int i=1;  
i = d;

**Java**  
double d=2.3;  
int i=1;  
i = ~~d~~; (int) d;

→ Tipos dinámicos

→ Tipos estáticos

→ Tipos estáticos

En algunos casos tipos dinámicos (modos “strict” en Typescript, Dart, Hack)

Hay lenguajes más flexibles que otros en cuanto a la posible pérdida de precisión

[Ver ejemplo](#)  
[compatibilidad.dart](#)

# Representación interna IEEE

C,C++, etc.

```
float j = 0.1;  
if (j == 0.1)  
    printf("que bien");  
else  
    printf("que mal");
```

¿qué imprime?

En varios lenguajes  
imprimirá “que mal”

¿por qué?

La variable no tiene el  
valor 0.1 almacenado en  
su celda de memoria

Ver ejemplos  
[representacion.cpp](#),  
[representacion.java](#),  
[representacion.cs](#),  
[representacion.py](#), etc.

Ej. convertir 0.625 de decimal a binario

1.  $2 \times 0.625 = 1.25$ ,  $[1.25] = 1$
2.  $2 \times 0.25 = 0.5$ ,  $[0.5] = 0$
3.  $2 \times 0.5 = 1$ ,  $[1] = 1$

Resultado = 0.101



- Existe una cantidad acotada de dígitos para la mantisa y el exponente
- Esto no permite almacenar números periódicos
- ¿dónde está la periodicidad del valor 0.1?

# Representación interna IEEE

## Ej. convertir 0.1 de decimal a binario

1.  $2 \times 0.1 = 0.2$ ,  $[0.2] = 0$
2.  $2 \times 0.2 = 0.4$ ,  $[0.4] = 0$
3.  $2 \times 0.4 = 0.8$ ,  $[0.8] = 0$
4.  $2 \times 0.8 = 1.6$ ,  $[1.6] = 1$
5.  $2 \times 0.6 = 1.2$ ,  $[1.2] = 1$
6.  $2 \times 0.2 = 0.4$ ,  $[0.4] = 0$
7.  $2 \times 0.4 = 0.8$ ,  $[0.8] = 0$
8.  $2 \times 0.8 = 1.6$ ,  $[1.6] = 1$
9.  $2 \times 0.6 = 1.2$ ,  $[1.2] = 1$
10.  $2 \times 0.2 = 0.4$ ,  $[0.4] = 0$
11. ...

**Resultado = 000110011001100..**

Es periódico, por lo tanto en IEEE no es posible almacenar 0.1 con exactitud

Lo mismo sucede con 0.01, 0.001, etc. y en general en todas las fracciones que tengan 5 como divisor

**Esto es un problema en aplicaciones del ámbito contable donde:**

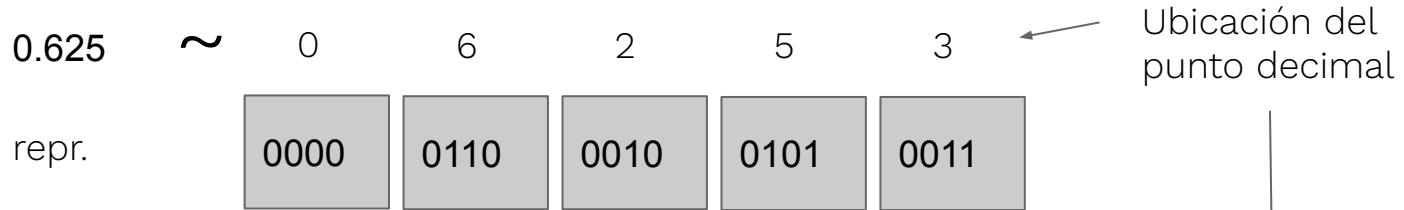
```
if (Total_debe - Total_haber == 0)  
    print ("resultados balanceados")
```

**¿Cómo lo solucionan los lenguajes en los que imprime “que bien” ?**

**Realizan la comparación con un rango de valores cercanos**

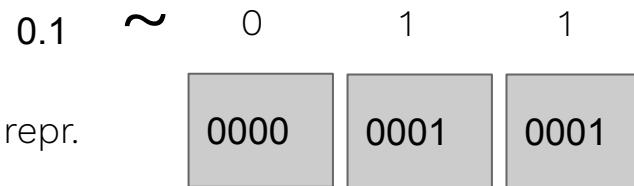
# Representación interna BCD

Utiliza 4 bits para codificar en binario un número en base 10



**Elimina la periodicidad del 0.1**

- Menos precisa que IEEE
- Ocupa más espacio que IEEE
- Ineficiente para implementar operaciones en Hardware



Muy pocos lenguajes la implementan

# Lenguajes y Representación

## Lenguajes con Representación fija

**BCD:** COBOL

**IEEE:** Fortran, Pascal, Algol, C, C++, Java, Kotlin, C#, Python, Swift, PHP, Ruby, Lua, Javascript, Dart, Hack, etc.

## Lenguajes con Representación variable nativa: ADA

**ADA**

```
procedure f is
  x: float;
  type BCD is delta 0.1 range
    0.0 .. 1.0;
  y : BCD;
begin
  y := BCD(x) + y;
end f;
```

por defecto  
es IEEE

Es BCD (con el delta y el rango el compilador puede  
calcular la cantidad de dígitos BCD)

Se realizan conversiones para operar entre IEEE y BCD

## Lenguajes con soporte de Representación BCD por medio de bibliotecas:

C, C++, Java, Python, Perl, Ruby, etc.

# Tipos de datos para Big Data

**Big Int**

**Big Float**

**Big Complex**

...

Permiten almacenar un valor tan grande como se necesite o con un rango definido por el programador (arbitrary precision numbers).

- BigFloat en C++
- BigNumber en Javascript
- BigInt y BigFloat en Python
- Decimal en C#
- BigInteger y BigFloat en Java
- ...

## ¿Cómo se representan internamente?

Con una secuencia de dígitos codificados en bits (BCD)

**Java**

```
import java.math.BigInteger;  
....  
BigInteger x;
```

Algunos lenguajes proveen Bibliotecas con definiciones de tipos BigInt, BigFloat, etc.

**Python**

```
x = 2;  
while(1):  
    x = x*x  
    print(type(x))  
    print(x)
```

¿Qué sucede  
en este caso?

# Precisión vs. Exactitud

**Precisión**



Que la representación ofrezca dígitos suficientes para almacenar el valor de una magnitud considerando su error.

Ej. el valor de un ángulo en radianes debe tener una precisión de 10e-40.

**Exactitud**



Que la representación permita almacenar correctamente el valor de una magnitud

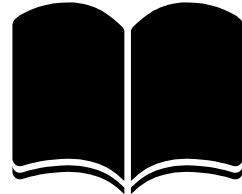
Ej. La diferencia entre dos totales debe ser cero expresada con 2 dígitos de precisión.



# **Tipos: Conversiones y Coerciones**

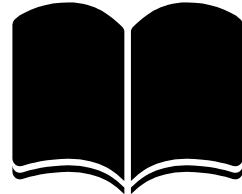
**Lenguajes de Programación I**

# Qué vimos hasta ahora



- **Introducción**
- **Sintaxis**
- **Semántica - Binding, Clasificación de lenguajes**
- **Alcance dinámico y estático**
- **Pila de ejecución**
- **Clasificación de variables en lenguajes de pila**
- **Manejo dinámico de memoria - Garbage Collection**
- **Interacción con el Sistema Operativo - concurrencia**
- **Pasaje de parámetros**
- **Asignaciones**
- **Tipos - Compatibilidades**
- Tipos - Conversiones
- Objetos

# **Tipos: Compatibilidades y conversiones**



- Registros/Estructuras
- Uniones
- Coerciones
  - Dereferencing
  - Deproceduring
  - Uniting
  - Widening
  - Rowing
  - Voiding
- Castings
  - Del lado derecho implícitos y explícitos
  - Del lado izquierdo
- Conversiones en Lenguajes dinámicos
- Type checkers

# Registros/estructuras

## ¿Qué son?

- Son una forma de tipos de datos compuestos
- A los componentes de las variables de estos tipos se accede explícitamente por medio de un identificador
- Cada componente tiene su propio offset
- No contienen unidades, salvo que estén almacenadas como closures.



## C++

```
structs.cpp  
int main ()  
{  
    struct {  
        int c1;  
        float c2;  
    } e;  
    e.c1 = 1;  
    e.c2 = 2.3;
```

## Pascal

```
type MiRegistro =  
record  
    c1 : integer;  
    c2 : real;  
end;  
var r : MiRegistro;  
begin  
    r.c1 := 1;  
    r.c2 := 2.3;  
end.
```

## C# structs.cs

```
struct MiStruct {  
    public int c1;  
    public float c2;  
};  
...  
MiStruct s1;  
s1.c1 = 1;  
s1.c2 = 2.3;  
MiStruct s2 = new MiStruct();  
s2.c1 = 1;  
s2.c2 = 2.3;
```

Esto las diferencia de un objeto

- Muchos lenguajes tienen estructuras.
- En Go y Rust son las únicas formas de representar la parte de datos de un objeto.

s1 es una estructura y s2 es un puntero a una estructura

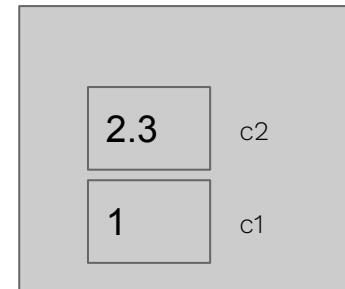
# Almacenamiento de Estructuras

C++  
structs.cpp

```
int main ()  
{  
    struct {  
        int c1;  
        float c2;  
    } e;  
    e.c1 = 1;  
    e.c2 = 2.3;
```

```
mov DWORD PTR [rbp-8], 1  
movss xmm0, DWORD PTR .LC0[rip]  
movss DWORD PTR [rbp-4], xmm0
```

Almacena cada componente con un offset diferente.



- ¿Existen instrucciones de creación de la estructura?
- ¿Cómo se organizan internamente los componentes?
- ¿Se deben almacenar marcas de comienzo y fin de las estructuras en el registro de activación?
- ¿Pueden ser variables estáticas?
- ¿Pueden ser variables dinámicas?
- ¿Pueden ser variables semi-dinámicas?

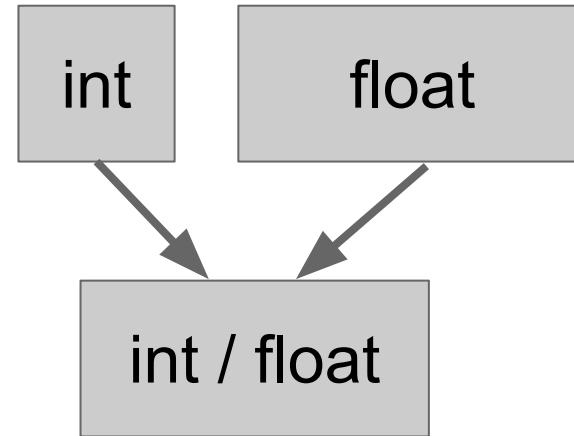
# Uniones

## ¿Qué son?

- Mecanismo para superponer variables en el mismo lugar de memoria.
- Surge por la necesidad de almacenar el mismo dato y tratarlo con diferentes tipos
- Todos los elementos tienen el mismo direccionamiento.

- Algunos lenguajes tienen uniones (C/C++, Pascal, Rust, Swift, Haxe)
- Pueden representar una situación de type unsafety y soft typing dependiendo del tipo de unión.

```
C++  
unions.cpp  
int main ()  
{  
    union {  
        int c1;  
        float c2;  
    } u1;  
    u1.c1 = 1;  
    u1.c2 = 2.3;
```



**¿Cómo se almacenan ambas variables en el mismo lugar?  
Se crea un espacio para el más grande**

**¿Cómo se sabe qué es lo que está almacenado en la unión?**  
**Depende del tipo de unión** (a veces lo sabe solamente el programador y a veces lo sabe también el lenguaje)

# Uniones propiamente dichas

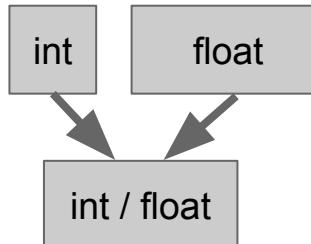
## ¿Qué son?

- Respetan la definición de unión al extremo (se destina lugar a las variables y el programador no hace nada explícitamente para determinar cuál variable está almacenada)

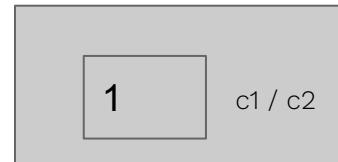
## ¿Cómo se compila esta instrucción?

C++  
[unions.cpp](#)

```
int main ()  
{  
    union {  
        int c1;  
        float c2;  
    } u1;  
    u1.c1 = 1;  
    u1.c2 = 2.3;  
    int x = u1.c1;
```



```
mov DWORD PTR [rbp-4], 1  
movss xmm0, DWORD PTR  
.LC0[rip]  
movss DWORD PTR [rbp-4], xmm0  
...  
mov eax, DWORD PTR [rbp-4]  
mov DWORD PTR [rbp-8], eax
```



R.A.  
main()

El lenguaje tiene tipos estáticos, por lo tanto el compilador genera instrucciones para mover un entero a un entero.

El lenguaje permitió que se almacene un valor de punto flotante y que ese valor se utilice como un entero

¿Cómo sabe el lenguaje cuál variable está almacenada?

No lo sabe. Solamente el programador lo sabe

# Uniones con discriminante implícito

## ¿Qué son?

- Tagged unions. Agregan a las uniones propiamente dichas, un discriminante implícito.
- El lenguaje utiliza este discriminante para saber cuál variable se encuentra almacenada.

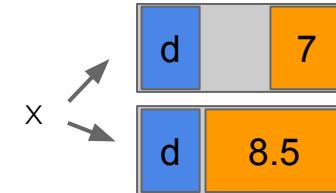
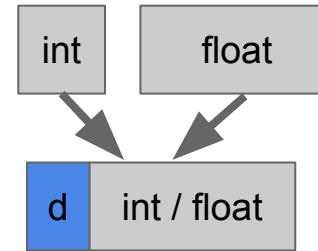
Se declaran x e y como variables del tipo tagged Union entre int y real

Se usa la unión del lado izquierdo. Del lado derecho se convierten 7 (int) y 8.5 (real) al tipo union

**ALGOL**

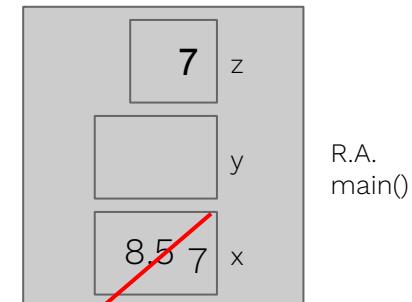
```
union (int,real) x,y;  
x = 8.5;  
x = 7;
```

```
int z;  
z := x;  
...  
case x in  
  (int) : z := x;  
  (real) : write("error");  
esac
```



¿Qué pasa si se quiere usar la unión del lado derecho? ¿Cómo se compila esta instrucción?  
No es posible. Error de compilación

- Se utiliza una cláusula de conformidad para determinar los casos donde la unión posee una variable de cada tipo.
- En cada caso de la cláusula, no hay problemas de compatibilidad.



# Uniones con discriminante implícito

Se define MiUnion como una tagged union

Se declara u como variable del tipo MiUnion

Se usa la unión del lado izquierdo. Del lado derecho se convierten los tipos int y float al tipo MiUnion

Se usa una cláusula de conformidad para saber cuál variable está conteniendo la unión para poder utilizarla del lado derecho

## Rust [taggedunions.rs](#)

```
enum MiUnion {  
    i(i16),  
    f(f32),  
}  
fn main() {  
    let mut u : MiUnion;  
    u = MiUnion::i(1);  
    u = MiUnion::f(2.3);  
    let mut x : i16;  
    let mut y : f32;  
    x = 2;  
    y = 3.4;  
    match u {  
        MiUnion::i(ii) => x = ii,  
        MiUnion::f(ff) => y = ff,  
    }  
    println!("{}{}", x, y);  
    println!("{}{}", y, x);  
}
```

Otros lenguajes que tienen este tipo de uniones: Swift, Haxe, Typescript.

No hay ningún componente que explícitamente utilice el programador para consultar por el tipo que contiene la unión.

En cada uno de estos casos no hay problemas de incompatibilidad de tipos

**¿Puede darse una situación de inseguridad de tipos (type unsafety)?**

**No. Estas uniones son seguras**

Rust además tiene uniones propiamente dichas (ver [unions.rs](#))

# Uniones con discriminante explícito

## Registros variantes

### Pascal/Delphi ([union.pas](#))

```
program Uniones;
type MiUnion = record
  case disc: Integer of
    1: (i: integer);
    2: (r: real);
end;
var
  u1 : MiUnion;
begin
  u1.disc :=1;
  u1.i := 5;
  u1.disc :=2;
  u1.r := 3.4;
  u1.i := 1;
```

Se define el tipo MiUnion como un registro

Discriminante ( contiene información del tipo de la variable que contenga la unión)

Se declara una variable de tipo MiUnion

Se asigna el valor 1 al campo discriminante

Se asigna 5 al campo i

Se asigna el valor 2 al campo discriminante

Se asigna 3.4 al campo r

**¿Qué sucede si se hace esto?**



R.A.  
Uniones

- El programador es responsable del discriminante
- El lenguaje no controla que el discriminante tenga la información de acuerdo al tipo de la unión.

**Estas uniones son inseguras (type unsafety)**

# Uniones con discriminante explícito

## Registros variantes

### ADA ([uniones.adb](#))

```
procedure Uniones is
```

```
    type Disc is (IsInt, IsReal);
```

```
    type MiUnion(tipo: Disc := IsInt) is record
```

```
        case tipo is
```

```
            when IsInt => i: Integer;
```

```
            when IsReal => r: Float;
```

```
        end case;
```

```
    end record;
```

```
    u : MiUnion; x: Integer;
```

```
begin
```

```
    u := (IsInt,1);
```

```
    x := u.i;
```

```
    u := (IsReal,1.2);
```

```
    x := u.i;
```

```
    u := (IsInt,2.3);
```

```
end Uniones;
```

Se define el tipo Discriminante como una enumeración

Se define el tipo MiUnion como un registro, con el tipo Discriminante

Se declara una variable de tipo MiUnion

Se asigna el valor 1 a la unión, colocando explícitamente el valor del discriminante

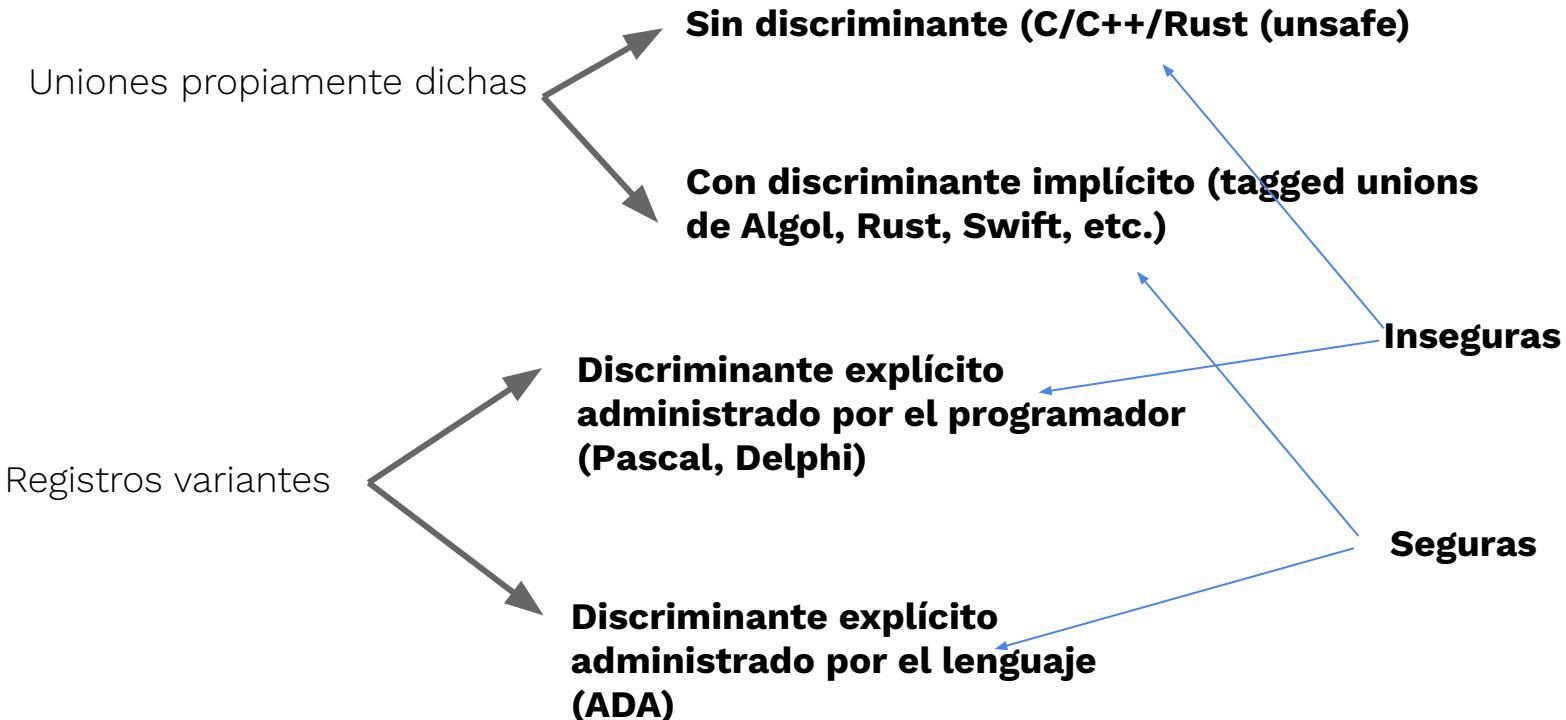
Se asigna el valor 1.2 a la unión, colocando explícitamente el valor del discriminante

### ¿Qué sucede si se hace esto?

- El lenguaje obliga a utilizar el discriminante junto con la unión. Siempre sabe el tipo que contiene la unión.
- Si se intenta utilizar el valor como otro tipo, se produce un error (del lado izquierdo en tiempo de compilación y del lado derecho en tiempo de ejecución).
- Nunca se almacena un valor que no corresponde al tipo.

**Estas uniones son seguras**

# Clasificación de las uniones



# Dereferencing

```
int a;  
int b;  
...  
a = 3;  
a = b;
```

¿Por qué estas  
asignaciones se tratan  
de forma diferente?

El lado izquierdo espera un entero, recibe un entero

El lado izquierdo espera un entero, recibe algo que tiene una dirección.

En lenguajes con tipos estáticos, el lado izquierdo recibe siempre un valor apto para colocarlo en la celda asociada al nombre de la variable

- ¿Qué espera el lado izquierdo?
- ¿Qué encuentra del lado derecho?

En todos los lenguajes que tienen asignación por valor existe **un dereferencing implícito**

En algunos lenguajes existen **dereferencing implícitos adicionales**

En la mayoría de lenguajes que admiten punteros, existen **dereferencing explícitos**

# Dereferencing

## Algol

```
int a, b;
ref int c,d;
ref ref int e,f;
```

## C

```
int a, b;
int *c, *d;
int **e, **f;
```



**Algol tiene dereferencing implícitos automáticos**

## Algol

```
a := b
a := c
a := e
```

## Deref

1  
2  
3

## C

```
a = b
a = *c
a = **e
```

Copian un entero

```
c := a
c := d
c := e
```

0  
1  
2

```
c = &a
c = d
c = *e
```

Copian la dirección de un entero

```
e := a
e := c
e := f
```

NO  
0  
1

```
e = NO
e = &c
e = f
```

Copian la dirección de la dirección de un entero

# Dereferencing adicionales

Implícitos



- Se escribe menos y el lenguaje es más flexible
- Se pierde legibilidad
- Se escribe más
- Se gana legibilidad

Explícitos



## C#

```
struct Estru {  
    public int valor;  
};  
public Estru c,a;  
c = new Estru();  
a = c;
```

c es un puntero a una variable de tipo Estru

se espera una variable de tipo Estru pero se da una dirección a una variable de tipo Estru (2 deref)

Otros lenguajes con dereferencing implícitos adicionales

## Rust [\(deref.rust\)](#)

```
fn main() {  
    let mut a = 1;  
    let mut pa = &a;  
    let mut b : i32;  
    println!("a: {}", a);  
    println!("pa: {:?}", pa);  
    println!("pa: {}", pa);  
    println!("pa: {}", *pa);  
    println!("pa: {}", &pa);  
}
```

Se espera un entero pero se da una dirección de un entero (2 defef)

Se espera un entero pero se da una dirección de una dirección de un entero (3 defef)

# Deproceduring

Una unidad (procedimiento) es una dirección de memoria donde se ubica la primera instrucción ejecutable

## Python

```
def f():  
    return 1  
def g():  
    return 2  
a = f()  
print(a)  
  
temp = f  
f = g  
g = temp  
a = f()  
print(a)
```

Se definen funciones normalmente  
Se convierte f a un valor (deproceduring)  
Se asigna la dirección de f  
Se asigna la dirección de g  
Se asigna la dirección de f almacenada en temp  
Se convierte f a un valor (deproceduring)

# Deproceduring

Existe en todos los lenguajes por defecto, en los que tienen la característica de manipular la dirección de las unidades, es posible anular el deproceduring

## Python

```
x=1  
def a():  
    y=2  
    def b():  
        print y+x  
        return b  
    def c():  
        r = 3  
        return r+x  
    z=a()  
    w=z  
    z()  
    w()  
    s = c()
```

¿En cuáles instrucciones hay deproceduring?

¿Qué sucede cuando se pasa una función como parámetro?

¿Qué sucede cuando se invoca a una función que devuelve un closure?

deproceduring

no hay deproceduring

## GO

```
func f(y func()) {  
    fmt.Println("En f")  
    y()  
}  
func g() {  
    fmt.Println("En g")  
}  
func main() {  
    f(g)  
}
```

# Uniting

Conversión de una variable de un tipo determinado a un tipo Unión

**C++**

unions.cpp

```
int main ()  
{  
    union {  
        int c1;  
        float c2;  
    } u1;  
    u1.c1 = 1;  
    u1.c2 = 2.3;
```

Conversión de las constantes 1 y 2.3 al tipo union

El tipo del lado derecho debe ser compatible con los tipos admitidos por la union

# Voiding

**Algol**

```
void q  
q := zz();
```

voiding

Conversión de una variable al tipo void

Se utiliza en lenguajes donde los procedimientos siempre devuelven valor y no se quiere utilizar el valor de retorno

Otros usos de void que no implican voiding:

En C/C++ se utiliza para especificar que una función no devuelve valor o para implementar punteros genéricos (por ejemplo void \* de la función malloc)

En Haskell se utiliza para especificar un conjunto vacío

# Widening

Convertir de un tipo menos abarcativo a un tipo más abarcativo

```
int a;  
float b;  
b := 7;  
b := a;
```

Antes de la asignación se realiza un widening

Antes de la asignación se realiza un dereferencing y luego un widening

**Prácticamente todos los lenguajes poseen Widening automático** (salvo ADA, ML, F# entre otros)

## Usos:

- Expresiones del lado derecho
- Asignaciones
- Pasaje de parámetros
- Valor de retorno de una función

C

```
int a;  
long c;  
c = a;
```

```
mov eax, DWORD PTR [rbp-4]  
cdqe  
mov QWORD PTR [rbp-16], rax
```

Widening

C

```
int a;  
float c;  
c = a;
```

```
cvtsi2ss xmm0, DWORD PTR [rbp-4]  
movss DWORD PTR [rbp-8], xmm0
```

# Rowing

Convertir de una variable de algún tipo a un arreglo de ese tipo

## ALGOL

```
[1:1] int a := 2;  
[,] real x := 3;
```

Se convierte la constante entera 2 a un arreglo unidimensional de 1 entero

Se aplica una conversión widening de la constante entera 3 a real y luego se realiza un rowing a un arreglo tridimensional de dimensiones 1:1, 1:1, 1:1.

**Muchos lenguajes dinámicos lo implementan** (J, R, Julia, APL, Matlab, Matcad, Octave, etc.)

## R

```
a <- c(1,2,3)  
b <- c(1,1,1)  
a = a + 1  
a = a + b
```

## Matlab

```
[1 2 3 4] + 1
```

## J

```
avg =: + / % #  
avg 1
```

Rowing

Siempre es del lado derecho, en lenguajes dinámicos puede provocar también una conversión de tipo en el lado izquierdo si no era de tipo arreglo

# Casting de Lado derecho

**C**

```
int x = 3;  
x = 2.3 + x / 2;
```

casting implícito

son  
equivalentes

**C**

```
int x = 3;  
x = (int) 2.3 + (double) x / 2;
```

casting explícito

**C**

```
void max(int i,double d)  
{  
printf("int double");  
};  
void max(double d,int i)  
{  
printf("double int");  
}  
  
max(3,2);  max((double) 3, (int) 2);
```

Error: Ambiguity between max(int,double) and max(double,int)

Las conversiones explícitas eliminan la ambigüedad

**Existen en la mayoría de los lenguajes estáticos, de tipo algol y dinámicos**

# Casting del lado izquierdo

**C++ castingizquierdo.cpp**

```
int main() {  
    int x = 1;  
    int *p = &x;  
    *(double *) p = 3.2;  
    printf("%d",x);  
}
```

\*p = 3.2;

**¿Qué significa esto?**

Tratar a la celda como...

```
movsd xmm0, QWORD PTR .LC0[rip]  
movsd QWORD PTR [rax], xmm0
```

En este caso tratar a la celda apuntada por p como si fuese un puntero a double. El compilador genera instrucciones para tratar a \*p como si fuese double

```
mov rax, QWORD PTR [rbp-16]  
mov DWORD PTR [rax], 3
```

**¿Cambia de tipo la variable?**



**Puede alterar valores  
de otras celdas  
adyacentes si se trata  
la celda como de un  
tipo más abarcativo.**

**Útil para el manejo  
de punteros  
genéricos (Raw  
Pointers)**

# Casting del lado izquierdo en Algol

## Algol

```
ref int c;  
ref int c = 3;
```



Cuando se quiere realizar una asignación que posea menos de 0 dereferencing

¿Cómo se haría esto en C/C++ otros lenguajes con punteros?

## C/C++

```
int *c;  
*c = 3;
```

- En ambos lenguajes es explícito
- El sistema de tipos del lenguaje debe permitir la asignación
- Existe en todos los lenguajes que poseen punteros con operadores explícitos (C,C++,Go,Rust y otros)

# Conversiones en Lenguajes Dinámicos

## Perl

[\(conversiones.pl\)](#)

```
#use warnings;  
$a = 1;  
$b = "Hola";  
$c = $a+$b;  
print $c;
```

Está utilizando \$b de tipo string como si fuese un entero



Tipos débiles (usar una variable de un tipo en términos de otro tipo)

## PHP / HACK

[\(conversiones.php\)](#)

```
<?php (cambiar a  
<?hh para hack)  
$a = 1;  
$b = "2Hola3";  
$c = $a+$b;  
print $c;
```

## Python

[\(conversiones.py\)](#)

```
a = 1  
b = "Hola"  
c = a + b  
print c
```

A non well formed numeric value encountered  
imprime 3  
(trata parte del string como entero)

## Ruby

[\(conversiones.rb\)](#)

```
a=1  
b="hola"  
c= a+b  
p c
```

imprime 12Hola3  
(trata entero como string)

conversiones.rb:3:in `+': String can't be coerced into Integer  
(TypeError)

## Javascript

[\(conversiones.js\)](#)

```
<script>  
var a = 1;  
var b = "2Hola3";  
var c = a + b;  
document.write(c);  
</script>
```

# Lenguajes Dinámicos con características Strong typing

## Dart ([compatibilidad.dart](#))

```
class A {  
    int x = 1;  
    var z = 2;  
}  
  
void main() {  
    int w;  
    w = 1;  
    w = "Hola";  
    var y;  
    y = 2;  
    y = "Hola";  
    print(w);  
    var a1 = A();  
    a1.x = 3;  
    a1.x = "Hola";  
    a1.z = "Hola";  
}
```

Declaración de atributo con anotación de tipo

Declaración de atributo sin anotación de tipo

Declaración de variable con anotación de tipo

**Error, no es posible asignar String a Int**

Declaración de variable sin anotación de tipo

Asignación con tipos diferentes

**Error, no es posible asignar String a Int**

**Error, no es posible asignar String a Int**

**¿El lenguaje posee tipos estáticos?**

- No, solamente tiene verificación de tipos en variables declaradas con anotaciones de tipo y atributos definidos en clases

# Lenguajes Dinámicos con características Strong typing

## TypeScript (compatibilidad.ts)

```
var a = 1  
var b = "hola"  
var c = a + b  
console.log(c)
```

```
var a2 = 1  
var b2 = "hola"  
var c2 : number = a2 +  
b2  
console.log(c2)
```

Imprime “1Hola”

Se declara c2 con anotación de tipo “number”

**Error, no es posible asignar String a Int**

**¿El lenguaje posee tipos estáticos?**

- No, solamente tiene verificación de tipos en variables declaradas con anotaciones de tipo y atributos definidos en clases.

Algunos Lenguajes dinámicos (Dart, Hack y TypeScript) han incorporado características de verificación de tipos y un modo “strict” que impide la ejecución del programa si puede darse una situación de incompatibilidad de tipos



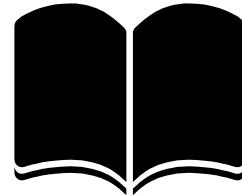
**No significa que estos lenguajes tengan tipos estáticos**



# Objetos

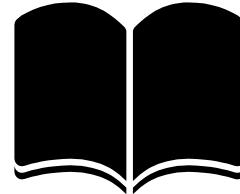
Lenguajes de Programación I

# Qué vimos hasta ahora



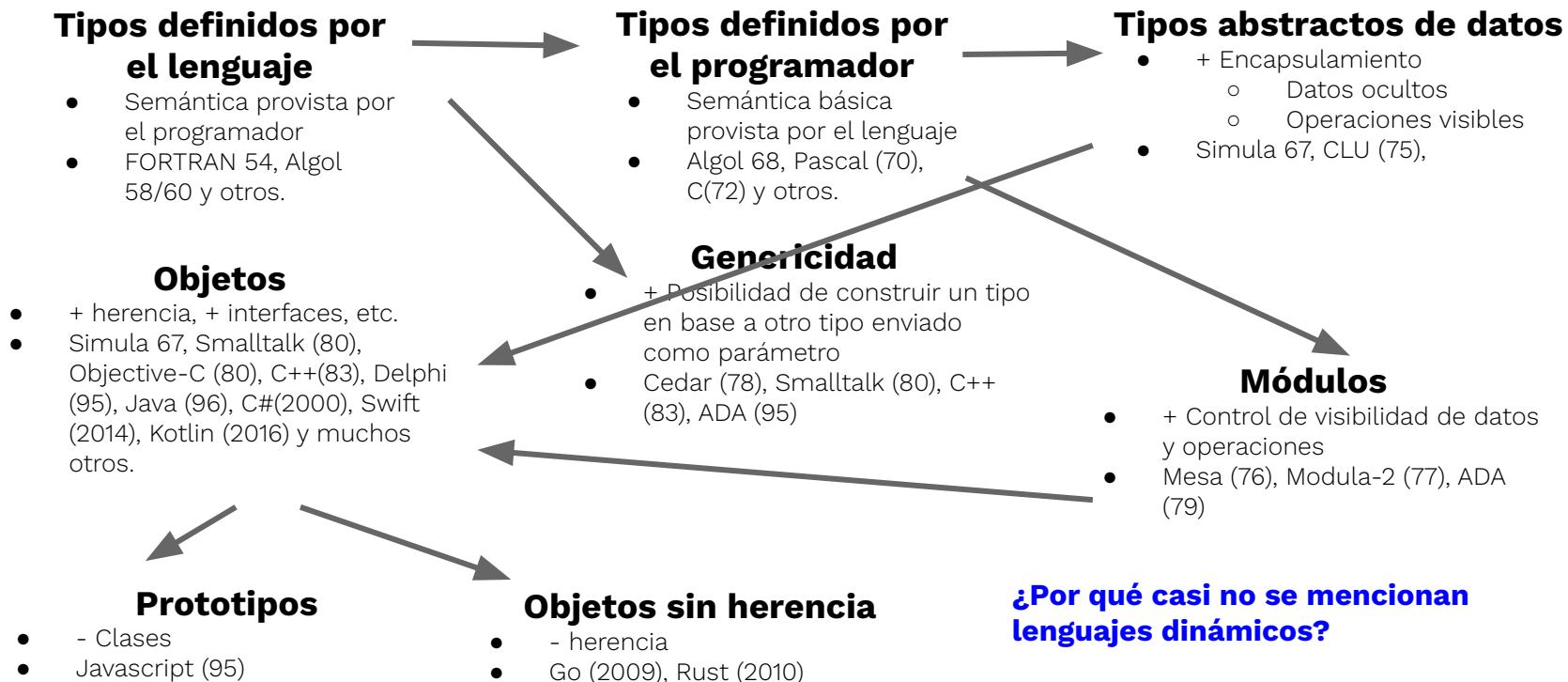
- **Introducción**
- **Sintaxis**
- **Semántica - Binding, Clasificación de lenguajes**
- **Alcance dinámico y estático**
- **Pila de ejecución**
- **Clasificación de variables en lenguajes de pila**
- **Manejo dinámico de memoria - Garbage Collection**
- **Interacción con el Sistema Operativo - concurrencia**
- **Pasaje de parámetros**
- **Asignaciones**
- **Tipos - Compatibilidades**
- **Tipos - Conversiones**
- **Objetos**

# Objetos



- Introducción y Evolución histórica (Tipos Abstractos de Datos, Módulos, Objetos)
- Almacenamiento
  - En lenguajes de pila
  - En lenguajes dinámicos
- Alcance
  - Control de visibilidad
  - Alcance por uso
- Métodos
  - Tablas de despacho
  - Polimorfismo
- Herencia
- Relación con compatibilidad de tipos

# Objetos - Introducción



# Objetos

- Abstracción
    - Atributos (campos, componentes, propiedades, etc.)
    - Operaciones asociadas (métodos, etc.)
    - El programador puede definir clases (tipo)
  - + Encapsulamiento - control de visibilidad
  - + Polimorfismo
  - + Herencia
  - - Clases
  - + Prototipos
  - - Herencia
  - + Composición
- 
- The diagram illustrates the relationships between various concepts in object-oriented programming:
- Tipos abstractos de datos** (Abstract Data Types) is connected to **Módulos** (Modules).
  - Módulos** is connected to **Objetos** (Objects).
  - Objetos** is connected to **Instancias/Prototipos** (Instances/Prototypes).
  - Instancias/Prototipos** is connected to **Objetos sin herencia** (Objects without inheritance).
  - Objetos sin herencia** is connected back to **Objetos**.

```
class A {  
    public int x;  
    private int y;  
    public void m() {  
        x=3;  
    }  
    public void m(int y) {  
        x=y;  
    }  
}  
class B : A {  
    public int x;  
    public int z;  
    public void m() {...}  
}  
  
void f() {  
    A a1;  
    a1.x = 3;  
    a1.m();  
    B b1;  
    b1.m();  
    a1 = b1;  
    a1.m();  
}
```

# Objetos: almacenamiento

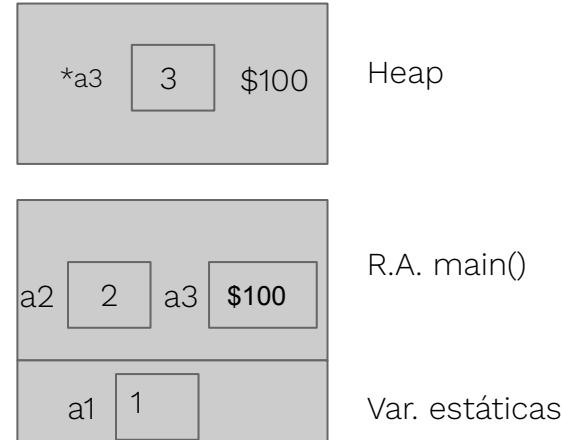
C++  
[almacenamiento1.cpp](#)

```
class A {  
    int x;  
}  
static A a1;  
int main() {  
    A a2;  
    A *a3 = new A();  
    a1.x = 1;  
    a2.x = 2;  
    a3->x = 3;  
}
```

¿Cuántos objetos hay?  
¿dónde se almacenan?

- a1 es un objeto (variable estática)
- a2 es un objeto (variable semi-estática)
- a3 es una variable semi-estática
- objeto anónimo en el heap

¿Cuántos dereferencing hay en  
esta instrucción?



**Los objetos son variables, se clasifican de acuerdo al  
almacenamiento de la misma manera**

Algunos compiladores no aceptan objetos con atributos semi-dinámicos

¿Conceptualmente hay algo que lo prohíba?

# Objetos: almacenamiento interno

## C++ almacenamiento2.cpp

```
class A {  
public: int x;  
        int y;  
};  
  
int main() {  
    static A a1;  
    A a2;  
    A *a3 = new A();  
    a1.x = 1;  
    a1.y = 2;  
    a2.x = 3;  
    a2.y = 4;  
    a3->x = 5;  
    a3->y = 6;  
}
```

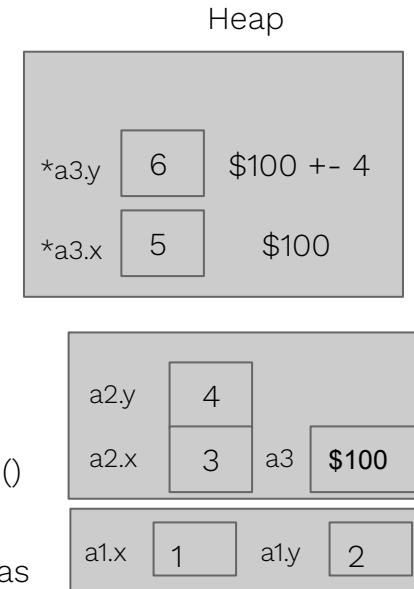
¿Qué sucede si hay más de un atributo en el objeto?

cada uno tendrá su offset relativo a donde se encuentre el objeto

```
movl $3, -8(%rbp)  
movl $4, -4(%rbp)
```

Independientemente de dónde esté almacenado el objeto:

Los atributos de un objeto se almacenan de la misma manera que las estructuras/registros  
(Class Instance Record)



# Objetos: almacenamiento interno estático

## C++ [almacenamiento3.cpp](#)

```
class A {  
public: int x;  
static int y;  
};  
int A::y = 1;
```

```
int main() {  
    A a2;  
    A *a3 = new A();  
    static A a1;  
    a1.x = 2;  
    a2.x = 3;  
    a3->x = 4;  
    a1.y = 5;  
    a2.y = 6;  
    a3->y = 7;  
}
```

¿Qué sucede si dentro del objeto tenemos diferente almacenamiento?

y es un atributo estático  
¿Existe un atributo estático por cada objeto de la clase A o todos comparten el mismo?  
¿Qué consecuencias semánticas trae esto?

se asigna 1 al atributo y compartido por todos los objetos de A

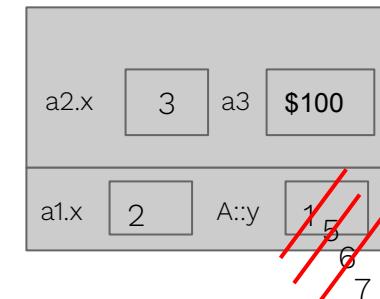
estas instrucciones alteran la misma celda de memoria

Todo el objeto o algunos atributos del mismo pueden ser estáticos

Heap



R.A.  
main()  
Var.  
estáticas



# Objetos: herencia

## C++ herencia.cpp.

```
class A {  
    public: int x;  
            int y;  
};  
class B : public A {  
    public: int y;  
            int z;  
};  
int main() {  
    A a1;  
    B b1;  
    b1.x = 1;  
    b1.y = 2;  
    b1.z = 3;  
}
```

¿Cómo se almacena un objeto que pertenece a una clase que hereda de otra?

La clase B hereda los atributos y métodos de A

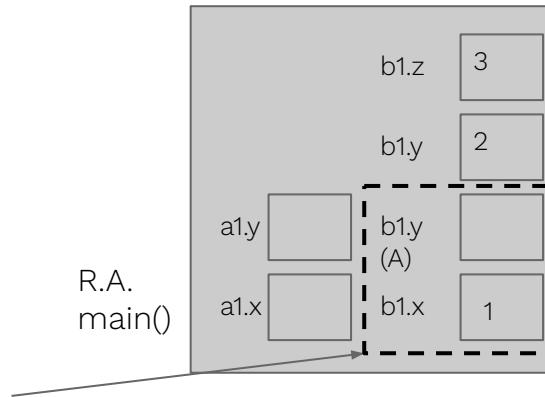
se sobreescribe el atributo y

```
movl $1, -32(%rbp)  
movl $2, -24(%rbp)  
movl $3, -20(%rbp)
```

¿Se observa algo extraño?

Todos los objetos tienen sus atributos y los heredados (aún los sobreescritos)

¿Por qué?



La herencia es una relación de tipos “es un”

Un objeto de la clase B debe poder participar del lado derecho como un A

# Objetos: Herencia y sobreescritura de atributos

## Java [herencia.java](#)

```
class A {  
    public static int w =1;  
    public int x =2;  
    public int y =3;  
}  
  
class B extends A {  
    public int y =4;  
    public int z =5;  
}  
... public static void main(String[] args) throws Exception{  
    A a1 = new A();  
    B b1 = new B();  
  
    a1 = b1;  
    b1.x = 7;  
    b1.y = 8;  
    b1.z = 9;  
    a1.x = 10;  
    a1.y = 11;  
}
```

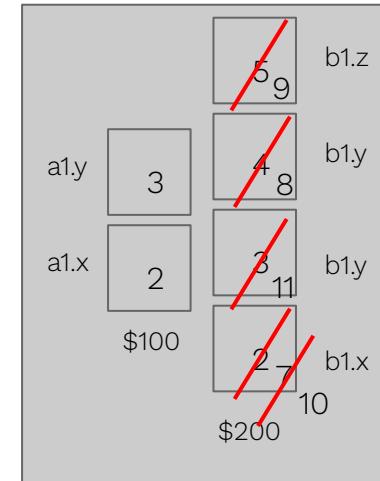
¿Cuántos atributos tiene b1?

¿Es posible realizar a1 = b1;?

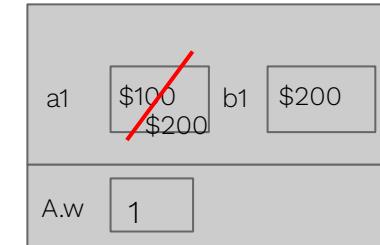
```
16: aload_2  
17: astore_1
```

Cada instrucción está compilada para acceder al offset correspondiente al tipo que tiene el objeto

Heap



R.A.  
main()



Var.  
estáticas

# Objetos: Herencia y sobreescritura de atributos

## Java [herencia2.java](#)

```
class A {  
    public static int w =1;  
    public int x =2;  
    public int y =3;  
}  
  
class B extends A {  
    public int y =4;  
    public int z =5;  
}  
... public static void main(String[] args) throws Exception{  
    A a1 = new A();  
    B b1 = new B();  
  
    b1 = a1; (B) a1;  
}
```

¿Es posible realizar `b1 = a1;`?

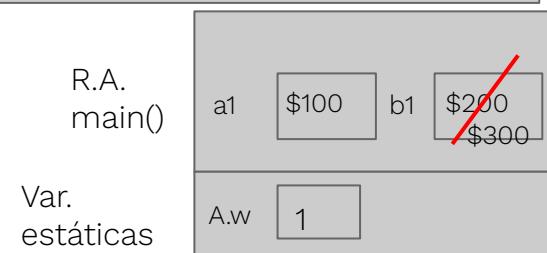
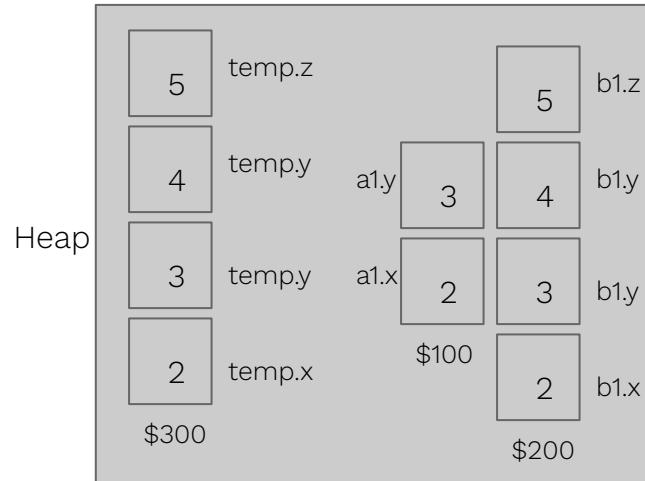
¿Cómo se soluciona?

¿Qué significa hacer eso?

```
18: aload_1  
19: checkcast    #4 //  
class B  
22: astore_2
```

Para que se pueda hacer es necesario proveer una función que realice la conversión explícita.

Si bien Java es un lenguaje compilado, el Run Time System de Java tiene un módulo cargador de clases donde podría en tiempo de ejecución cargarse una clase que contenga la conversión.



# Objetos: Herencia y sobreescritura de atributos

## C++ [herencia3.cpp](#)

```
class A {  
    public int x = 1;  
    public int y = 2;  
}  
  
class B extends A {  
    public int y=3;  
    public int z =4;  
}  
  
int main() {  
    A a1;  
    B b1;  
    b1 = a1;    (B) a1;  
  
    A *pa1 = new A();  
    B *pb1 = new B();  
    pa1->x = 5;  
    pa1->y = 6;  
    pb1 = pa1;  (B*) pa1;  
    pb1->y = 7;  
  
    pa1 = (A*) pb1;  
}
```

¿Es posible realizar `b1 = a1;`?

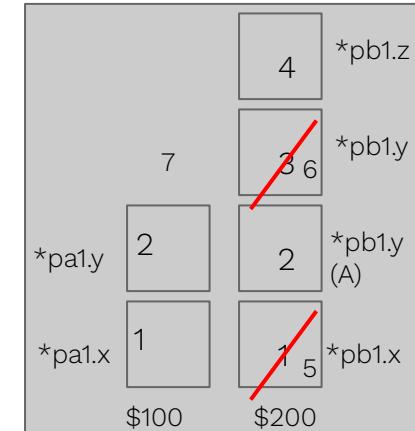
¿Es posible realizar `b1 = (B) a1;`?

¿Qué sucede si tenemos punteros a objetos?

¿Es posible hacer `pb1 = pa1;`?

¿Es posible hacer `pa1 = pb1;`?

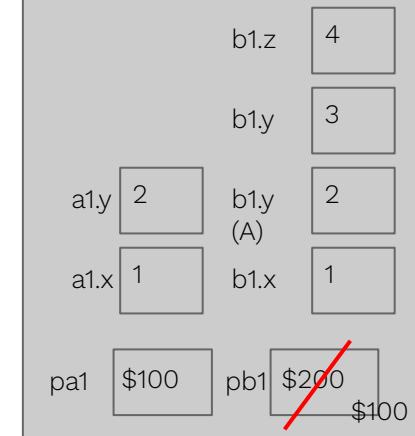
Heap



R.A.  
main()

```
mov rax, QWORD PTR [rbp-24]  
mov QWORD PTR [rbp-32], rax
```

```
mov rax, QWORD PTR [rbp-32]  
mov QWORD PTR [rbp-24], rax
```



# Objetos: Alcance de atributos y métodos

## C++ alcance.cpp

```
class A {  
    private: int x=1;  
    protected: int y=2;  
};  
class B : public A {  
public: int z=3;  
void m() {  
    x = 4;  
    y = 5;  
}  
};  
int main() {  
    A a1;  
    B b1;  
b1.x = 6;  
b1.y = 7;  
    b1.z = 8;  
}
```

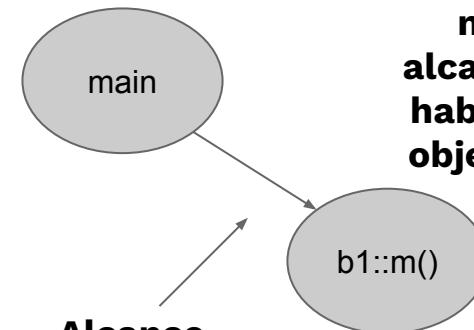
¿Qué aspecto semántico se relaciona a private, protected y public?

¿Cómo es el árbol de anidamiento de este programa?

No está al alcance por ser privada

Está al alcance por ser protected

No está al alcance por ser protected



**m de B está al alcance de Main por haber declarado un objeto de la clase B**

**Una unidad que declara un objeto de una determinada clase tiene alcance “por uso” a todos los atributos y métodos públicos de la clase.**

# Objetos: Invocación a métodos

## C++ metodos.cpp

```
class A {  
public: int x=1; “void m(this);”  
    void m() {  
        x=2;  
    }  
};  
  
int main() {  
    A a1;  
    a1.m();  
    “m(a1);”  
}
```

```
lea rax, [rbp-16]  
mov rdi, rax  
call A::m()
```

¿cómo hace el main para conocer donde está el método m() de la clase A?

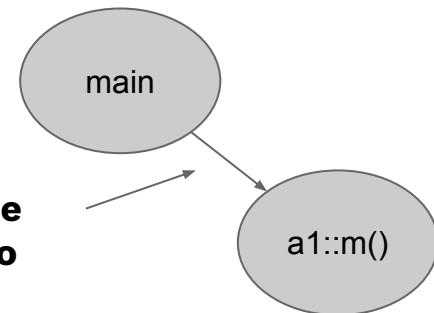
¿cómo hace el método para acceder a los atributos del objeto?

Recibe el objeto como parámetro

¿El pasaje del objeto es por copia valor o por referencia?

Los métodos deben poder acceder al objeto desde el cual se llamó y deben poder modificarlo

Alcance por uso



R.A.  
m()  
(a1)



R.A.  
main()



# Objetos: Invocación a métodos

Reciben implícitamente el objeto como parámetro por referencia

## Java [metodos.java](#)

```
class A {  
    public int x=1;  
    public void m() {  
        x = 2;  
    }  
};  
  
public class metodos {  
    public static void main(String[] args) {  
        A a1 = new A();  
        a1.m();  
    }  
}
```

## C# [metodos.cs](#)

```
class A {  
    public int x=1;  
    public void m() {  
        x = 2;  
    }  
};  
  
public class metodos {  
    public static void main(String[] args) {  
        A a1 = new A();  
        a1.m();  
    }  
}
```

## Swift [metodos.swift](#)

```
class A  
class A {  
    var x: Int  
    func m() {  
        x = 1  
    }  
}  
  
var a1 = A()  
a1.m()
```

## Python [metodos.py](#)

```
class A:  
    x = 1  
    def m(self):  
        self.x = 2  
a1 = A()  
a1.m()
```

Se debe especificar el parámetro explícitamente

**Envía como parámetro el objeto a1 por referencia**

**¿Qué ventajas ofrece ?**

**El programador es consciente del uso de los atributos del objeto.**

# Objetos: Invocación a métodos

## C++ metodosherencia.cpp

```
#include <iostream>
void f() {...}
class A {
    public: int x=1;
virtual void m() {
    printf("%d\n",x);
}
class B : public A {
    public: void m() {
        x = 2;
        printf("%d\n",x);
    }
};
int main() {
    A *pa1 = new A();
    B *pb1 = new B();
    f();
    pa1->m();
    pa1 = pb1;
    pa1->m();
}
```

El binding de esta llamada es estático

call f()

call A::m()

**Si los métodos son virtuales, entonces el binding de invocación debe ser dinámico (late binding call).**

**¿Dónde se almacena la información de cuál método se debe invocar?**

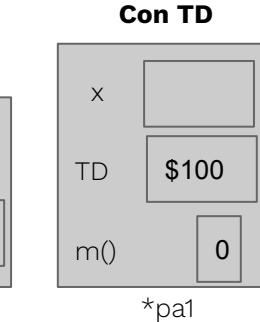
**Si se usa Tabla de Despacho**

### objeto

- Puntero a la tabla de despacho
- Offset de método

### Tabla de despacho

- Puntero a la dirección del método



\*pa1

\*pa1

**TD (A)** \$100

**TD (B)** \$200

A::m() \$32654

A:: otros métodos de A

B::m() \$36320

B:: otros métodos de B

**¿Dónde se encuentra realmente el código del método?**

\$32654

\$36320

m(): mov ...  
...  
m(): mov ..  
...

Programa ejecutable

# Objetos: Invocación a métodos

## Java

### [metodosherrencia.java](#)

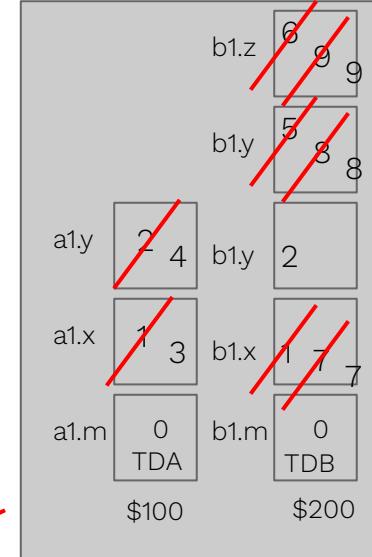
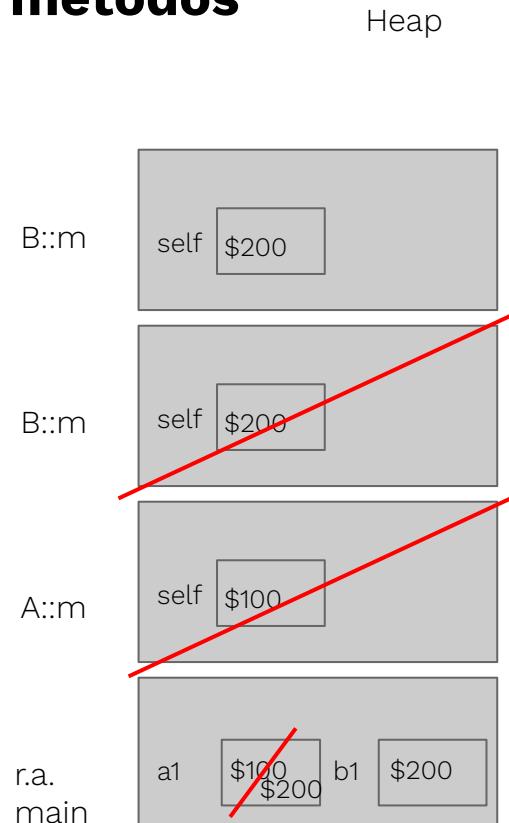
```
class A {  
    public  
        int x=1;  
        int y=2;  
        void m() {  
            x = 3;  
            y = 4;  
        }  
};  
  
class B extends A {  
    public  
        int y=5;  
        int z=6;  
        void m() {  
            x = 7;  
            y = 8;  
            z = 9;  
        }  
};
```

```
public class alcance {  
    public static void  
    main(String[] args) {  
        A a1 = new A();  
        B b1 = new B();  
        a1.m();  
        b1.m();  
        a1 = b1;  
        a1.m();  
    }  
}
```

¿A cuál método m() se llama?

En Java los métodos son virtuales por defecto

¿Qué sucede si se declaran los métodos como static?



# Objetos: Lenguajes dinámicos

## Python

[metodosherencia.py](#)

```
class A:  
    x = 1  
    y = 2  
    def m(self):  
        self.x = 3  
        self.y = 4  
class B(A):  
    y = 5  
    z = 6  
    def m(self):  
        self.x = 7  
        self.y = 8  
        self.z = 9  
  
a1 = A()  
b1 = B()  
a1.m()  
b1.m()  
a1 = b1  
a1.m()  
print(a1.y)
```

a1 cambia de tipo

esta instrucción accede al objeto como si fuese de tipo B

**En Lenguajes con tipos dinámicos no hay ningún inconveniente, ya que la variable cambia efectivamente de tipo**

Heap

## Tabla de Símbolos

B::m



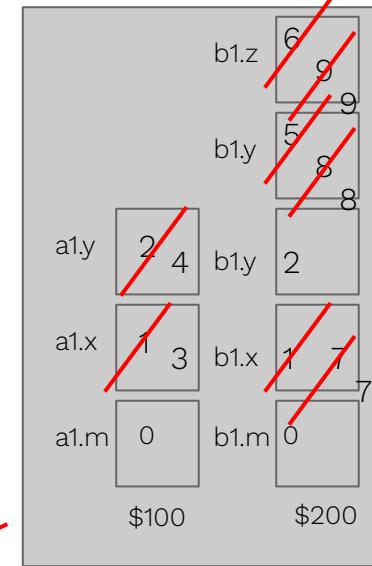
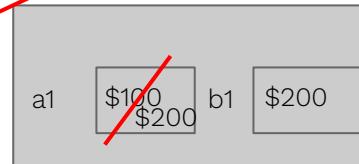
B::m



A::m



main



# Objetos: Lenguajes sin herencia

## GO [objetos.go](#)

```
type A struct {  
    x float32;  
    y int  
}  
  
type B struct {  
    x int  
    z int  
    A  
}  
  
func (e *A) m() {  
    e.x = 3  
    e.y = 4  
    fmt.Printf("%v",e.x)  
    fmt.Printf("%v",e.y)  
}
```

Los métodos se definen como funciones

¿A cuál lenguaje es similar sintácticamente la declaración de métodos?

Se definen las clases como de tipo estructuras que contienen los atributos

La “herencia” se implementa como una composición de clases

atributo anónimo

```
func main() {  
    var a1 = A{}  
    a1.x = 1.2;  
    a1.m()  
    var b1 = B{}  
    b1.x = 2;  
    b1.y = 3;  
    fmt.Printf("%v",b1.x)  
    fmt.Printf("%v",b1.y)  
    fmt.Printf("%v",b1.z)  
}
```

Se invoca al método pasando el objeto como referencia

Se utilizan los atributos de la clase A a través del atributo anónimo de B

¿Esta semántica de herencia es realmente diferente a la semántica de herencia de Java/C++ en cuanto al almacenamiento?

¿Cuál ventaja tiene separar sintácticamente los métodos de las clases?

# **Relación con compatibilidad de tipos**

- **Las clases son tipos**
- **Compatibilidad por nombre**
- **Herencia -> Subtipos de ADA**
- **Duck Typing (el tipo lo determina el contenido del objeto en un determinado momento) -> Lenguajes con tipos dinámicos**
- **Métodos virtuales (Algunos aspectos de Duck Typing a través de late binding)**
- **Herencia múltiple (Kotlin)**
- **Interfaces / Traits**
- **Lenguajes basados en instancias/prototipos**

# ¿Cuál es el mejor lenguaje?

## ¿Para qué?

- **Arquitectura**
  - Back-end (Java, C/C++, Go, Python, ...)
  - Front-end (Javascript, Typescript, Dart, Pyscript, ...)
- **Curva de aprendizaje - Costo de desarrollo**
  - C/C++ vs. Python
- **Comunidad de desarrolladores**
  - Javascript (17.4 M) vs. Dart (1.8 M)
- **Seguridad**
  - C++ vs. Rust (Borrowing, Ownership), ADA (compatibilidad)
- **Escalabilidad**
  - C++, Java, Ruby vs. Python, Javascript
- **Retrocompatibilidad**
  - C vs. Python
- **Portabilidad**
  - Java vs. C
- **Ámbito de aplicación**
  - Software de Base (C, Rust vs. Java)
  - Gestión (Java vs. Perl)
  - Finanzas (COBOL, ADA, Solidity vs. C/C++)
  - Data Science (C, Python vs. PHP)
  - ...
- ...

# Tendencias en Lenguajes de Programación

- **Sintaxis simplificada** (inferencia de tipos, iteradores, variables implícitas, etc.)
- **Unidades de primera clase** (lambdas, closures, blocks)
- **Objetos simplificados** (Prototipos e instancias)
- **Green Threads** (Corrutinas)
- **Canales** (Comunicación entre hilos y corrutinas)
- **Sound Typing** (Borrowing, Ownership, Lifetime analysis)
- **Type Checkers** (Anotaciones y Verificación de tipos en lenguajes interpretados)
- **Inmutabilidad** (La eventual modificación de una variable produce otra variable)
- **Programación Funcional**
  - High Order Functions
  - Algebraic data types
  - Pattern matching
  - Nullable Types
  - Reflexividad
- ...

# Fuentes de actualización



<https://www.sigplan.org/>



PL Perspectives

<https://blog.sigplan.org/>



<https://www.youtube.com/c/ContextFree>  
<https://www.youtube.com/c/CodingTech>  
<https://twitter.com/CppCon>  
<https://twitter.com/ahejlsberg>  
[https://twitter.com/michael\\_w\\_hicks](https://twitter.com/michael_w_hicks)



Rust Blog



Q#



Bosque

Carbon®