



# SEMANTICA OPERACIONAL

## **Repaso Clase Anterior**

# REPASO CLASE ANTERIOR

- Definición de Semántica
  - Semántica Estática
    - Formal: Gramática de Atributos
  - Semántica Dinámica
    - Formal: Semántica Axiomática – Semántica Denotacional
    - No Formal: Semántica Operacional
- Procesamiento de los lenguajes
  - Traductores
    - Intérpretes
    - Compiladores
  - Proceso del compilador
    - Análisis: Léxico, Sintáctico, Semántica Estática
    - Síntesis: Optimización, Generación del código





# SEMANTICA OPERACIONAL VARIABLE

# SEMÁNTICA DE LOS LENGUAJES DE PROGRAMACIÓN

## ENTIDAD

- Variable
- Rutina
- Sentencia

## ATRIBUTO

- nombre, tipo, área de memoria, etc
- nombre, parámetros formales, parámetros reales, etc
- acción asociada

**DESCRIPTOR:** lugar donde se almacenan los atributos



# CONCEPTO DE LIGADURA (BINDING)

Los programas trabajan con **entidades**



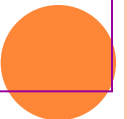
Las entidades tienen **atributos**



Estos atributos tienen que establecerse antes de poder usar la entidad



**LIGADURA:** es la asociación entre la entidad y el atributo



# LIGADURA

## Diferencias entre los lenguajes de programación

- El número de **entidades**
- El número de **atributos** que se les pueden ligar
- El **momento** en que se hacen las ligaduras (**binding time**).
- La **estabilidad** de la ligadura: una vez establecida se puede modificar?



# MOMENTO DE LIGADURA

- Definición del lenguaje
- Implementación del lenguaje
- Compilación (procesamiento)
- Ejecución



E  
S  
T  
A  
T  
I  
C  
O



D  
I  
N  
A  
M  
I  
C  
O



# MOMENTO Y ESTABILIDAD

- Una **ligadura es estática** si se establece antes de la ejecución y no se puede cambiar. El termino estático referencia al momento del binding y a su estabilidad.
- Una **ligadura es dinámica** si se establece en el momento de la ejecución y puede cambiarse de acuerdo a alguna regla especifica del lenguaje.

Excepción: constantes





# MOMENTO Y ESTABILIDAD

## Ejemplos:

### ○ En **Definición**

- Forma de las sentencias
- Estructura del programa
- Nombres de los tipos predefinidos

### ○ En **Implementación**

- Representación de los números y sus operaciones

### ○ En **Compilación**

- Asignación del tipo a las variables

## En lenguaje C

*int*

Para denominar a los enteros

*int*

- Representación
- Operaciones que pueden realizarse sobre ellos

*int a*

- Se liga tipo a la variable



## ○ En Ejecución

- Variables con sus valores
- Variables con su lugar de almacenamiento

*int a*

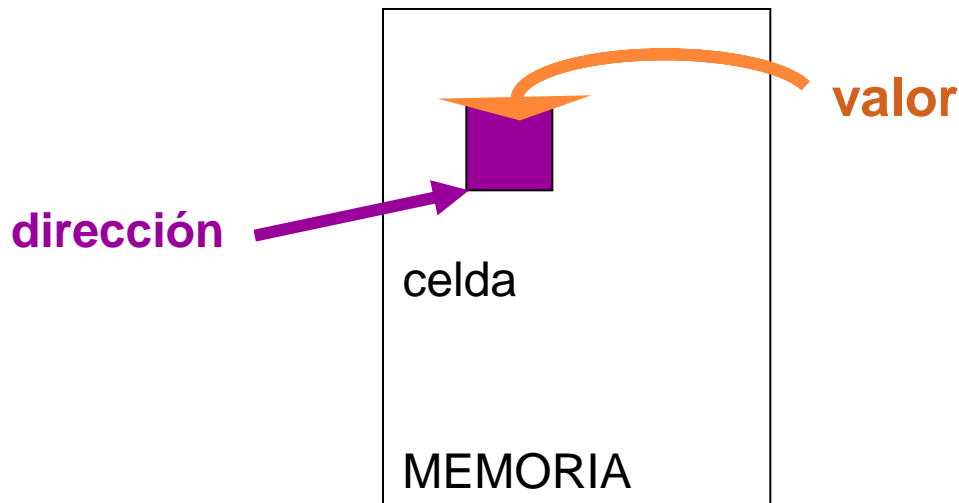
- el valor de una variable entera se liga en ejecución y puede cambiarse muchas veces.



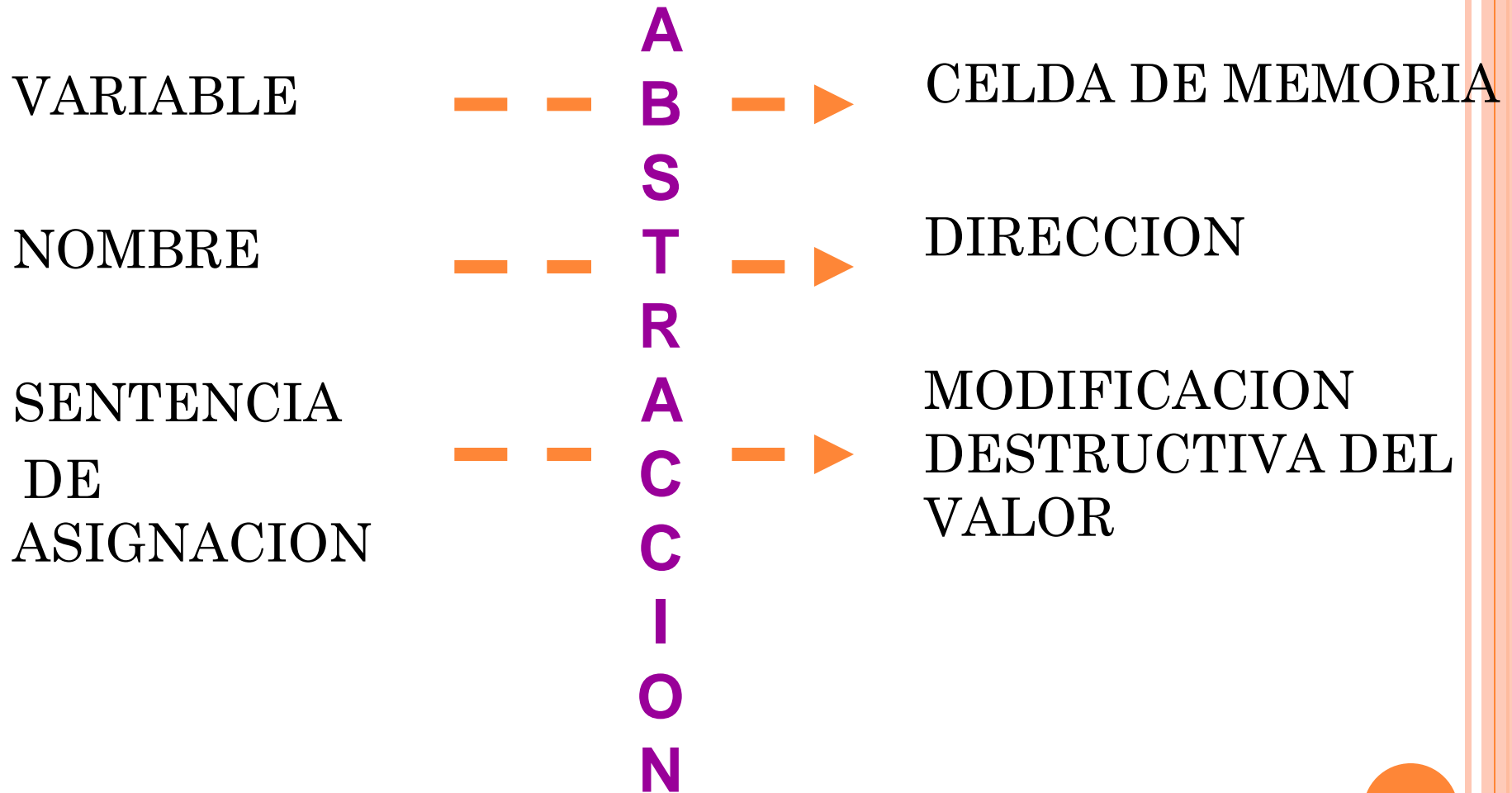
# VARIABLES

## CONCEPTO

- **Memoria principal: celdas** elementales, identificadas por una **dirección**.
- El contenido de una celda es una **representación codificada de un valor**



# VARIABLE



# <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

- ***Nombre:*** string de caracteres que se usa para referenciar a la variable. (**identificador**)
- ***Alcance:*** es el rango de instrucciones en el que se conoce el nombre
- ***Tipo:*** valores y operaciones
- ***L-value:*** es el lugar de memoria asociado con la variable (**tiempo de vida**)
- ***R-value:*** es el valor codificado almacenado en la ubicación de la variable



# <NOMBRE, ALCANCE ,TIPO, L-VALUE, R-VALUE>

## Aspectos de diseño:

- Longitud máxima

Algunos ejemplos: **Fortran**:6 **Python**: sin límite

**C**: depende del compilador, suele ser de 32 y se ignora si hay más

**Pascal, Java, ADA**: cualquier longitud

- Caracteres aceptados (**conectores**)

Ejemplo: Python, C, Pascal: \_

Ruby: solo letras minúsculas para variables locales

- Sensitivos

Sum = sum = SUM ?

Ejemplos: C y Python sensibles a mayúsculas y minúsculas

Pascal no sensible a mayúsculas y minúsculas

**palabra reservada - palabra clave**



## <NOMBRE, ALCANCE ,TIPO, L-VALUE, R-VALUE>

- El **alcance** de una variable es el rango de instrucciones en el que se conoce el nombre. **(visibilidad)**
- Las instrucciones del programa pueden **manipular una variable** a través de su nombre **dentro de su alcance**
- Los diferentes lenguajes adoptan diferentes reglas para ligar un nombre a su alcance.




# <NOMBRE, ALCANCE ,TIPO, L-VALUE, R-VALUE>

## ○ Alcance estático

- Llamado **alcance léxico**.
- Define el alcance en términos de la estructura léxica del programa.
- Puede ligarse estáticamente a una declaración (explícita o implícita) examinando el texto del programa, sin necesidad de ejecutarlo.
- La mayoría de los lenguajes adoptan reglas de ligadura de alcance estático.

## ○ Alcance dinámico

- Define el alcance del nombre de la variable en términos de la ejecución del programa.
  - Cada declaración de variable extiende su efecto sobre todas las instrucciones ejecutadas posteriormente, hasta que una nueva declaración para una variable con el mismo nombre es encontrado durante la ejecución.
  - **APL**, **Lisp** (original), **Afnix** (llamado *Aleph* hasta el 2003), **Tcl** (Tool Command Language), **Perl**
- 



```

int x;
{
  /*bloque A*/
  int x;
}

{
  /*bloque B*/
  int x;
}

{
  /*bloque C*/
  x = ...;
  ...
}
....

```

## Ejecución:

- Con **alcance Dinámico**,  
si:

A C  
x de A

B C  
x de B

- Con **alcance Estático**  
en ambos casos hace  
referencia a **x externa**

Dinámico: menos legible



# PASCAL - LIKE

```
1 Program Alcance;
2   var
3       a : Integer;
4       z , b: Real;
5   procedure uno();
6   var
7       b: Integer;
8   procedure dos();
9       begin
10          z:=a+1+b;
11      end;
12   begin
13       b:= 20;      dos();
14   end;
15   procedure tres();
16   var
17       a: Real;
18   begin
19       a:=20;      uno();
20   end;
21 Begin
22 a:= 4;      b:= 2;      z:=10;      tres();
23 end.
```

Ejecución:

## ○ Alcance estático:

Al invocar a *tres*:

- Se invoca a **uno**
- Se invoca a **dos** y  
 $z := a + 1 + b;$

Toca a **z** de **Alcance**

La variable **a** es la de **Alcance** y la variable **b** es de **uno**

## ○ Alcance dinámico:

Al invocar a *tres*:

- Se invoca a **uno** y
- Se invoca a **dos** y  
 $z := a + 1 + b;$

Toca a **z** de **Alcance**

La variable **a** es la de **tres**

La variable **b** es la de **uno**

# ALCANCE EN C - ESTÁTICO

compileonline.com - Compile and Execute C Online (GNU GCC version 4.8.1)

Compile & Execute main.c input.txt Default Ace Editor Multiple Files

```
1 #include <stdio.h>
2 int x;
3 int y;
4
5 void uno()
6 {
7     printf ("\n EN uno \n");
8     x= x+y;
9     printf ("x en uno= %d \n", x);
10    printf ("e y en uno= %d\n", y);
11 }
12
13 void main()
14 {
15     x=1;
16     y=1;
17     printf (" ANTES de entrar al bloque \n");
18     printf ("x en main= %d\n", x);
19     printf ("y en main= %d\n", y);
20
21     {
22         printf ("\n EN el bloque \n");
23         int x;
24         x=10;
25         x=x+y;
26         printf ("x en el bloque= %d\n", x);
27         printf ("y en bloque= %d\n", y);
28         uno ();
29     }
30
31     printf ("\n DESPUES de salir al bloque \n");
32     printf ("x en main= %d\n", x);
33     printf ("y en main= %d\n", y);
34 }
35
36
```

X Y X'

El alcance de un nombre se extiende desde su declaración hacia los bloques anidados a menos que aparezca otra declaración para el nombre

# ALCANCE EN PASCAL - ESTÁTICO

compileonline.com - Compile and Execute Pascal Online (fpc 2.6.2)

Compile & Execute

Main Program

input.txt

Default Ace Editor

☐ Unit Support

```
1 Program Alcance;
2
3 var
4   x: integer;
5   y: integer;
6
7 procedure uno();
8 begin
9   x:= x+y;
10  writeln('"x" en uno= ', x, ' e "y" en uno= ', y);
11 end;
12
13 procedure dos();
14 var x:integer;
15
16 procedure tres();
17 begin
18   x:=x+10;
19   writeln('"x" en tres= ', x, ' e "y" en tres= ', y);
20   uno();
21 end;
22 begin
23   x:=10;
24   tres();
25 end;
26
27
28 begin
29   x:=1;
30   y:=1;
31   writeln('"x" en main= ', x, ' e "y" en main= ', y, ' ANTES de llamar a procedimiento dos');
32   dos();
33   writeln('"x" en main= ', x, ' e "y" en main= ', y, ' DESPUES de llamar a procedimiento dos');
34 end.
35
```

# ALCANCE EN ADA - ESTÁTICO

Compile | Execute hello.adb x

```
1 with Text_IO, Ada.Integer_Text_IO;
2 use Text_IO, Ada.Integer_Text_IO;
3
4 procedure Principal is
5   y: integer;
6   procedure Prueba is
7     x: constant integer := 3+y;
8     y: integer:=4;
9     begin
10      Put("El valor de la constante x es:");
11      Put(x);
12      Put("    El valor de la variable y es:");
13      Put(y);
14    end Prueba;
15
16
17 begin
18
19   y:=7;
20   Prueba;
21
22 end Princip
```



Demuestra que el alcance es de dónde se declara hacia abajo

Terminal

```
gcc -c hello.adb
hello.adb:4:11: warning: file name does not match unit name, should be "principal.adb"
gnatbind -x hello.ali
gnatlink hello.ali -o hello
sh-4.2# hello
El valor de la constante x es: 10 El valor de la variable y es: 4
sh-4.2#
```

# ALCANCE EN PYTHON - ESTÁTICO

```
1 def alcance1():
2     print x+ ' Juan'
3
4
5 def alcance2():
6     x='Chau'
7     alcance1()
8
9 x='Hola'
10 alcance2()
11
12
```

El archivo «C:\Users\Viviana\Desktop\...

0 INS TAB mode: Unix (LF)

Demuestra que el alcance es estático. Por más que desde alcance1 se lo llame desde alcance2, la variable x tomada es la del programa principal

```
C:\Windows\system32\cmd.exe
Hola Juan
Presione una tecla para continuar . . .
```

# ALCANCE EN PYTHON

```
*Python 3.3.4: alcance-2.py - C:/Python33/alcance-2.py*
File Edit Format Run Options Windows Help

x=1
y=1

def uno():
    y=x+1
    print ('En uno')
    print ('y es= ',y, 'x es=', x)

def dos():
    x=y+10
    def tres():
        x=x+y
        print ('En tres')
        print ('y es= ',y, 'x es=', x)
        uno ()
    print ('En dos')
    print ('y es= ',y, 'x es=', x)
    tres()

print ('ANTES de llamar a dos')
print ('y en el principal es= ',y, 'x en el principal es=', x)
dos()
print ('DESPUES de llamar a dos')
print ('y en el principal es= ',y, 'x en el principal es=', x)
```

¿Por qué el error???

X Y Y' X' X''

Ln: 25 Col: 0



# ESTÁTICO VS DINÁMICO

- Las reglas dinámicas son mas fáciles de implementar
- Son menos claras en cuanto a disciplina de programación
- El código se hacen mas difícil de leer





# CONCEPTOS ASOCIADOS CON EL ALCANCE

- **Local:** Son todas la referencias que se han creado dentro del programa o subprograma.
- **No Local:** Son todas las referencias que se utilizan dentro del subprograma pero que no han sido creadas en él.
- **Global:** Son todas las referencias creadas en el programa principal



# CONCEPTOS ASOCIADOS CON EL ALCANCE

compileonline.com - Compile and Execute Pascal Online (fpc 2.6.2)

Compile & Execute Main Program input.txt Default Ace Editor Unit Support

```
1 Program Alcance;
2
3 var
4   x: integer;
5   y: integer;
6
7 procedure uno();
8 begin
9   x:= x+y;
10  writeln("x en uno= ", x, ' e "y" en uno= ', y);
11 end;
12
13 procedure dos();
14   var x:integer;
15
16   procedure tres();
17   begin
18     x:=x+10;
19     writeln("x en tres= ", x, ' e "y" en tres= ', y);
20     uno();
21   end;
22 begin
23   x:=10;
24   tres();
25 end;
26
27
28 begin
29   x:=1;
30   y:=1;
31   writeln("x en main= ", x, ' e "y" en main= ', y, ' ANTES de llamar a procedimiento dos');
32   dos();
33   writeln("x en main= ", x, ' e "y" en main= ', y, ' DESPUES de llamar a procedimiento dos');
34 end.
```

Referencia No Local

Referencia Global

Referencia Local

# ESPACIOS DE NOMBRES

## ○ Definición:

- Un espacio de nombre es una zona separada donde se pueden declarar y definir objetos, funciones y en general, cualquier identificador de tipo, clase, estructura, etc.; al que se asigna un nombre o identificador propio.

## ○ Utilidad:

- Ayudan a evitar problemas con identificadores con el mismo nombre en grandes proyectos o cuando se usan bibliotecas externas.



<NOMBRE, ALCANCE, TIPO, L-VALUE,  
R-VALUE>

○ **Definición:**

- **Conjunto de valores**
  - **Conjunto de las operaciones**
- **Antes de que una variable pueda ser referenciada debe ligársele un tipo**
- **Protege a las variables de operaciones no permitidas**

**Chequeo de tipos:** verifica el uso correcto de las variables



<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

- Predefinidos
  - Tipos base
- Definidos por el usuario
  - Constructores
- TADs



<NOMBRE, ALCANCE, TIPO, L-VALUE,  
R-VALUE>

## ○ Tipos predefinidos:

- Son los tipos base que están descriptos en la definición

### Tipo boolean

valores: *true*, *false*

operaciones: *and*, *or*, *not*

- Los valores se ligan en la implementación a representación de maquina

<i>true</i>	string	<i>000000.....1</i>
-------------	--------	---------------------

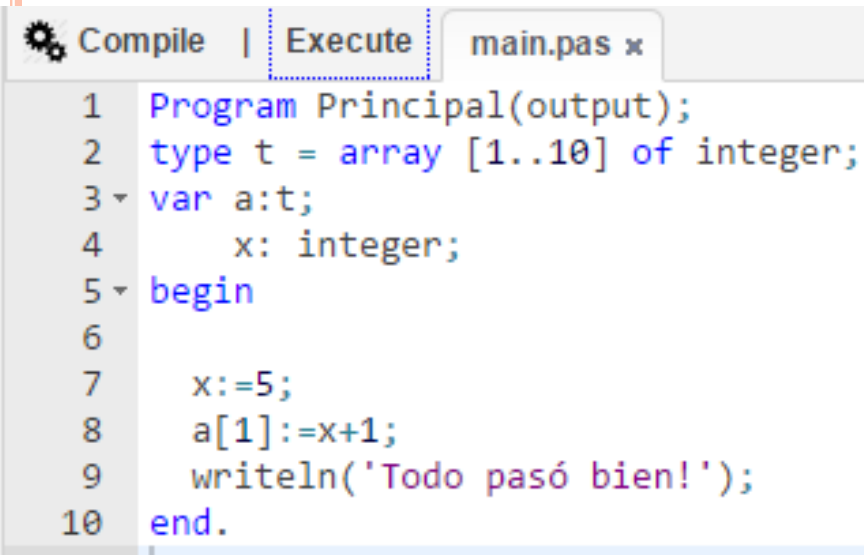
<i>false</i>	string	<i>0000.....000</i>
--------------	--------	---------------------



<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

## ○ Tipos definidos por el usuario:

- Los lenguajes permiten al programador mediante la declaración de tipos definir nuevos tipos a partir de los predefinidos y los constructores



```
1 Program Principal(output);
2 type t = array [1..10] of integer;
3 var a:t;
4     x: integer;
5 begin
6
7     x:=5;
8     a[1]:=x+1;
9     writeln('Todo pasó bien!');
10 end.
```

Se establece una ligadura (en traducción) del nombre del **tipo *t*** con el **arreglo de 10 enteros**

El **tipo *t*** tiene todas las operaciones de la estructura de datos (arreglo), y por lo tanto es posible leer y modificar cada componente de un objeto de tipo *t* indexando dentro del arreglo

<NOMBRE, ALCANCE, TIPO, L-VALUE,  
R-VALUE>

## ○ Tipos de Datos Abstractos:

- No hay ligadura por defecto, el programador debe **especificar la representación y las operaciones**

### TAD

- Estructura de datos que representan al nuevo tipo
- Rutinas usadas para manipular los objetos de este nuevo tipo





# TIPOS ABSTRACTOS (EJEMPLO EN C++)

*Estructura  
interna*

*Comportamiento  
(operaciones)*

```
#include<iostream>
#include<process.h>
#include<conio.h>
using namespace std;

class Clistpila
{
    protected:
        struct lista    // Estructura del Nodo de una lista
        {
            int dato;
            struct lista *nextPtr;    //siguiente elemento de la lista
        };

        typedef struct lista *NODELISTA;    //tipo de dato *NODOLISTA

        struct NodoPila
        {
            NODELISTA startPtr;    //tendrá la dirección del fondo de la pila
        } pila;

        typedef struct NodoPila *STACKNODE;    //Tipo Apuntador a la pila

    public:
        Clistpila( );    // Constructor
        ~Clistpila( );    // Destructor

        void push(int newvalue);    // Función que agrega un elemento a la pila
        int pop( );    // Función que saca un elemento de la pila
        int PilaVacía( );    // Verifica si la pila está vacía
        void MostrarPila( );    // Muestra los elementos de la Pila

        friend void opciones(void);    // función amiga
};

//Funciones Miembro de la clase
Clistpila :: Clistpila( )
{
    pila.startPtr = NULL;    //se inicializa el fondo de la pila.
}

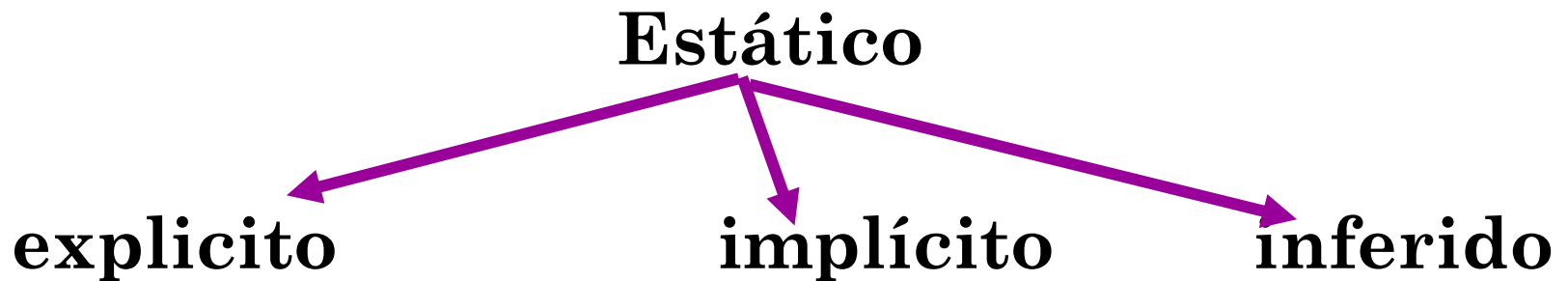
int Clistpila :: PilaVacía( )
{
    return((pila.startPtr == NULL)? 1:0);//note que si la pila esta vacía retorna 1, sino 0
}

void Clistpila :: push(int newvalue)    //se puede insertar en cualquier momento
{
    NODELISTA nuevoNodo;    //un nodo al tope de la pila
    nuevoNodo = new lista;    //crear el nuevo nodo
    if(nuevoNodo != NULL)    //si el espacio es disponible
        ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
```

<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

## Momentos - Estático

- El tipo se **liga en compilación y no puede ser cambiado**
  - El chequeo de tipo también será estático



Pascal, Algol, Simula, ADA, C, C++, Java, etc



<NOMBRE, ALCANCE, TIPO, L-VALUE,  
R-VALUE>

## ○ Momento – Estático - **Explícito**

- La ligadura se establece mediante una **declaración**

`int x, y`

`bool z`

`y := z`

illegal

`y := not y`

illegal



<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

## ○ Momento – Estático - **Implícito**

- La ligadura se deduce por **reglas**
- Ej. Fortran:
  - Si el nombre comienza con I a N es entera
  - Si el nombre comienza con letra A-H ó O- Z es real

**Semánticamente** la explícita y la implícita son equivalentes, con respecto al tipado de las variables, ambos son estáticos. El momento en que se hace la ligadura y su estabilidad es el mismo en los dos lenguajes.



<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

## ○ Momento – Estático - Inferido

- El tipo de una expresión se deduce de los tipos de sus componentes
- Lenguaje funcional. Ej. Lisp

Si se tiene en un script

**doble x = 2 \* x**

Si no está definido el tipo se infiere

**doble :: num -> num**



<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

## ○ Momento – **Dinámico**

- **El tipo se liga en ejecución y puede cambiarse**
  - Mas flexible: programación genérica
  - Mas costoso en ejecución: mantenimiento de descriptores
  - Variables polimórficas.
  - Chequeo dinámico
  - Menor legibilidad

APL, Snobol, Smalltalk, Python, Ruby, etc



<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

- **Área de memoria ligada a la variable**
- **Tiempo de vida (lifetime) o extensión:**

Periodo de tiempo que existe la ligadura

- **Alocación:**

Momento que se reservar la memoria

**El tiempo de vida es el tiempo en que la variable esté alocada en memoria**



<NOMBRE, ALCANCE, TIPO, L-VALUE,  
R-VALUE>

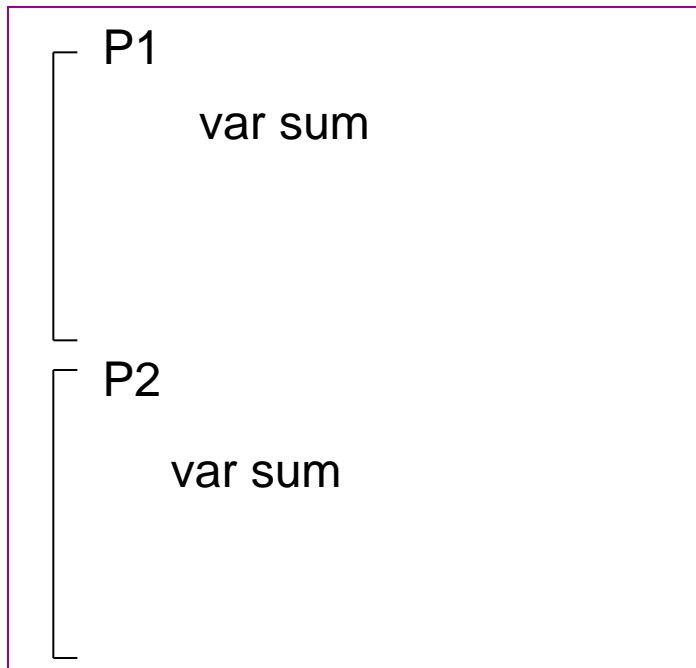
## Momentos - **Alocación**

- **Estática:** sensible a la historia
- **Dinámica**
  - Automática; cuando aparece la declaración
  - Explícita: a través de algún constructor
- **Persistente:** su tiempo de vida no depende de la ejecución:
  - existe en el ambiente
  - Archivos - Bases de datos



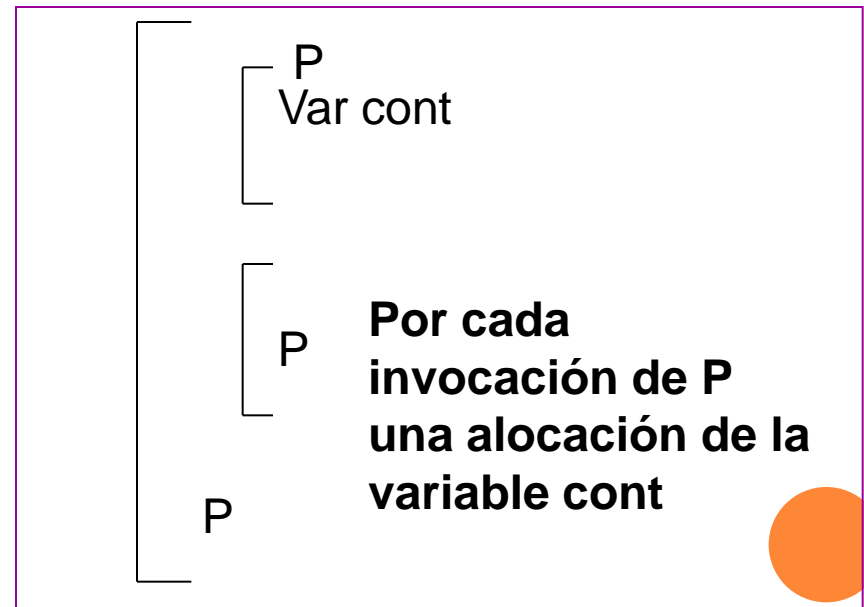
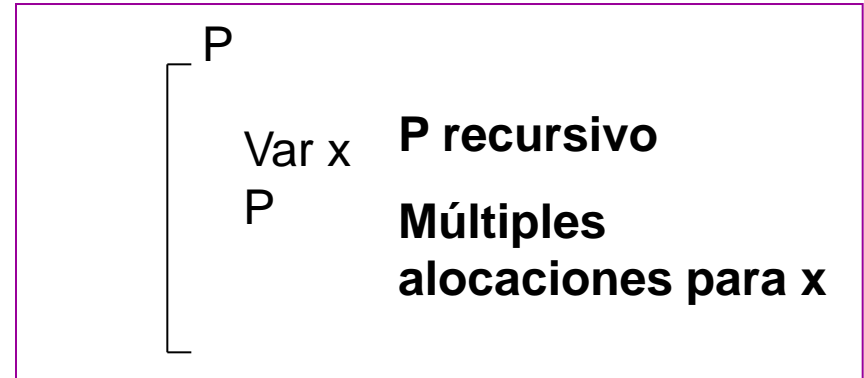


# EJEMPLOS DE ALOCACIONES



**Dos alocaiones  
diferentes para sum:**

- sum de P1 y
- sum de P2



## File 1

```
int x;
```

Variable **EXTERNA**

```
static int n;  
int func1()  
{
```

```
    extern int i;
```

```
    float z; ← Variable INTERNA
```

```
    static int m;
```

```
}
```

```
..
```

```
int main()  
{
```

```
    .....
```

```
}
```

## File 2

```
extern int x;
```

```
....
```

```
int i;
```

```
extern static int n;
```

```
float func2()  
{
```

```
    int x
```

```
    int z;
```

```
}
```

```
.....
```

Para hacer uso de la cláusula EXTERN, la variable debe estar definida previamente y debe ser externa

## Alcances:

Con la declaración “**extern**” se extendió el alcance a File 2.

Ahora x de File 1 tiene alcance en todo File 1 y en File 2, MENOS en func2

La nueva declaración

“**extern**” extiende el alcance de i del File 2 a la función func1 de File 1

Tiempo de vida: Todo el programa

Alcance: **SOLO** en la función dónde está definida

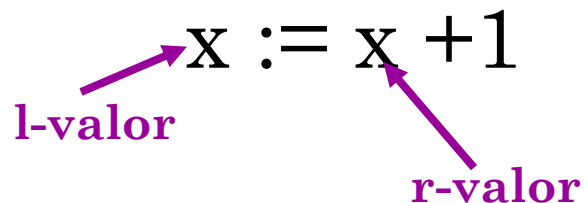
Si está definida fuera de la función el alcance es desde dónde está al final del archivo. NO se le puede extender el alcance hacia otro archivo

Tiempo de vida:

Desde que comienza el programa hasta que termina

# <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

- Valor almacenado en el l-valor de la variable
- Se interpreta de acuerdo al tipo de la variable
- Objeto: (l-valor, r-valor)

  
The diagram shows the assignment statement  $x := x + 1$ . A purple arrow labeled "l-valor" points to the variable  $x$  on the left side of the assignment. Another purple arrow labeled "r-valor" points to the variable  $x$  on the right side of the assignment, which is part of the expression  $x + 1$ .

Se accede a las variable a través del **l-valor**  
Se puede modificar el **r-value**



<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

## Momentos:

- **Dinámico:** por naturaleza

$b := a$  se copia el r-valor de  $a$  en el l-valor de  $b$   
 $a := 17$

- **Constantes:** se congela el valor

*const*  
*pi* = 3.1416

Pascal: estático  
Ada dinámico estable

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Inicializacion is
3    x: Integer:=4;
4    procedure Uno is
5      z: constant Integer := x+5;
6    begin
7      Put_Line("Estoy en uno");
8    end Uno;
9  begin
10   Uno;
11 end Inicializacion;
```

<NOMBRE, ALCANCE, TIPO, L-VALUE,  
R-VALUE>

## Inicialización

- ¿Cuál es el r-valor luego de crearse la variable?
  - Ignorar el problema: lo que haya en memoria
  - Estrategia de inicialización:
    - Inicialización por defecto:
      - Enteros se inicializan en 0, los caracteres en blanco, etc.
    - Inicialización en la declaración:

C   `int i =0, j= 1`

ADA   `I,J INTEGER:=0`

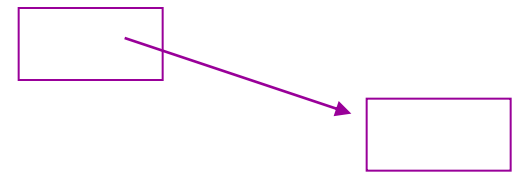
Opcionales

# VARIABLES ANÓNIMAS Y REFERENCIAS

- Algunos lenguajes permiten que el r-valor de una variable sea una referencia al l-valor de otra variable

Puntero a entero

*type pi* = <sup>instanciacion</sup> *integer*;  
*var pxi : pi* Aloca variable anónima setea el puntero  
*new (pxi)*



---

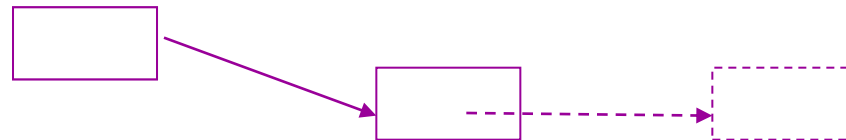
*type ppi* = <sup>instanciacion</sup> *pi*;

*var ppxi : ppi*;

....

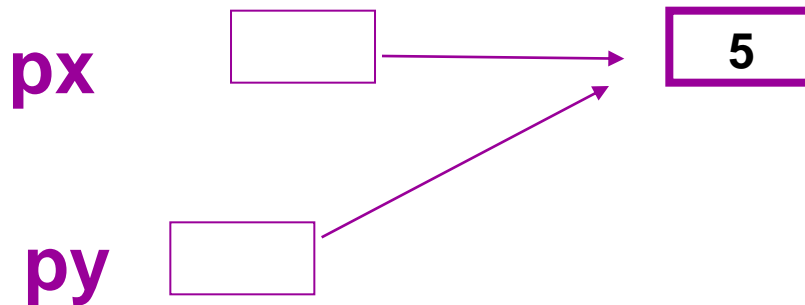
*new(ppxi);*

Puntero a un puntero



# ALIAS

- Dos variables comparten un objeto si sus caminos de acceso conducen al objeto. Un objeto compartido modificado vía un camino, se modifica para todos los caminos
- `int x = 5;`
- `int*px,`
- `px = &x ;`
- `py =px`



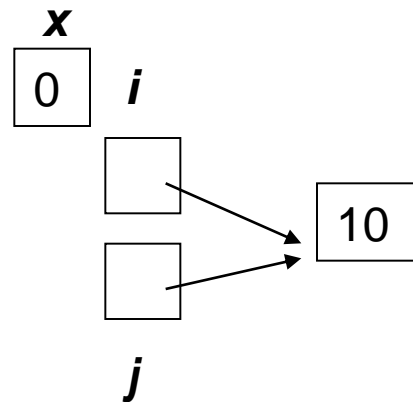
# ALIAS

**Alias:** Dos nombres que denotan la misma entidad en el mismo punto de un programa.

**distintos nombres  $\longrightarrow$  1 entidad**

- Dos variables son **alias** si **comparten el mismo objeto** de dato en el mismo **ambiente de referencia**. El uso de alias puede llevar a programas de **difícil lectura y a errores**.

```
int x = 0;  
int *i = &x;  
int *j = &x;
```



```
*i = 10;
```

**Efecto lateral:** modificación de una variable no local



# CONCEPTO DE SOBRECARGA

## Sobrecarga:

1 nombre → distintas entidades

*int i,j,k;*

*float a,b,c;*

.....

*i = j + k ;*

*a = b + c;*

¿Qué sucede con el operador + (nombre)?..

Los tipos permiten que se desambigüe en compilación.

**Sobrecarga:** un nombre esta **sobrecargado** si:

- En un momento, referencia **mas de una entidad** y
- **Hay suficiente información** para permitir establecer la ligadura unívocamente.

1 nombre —————→ 1 entidad  
No hay ambigüedad



# CONCEPTO DE SOBRECARGA Y ALIAS

- **Sobrecarga**

**1 nombre → distintas entidades**

- **Alias**

**distintos nombres → 1 entidad**

