



SEMANTICA OPERACIONAL

Repaso Clase Anterior

REPASO CLASE ANTERIOR

- Los lenguajes de programación trabajan con entidades
 - Variables
 - Unidades
 - Sentencias
- Las entidades tienen atributos
- Los atributos deben tener un valor antes de usar la entidad
- El momento de asociar un valor a un atributo se lo llama “binding o ligadura”
 - Ligadura estática
 - Ligadura dinámica
- Concepto de estabilidad



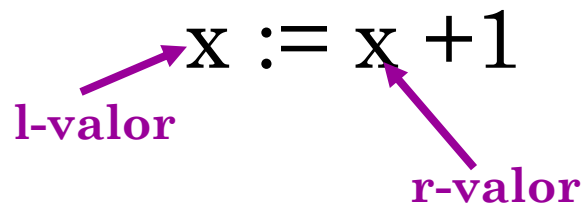
REPASO CLASE ANTERIOR

- Diferentes momentos de binding
 - Definición del lenguaje
 - Implementación
 - Compilación
 - Ejecución
- Entidad Variable
 - Atributos:
 - Nombre
 - Alcance
 - » Estático
 - » Dinámico
 - Tipo
 - L-valor
 - Tiempo de vida
 - R-valor



<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)



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`

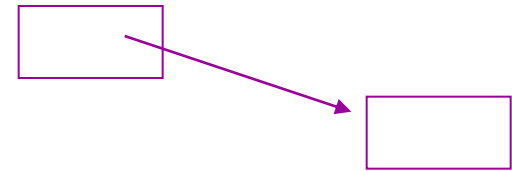
ADA `I,J INTEGER:=0`

Opcionales

VARIABLES ANÓNIMAS Y REFERENCIAS

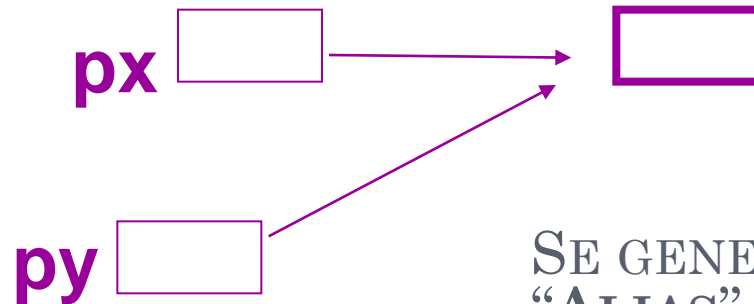
- Algunos lenguajes permiten que el r-valor de una variable sea una referencia al l-valor de otra variable

```
type pi = ^ integer;  
var pxi :pi  
new (pxi)
```



- Puede suceder que....

```
int x = 5;  
int*px,  
px = &x ;  
py =px
```



SE GENEREN
“ALIAS”



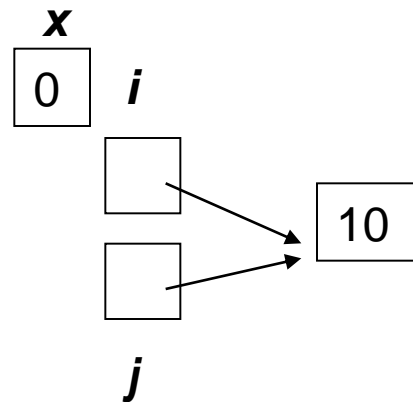
ALIAS

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



SEMANTICA OPERACIONAL

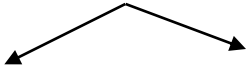

UNIDADES DE PROGRAMA

UNIDADES

- Los lenguajes de programación permiten que un programa este compuesto por **unidades**.

UNIDAD  **acción abstracta**

- En general se las llama **rutinas**


PROCEDIMIENTOS **FUNCIONES**  un valor

- Analizaremos las características sintácticas y semánticas de las rutinas y los mecanismos que controlan el flujo de ejecución entre rutinas con todas las ligaduras involucradas.

Hay lenguajes que SOLO tienen “funciones” y “simulan” los procedimientos con “funciones que devuelven void”. Ej.: C, C++, Python, etc



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

Nombre:

- String de caracteres que se usa para invocar a la rutina. (identificador)
- El nombre de la rutina se introduce en su declaración.
- El nombre de la rutina es lo que se usa para invocarlas.

The screenshot shows a C program in an IDE with two tabs: 'main.c' and 'input.txt'. The code is as follows:

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int uno()
5 {
6     int x=10;
7     printf("%d", x );
8 }
9
10 main()
11 {
12     float x=3.8;
13     uno();
14 }
15
```

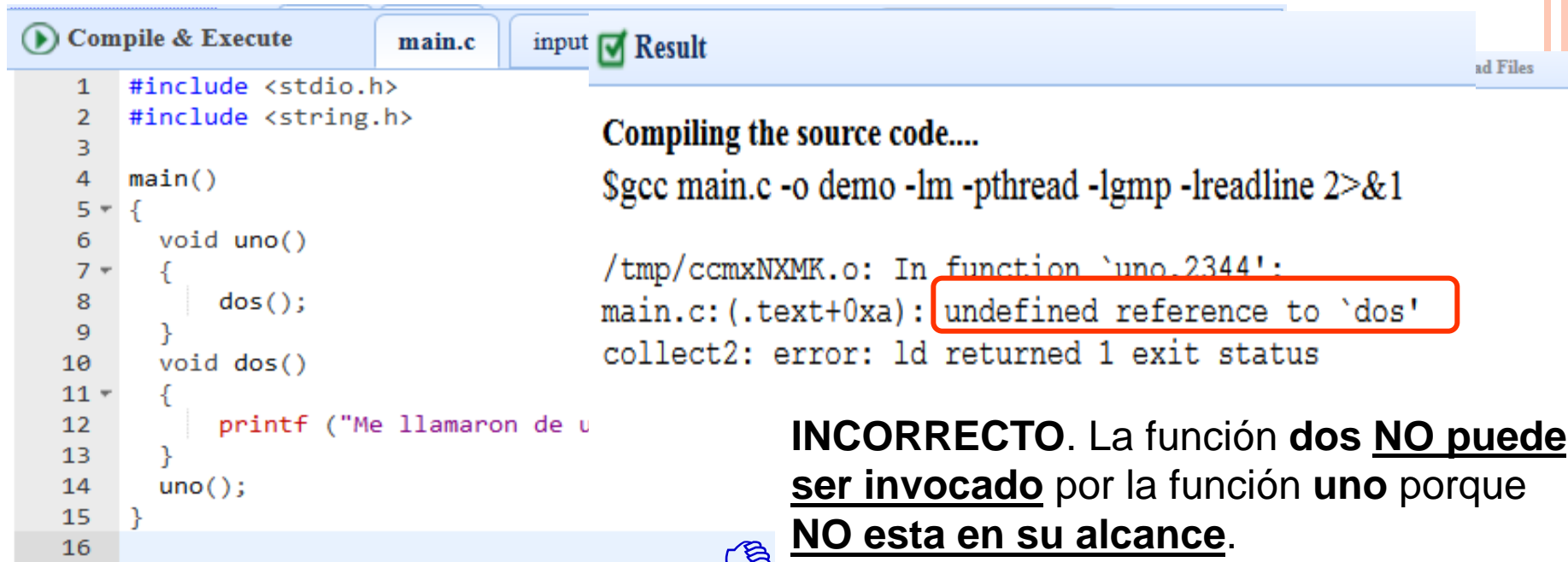
Annotations on the code:

- A purple box highlights the function signature `int uno()` on line 4, with an arrow pointing to a purple box labeled **declaración**.
- A purple box highlights the function call `uno();` on line 13, with an arrow pointing to a purple box labeled **invocación**.
- A black box on the right contains the text: "La llamada debe estar **dentro** del alcance del nombre de la rutina".

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

Alcance:

- ***Rango de instrucciones donde se conoce su nombre.***
 - El alcance se extiende desde el punto de su declaración hasta algún constructor de cierre.
 - Según el lenguaje puede ser estático o dinámico.
- **Activación:** la llamada puede estar solo dentro del alcance de la rutina



```
1 #include <stdio.h>
2 #include <string.h>
3
4 main()
5 {
6     void uno()
7     {
8         dos();
9     }
10    void dos()
11    {
12        printf ("Me llamaron de u
13    }
14    uno();
15 }
16
```

Compile & Execute main.c input ☒ Result ad Files

Compiling the source code....
\$gcc main.c -o demo -lm -pthread -lgmp -lreadline 2>&1

/tmp/ccmxNXMK.o: In function `uno.2344':
main.c:(.text+0xa): **undefined reference to `dos'**
collect2: error: ld returned 1 exit status

INCORRECTO. La función **dos** NO puede ser invocado por la función **uno** porque NO esta en su alcance.

DEFINICIÓN VS DECLARACIÓN

- Algunos lenguajes (C, C++, Ada, etc) hacen distinción entre Definición y Declaración de las rutinas

/ sum es una funcion que suma los n primeros naturales,
1 + 2 + ... + n; suponemos que el parametro n es positivo */*

int sum(int n)

Encabezado

Declaración

{

int i, s;

s = 0;

for (i = 1; i <= n; ++i)

s += i;

return s;

}

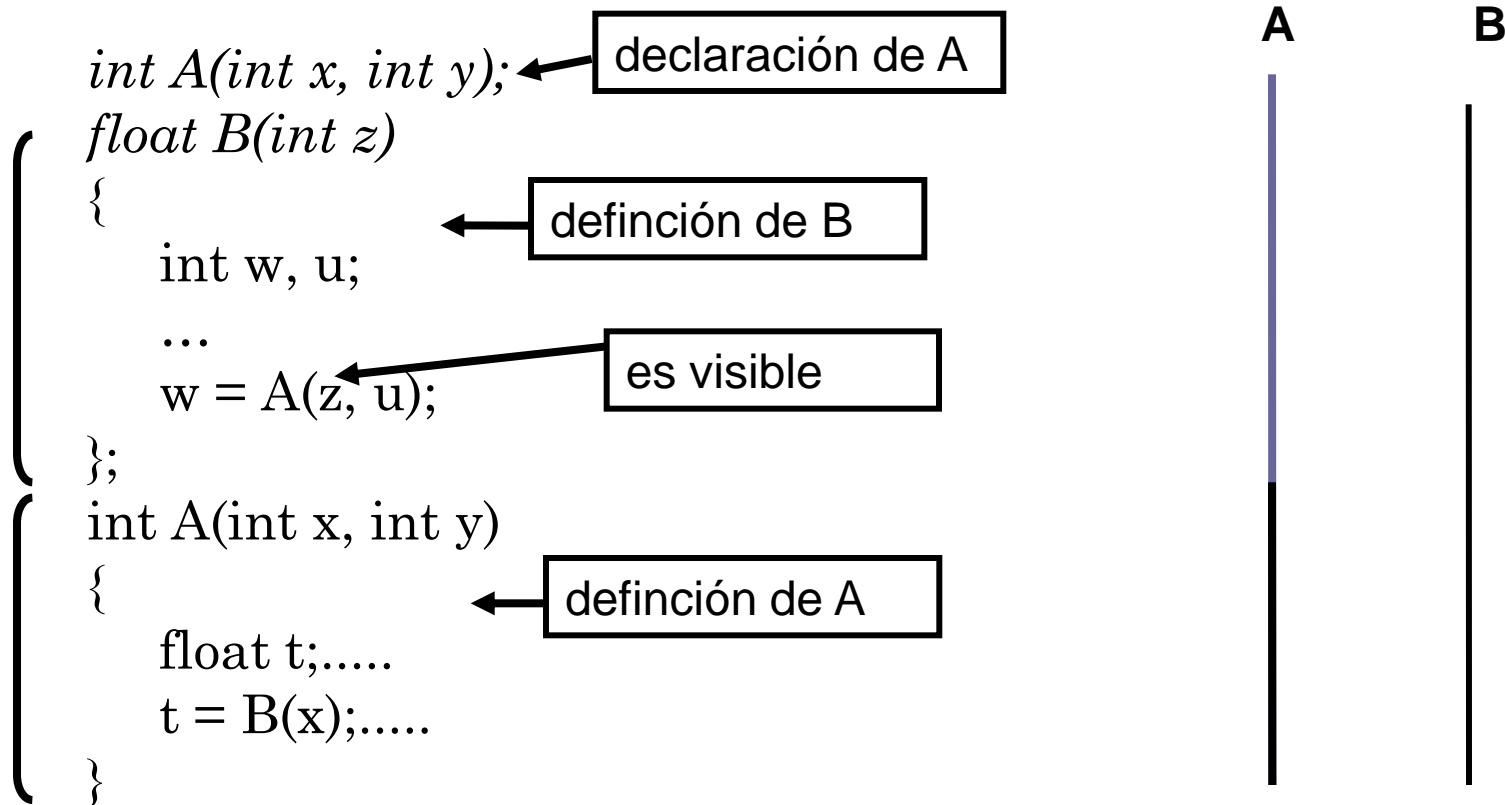
Cuerpo

Definición

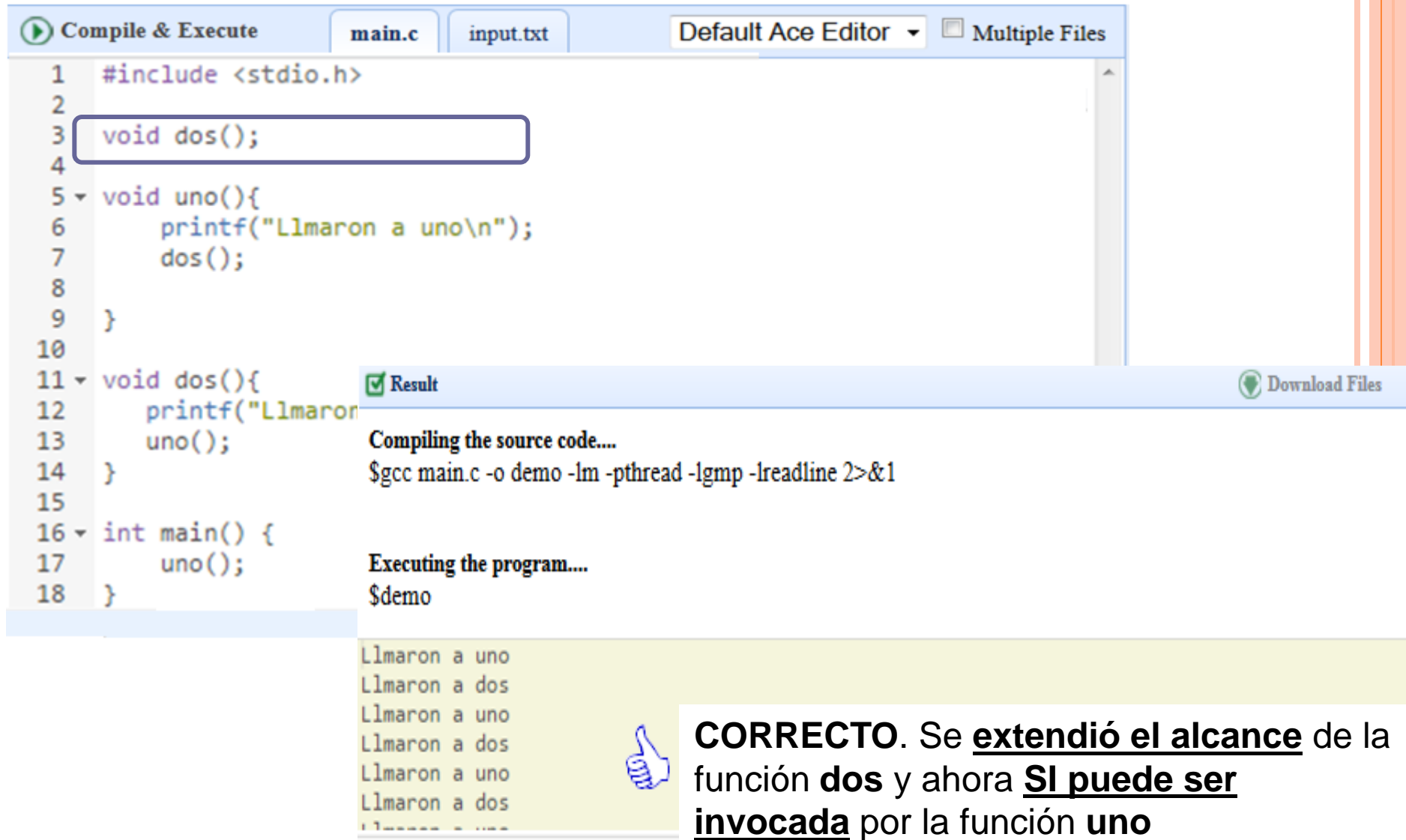


DEFINICIÓN VS DECLARACIÓN

- Si el lenguaje distingue entre la declaración y la definición de una rutina permite manejar esquemas de **rutinas mutuamente recursivas**.



- Extendiendo el alcance de las rutinas




```
1 #include <stdio.h>
2
3 void dos();
4
5 void uno(){
6     printf("Llmaron a uno\n");
7     dos();
8 }
9
10
11 void dos(){
12     printf("Llmaron a dos\n");
13     uno();
14 }
15
16 int main() {
17     uno();
18 }
```

Result

Compiling the source code....
\$gcc main.c -o demo -lm -pthread -lgmp -lreadline 2>&1

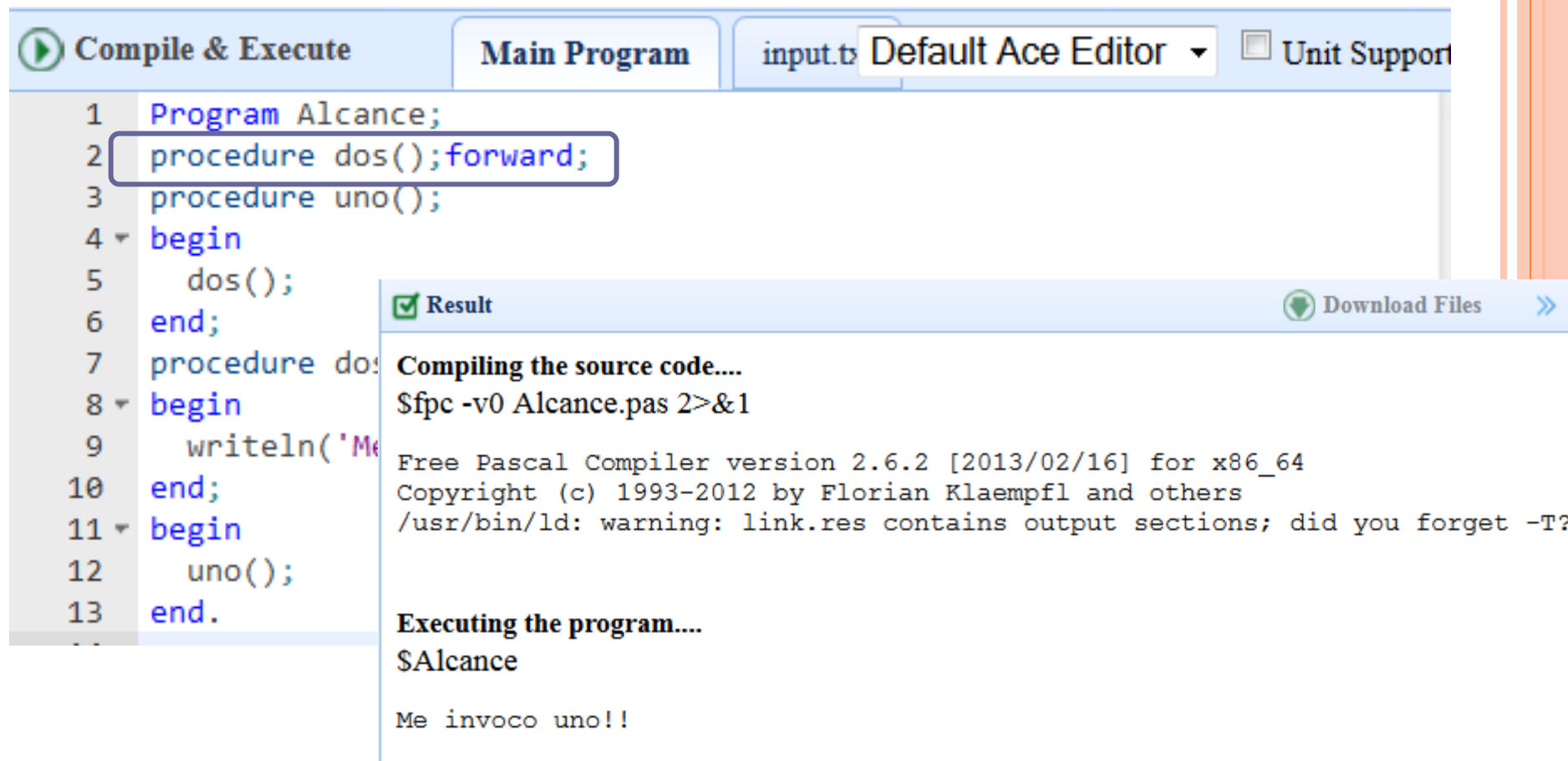
Executing the program....
\$demo

Llmaron a uno
Llmaron a dos
Llmaron a uno
Llmaron a dos
Llmaron a uno
Llmaron a dos

 **CORRECTO.** Se extendió el alcance de la función **dos** y ahora SI puede ser invocada por la función **uno**

DEFINICIÓN VS DECLARACIÓN

- Otro ejemplo: en Pascal uso de **forward**



The screenshot shows a Pascal IDE with a code editor and a console window. The code editor contains the following Pascal code:

```
1 Program Alcance;  
2 procedure dos();forward;  
3 procedure uno();  
4 begin  
5     dos();  
6 end;  
7 procedure dos;  
8 begin  
9     writeln('Me  
10 end;  
11 begin  
12     uno();  
13 end.
```

The line `procedure dos();forward;` is highlighted with a blue box. The console window shows the following output:

```
Result  
Compiling the source code....  
$fpc -v0 Alcance.pas 2>&1  
Free Pascal Compiler version 2.6.2 [2013/02/16] for x86_64  
Copyright (c) 1993-2012 by Florian Klaempfl and others  
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?  
  
Executing the program....  
$Alcance  
  
Me invoco uno!!
```


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

Tipo:

- El encabezado de la rutina define el tipo de los parámetros y el tipo del valor de retorno (si lo hay).
- **Signatura:** permite especificar el tipo de una rutina
Una rutina *fun* que tiene como entrada parámetros e tipo T1, T2, Tn y devuelve un valor de tipo R, puede especificarse con la siguiente signatura
$$fun: T1xT2x...Tn \rightarrow R$$
- Un llamado a una rutina es correcto si esta de acuerdo al tipo de la rutina.
- La conformidad requiere la correspondencia de tipos entre parámetros formales y reales.

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

Ejemplo:

/ sum es una funcion que suma los n primeros naturales,
1 + 2 + ... + n; suponemos que el parametro n es positivo */*

```
int sum(int n)  
{  
    int i, s;  
    s = 0;  
    for (i = 1; i <= n; ++i)  
        s += i;  
    return s;  
}
```

El tipo de la función sería:

sum: enteros \longrightarrow *enteros*

sum es una rutina con un parámetro entero que devuelve un entero

i = sum(10) **correcto!**

i = sum (5.3) **incorrecto!**



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

- ***l-value***: Es el lugar de memoria en el que se almacena el cuerpo de la rutina.
- ***r-value***: La llamada a la rutina causa la ejecución su código, eso constituye su r-valor.
 - **estático**: el caso mas usual.
 - **dinámica**: variables de tipo rutina.Se implementan a través de punteros a rutinas



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

r-value: Ejemplo de variables rutinas (binding dinámico)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void uno ()
5  {
6      printf("Me invocaron a través de un
7      puntero" );
8  }
9
10 main()
11 {
12     void (*punteroAFuncion());
13
14     /* Asigno dirección de la función uno a
15     punteroAFuncion*/
16     punteroAFuncion=&uno;
17
18     /* Invoco a función apuntada por puntero
19     */
20     punteroAFuncion();
21 }
22
```

Definición de variable puntero a función

Asignación de variable puntero a función

Invocación función

Result

Compiling the source code....

Sgcc main.c -o demo -lm -pthread -lgmp -lreadline 2>&1

Executing the program....

Sdemo

Me invocaron a través de un puntero

El uso de punteros a rutinas permite una política dinámica de invocación de rutinas

REPRESENTACION EN EJECUCION

- La definición de la rutina especifica un proceso de computo.
- Cuando se invoca una rutina se ejecuta una instancia del proceso con los particulares valores de los parámetros.
- **instancia de la unidad:** es la representación de la rutina en ejecución.




Segmento de código

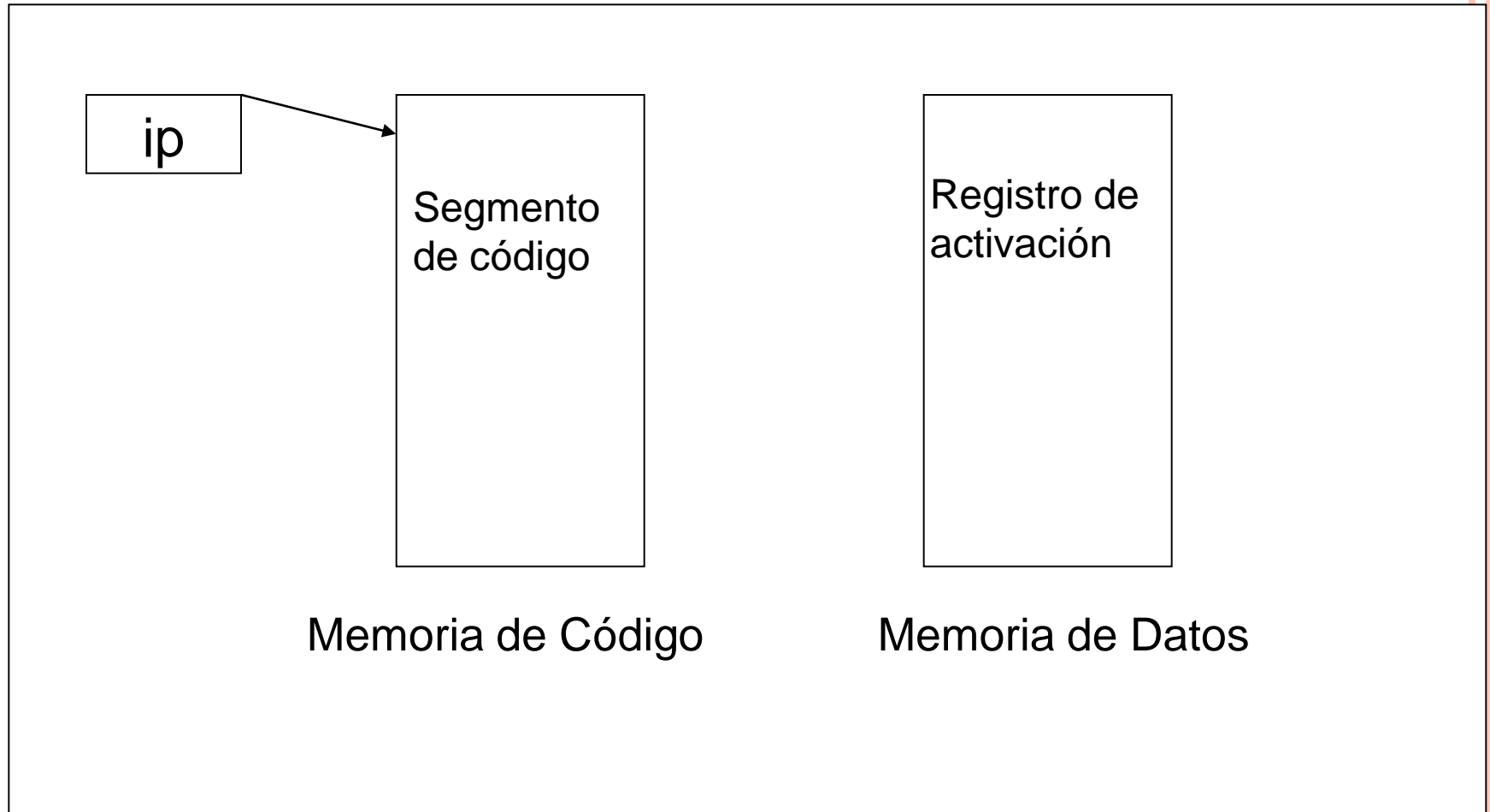
Instrucciones de la unidad
se almacena en la memoria
de instrucción **C**
Contenido fijo

Registro de activación

Datos locales de la unidad
se almacena en la memoria
de datos **D**
Contenido cambiante



PROCESADOR ABSTRACTO - SIMPLESEM



PROCESADOR ABSTRACTO - UTILIDAD

- El procesador nos servirá para comprender que efecto causan las instrucciones del lenguaje al ser ejecutadas.
- Semántica intuitiva.
- Se describe la semántica del lenguaje de programación través de reglas de cada constructor del lenguaje traduciéndolo en una secuencia de instrucciones equivalentes del procesador abstracto



PROCESADOR ABSTRACTO - SIMPLESEM

Memoria de Código: $C(y)$ valor almacenado en la y ésima celda de la memoria de código. Comienza en cero.

Memoria de Datos: $D(y)$ valor almacenado en la y ésima celda de la memoria de datos. Comienza en cero.

“ y ” representa el l-valor, $D(y)$ o $C(y)$ su r-valor

Ip: puntero a la instrucción que se esta ejecutando.

- Se inicializa en cero en cada ejecución se actualiza cuando se ejecuta cada instrucción.
- Direcciones de C

Ejecución:

- obtener la instrucción actual para ser ejecutada ($C[ip]$)
- incrementar ip
- ejecutar la instrucción actual



PROCESADOR ABSTRACTO - INSTRUCCIONES

SET: setea valores en la memoria de datos

set target,source

Copia el valor representado por **source** en la dirección representada por **target**

set 10,D[20]



copia el valor almacenado en la posición 20 en la posición 10.

E/S: read y write permiten la comunicación con el exterior.

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



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



combinación de expresiones

*set 99, D[15]+D[33]*D[4]* expresión para modificar el valor



PROCESADOR ABSTRACTO – INSTRUCCIONES (CONT.)

JUMP: bifurcación incondicional
jump 47

la próxima instrucción a ejecutarse será la que este almacenada en la dirección 47 de C

JUMPT: bifurcación condicional, bifurca si la expresión se evalúa como verdadera
jumpt 47,D[13]>D[8]

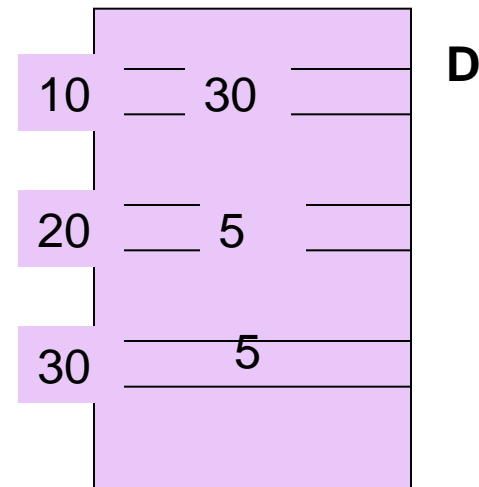
bifurca si el valor almacenado en la celda 13 es mayor que el almacenado en la celda 8

direccionamiento indirecto:

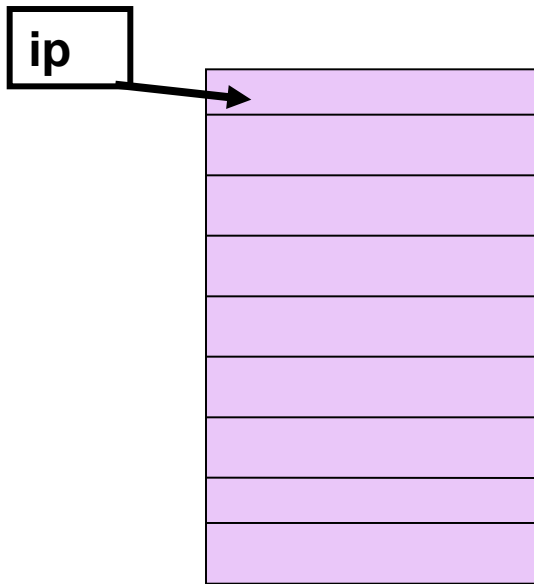
set D[10],D[20]

jump D[30]

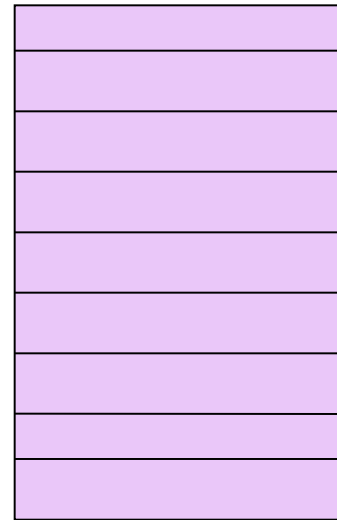
Ip= 5 posición 5 en C



PROCESADOR ABSTRACTO - MEMORIA



C



D



ELEMENTOS EN EJECUCIÓN

- **Punto de retorno**

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

- **Ambiente de referencia**

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



ESTRUCTURA DE EJECUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

- **Estático**
- **Basado en pila**
- **Dinámico**



ESTATICO: ESPACIO FIJO

- El espacio necesario para la ejecución se deduce del código
- Todo los requerimientos de memoria necesarios se conocen antes de la ejecución
- La alocaación puede hacerse estáticamente
- No puede haber recursión



BASADO EN PILA:

ESPACIO PREDECIBLE

- El espacio se deduce del código. Algol-60
- Programas más potentes cuyos requerimientos de memoria no puede calcularse en traducción.
- La memoria a utilizarse es **predecible** y sigue una disciplina last-in-first-out.
- Las variables se alocan automáticamente y se desalocan cuando el alcance se termina
- Se utiliza una estructura de pila para modelizarlo.
- Una pila no es parte de la semántica del lenguaje, es parte de nuestro modelo semántico.



DINAMICO: ESPACIO IMPREDECIBLE

- Lenguajes con impredecible uso de memoria.
- Los datos son alocados dinámicamente solo cuando se los necesita durante la ejecución.
- No pueden modelizarse con una pila, el programador puede crear objetos de dato en cualquier punto arbitrario durante la ejecución del programa.
- Los datos se alocan en la zona de memoria heap



C1: LENGUAJE SIMPLE

- Sentencias simples
- Tipos simples
- Sin funciones
- Datos estáticos de tamaño fijo
- un programa = una rutina main()
 - Declaraciones
 - Sentencias
- E/S: get/print

enteros
reales
arreglos
estructuras



C1

Zona DATOS

```
main()
```

```
{
```

```
    int i, j;
```

```
    get(i, j);
```

```
    while (i != j)
```

```
        if (i > j)
```

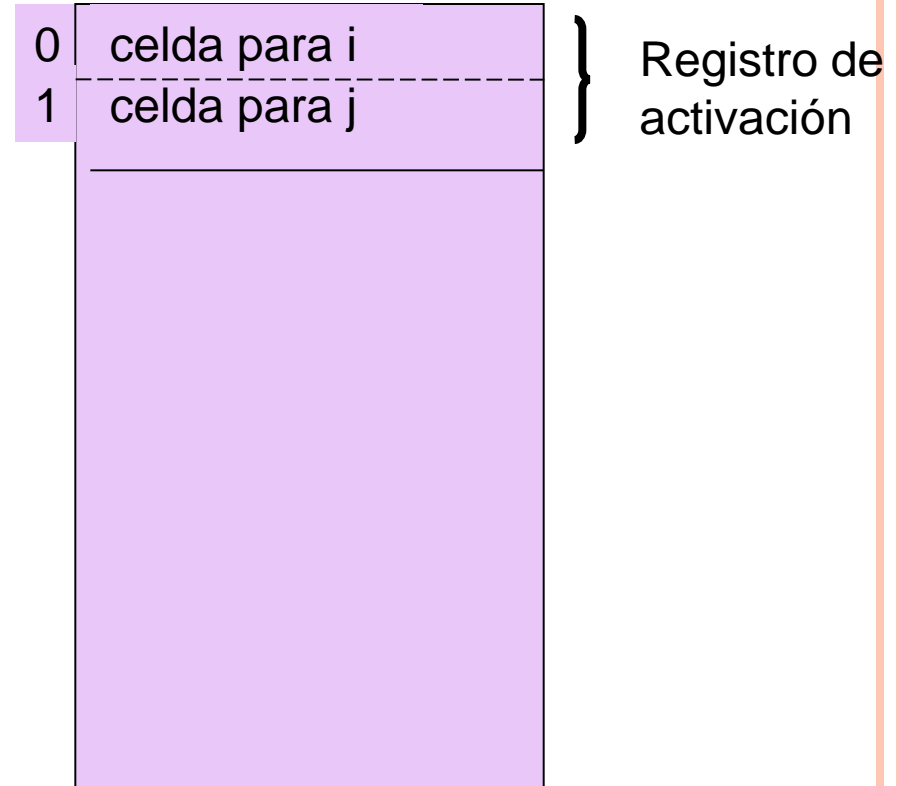
```
            i -= j;
```

```
        else
```

```
            j -= i;
```

```
    print(i);
```

```
}
```

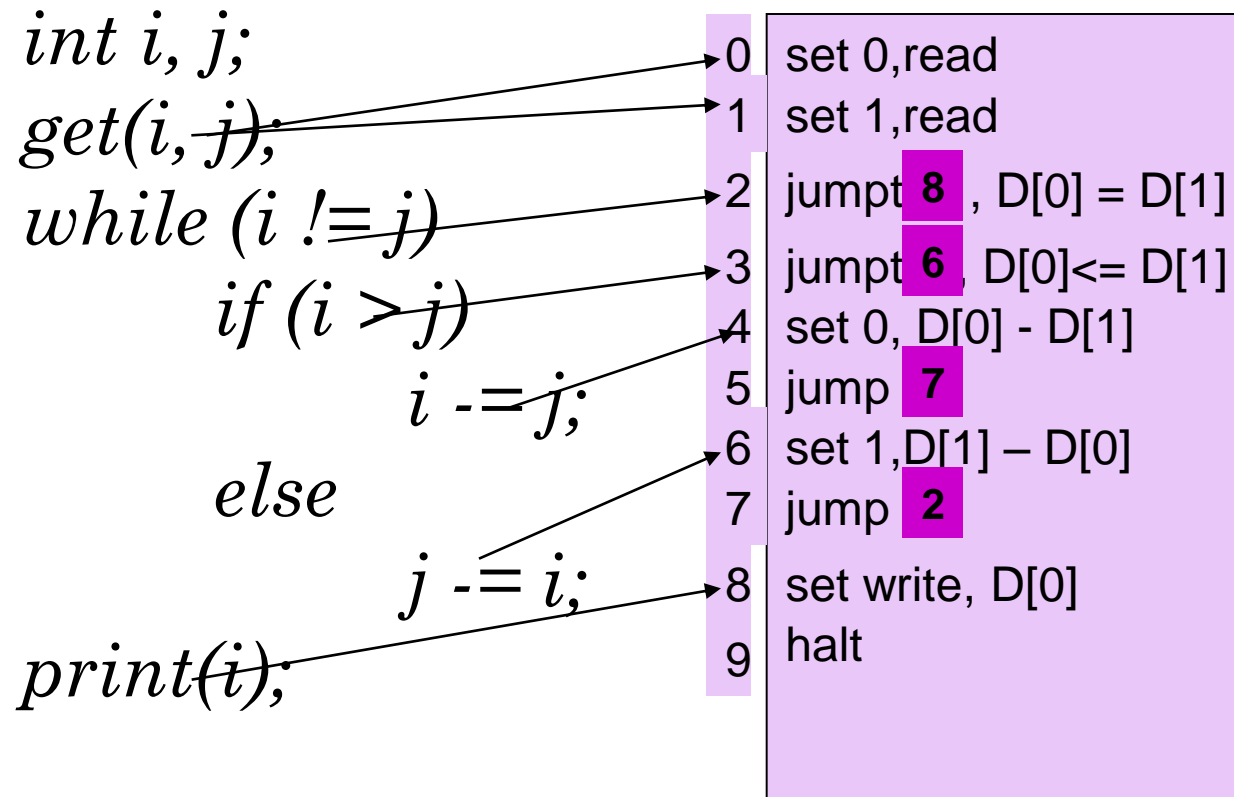


C1

¿Cómo sería el **CÓDIGO**?

```
main()  
{
```

Zona **CÓDIGO**



C1

C

D

0

0	set 0,read
1	net 1,read
2	jumpt 8, $D[0] = D[1]$
3	jumpt 6, $D[0] \leq D[1]$
4	set 0, $D[0] - D[1]$
5	jump 7
6	set 1, $D[1] - D[0]$
7	jump 2
8	set write, $D[0]$
9	halt

0	celda para i
1	celda para j

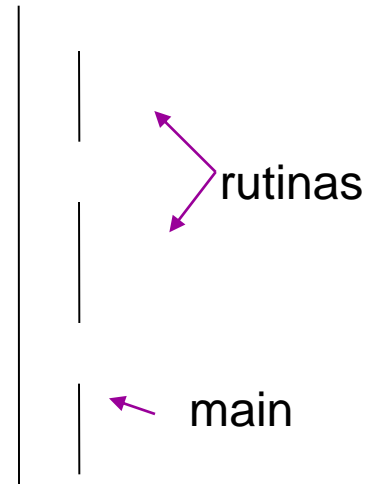
**Registro de
activación**



C2 : C1 + RUTINAS INTERNAS

Definición de rutinas internas al main

- Programa =
 - Datos globales
 - Declaraciones de rutinas
 - Rutina principal
- Rutinas internas
 - Disjuntas: no pueden estar anidadas
 - No son recursivas
- Ambiente de las rutinas internas
 - Datos locales
 - Datos globales



C2

```
int i = 1, j = 2, k = 3;
alpha()
{
```

```
    int i = 4, j = 5;
    ...
    i += k + j;
```

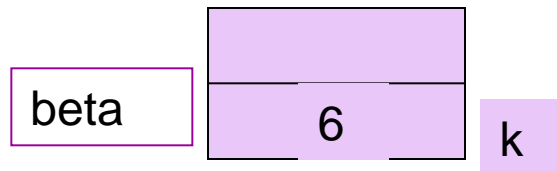
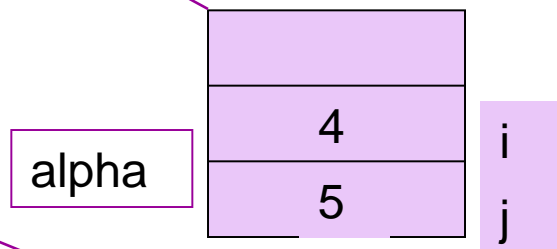
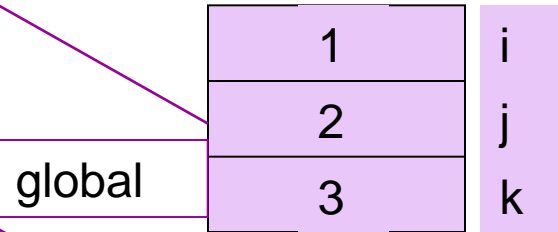
```
};
beta()
{
```

```
    int k = 6;
    ...
    i = j + k;
    alpha();
```

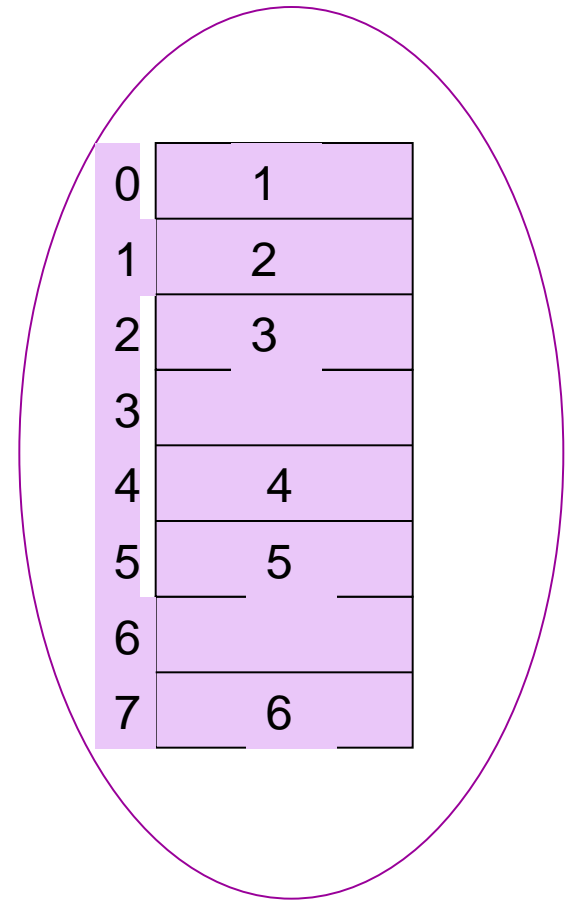
```
};
main()
{
```

```
    ...
    beta();
```

```
}
```



Todo estático

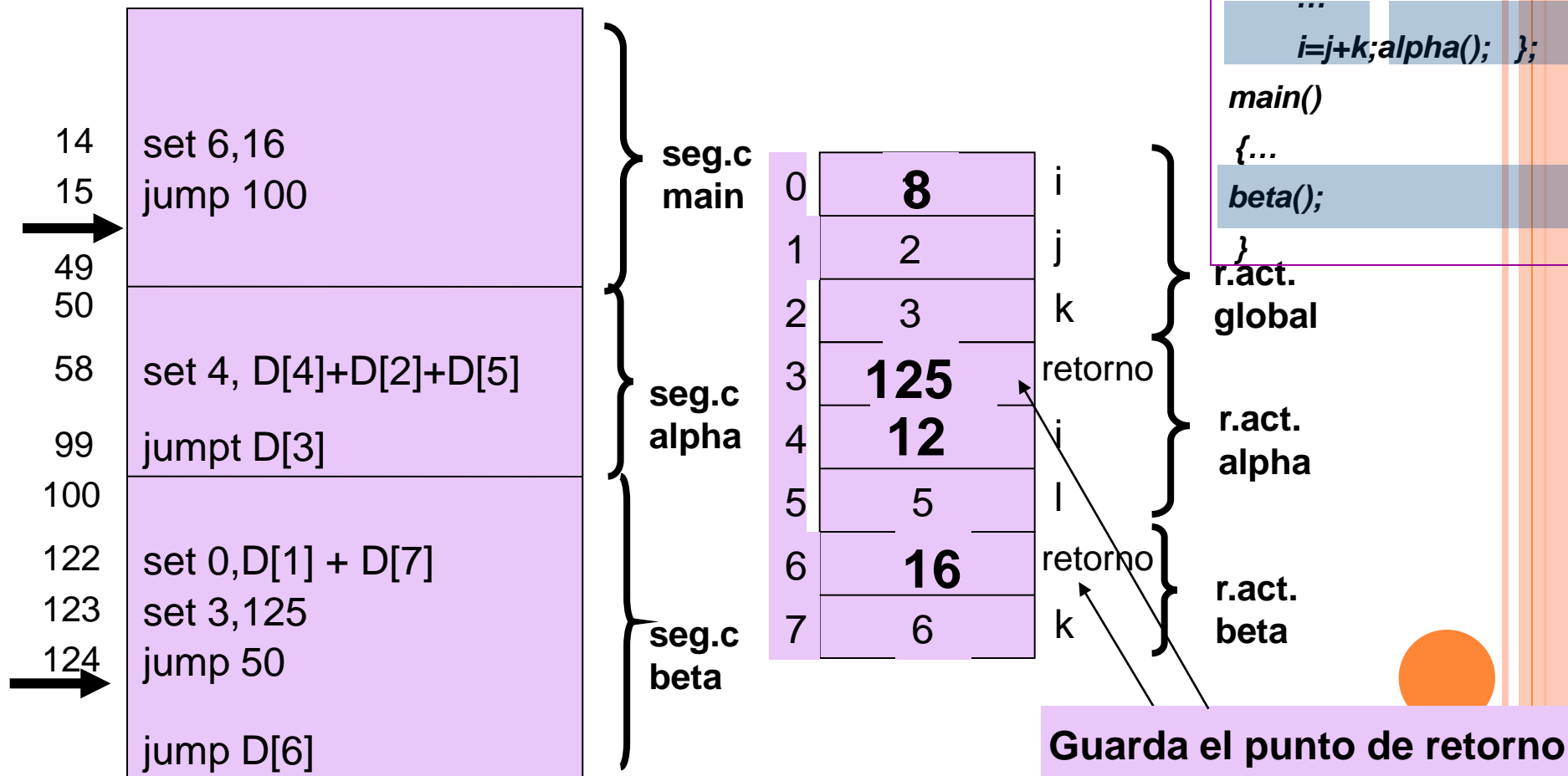


¿Qué RA se alocan en la zona de Datos?



C2: CALL-RETURN

¿Cómo cambia la información en la zona de **Datos**?



C2': RUTINAS COMPILADAS SEPARADAS

file 1

```
int i = 1, j = 2, k = 3;  
extern beta();  
main()
```

```
{...  
beta();  
...} ...
```

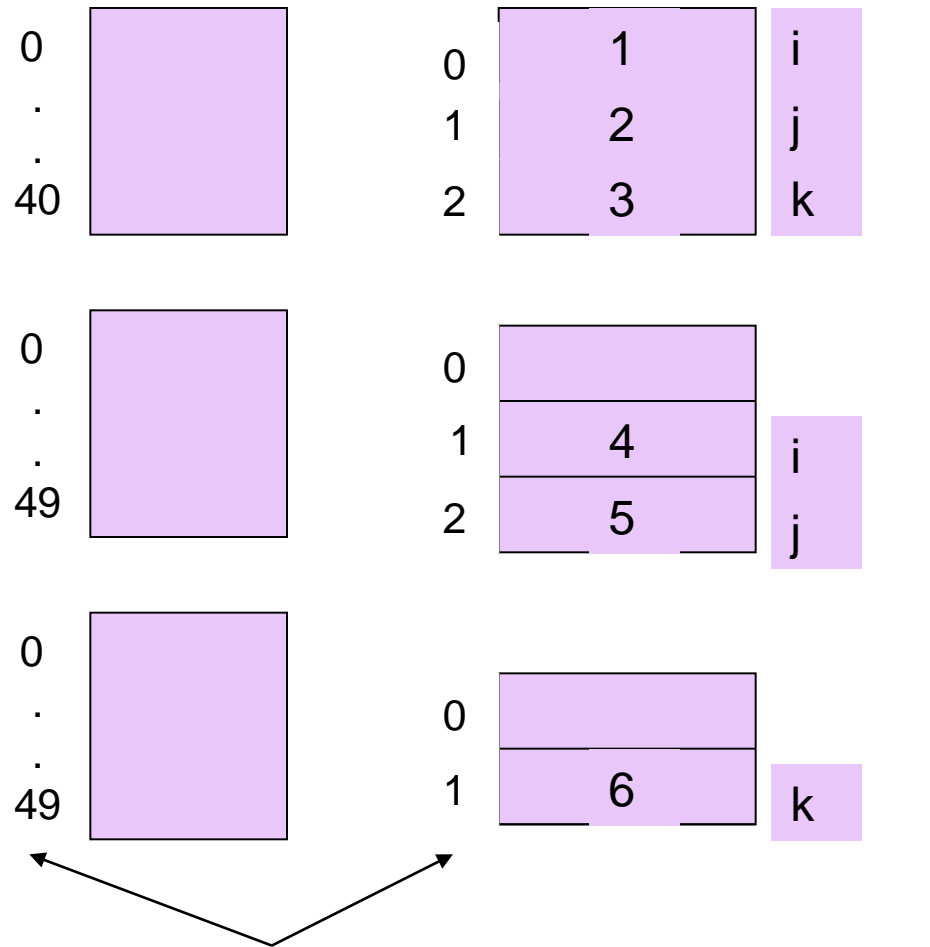
file 2

```
extern int k;  
alpha() {...}
```

file 3

```
extern int i, j;  
extern alpha();  
beta() { }
```

```
...  
alpha();...
```



C2'

- El compilador no puede ligar variables locales a direcciones absolutas
- Tampoco variables globales
- Linkeditor:
 - encargado de combinar los módulos
 - ligar la información faltante
- C2 y C2' no difieren semánticamente

