



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





SEMANTICA OPERACIONAL

UNIDADES DE PROGRAMA

UNIDADES

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

UNIDAD  **acción abstracta**

- En general se las llama **rutinas**

PROCEDIMIENTOS  **FUNCIONES**  **retornan 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.

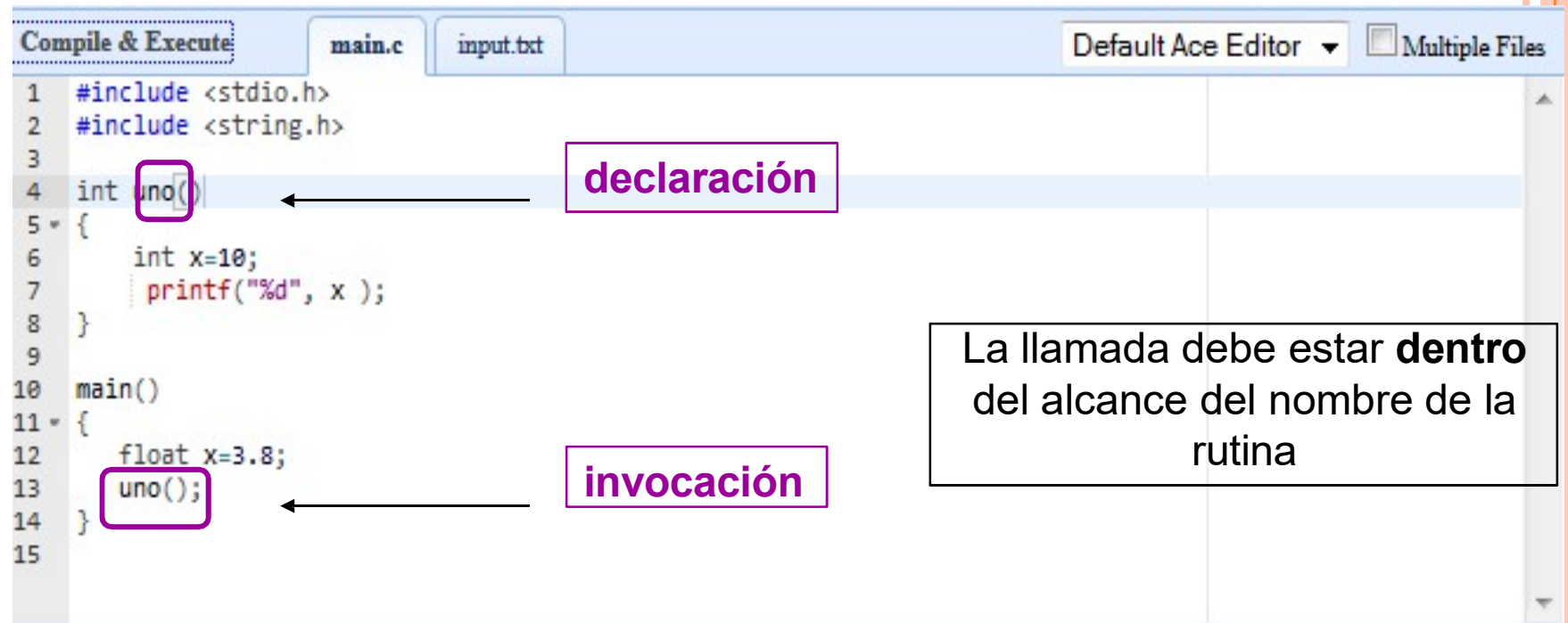
Hay lenguajes que SOLO tienen “funciones” y “simulan” los procedimientos con funciones que devuelven "void" o “none”.
Ej.: C, C++, Python, etc



<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 code editor window with two tabs: 'main.c' and 'input.txt'. The 'main.c' tab is active, displaying the following C code:

```
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".

Compile and Execute Pascal Online (Free Pascal v3.0.2)

```
Execute | Share main.pas STDIN Result
```

```
1 Program Alcance(output) ;
2
3 FUNCTION suma (a:integer; b:integer): integer;
4 begin
5     suma:= a + b;
6 end;
7 begin
8     writeln('La suma es:', suma( 7, 3));
9 end.
```

```
$fpc -vw main.pas
Free Pascal Compiler version 3.0.2
Copyright (c) 1993-2017
Target OS: Linux for x86_64
Compiling main.pas
Linking main
9 lines compiled, 0.2 seconds
/usr/bin/ld: warning: libc.so.6: no version information available (required by /usr/bin/ld)
$main
La suma es:10
```

C++ shell

```
1 // Alcance
2
3 #include <stdio.h>
4
5 int suma(int a, int b)
6 {
7     return a + b;
8 }
9
10
11
12 int main(void)
13 {
14     /* Llamando a una función */
15     printf("El resultado de la suma es: %i ", suma( 7, 3));
16
17     return 0;
18 }
19
20
21
22
23
```

Compile and Execute C Online (GNU GCC v7.1.1)

```
Execute | Share main.c STDIN Result
```

```
1 #include <stdio.h>
2
3 int suma (int a, int b)
4 {
5
6     return a + b;
7 }
8
9 int main(void)
10 {
11     /* Se llama a la función */
12     printf("%i ", suma( 7, 3));
13
14     return 0;
15 }
16
```

```
$gcc -o main *.c
$main
10
```

Get URL

options compilation execution

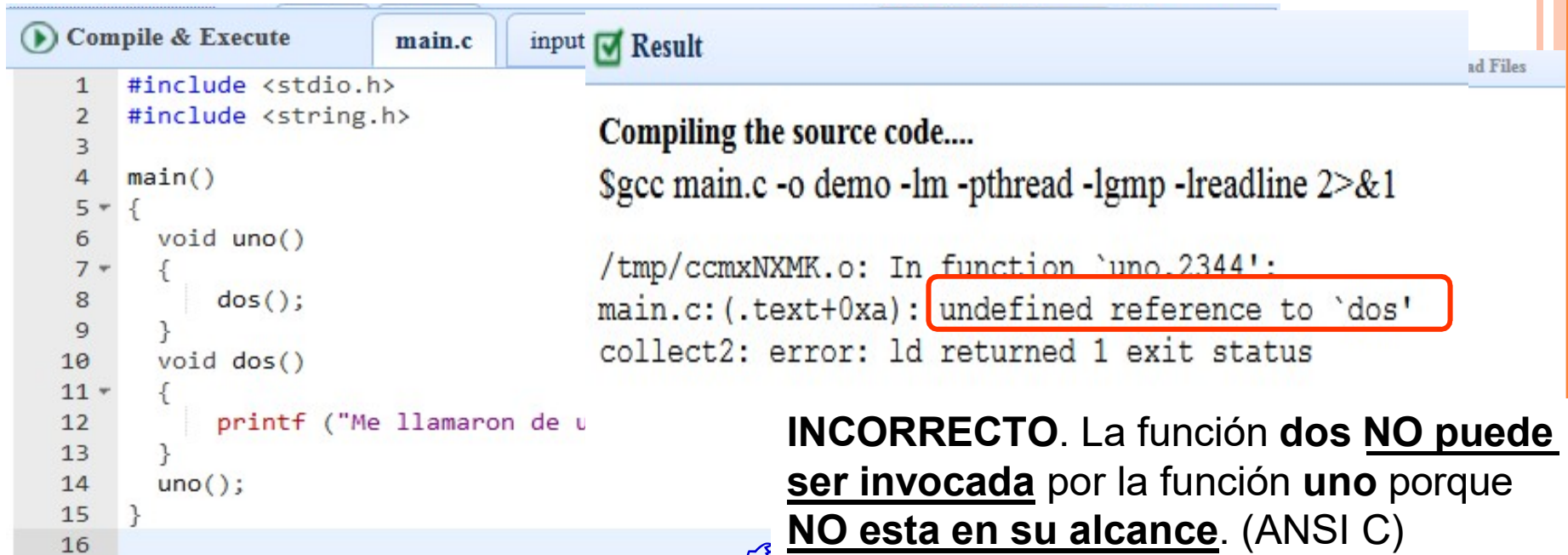
El resultado de la suma es: 10

Las funciones se han ejecutado. ¿Por qué?
¿Qué pasaría si en C o C++ coloco la definición
de la función DEBAJO del main? ¿O en Pascal,
la función suma llama a otra que la vuelve a
llamar a ella?

<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 o invocación 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

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

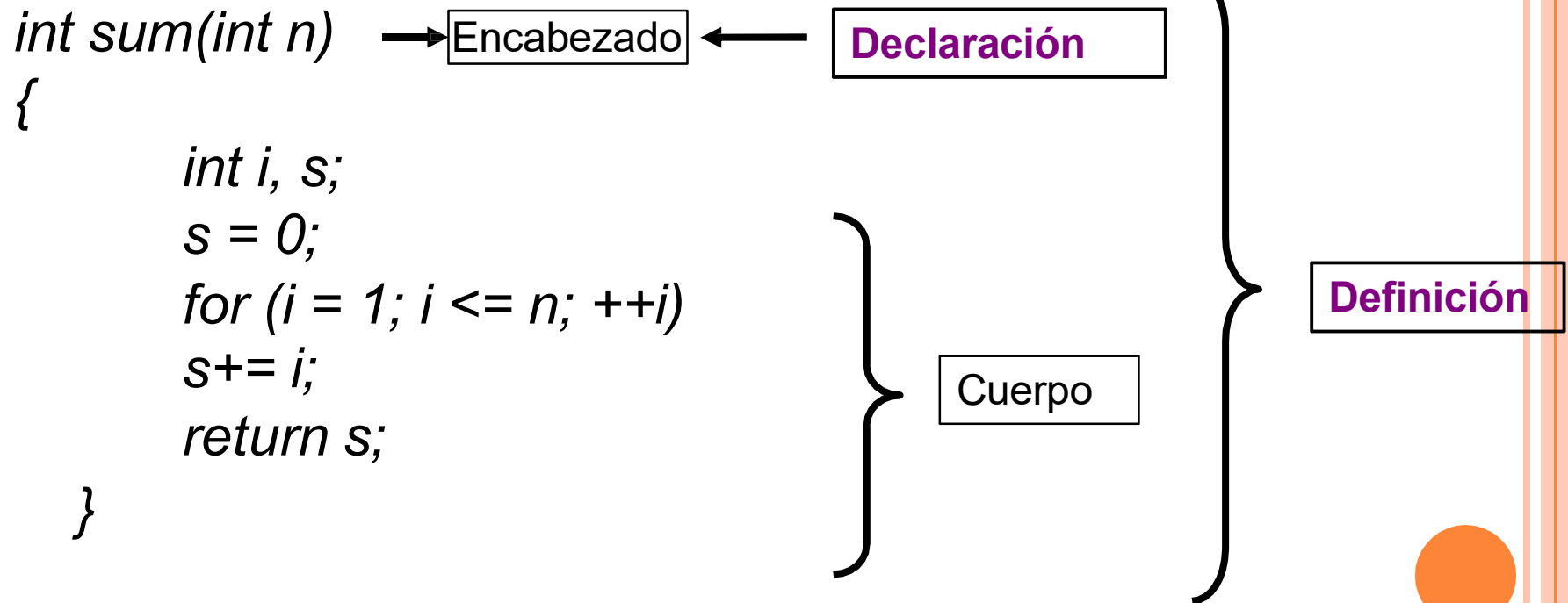
/tmp/ccmxNXMK.o: In function `uno.2344':
main.c:(.text+0xa): **undefined reference to `dos'**
collect2: error: ld returned 1 exit status

INCORRECTO. La función **dos** NO puede ser invocada por la función **uno** porque NO esta en su alcance. (ANSI C)

DEFINICIÓN VS DECLARACIÓN

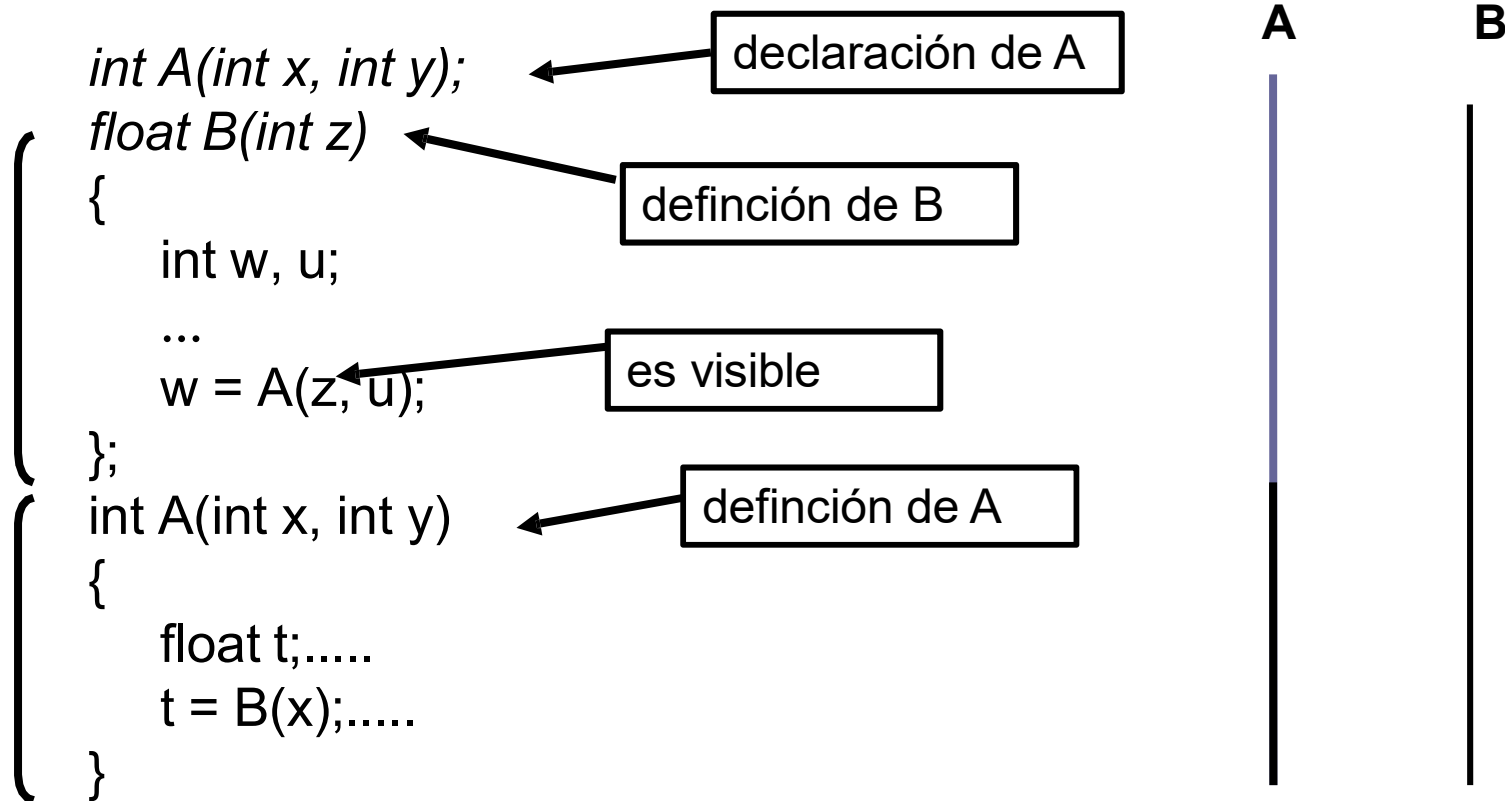
- Algunos lenguajes (C, C++, Ada, etc) hacen distinción entre Definición y Declaración de las rutinas

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



DEFINICIÓN VS DECLARACIÓN

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



PROTOTIPO

Execute | Share | main.c | STDIN | Result

```
1 #include <stdio.h>
2
3 // Alcance
4
5 int main(void)
6 {
7     /* Se llama a la función */
8     printf("%f ", suma( 7, 3));
9
10    return 0;
11 }
12
13 float suma(int a, int b)
14 {
15     float z;
16     z= a+ b;
17     return (z);
18 }
19
20
```

```
$gcc -o main *.c
main.c: In function 'main':
main.c:8:16: warning: implicit declaration of function 'suma' [-Wimplicit-function-declaration]
printf("%f ", suma( 7, 3));
               ^~~~~
main.c: At top level:
main.c:14:7: error: conflicting types for 'suma'
float suma(int a, int b)
      ^~~~~
main.c:8:16: note: previous implicit declaration of 'suma' was here
printf("%f ", suma( 7, 3));
               ^~~~~
```

Da ERROR! Cuando no coincide el tipo de retorno con lo que “supuso” por defecto el compilador...

Execute | Share | main.c | STDIN | Result

```
1 #include <stdio.h>
2
3 // Alcance
4
5 int main(void)
6 {
7     /* Se llama a la función */
8     printf("%i ", suma( 7, 3));
9
10    return 0;
11 }
12
13 int suma(int a, int b)
14 {
15     return (a+ b);
16 }
```

```
$gcc -o main *.c
main.c: In function 'main':
main.c:8:16: warning: implicit declaration of function 'suma' [-Wimplicit-function-declaration]
printf("%i ", suma( 7, 3));
               ^~~~~

$main
10
```

NO da ERROR! Cuando coincide el tipo de retorno con lo que “supuso” por defecto el compilador: enteros

En algunas implementaciones de C, si no se encuentra la declaración de la función o prototipo, **se asume un prototipo** que devuelve enteros

C++ shell

```
1 #include <stdio.h>
2
3 // Alcance
4
5 int main(void)
6 {
7     /* Se llama a la función */
8     printf("%i ", suma( 7, 3));
9
10    return 0;
11 }
12
13
14 int suma(int a, int b)
15 {
16     return (a+ b);
17 }
18
19
```

Get URL

options

compilation

execution

In function 'int main()':
8:26: error: 'suma' was not declared in this scope

[C++ Shell, 2014-2015](#)

C++ shell

```
1 #include <stdio.h>
2
3 // Alcance
4 int suma(int, int);
5
6 int main(void)
7 {
8     /* Se llama a la función */
9     printf("%i ", suma( 7, 3));
10
11    return 0;
12 }
13
14
15 int suma(int a, int b)
16 {
17     return (a+ b);
18 }
19
20
```

Get URL

Run

options

compilation

execution

10

Exit code: 0 (normal program termination)

[C++ Shell, 2014-2015](#)

En C++ no asume prototipo pero permite adelantar la declaración

DEFINICIÓN VS DECLARACIÓN

compile pascal online

SPONSOR Hotjar See how your visitors are really using your website.

Language: Pascal Editor: CodeMirror Layout: Vertical

```
1 //fpc 3.0.0
2
3 program Alcance;
4 FUNCTION suma(a: integer; b:integer):integer;
5 begin
6     if (b > 0) then resta (a, b)
7     else suma:= a + b;
8 end;
9
10 FUNCTION resta (x: integer; z:integer):integer;
11 begin
12     z:= z-1;
13     resta:= suma(x,z);
14 end;
15 begin
16     writeln('Resultado de invocar a suma con 7 y 2', suma (7,2));
17 end.
```

Run it (F8) Save it ☒ Show compiler warnings [+] Show input

Compilation time: 0.12 sec, absolute service time: 0,21 sec

Error(s):

source.pas(6,25) Error: Identifier not found "resta"
source.pas(18) Fatal: There were 1 errors compiling module, stoppi
Error: /usr/bin/ppcx64 returned an error exitcode

En Pascal uso de
forward

compile pascal online

SPONSOR Hotjar See how your visitors are really using your website.

Language: Pascal Editor: CodeMirror Layout: Vertical

```
1 //fpc 3.0.0
2
3 program Alcance;
4 FUNCTION resta (x: integer; z:integer):integer; forward;
5 FUNCTION sumaRara(a: integer; b:integer):integer;
6 begin
7     if (b > 0) then resta (a, b);
8     sumaRara:= a + b;
9 end;
10
11 FUNCTION resta (x: integer; z:integer):integer;
12 begin
13     z:= z-1;
14     resta:= sumaRara(x,z);
15 end;
16 begin
17     writeln('Resultado de invocar a suma con 7 y 2 es: ', sumaRara (7,2));
18 end.
```

Run it (F8) Save it ☒ Show compiler warnings [+] Show input

Compilation time: 0.12 sec, absolute running time: 0.07 sec, cpu time: 0.01 sec, memory peak: 3 Mb, absolute service time: 0.01 sec

Resultado de invocar a suma con 7 y 2 es: 9

<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 de tipo T1, T2, Tn y devuelve un valor de tipo R, puede especificarse con la siguiente signatura

$$fun: T1 \times T2 \times \dots \times Tn \rightarrow R$$

- Un llamado a una rutina es correcto si está de acuerdo al tipo de la rutina.
- La **conformidad** requiere la correspondencia de tipos entre parámetros formales y reales.



<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



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

L-VALUE y 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 más 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)

```
main.c
1  /*****
2      Online C Compiler.
3      Code, Compile, Run and Debug C program online.
4      Write your code in this editor and press "Run" button to compile and execute it.
5      *****/
6
7  #include <stdio.h>
8  void uno( int valor)
9  { if (valor == 0) printf ("Me invocaron con el identificador uno\n");
10    else printf ("Me invocaron a través de un puntero a función\n");
11  }
12
13  int main()
14  {
15      int y;
16      void (*punteroAFuncion)();
17
18      printf("Probando R-VALUE funciones\n");
19
20      /* Pureba de Llamada a función R-VALUE estático*/
21      y= 0;
22      uno(y);
23
24      /* Pureba de Llamada a función R-VALUE dinámico*/
25      y= 1;
26      punteroAFuncion = &uno;
27      punteroAFuncion(y);
28
29      return 0;
30  }
31
```

Definición de variable puntero a función

Asignación de variable puntero a función

Invocación función

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

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

R-VALUE: Ejemplo de variables rutinas (binding dinámico)

```
main.c
1  /*****
2      Online C Compiler.
3      Code, Compile, Run and Debug C program online.
4      Write your code in this editor and press "Run" button to compile and execute it.
5      *****/
6
7  #include <stdio.h>
8  void uno( int valor)
9  { if (valor == 0) printf ("Me invocaron con el identificador uno\n");
10     else printf ("Me invocaron a través de un puntero a función\n");
11 }
12
13 int main()
14 {
15     int y;
16     void (*punteroAFuncion)();
17
18     printf("Probando R-VALUE funciones\n");
19
20     /* Pureba de Llamada a función R-VALUE estático*/
21     y= 0;
22     uno(y);
23
24     /* Pureba de Llamada a función R-VALUE dinámico*/
25     y= 1;
26     punteroAFuncion = &uno;
27     punteroAFuncion(y);
28
29     return 0;
30 }
31
```

Definición de variable puntero a función

Asignación

Invocación función

input

```
Probando R-VALUE funciones
Me invocaron con el identificador uno
Me invocaron a través de un puntero a función

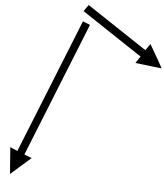
...Program finished with exit code 0
Press ENTER to exit console.
```

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

COMUNICACIÓN ENTRE RUTINAS

- **Ambiente no local**

- **Parámetros**



Diferentes datos en cada llamado

Mayor legibilidad y modificabilidad.

- **Parámetros formales:** los que aparecen en la definición de la rutina
- **Parámetro reales:** los que aparecen en la invocación de la rutina. (dato o rutina)



LIGADURA ENTRE PARÁMETROS FORMALES Y REALES

- **Método posicional:** se ligan uno a uno
routine S (F1,F2,.....,Fn) **Definición**
call S (R1, R2,..... Rn) **Llamado**

Los parámetros reales *Ri* se ligan a los parámetros formales *Fi* por posición para i de 1 a n. Deben conocerse las posiciones.

Variante: combinación con valores **por defecto**

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

distancia() \longrightarrow *distancia (0,0)*

distancia(10) \longrightarrow *distancia (10,0)*



LIGADURA ENTRE PARÁMETROS FORMALES Y REALES

- **Método por nombre:** se ligan por el nombre
deben conocerse los nombres de los formales
Ada: permite valor por defecto, métodos por posición y por
nombre. Ej *procedure Ejem (A:T1; B: T2:= W; C:T3);*

↓
Valor por defecto

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

Ejem (X,Y,Z) → asociación posicional

Ejem (X,C => Z) → X se liga a A por posición,
B toma el valor por defecto W
C se liga a Z por nombre

Ejem (C =>Z, A=>X, B=>Y) →
se ligan todos por nombre



REPRESENTACION EN EJECUCION

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



Segmento de código

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

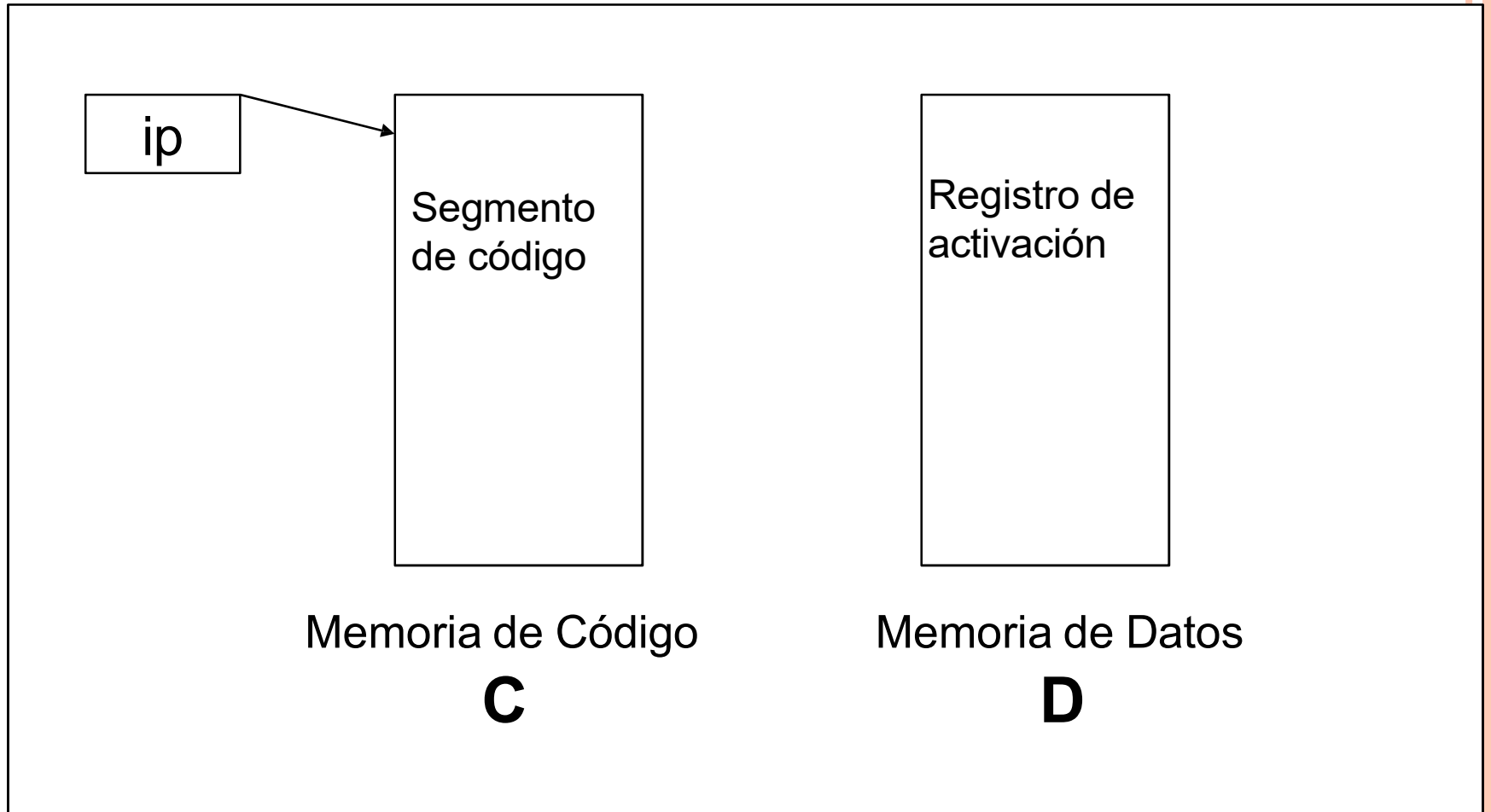
Contenido fijo

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 está 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 QUE ACCEDEN A D

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 QUE ACCEDEN A C

JUMP: bifurcación incondicional a direcciones de C

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]

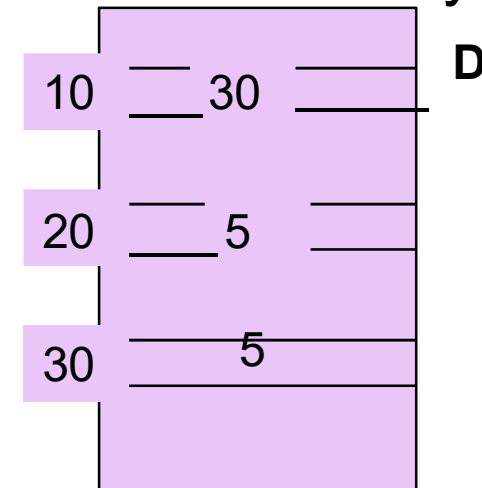
bifurca si el valor almacenado en la celda 13 es mayor que el almacenado en la celda 8

direccionamiento indirecto:

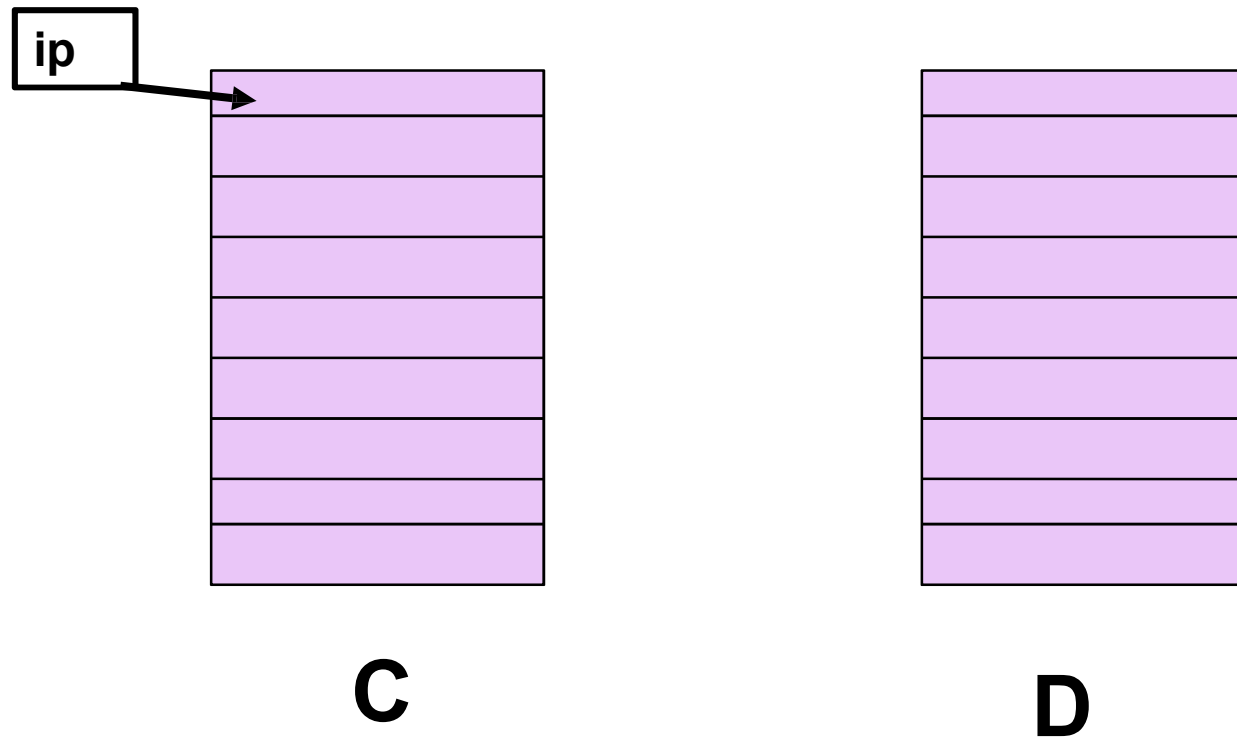
set D[10], D[20]

jump D[30]

lp= 5 posición 5 en C



PROCESADOR ABSTRACTO - MEMORIA



IP puntero a la instrucción que se está
ejecutando en el área de código



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 ya que no pueden mantener diferentes instancias.
- Lenguajes estáticos como las versiones originales FORTRAN y COBOL



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.



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**



ESQUEMAS DE EJECUCIÓN

CASOS **C1**, **C2** y **C2'**

- **Estático**
- **Basado en Pila**
- **Dinámico**

C1: CASO DE UN LENGUAJE ESTÁTICO SIMPLE

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

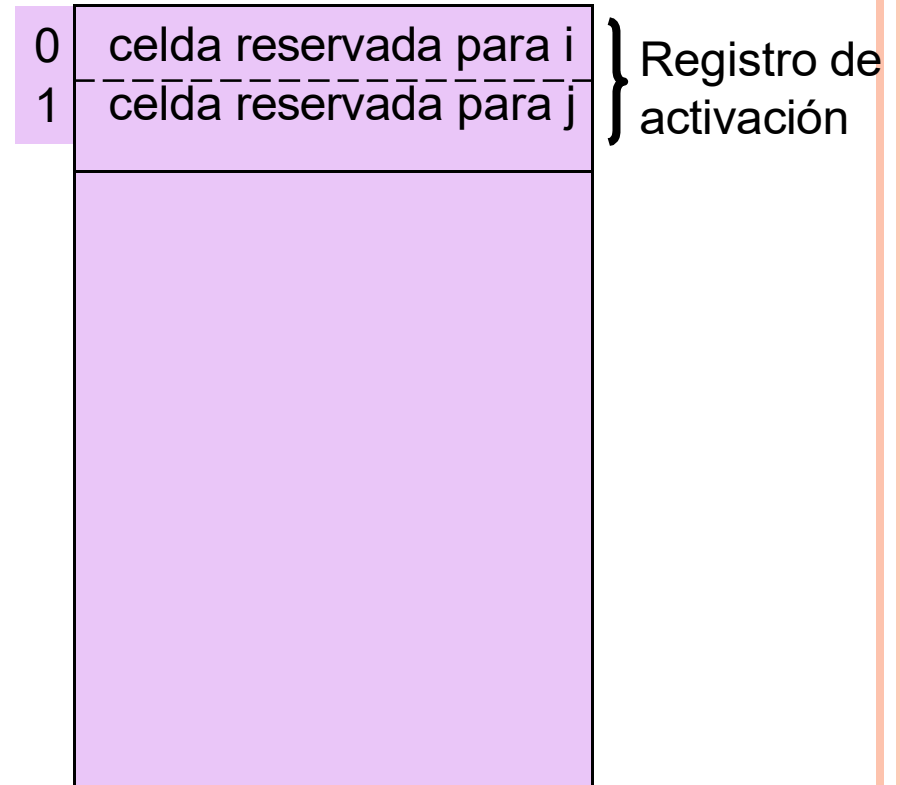
enteros
reales
arreglos
estructuras



C1

```
main()
{
    int i, j; get(i, j);
    while (i != j)
        if (i > j)
            i -= j;
        else
            j -= i;
    print(i);
}
```

Zona **DATOS**



C1

¿Cómo sería el **CÓDIGO**?

```
main()  
{
```

```
    int i, j;
```

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

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

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

```
            i -= j;
```

```
        else
```

```
            j -= i;
```

```
    print(i);
```

```
}
```

Zona **CÓDIGO**

0	set 0, read
1	set 1, read
2	jump 8 , D[0] = D[1]
3	jump 6 , D[0] <= 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



C1

C

0

0	set 0,read
1	net 1,read
2	jumpr 8, $D[0] = D[1]$
3	jumpr 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

D

0	celda reservada para i
1	celda reservada para j

**Registro de
activación**



C2: C1 + RUTINAS INTERNAS

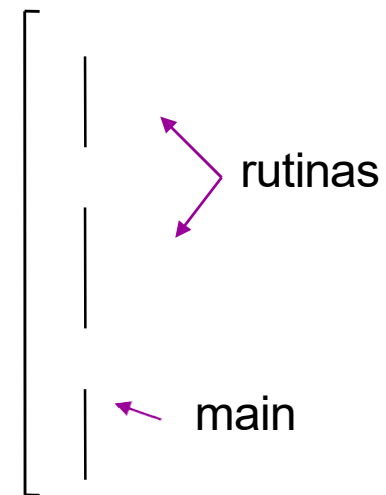
Definición de rutinas internas al main

- Programa puede tener:
 - Datos globales
 - Declaraciones de rutinas
 - Rutina principal
 - Datos locales
 - Se invoca automáticamente en ejecución
- En zona de datos: SOLO **datos locales y punto de retorno**



C2

- El tamaño de cada R.A de cada unidad puede determinarse en compilación.
- Todos los R.A pueden alocarse antes de la ejecución (alocación estática), se invoquen o no.
- Cada variable puede ser ligada a direcciones de memoria D antes de ejecución.
- Rutinas internas
 - Disjuntas: no pueden estar anidadas
 - No son recursivas
- Ambiente de las rutinas internas
 - Datos locales
 - Datos globales



rutinas

C2

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

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

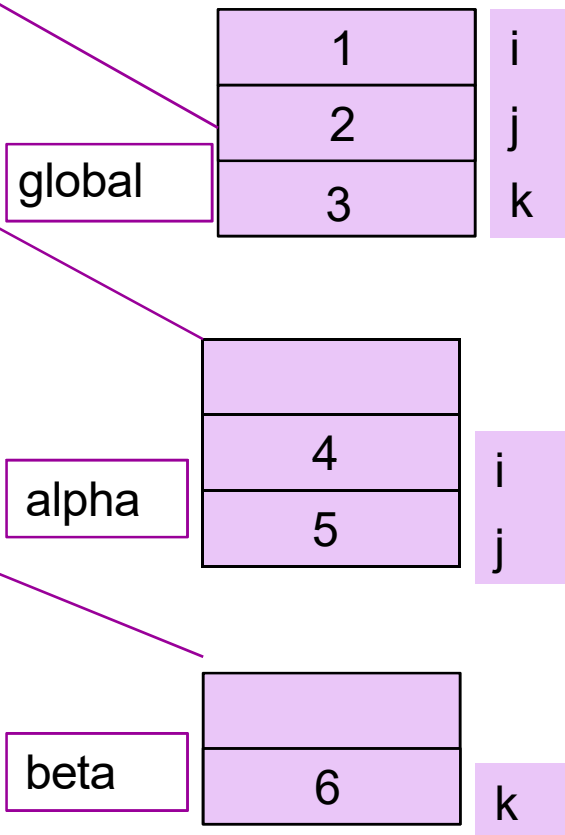
```
};  
beta()  
{
```

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

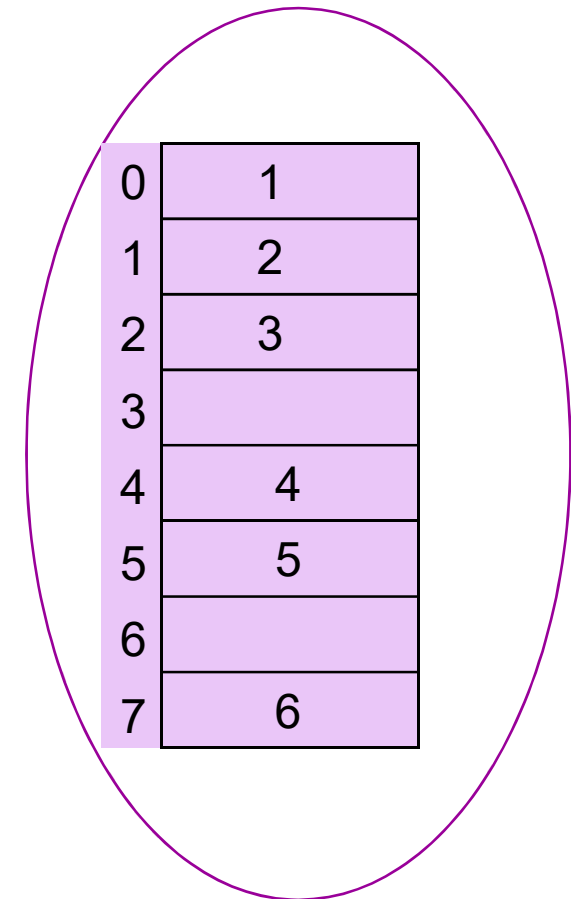
```
};  
main()  
{
```

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

```
}
```



Todo estático



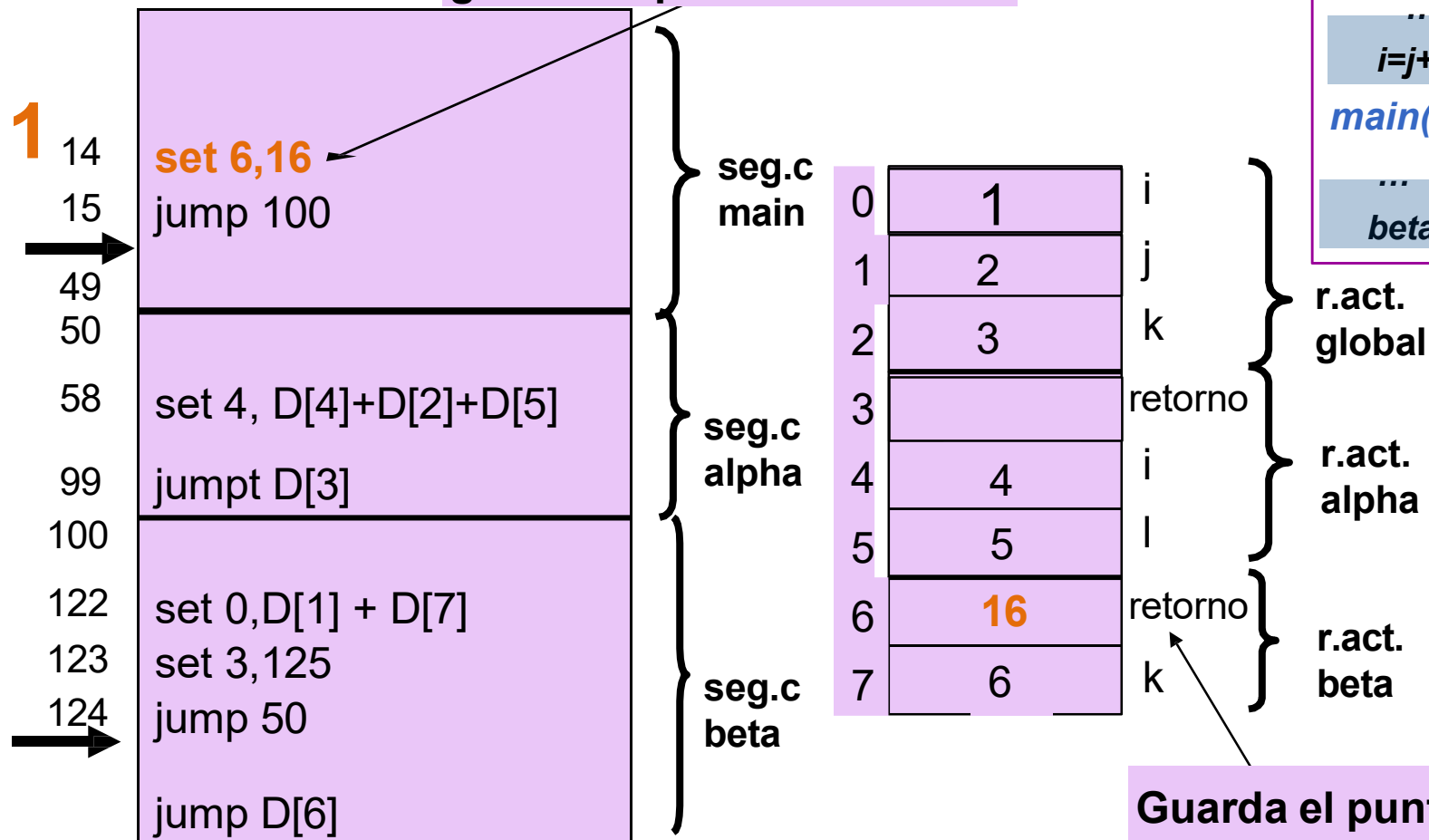
¿Qué RA se alocan en la zona de Datos?



C2: CALL-RETURN

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

Antes de saltar a Beta()
guarda el punto de retorno



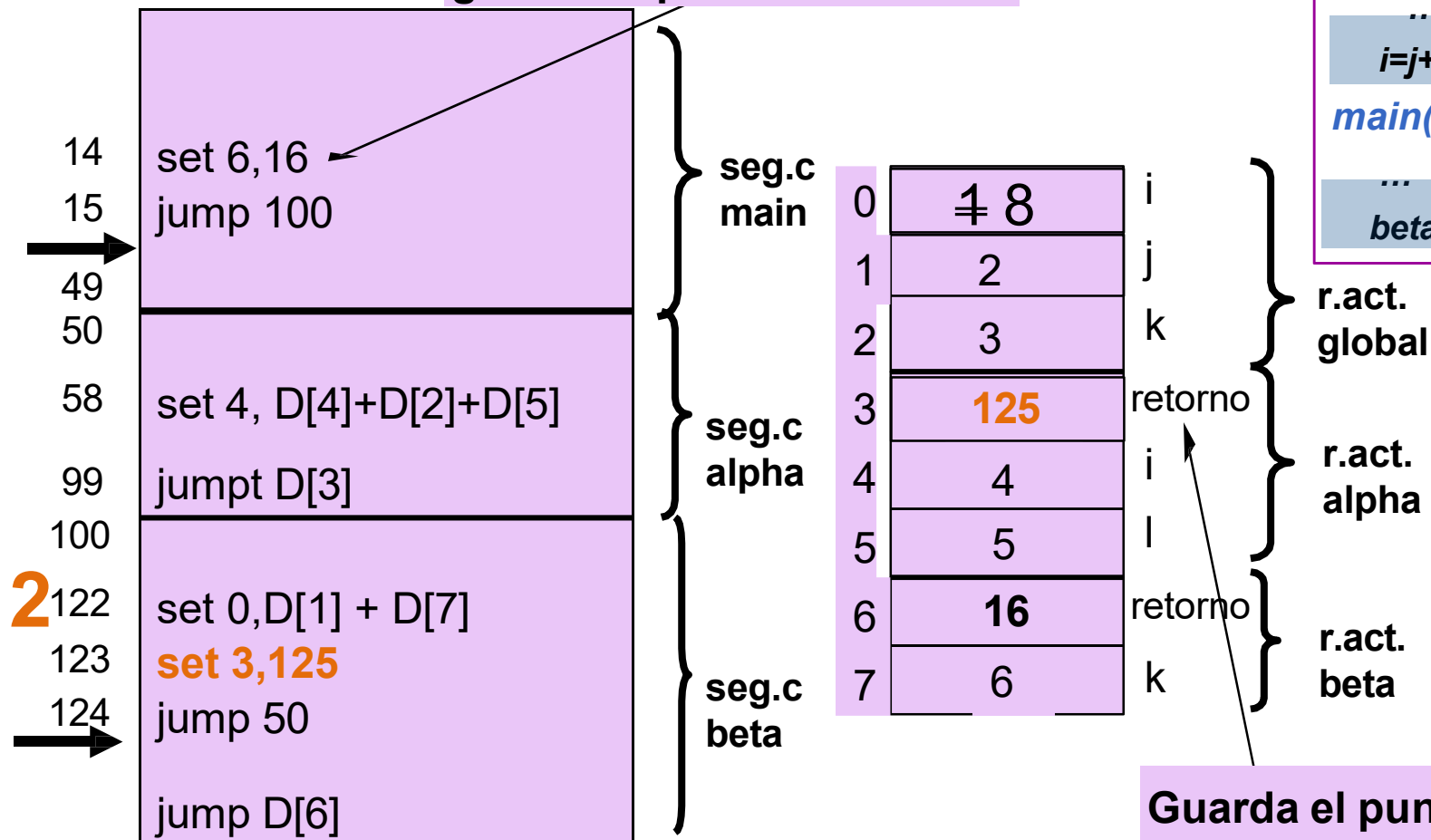
```
int i = 1, j = 2, k = 3;  
alpha() {  
    int i = 4, l = 5;  
    ...  
    i+=k+l; }  
beta() {  
    int k = 6;  
    ...  
    i=j+k; alpha(); }  
main() {  
    ...  
    beta(); } 1
```

Guarda el punto de
retorno al main

C2: CALL-RETURN

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

Antes de saltar a Beta()
guarda el punto de retorno



```
int i = 1, j = 2, k = 3;
```

```
alpha() {
```

```
    int i = 4, l = 5;
```

```
    ...
```

```
    i+=k+l;};
```

```
beta() {
```

```
    int k = 6;
```

```
    ...
```

```
    i=j+k;
```

```
main() {
```

```
    ...
```

```
    beta(); }
```

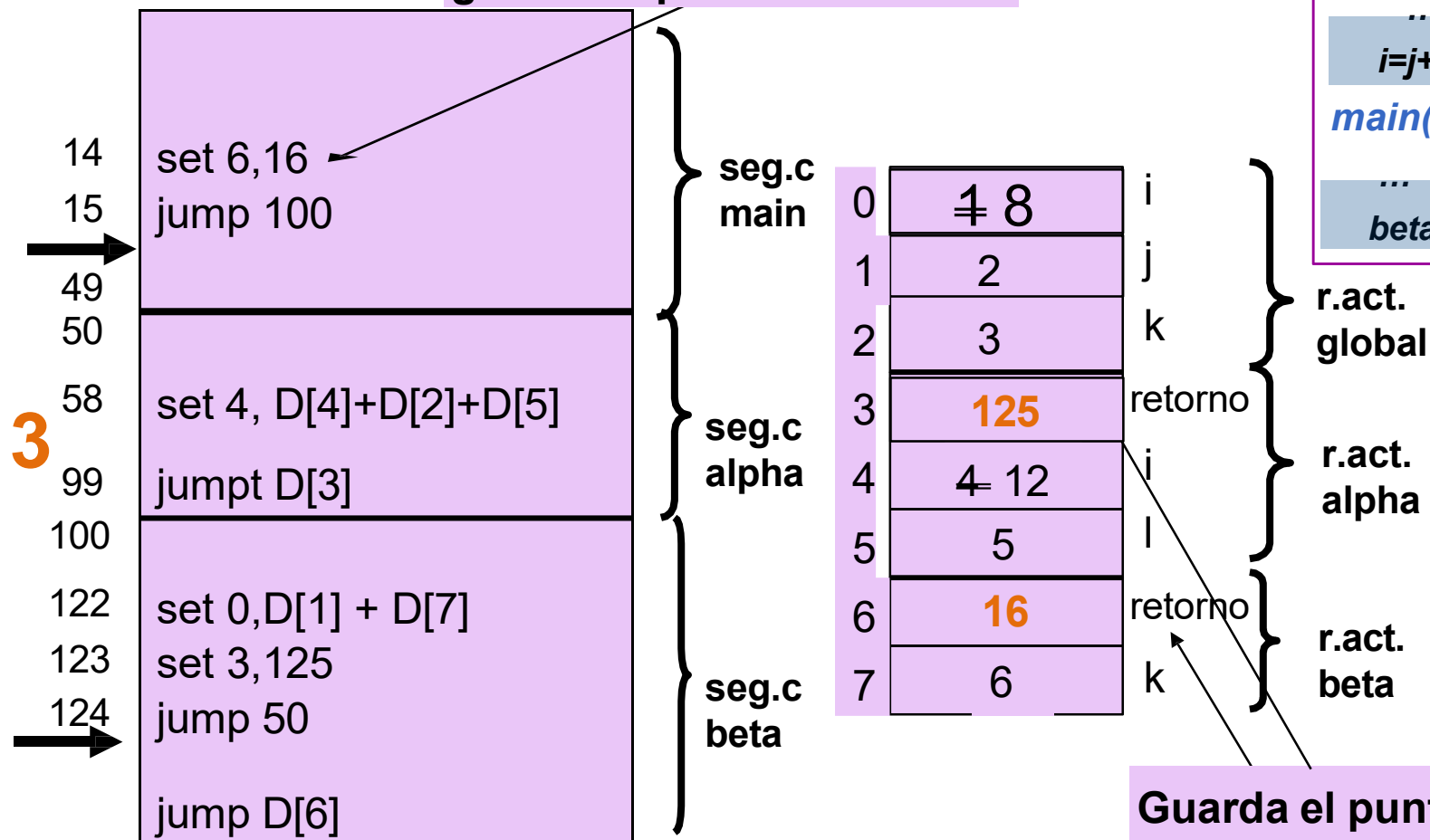
2

Guarda el punto de
retorno a beta

C2: CALL-RETURN

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

Antes de saltar a Beta()
guarda el punto de retorno

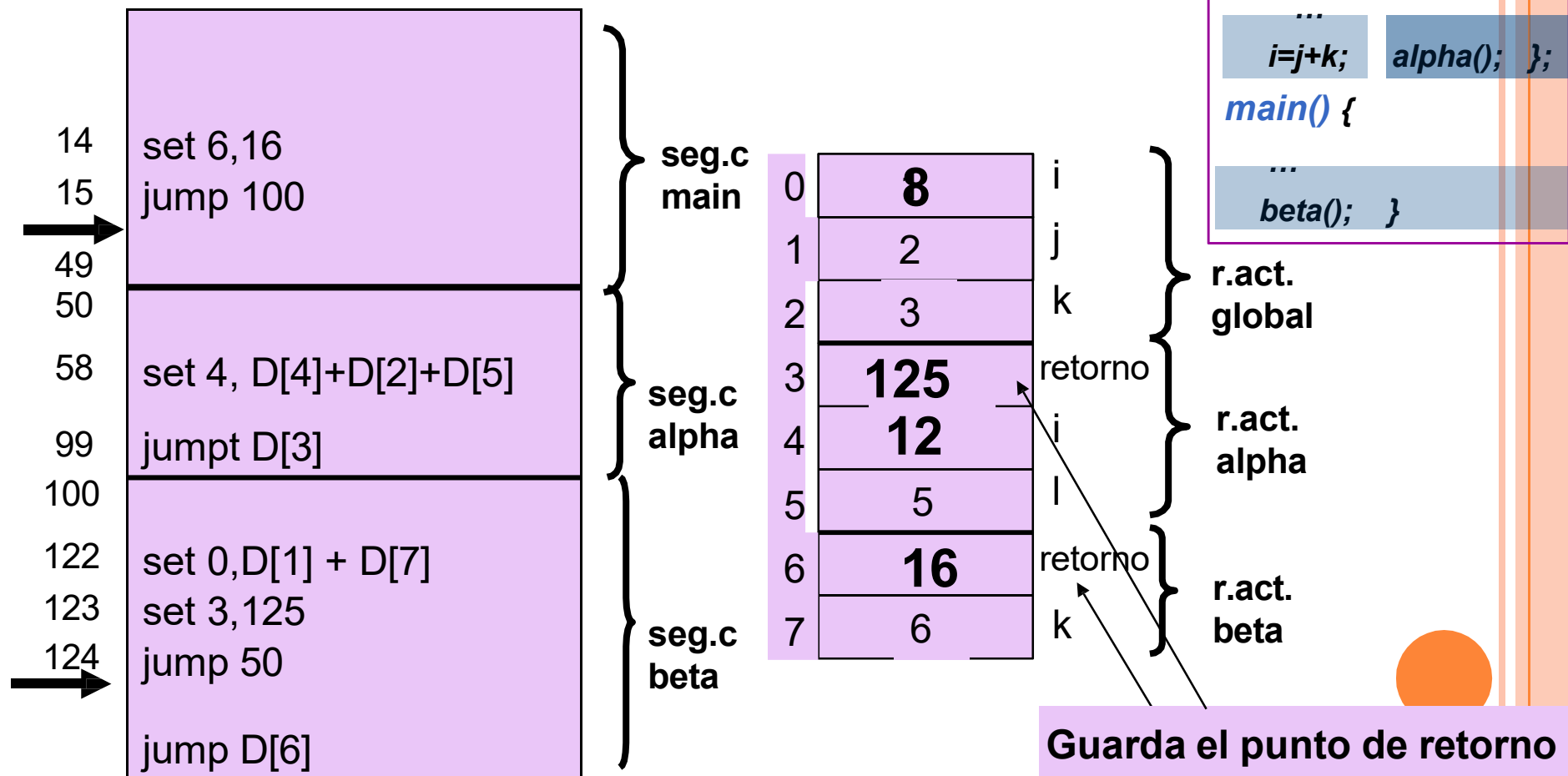


```
int i = 1, j = 2, k = 3;
alpha() {
    int i = 4, l = 5;
    ...
    i+=k+l;
}
beta() {
    int k = 6;
    ...
    i=j+k;
    alpha();
}
main() {
    ...
    beta();
}
```

Guarda el punto de retorno

C2: CALL-RETURN

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



C2' C2 PERO CON RUTINAS COMPILADAS POR SEPARADO

- En este caso, las unidades del programa se encuentran en archivos separados.
- Cada archivo es compilado por separado y en orden arbitrario.

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': RUTINAS COMPILADAS SEPARADAS

file 1

int i = 1, j = 2, k = 3;

extern beta();

main()

{...

beta();

... } ...

file 2 *extern*

int k; alpha()

{...} **file 3**

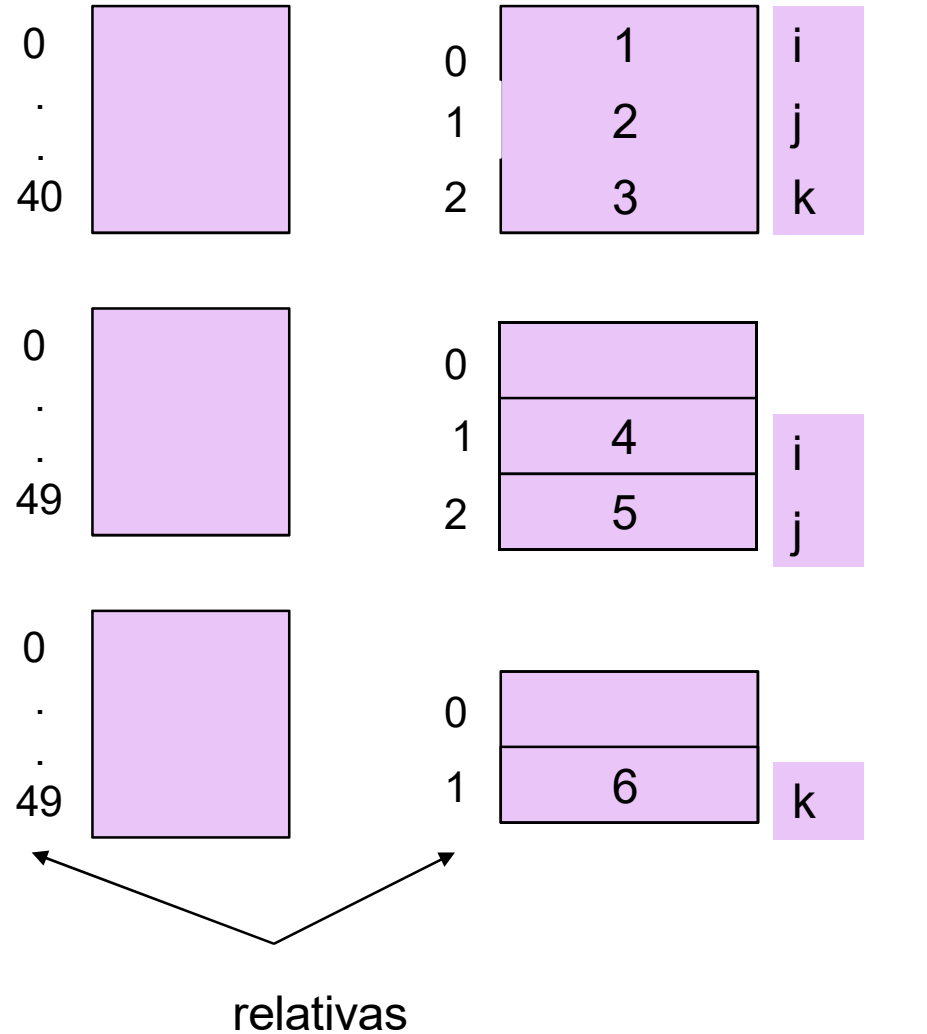
extern int i, j;

extern alpha();

beta() { }

...

alpha();...



C2' C2 PERO CON RUTINAS COMPILADAS POR SEPARADO

- En este caso, como cada unidad es compilada en distinto orden y por separado, entonces el compilador ya no puede:
 - Ligar variables locales a direcciones absolutas.
 - Tampoco variables globales pueden ligarse con sus desplazamientos en el R.A. global.
 - Las invocaciones a las rutinas no pueden ligarse con la dirección de inicio de los correspondientes segmentos de código.



C2' C2 PERO CON RUTINAS COMPILADAS POR SEPARADO

- Surge el **Linkeditor**:
 - encargado de combinar los módulos
 - ligar la información faltante
- El Linkeditor se encarga de asignar los varios segmentos de código a almacenamiento en C
- Se encarga de asignar los varios registros de activación dentro de D
- Y completa toda información faltante que el compilador no podía evaluar.
- C2 y C2' no difieren semánticamente, una vez que el linkeditor reuna todas las unidades compiladas separadamente.

