Tema 9 | Organización de la Memoria

Contenidos

Bibliografía básica

09/12/2021

[Aho90] Alfred V. Aho, Ravi Sethi, Jffrey D. Ullman
Compiladores. Principios, técnicas y herramientas. Addison-Wesley Iberoamericana 1990.

[Loud04] K. Louden
*Construcción de Compiladores: Principios y Práctica.*ITP 2004.

9.1 Introducción a la gestión de memoria

El **código intermedio** aporta lo siguiente:

- **Expresiones**: Descompone expresiones complejas en expresiones simples (unarias o binarias) introduciendo *símbolos temporales*.
- **Tabla de símbolos**: Están los símbolos que usa el programa fuente más los temporales introducidos en la fase de generación de código intermedio.

En la generación de código es necesario establecer estrategias de organización para:

- Asignar memoria a los símbolos (**enlace**).
- Acceder a los valores almacenados (**estado**).

9.1 Introducción a la gestión de memoria

Para el diseño del **mapa de memoria** hay que determinar las cuestiones siguientes:

- Permitir (o no) la **recursividad** de los procedimientos.
- Qué hacer con los **valores locales** al salir de un procedimiento.
- Uso (o no) de **valores no locales** en los procedimientos.
- Técnica de **paso de parámetros** a los procedimientos.
- Pasar (o no) **procedimientos** como **parámetros**.
- Devolver (o no) **procedimientos** como **resultados**.
- **Asignación** (o no) **dinámica de memoria** bajo el control del programa.

9.1 Introducción a la gestión de memoria

- **Semántica de los procedimientos**

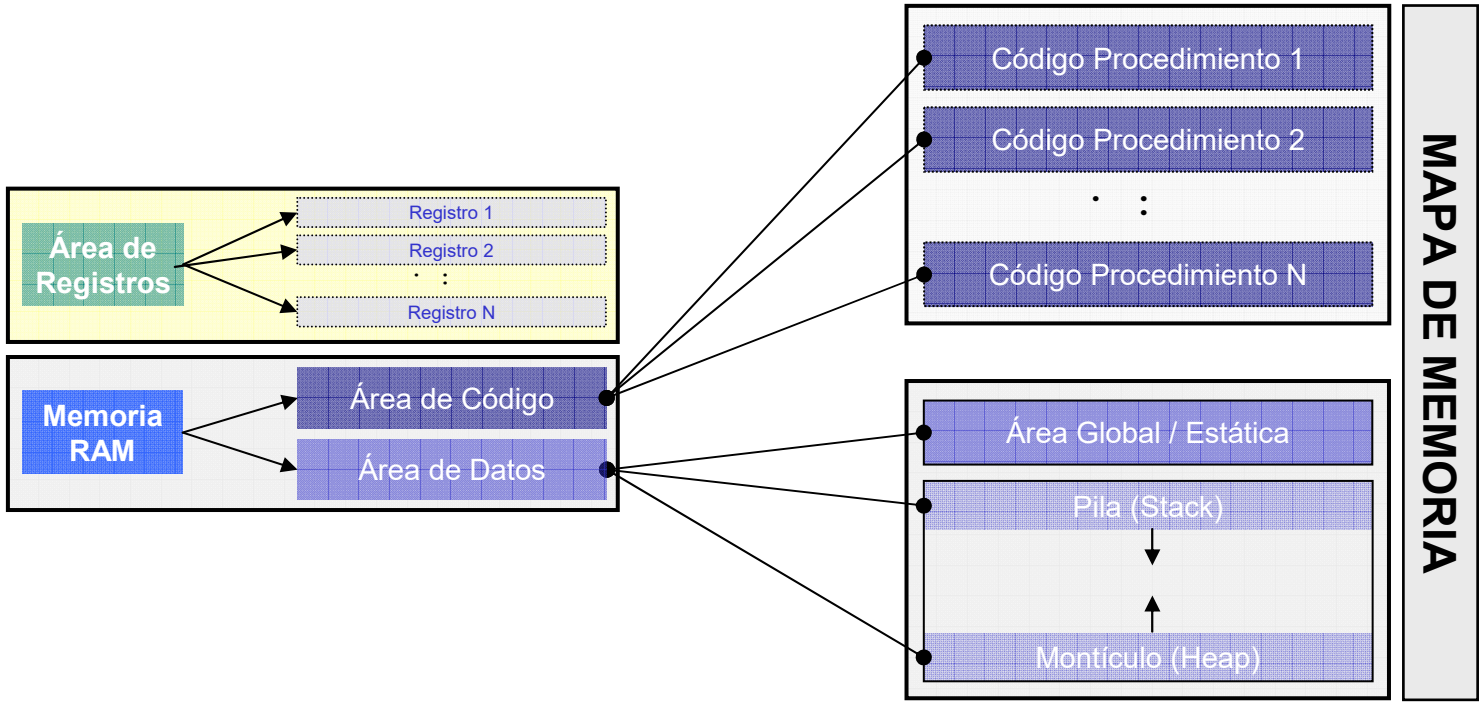
  - **Activación**: Ejecución del conjunto de instrucciones del procedimiento.

  - **Desactivación**: Salto a la instrucción siguiente inmediata que lo activó.

  - **Árbol de activación**:
    - *Nodo raíz*: Programa principal.
    - *Nodos hijos*: Procedimientos activados por el nodo ancestro o padre.
    - *Pila de activación*: Simula el control de activación. Cuando aparece un subprograma introducimos en la pila la activación del mismo y cuando finaliza se extrae la activación del tope de la pila. En cada momento, la pila representa los procedimientos que han sido activados.

9.2 Organización de la memoria durante la ejecución

La memoria de un ordenador está dividida en:



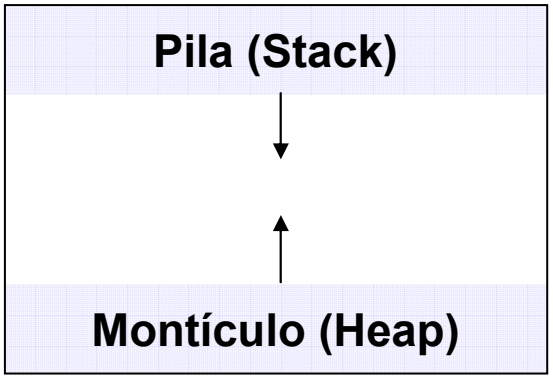
9.2 Organización de la memoria durante la ejecución

Esta distribución obedece a lo siguiente:

- El **punto de entrada** de cada **procedimiento** se conoce en tiempo de compilación.
- **Ciertos datos** pueden asignarse en ubicaciones fijas de memoria antes de la ejecución
  - En **`Fortran77`** todos los datos son de esta forma, es decir, estáticos.
  - En **`Pascal`** serían las variables globales.
  - En **`C`** serían las variables estáticas y externas.
- El área de datos global de los procedimientos también es conocido en tiempo de compilación.
- El resto de información que depende de la ejecución del programa deberá almacenarse en otra zona denominada **dinámica**.

9.2 Organización de la memoria durante la ejecución

Una organización típica del área de memoria para asignación dinámica divide la memoria en dos áreas:



- **Pila (Stack):** Datos cuya asignación se corresponde con la gestión típica de pila de arquitecturas típicas en cuanto a llamadas y retornos de procedimientos.

- **Montículo (Heap):** Área reservada para asignación y liberación de datos cuyo control se realiza desde el propio programa.

9.2 Organización de la memoria durante la ejecución

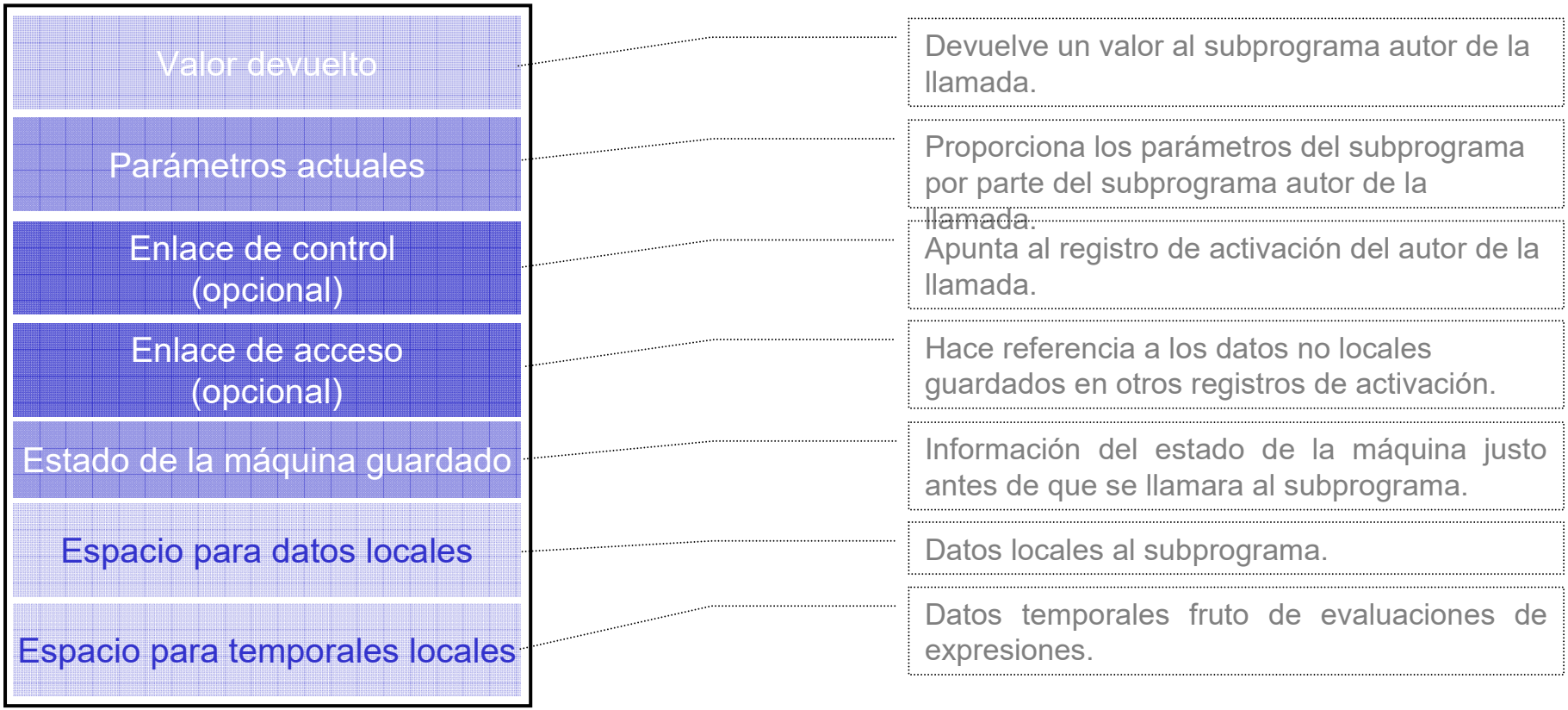
Un elemento importante de la asignación de memoria es el **registro de activación** que es un bloque contiguo de información necesaria para cada ejecución de un procedimiento.

Un **registro de activación** debe contener, como mínimo, la siguiente información:

Detalles específicos del registro de activación dependerán de la arquitectura de la máquina objeto, de las propiedades del lenguaje que se esté compilando y de las preferencias establecidas en la construcción del propio traductor.

9.2 Organización de la memoria durante la ejecución

En **[Aho90]** se puede ver una composición general de registro de activación:



9.2 Organización de la memoria durante la ejecución

Registro de activación

- El tamaño de cada uno de los campos del **registro de activación** es conocido cuando es activado. En general, el tamaño es conocido en tiempo de compilación, salvo los datos tipo array con dimensión en los parámetros actuales.

- Dependiendo del tipo de lenguaje, los **registros de activación** pueden ser asignados en el área estática, como en `Fortran77`, en el área de la pila, como en `C` y `Pascal`, o en el montículo, como es el caso de `Lisp`.

- Los registros del procesador también forman parte de la estructura de la memoria. Si una arquitectura posee multitud de registros (procesadores RISC más recientes), todo el área estática y los **registros de activación** completos pueden ser conservados en registros.

- Los procesadores poseen **registros** de propósito específico: *contador de programa*, *apuntador de pila*, *apuntador de registro de activación*, *apuntador de argumento*, ...

9.2 Organización de la memoria durante la ejecución

Estrategias para la asignación de memoria

1. **Asignación estática**: Dispone la memoria para todos los objetos de datos en tiempo de compilación.

2. **Asignación por medio de pila**: Trata la memoria en ejecución como una pila (stack).

3. **Asignación por medio de montículo**: Asigna y libera la memoria conforme se necesita durante la ejecución a partir de un área de datos conocida como montículo (heap).

9.3 Organización estática

Se caracteriza por lo siguiente:

- Los nombres se establecen en memoria en tiempo de compilación.
- No es necesario un paquete de apoyo para la ejecución.
- Todos los datos son estáticos y permanecen fijos en la memoria mientras dure la ejecución del programa.

Tiene las siguientes limitaciones:

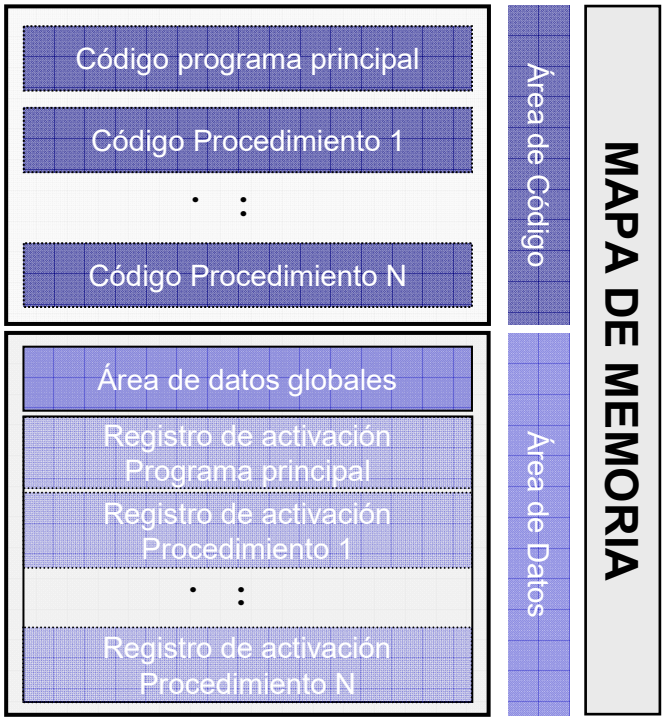
- El tamaño de los objetos de datos y las posiciones de memoria deben conocerse en tiempo de compilación.
- No se permiten llamadas recursivas.
- No permite creación dinámica de datos al carecer de mecanismos para asignación y liberación de memoria durante la ejecución.

9.3 Organización estática

Hay relativamente poco gasto general en términos de administración de información.

La memoria se distribuye en dos áreas:

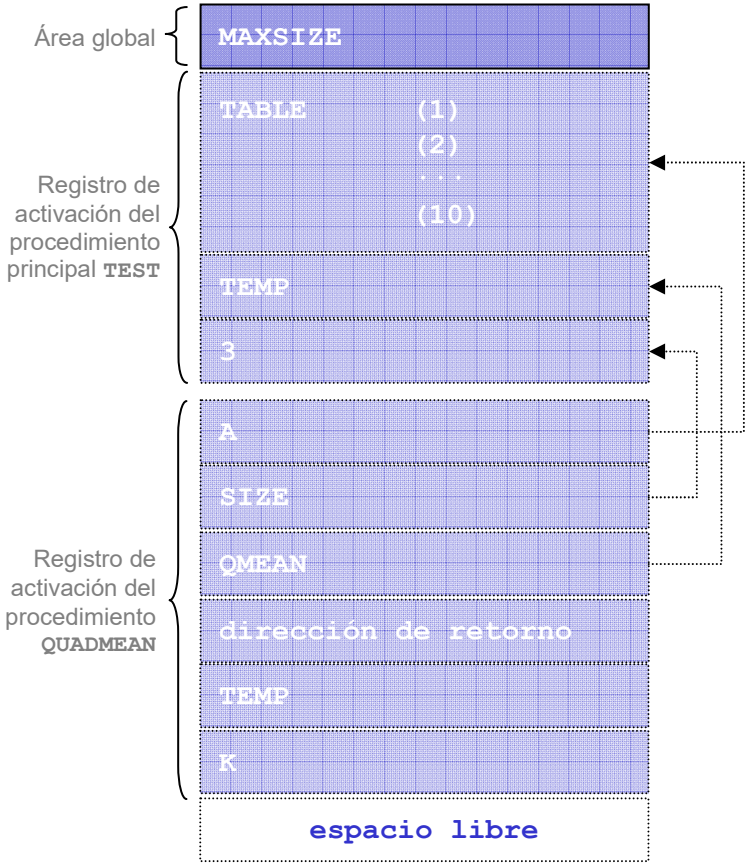
- **Área de código**: Código del programa principal y de cada uno de los procedimientos de forma consecutiva.
- **Área de datos**: Datos globales y registros de activación para cada uno de los procedimientos de forma consecutiva.



9.3 Organización estática

Ejemplo 9.1: Mapa de memoria del área de datos de un programa en **Fortran77** constituido por un módulo principal y un solo procedimiento adicional.

```
      PROGRAM TEST
      COMMON MAXSIZE
      INTEGER MAXSIZE
      REAL TABLE(10),TEMP
      MAXSIZE = 10
      READ *, TABLE(1), TABLE(2), TABLE(3)
      CALL QUADMEAN (TABLE, 3, TEMP)
      PRINT *, TEMP
      END

      SUBROUTINE QUADMEAN (A, SIZE, QMEAN)
      COMMON MAXSIZE
      INTEGER MAXSIZE, SIZE
      REAL A(SIZE), QMEAN, TEMP
      INTEGER K
      TEMP = 0.0
      IF ((SIZE .GT. MAXSIZE) .OR. (SIZE .LT. 1)) GOTO 99
      DO 10 K = 1, SIZE
         TEMP = TEMP + A(K)*A(K)
10    CONTINUE
99    QMEAN = SQRT (TEMP/SIZE)
      RETURN
      END
```



9.4 Organización basada en pila (stack)

En lenguajes en el que se permitan las llamadas recursivas, los registros de activación no pueden ser asignados de manera estática.

Pila de registros de activación: La gestión de llamadas se realiza mediante la gestión de una pila. También se la conoce como pila de ejecución o pila de llamadas.

Además de almacenar los registros de activación es necesario contemplar la secuencia de llamadas a los procedimientos.

En estas condiciones, según sea el ámbito de los procedimientos tendremos diferentes gestiones:

- Gestión de pila sin procedimientos locales (procedimientos globales).
- Gestión de pila con procedimientos locales (procedimientos anidados).

9.4.1 Pila sin procedimientos anidados

Gestión adecuada en lenguajes donde los procedimientos sean de ámbito global (como el lenguaje C).

La gestión de la memoria mediante pila requiere dos cosas:

- **Apuntador** hacia el registro de activación actual para permitir acceso a variables locales.
- **Registro de la posición** o tamaño del registro de activación inmediatamente precedente (el registro de activación del que lo llamó).

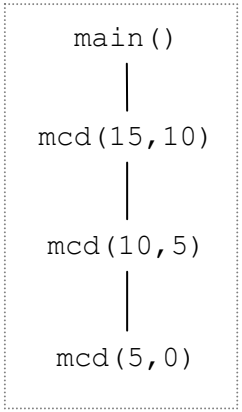
9.4.1 Pila sin procedimientos anidados

Ejemplo 9.2: Mapa de memoria del área de datos de un programa en **C** que implementa de forma recursiva el algoritmo de Euclides para calcular el máximo común divisor de dos enteros no negativos.

```
#include <stdio.h>

int x, y ;

int mcd ( int u, int v )
{
   if (v == 0)
      return u ;
   else
      return mcd (v, u % v) ;
}


int main()
{
   scanf ("%d%d", &x, &y) ;
   printf ("%d\n", mcd(x,y)) ;
   return 0 ;
}
```

Ejecución:

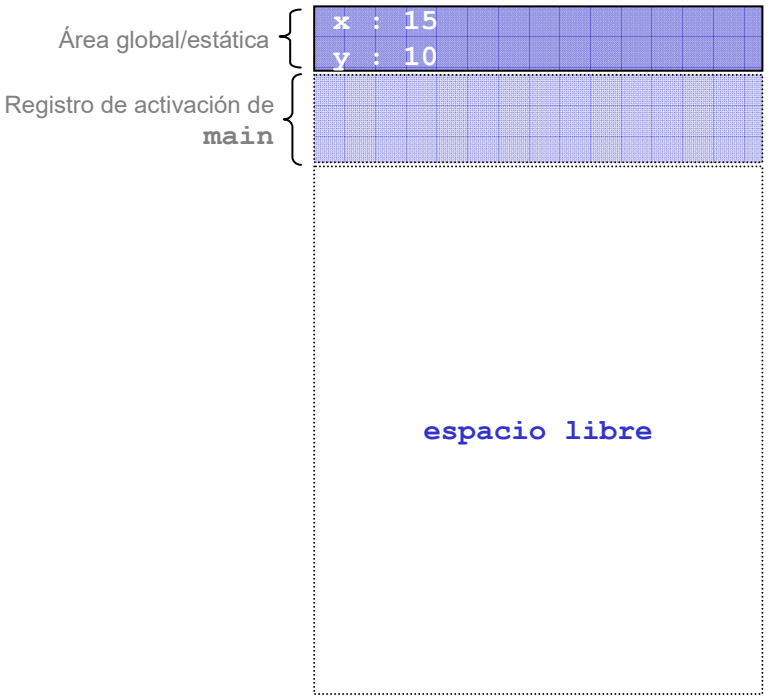
- El usuario introduce los valores **15** y **10**
- Primera llamada a **mcd**: **mcd(15,10)**
- Segunda llamada a **mcd**: **mcd(10,5)**
- Tercera llamada a **mcd**: **mcd(5,0)**



9.4.1 Pila sin procedimientos anidados

- El usuario introduce los valores **15** y **10**

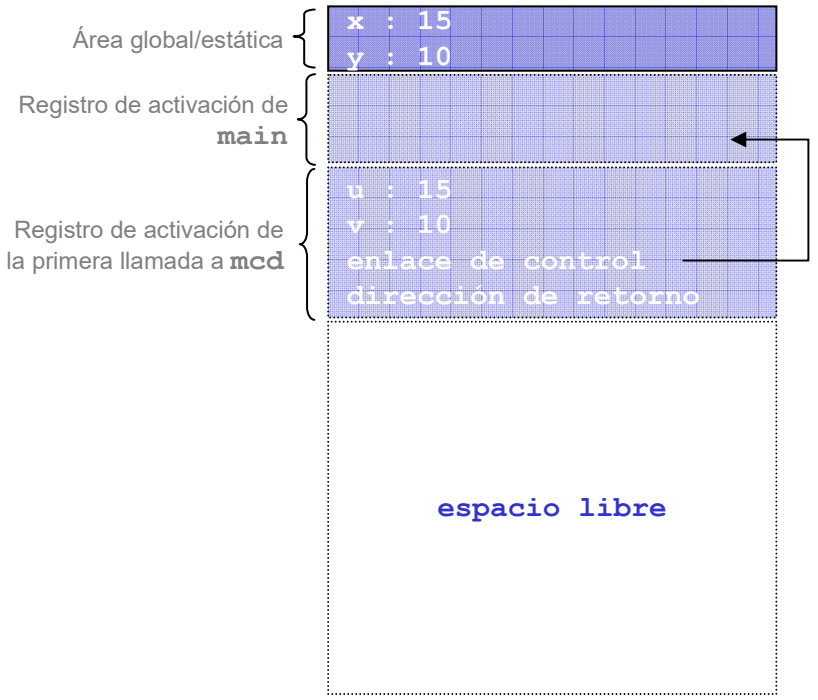
```
#include <stdio.h>

int x, y ;

int mcd ( int u, int v )
{
   if (v == 0)
      return u ;
   else
      return mcd (v, u % v) ;
}


int main()
{
   scanf ("%d%d", &x, &y) ;
   printf ("%d\n", mcd(x,y)) ;
   return 0 ;
}
```

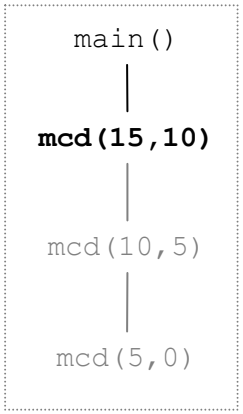


9.4.1 Pila sin procedimientos anidados

- Primera llamada a **mcd**: **mcd(15,10)**

```
#include <stdio.h>

int x, y ;

int mcd ( int u, int v )
{
    if (v == 0)
       return u ;
    else
       return mcd (v, u % v) ;
}


int main()
{
    scanf ("%d%d", &x, &y) ;
    printf ("%d\n", mcd(x,y)) ;
    return 0 ;
}
```

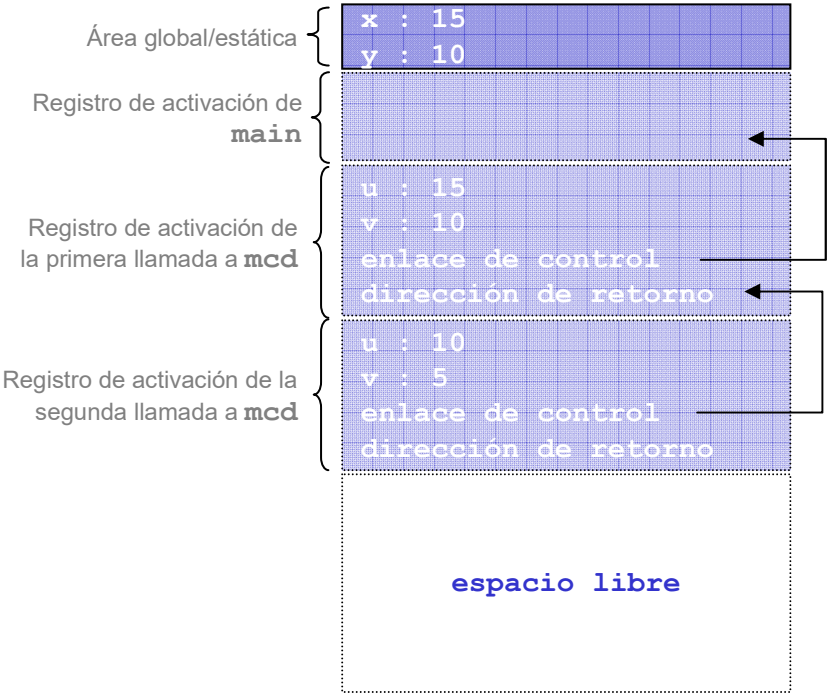


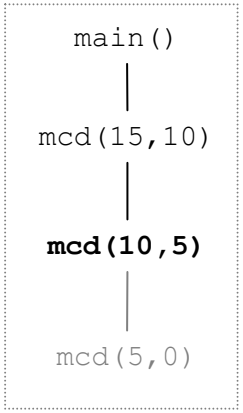


9.4.1 Pila sin procedimientos anidados

- Segunda llamada a **mcd**: **mcd(10,5)**

```
#include <stdio.h>

int x, y ;

int mcd ( int u, int v )
{
    if (v == 0)
       return u ;
    else
       return mcd (v, u % v) ;
}


int main()
{
    scanf ("%d%d", &x, &y) ;
    printf ("%d\n", mcd(x,y)) ;
    return 0 ;
}
```

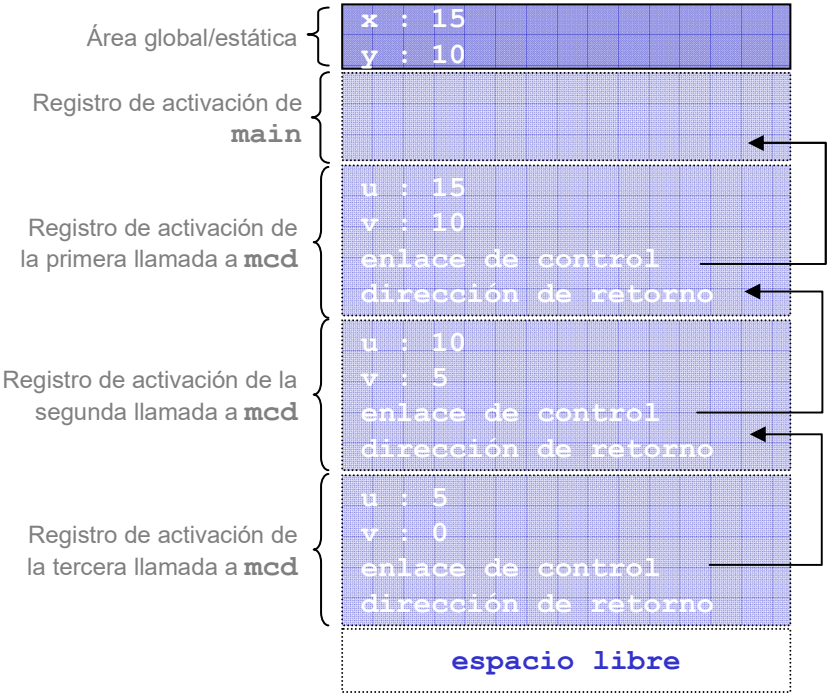


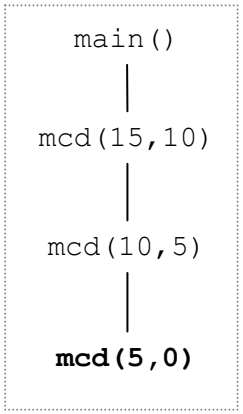


9.4.1 Pila sin procedimientos anidados

- Tercera llamada a **mcd**: **mcd(5,0)**

```
#include <stdio.h>

int x, y ;

int mcd ( int u, int v )
{
    if (v == 0)
       return u ;
    else
       return mcd (v, u % v) ;
}


int main()
{
    scanf ("%d%d", &x, &y) ;
    printf ("%d\n", mcd(x,y)) ;
    return 0 ;
}
```





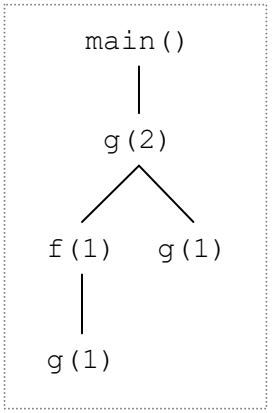
9.4.1 Pila sin procedimientos anidados

Ejemplo 9.3: Mapa de memoria del área de datos de un programa en **C** con llamadas complejas.

```
int x = 2 ;

void f (int n)
{
   static int x = 1 ;
   g(n);
   x-- ;
}

void g (int m)
{
   int y = m - 1 ;
   if (y > 0)
   {
      f(y) ;
      x-- ;
      g(y) ;
   }
}

int main()
{
   g(x) ; return 0 ;
}
```

Ejecución:

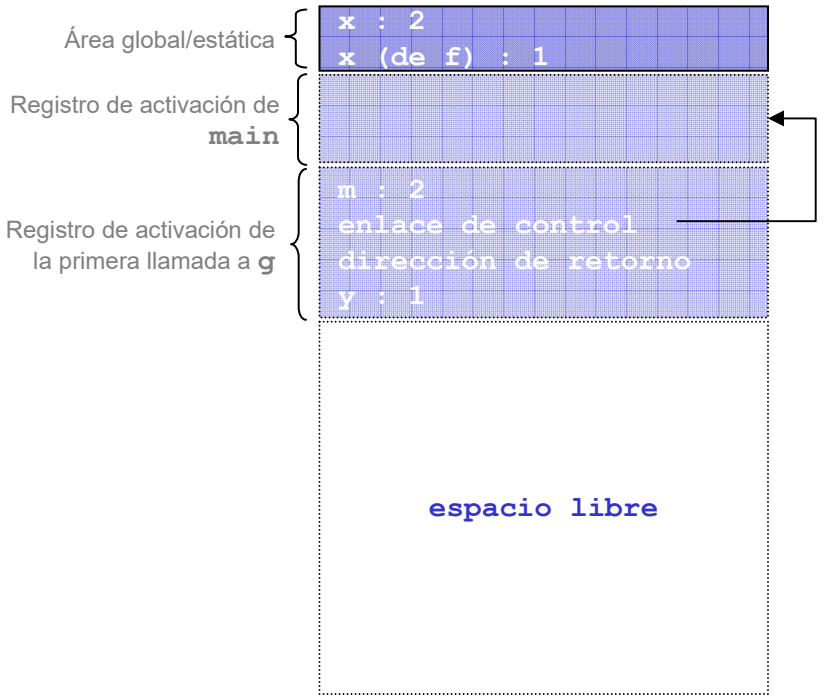
- Primera llamada a **g** (desde **main**): **g(2)**
- Primera llamada a **f** (desde **g**): **f(1)**
- Segunda llamada a **g** (desde **f**): **g(1)**
- Tercera llamada a **g** (desde **f**): **g(1)**

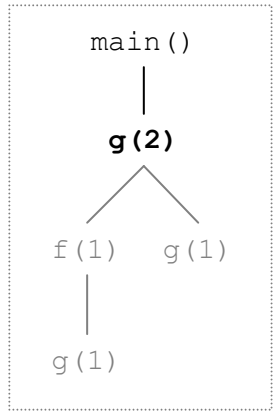


9.4.1 Pila sin procedimientos anidados

- Primera llamada a **g** (desde main): **g(2)**

```
int x = 2 ;

void f (int n)
{
    static int x = 1 ;
    g(n);
    x-- ;
}

void g (int m)
{
    int y = m - 1 ;
    if (y > 0)
    {
        f(y) ;
        x-- ;
        g(y) ;
    }
}

int main()
{
    g(x) ; return 0 ;
}
```

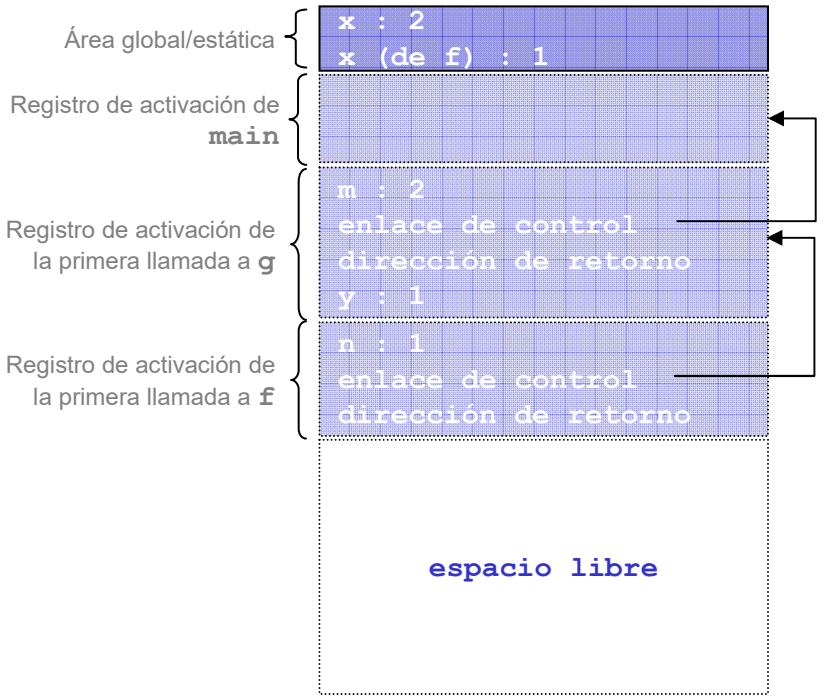


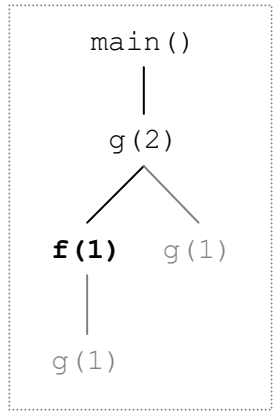


9.4.1 Pila sin procedimientos anidados

- Primera llamada a **f** (desde g): **f(1)**

```
int x = 2 ;

void f (int n)
{
   static int x = 1 ;
   g(n);
   x-- ;
}

void g (int m)
{
   int y = m - 1 ;
   if (y > 0)
   {
      f(y) ;
      x-- ;
      g(y) ;
   }
}

int main()
{
   g(x) ; return 0 ;
}
```

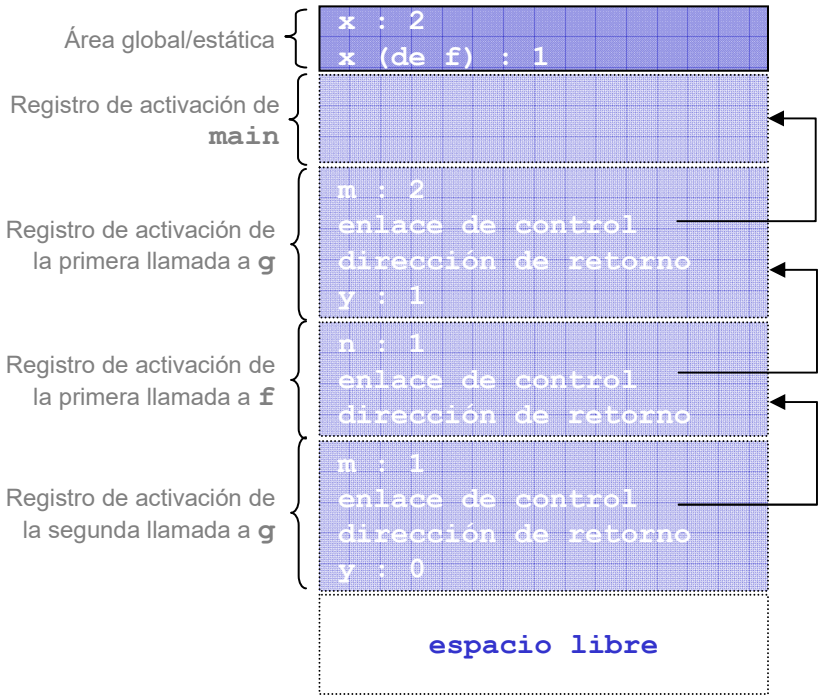


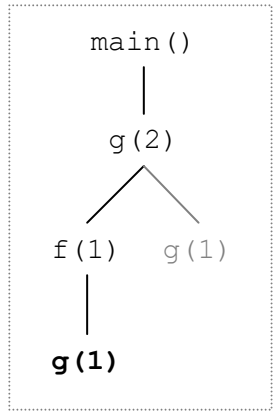


9.4.1 Pila sin procedimientos anidados

- Segunda llamada a **g** (desde **f**): **g(1)**

```
int x = 2 ;

void f (int n)
{
   static int x = 1 ;
   g(n);
   x-- ;
}

void g (int m)
{
   int y = m - 1 ;
   if (y > 0)
   {
      f(y) ;
      x-- ;
      g(y) ;
   }
}

int main()
{
   g(x) ; return 0 ;
}
```

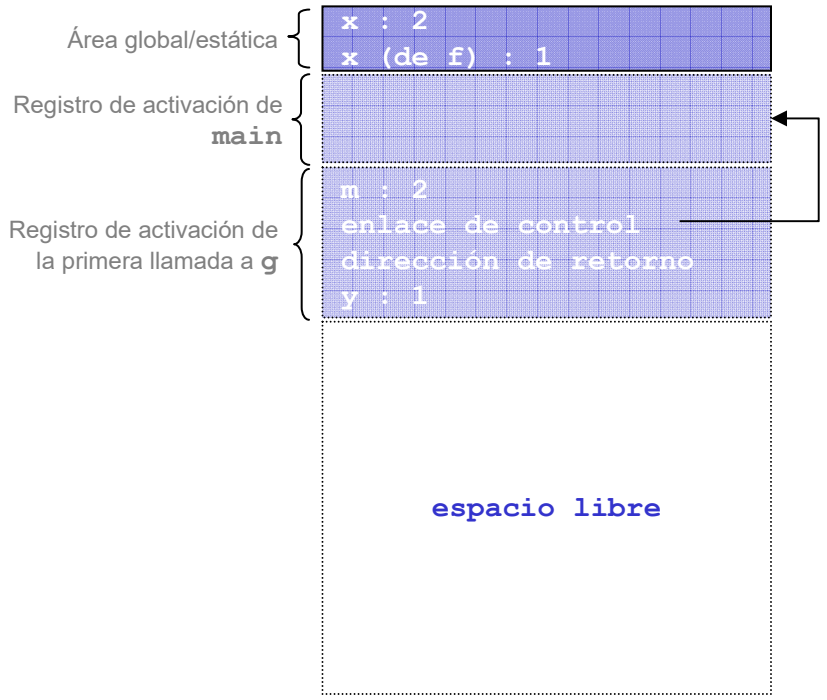


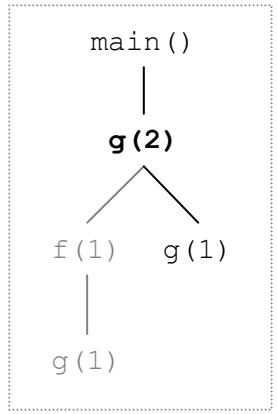


9.4.1 Pila sin procedimientos anidados

- Antes de la tercera llamada a **g** (desde **g**): **g(1)**

```
int x = 2 ;

void f (int n)
{
   static int x = 1 ;
   g(n);
   x-- ;
}

void g (int m)
{
   int y = m - 1 ;
   if (y > 0)
   {
      f(y) ;
      x-- ;
      g(y) ;
   }
}

int main()
{
   g(x) ; return 0 ;
}
```

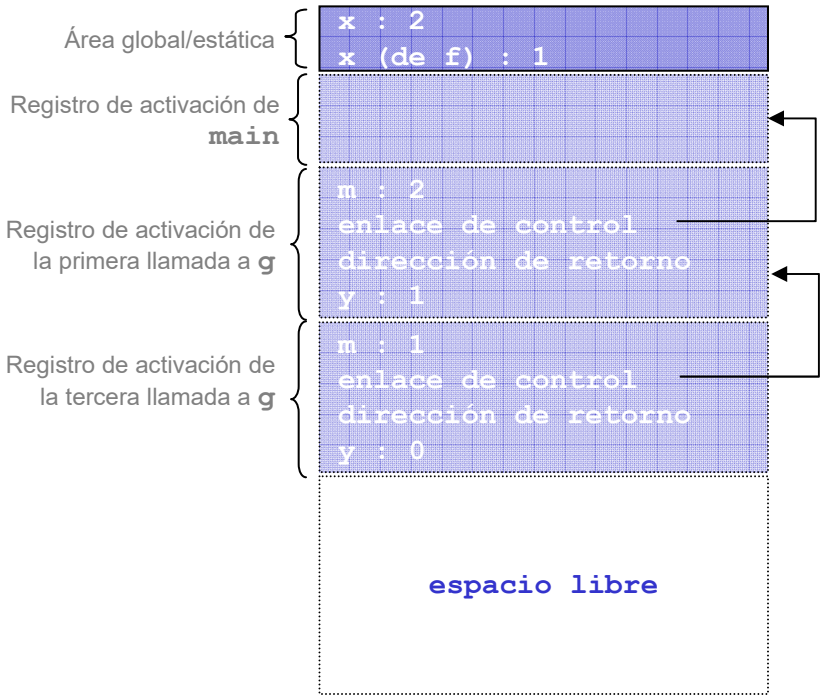


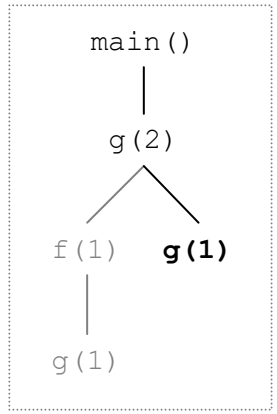


9.4.1 Pila sin procedimientos anidados

- Después de la tercera llamada a **g** (desde **g**): **g(1)**

```
int x = 2 ;

void f (int n)
{
   static int x = 1 ;
   g(n);
   x-- ;
}

void g (int m)
{
   int y = m - 1 ;
   if (y > 0)
   {
      f(y) ;
      x-- ;
      g(y) ;
   }
}

int main()
{
   g(x) ; return 0 ;
}
```





9.4.1 Pila sin procedimientos anidados

Cuestiones relacionadas con esta forma de organizar la memoria:

- **Árbol de activación**: Representación jerárquica de las llamadas a procedimientos donde cada registro de activación se convierte en un nodo en este árbol.
- **Acceso a nombres**: En una organización mediante pila, el acceso a variables locales y parámetros no se realizan mediante direcciones fijas sino a partir de desplazamientos desde el apuntador del marco actual.
- **Secuencia de llamada**: Asigna un registro de activación e introduce información en sus campos.
- **Secuencia de retorno**: Restablece el estado de la máquina para el procedimiento que llama y, de ese modo, continuar su ejecución.

9.4.1 Pila sin procedimientos anidados

No hay una frontera clara entre quién hace las tareas correspondientes a la secuencia de llamada. Un reparto de las tareas válido podría ser el siguiente:

- **Procedimiento llamador**:
  - Evalúa los parámetros reales.
  - Almacena la dirección de retorno y el valor anterior del tope de la pila en el registro de activación del subprograma llamado.

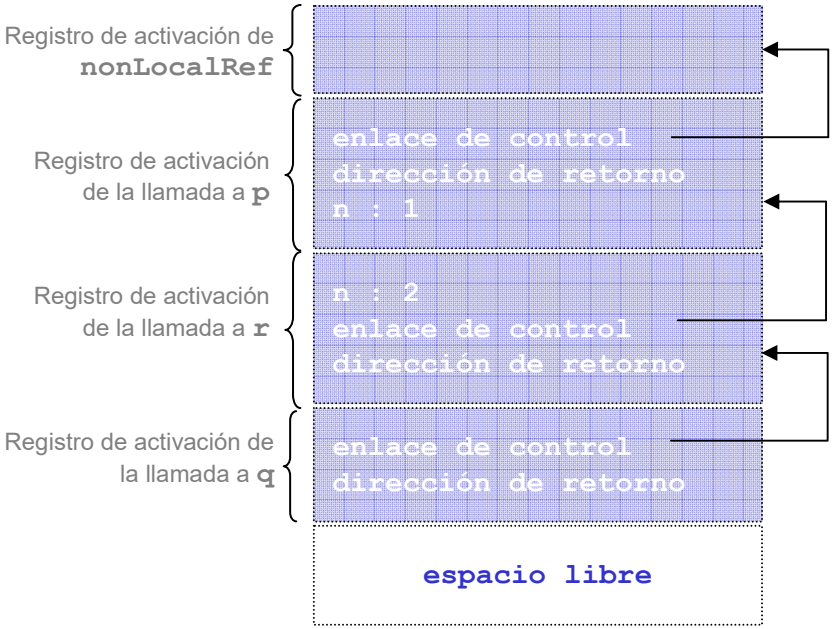
- **Procedimiento llamado**:
  - Guarda información de los registros y del estado de la máquina.
  - Asigna valores iniciales a sus datos locales y comienza la ejecución.
  - Coloca un valor devuelto a continuación del registro de activación del que lo llama.
  - Restablece el tope de la pila y salta a la dirección de retorno mediante la información del campo de estado.

El llamador puede copiar el valor devuelto por el llamado en su propio registro de activación.

9.4.2 Pila con procedimientos anidados

Gestión adecuada en lenguajes en que se permite anidamiento de procedimientos (como en **PASCAL**). La gestión anterior no contempla referencias **no locales** y **no globales**.

Ejemplo 9.4: Mapa de memoria de un programa en **Pascal** con procedimientos anidados.




```
program nonLocalRef;
procedure p ;
var n: integer ;
    procedure q ;
    begin
      (* una referencia a n es
         ahora no local y no global *)
      ...
    end ; (* q *)
    procedure r (n: integer) ;
    begin
       q ;
    end ; (* r *)
begin (* p *)
  n:= 1; r(2) ;
end; (* p *)

begin (* main *)
  p ;
end.
```

9.4.2 Pila con procedimientos anidados

Solución en el ámbito dinámico

Conservar las tablas de símbolos locales de cada procedimiento durante la ejecución para buscar un determinado identificador y, en su caso, calcular su desplazamiento.

Problema → Se soluciona de forma dinámica algo que puede conocerse en tiempo de compilación.

Solución en el ámbito estático

Agregar un elemento de información extra de administración que se conoce como **enlace de acceso** (en **[Loud04]** lo traducen como *vínculo de acceso*).

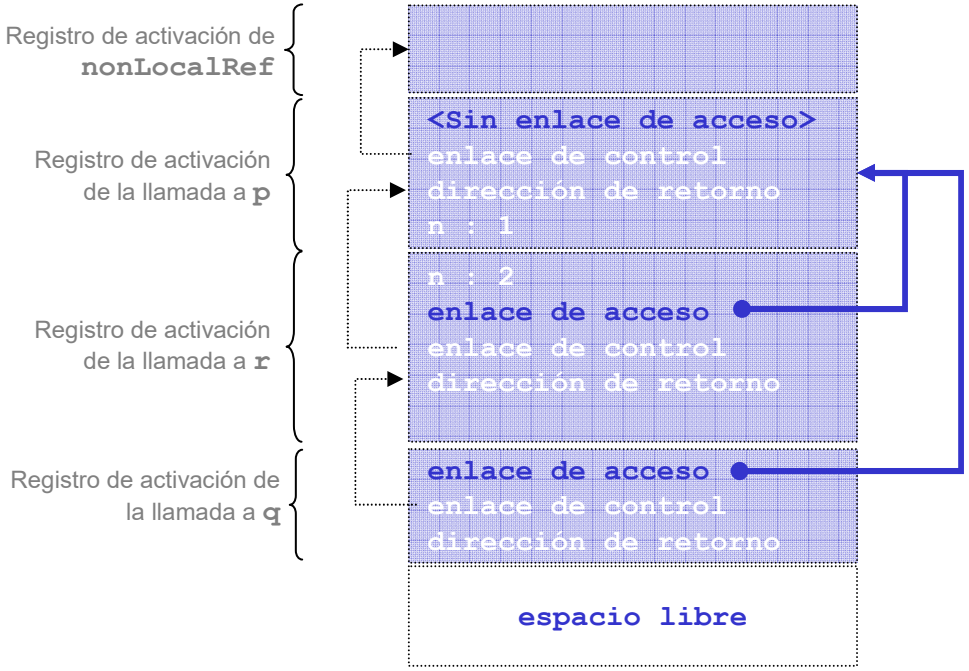
9.4.2 Pila con procedimientos anidados

Enlace de acceso

- Es una forma directa de implementar el ámbito cuando hay anidamiento de procedimientos.
- A cada registro de activación de los procedimientos se le añade un campo para el **enlace de acceso** que apunta al procedimiento que lo abarca.
- Para acceder a un nombre no local cuando se usa el enlace de acceso es necesario conocer la profundidad del anidamiento.

9.4.2 Pila con procedimientos anidados

Ejemplo 9.5: Mapa de memoria, usando enlaces de acceso, del Ejemplo 9.4.




```
program nonLocalRef;
procedure p ;
var n: integer ;
    procedure q ;
    begin
      (* una referencia a n es
         ahora no local y no global *)
      ...
    end ; (* q *)
    procedure r (n: integer) ;
    begin
       q ;
    end ; (* r *)
begin (* p *)
  n:= 1; r(2) ;
end; (* p *)

begin (* main *)
  p ;
end.
```

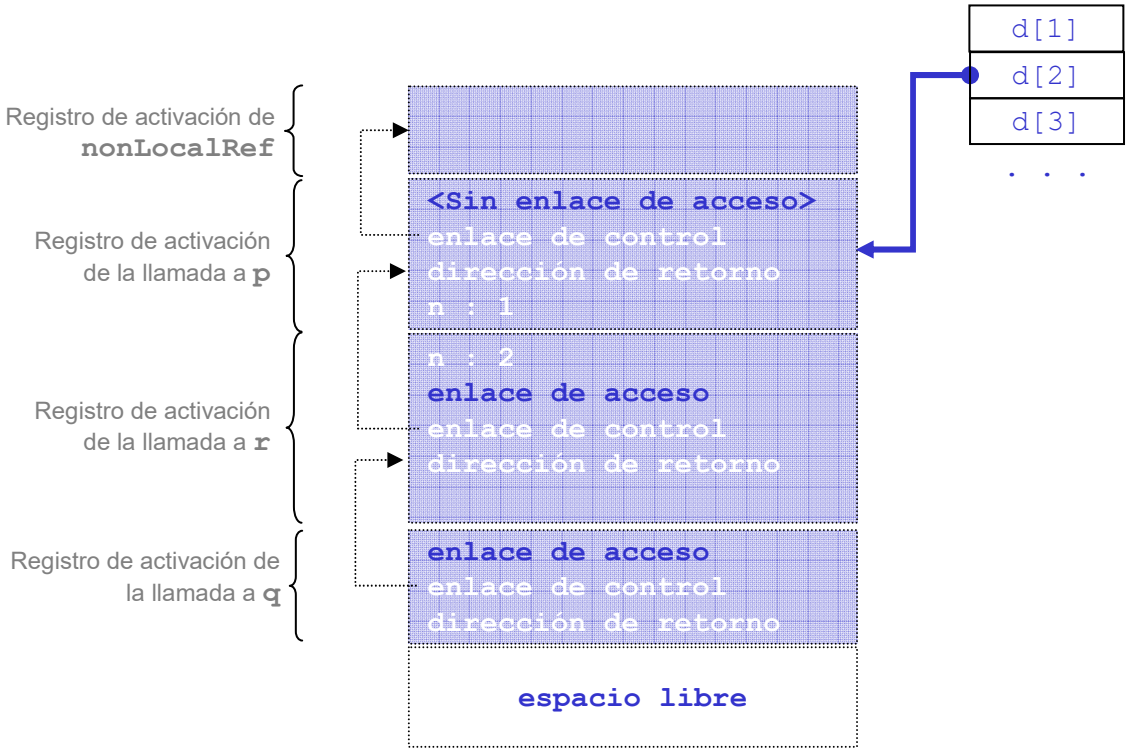
9.4.2 Pila con procedimientos anidados

Estructura de datos tipo *display*

- Es una alternativa más rápida para acceder a los nombres no locales que el ofrecido por los enlaces de acceso.
- Consiste en utilizar una matriz d de apuntadores a registros de activación. Un nombre no local A con profundidad de anidamiento i aparece en el registro de activación apuntado por el elemento $d[i]$ del *display*.
- Una solución sencilla para mantener actualizado el display es utilizar los enlaces de acceso además del display:
  - Como parte de las secuencias de llamadas y de retorno, el display se actualiza siguiendo la cadena de enlaces de acceso.
  - Cuando se sigue el enlace de acceso a un registro de activación a profundidad de anidamiento n, el display $d[n]$ se actualiza para que apunte a dicho registro de activación.
- En realidad, el display duplica la información de la cadena de enlaces de acceso.

9.4.2 Pila con procedimientos anidados

Ejemplo 9.6: Mapa de memoria, usando display, del Ejemplo 9.4.




```
program nonLocalRef;
procedure p ;
var n: integer ;
    procedure q ;
    begin
      (* una referencia a n es
         ahora no local y no global *)
      ...
    end ; (* q *)
    procedure r (n: integer) ;
    begin
       q ;
    end ; (* r *)
begin (* p *)
  n:= 1; r(2) ;
end; (* p *)

begin (* main *)
  p ;
end.
```

9.4.3 Pila con procedimientos como parámetros

Algunos lenguajes permiten el paso de procedimientos como parámetros.

- Cuando se llama a un procedimiento que ha sido pasado como un parámetro, es imposible para un compilador generar código que permita calcular el enlace de aceso en el momento de la llamada.

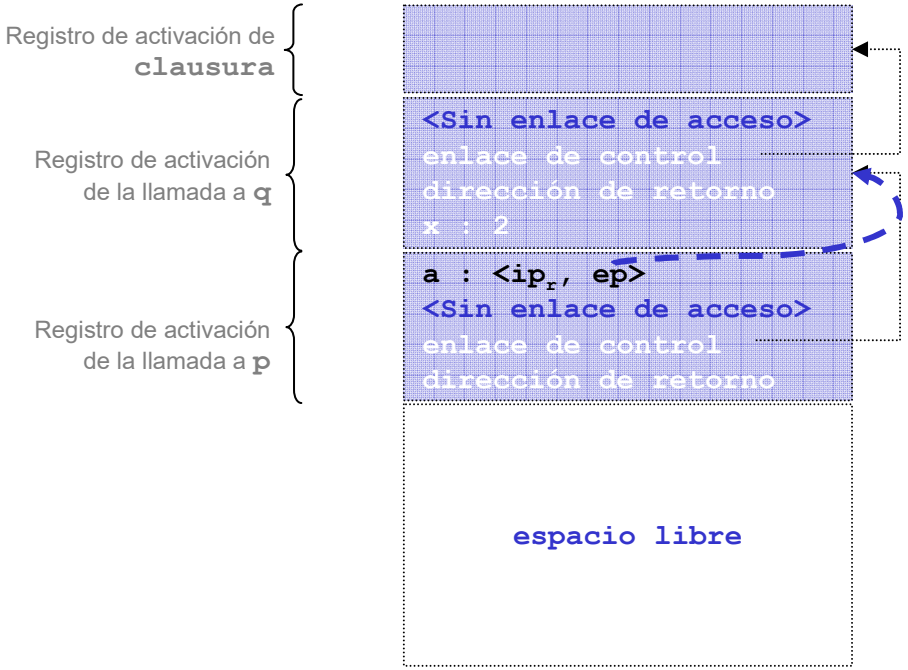
- Un procedimiento como parámetro es, inicialmente, un **apuntador al código** de dicho procedimiento. Denotaremos $\mathbf{ip_a}$ al apuntador de la instrucción del procedimiento "a".

Solución

Incluir, además del **apuntador al código** del procedimiento, un **apuntador de ambiente** (enlace de acceso) para resolver las referencias no locales. Denotaremos **ep** al apuntador de ambiente.

9.4.3 Pila con procedimientos como parámetros

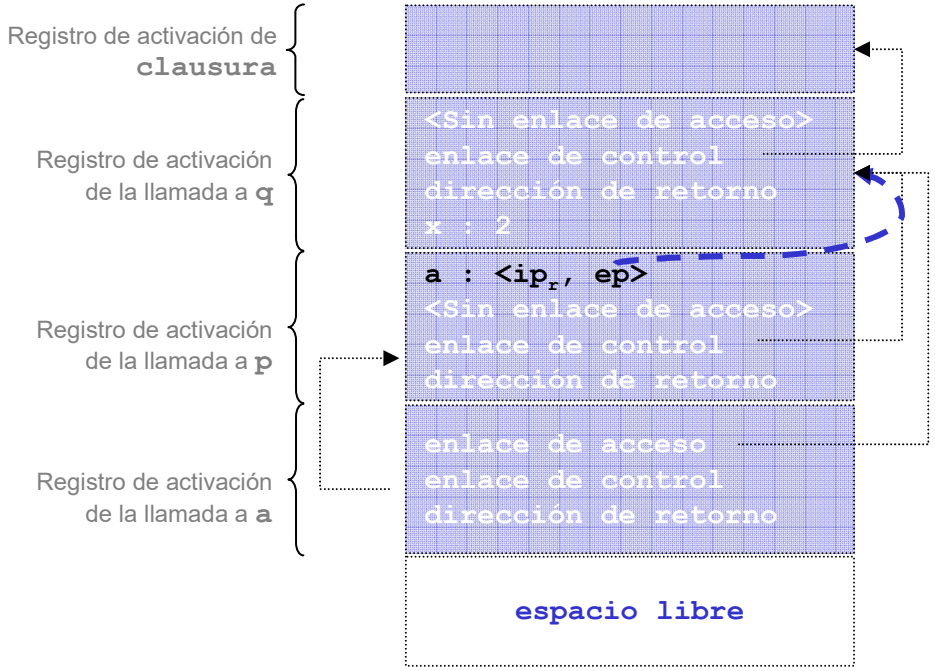
Ejemplo 9.7: Mapa de memoria de un programa en **Pascal** con procedimientos anidados y un procedimiento como parámetro (justo después de la llamada al procedimiento **p**).




```
program clausura;
procedure p (procedure a) ;
begin
   a ;
end ;  (* p *)

procedure q ;
var x : integer ;

   procedure r ;
   begin
     writeln(x) ;
   end ;  (* r *)

begin (* q *)
  x:= 2;
  p(r) ;
end; (* q *)

begin (* main *)
  q ;
end.
```

9.4.3 Pila con procedimientos como parámetros

Ejemplo 9.7 (cont.): Mapa de memoria de un programa en **Pascal** con procedimientos anidados y un procedimiento como parámetro (justo después de la llamada al procedimiento **a**).




```
program clausura;
procedure p (procedure a) ;
begin
   a ;
end ;  (* p *)
procedure q ;
var x : integer ;

   procedure r ;
   begin
     writeln(x) ;
   end ;  (* r *)

begin (* q *)
  x:= 2;
  p(r) ;
end; (* q *)

begin (* main *)
  q ;
end.
```

9.4.3 Pila con procedimientos como parámetros

A la hora de confeccionar un compilador, por simplicidad o uniformidad, es posible tratar los procedimientos ordinarios y los procedimientos dados como parámetros de manera equivalente (obviamente, mediante esta estrategia).

Algunos lenguajes evitan este tipo de complicaciones:

- `C`: No posee procedimientos locales (aunque tiene variables y procedimientos como parámetros).
- `Modula-2`: Tiene una regla especial que restringe los procedimientos como parámetros a procedimientos que sean globales.
- `Ada`: No tiene procedimientos como parámetros.

9.5 Organización dinámica

Una organización basada en pila (stack) es la forma más común de gestionar la memoria en lenguajes como **C**, **Pascal** y **Ada**.

Limitación: Referencia a una **variable local** en un procedimiento y que pueda **ser devuelta**.

Un ambiente basado en pila producirá una referencia colgante al salir del procedimiento al ser eliminado el registro de activación de la pila.

Ejemplo 9.8: Código que devuelve una dirección a una variable local.

```
int * fvarlocal (void)
{
   int x ;
   return &x ;
}
```

Una asignación del tipo **`v = fvarlocal()`**, causa ahora que **`v`** tenga la dirección insegura en la pila de activación (valor que se modificaría arbitrariamente por llamadas posteriores a cualquier procedimiento).

9.5 Organización dinámica

Ejemplo 9.9: Código que devuelve una dirección a un procedimiento local.

```
typedef int (* proc)(void) ;
proc g (int x)
{
   int f (void)   /* funcion local a g */
   {
      return x ;
   }
   return f;
}
int main (void)
{
   proc c ;
   c = g(2) ;
   printf ("%d\n", c());  /* debería imprimir 2 */
   return 0 ;
}
```

Produce una referencia colgante indirecta al parámetro **x** de **g**, al cual se puede tener acceso llamando a **f** después de que **g** ha finalizado su ejecución.

9.5 Organización dinámica

Una organización **completamente dinámica** es necesaria cuando los registros de activación tienen validez hasta que no desaparezcan todas las referencias hacia ellos. Por tanto, serán liberados en momentos arbitrarios durante la ejecución.

Una organización dinámica se caracteriza por:

- Es mucho más complicado que una organización basada en pila por tener que seguir las referencias durante la ejecución.
- Encontrar y liberar áreas inaccesibles de memoria en momentos arbitrarios durante la ejecución (**recolección de basura**).
- La estructura básica de los registros de activación permanece igual.

Conclusión: Necesidad de un administrador de memoria que reemplaza las operaciones sobre una pila con rutinas más generales de **asignación** y **liberación** de espacios de memoria.

9.5.1 Organización dinámica en lenguajes orientados a objetos

Los lenguajes orientados a objetos requieren mecanismos especiales en cuanto a organización de la memoria para incrementar sus características adicionales: objetos, métodos, herencia y ligadura dinámica.

Lenguajes como **Smalltalk** son de organización completamente dinámica (como **Lisp**), mientras que en **C++** concurren las estrategias basadas en pila y dinámica.

Aspectos a considerar a la hora de implementar los objetos:

- Inicialización de los objetos.
- Almacenamiento de la lista de apuntadores de código para métodos disponibles en cada clase (tabla de métodos virtuales) → almacenado en memoria estática.

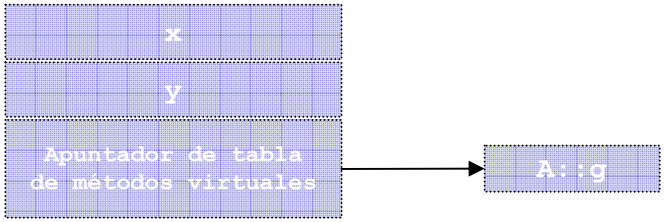
9.5.1 Organización dinámica en lenguajes orientados a objetos

Ejemplo 9.10: Declaraciones de clases en C++.

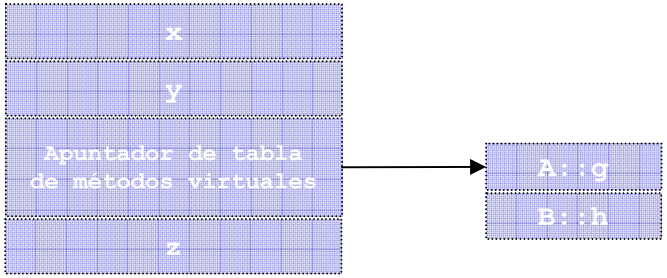
```
class A
{
  public:
     double x, y ;

     void f () ;
     virtual void g () ;
} ;
```

```
class B : public A
{
  public:
     double z ;

     void f () ;
     virtual void h () ;
} ;
```

Un objeto de la clase A aparecería en la memoria (con su tabla de métodos virtuales) de la siguiente forma:



Un objeto de la clase B aparecería así:



9.5.1 Organización dinámica en lenguajes orientados a objetos

Ejemplo 9.10 (cont.): Declaraciones de clases en C++.

```
class A
{
  public:
     double x, y ;

     void f () ;
     virtual void g () ;
} ;
```

```
class B : public A
{
  public:
     double z ;

     void f () ;
     virtual void h () ;
} ;
```

Cabe destacar que la tabla de métodos virtuales aparecerá en una zona contigua y fija de memoria (se puede conocer su desplazamiento antes de la ejecución).

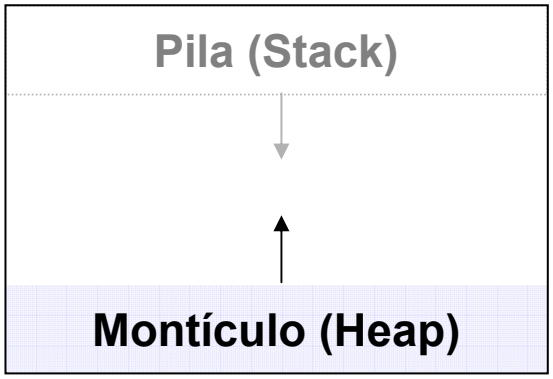
El procedimiento **f** no obedece a una parte dinámica en **C++** al no ser declarado como virtual. No aparece en la tabla de métodos virtuales ya que una llamada a **f** se resuelve en tiempo de compilación.

9.5.2 Gestión del montículo (heap)

Una gestión de la memoria totalmente dinámica requiere de un **montículo** (heap) que es un bloque lineal de memoria cuyas operaciones son:

- **Asignación** (allocate): Toma un parámetro de tamaño (explícito o implícito) y devuelve un apuntador a un bloque de memoria del tamaño correcto o nulo si no existe espacio suficiente.

- **Liberación** (free): Toma un apuntador a un bloque de memoria asignado y lo marca como libre.



9.5.2 Gestión del montículo (heap)

Modos de gestionar el montículo

1. Bloques fijos.
2. Bloques variables.
3. Primero apto (First-fit).
4. Bordes marcados.

Modos de liberar la memoria

- Contador de referencias.
- Recolector de basura (garbage collection).
- Compactación.

9.5.2 Gestión del montículo (heap)

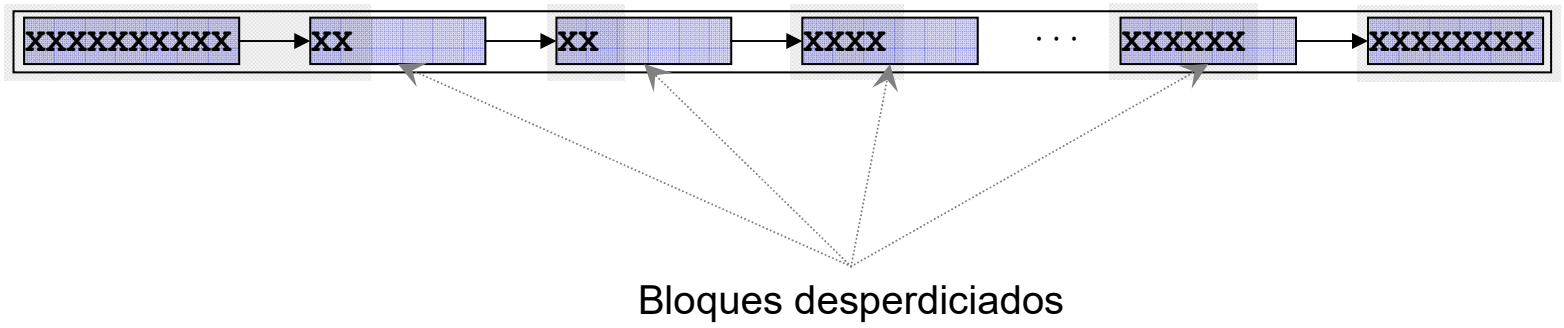
1. Gestión del montículo mediante bloques fijos

Dispone de una lista con los bloques disponibles (todos de tamaño fijo). Sus operaciones son:

- Tomar bloque de lista disponible.
- Liberar bloque, pasando a la cabeza de la pila.

Problema:

- Estructuras mayores que los bloques → desperdicio de memoria.



9.5.2 Gestión del montículo (heap)

2. Gestión del montículo mediante bloques variables

Se parte del Montículo como un único bloque. Cuando se requiere un bloque, se fragmenta en dos bloques (uno con el tamaño requerido y el otro con el resto). El bloque no ocupado pasará a formar parte de una lista de bloques libres.

Problema: Fragmentación del montículo debido a:

- *Causas externas*: Liberación de bloques pequeños.
- *Causas internas*: Al no encontrar bloques del tamaño requerido.

9.5.2 Gestión del montículo (heap)

3. Gestión del montículo mediante el primero apto (First-fit)

Se parte de una lista de bloques disponibles en cuya cabecera aparece el tamaño. Cuando se requiere un bloque, se recorre la lista de bloques libres hasta encontrar el primero que cumpla los requisitos.

Problema: Fragmentación cuando el primero que se encuentra es mucho más grande de lo requerido.
Optimización: Ordenar los bloques según tamaño.

4. Gestión del montículo mediante bordes marcados

Cada bloque dispone de una marca de comienzo y final. Cuando se libera un bloque y existen bloques contiguos libres, se unen en un solo bloque eliminado las marcas.

9.5.2 Gestión del montículo (heap)

Estrategias para liberar la memoria

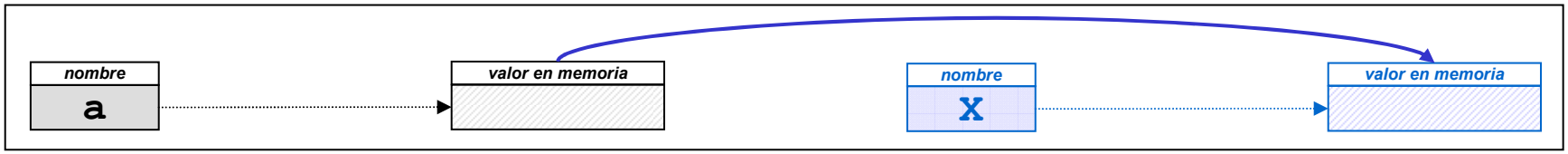
- **Contador de Referencias**: Se añade a cada bloque un contador de referencias. Cuando el contador es 0 (nadie referencia dicha zona de memoria), se libera.

- **Recolector de Basura**: Cuando se agota el montículo se debe invocar a un procedimiento para liberar bloques libres. Las fases son:
  - Cuando se usa un bloque, se marca.
  - Cuando se deja de usar un bloque, se le quita la marca para que el proceso recolector pueda reconocerlo.

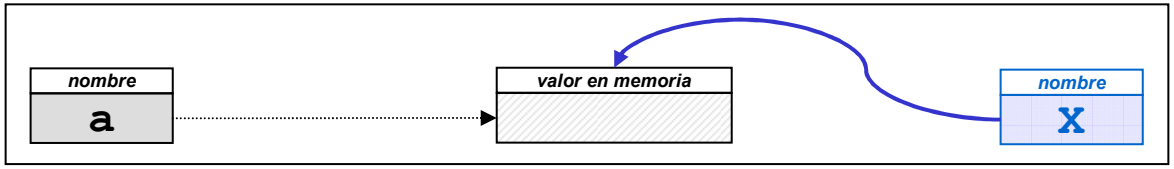
- **Compactación**: Procura ocupar los bloques en el final del montículo, cambiando de situación los bloques para intentar compactarlos.

9.6 Mecanismos de paso de parámetros

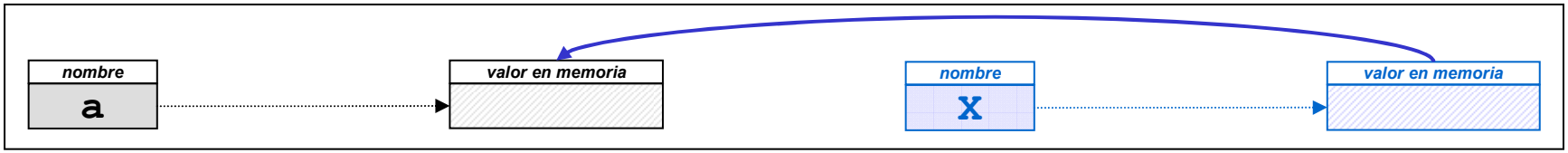
Paso por **Valor:**



Paso por **Referencia:**

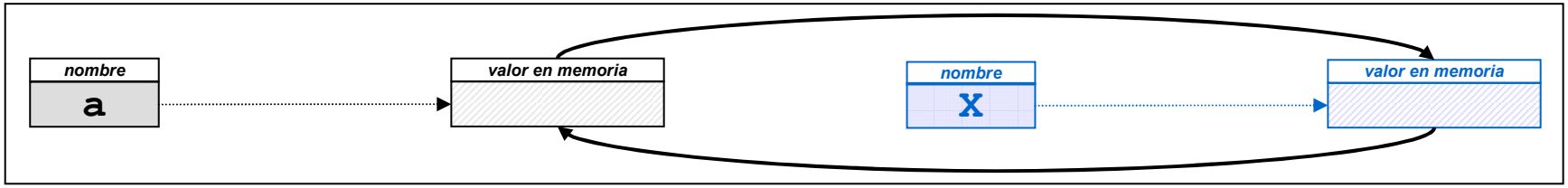


Paso por **Resultado:**

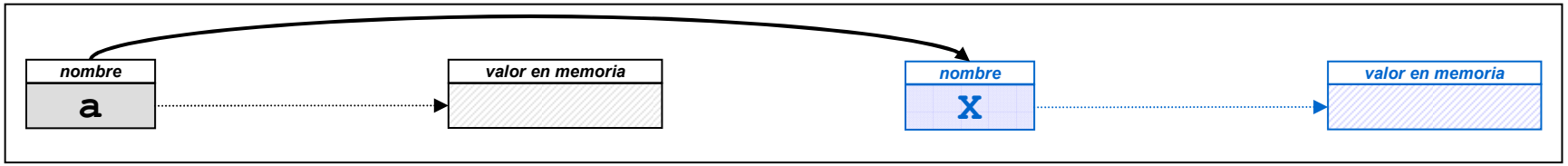


9.6 Mecanismos de paso de parámetros

Paso por **Valor-Resultado:**



Paso por **Nombre:**



9.7 Ejemplo: Asignación de espacio a las variables

Declaraciones dentro de un procedimiento

Distribuir la memoria para los nombres locales al procedimiento mediante la inclusión en la tabla de símbolos con la dirección relativa (desplazamiento desde la parte de datos local en un registro de activación).

El esquema de traducción para las declaraciones dentro de un procedimiento utilizará el siguiente procedimiento:

***introduce* (*nombre*, *tipo*, *desplazamiento*)**
Crea una entrada en la tabla de símbolos para ***nombre***, le da el tipo ***tipo*** y la dirección relativa ***desplazamiento*** en su área de datos.

9.7 Ejemplo: Asignación de espacio a las variables

Gramática con Atributos: Declaraciones dentro de un Procedimiento	
P → D	{ *desplazamiento* = 0 ; }
D → D ; D	
D → **id** : T	{ *indroduce* (**id**.*nombre*, T.*tipo*, *desplazamiento*) ; *desplazamiento*= *desplazamiento* + T.*ancho* ; }
T → **integer**	{ T.*tipo*= *integer* ; T.*ancho*= *sizeof* (*integer*) ; }
T → **real**	{ T.*tipo*= *real* ; T.*ancho*= *sizeof* (*real*) ; }
T → **array** [**num**] **of** T_1	{ T.*tipo*= *array* (**num**.*val*, T_1.*tipo*) ; T.*ancho*= **num**.*val* * T_1.*ancho* ; }
E → ^ T_1	{ T.*tipo*= *pointer* (T_1.*tipo*) ; T.*ancho*= *sizeof* (*pointer*) ; }

9.7 Ejemplo: Asignación de espacio a las variables

Determinación del ámbito de los símbolos

Se crean una serie de operaciones para determinar el ámbito:

creatabla (previa)
Crea una nueva tabla. El apuntador previa apunta a la tabla previa. Inicializa la cabecera de la nueva tabla con la profundidad del anidamiento.

introduce (tabla, nombre, tipo, desplazamiento)
Descrito anteriormente.

añadeancho (tabla, ancho)
Crea una nueva tabla. El apuntador previa apunta a la tabla previa. Inicializa la cabecera de la nueva tabla con la profundidad del anidamiento.

introduceproc (tabla, nombre, tablanueva)
Crea una nueva tabla. El apuntador de la tabla previa apunta a la tabla anterior. Inicializa la cabecera de la nueva tabla con la profundidad del anidamiento.

9.7 Ejemplo: Asignación de espacio a las variables

Determinación del ámbito de los símbolos

Cuando aparece la declaración de un procedimiento se realizan las siguientes operaciones:

- Se introduce el símbolo identificador del nuevo procedimiento en la tabla del procedimiento abarcador.
- Suspender la actividad en la tabla del proceso abarcador.
- Crear una nueva tabla.

9.7 Ejemplo: Asignación de espacio a las variables

Gramática con atributos: Declaración de procedimientos	
P → M D	{ *añadeancho* (*tope*(tblapn), *tope* (*desplazamiento*)) ; *saca* (tblapn) ; *saca* (*desplazamiento*) ; }
M → ε	{ t= *creatabla* (void) ; *mete* (t, tblapn) ; *mete* (0, *desplazamiento*) ; }
D → D_1 ; D_2	
D → **proc id ;** N D **;** S	{ t= *tope* (tblapn) ; *añadeancho* (t, tope (*desplazamiento*) ; *saca* (tblapn) ; *saca* (*desplazamiento*) ; *introduceproc* (*tope* (tblapn), **id**.*nombre*, t)) ; }
D → **id :** T	{ *introduce* (*tope* (tblapn), **id**.*nombre*, T.*tipo*), *tope* (*desplazamiento*)) ; *tope* (*desplazamiento*)= *tope* (*desplazamiento*) + T.*ancho* ; }
N → ε	{ t= *creatabla* (*tope* (tblapn)) ; *mete* (t, tblapn) ; *mete* (0, *desplazamiento*) ; }

9.7 Ejemplo: Asignación de espacio a las variables

Gramática con atributos: Declaración de registros	
T → **record** L D **end**	{ T.*tipo*= record (*tope* (tblapn)) ; T.*ancho*= *tope* (*desplazamiento*) ; *saca* (tblapn) ; *saca* (*desplazamiento*) ; }
L → ε	{ t= *creatabla* (void) ; *mete* (t, tblapn) ; *mete* (0, *desplazamiento*) ; }