

# tema 5: Gestión de memoria en tiempo de ejecución

---

## 5.1 Introducción y asignación estática

## 5.2 Asignación dinámica de memoria utilizando una pila

- » Acceso a variables no locales
- » Paso de parámetros

## 5.3 Asignación dinámica de memoria: "Heap allocation"

- » Asignación y desasignación explícita e implícita de memoria
- » Técnicas para la asignación dinámica de la memoria

# 0.1 Gestion de memoria en tiempo de ejecución

---

Elementos para los que debe asignarse memoria:

- » Programa objeto
- » Estructuras, variables y constantes que aparecen en el programa objeto
- » Variables temporales, buffers de entrada/salida, enlaces,...

# 5.1 Asignación estática

---

Lenguajes para los que es posible determinar en tiempo de compilación el tamaño de todos los objetos a los que hay que asignar memoria

» Ejemplo: FORTRAN

El procedimiento de asignación estática de memoria se basa en la reserva del espacio de memoria asociado a cada identificador o constante del programa

# 1 Asignación dinámica de memoria

---

Características del lenguaje que determinan la necesidad de gestionar la memoria de manera dinámica:

1. Soporte de la recursividad (C, Ada, Pascal, etc)
  - Estructura de bloques anidados
2. Presencia en el lenguaje de constructores que asignan y desasignan memoria (new, alloc, setf, etc)
3. Estructuras de Datos de tamaño variable

## 5.2 Asignación de memoria por medio de una pila

---

Parte de la memoria gestionada por el programa es gestionada como una pila

Se agrupa todo el espacio de memoria asociado a la ejecución de un subprograma en un bloque: el **registro de activación**

- los parámetros
- los valores a devolver (funciones)
- las variables locales
- variables temporales
- dirección de retorno
- enlaces a otros reg. de activación

# Llamadas a un procedimiento

---

Acciones a realizar en el momento de la llamada:

- › asignación de espacio para el registro de activación del procedimiento llamado
- › evaluación de los parámetros actuales de la llamada y colocación en el lugar correspondiente del reg. de activación
- › establecer los enlaces que permitan el acceso a objetos globales al procedimiento
- › salvar el "estado" del subprograma llamador
- › almacenar la dirección de retorno, a la que habrá que devolver el control

# Llamadas a un procedimiento (II)

---

Final de la ejecución:

- › Si el subprograma es una función, dejar disponible el valor devuelto
- › Restaurar la situación del reg. de act. del procedimiento llamador
- › Ceder el control a la instrucción siguiente a aquella que realizó la llamada (a la dir. de retorno)

Las acciones anteriores las realiza el código generado por el compilador. Ese código puede formar parte de la traducción del subprograma o de la traducción de la llamada

# Modos de acceso

---

En un lenguaje que permite la asignación estática de memoria todos los accesos se realizan a la zona de memoria gestionada estáticamente



# Modos de acceso (II)

---

En un lenguaje que permite la recursividad, pero no la emulación de bloques (C), desde el código de un procedimiento sólo se puede acceder a:

- » Variables Globales
- » Parámetros o variables locales al procedimiento

En tiempo de ejecución esto conllevará accesos (directos) a:

- » La zona de memoria gestionada estáticamente
- » El registro de activación en el tope de la pila

# Modos de acceso (III)

---

En un lenguaje que permite la recursividad y la empujación de bloques (Pascal, Ada) desde el código de un procedimiento se puede acceder a:

- » Variables Globales
- » Parámetros o variables locales al procedimiento
- » Parámetros o variables de los procedimientos que engloban al procedimiento en cuestión

# Modos de acceso (IV)

---

En tiempo de ejecución esto conllevará acceso (directos) a:

- La zona de memoria gestionada estáticamente
- El registro de activación en el tope de la pila

E indirectos a:

- La zona donde reside el registro de activación correspondiente a la invocación más reciente de procedimiento donde está definida la variable (parámetro)

# A recordar

---

Si un procedimiento está ejecutándose de manera recursiva, en la pila habrá tantos registros de activación como invocaciones se estén procesando. En un momento dado sólo una de las invocaciones tiene el control, el resto está pendiente de que le llegue su ejecución.

Para que un procedimiento definido dentro de otro esté ejecutándose en un momento dado es imprescindible que previamente haya sido invocada la función o el procedimiento que lo engloba.

# A recordar

---

La gestión de enlaces estáticos es una necesidad asociada a la posibilidad de la imbricación de bloques en el L.A.N.

Los accesos a variables definidas en un procedimiento distinto a aquél en que se utilizan conllevan (en tiempo de ejecución) el recorrido de la cadena de enlaces estáticos.

El compilador tiene la información suficiente para generar el código: la información de alcance en la Tabla de Símbolos

# Ejemplo

La misma instrucción se traduce en código distinto en función de su aparición en un procedimiento o en otro

```
procedure P1...  
  var a...  
    Procedure P11...  
      Procedure P111...  
        begin  
          ...  
          a:=  
          ...  
        end  
      begin  
        ...  
        a:=  
        ...  
      end  
    begin  
      ...  
      a:=  
      ...
```

# Display

---

El coste asociado al acceso a la posición de memoria correspondiente a una variable es función de la diferencia de niveles entre el lugar de su declaración y el de su uso

Para paliar el problema anterior se utilizan como **alternativa** los DISPLAYS. Un DISPLAY es una tabla virtual que en un momento dado de la ejecución del programa mantiene los apuntadores a todos los registros de activación accesibles en ese momento

# Display

---

La gestión del DISPLAY la hace el código generado por el compilador, de forma que el DISPLAY:

- › Aumenta de tamaño cuando se llama a un procedimiento "más profundo"
- › Mantiene el tamaño cuando se llama a un procedimiento del mismo nivel
- › Disminuye de tamaño si se llama a un procedimiento de nivel "más global"



# Ejemplo (I)

---

```
program ejemplo;
```

```
...
```

```
  procedure P1;
```

```
  ...
```

```
  procedure P3;
```

```
  ...
```

```
    procedure P31;
```

```
    ...
```

```
      procedure P311;
```

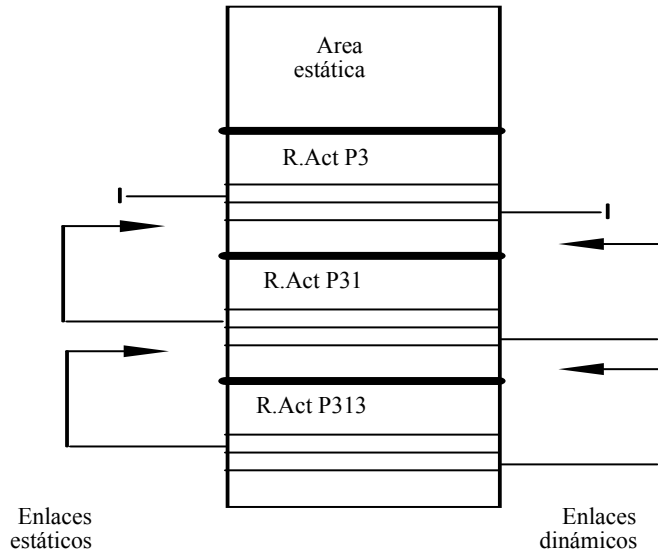
```
      ...
```

```
      procedure P313;
```

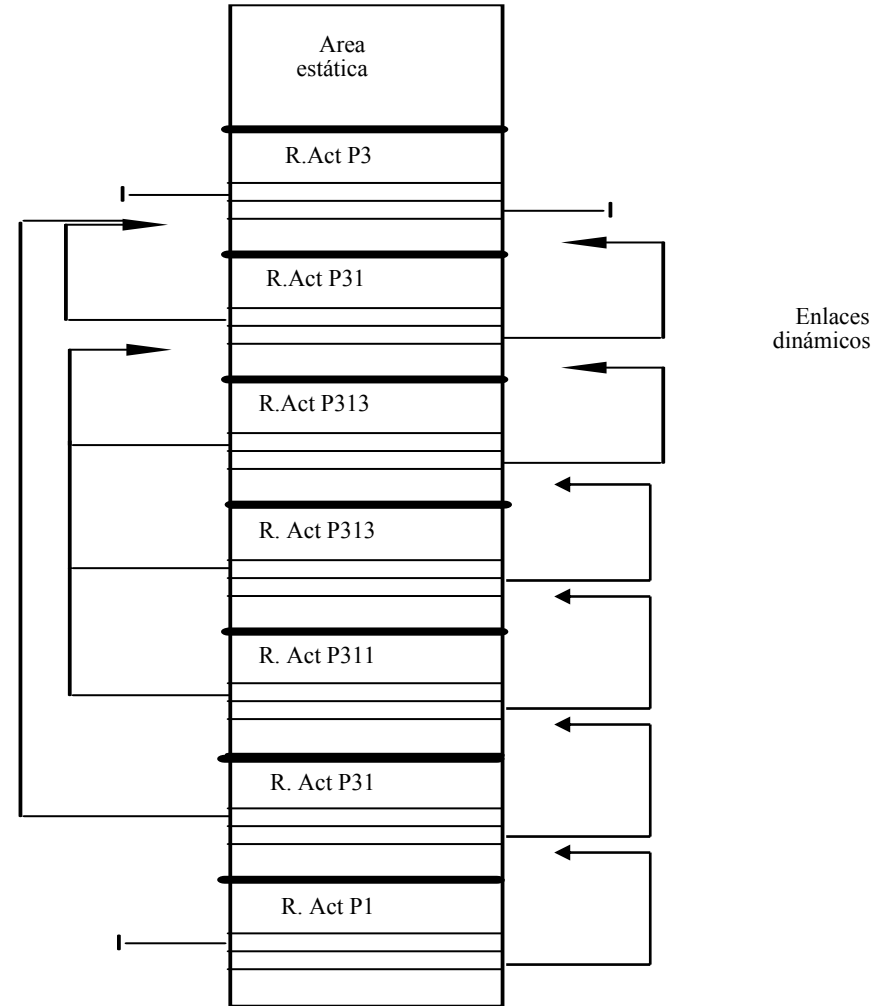
```
      ...
```

» ejemplo -> P3 -> P31 -> P313 -> P311 -> P311 -> P31 -> P1

# Ejemplo (II)



**Figura a**



**Figura b**

# Ejemplo (III)

---

```
program ejemplo;
```

```
...
```

```
  procedure P1;
```

```
  ...
```

```
  procedure P3;
```

```
  ...
```

```
    procedure P31;
```

```
    ...
```

```
      procedure P311;
```

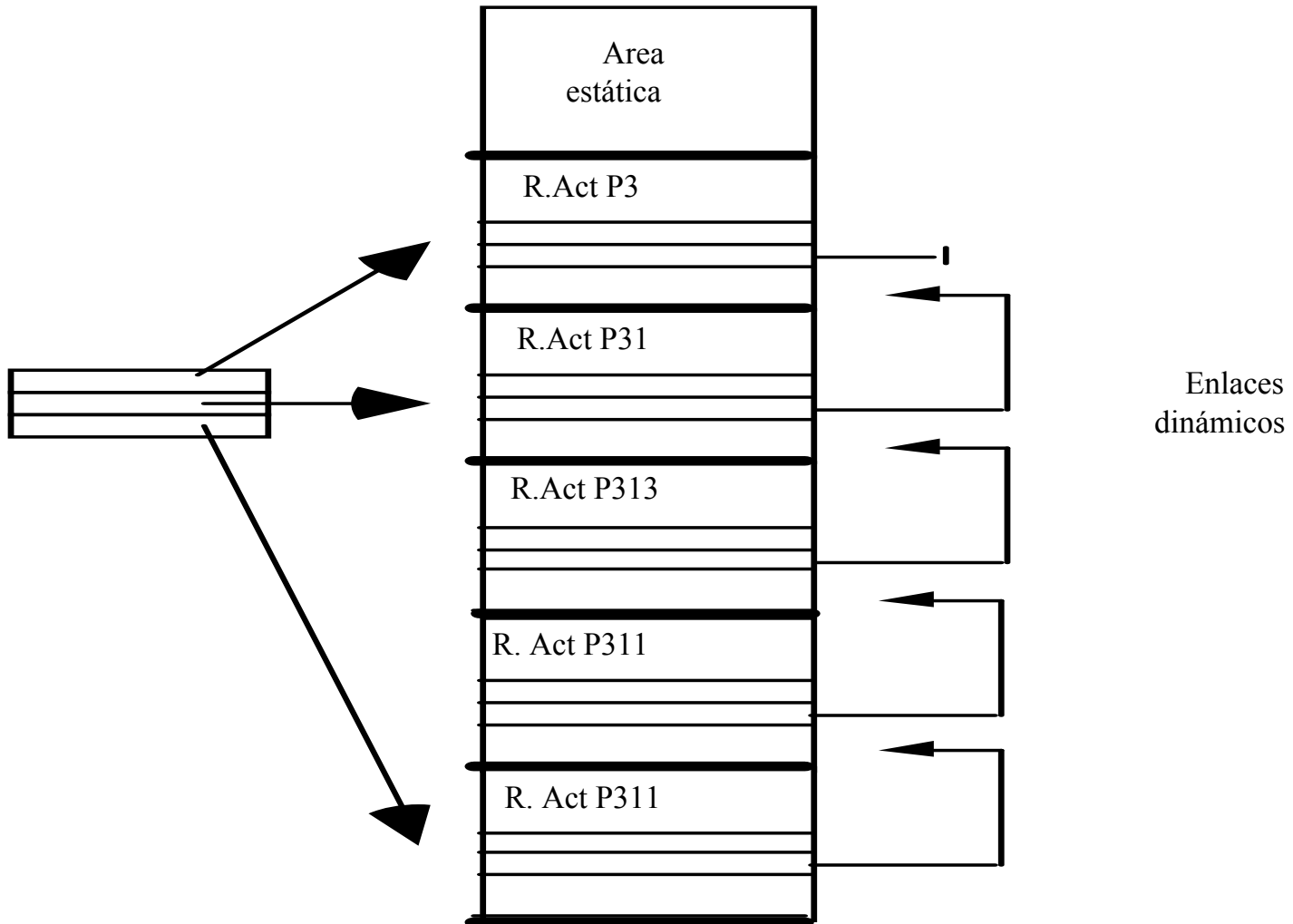
```
      ...
```

```
      procedure P313;
```

```
      ...
```

» ejemplo -> P3 -> P31 -> P313 -> P311 -> P311

# Ejemplo (IV)



# Paso de parámetros

---

- Valor (*copia*)
- Referencia
- Copia y restauración

# Ejemplo

---

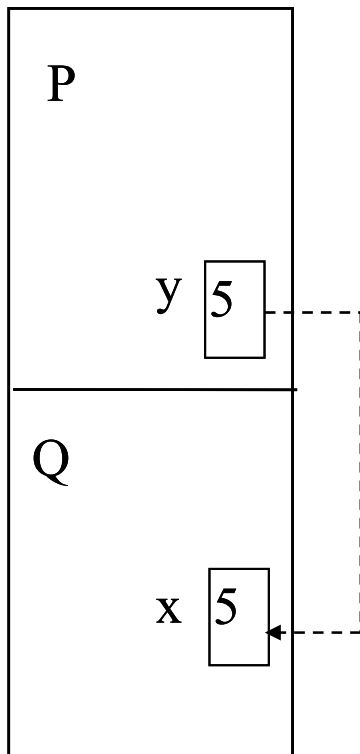
```
procedure P
y: Integer;
  procedure Q(x: integer);
  begin ...
    write(y);
    x := 10;
    write(x);
  end; (* Q *)
begin
  ...
  y := 5;
  Q(y);
  write(y);
end; (* P *)
```

# Valor (copia)

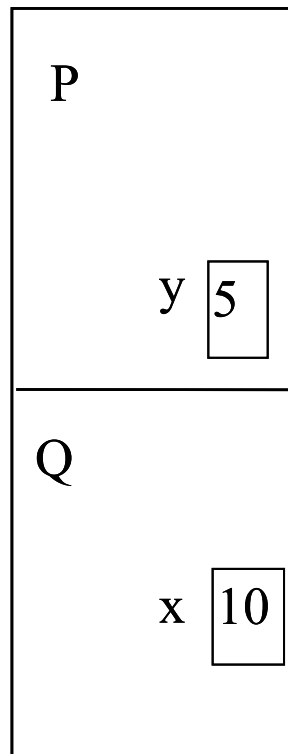
subprograma llamado usa una copia del parámetro al.

**la de registros de activación:**

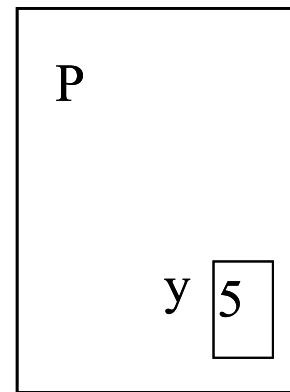
llamada a Q



al final de Q



después de acabar Q

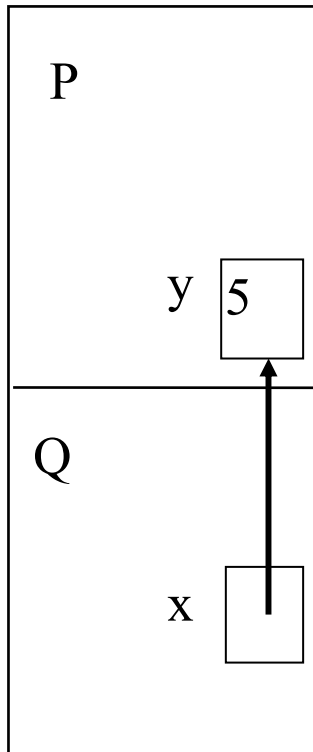


# Referencia

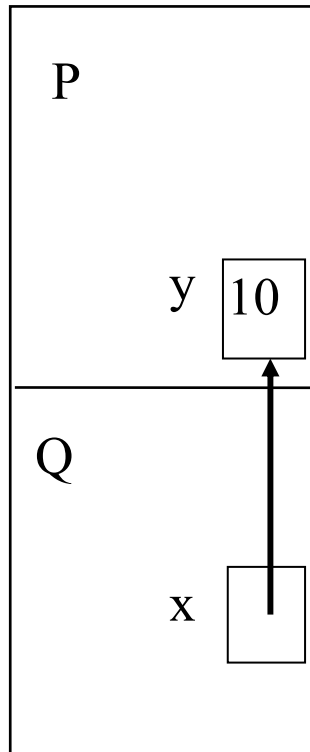
subprograma llamado usa la dirección de memoria del parámetro.

**la de registros de activación:**

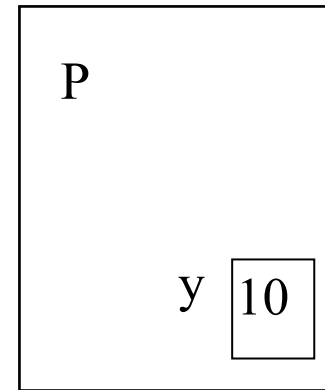
llamada a Q



al final de Q



después de acabar Q





# Copia-Restauración

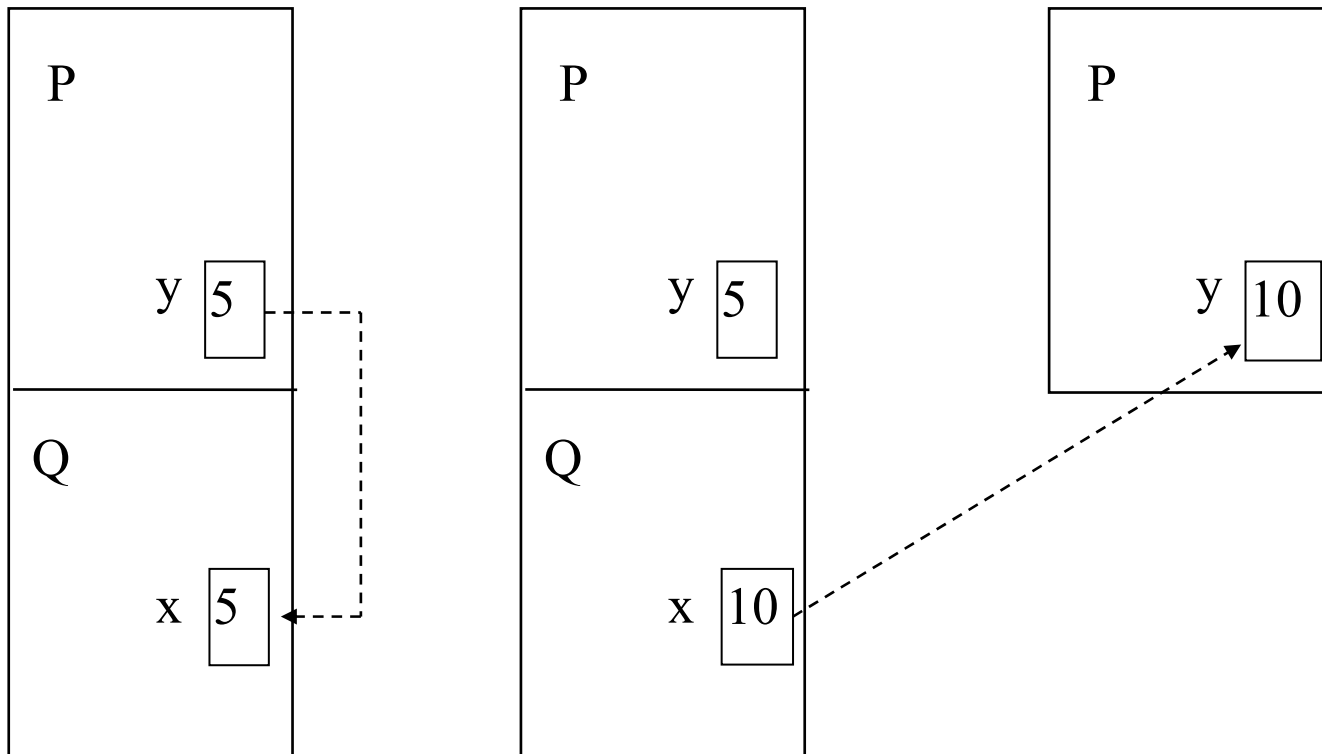
subprograma llamado usa una copia del parámetro real y al final de la ejecución del procedimiento llamado, se copia el resultado en el parámetro real.

**la de registros de activación:**

llamada a Q

al final de Q

después de acabar Q



## 5.3 Heap allocation

---

**Heap:** "montón" de posiciones de memoria. Es un área de memoria de donde se pueden coger posiciones de memoria según se necesiten.

Se usa cuando la asignación por medio de una pila no sirve.

Ejemplo: punteros. La memoria se coge y se libera en cualquier orden. Operaciones:

*new(p)*: coge un área de memoria para guardar un objeto de tipo *p*

*dispose(p)*: libera el área de memoria ocupada por *p*

## Organización de la memoria en tiempo de ejecución:

## Organización de la memoria en tiempo de ejecución:

# Heap allocation (III)

---

Una posible implementación del área "heap": una lista de posiciones libres (y otra de posiciones ocupadas).

- new(p): asignar a "p" la primera posición de la lista de libres. Eliminar esa posición de la lista de libres.
- dispose(p): añadir la memoria ocupada por "p" a la lista de libres.

# Heap allocation (IV)

---

Quando los bloques pueden ser de diferente longitud (según el tamaño de "p") hay diferentes opciones:

- Coger el primer conjunto de posiciones libres donde quepa "p". Problema: fragmentación de la memoria.
- Coger un conjunto de posiciones donde quepa más exactamente.
- Tener varias listas de posiciones libres de diferente tamaño.

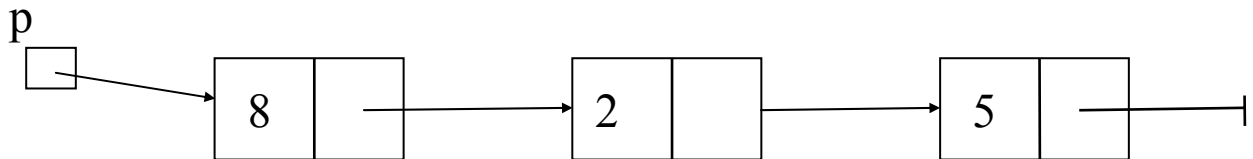
# Heap allocation (V)

## Desasignación implícita.

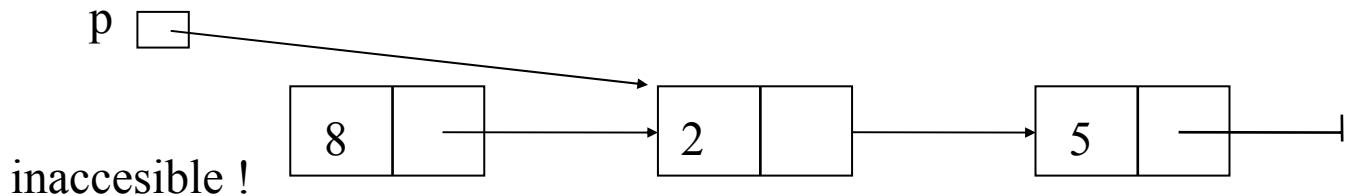
En muchos lenguajes no existe instrucción de liberación de memoria (dispose), como en Lisp o Ada.

Se debe ejecutar de vez en cuando la recogida de basura (garbage collection): recuperar las posiciones de memoria que son inalcanzables pero que se encuentran como ocupadas.

Por ejemplo:



Después de hacer "p := p.next" :



# Heap allocation (VI)

---

Ventajas de la desasignación implícita:

- Mayor abstracción, el sistema se ocupa de ello, y no es tarea del programador.
- Mayor seguridad: no hay posibilidad de hacer "dispose" por error de una posición que todavía se quiere usar.

Desventaja:

- El sistema ejecuta el programa de "garbage collection" en mitad de la ejecución de un programa.