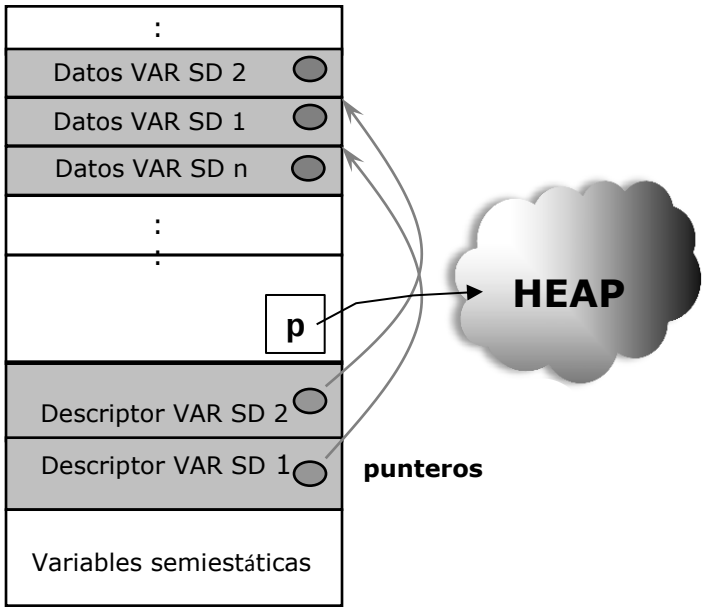
# Lenguajes y Compiladores

## Administración del Heap

# Heap o Memoria Dinámica



El heap o montículo es un bloque de almacenamiento dentro del cual se asignan o liberan segmentos de alguna manera no estructurada.



# Heap o Memoria Dinámica

- Se ubica en una región de la memoria cuya dirección se encuentra alejada de la pila de registros de activación cambiando su tamaño en cualquier momento.

- La administración del heap es responsabilidad del lenguaje.

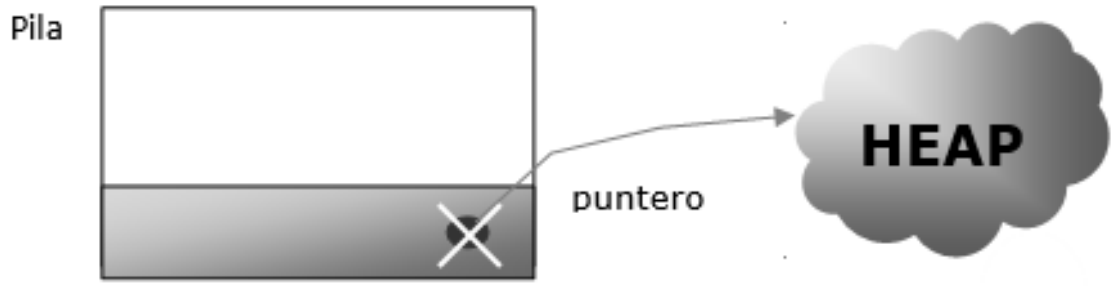
- La necesidad de almacenamiento en el heap surge cuando un lenguaje de programación permite asignar y liberar almacenamiento en puntos arbitrarios durante la ejecución del programa (tiempo de ejecución).

# Problemas con Punteros



## Caso 1 : Pila a Heap

# Problemas con Punteros

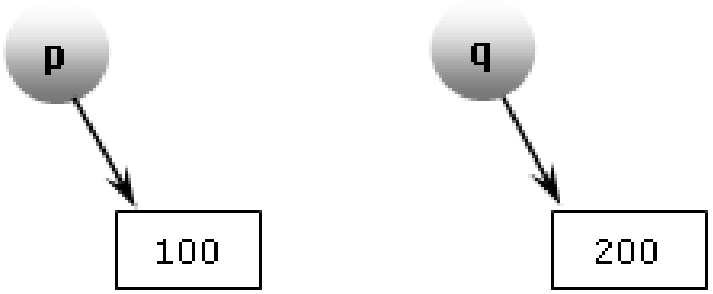



- Si por algún motivo se produce un problema con el puntero, esto implica la presencia de basura (garbage).
- Es posible que se “pierda” la referencia del puntero, esto ocasionará que exista un segmento o bloque de datos en el heap que resulte inaccesible.

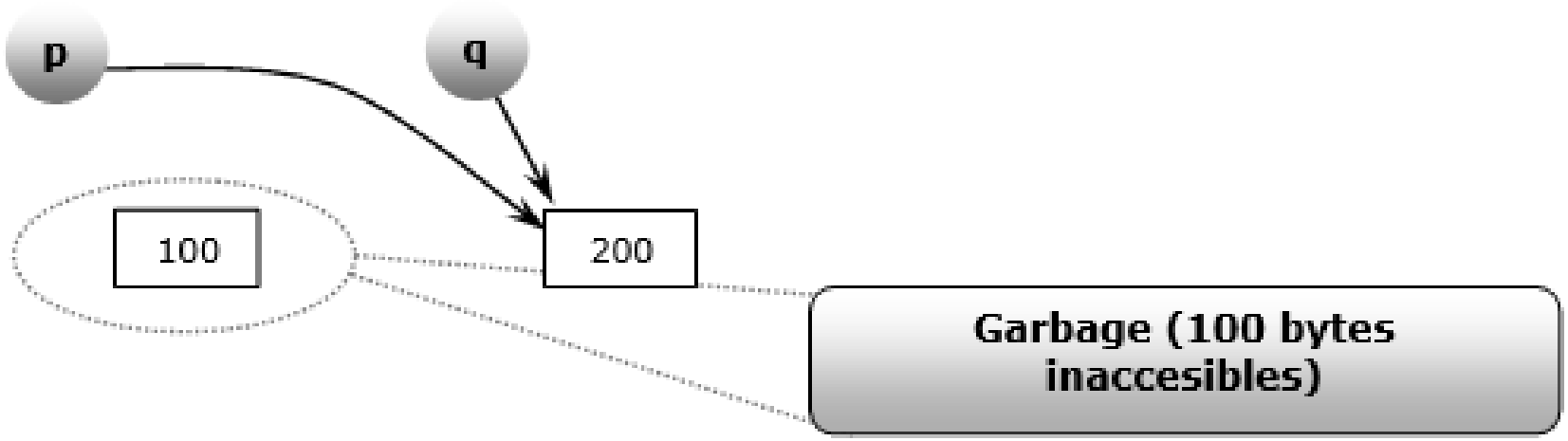
**GARBAGE ➡ USADO + INACCESIBLE**

# Problemas con Punteros

```
int *p,*q
p=malloc (100);
q=malloc (200) ;
```




```
p = q ;
```

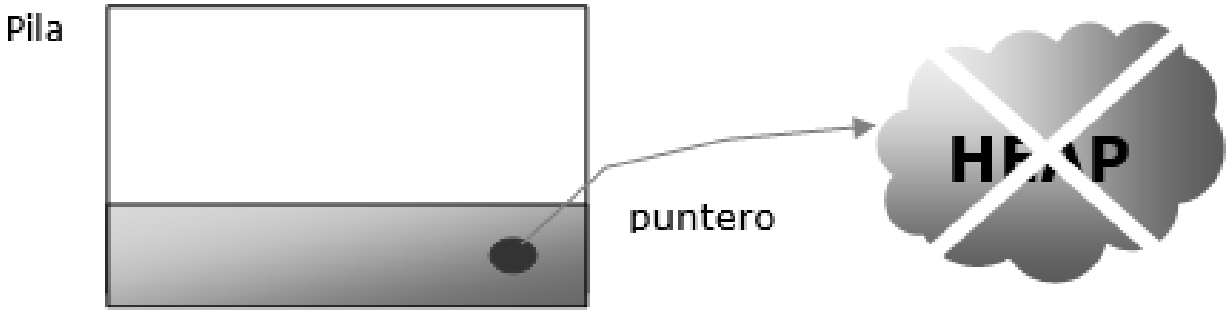


# Problemas con Punteros

```
int *p,*q
p = malloc (100);
q = malloc (200);
If a > 0 then return;
```

**En este caso se genera garbage porque las variables p y q desaparecen luego del *return*, quedando bloques en el heap que no son accesibles**

# Problemas con Punteros

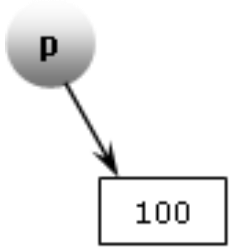



- Si por algún motivo se produce un problema con lo apuntado, esto implica la ***presencia de punteros colgados (dangling pointer o dangling reference)***.

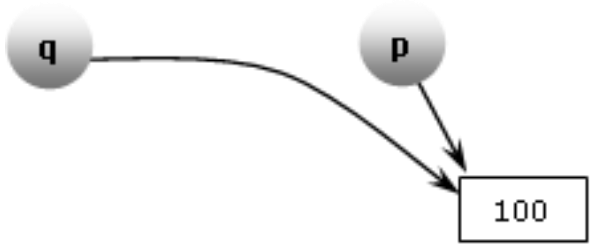
- Se puede “perder” el bloque de datos existente en el heap por lo que existirá un puntero cuya referencia sea inválida, o sea que se apunte a “algo” que no existe más

# Problemas con Punteros

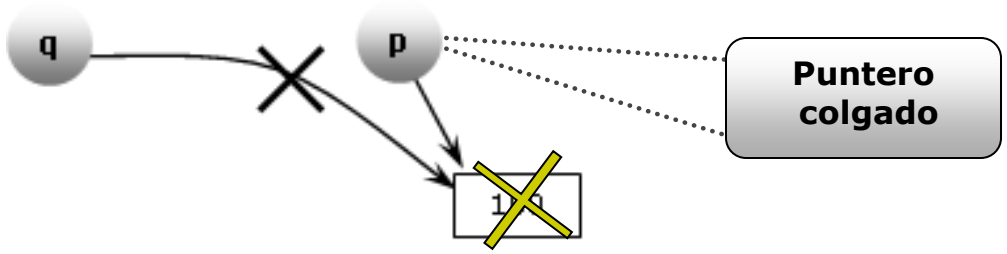
int *p,*q
p=malloc(100)




q = p




free (q)



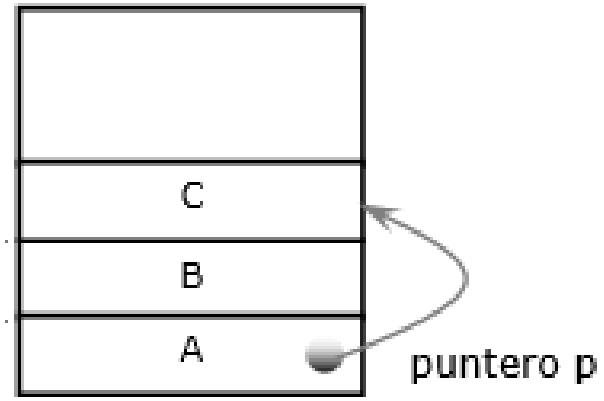
# Problemas con Punteros



## Caso 2: Pila a Pila

# Problemas con Punteros

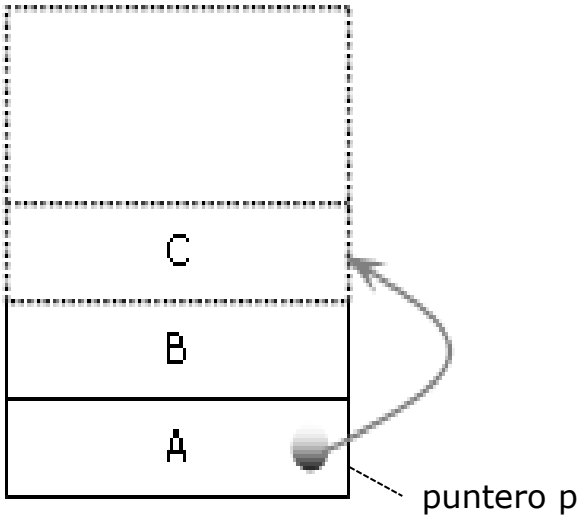
- Todas las variables de tipo puntero hacen referencia a una dirección dentro de la misma pila.




- Si existiese un puntero que apunta a un elemento en una dirección más nueva, cuando finaliza el *registro de activación* que contiene a este elemento, se produce un *puntero colgado.*

# Problemas con Punteros






**La variable tipo puntero *p* se encuentra apuntando a una dirección más alta (en el bloque C), en el momento en que C termina, esta dirección no existe más y el puntero *p* queda colgado.**

# Problemas con Punteros

## Algol

```
A
   begin
     int a
     ref int p
          B
              begin
                 int b
                 p=a


                 p=b


              end
   end
```

**Se copia la dirección de la variable a en p. Esto no genera inconveniente ya que ambas están en el mismo ámbito, es decir están declaradas en el bloque más externo (bloque A).**

**El puntero *p* vive más tiempo que la variable apuntada. Esto generaría un puntero colgado al finalizar el bloque B. En Algol, esta situación genera un *error en tiempo de compilación.***

# Problemas con Punteros

## Algol

```
A
   begin
    int a
    ref int p
        B
           begin
             int b
             p=a

             p=b

           end
    end
```

**En el lenguaje Algol existe lo que se denomina "Regla de Alcance" que indica que «en cada copia de referencias, el alcance de lo apuntado debe ser mayor o igual al alcance del apuntador»**

**En la instrucción 2 del ejemplo, el puntero p tiene mayor alcance que la variable b (objeto apuntado) y es por eso que se produce el error en compilación, evitando así la generación de punteros colgados en la pila.**

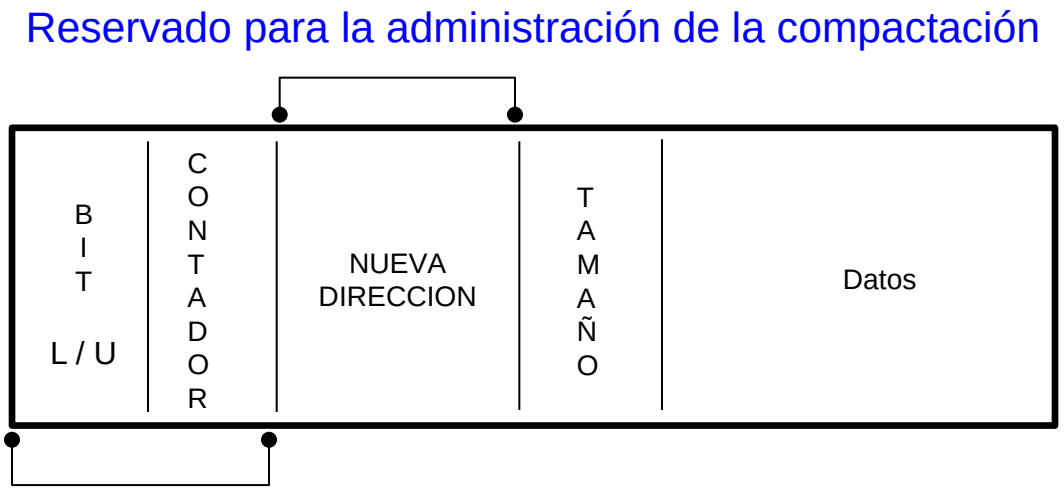
# Problemas con Punteros

El manejo de punteros por parte del programador produce dos tipos de conflictos muy serios y difíciles de encontrar: los **punteros colgados** (en la pila o en el heap) y **garbage** (sólo en el heap).

# Gestión del Heap

## Bloques genéricos en el Heap

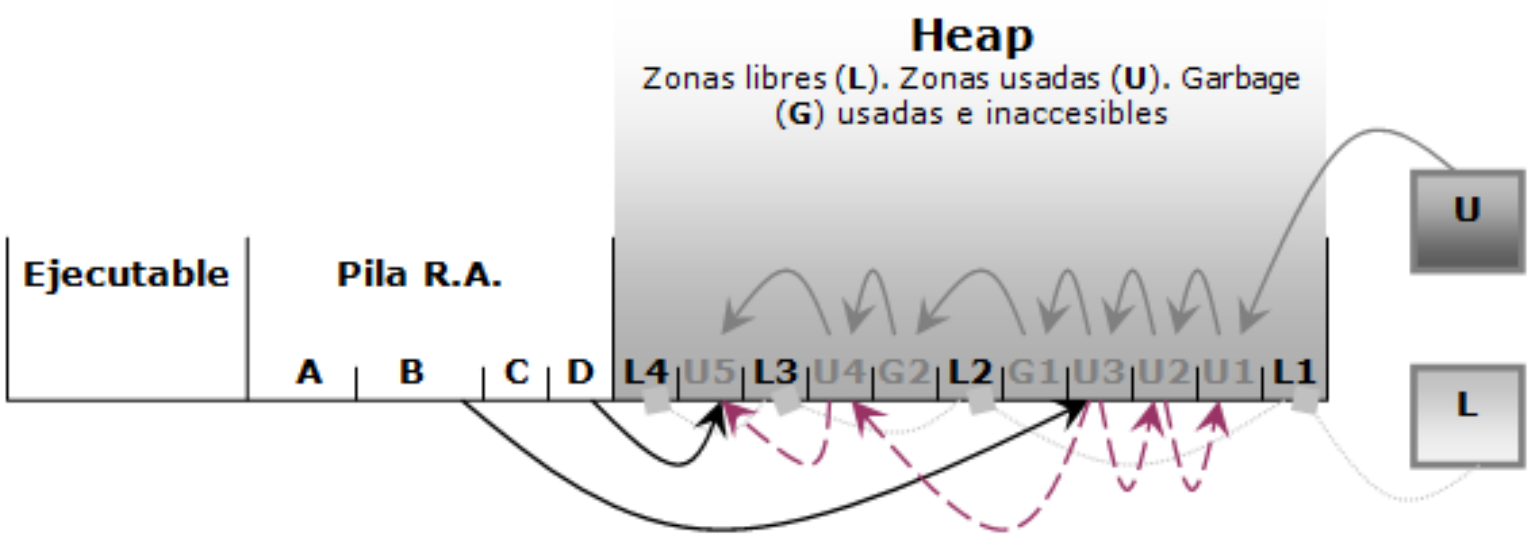
Un bloque o variable dinámica en el heap presenta en general el siguiente formato




El lenguaje mantiene una lista de bloques usados y otra de bloques libres apuntados por punteros fuera del heap llamados « cabecera de usados» y «cabecera de libres» respectivamente

# Gestión del Heap

## Bloques genéricos en el Heap




¿Qué sucede cuando el programador crea una variable dinámica anónima (con malloc, new, allocate, etc.)?

- El lenguaje busca entre los bloques libres siguiendo la lista L (libres), el bloque más conveniente
- El lenguaje adiciona el nuevo bloque a la lista U (usados)

# Gestión del Heap

¿Cómo asigna el espacio?

**BEST FIT**
**WORST FIT**
**FIRST FIT**

# Gestión del Heap

## Bloques genéricos en el Heap

**¿Y si no hay un bloque libre pero hay espacio?**

**Necesita correr algún recolector de basura y/o compactar**

# Gestión del Heap

## Recolección de basura (garbage collection)

### Lenguajes estáticos

- No existe heap.

### Lenguajes dinámicos

- La única memoria para almacenar datos es el heap.
- Todas las variables se encuentran allí (no poseen *pila de registros de activación).*
- No hay punteros visibles por el programador $\Rightarrow$ se generan bloques inaccesibles en el heap, el lenguaje se encarga de crearlos y borrarlos. **No se genera garbage.**
- La memoria se fragmenta y las zonas libres se encuentran “sueltas”, si es necesaria más memoria deben juntarse los espacios libres (compactación).

### Lenguajes tipo Algol

- Pueden existir punteros al heap con lo cual algunos bloques del mismo pueden resultar ***garbage***.

# Gestión del Heap

## Recolección de basura (garbage collection)

**Garbage collection en los lenguajes tipo Algol (C, Pascal, Ada, etc.) ⇒ eliminación de garbage + compactación.**

**Garbage collection en los lenguajes dinámicos ⇒ compactación.**

# Gestión del Heap



## Garbage Collection

# Gestión del Heap

## Garbage collection

Sólo sucede en los lenguajes tipo Algol u orientados a la pila, los que además pueden presentan fragmentación de la memoria.

Mecanismos

- **Eliminadores de basura (garbage collectors)**
- **Eliminadores de basura y compactadores**

Algoritmos

- **Marcado y barrido**
- **Conteo de referencias**
- **Copia en dos direcciones**
- **Marcado y barrido local**

# Gestión del Heap

## Marcado y barrido

- Esta técnica solamente funciona como **recolector de basura.**

- Los lenguajes que utilizan este mecanismo deben **compactar la memoria** luego de correr este algoritmo

# Gestión del Heap

## Marcado y barrido

### Funcionamiento

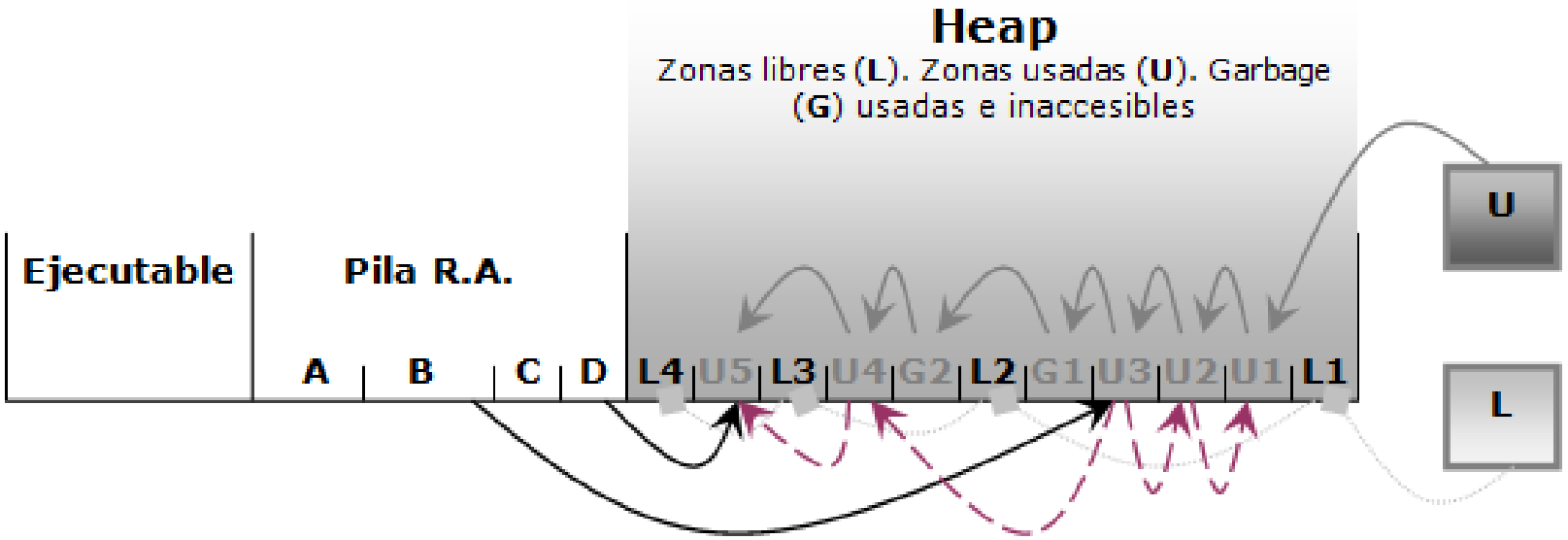
- el lenguaje mantiene un puntero fuera del heap a una lista de bloques libres, y otro a una lista de bloques usados
- el programador solicita un bloque de memoria en el heap
- el lenguaje trata de ubicarlo siguiendo la lista de espacios libres
- Si no es posible ubicar el bloque requerido, entonces se ejecutará el algoritmo de eliminación de basura (garbage collector).

**Filosofía**

**“no recuperar hasta que no se encuentre mas lugar”**

# Gestión del Heap

## Marcado y barrido (ALGORITMO)




- Se marcan todos los bloques usados en el heap, bit L/U. (LU)
- Se recorre la pila. Se referencia cada uno de los punteros existentes en la pila, quitando la marca al bloque apuntado en el heap.

  Si el bloque apuntado posee a su vez una referencia a otro bloque, también se borra la marca a este ultimo referenciado.
- Se recorre el heap y aquellos bloques que quedan marcados, representan **garbage** por lo que se enlazan a la lista de espacios libres.

# Gestión del Heap

## Conteo de referencias

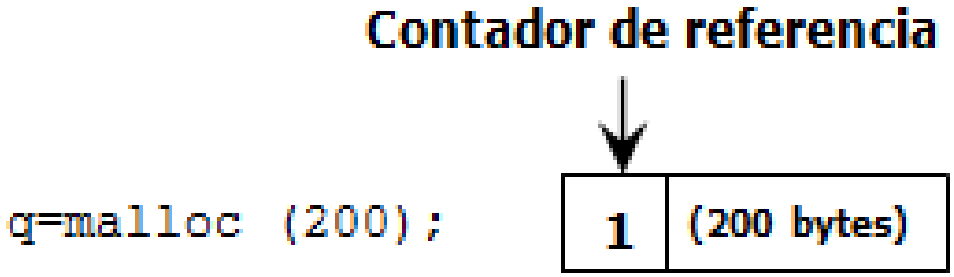
- Los lenguajes que implementan esta estrategia no requieren ejecutar un algoritmo de recolección, ya que no permite generar basura, pero no elimina la fragmentación.

- El lenguaje debe mantener un control respecto de las variables que apuntan a los bloques del heap, por lo que en cada uno de los bloques se reserva un espacio para un contador de referencias que el lenguaje debe administrar lo que implica un **mayor costo.**

- El lenguaje libera el espacio cuando el contador se vuelve 0.

# Gestión del Heap

## Conteo de referencias




Con la función *malloc (200)* se crea una variable anónima dinámica en el heap generando un bloque de memoria de 200 bytes que es referenciado por el puntero q, entonces su contador de referencias se vuelve automáticamente 1 (una referencia a este bloque).


Con p=q, al puntero p se le copia la dirección de q por lo tanto el contador de referencias del bloque apuntado por q pasa a 2 ya que son dos los punteros que hacen referencia a dicho espacio.

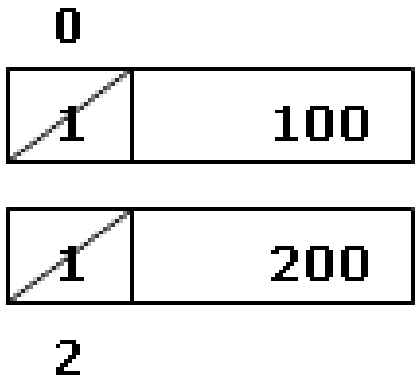
# Gestión del Heap

## Conteo de referencias

```
p=malloc (100);
```


```
q=malloc (200);
```


```
p=q;
```

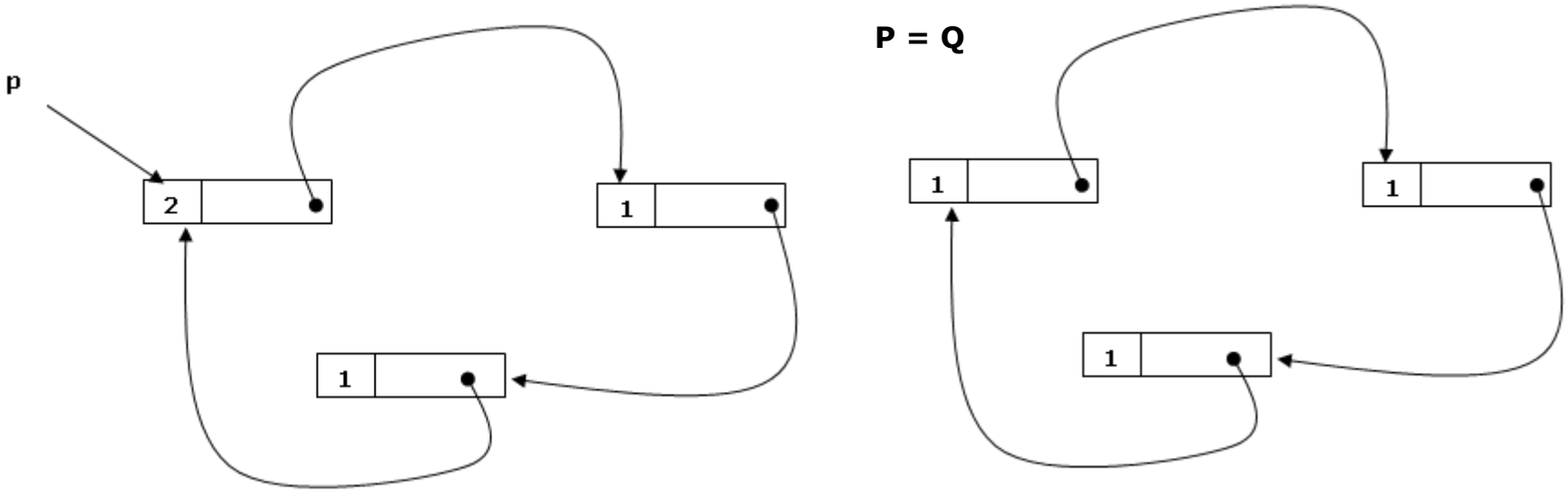



Cuando ***p*** deja de apuntar a su espacio de memoria y apunta al de ***q***, entonces se desreferencia y se decrementa el contador de referencias (que pasa a ser 0). Y al espacio de memoria referenciado por ***q*** llegan 2 punteros (***p*** y ***q***), su contador de referencia es 2.

# Gestión del Heap

## Conteo de referencias

**Este algoritmo podría no funciona con estructuras cíclicas**




**Si por alguna razón p es asignado a otra variable entonces los tres bloques pasarían a ser basura.**

# Gestión del Heap

## Conteo de referencias

**En este algoritmo se debe tener en cuenta el costo del contador de referencias, y de la operación a realizar para cada asignación**

**Para P = Q**

a. Obtener acceso al elemento al que apunta P y decrementar 1 (-1)
b. Si es cero, devolver el elemento a la lista de espacios libres.
c. Copiar en P el valor de Q.
d. Obtener acceso al elemento al que punta Q e incrementar 1 (+1)

# Gestión del Heap

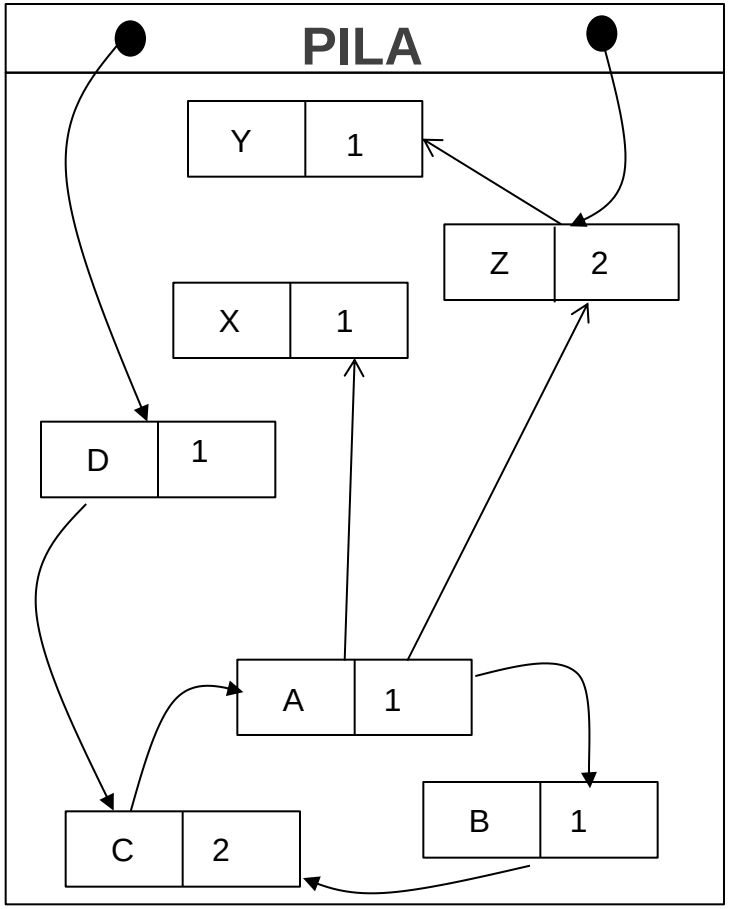
## Copia en dos direcciones

- Esta estrategia funciona dividiendo el heap en dos partes iguales.
- Cuando es requerido un bloque en el heap, el lenguaje ubica los espacios que solicita el programador en la primera mitad del heap hasta que el mismo no tenga más capacidad.
- Cuando esto sucede los bloques realmente usados son copiados a la segunda mitad hasta que colme su capacidad y pueda repetirse la estrategia con la primera mitad sucesivamente.
- Los lenguajes que utilizan esta estrategia no necesitan compactar la memoria.

# Gestión del Heap

## Marcado y barrido local

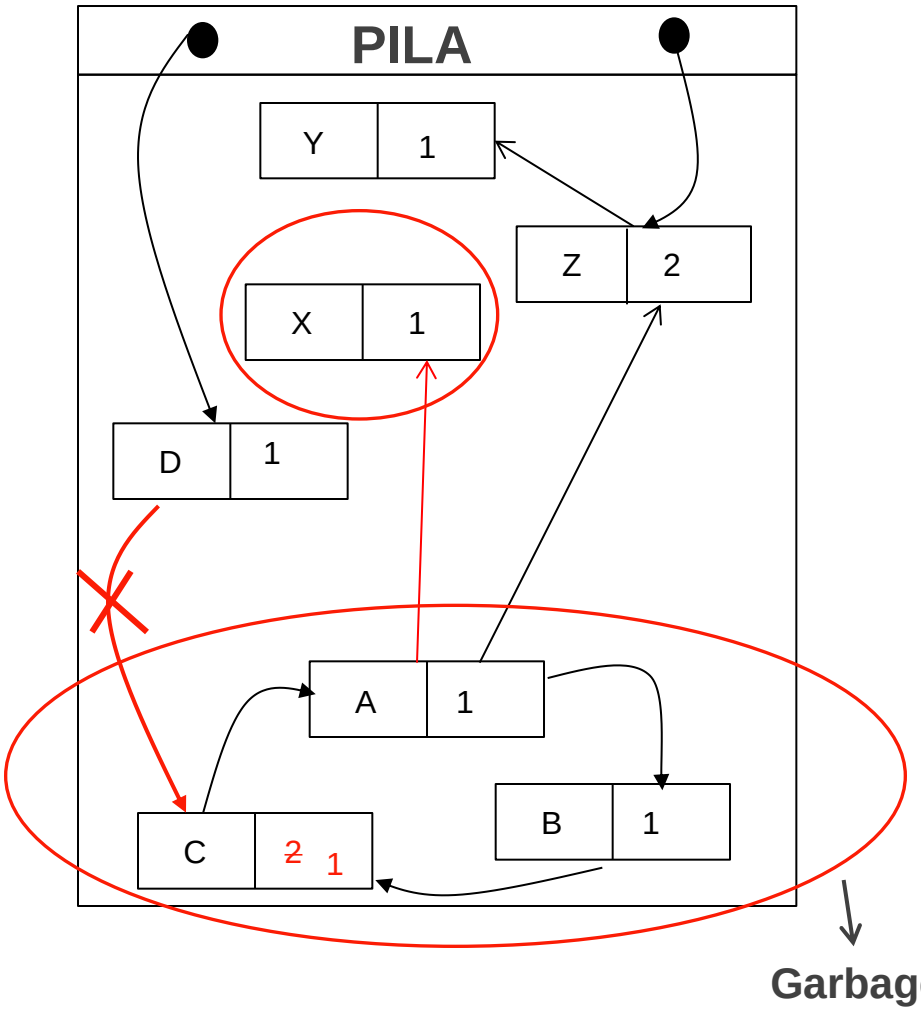
Combina el método de marcado y barrido con el método de conteo de referencia.




**Ahora supongamos que se pierde el puntero al bloque rotulado con C**

# Gestión del Heap

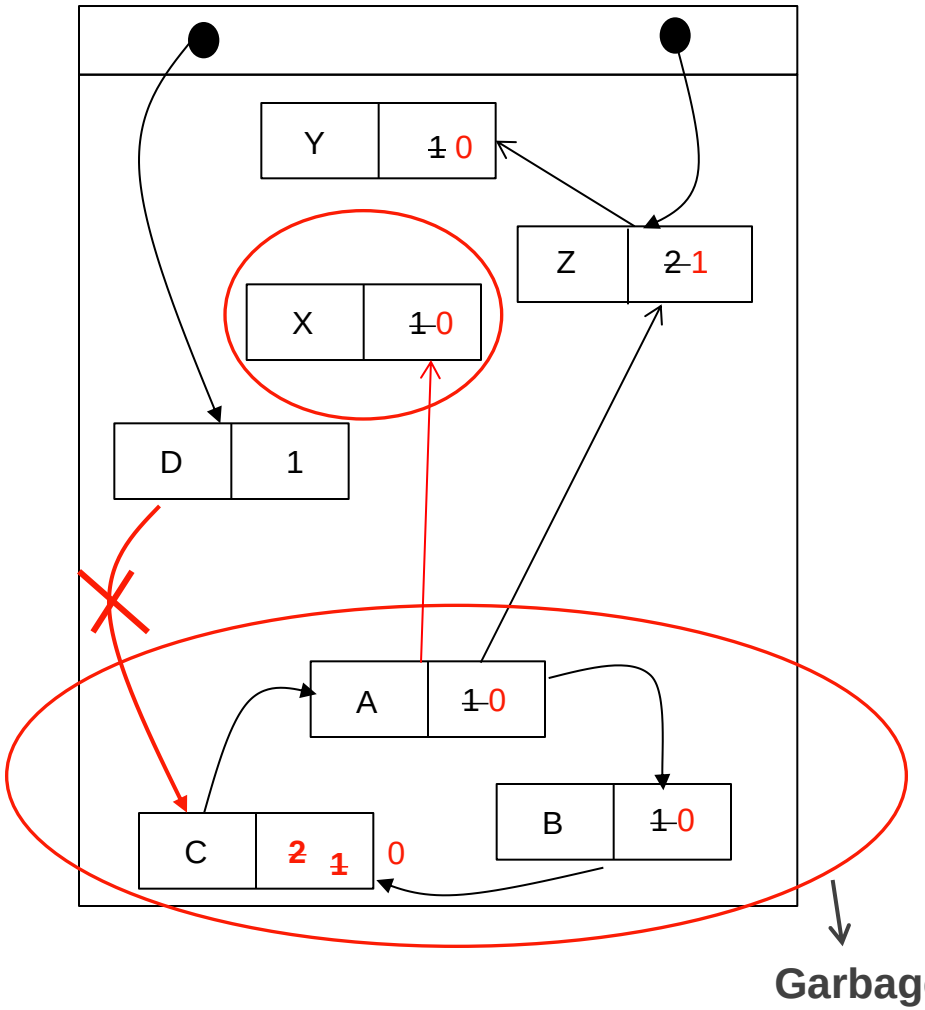
## Marcado y barrido local




**Garbage en los bloques A B C y X**

# Gestión del Heap

## Marcado y barrido local

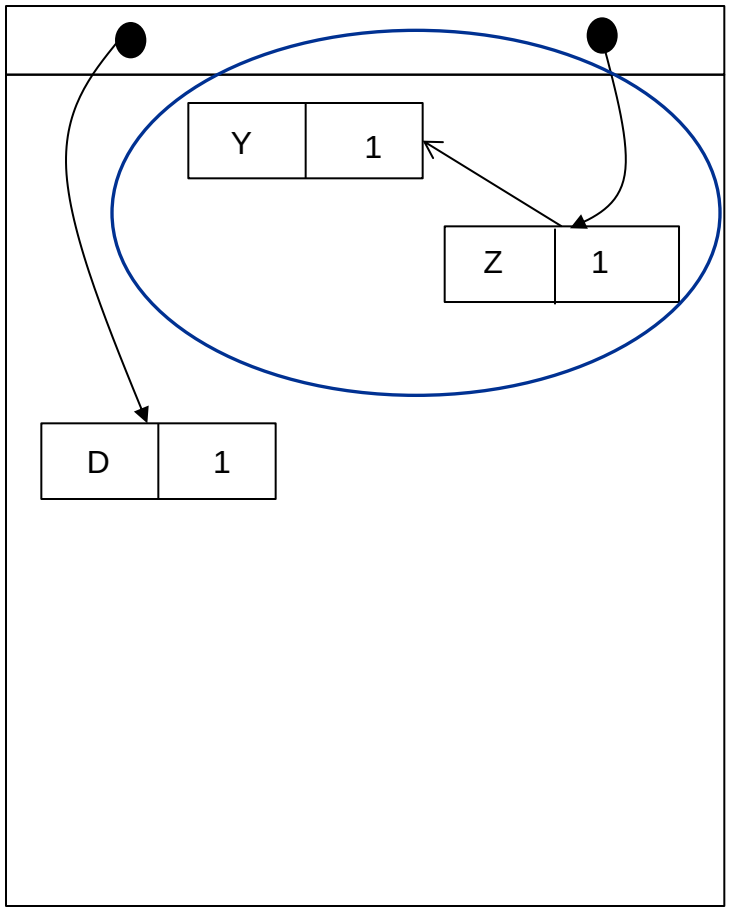



### ALGORITMO

- Visitar todos los arcos accesibles desde el arco borrado decrementando su contador en 1
- Todos los bloques que quedaron en 0 son sospechosos de ser garbage (A B C Y X)
- Se identifican aquellos bloques con su contador distinto de cero (Z)
- Se visitan todos los arcos desde los bloques identificados (arco de Z a Y) restaurando sus contadores
- Los bloques resultantes del proceso no generarán garbage

# Gestión del Heap

## Marcado y barrido local




- Los bloques resultantes del proceso no generarán garbage

# Gestión del Heap



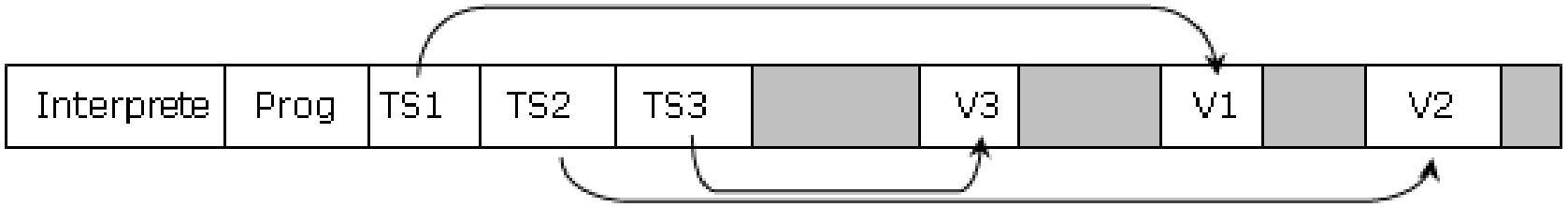
**Compactación**

# Gestión del Heap

## Compactación

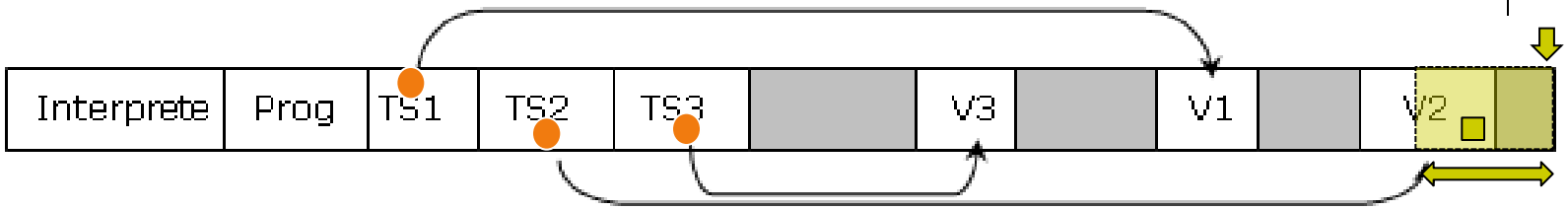
- El heap se encuentra colmado de espacios libres intercalados con los usados, por lo que es necesario llevar a estos espacios usados hacia un extremo ya que sin cierta compactación de bloques libres a bloques mas grandes, la ejecución se detendrá por falta de almacenamiento libre mas pronto de lo necesario.

- La técnica consiste en buscar las variables que están en las direcciones más altas colocándolas más atrás (posiciones mas bajas). Esto resulta sencillo pero muy lento, y los lenguajes dinámicos lo realizan en todo momento ya que de lo contrario no podrían funcionar.



# Gestión del Heap

## Algoritmo de compactación (GC en lenguajes dinámicos)




Se recorre la tabla de símbolos buscando el puntero que contenga la variable con la dirección más alta en el heap. (TS2 en el ejemplo).

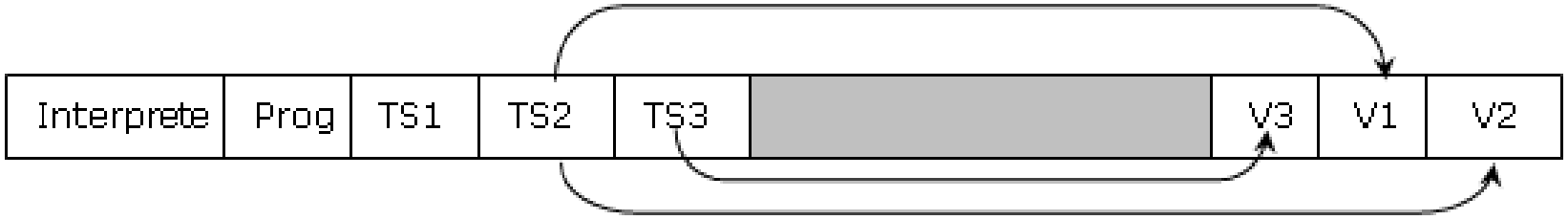
Se inserta una nueva dirección en el bloque (dirección) en forma temporal, que se refiere a la ubicación del extremo del heap.(0 en el primer caso)

Se suma el tamaño del bloque para ubicar al próximo elemento en forma contigua.

Se repiten estos pasos para todos los punteros de la tabla de símbolos o de la pila.

Se modifican los punteros en la tabla de símbolos con las nuevas direcciones.

Se desplazan todos los elementos uno por uno ya en forma definitiva a su nueva dirección, comenzando desde atrás hacia delante.



# Heap en objetos

## Objetos

Un **objeto** es una unidad dentro de un programa que posee un estado un comportamiento.

Los objetos pooseen mecanismos de acceso que actúan de forma idéntica respecto de las variables.

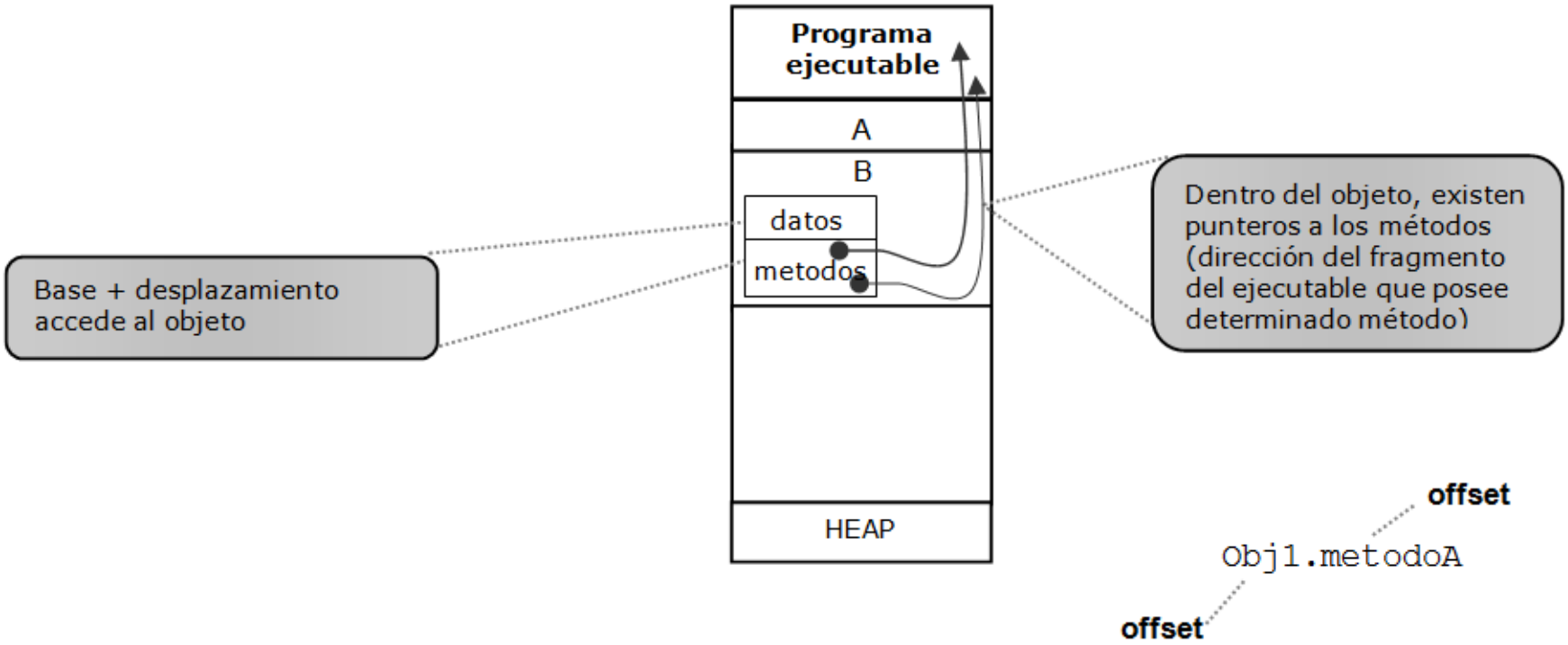
Estos se diferencian de las variables por su propia estructura, ya que se encuentran conformados por datos y direcciones de los métodos.

Los objetos, al igual que las variables, también poseen la característica de ubicarse en la pila o en el heap.

# Heap en objetos

## Objetos

- *Objetos en la pila (declarados).*

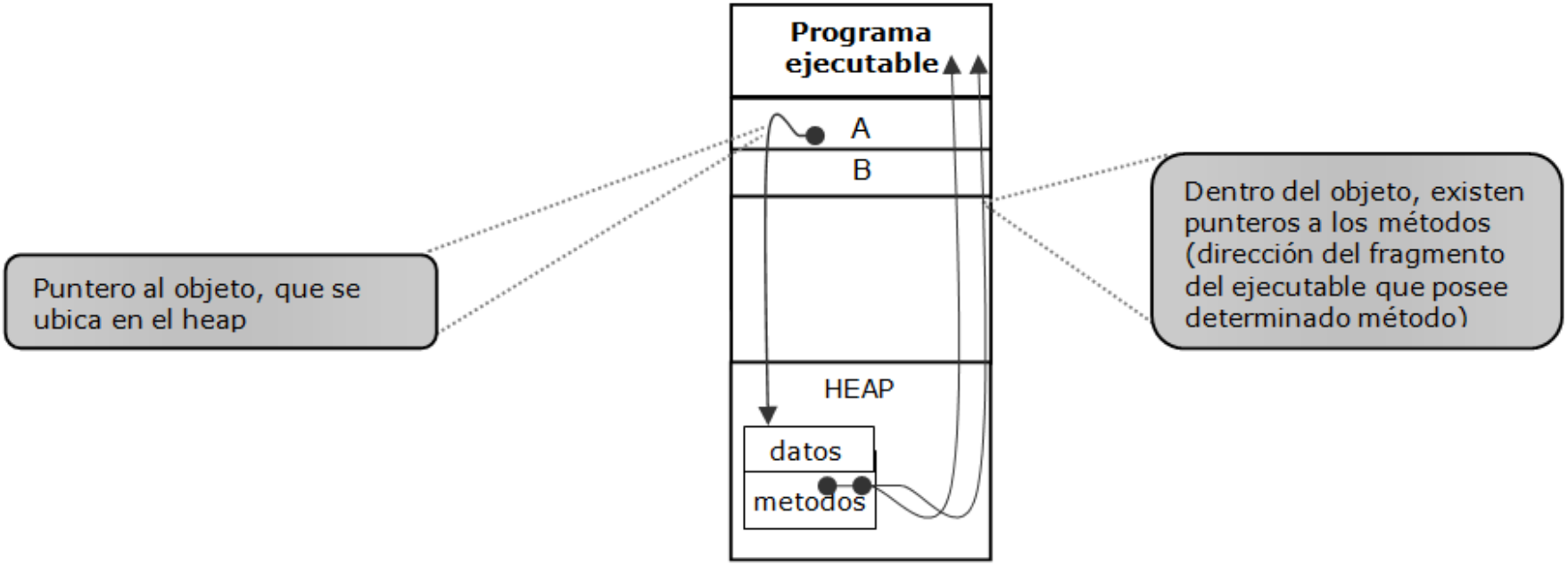



Para acceder al objeto se utiliza el mecanismo ***base + desplazamiento***, y luego mediante un puntero se accede a los métodos.

# Heap en objetos

## Objetos

- *Objetos en el heap (creado con el comando new o malloc)*




Existen lenguajes que poseen distintos tipos de objetos (SE y DIN)

En un lenguaje como Smalltalk (dinámico) los objetos se acceden con una entrada en la TS y un puntero hacia el objeto en el *heap*.

# Heap en objetos

## Objetos .NET y JAVA

En .NET los ***valores tipo*** son los que comúnmente conocemos como tipos simples (int, char, long, byte…) y en memoria se almacenan en una estructura de pila. Los ***valores referencia*** son los que comúnmente conocemos como objetos.

Estos lenguajes utilizan un **colector de basura generacional**.

Esto significa que clasifica los objetos en distintas generaciones, lo cual le permite realizar colecciones de basura parciales (de una o varias generaciones) y así evitar hacer siempre colecciones de basura completas de todo el heap

# Heap en objetos

## Objetos .NET y JAVA

**El colector de basura generacional** es indispensable para alcanzar el nivel de rendimiento necesario en una aplicación de alta concurrencia, y se basa en la siguiente regla heurística:

"Los objetos que han existido mucho tiempo, van a seguir existiendo durante mucho tiempo más.."

Es decir que si un objeto ha sobrevivido y ha promocionado de generación, lo más probable es que vaya a seguir sobreviviendo.

# Heap en objetos

## Objetos .NET y JAVA

JAVA (JVM) introdujo la recolección de basura generacional luego de que lo haya hecho el CLR (máquina virtual) de .NET

JAVA permite mas flexibilidad en la configuración del algoritmo de CG, a diferencia de .NET

Ambos lenguajes utilizan el "***algoritmo de marcado y barrido***" y se ejecutan en forma automática o a pedido del programador.

# Heap en objetos

## Conteo de referencias

Python tiene un sistema de tipado dinámico y manejo automatizado de memoria utilizando conteo de referencias.

También poseen esta característica:

- Perl
- Ruby
- Scheme
- Visual Basic 6.0
- Objective C

# Gestión del Heap



¿Preguntas?