Universidad de Huelva

# Tema 6
# Organización y gestión de la memoria

# Índice

6.1 Organización de la memoria en tiempo de ejecución

6.2 Zona de código

6.3 Memoria estática

6.4 Memoria de pila

6.5 Memoria con reserva dinámica

6.6 Recolección de basura

# Índice

**6.1 Organización de la memoria en tiempo de ejecución**

6.2 Zona de código

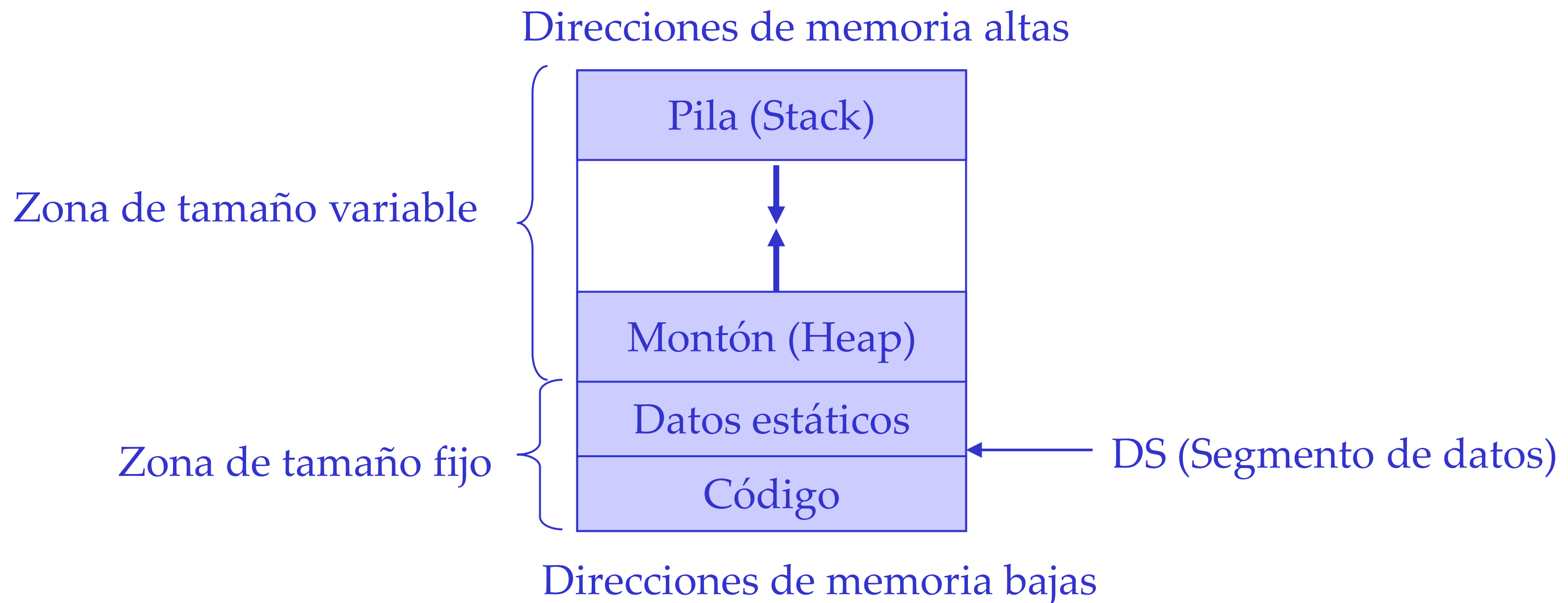
6.3 Memoria estática

6.4 Memoria de pila

6.5 Memoria con reserva dinámica

6.6 Recolección de basura

- La organización depende del tipo de lenguaje (declarativos, imperativos), del compilador y del sistema operativo.
- Estructura general de los lenguajes imperativos (Fortran, C, C++) :



- El primer responsable de la gestión de la memoria es el sistema operativo.
- Al ejecutar un programa, un módulo del sistema operativo, llamado *cargador*, asigna la cantidad de memoria y carga el código a ejecutar (almacenado en un fichero) en la zona de código.
- El sistema operativo debe detectar la posible colisión entre el montón y la pila. En estos casos puede abortar el programa o aumentar la cantidad de memoria asignada.

- En sistemas con paginación de memoria, el esquema anterior puede estar troceado y disperso entre la memoria real del sistema y la memoria virtual.
- Toda referencia a una posición de memoria dentro del código debe ser relativa a la posición asignada al proceso por el sistema operativo.

6.1 Organización de la memoria en tiempo de ejecución

**6.2 Zona de código**

6.3 Memoria estática

6.4 Memoria de pila

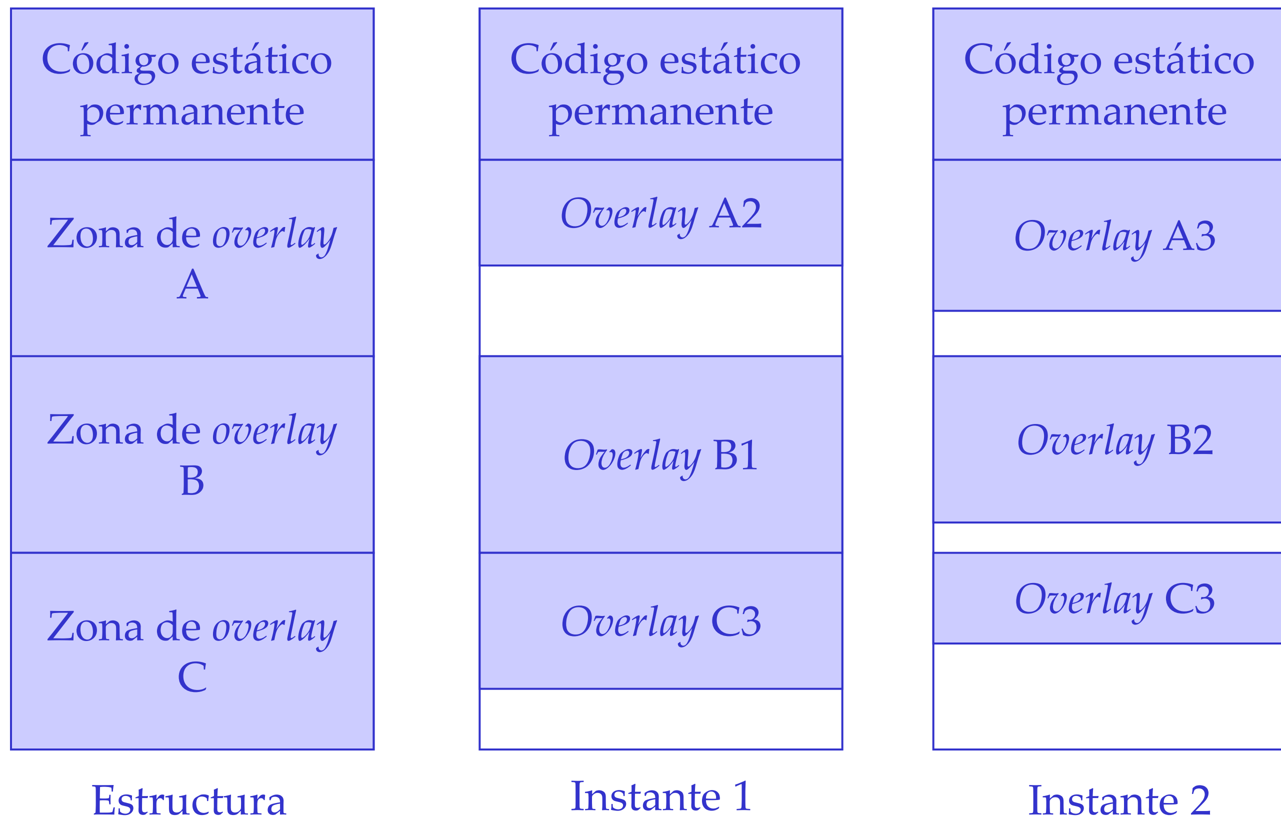
6.5 Memoria con reserva dinámica

6.6 Recolección de basura

- Contiene las intrucciones del programa a ejecutar, escritas en código máquina, es decir, la traducción a código máquina de todas los procedimientos y funciones del programa.
- El tamaño del código y su contenido se calculan en tiempo de compilación.
- El fichero ejecutable contiene todo este código, además de información relativa al tamaño de los diferentes bloques de memoria necesitados. Esta información es utilizada para cargar el programa en memoria para su ejecución.

- Normalmente el código se considera un bloque compacto.
- Algunos compiladores, por el contrario, fragmentan el código en solapas (*overlays*) cuando la memoria disponible es menor que el tamaño del programa.
- Las solapas son secciones de código que se cargan en memoria de manera independiente. A lo largo del tiempo, dos solapas pueden haber ocupado la misma zona de memoria.
- El compilador debe decidir cómo agrupar las funciones en solapas de manera que no se realicen demasiadas cargas de código a lo largo de la ejecución.

- Ejemplo de un programa con varios overlays:



6.1 Organización de la memoria en tiempo de ejecución

6.2 Zona de código

**6.3 Memoria estática**

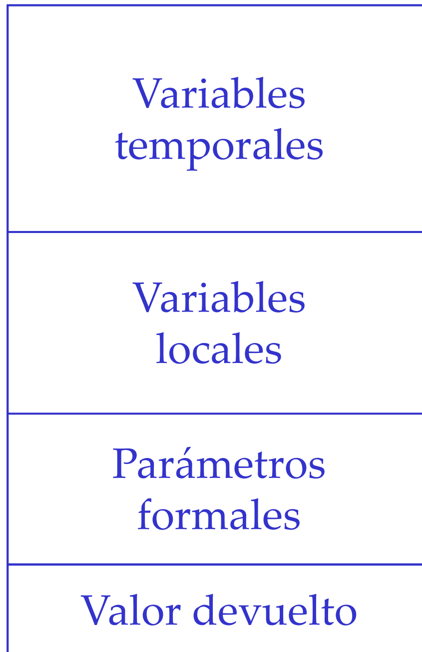
6.4 Memoria de pila

6.5 Memoria con reserva dinámica

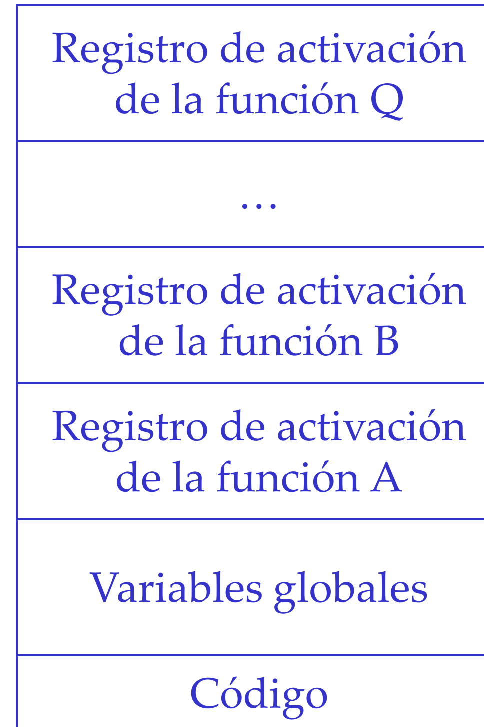
6.6 Recolección de basura

- Es la forma más sencilla de gestión de la memoria.
- El compilador asigna una posición de memoria fija a cada variable que se utilice en el programa.
- Era la única forma de gestión de memoria en los primeros compiladores (primeras versiones de FORTRAN).
- Actualmente sólo se utiliza para almacenar las constantes y variables globales del programa.
- La asignación de memoria se realiza en tiempo de compilación de forma consecutiva, teniendo en cuenta el tamaño de la variable a asignar.

- La dirección asociada a cada variable es constante y relativa al comienzo del segmento de datos (DS).
- Características de los compiladores basados exclusivamente en memoria estática:
  - no pueden manejar funciones recursivas, ya que estas requieren varias instancias de cada variable de la función.
  - cada función tiene asignado un registro de activación que contiene los parámetros, variables locales y variables temporales de la función.
  - la memoria estática contiene las variables globales y la secuencia de registros de activación de cada función.




Estructura de un registro de activación




Estructura de memoria de un programa basado en memoria estática

6.1 Organización de la memoria en tiempo de ejecución

6.2 Zona de código

6.3 Memoria estática

**6.4 Memoria de pila**

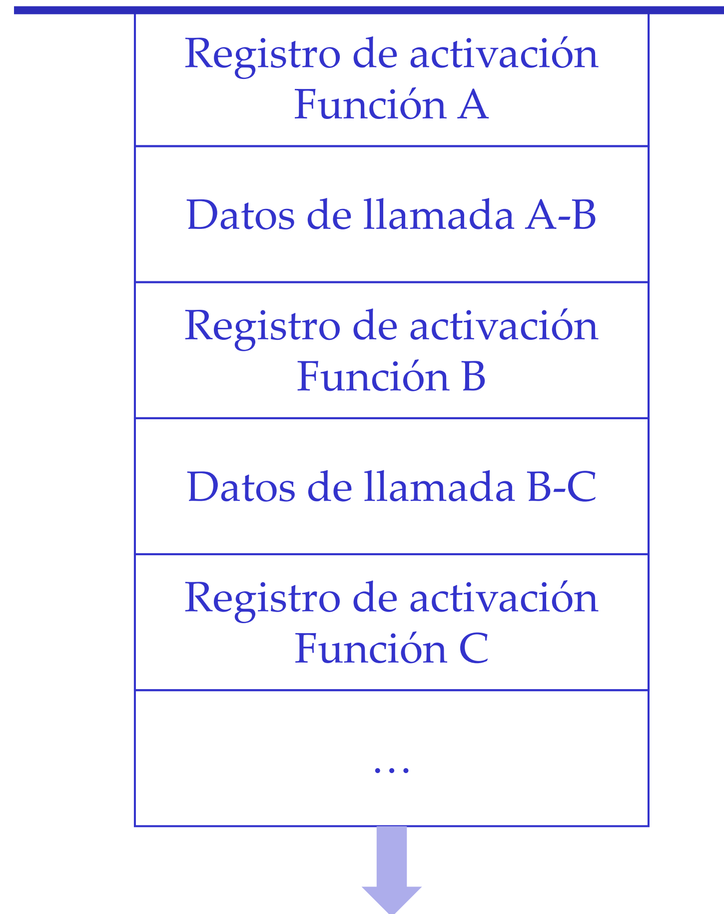
6.5 Memoria con reserva dinámica

6.6 Recolección de basura

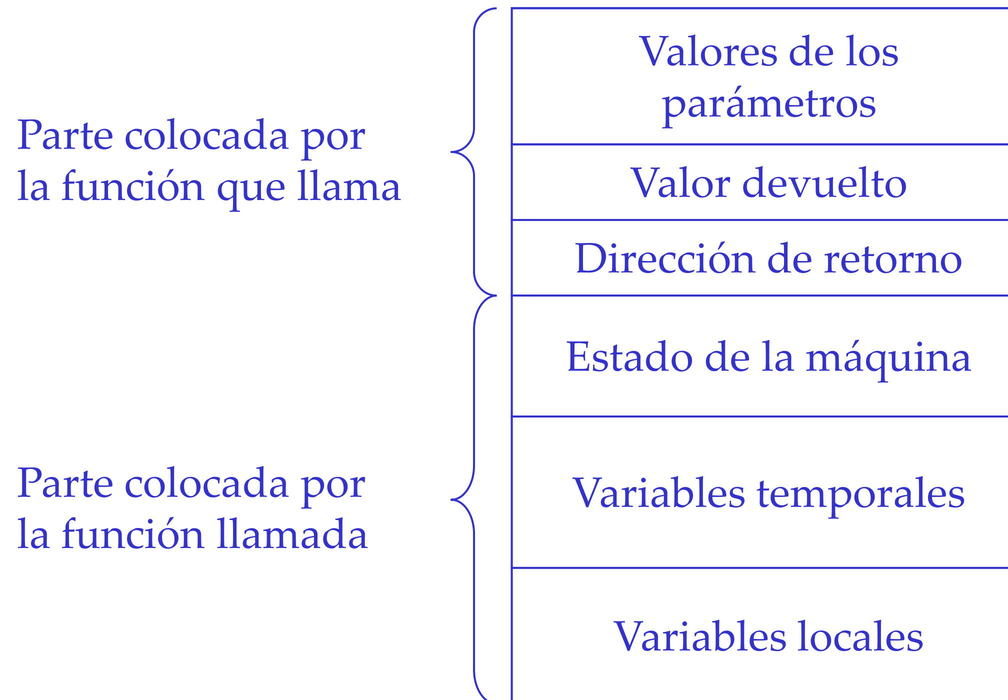
- Se introdujo para manejar lenguajes estructurados con llamadas recursivas.
- Las funciones recursivas requieren manejar diferentes instancias del registro de activación, es decir, del conjunto de valores de sus variables en cada ejecución de la función.
- Estos registros de activación se almacenan en forma de pila, de manera que el registro de la cima de la pila corresponde a la función en ejecución.
- Al terminar la ejecución de una función debe desapilarse el registro de activación y pasar el control a la función que se encuentre en la cima de la pila.

- Al traspasar el control de una función a otra es necesario almacenar el estado de la máquina, es decir, el conjunto de valores de los registros del procesador.
- También es necesario almacenar la dirección del comienzo del registro de activación de la función llamante (*Frame Pointer*), para poder retomar el control una vez terminada la función llamada.
- Esta información se añade al contenido del registro de activación de cada función.

- Estructura general de la pila:



- Estructura de un registro de activación para un programa con gestión de memoria basada en una pila:

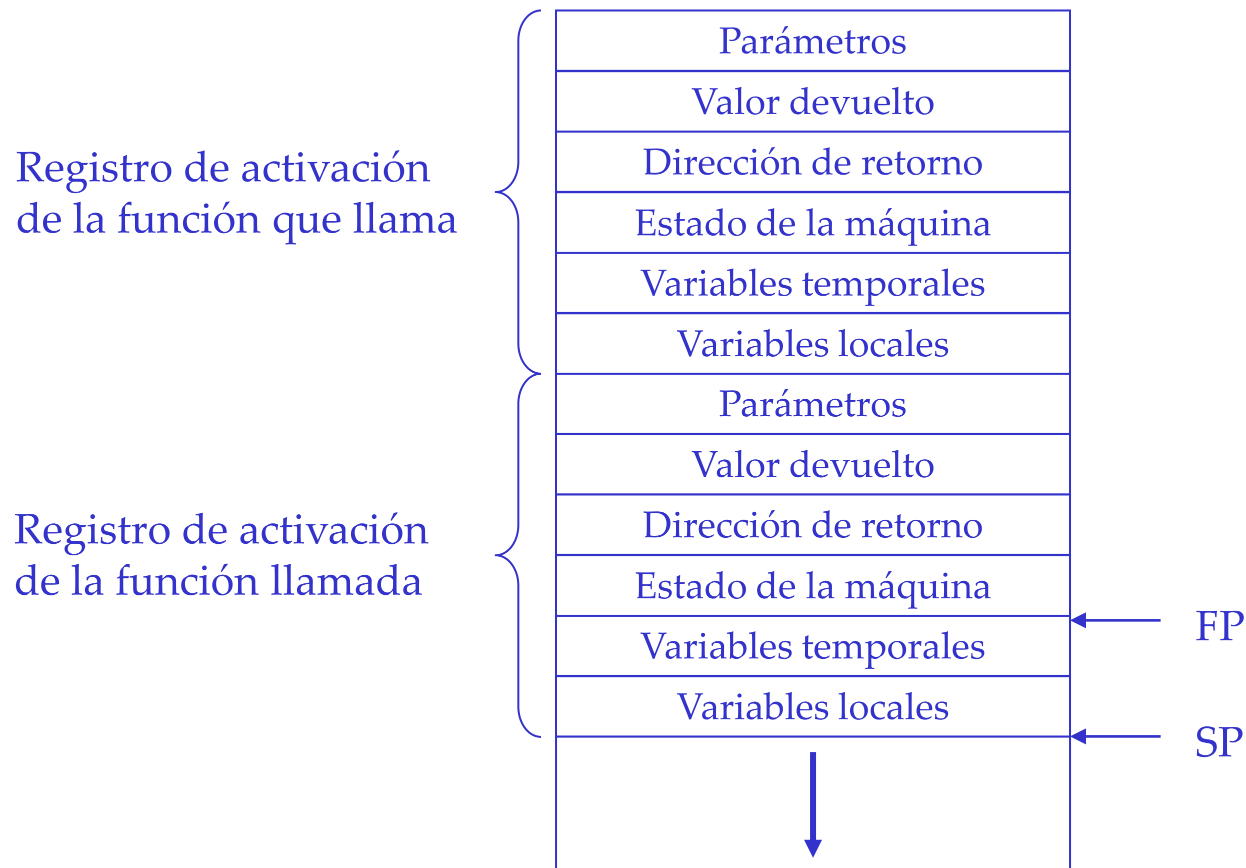


- El procesador maneja dos registros especiales:
  - Stack Pointer (SP): dirección de la cima de la pila.
  - Frame Pointer (FP): dirección base del registro de activación de la función activa.

- El procedimiento de llamada es el siguiente:
  - Se almacena el valor de los parámetros que se pasan a la función llamada.
  - Se reserva espacio para el valor devuelto.
  - Se almacena el valor de la dirección del código de la función que llama (dirección de retorno).
  - Se pasa el control a la función llamada.
  - Se almacena el estado de la máquina (incluye SP y FP).
  - Comienza la ejecución del código de la función llamada.

- El procedimiento de retorno es el siguiente:
  - Se asigna el valor devuelto.
  - Se restaura el contenido del estado de la máquina. Esto modifica el valor de FP y SP y provoca la liberación de la memoria ocupada por el registro de activación de la función llamada.
  - Se restaura el valor de la dirección del código de la función que llama (dirección de retorno).
  - Se devuelve el control a la función que llama.
  - Se almacena el valor devuelto en la variable local o temporal adecuada.

- Ejemplo del estado de la pila en un proceso de llamadas entre funciones.



6.1 Organización de la memoria en tiempo de ejecución

6.2 Zona de código

6.3 Memoria estática

6.4 Memoria de pila

**6.5 Memoria con reserva dinámica**

6.6 Recolección de basura

## Características generales

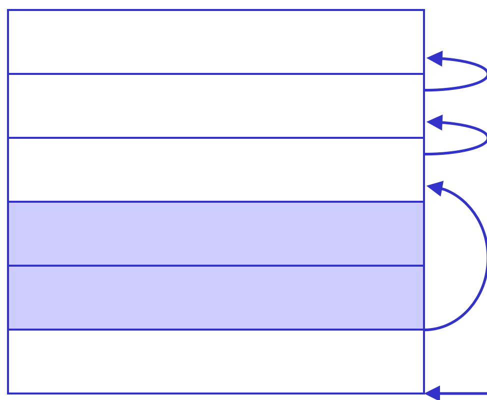
- Se conoce como *montículo* o *montón* (*heap*).
- Se introdujo para manejar estructuras dinámicas: listas, árboles, grafos, etc.
- La programación orientada a objetos hace un uso intensivo de esta memoria, ya que todo objeto se almacena en el montón.
- Las operaciones básicas sobre el montón son el alojamiento y el desalojo: *malloc/free* en lenguajes orientados a proceso o *new/delete* en lenguajes orientados a objetos.
- Las operaciones de alojamiento y desalojo se desarrollan en una librería especial que añade el compilador a todos los programas y que se encarga de la gestión de la memoria dinámica.

## Características generales

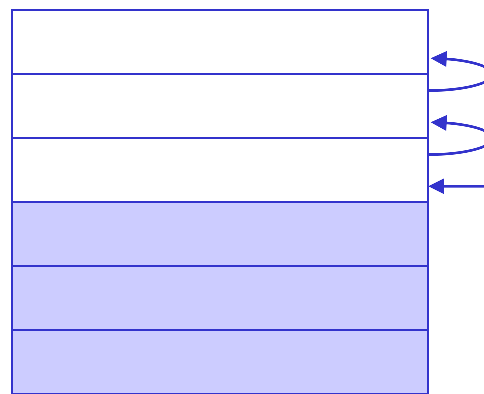
- El alojamiento consiste en asignar un bloque de memoria de un tamaño dado.
- El desalojo consiste en liberar un bloque de memoria, indicando que dicho espacio puede ser reutilizado en nuevos alojamientos.
- La gestión del montón requiere de técnicas adecuadas que optimicen el espacio que se ocupa y el tiempo de ejecución de las operaciones de alojo y desalojo.
- En los siguientes subapartados se discuten algunas de estas técnicas.

## Bloques de longitud fija

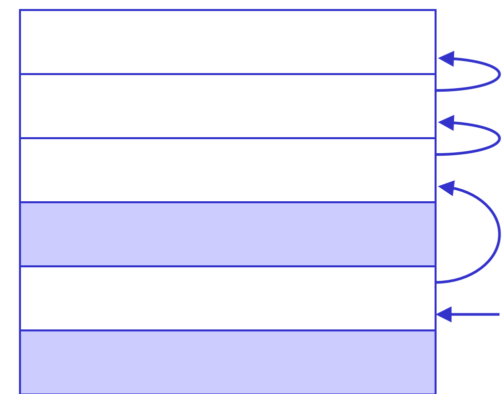
- El montón se gestiona como una lista enlazada de bloques del mismo tamaño. Sólo se almacena el puntero al primer bloque.
- El alojamiento consiste en asignar el primer bloque. El desalojo consiste en colocar el bloque liberado como primer bloque
- Problema: en la práctica las peticiones de memoria no son de tamaño fijo.



(a) Estado inicial



(b) Alojamiento



(c) Desalojo

## Montón con estructura simple

- El montón se gestiona como una lista enlazada de bloques de distinto tamaño. Sólo se almacena el puntero al primer bloque.
- La estructura de cada bloque es la siguiente:

- Al asignar un bloque, se puede rellenar con datos la zona libre y el enlace al siguiente bloque, pero no se puede utilizar el campo del tamaño.
- Alojamiento:
  - Estrategia del primer bloque: se recorre la lista hasta encontrar un bloque de tamaño mayor o igual a la memoria solicitada. Se divide el bloque en dos trozos: el primero se devuelve como memoria asignada y el segundo se almacena como un nuevo bloque de menor tamaño.

## Montón con estructura simple

- Alojamiento:
  - Estrategia del mejor bloque: se recorre la lista hasta encontrar el bloque de tamaño más parecido a la memoria solicitada. Esta estrategia genera una alta fragmentación, por lo que no se suele utilizar.
  - Estrategia del peor bloque: se asigna siempre el bloque de mayor tamaño. El problema es que elimina los bloques grandes que pueden necesitarse para objetos grandes.
- Desalojo:
  - Asignar el bloque como el primero de la lista. Esto tiende a fragmentar mucho los bloques ya que hace difícil la fusión de bloques contiguos.
  - Mantener una lista ordenada por posición de memoria. Exige insertar el bloque liberado en su posición (es más lento), pero permite fundir bloques rápidamente.

## Estructura de bloques marcados

- Estructura de los bloques:

| Indicador de bloque libre | Tamaño | Enlace al bloque anterior | Zona libre | Enlace al bloque siguiente | Tamaño | Indicador de bloque libre |
|---|---|---|---|---|---|---|

- Campos (almacenados simétricamente al principio y al final):
  - Indicador de bloque libre
  - Tamaño máximo
  - Enlaces (mantiene una lista doblemente enlazada)
- La zona de datos incluye la zona libre y los enlaces
- Alojamiento:
  - Admite las mismas estrategias que la estructura simple

## Estructura de bloques marcados

- Desalojo: Se pueden dar 4 casos

(1º) Los bloques anterior y posterior están ocupados

- Se marca el bloque como libre y se asignan los enlaces para insertar el bloque al comienzo de la lista

(2º) El bloque anterior está libre y el posterior ocupado

- Se amplía el bloque anterior uniéndole el bloque a liberar.
- Los valores de los enlaces corresponden a los del bloque anterior.

## Estructura de bloques marcados

(3º) El bloque anterior está ocupado y el posterior está libre

- Se amplía el bloque posterior uniéndole el bloque a liberar.
- Los valores de los enlaces corresponden a los del bloque posterior.

(4º) Los bloques anterior y posterior están libres

- Se extrae el bloque posterior de la lista (actualización de los enlaces).
- Se amplía el bloque anterior con el espacio del bloque a liberar y el del bloque posterior.
- Los enlaces son los correspondientes al bloque anterior.

## Estructura de división fija

- Sólo admite bloques de ciertos tamaños y mantiene listas doblemente enlazadas para cada uno de los tamaños disponibles
- Estructura de los bloques:

| Indicador de bloque libre | Tamaño | Código de división | Enlace al bloque anterior | Zona libre | Enlace al bloque siguiente |
|---|---|---|---|---|---|

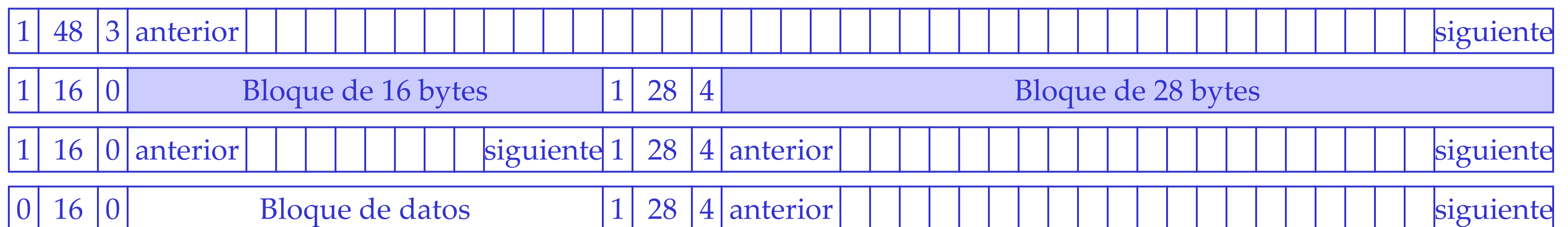
- La zona de datos incluye la zona libre y los enlaces
- Tamaños disponibles:
  - Bytes de control (C): bloque libre + tamaño + código de división
  - Serie de Fibonacci: Fn = Fn-1 + Fn-2 + C
  - Serie de división binaria: Fn = Fn-1 + Fn-1 + C

## Estructura de división fija

- Ejemplos de series de tamaños:
  - Bytes de control (4) : bloque libre (1), tamaño (2), código de división (1)
  - Serie de Fibonacci: F1 = 8; F2 = 16;  Tamaños: 8, 16, 28, 48, 80, 132, 216, ...
  - Serie de división binaria: F1 = 8; .... Tamaños: 8, 20, 44, 92, 188, 380, ...
- Alojamiento:
  - Dada una petición de tamaño L, buscar el tamaño Fn igual o inmediatamente superior (Fn-1 < L  <= Fn )
  - Si no hay bloques libres de tamaño Fn, buscar el primer bloque disponible de tamaño mayor (Fm)
  - Si Fm > Fn dividir el bloque Fm en subbloques de los tamaños de la serie hasta alcanzar el tamaño Fn, actualizando los enlaces para mantener las listas.

## Estructura de división fija

- Alojamiento:
  - Cada vez que se divide un bloque, el subbloque izquierdo tiene un control de división 0 y al subbloque derecho se le incrementa el control de división.
  - Ejemplo:
    - Tamaños: 4, 8 , 16, 28, 48, 80, 132
    - Tamaño solicitado 12; tamaño a asignar 16; primer tamaño libre 48.



## Estructura de división fija

- Desalojo:
  - Estudiar el código de división para saber si es un bloque izquierdo ( c = 0 ) o derecho ( c > 0 ).
  - Buscar la información del bloque compañero. Si es un bloque izquierdo, el bloque compañero estará en la posición del bloque más el tamaño. Si es un bloque derecho, entonces proviene de una división en la que se conoce el tamaño del bloque izquierdo, por lo que la posición del bloque compañero será la posición del bloque menos el tamaño del compañero.
  - Si el bloque compañero está libre, fundir ambos bloques en el bloque original y repetir el proceso hasta que no se pueda agrupar más.
  - Actualizar los enlaces a medida que se van realizando las operaciones de liberación y agrupación.

## Estructura de división fija

- Ventajas de la estructura de división fija:
  - Mantiene listas a bloques de tamaño fijo.
  - Conocido el tamaño a alojar, el bloque a asignar es el primero de la lista. No hay que hacer recorridos de una lista para buscar un tamaño adecuado.
  - Al desalojar, se puede colocar el bloque liberado al comienzo de la lista. No es necesario buscar su posición ya que la información de control permite conocer cuales son los bloques adyacentes.
- Inconvenientes:
  - Se desperdicia memoria al considerar tamaños fijos (por ejemplo, para alojar un bloque de 17 bytes de datos se necesita un bloque de 28 bytes).
  - Sólo une dos bloques contiguos si son bloques compañeros.

6.1 Organización de la memoria en tiempo de ejecución

6.2 Zona de código

6.3 Memoria estática

6.4 Memoria de pila

6.5 Memoria con reserva dinámica

**6.6 Recolección de basura**

## Definición

- En la mayoría de los lenguajes imperativos (C, C++, Fortran, ...) la asignación y el desalojo de la memoria es responsabilidad del programador.
- En lenguajes declarativos (Prolog, Lisp, ...) , la liberación de la memoria es responsabilidad del compilador o el intérprete del lenguaje.
- Algunos lenguajes, como Java o Python, también liberan al programador de la tarea de desalojar la memoria dinámica.
- La liberación automática de memoria dinámica se conoce como recolección de basura (*Garbage collection*).
- Existen varias técnicas para desarrollar la recolección de basura. Las más utilizadas son *marcar y barrer* (*mark-and-sweep*) y *recolección por copia* (*copying collection*).

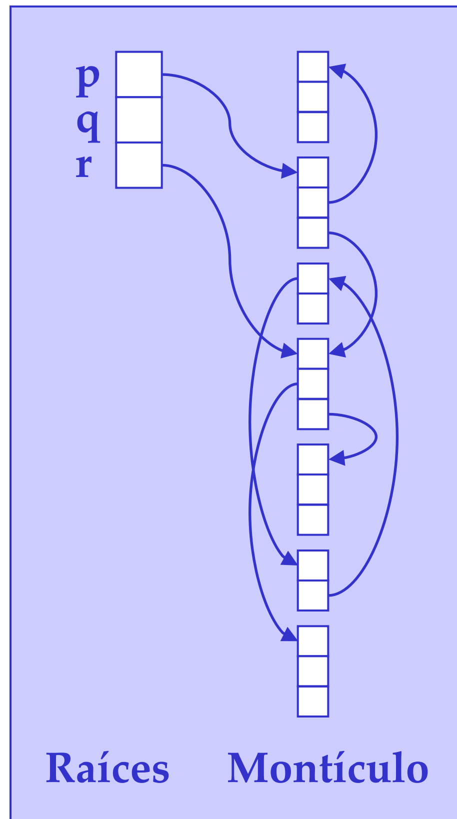
## Desocupación sobre la marcha (*free as you go*)

- Consiste en desalojar los objetos en el momento en el que no están referenciados por ningún puntero.
- Se utiliza un contador de enlaces en cada objeto. Cuando un puntero se referencia a un objeto, se incrementa su contador. Cuando un puntero deja de referenciar a un objeto, se decrementa su contador. Si al decrementar el contador se llega a cero, se libera el objeto.
- El inconveniente es que no permite liberar estructuras circulares (ya que los contadores nunca llegan a cero) y que requiere muchas instrucciones para cambiar el valor de un puntero, lo que hace que el código sea más lento.
- En la práctica estos inconvenientes provocan que no se utilice esta técnica salvo en lenguajes que limitan por definición el uso de referencias a objetos (Rust).

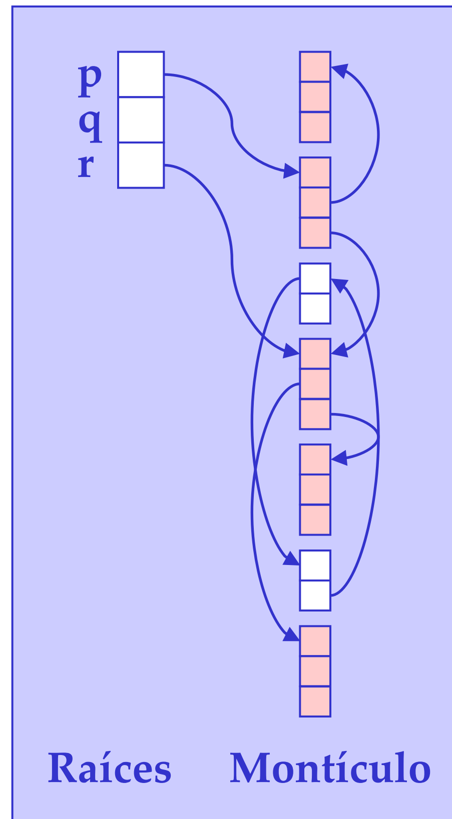
## Marcar y barrer (*mark-and-sweep*)

- El origen de los enlaces a los objetos alojados en la memoria dinámica se encuentra en las variables del programa. Estas variables se conocen como raíces (*roots*).
- Todos los objetos deben tener un campo que permite marcarlos.
- El método de recolección de basura tiene dos fases: marcado y barrido.
- El método de marcado se ejecuta sobre todas las raíces y hace lo siguiente:
  - Si el objeto ya está marcado, no hace nada.
  - Si el objeto no está marcado, lo marca y ejecuta el método de marcar para todos los campos del objeto que sean referencias a otro objeto.
- El método de barrido consiste en un recorrido de todos los bloques del montículo.
  - si el bloque está marcado se desmarca.
  - si el bloque no está marcado se libera.

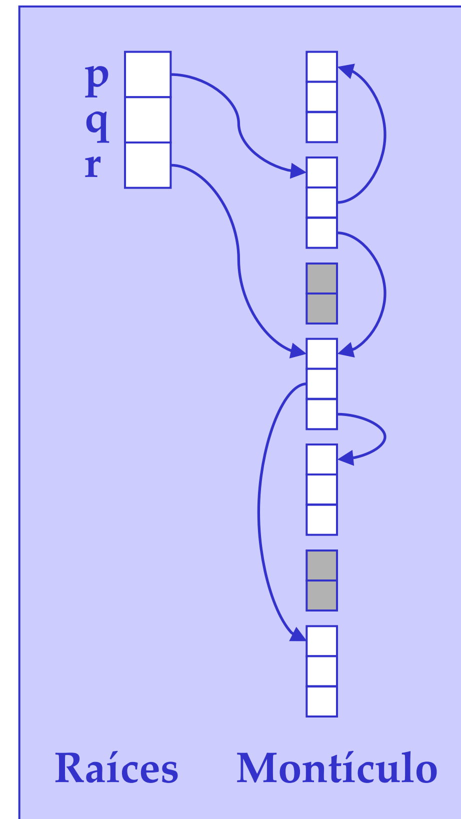
## Marcar y barrer (*mark-and-sweep*)



Inicio



Marcado



Barrido

## Marcar y barrer (*mark-and-sweep*)

- Para detectar las raices se utilizan contadores en los objetos, pero en este caso solo se cuentan el número de referencias de las variables locales y globales. Si un campo de un objeto referencia a otro, esto no incrementa el contador de enlaces.
- Al desapilar el registro de activación de una función hay que decrementar los contadores de los objetos referenciados por las variables locales de la función.
- De esta forma, los objetos que tienen su contador en positivo son los que están siendo referenciados por variables y, por tanto, son las raices del heap.

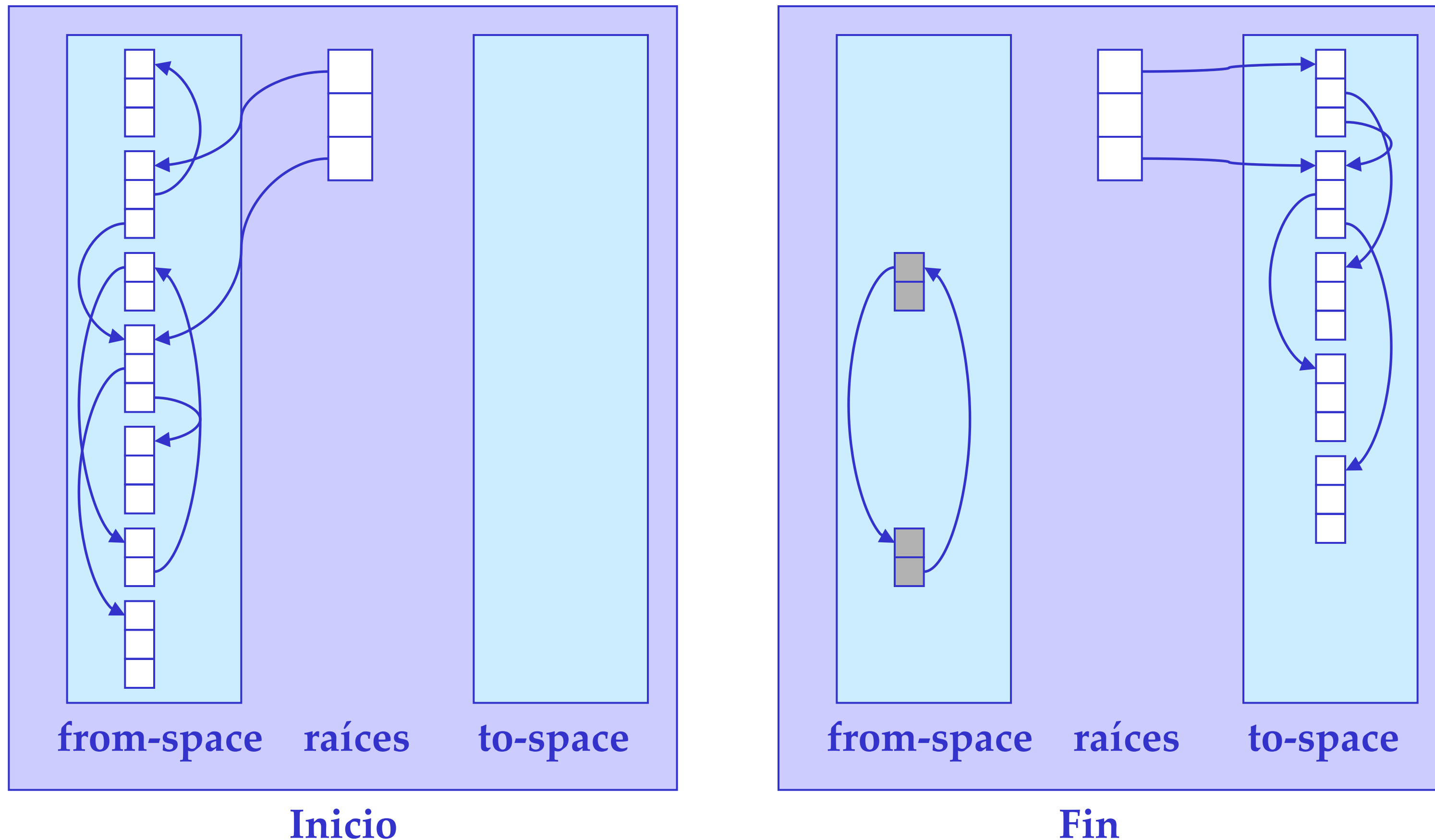
## Recolección por copia (*copying collection*)

- Consiste en considerar dos bloques de memoria dinámica. Un bloque que se utiliza realmente como montículo (*from-space*) y otro del mismo tamaño que se considera libre (*to-space*).
- El método consiste en copiar desde *from-space* a *to-space* todos los objetos alcanzables desde las raíces e intercambiar los papeles de los bloques.
- El algoritmo de copia debe realizar un recorrido desde las raíces, pasando por todos los campos de cada objeto que sean referencias a otros objetos.
- El método de copia hace lo siguiente:
  - Si el objeto no ha sido copiado, se copia su contenido desde *from-space* a *to-space*. El original en *from-space* se marca como copiado y se indica la nueva posición.
  - Si el objeto ya ha sido copiado, se actualiza la referencia a la nueva posición en *to-space*.

## Recolección por copia (*copying collection*)

- Esta técnica tiene la ventaja de que el alojamiento es secuencial y no se necesita ninguna forma de desalojo. La desventaja es que requiere el doble de memoria.
- Otra desventaja importante de esta técnica es el control de las raices. En este caso, la posición de los objetos cambia y, por tanto, los valores de las variables locales y globales que referencien a los objetos deben cambiar también. Esto obliga a identificar las raices (no solo a manejar un contador).
- Para identificar las raices se consideran los registros de activación como si fueran representaciones de objetos (en los que debemos conocer cuales son los campos que almacenan referencias). Además, estos registros de activación deben incluir un enlace al registro de activación de la función anterior. De esta forma se pueden leer secuencialmente los registros e identificar las raices.

## Recolección por copia (*copying collection*)



## Recolección generacional

- Los objetos recién creados tienen más probabilidad de ser liberados a corto plazo. Los objetos que no se han eliminado en algunas iteraciones probablemente formen parte del núcleo duro de las estructuras de datos del programa y nunca sean des-referenciados.
- El montículo se divide en generaciones. Los objetos más jóvenes se encuentran en la primera generación (G0). La generación G1 contiene objetos más antiguos que los de G0. Los de G2 son más viejos que los de G1, etc.
- La recolección se centra en G0. Cada cierto número de recolecciones se estudia también G1. Tras un número de recolecciones mayor se estudia G2, etc.
- Los objetos que sobreviven un cierto número de recolecciones, se copian a una generación superior.
- Esto permite reducir el tiempo de recolección, ya que el tamaño del montículo estudiado es menor.

## Recolección generacional

- La recolección generacional es una versión modificada de la recolección por copia donde el bloque *from-space* no corresponde al montículo completo sino al trozo que contiene una generación.
- La gestión de las raices es similar a la utilizada en el método de recolección por copia.

## Recolección incremental

- La recolección de basura es muy costosa en tiempo y puede ser problemática en sistemas en tiempo real.
- Los algoritmos incrementales se basan en que el proceso recolector no paralice al programa, sino que se ejecute en paralelo a este.
- Los objetos se dividen en tres tipos: blancos, que no han sido estudiados por el recolector; grises, que han sido estudiados pero cuyos campos no lo han sido; y negros, que han sido estudiados y cuyos hijos también lo han sido.
- Cuando no existen objetos de tipo gris, entonces todos los objetos de tipo blanco son basura y pueden ser eliminados.
- La ejecución del programa puede provocar que los objetos cambien de tipo (muten). El código del programa debe modificarse para que la creación de objetos y los cambios de punteros provoquen mutaciones en el color de los objetos.
- Esta técnica pretende realizar una versión paralela de la recolección por *mark-and-sweep*.