

Memoria y Punteros

- ▶ Materia Algoritmos y Estructuras de Datos
- ▶ Cátedra Schmidt
- ▶ Esquema de memoria y manejo del tipo puntero

Todo dato con que trabaja la computadora, (así como toda instrucción cuando se ejecuta), en los modelos habitualmente usados de computadoras, está en la memoria principal.

Tanto los datos como las instrucciones están codificados como sucesiones de ceros y unos.

Al momento de ejecutar un programa, para ese programa en particular se distinguen cuatro zonas distintas, todas ubicadas en la memoria principal de la computadora.

Cada zona es un 'segmento' y tiene una función determinada.

1 Memoria

1.1 División

A la memoria del ordenador la podemos considerar dividida en 4 segmentos lógicos:

CD	: <u>Code Segment</u>
DS	: <u>Data Segment</u>
SS	: <u>Stack Segment</u>
ES	: <u>Extra Segment (ó Heap)</u>

El Code Segment es el segmento donde se localizará el código resultante de compilar nuestra aplicación, es decir, la algoritmia en Código Máquina.

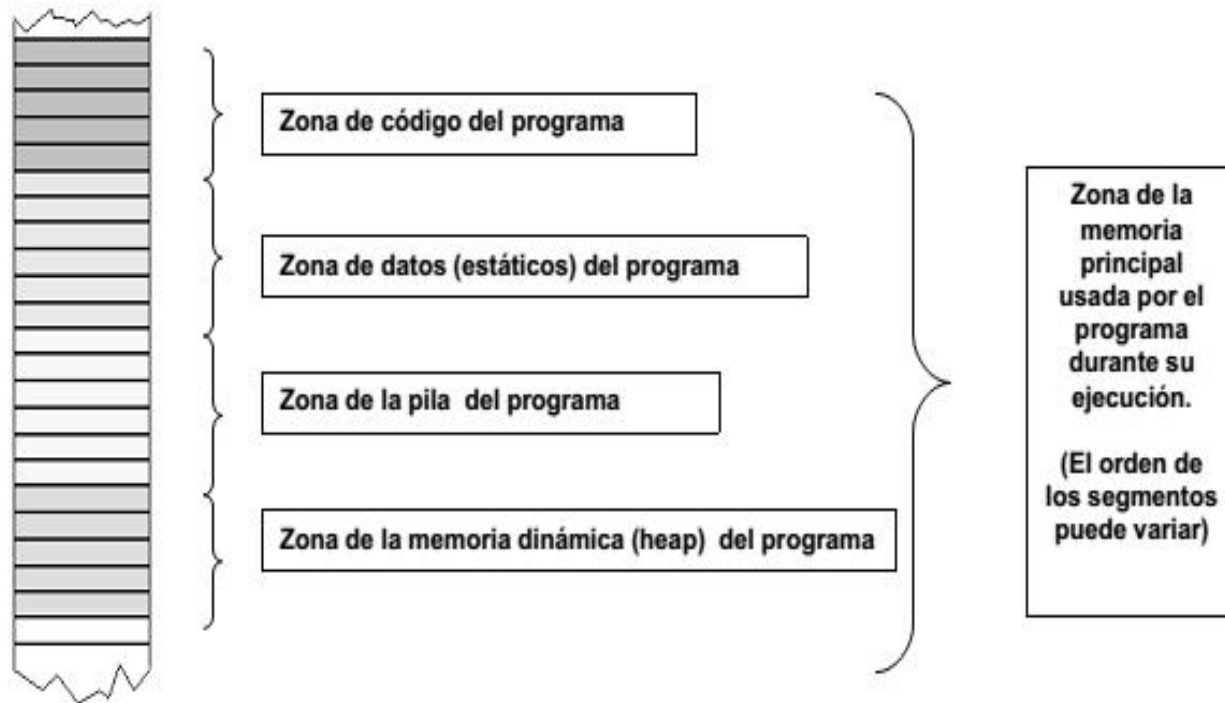
El Data Segment almacenará el contenido de las variables definidas en el modulo principal (variables glabales)

El Stack Segment almacenará el contenido de las variables locales en cada invocación de las funciones y/o procedimientos (incluyendo las del main en el caso de C/C++).

El Extra Segment está reservado para peticiones dinámicas de memoria.

Formas de graficar la memoria

Grafico de memoria contigua



1.2 Codificación

La información se almacena en la memoria en forma electromagnética.

La menor porción de información se denomina **bit** y permite representar **2 símbolos** (0 o 1, apagado o prendido, no o si, etc).

Se denomina **Byte** a la combinación de 8 bits y permite representar **256 símbolos** posibles.

Se denomina **Word** a la combinación de bytes que maneja el procesador (en procesadores de 16 bits serán 2 Bytes, en procesadores de 32bits serán 4 bytes, etc.)

Para definir capacidades de memoria, se suelen utilizar múltiplos del Byte como son:

8 bit	1 <u>Byte</u>
1024 <u>Bytes</u>	1 <u>Kilo</u> Byte (KB)
1024 <u>KBytes</u>	1 Mega Byte (MB)
1024 <u>MBytes</u>	1 <u>Giga</u> Byte (GB)
1024 <u>GBytes</u>	1 Tera Byte (TB)

Vamos a considerar en particular a los datos.
¿De qué modo se almacenan en la memoria?

Hay que recordar algunos conceptos: la memoria de la computadora se compone de celdas. Cada celda es un byte, es decir que está formada por 8 bits.

Todas las celdas de la memoria tienen el mismo tamaño, y son idénticas entre sí; solo se diferencian entre ellas por el número de posición que tienen asignado.

El número de posición de una celda es la **dirección** de la celda. Esa dirección es única para cada celda y no puede ser cambiada.

En ningún momento una computadora tiene ‘celdas vacías’. En toda celda hay bits. Cada bit puede tomar valor cero o uno.

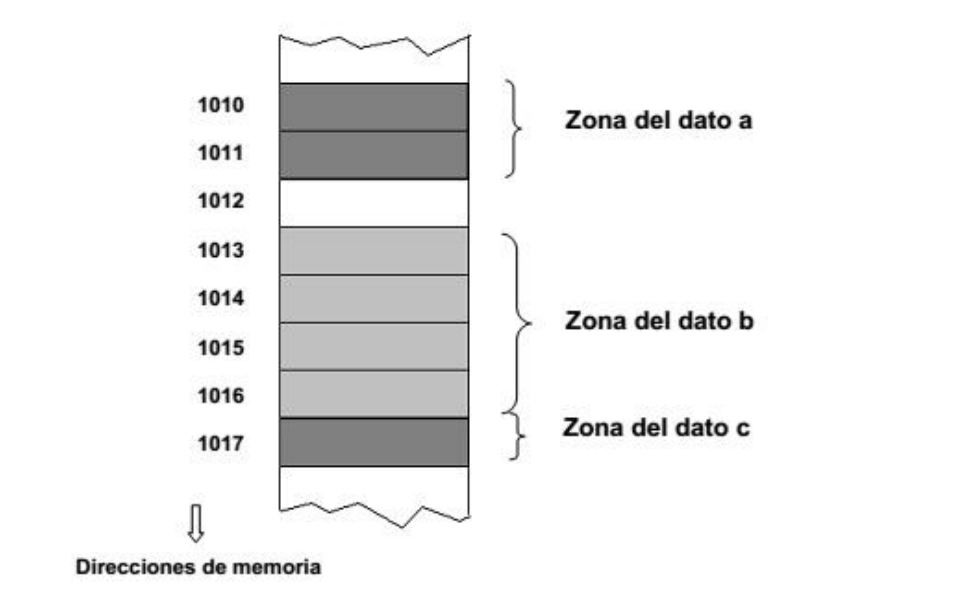
Mediante la codificación con dígitos binarios la computadora trata tanto los datos como las instrucciones.

1001	10110011
1002	00100110
1003	00100101
1004	11101101
1005	00001100

Cada dato contenido en la memoria ocupa una o más celdas, dependiendo del tamaño del mismo.

Por ejemplo, los caracteres ocupan una sola celda de memoria, los enteros ocupan al menos 2 celdas (en C, en C++ el tamaño está estandarizado), los reales ocupan al menos 4 celdas, y otros datos ocupan más celdas.

En todo caso, lo que siempre se verifica es que un dato, del tipo que sea, ocupa un conjunto de celdas que deben ser contiguas.



1.3 Sistemas de Numeración

Matemáticamente existe el concepto de Sistemas de Numeración, de los cuales el Sistema Decimal es el que utilizamos habitualmente y cuyos 10 símbolos bien conocemos:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Ahora bien, dijimos que la memoria es bi-estable, es decir, sólo maneja 2 símbolos, y para representar su contenido se suele utilizar el **Sistema Binario** de Numeración, cuyos símbolos son:

0, 1

De esta forma, la información contenida en 1 Byte se representaría como la combinación de 8 de estos símbolos (se los suele suceder con la letra “b” para indicar que esta en binario):

01011000b

Tener en cuenta que los bits de un Byte se numeran desde 0 de derecha a izquierda. Y a esta numeración se la suele llamar peso del bit. Así, el bit 1 vale 0 en el ejemplo anterior.

Como pueden notar, esto es poco práctico a la hora de manejar muchos bytes, con lo que se suele utilizar otro Sistema a tales efectos. Al mismo se lo denomina **Sistema Hexadecimal** y comprende 16 símbolos; a saber:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Su utilidad radica en que es fácilmente decodificable en **bits** ya que un número hexadecimal de 2 dígitos corresponde exactamente a un número binario de 8 dígitos.

1.4 Reglas de Conversión entre Sistemas de Numeración

Una regla práctica de conversión entre los símbolos esta dada por la siguiente tabla:

Hexadecimal	Binario	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Binario > Hexadecimal : conformar grupos de 4 dígitos binarios y cambiarlos por sus correspondientes dígitos hexadecimales (de requerir, agregar ceros a la izquierda del número inicial).

1011000b > 0101 1000 > 5 8 > 58h

Hexadecimal \rightarrow Binario : cambiar cada dígito hexadecimal por sus 4 correspondientes binarios.

58h \rightarrow 5 8 \rightarrow 0101 1000 \rightarrow 1011000b

Binario \rightarrow Decimal : multiplicar cada bit por (2 elevado al peso del bit) y sumar los resultados

Dígitos	1	0	1	1	0	0	0
Peso	6	5	4	3	2	1	0
2^{peso}	64	32	16	8	4	2	1
Dígito $\times 2^{\text{peso}}$	64	0	16	8	0	0	0
Suma	88						

Decimal › Binario : se divide sucesivamente el número decimal por 2 hasta tanto se obtenga como resultado un 0 o 1. Luego se compone el número binario con el cociente final seguido de la secuencia de restos hasta el resto inicial.

Dividendo	Divisor	Cociente	Resto
88	2	44	0
44	2	22	0
22	2	11	0
11	2	5	1
5	2	2	1
2	2	1	0

1011000

Hexadecimal › Decimal : multiplicar cada dígito por (16 elevado al peso del bit) y sumar los resultados

Dígitos	5	8
Peso	1	0
7^{peso}	16	1
Dígito x 2^{peso}	80	8
Suma	88	

Decimal › Hexadecimal : se divide sucesivamente el número decimal por 16 hasta tanto se obtenga como resultado un número entre 0 y 15. Luego se compone el número hexadecimal con el cociente final seguido de la secuencia de restos hasta el resto inicial. Representar cada uno de estos valores con su correspondiente símbolo hexadecimal.

Dividendo	Divisor	Cociente	Resto
88	16	5	5

55

(Nota: 5 es el dígito hexadecimal que le corresponde al 5 decimal)

1.5 Almacenamiento

Antes de explicar el tema en cuestión, es necesario explicar que cuando se almacena un número en memoria compuesto por más de un byte, se denomina byte más significativo al que guarda la porción de mayor orden de magnitud.

El número 516 equivale al binario 1000000100. Por ende son 2 bytes → el 00000010 y el byte 00000100. El primer byte es al que denominamos mas significativo por cuanto sus dígitos representan el mayor orden de magnitud (representa 512) en tanto que el otro se denomina menos significativo porque representa la fracción de menor orden de magnitud (en este caso 4).

Existen 2 técnicas para efectuar el almacenamiento de un **Word** en memoria. Las mismas se conocen bajo el nombre de:

Big Endian → el byte más significativo precede al menos significativo.

Es decir, el **0058h** se guardaría como **00 58**

Little Endian → el byte menos significativo precede al más significativo.

Es decir, el **0058h** se guardaría como **58 00**

En las PC se utiliza el segundo esquema y por ser éstas las más accesibles a los lectores del presente, es el que se utilizará para el desarrollo de los ejemplos de las secciones posteriores.

1.6 Direcccionamiento

Modélese a la memoria cual una matriz de 16 columnas donde la cantidad de filas dependerá de la memoria disponible. De esta forma, podemos referenciar una celda de la misma mediante el número de **Fila y Columna**, a los que llamaremos **Segmento y Desplazamiento** respectivamente y que por lo general se expresarán en **hexadecimal**.

	Desplazamiento															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Segmento 0																
Segmento 1																
...																
Segmento n																

De esta forma, en un procesador con palabras de 2 bytes, la celda señalada sería la **\$0001:0007** donde los 2 primeros bytes (Segmento) serían una palabra y los 2 segundos (Desplazamiento) serían otra palabra.

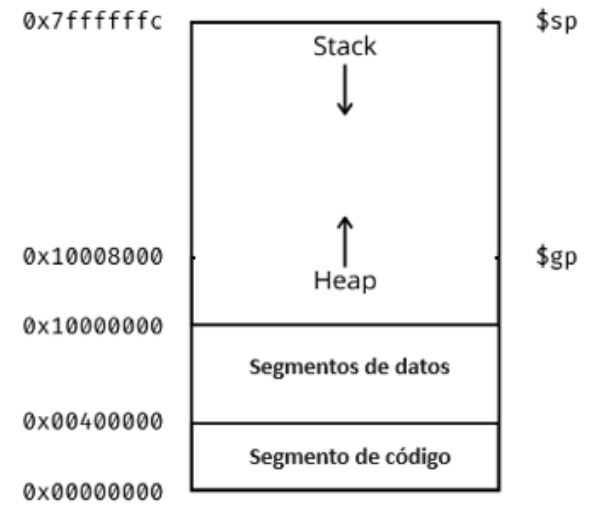
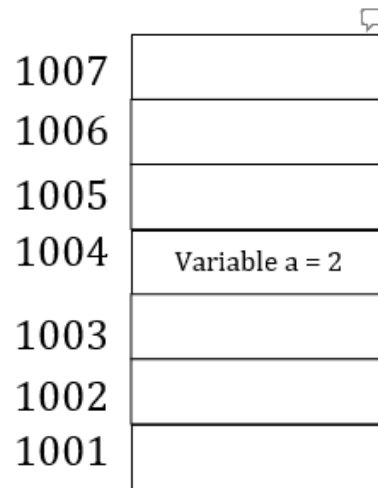
Esta conformación de la dirección no es caprichosa ya que permite “desplazarnos” sin cambiar de Segmento cada 16 bytes permitiéndonos mantener un punto fijo (Data Segment, Extra Segment, u otro). De hecho analicemos la siguiente dirección y concluyamos que se trata de la misma celda que la del esquema anterior.

\$0000:0017

	Desplazamiento															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Segmento 0																
Segmento 1																
...																
Segmento n																

Diagrama de memoria contigua

```
11  
12 int main() {  
13     short a = 2;  
14     return 0;  
15 }  
16
```



Al momento de terminar de ejecutar la línea 13, la variable a queda posicionada en la dirección de memoria 1004. Y el valor de a es 2.

Diagrama de memoria practico

```
11  
12 int main() {  
13     short a = 2;  
14     return 0;  
15 }  
16
```

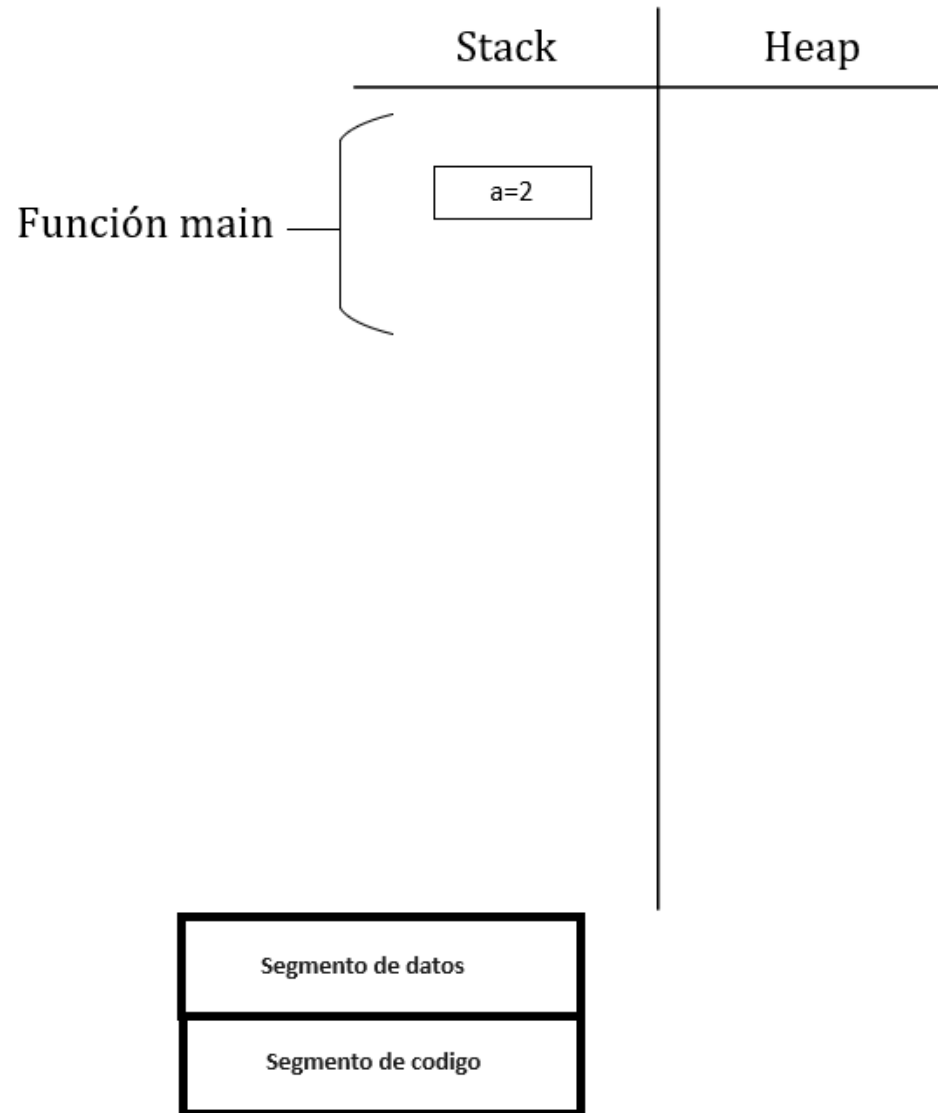


Diagrama de memoria practico

```
12
13 int funcion1() {
14     short a = 2;
15     return 0;
16 }
17
18 int main() {
19     short a = 2;
20     funcion1();
21     return 0;
22 }
23
```

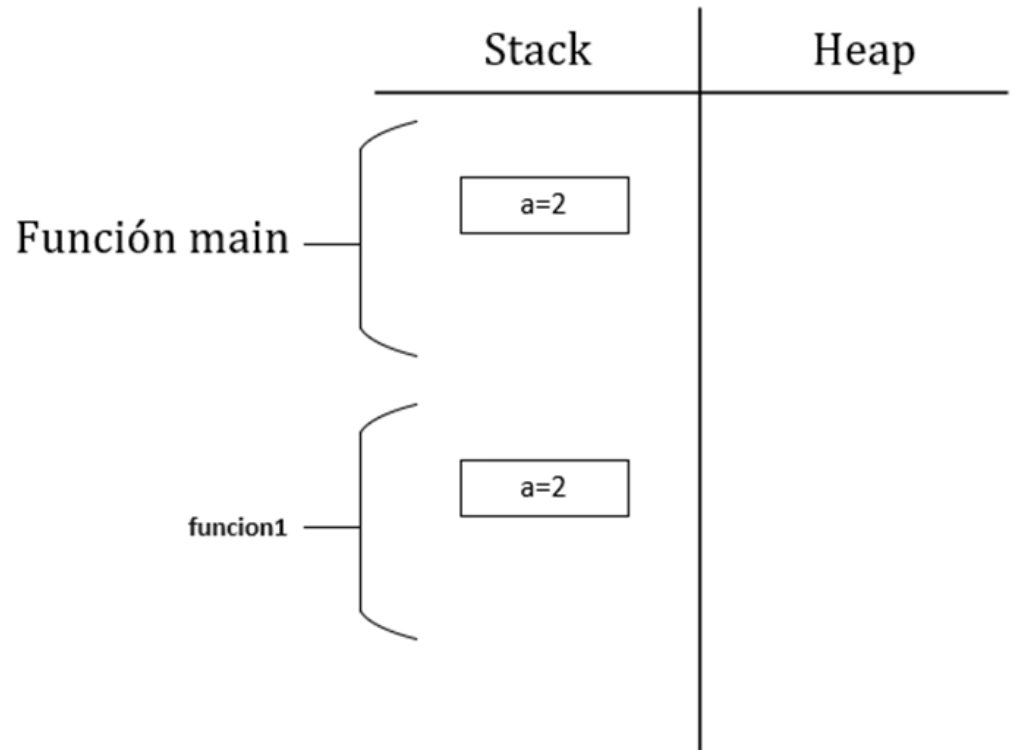
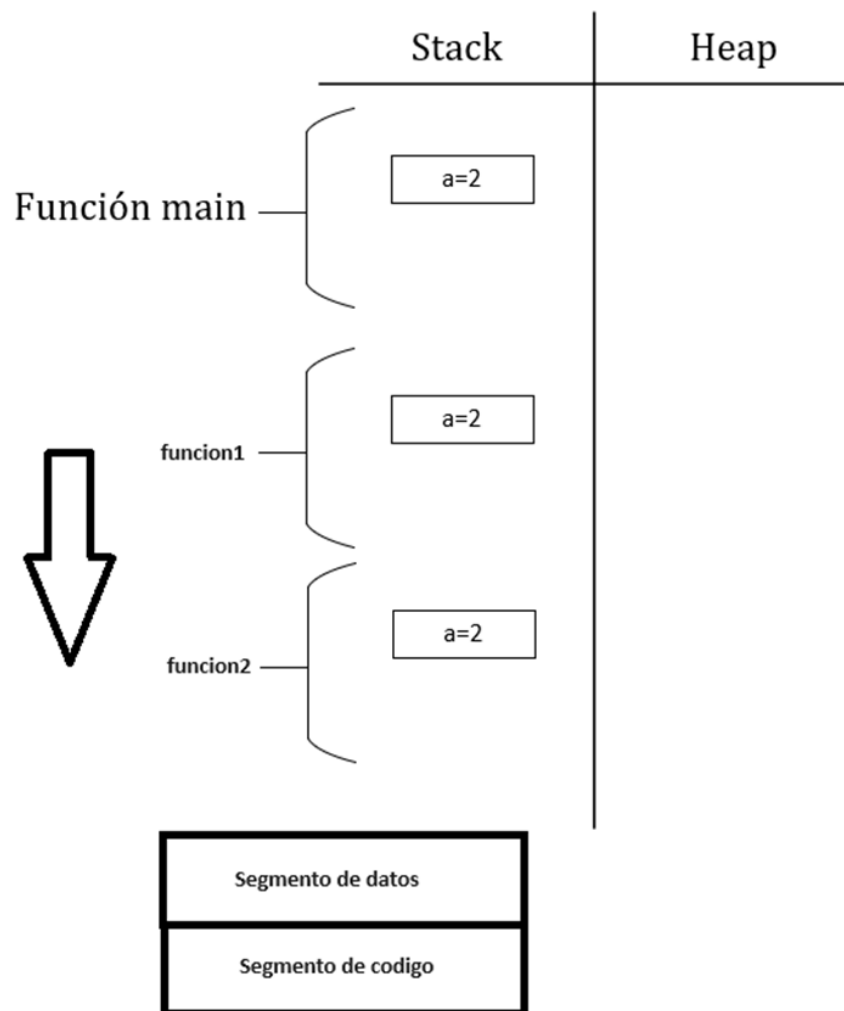


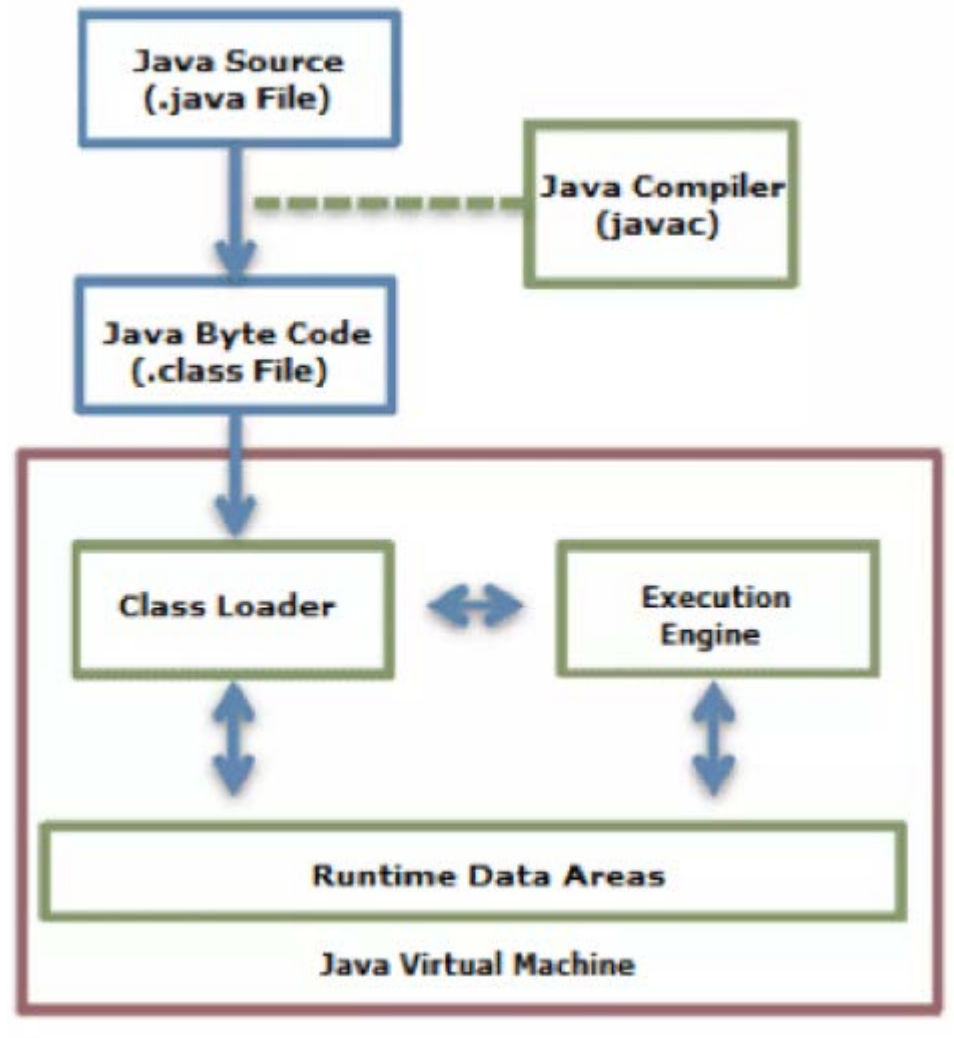
Diagrama de memoria practico

```
11
12 int funcion2() {
13     short a = 2;
14     return 0;
15 }
16
17
18 int funcion1() {
19     short a = 2;
20     funcion2();
21     return 0;
22 }
23
24 int main() {
25     short a = 2;
26     funcion1();
27     return 0;
28 }
29
```



Manejo de Memoria en JAVA

La arquitectura de la JVM contiene diversas partes, las cuales cumplen distintos objetivos. Al referirnos al manejo de memoria debemos mencionar que estamos hablando del área llamada "Runtime Data Areas"



La gestión de memoria en Java se maneja principalmente a través de un proceso llamado garbage collection (recolección de basura). A continuación, te explico cómo funciona:

1. Heap Memory:

- I. En Java, cuando creas un objeto utilizando new, este objeto se almacena en una zona de la memoria llamada heap.
- II. La memoria heap se divide en diferentes generaciones:
 - a) Young Generation: Aquí se almacenan los objetos recién creados. Se divide a su vez en:
 - i. Eden Space: Donde se crean inicialmente los objetos.
 - ii. Survivor Spaces (S0 y S1): Donde se mueven los objetos sobrevivientes del Eden Space.
 - b) Old Generation: Los objetos que sobreviven varias recolecciones de basura en la Young Generation se trasladan aquí.
 - c) Permanent Generation (Metaspace en versiones modernas de Java): Contiene la información de las clases y metadatos.

2. Garbage Collection:

- I. El garbage collector (GC) es responsable de liberar la memoria ocupada por objetos que ya no se están utilizando. Esto ocurre automáticamente sin que el programador tenga que liberar la memoria manualmente, como en C o C++.
 - a) Java utiliza varios algoritmos de recolección de basura, los más comunes son:
 - i. Serial GC: Un único hilo para la recolección de basura. Es simple pero puede pausar la aplicación.
 - ii. Parallel GC: Utiliza múltiples hilos para la recolección, mejorando el rendimiento en sistemas con múltiples núcleos.
 - iii. CMS (Concurrent Mark-Sweep) GC: Recolecta la basura sin detener completamente la aplicación.
 - iv. G1 (Garbage First) GC: Divide el heap en regiones y se centra en las que contienen más basura, buscando minimizar las pausas.

3. Referencias y Finalización:

- I. Java utiliza diferentes tipos de referencias (Strong, Soft, Weak, Phantom) para controlar la vida útil de los objetos.
- II. Cuando un objeto no tiene más referencias activas, el GC lo considera elegible para la recolección.
- III. El método `finalize()` se puede sobrescribir en una clase para realizar una limpieza antes de que el GC recoja el objeto, aunque su uso no es recomendado en versiones recientes.

4. Manejo de Memoria fuera del Heap:

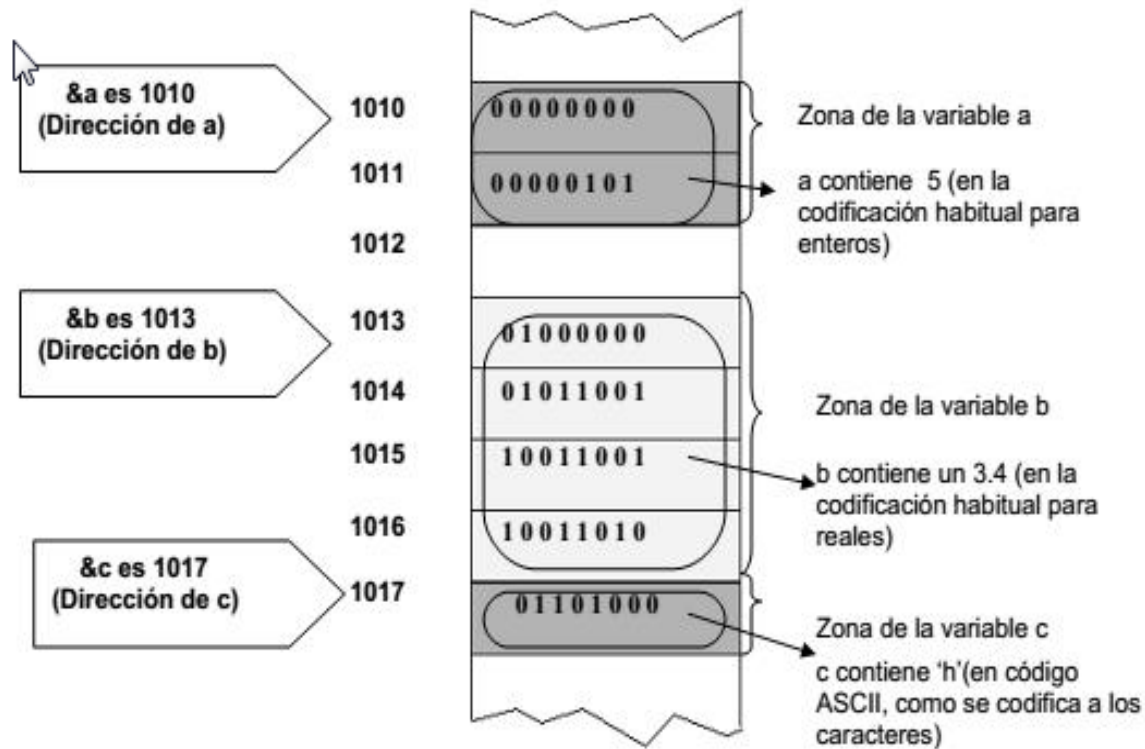
- I. Java también utiliza stack memory para manejar variables locales y llamadas a métodos.
- II. Direct Memory se utiliza en algunas operaciones de E/S y está fuera del control directo del garbage collector.

5. Optimización:

- I. Es importante entender cómo funciona la gestión de memoria en Java para optimizar el rendimiento de aplicaciones, evitando fugas de memoria y asegurando que el garbage collector funcione de manera eficiente.

Los punteros

Un puntero es un tipo de dato que corresponde a una dirección de memoria, o al valor NULL (0, cero). Sirven para manipular direcciones de elementos (de 'objetos', usando esta palabra en sentido general).



En el gráfico anterior se ha indicado el contenido de cada variable tal como se almacena; a la derecha se especifica el valor correspondiente a cada secuencia de bits (los codificados en ASCII, los enteros en complemento a 2, los reales en la notación IEEE 754).

Según la situación graficada antes a modo de ejemplo, a contiene el dato 5.

La dirección de a, llamada &a es 1010, b contiene el dato 3.4.

La dirección de b es &b, con valor 1013 c contiene el dato 'h'.

La dirección de c es &c, con valor 1017.

2 Punteros

2.1 Concepto

Un puntero es un tipo de variable que **almacena direcciones de memoria**.

Para llevar a cabo su función de almacenamiento debe ocupar el tamaño de una palabra (Word). En el caso de procesadores de 16 bits, serían 2 bytes, para los de 32, de 4 bytes y, para los de 64, 8 bytes.

2.2 Usos y ventajas de los punteros

- Permiten el acceso a cualquier posición de la memoria, para ser leída o para ser escrita (en los casos en que esto seaposible).
- Permiten una manera alternativa de pasaje por referencia.
- Permiten solicitar memoria que no fue reservada al inicio del programa. Esto se conoce como uso de memoria dinámica.
- Son el soporte de enlace que utilizan estructuras de datos dinámicas en memoria como las listas, pilas, colas y árboles.

Declaración

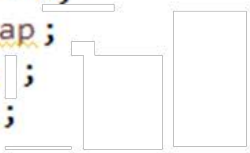
Para definir un tipo de dato como puntero tenemos 2 alternativas.

Puntero Genérico:

```
Object punteroGenerico;
```

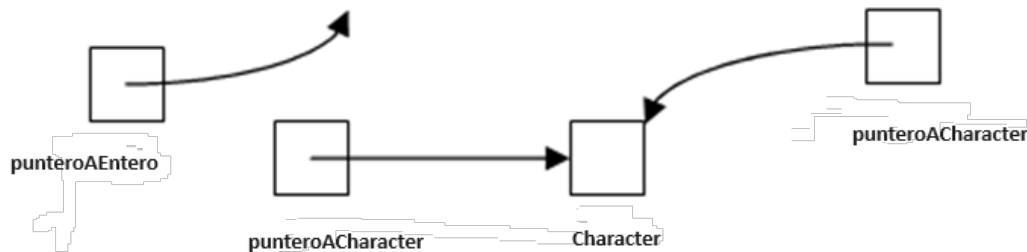
Puntero a un Tipo de Dato:

```
int enteroEnStack = 0;  
Integer enteroEnHeap;  
String textoEnHeap;  
int[] vectorEnHeap;
```



Diagramas

Los punteros son variables, por lo que se los modela como tales, es decir: cajas contenedoras con un nombre asociado. La diferencia esta en la diagramación de su contenido, el cual bien podría ser una dirección de memoria (\$FA00:4567) pero sería muy confuso trabajarlo. En lugar de ello, se suele representar con flechas que llegan a la dirección de memoria a la cual apuntan.



Operaciones sobre Punteros

- **Asignación:** el puntero al que se le asigna la dirección debe ser un puntero a un tipo de dato compatible con el de la dirección de memoria que se le quiere asignar. El puntero genérico es siempre compatible.
- **Comparación por Igualdad / Desigualdad:** evalúa si dos direcciones de memoria son las mismas. Los tipos de datos a los que apuntan los punteros deben ser compatibles.
- **Incrementar / Decrementar un puntero:** consiste básicamente en sumar a la dirección de memoria un número entero n . Este número entero no está expresado en bytes sino que se corresponde con $n * \langle\langle\text{tamaño del tipo de dato apuntado}\rangle\rangle$ bytes. Es decir, si apunto a un entero, y le sumo 1 quedará apuntando al siguiente entero, si apunto a un registro y le sumo 2 quedará apuntando a un área de memoria distante de la primera 2 veces el tamaño del registro. Esto es utilizado para iterar vectores.

	C++
Asignación	<code>P1 = P2;</code>
Comparación por Igualdad / Desigualdad	<code>P1 == P2;</code> <code>P1 != P2;</code>

No es posible realizar las siguientes operaciones:

- Sumar punteros (esta operación no tiene sentido lógico).
- Multiplicar punteros.
- Dividir punteros.
- Sumarles cantidades que no son enteras.
- Operaciones de desplazamiento.
- Enmascarar punteros (operaciones lógicas).

Ejemplo

Declaramos una clase simple:

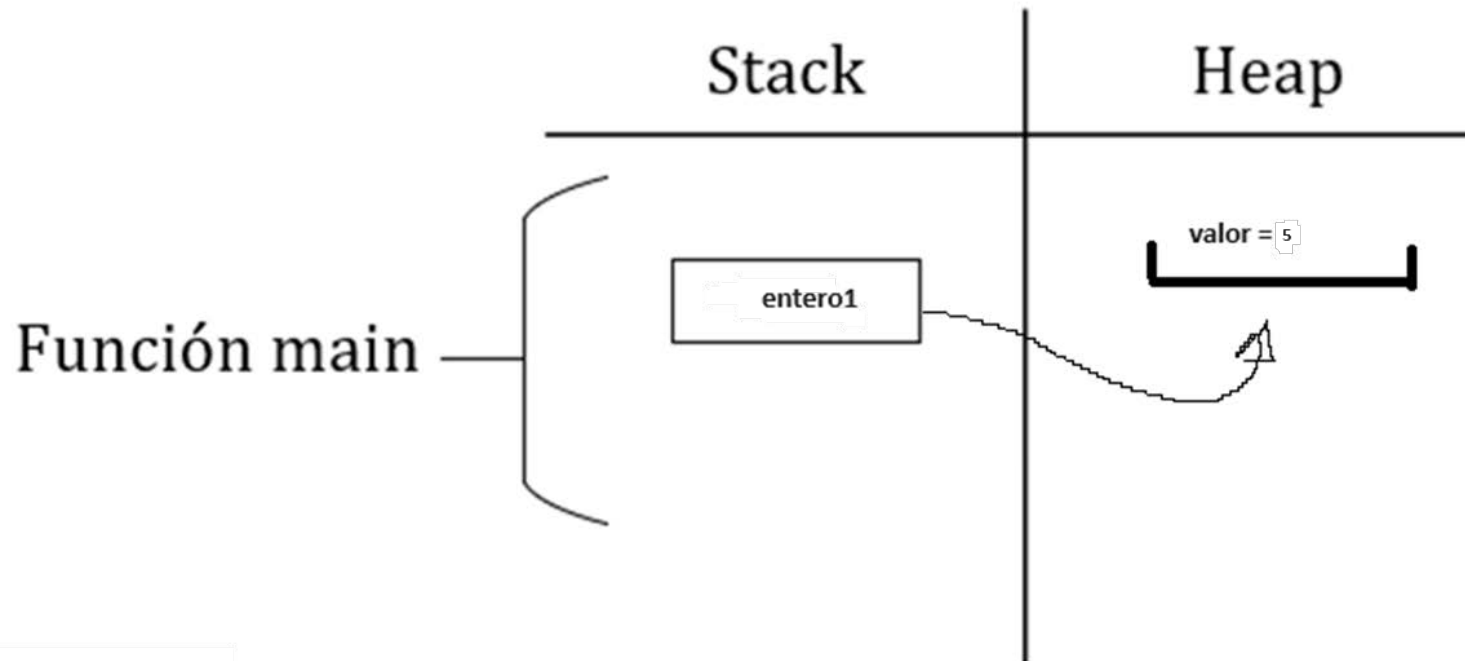
```
3 public class Entero {  
4     int valor;  
5 }  
6
```

Luego declaramos un main:

```
Entero entero1 = new Entero();  
entero1.valor = 5;  
System.out.println("El valor de entero1 es " + entero1.valor);
```

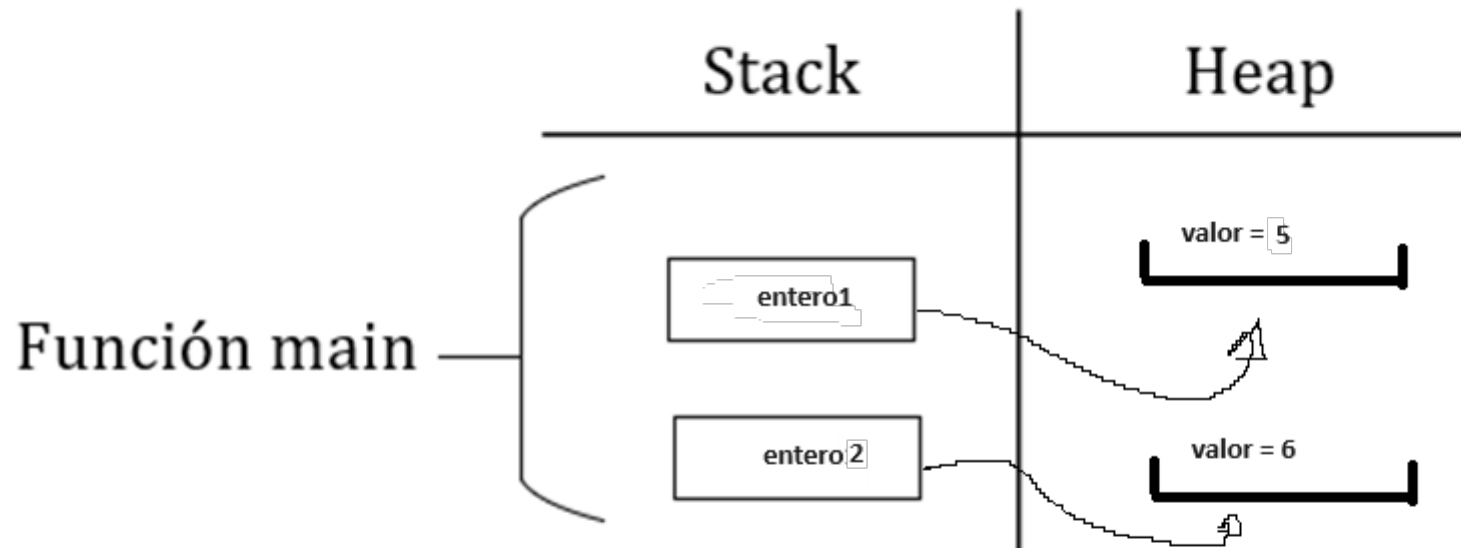
En este caso declaramos un puntero a la clase Entero y luego le asignamos un valor

Diagrama de memoria practico



```
Entero entero1 = new Entero();  
entero1.valor = 5;  
System.out.println("El valor de entero1 es " + entero1.valor);
```

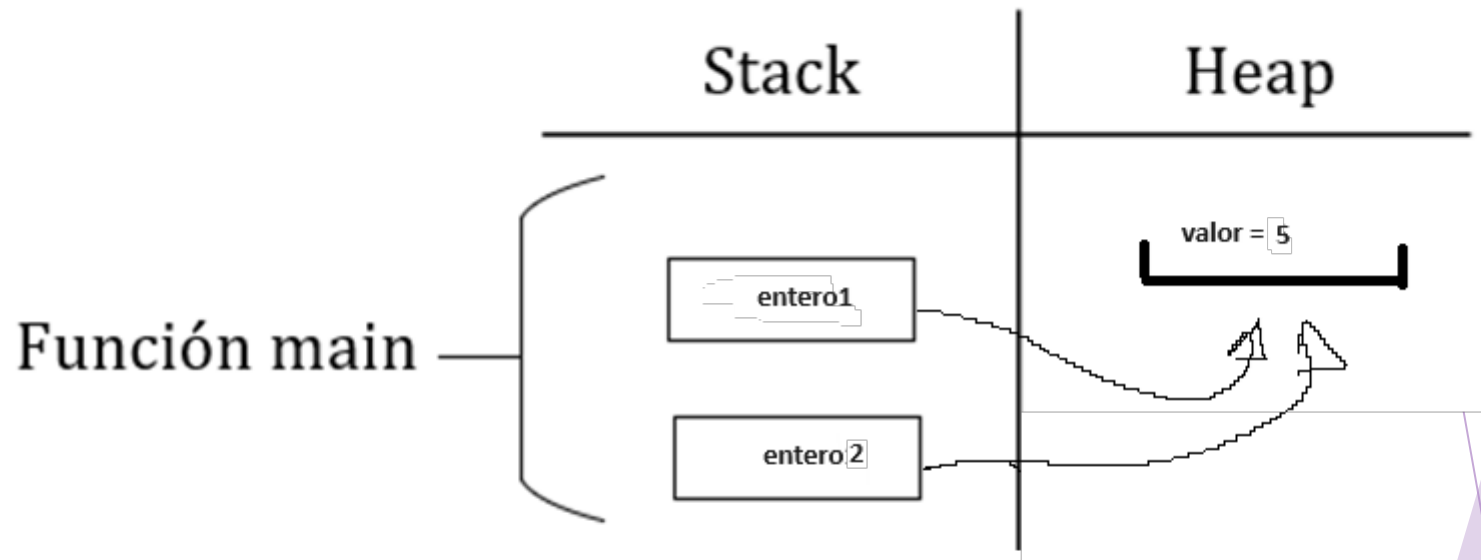
Diagrama de memoria practico



```
Entero entero1 = new Entero();  
entero1.valor = 5;  
System.out.println("El valor de entero1 es " + entero1.valor);
```

```
Entero entero2 = new Entero();  
entero2.valor = 6;  
System.out.println("El valor de entero1 es " + entero1.valor);  
System.out.println("El valor de entero2 es " + entero2.valor);
```


Diagrama de memoria practico



```
Entero entero1 = new Entero();  
entero1.valor = 5;  
System.out.println("El valor de entero1 es " + entero1.valor);
```

```
Entero entero2 = entero1;  
entero2.valor = 6;  
System.out.println("El valor de entero1 es " + entero1.valor);  
System.out.println("El valor de entero2 es " + entero2.valor);
```

Si queremos imprimir la dirección de memoria

```
Entero entero1 = new Entero();  
entero1.valor = 5;  
System.out.println("El valor de entero1 es " + entero1);
```

Y esto nos imprime

El valor de entero1 es cb100.Entero@65ae6ba4

En Java, no existe el concepto de "punteros" como en lenguajes como C o C++, por lo que no es posible hacer un cast directo de un "puntero" de byte a un "puntero" de char. Sin embargo, si lo que necesitas es convertir una matriz de byte en una matriz de char, o interpretar un byte como un char, hay varias formas de hacerlo.

Convertir un byte a char:

Si tienes un solo byte y quieres convertirlo a char, puedes hacer un cast simple, pero esto solo funciona correctamente si el byte representa un carácter ASCII (es decir, está en el rango de 0-127).

```
byte b = 65; // El valor 65 en ASCII corresponde a 'A'
char c = (char) b;
System.out.println(c); // Imprime 'A'
```


El operador new en Java se utiliza para crear nuevas instancias de objetos. Es un mecanismo clave que permite asignar memoria para un nuevo objeto en el heap y luego invocar su constructor para inicializarlo. A continuación, te explico en detalle cómo funciona el operador new en Java:

1. Asignación de Memoria en el Heap:

- I. Cuando usas new, la JVM asigna memoria en el heap para almacenar el nuevo objeto.
- II. La cantidad de memoria asignada depende del tamaño del objeto, que a su vez depende de sus atributos y su clase.
- III. Si no hay suficiente memoria disponible en el heap, se lanza un OutOfMemoryError.

```
MyClass obj = new MyClass(); // Asigna memoria para MyClass en el heap
```

2. Devolución de la Referencia:

- I. Una vez que el objeto ha sido creado e inicializado, new devuelve una referencia al objeto, que se almacena en una variable.
- II. Esta referencia apunta al lugar en el heap donde se almacena el objeto.java

Optimización (Escape Analysis):

- I. En ciertos casos, la JVM puede optimizar el uso del heap mediante técnicas como escape analysis, que pueden permitir que algunos objetos se alojen en el stack si no "escapan" del método donde se crean, aunque esto es transparente para el programador.

Resumen:

El operador new asigna memoria en el heap, llama al constructor de la clase, inicializa el objeto y devuelve una referencia a ese objeto.

Es fundamental en la gestión de memoria y en la creación de objetos en Java.

NULL

La palabra reservada null en Java representa una referencia nula o vacía, es decir, una referencia que no apunta a ningún objeto en la memoria. En otras palabras, null indica que una variable de tipo objeto (por ejemplo, una referencia a una instancia de una clase) no está vinculada a ningún objeto en particular.

Características y Uso de null:

1. Inicialización:

- I. Cuando declaras una variable de tipo objeto (como una instancia de una clase, un array, o cualquier tipo de referencia), puedes inicializarla a null si aún no tienes un objeto específico para asignarle.

```
String str = null; // str no apunta a ningún objeto  
MyClass obj = null; // obj no apunta a ninguna instancia de MyClass
```

2. Comparación con null:

1. Puedes verificar si una referencia es null utilizando el operador de comparación == o !=.

```
if (str == null) {  
    System.out.println("La variable str no está asignada a ningún objeto.");  
}
```

```
if (str != null) {  
    System.out.println("La variable str no está asignada a ningún objeto.");  
}
```

3. Dereferenciación Nula:

Si intentas acceder a un método o campo de un objeto a través de una referencia nula, la JVM lanzará una excepción `NullPointerException`.

```
String str = null;  
int length = str.length(); // Lanza NullPointerException porque str es null
```


4. Pasar null como Argumento:

Puedes pasar null como argumento a un método que espera un tipo de referencia. Esto es útil si quieres indicar que no hay un objeto disponible para ese argumento.

```
public void printString(String str) {  
    if (str == null) {  
        System.out.println("El argumento es null.");  
    } else {  
        System.out.println(str);  
    }  
}  
  
printString(null); // Imprime "El argumento es null."
```

Clases Envolventes (Wrapper Classes):

Para tipos primitivos (como int, boolean, etc.), Java proporciona clases envolventes (Integer, Boolean, etc.) que permiten representar esos tipos como objetos. Estas clases envolventes pueden tener un valor null, a diferencia de los tipos primitivos que siempre deben tener un valor.

```
Integer num = null; // Es válido, num no apunta a ningún Integer
```

Garbage Collector

El Garbage Collector (GC) en Java es un proceso automático de la Java Virtual Machine (JVM) que gestiona la memoria dinámica de un programa. Su principal función es identificar y liberar la memoria ocupada por objetos que ya no son accesibles ni utilizados por el programa, es decir, aquellos objetos que ya no tienen referencias activas y, por lo tanto, no pueden ser alcanzados en el flujo de ejecución del programa.

¿Por qué es necesario el Garbage Collector?

En lenguajes como C o C++, los programadores deben gestionar manualmente la asignación y liberación de memoria, lo que puede llevar a errores como fugas de memoria o uso de memoria no válida. En Java, el Garbage Collector se encarga de estas tareas, simplificando el proceso de gestión de memoria y reduciendo el riesgo de estos errores.

Funcionamiento del Garbage Collector:

1) Asignación de Memoria: Cuando se crea un objeto en Java utilizando el operador new, la memoria para ese objeto se asigna en el heap, una región de memoria dedicada al almacenamiento dinámico.

2) Detección de Objetos No Referenciados: El Garbage Collector busca en el heap objetos que ya no son accesibles desde ningún punto del programa. Estos son objetos a los que ninguna variable o referencia del programa puede acceder, ya que todas las referencias a ellos han sido eliminadas o reasignadas.

3) Algoritmos de Recolección de Basura:

- I. Existen varios algoritmos de recolección de basura que la JVM puede usar para identificar y recolectar los objetos no referenciados. Algunos de los más comunes son:
 - i. Mark-and-Sweep: Marca todos los objetos que son alcanzables y luego barre el heap para liberar los objetos que no están marcados.
 - ii. Generational Garbage Collection: Divide los objetos en generaciones basadas en su edad (jóvenes, adultos, viejos) y aplica la recolección de basura de manera más eficiente, ya que los objetos jóvenes son recolectados con más frecuencia que los viejos.
 - iii. Copying: Copia los objetos vivos a un nuevo espacio de memoria y luego libera todo el espacio antiguo de una vez.
 - iv. Concurrent Mark-Sweep (CMS): Un recolector de basura concurrente que intenta minimizar las pausas del programa al hacer gran parte del trabajo mientras el programa aún se está ejecutando.

4) Compactación de Memoria:

Después de liberar los objetos no utilizados, algunos recolectores de basura también compactan la memoria, es decir, reorganizan los objetos restantes en el heap para reducir la fragmentación de la memoria y hacer un uso más eficiente del espacio disponible.

5) Pausas del Garbage Collector:

Durante la recolección de basura, puede ser necesario detener temporalmente la ejecución de todos los hilos del programa para realizar ciertas tareas (como marcar objetos), lo que se conoce como Stop-the-World. Algunos recolectores de basura están diseñados para minimizar estas pausas.

Tipos de Garbage Collectors en Java:

La JVM ofrece varios tipos de Garbage Collectors que pueden ser seleccionados según las necesidades de la aplicación:

Serial Garbage Collector:

Utiliza un único hilo para la recolección de basura, adecuado para aplicaciones pequeñas.

Parallel Garbage Collector:

Utiliza múltiples hilos para la recolección, mejorando el rendimiento en aplicaciones multicore.

CMS (Concurrent Mark-Sweep) Garbage Collector:

Intenta minimizar las pausas del programa al realizar gran parte del trabajo concurrentemente.

G1 (Garbage First) Garbage Collector:

Diseñado para manejar aplicaciones grandes con baja latencia, dividiendo el heap en regiones y recolectando primero las regiones que contienen más basura.

Beneficios del Garbage Collector:

- 1) Automatización: Elimina la necesidad de gestionar manualmente la memoria, reduciendo la posibilidad de errores humanos.
- 2) Eficiencia: Optimiza el uso de la memoria y evita fugas de memoria al liberar objetos no utilizados.
- 3) Simplificación: Facilita el desarrollo de software al abstraer la complejidad de la gestión de memoria.

Desventajas o Consideraciones:

- 1) Pausas del Programa: Las pausas causadas por el Garbage Collector pueden afectar la latencia de aplicaciones en tiempo real.
- 2) Sobrecarga: Aunque automatiza la gestión de memoria, el proceso del Garbage Collector consume recursos de CPU, lo que puede impactar el rendimiento.
- 3) Configuración: Es posible que se requiera ajustar o elegir un Garbage Collector específico para optimizar el rendimiento de aplicaciones con requisitos específicos.

Resumen:

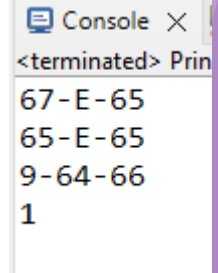
El Garbage Collector en Java es un componente crítico que gestiona automáticamente la memoria, liberando espacio ocupado por objetos que ya no son utilizados por la aplicación. Aunque simplifica el desarrollo y mejora la seguridad del manejo de memoria, también introduce ciertos desafíos, como la necesidad de manejar pausas del programa y optimizar su configuración para aplicaciones exigentes.

Indicar la salida

```
public static void main(String[] args) {
    int A, F;
    int [] vector1 = new int[3];
    int [] vector2 = vector1;
    Integer B;
    char G = 'E';
    int H = 66;
    A = 65;
    F = H;
    for(int i = 0; i < vector1.length; i++) {
        vector1[i] = A + i;
    }
    System.out.println(vector2[2] + "-" + G + "-" + A);

    B = A;
    vector2[1] = G;
    vector2[2] = (byte) G;
    System.out.println(vector1[0] + "-" + G + "-" + A);

    while (vector2[2] > 10) {
        H--;
        vector1[2] -= 30;
    }
    System.out.println(vector1[2] + "-" + H + "-" + F);
    B -= H;
    System.out.println(B);
}
```



Console X
<terminated> Prin
67-E-65
65-E-65
9-64-66
1

Fin