Asignación dinámica de memoria

2.1. Gestión Dinámica de Memoria

Los tipos de datos, tanto simples como estructurados, sirven para describir datos o estructuras de datos cuyos tamaños y formas se conocen de antemano. Sin embargo, en muchos programas es necesario que las estructuras de datos estén diseñadas de manera que su tamaño y forma varíe a lo largo de la ejecución de aquellos. Con esto se consigue, fundamentalmente, que estos programas funcionen de manera más eficiente y con un aprovechamiento óptimo de los recursos de almacenamiento en memoria principal. Las variables de todos ***los tipos de datos vistos hasta el momento son denominadas variables estáticas***:

- en el sentido en que se declaran en el programa,
- se designan por medio del identificador declarado,
- y se reserva para ellas un espacio en memoria en tiempo de compilación de los programas.

El contenido de la variable estática puede cambiar durante la ejecución del programa o subprograma donde está declarada, pero no así el tamaño en memoria reservado para ella.
Esto significa que la dimensión de las estructuras de datos a las que se refieren estas variables debe estar determinada en tiempo de compilación, lo que puede suponer una gestión ineficiente de la memoria, en el sentido de que puede implicar el desperdicio (por sobredimensionamiento) o la insuficiencia (por infradimensionamiento) de memoria.
Para esto C, como muchos los lenguajes de programación que ***ofrecen la posibilidad de crear y destruir variables en tiempo de ejecución, de manera dinámica, a medida que van siendo necesitadas durante la ejecución del programa.***
Puesto que estas variables no son declaradas explícitamente en el programa y no tienen identificador (nombre) asignado, se denominan variables anónimas (punteros).

Datos estáticos*: su tamaño y forma es constante durante la ejecución de un programa y, por tanto, se determinan en tiempo de compilación. El ejemplo típico son los arrays. Tienen el problema de que hay que dimensionar la estructura de antemano, lo que puede conllevar desperdicio o falta de memoria.*
Datos dinámicos: *su tamaño y forma es variable (o puede serlo) a lo largo de un programa, por lo que se crean y destruyen en tiempo de ejecución. Esto permite dimensionar la estructura de datos de una forma precisa: se va asignando memoria en tiempo de ejecución según se va necesitando.*

2.1.2. Asignación dinámica de memoria

Cuando se habla de asignación dinámica de memoria se hace referencia al hecho de crear variables anónimas -es decir, reservar espacio en memoria para estas variables en tiempo de ejecución del programa- así como liberar el espacio reservado para dichas variables anónimas, cuando ya no son necesarias, también durante el tiempo de ejecución.

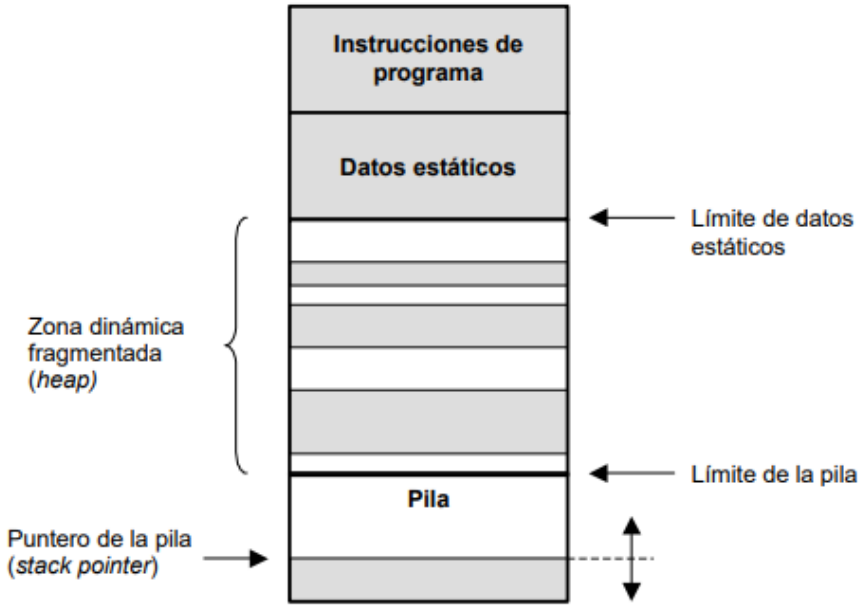


Figura 2.1. Esquema de asignación de memoria

La zona de la memoria principal del computador donde se reservan espacios para asignarlos a variables dinámicas se denomina heap o montón. Cuando el sistema operativo carga un programa para ejecutarlo y lo convierte en proceso, le asigna cuatro partes lógicas en memoria principal:

- instrucciones, datos (estáticos), pila y una zona libre.

Esta zona libre (heap) es la que va a contener los datos dinámicos. En cada instante de la ejecución del programa, el heap tendrá partes asignadas a datos dinámicos y partes libres disponibles para asignación de memoria, como puede observarse en la figura

2.1. El mecanismo de asignación-liberación

Este mecanismo de memoria durante la ejecución del programa hace que esta zona esté usualmente fragmentada (ver figura 2.1), siendo posible que se agote su capacidad si no se liberan las partes utilizadas ya inservibles. La pila también varía su tamaño dinámicamente, pero la gestiona el sistema operativo, no el programador.
Para trabajar con datos dinámicos son necesarias dos cosas:
· Subalgoritmos predefinidos en el lenguaje (pseudolenguaje) que permitan gestionar la memoria de forma dinámica (asignación y liberación).
· Algún tipo de dato con el que sea posible acceder a esos datos dinámicos (ya que con los tipos vistos hasta ahora en las asignatura sólo se puede acceder a datos con un tamaño y forma ya determinados).

2.2. Tipo Puntero

El tipo puntero y las variables declaradas de tipo puntero se comportan de manera diferente a las variables estáticas estudiadas en los temas anteriores. Hasta ahora, cuando se declaraba una variable de un determinado tipo, ésta podía contener 'directamente' un valor de dicho tipo, simplemente llevando a cabo una asignación de ese valor a la variable. Con las variables de tipo puntero esto no es así. *Las variables de tipo puntero permiten referenciar datos*

dinámicos, es decir, estructuras de datos cuyo tamaño varía en tiempo de ejecución. Para ello, es necesario diferenciar claramente entre:
· la variable referencia o apuntadora, de tipo puntero,
· y la variable anónima referenciada o apuntada, de cualquier tipo, tipo que está asociado siempre al puntero.

Físicamente, el puntero no es más que una dirección de memoria. En la figura 2.2 se muestra un ejemplo, a modo de esquema teórico, de lo que podría ser el contenido de varias posiciones de memoria principal, en la que se puede ver cómo una variable puntero, almacenado en la posición de memoria 7881(16, contiene, a su vez, otra dirección de memoria, la 78AC(16, la de la variable referenciada o anónima, que contiene el dato 6677(16.
De esta manera se ilustra cómo el puntero contiene una dirección de memoria que 'apunta' a la posición de memoria donde se almacena un dato de cierto tipo asociado al puntero

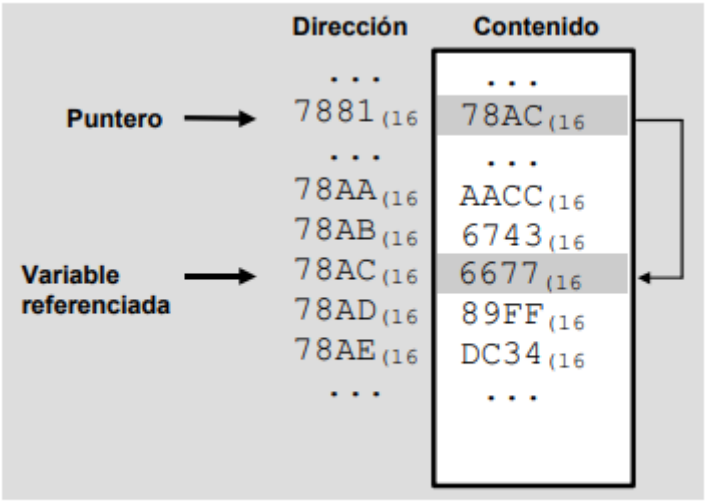


Figura 2.2. Esquema de posiciones de memoria con punteros

Definición: un puntero es una variable cuyo valor es la dirección de memoria de otra variable

Según su definición, un puntero se 'refiere' indirectamente a un valor, por lo que no hay que confundir una dirección de memoria con su contenido (ver figura 2.3).

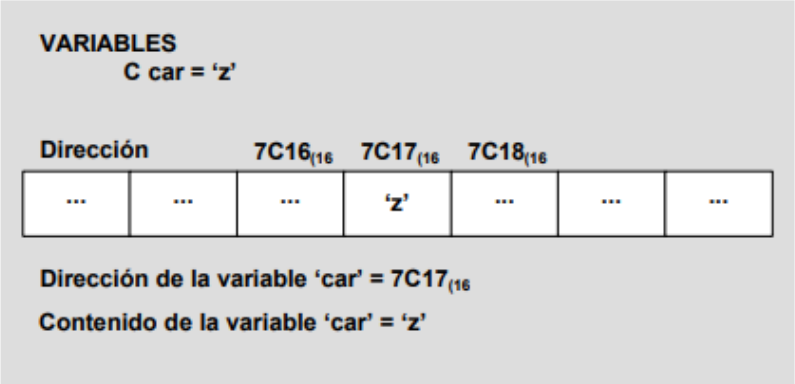
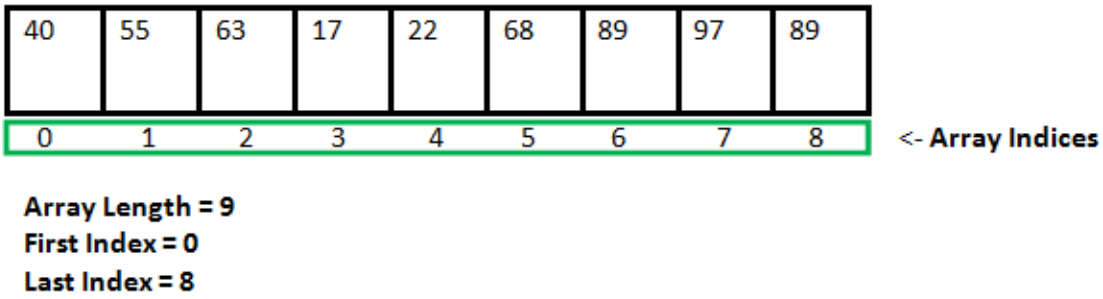


Figura 2.3. Esquema de posiciones de memoria donde se muestra la diferencia entre la dirección de una variable y su contenido

Una variable de tipo puntero no puede apuntar a cualquier variable anónima; debe apuntar a variables anónimas de un determinado tipo. El tipo de la variable anónima debe ser incluido en la especificación del tipo de la variable puntero.

Usando malloc(), calloc(), free() y realloc()

Dado que C es un lenguaje estructurado, tiene algunas reglas fijas para la programación. Uno de ellos incluye cambiar el tamaño de una matriz. Una matriz es una colección de elementos almacenados en ubicaciones de memoria contiguas (asignación estática de memoria)



Como se puede ver el tamaño de la matriz anterior es 9. Pero, ¿qué pasa si hay un requisito para cambiar esta longitud (tamaño)? Por ejemplo

- Si hay una situación en la que solo se necesitan ingresar 5 elementos en esta matriz. En este caso, los 4 índices restantes solo están desperdiciando memoria en esta matriz. Por lo tanto, existe el requisito de disminuir la longitud (tamaño) de la matriz de 9 a 5.

- Tomemos otra situación. En esto, hay una matriz de 9 elementos con los 9 índices llenos. Pero es necesario ingresar 3 elementos más en esta matriz. En este caso, se requieren 3 índices más. Por lo tanto, la longitud (tamaño) de la matriz debe cambiarse de 9 a 12.

Este procedimiento se **denomina asignación dinámica de memoria en C**. Por lo tanto, la **asignación de memoria dinámica** de C se puede definir como un procedimiento en el que el tamaño de una estructura de datos (como Array) se cambia durante el tiempo de ejecución. C proporciona algunas funciones para lograr estas tareas.

Hay 4 funciones de biblioteca proporcionadas por C definidas en el archivo de cabecera **<stdlib.h>** para facilitar la asignación de memoria dinámica en la programación C. Son los siguientes:

1. malloc()
2. calloc()
3. free()
4. realloc()

Veamos cada uno de ellos con mayor detalle.

- # malloc() método

El método "**malloc**" o "asignación de memoria" en C se utiliza para asignar dinámicamente un único bloque grande de **memoria** con el tamaño especificado. Devuelve un puntero de tipo void que se puede convertir en un puntero de cualquier forma. No inicializa la memoria en el momento de la ejecución para que haya inicializado cada bloque con el valor de basura predeterminado inicialmente.

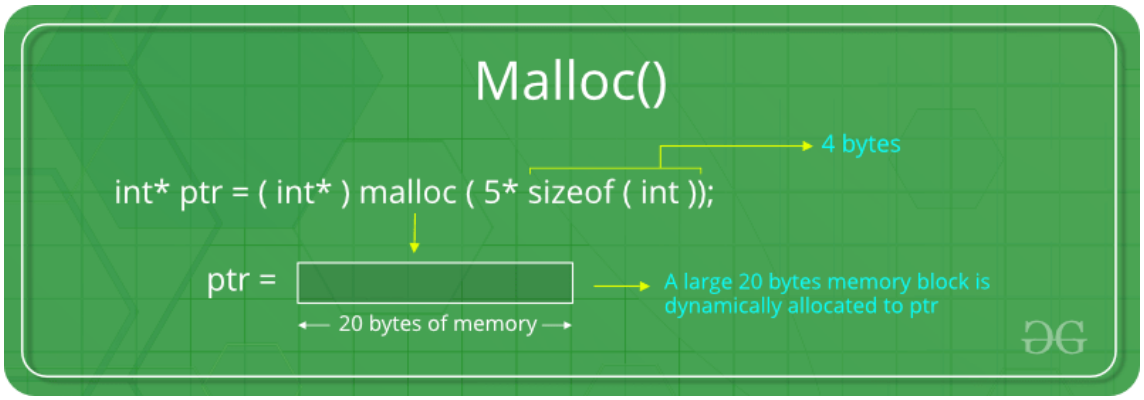
Sintaxis de malloc()

```
ptr = (casteo al tipo de dato*) malloc(tamaño en bytes)
```

Ejemplo:

ptr = (int)* ***malloc(100 * sizeof(int)); //CON CASTEO***

Dado que el tamaño del tipo int es de 4 bytes, esta instrucción asignará 400 bytes de memoria. Y, el puntero ptr contiene la dirección del primer byte en la memoria asignada.



Si el espacio en memoria es insuficiente, se produce un error en la asignación y devuelve un puntero NULL.

Ejemplo de malloc() en C // `VER proyecto en codeblock`

- # calloc()
  1. El método "**calloc**" o **"asignación contigua"** en C se utiliza para asignar dinámicamente el número especificado de bloques de memoria del tipo especificado. Es muy similar a malloc, pero tiene dos puntos diferentes y estos son:
  2. Inicializa cada bloque con un valor predeterminado '0'.
  3. Tiene dos parámetros o argumentos en comparación con malloc.

Sintaxis calloc

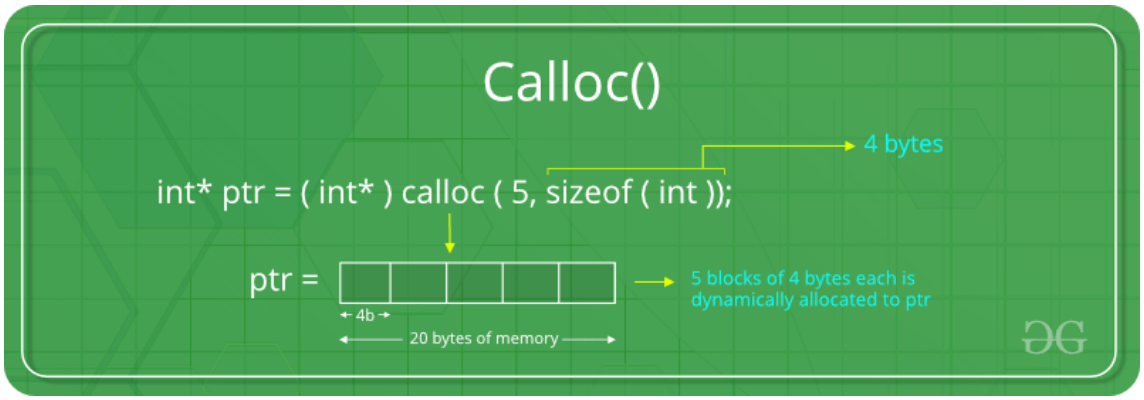
```
ptr = (casteo al tipo de dato*) calloc(n, element-size);

n es el nro de elementos y element-size es el tamaño de cada

elemento.
```

Ejemplo:

ptr = (float*) calloc(25, sizeof(float));
Esta instrucción asigna espacio contiguo en memoria para 25 elementos, cada uno con el tamaño del float.



Si el espacio es insuficiente, se produce un error en la asignación y devuelve un puntero igual a NULL.

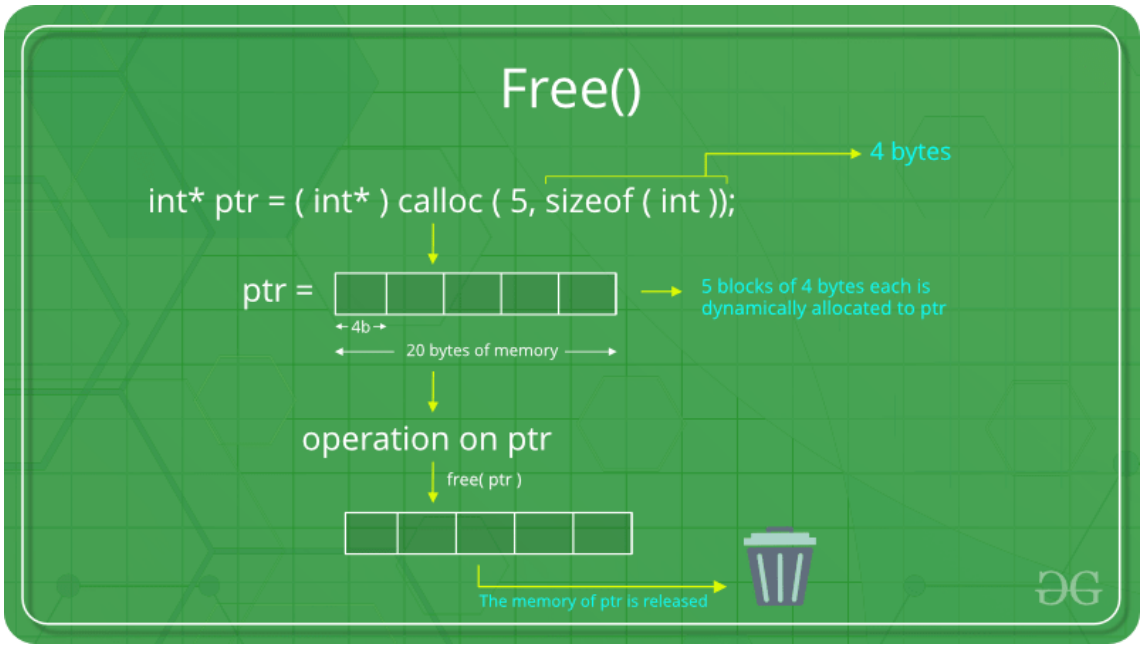
Ejemplo de calloc() en C // `VER proyecto en codeblock`

- **free()**

El método "**free**" en C se utiliza para **desasignar** dinámicamente la memoria. La memoria asignada usando las funciones malloc() y calloc() no se desasigna por sí sola. Por lo tanto, se utiliza el método free(), siempre que se realiza la asignación dinámica de memoria. Ayuda a reducir el desperdicio de memoria liberándolo.

Sintaxis de free()

```
free(ptr);
```



Ejemplo de free() en C

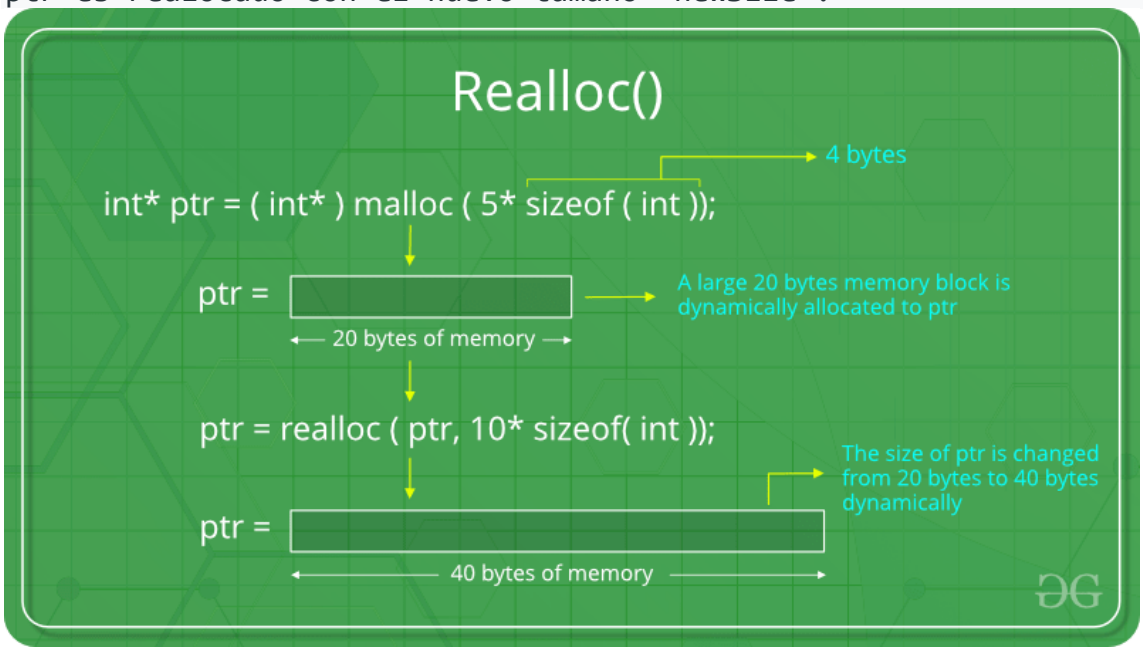
- # realloc()

El método "**realloc**" o "**reasignación**" en C se utiliza para cambiar dinámicamente la asignación de memoria de una memoria previamente asignada. En otras palabras, si la memoria previamente asignada con la ayuda de malloc o calloc es insuficiente, realloc se puede utilizar para **reasignar dinámicamente la memoria**. La reasignación de memoria mantiene el valor ya presente y los nuevos bloques se inicializarán con el valor de basura predeterminado.

Sintaxis de realloc()

```
ptr = realloc(ptr, newSize);
```

```
ptr es realocado con el nuevo tamaño 'newSize'.
```



Si el espacio es insuficiente, se produce un error en la asignación y devuelve un puntero NULL.

Ejemplo de realloc() en C // `VER proyecto en codeblock`

¡¡¡Diferencias!!!

Inicialización

malloc() asigna un bloque de memoria de un tamaño dado (en bytes) y devuelve un puntero al principio del bloque. *malloc() no inicializa la*

memoria asignada. Si intenta leer de la memoria asignada sin inicializarla primero, *invocará un comportamiento indefinido,* lo que generalmente significará que los valores que lea serán basura.
calloc() asigna la memoria y también inicializa cada byte en la memoria asignada a 0. Si intenta leer el valor de la memoria asignada sin inicializarla, obtendrá 0 ya que calloc() ya la ha inicializado en 0.

Valor de devolución

Después de una asignación exitosa en malloc() y calloc(), se devuelve un puntero al bloque de memoria, de lo contrario, se devuelve NULL, lo que indica un error.

Entonces:

	malloc()	**calloc()**
1.	Es una función que crea un bloque de memoria de un tamaño fijo.	Es una función que asigna más de un bloque de memoria a una sola variable.
2.	Solo se necesita un argumento	Se necesitan dos argumentos.
3.	Es más rápido que calloc.	Es más lento que malloc()
4.	Tiene una alta eficiencia de tiempo	Tiene baja eficiencia de tiempo
5.	Se utiliza para indicar la asignación de memoria	Se utiliza para indicar la asignación de memoria contigua
6.	Sintaxis : void* malloc(size_t size);	Sintaxis : void* calloc(size_t num, size_t tamaño);
8.	No inicializa la memoria a cero	Inicializa la memoria a cero
9.	No agrega ninguna sobrecarga de memoria adicional	Agrega algo de sobrecarga de memoria adicional