Memoria Dinámica

`malloc y free`



Supongamos que...

Queremos almacenar las **notas de un curso**, pero la cantidad de alumnos se conoce recién cuando se **ejecuta el programa**.

Hasta ahora, podemos declarar un vector con una capacidad fija:

```
int notas[100];
```

¿Qué problema se les ocurre que puede traer esto?

- Si tenemos 15 alumnos, quedan posiciones sin utilizar.
- Si tenemos 120 alumnos, el vector no alcanza.

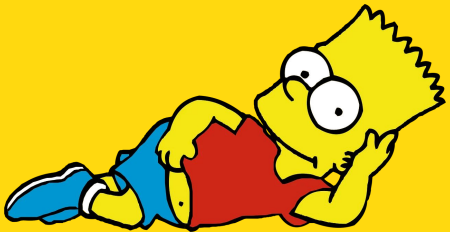
¿Cómo podemos reservar espacio para la cantidad de elementos que necesitamos?

Memoria Dinámica

La **memoria dinámica** permite reservar memoria durante la ejecución del programa.

Estructura de Memoria (Heap)





Características

- Podemos decidir cuánto espacio necesitamos a partir de un dato ingresado por el usuario.
- La reserva se realiza en una región de memoria llamada **heap**.
- El programador debe **liberar** esa memoria cuando deja de necesitarla.

// Ejemplo

Si el usuario ingresa **15 alumnos**, reservamos espacio exactamente para **15 notas**.

¿Qué son malloc y free?

Son funciones de la biblioteca estándar de C que permiten administrar memoria dinámica.

`malloc()`

Reserva un bloque de memoria y devuelve su dirección.

`free()`

Nos permite devolver esa memoria cuando ya no la usamos.

Para trabajar con memoria dinámica:

1. Reservamos memoria.
   malloc()
2. Comprobamos que la reserva haya funcionado.
3. Utilizamos la memoria.
4. La liberamos.
   free()

Biblioteca <stdlib.h>

Para utilizar **`malloc()`** y **`free()`**, debemos incluir la biblioteca estándar correspondiente.

Cada encabezado declara las funciones que vamos a utilizar dentro de nuestro programa.

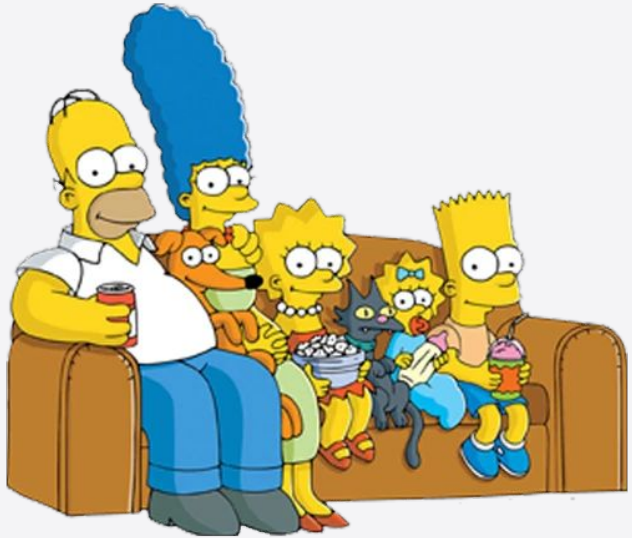
```c
// Inclusión de bibliotecas en C

#include <stdlib.h> // malloc, free
#include <stdio.h> // printf, scanf

malloc( ... );
free( ... );
```

<stdlib.h>

Contiene las funciones para la **gestión de memoria dinámica**. Declara también otras funciones avanzadas como `calloc()` y `realloc()`.



malloc(): memory allocation

- Recibe la **cantidad de bytes** que queremos reservar.
- Si la reserva funciona, devuelve un puntero al comienzo del bloque.
- Si falla, devuelve **`NULL`**.
- La memoria reservada **no está inicializada**.

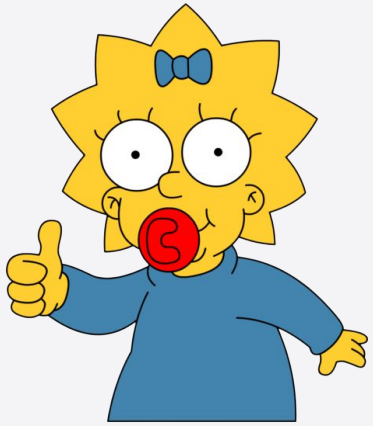
```c
#include <stdlib.h>
void *malloc(size_t size);
```

Para reservar espacio para un entero:

```c
int *numero = malloc(sizeof(int));
```

`sizeof(int)` indica cuántos bytes ocupa un entero.

No debemos asumir que un `int` siempre ocupa 4 bytes.



Entonces...
¿Solo con usar malloc() alcanza?



NOOO!

malloc() es una función que puede fallar
SIEMPRE que la usemos hay que validar que haya funcionado correctamente!!



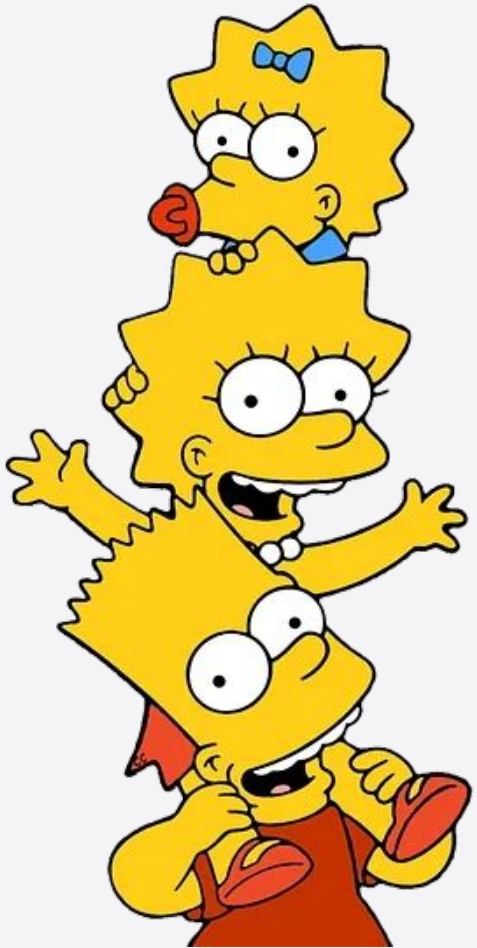
¿Cómo manejamos el error?

malloc() cuando falla nos devuelve NULL.
De esta manera podemos hacer:

```
int* puntero = malloc(sizeof(int));
if (puntero == NULL){

      printf("Error al reservar memoria…");

      return -1;
}
```



Ejemplo: Reservar un Entero

```c
// Ejemplo completo en C
int *numero = malloc(sizeof(int));
if (numero == NULL) {
      printf("No se pudo reservar ... \n");
      return ERROR;
}
*numero = 25;
printf("Valor: %d\n", *numero);
free(numero);
numero = NULL;
```

Explicación paso a paso

- `numero` guarda la dirección de la memoria reservada en el heap.
- `*numero` permite acceder y modificar el entero almacenado.
- `free(numero)` libera la reserva para evitar fugas de memoria.

Regla de Oro

Primero **comprobamos el puntero** (que no sea NULL) y solo después accedemos a la memoria.

Ejemplo: Reservar un vector

Para reservar un vector, multiplicamos:

Cantidad de elementos × tamaño de cada elemento

```c
// Reserva de un vector de 7 enteros
int cantidad = 7;

int *notas = malloc(cantidad * sizeof(int));

// Acceso a los elementos
notas[0] = 8; notas[1] = 6;
```

- Reservamos un **bloque contiguo** con espacio para 7 enteros.
- Los índices válidos van desde `0` hasta `cantidad - 1` (0 a 6).

Representación en memoria (Heap)



Detalles del bloque reservado:

- Total memoria: `7 * sizeof(int)` bytes.
- Acceso contiguo y directo por índice.
- Los valores no inicializados contienen basura de memoria (`?`).



¿Qué pasa si solo uso malloc()?

Si solo hago malloc() , entonces queda memoria reservada en el heap que nadie libera.

Entonces... ¿Qué hago ahora con la memoria que reserve?



free()

`free()` libera un bloque obtenido mediante una función de reserva dinámica.

- Debemos pasar la dirección inicial del bloque.
- Cada reserva debe liberarse **una sola vez**.
- Después de liberarla, **no** podemos seguir **accediendo** a ella.
- `free(NULL)` es válido y no realiza ninguna acción.

```
free(puntero);
```



Ejemplo: Uso y liberación del vector

```c
// Continuación del ejemplo anterior
for (int i = 0; i < cantidad; i++) {
    notas[i] = 0;
}
notas[0] = 8;
printf("Primera nota: %d\n", notas[0]);
free(notas);
notas = NULL;
```



Explicación y Detalles

- Inicializamos los elementos antes de leerlos.
- Usamos el vector con índices, como siempre.
- Una sola llamada a `free(notas)` libera todo el bloque.

¡Atención!

`malloc()` no guarda la cantidad de elementos para nosotros: **debemos conservarla en otra variable**.

El puntero y la memoria reservada

```
// Siguiendo con el ejemplo…
int cantidad = 7;
int *notas = malloc(cantidad * sizeof(int));
```

Diferencia clave

El puntero y el bloque al que apunta son elementos distintos.

Al finalizar la función, las variables locales (`cantidad` y `notas`) **dejan de existir** en el Stack.

El bloque del Heap **sigue reservado** hasta que lo liberamos explícitamente con `free()`.

Ubicación y Contenido en Memoria

Elemento	Ubicación	Qué contiene
`cantidad`	Stack	El número 7
`notas`	Stack	La dirección del bloque
`Bloque reservado`	Heap	Espacio para 7 enteros

Resumen:

- `cantidad`: Variable **local** con valor primitivo.
- `notas`: Puntero **local** que almacena dirección.
- `Heap`: Memoria **dinámica** independiente del scope.

Buenas prácticas

```c
// Patrón de reserva y liberación segura
int *v = malloc(n * sizeof(int));
if (v == NULL) return ERROR;
free(v);
v = NULL;
```

Regla de Oro

"Reservar memoria también implica hacerse responsable de liberarla."

Cada llamada a `malloc()` debe tener su correspondiente liberación con `free()` cuando la memoria ya no sea necesaria.

Checklist de Buenas Prácticas

- **Calcular el tamaño:** Usar siempre `sizeof()` para garantizar portabilidad.
- **Validar el puntero:** Comprobar el resultado de `malloc()` antes de acceder al bloque.
- **Inicializar datos:** Asignar valores iniciales antes de leer la memoria reservada.
- **Respetar límites:** Evitar accesos fuera de los límites del vector reservado.
- **Liberación oportuna:** Liberar cuando deje de usarse, incluyendo caminos de error.
- **Evitar fugas:** No perder el último puntero a una reserva sin haberla liberado.
- **Acceso seguro:** No acceder a memoria liberada ni liberar un bloque dos veces.

Stack vs. Heap

Stack: Diseñado para **variables locales y automáticas**. Es rápido, pero su tiempo de vida está estrictamente limitado al bloque de ejecución.

Heap: Espacio para **memoria dinámica**. Ofrece flexibilidad total en tamaño y persistencia, requiriendo gestión manual explícita.

Diferencia Crítica: Olvidar liberar la memoria en el Heap causa **fugas de memoria**, mientras que el Stack se limpia solo de forma automática.

Criterio de Elección

La elección depende del **tamaño de los datos** y de **cuánto tiempo necesitamos conservarlos** durante la ejecución del programa.

Aspecto	Stack	Heap
Reserva	Automática al ejecutar	Explícita (ej. malloc)
Liberación	Automática al salir	Explícita (mediante free)
Vida	Limitado al bloque	Hasta ser liberado
Tamaño	Fijo	Dinámico / Variable
Ventaja	Sencillo y automático	Flexibilidad total
Riesgos	Overflow / Punteros inválidos	Fugas / Doble free
Uso típico	Variables temporales	Datos persistentes/grandes

Preguntas?

