Clase completa: `realloc()`, `void *` y Casteo en C

Materia: Fundamentos de Programación – FIUBA
Nivel: Primer año – Ingeniería en Informática
Duración sugerida: 2 módulos (teoría + práctica)

🧭 Índice

---

1. Introducción

En esta clase vamos a ver cómo trabajar con memoria dinámica de forma segura y flexible utilizando tres conceptos clave del lenguaje C:

- El puntero `void *`, que **permite representar cualquier tipo de puntero sin información de tipo**.
- El mecanismo de *casteo*, que nos permite interpretar memoria genérica (`void *`) como tipos específicos.
- La función `realloc()`, que *permite modificar el tamaño de bloques de memoria dinámica previamente reservados*.

---

2. El puntero `void *`: definición y motivación

- Un puntero `void *` es un **puntero genérico** en C.
- Su principal característica es que **puede almacenar la dirección de cualquier tipo de dato**,
- pero **no puede ser desreferenciado directamente**.

Esto lo hace útil para funciones que deben ser genéricas, como:

```c
void print_pointer(void *ptr, char type);
```

```c
#include <stdio.h>

void print_value(void *data, char type) {
    switch (type) {
        case 'i':
            printf("Entero: %d\n", *((int *)data));
            break;
        case 'f':
            printf("Flotante: %.2f\n", *((float *)data));
            break;
        case 'c':
            printf("Caracter: %c\n", *((char *)data));
            break;
        default:
            printf("Tipo desconocido\n");
    }
}

int main() {
    int entero = 42;
    float flotante = 3.14;
    char caracter = 'A';

    print_value(&entero, 'i');
    print_value(&flotante, 'f');
    print_value(&caracter, 'c');

    return 0;
}
```

---

3. Casteo en C: teoría, sintaxis y uso

Ejemplo 1: leer un `int`

```c
int x = 10;
void *p = &x;
printf("%d\n", *((int *)p)); // imprime 10
```

```
x = 10
+--------+
|   10   |
+--------+
   ^
```

```
   |
p guarda &x

(int *)p → puntero a int

*((int *)p) → valor 10
```

Ejemplo 2: modificar un `float`

```c
float f = 3.14;
void *p = &f;
*((float *)p) = 2.71;
```

```
f = 3.14
+---------+
|  3.14   |
+---------+
   ^
   |
(float *)p → puntero a float
```

Ejemplo 3: acceder a arreglo dinámico

```c
int *arr = malloc(3 * sizeof(int));
arr[0] = 10; arr[1] = 20; arr[2] = 30;
void *v = arr;
printf("%d\n", *((int *)v + 1); // imprime 20
```

```
arr / v →
+------+------+------+
|  10  |  20  |  30  |
+------+------+------+
          ^
     ((int *)v)[1]
```

```c
((int *)v)[1]       // forma de arreglo
*((int *)v + 1)     // forma de puntero
```

---

4. `realloc()` – Manual Linux (Sección 3) traducido

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

> La función `realloc()` cambia el tamaño del bloque de memoria apuntado por `ptr` a `size` bytes.
> El contenido permanecerá sin cambios en el rango desde el inicio de la región hasta el mínimo entre el tamaño viejo y el nuevo.
> Si el nuevo tamaño es mayor, la memoria adicional no se inicializa.
> Si `ptr` es `NULL`, la llamada es equivalente a `malloc(size)`.
> Si `size` es cero y `ptr` no es `NULL`, la llamada es equivalente a `free(ptr)`.
> Si la memoria es movida, se libera el bloque anterior. Si falla, el bloque original queda intacto.

---

5. Ejemplo básico de `realloc()`

```c
int *arr = malloc(2 * sizeof(int));
arr[0] = 1; arr[1] = 2;

arr = realloc(arr, 4 * sizeof(int));
arr[2] = 3; arr[3] = 4;

for (int i = 0; i < 4; i++)
    printf("%d ", arr[i]);
free(arr);
```

```
arr →
+-----+-----+
|  1  |  2  |
+-----+-----+

Después del realloc:

arr →
+-----+-----+-----+-----+
|  1  |  2  |  3  |  4  |
+-----+-----+-----+-----+
```

Ejemplo 6: `realloc()` + `void *`

```c
int *arr = malloc(2 * sizeof(int));
arr[0] = 1; arr[1] = 2;
```

```
void *v = arr;
int *tmp = realloc(v, 4 * sizeof(int));
tmp[2] = 3; tmp[3] = 4;
```

```
Antes:
v →
+------+------+
|  1   |  2   |
+------+------+

Después:
tmp →
+------+------+------+------+
|  1   |  2   |  3   |  4   |
+------+------+------+------+
```

7. Ejemplo intermedio: crecimiento dinámico

```
int *nums = NULL;
int n = 0;

for (int i = 0; i < 5; i++) {
    int *tmp = realloc(nums, (n + 1) * sizeof(int));
    if (!tmp) {
        free(nums);
        return 1;
    }
    nums = tmp;
    nums[n++] = i * 10;
}

for (int i = 0; i < n; i++)
    printf("%d ", nums[i]);
free(nums);
```

```
nums →
+------+
|  0   |
+------+

Iteración 2:

nums →
+------+------+
|  0   | 10   |
+------+------+
```

```
Iteración 3:

nums →
+------+------+------+
|  0   | 10   | 20   |
+------+------+------+

Iteración 4:

nums →
+------+------+------+------+
|  0   | 10   | 20   | 30   |
+------+------+------+------+

Iteración 5:

nums →
+------+------+------+------+------+
|  0   | 10   | 20   | 30   | 40   |
+------+------+------+------+------+
```

---

8. Ejemplo intermedio: decrecimiento dinámico

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr = malloc(5 * sizeof(int));
    if (!arr) return 1;

    for (int i = 0; i < 5; i++)
        arr[i] = (i + 1) * 10;

    printf("Antes de reducir:\n");
    for (int i = 0; i < 5; i++)
        printf("%d ", arr[i]);
    printf("\n");

    // Reducimos el bloque a solo 3 elementos
    arr = realloc(arr, 3 * sizeof(int));

    printf("Después de reducir:\n");
    for (int i = 0; i < 3; i++)
        printf("%d ", arr[i]);
    printf("\n");

    free(arr);
    return 0;
}
```

```
Antes del realloc() (5 enteros):

arr →
+-----+-----+-----+-----+-----+
| 10  | 20  | 30  | 40  | 50  |
+-----+-----+-----+-----+-----+

Después del realloc() (3 enteros):

arr →
+-----+-----+-----+
| 10  | 20  | 30  |
+-----+-----+-----+

Los valores 40 y 50 ya no están accesibles y se consideran descartados por
el programa.
```

8. Buenas prácticas

- Siempre verificar el retorno de `realloc()`.
- Usar una variable temporal (`tmp`) antes de reemplazar el puntero original.
- Liberar la memoria con `free()` al finalizar.
- No acceder a `void *` sin castearlo.
- Inicializar el puntero como `NULL` si se va a crecer dinámicamente.

---

9. Conclusiones

- `void *` permite escribir código genérico, pero requiere casteo.
- `realloc()` es fundamental para estructuras dinámicas como vectores o buffers.
- El manejo correcto de memoria dinámica es clave para evitar errores y fugas.
