Capítulo 12 — Arenas (asignadores de región)

> *Capítulo avanzado. Se apoya en lo aprendido sobre punteros (`08`), `struct` (`09`) y memoria dinámica (`09-memoria-dinamica`). Leelo después de tener claros `malloc`, `free` y las operaciones con punteros.*

12.1 El problema: `malloc` para todo, `free` para todo

Cuando recién empezás, la memoria dinámica parece simple:

```
Node *n = malloc(sizeof(Node));   // reservo
/* ... uso ... */
free(n);                          // libero
```

Ese par `malloc` / `free` funciona bien para **un** objeto. El problema aparece cuando tu programa necesita reservar **muchísimas cosas pequeñas**: nodos de un árbol, tokens de un parser, cadenas temporales, estructuras de un request HTTP, filas de una tabla, etc.

Ahí empezás a ver estos dolores:

- **Seguís el rastro de cada `free`.** Si te olvidás de uno, tenés una *fuga de memoria* (memory leak).
- **Liberás dos veces** lo mismo y el programa explota (*double free*).
- **Usás memoria después de liberarla** (*use-after-free*): el peor bug, porque a veces "funciona".
- **Reservar mil veces** un pedacito tiene costo: cada `malloc` lleva *overhead* interno.
- **Manejar un error a mitad de camino** (por ejemplo, falló el `malloc` número 400 de 1000) te obliga a liberar todo lo anterior. Código feo y frágil.

La pregunta clave es: **¿y si en lugar de reservar y liberar objeto por objeto, reservo un bloque grande y me olvido de liberar cada cosa individualmente?**

12.2 ¿Qué es una arena?

Una **arena** (también llamada *arena allocator*, *region allocator* o *bump allocator*) es una estrategia de manejo de memoria donde:

1. Reservás **un único bloque grande** de memoria de una sola vez (por ejemplo, 1 MiB).
2. Repartís ese bloque **de a poco**, avanzando un *cursor* cada vez que alguien pide memoria.
3. **No liberás objetos individuales.** Liberás **todo junto** al final, de una sola vez.
4. Para "empezar de cero" sin devolverle la memoria al sistema, **rebobinás el cursor** al inicio.

La analogía más simple: pensá en una **hoja de papel rayada**.

- La hoja entera es el bloque que reservaste.
- El renglón donde vas escribiendo es el **cursor/offset** (`offset`).
- Cada vez que escribís algo, bajás al renglón siguiente: **nunca reusás un renglón ya escrito**.
- Cuando querés empezar de nuevo, **raspás toda la hoja** (reiniciás el cursor a 0). No vas borrando palabra por palabra.
- Al final, **tirás la hoja a la basura** (un solo `free`).

Esa es toda la idea. Un cursor que solo avanza. Por eso se le dice *bump allocator*: "bump" = golpear el cursor hacia adelante.

Ventajas

- **Rapidísimo**: asignar es sumar al offset. Sin listas internas, sin buscar huecos.
- **Imposible olvidarse de un `free` individual**: hay uno solo, el de la arena completa.
- **Sin fragmentación visible**: al liberar todo junto, no quedan "agujeros" sueltos.
- **Manejo de errores simple**: si algo falla a mitad de camino, tirás la arena entera.

Desventajas / cuándo NO usarla

- **No podés liberar un objeto puntual.** Si necesitás liberar *una cosa* y no todo, la arena no es la herramienta (o necesitás una arena con "checkpoints", ver 12.5).
- **La memoria crece hasta el pico de uso.** El offset solo sube; no se recupera hasta el reset.
- **No sirve para estructuras de vida larga y tamaño impredecible** que necesitan liberarse por partes (por ejemplo, una lista donde vas borrando el medio continuamente).

El caso de uso ideal

Cuando tenés **muchos objetos pequeños con la misma vida útil** y los liberás **todos juntos**:

- Parsear un documento: todos los tokens viven hasta que terminás de procesarlo.
- Procesar un request: toda la memoria temporal muere cuando respondés.
- Compilar / analizar texto: reservás muchísimo y tirás todo al final.
- Un juego: cada "nivel" o "frame" reserva y libera su propia arena.

12.3 Un primer ejemplo mínimo (comentado)

> *Este es **material didáctico para entender el concepto**. Los ejercicios del 12.8 te van a pedir que lo **extiendas** con features que acá no están. No es la solución de ningún ejercicio.*

La estructura central tiene solo **tres campos**:

```c
#include <stddef.h>   // size_t

typedef struct {
    unsigned char *buffer;  // el bloque grande (donde vive toda la memoria)
    size_t         capacity; // cuántos bytes entran en total
    size_t         offset;   // cuántos bytes ya se usaron (el cursor)
} Arena;
```

- `buffer` es el `malloc` grande. Es `unsigned char *` porque la arena es "bytes crudos": ella no sabe ni le importa qué tipo vas a guardar.
- `capacity` es el tamaño total. Sirve para saber cuándo **no hay más lugar**.
- `offset` es el cursor. Empieza en 0. Solo avanza.

> ***¿Una arena guarda un solo tipo o varios?*** *Una arena **no tiene tipo**: guarda bytes. El tipo no vive en la arena, sino en **cómo vos interpretás** el puntero que te devolvió `arena_alloc`. Por eso una **misma** arena puede contener, al mismo tiempo, tipos distintos — `int`, `double`, structs, cadenas—, cada uno en su propio hueco, siempre que respetes la **alineación** (ver 12.4.1). El ejemplo de la `main` en 12.3 ya lo hace: pide un `Point` y, del **mismo** bloque, un arreglo de 100 `int`. Lo único que la arena **no** hace es acordarse por vos a qué tipo corresponde cada puntero: eso queda de tu lado.*

Crear y destruir la arena

```c
#include <stdlib.h>  // malloc, free

// Reserva el bloque grande. Devuelve una Arena lista para usar.
Arena arena_create(size_t capacity) {
    Arena a;
    a.buffer   = (unsigned char *)malloc(capacity);
    a.capacity = capacity;
    a.offset   = 0;                     // cursor al principio
    return a;
}

// Libera TODO. Un solo free para toda la arena.
void arena_destroy(Arena *a) {
    free(a->buffer);      // chau bloque entero
    a->buffer   = NULL;
    a->capacity = 0;
    a->offset   = 0;
}
```

Fijate algo importante: `arena_create` devuelve una `Arena` **por valor**. La estructura es chica (3 campos), copiarla es barato, y así evitás que el llamador maneje memoria para la propia arena.

Repartir memoria: la única operación de asignación

```c
// Pide `size` bytes a la arena. Devuelve un puntero a memoria SIN inicializar,
// o NULL si no queda espacio. NUNCA necesita saber cuándo liberar.
void *arena_alloc(Arena *a, size_t size) {
    // 1) Si la arena no queda con lugar suficiente, no hay nada que hacer.
    if (a->offset + size > a->capacity) {
        return NULL;  // "arena llena"
    }

    // 2) El punto de reparto es el cursor actual.
    void *ptr = a->buffer + a->offset;

    // 3) El cursor avanza. Esto es TODO lo que cuesta asignar: una suma.
    a->offset += size;

    return ptr;
}
```

Esa suma es el corazón de la arena. `return a->buffer + a->offset;` es literalmente "te doy el renglón donde está el cursor" y `a->offset += size;` es "bajo al renglón siguiente".

Mirá que **nunca hay un `free` por objeto**. No existe `arena_free(ptr)` porque no hace falta: cuando el llamador termine, llama a `arena_reset` o a `arena_destroy`.

> ***Reservar es solo el primer paso.*** *El puntero que devuelve `arena_alloc` es una dirección común y corriente: guardás y recuperás datos por él (`*p = ...` , `x = *p`), igual que con cualquier otro puntero. La arena no es una caja fuerte ni un tipo especial: es memoria normal que se libera en bloque. Por eso, en los ejercicios no alcanza con pedir bytes — hay que **escribir** en ese espacio y **leerlo de vuelta** para comprobar que sirve.*

Reiniciar sin liberar: reaprovechar el bloque

```c
// Vuelve el cursor a cero. La memoria NO se devuelve al sistema,
// pero podés volver a repartirla como si estuviera vacía.
void arena_reset(Arena *a) {
    a->offset = 0;
}
```

Esto es oro puro para bucles donde, en cada iteración, reservás un montón y al final ya no te importa: en vez de reservar y liberar 1000 veces, reseteás la arena y listo.

Cómo se usa todo junto (ejemplo de uso comentado)

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

typedef struct { int x, y; } Point;

int main(void) {
    // Una sola reserva grande para todo el programa.
    Arena arena = arena_create(1024 * 1024); // 1 MiB

    // Reservo un Point (nótese: uso sizeof del TIPO, no cuántos bytes "a mano").
    Point *p = (Point *)arena_alloc(&arena, sizeof(Point));
    p->x = 3;
    p->y = 4;

    // Reservo un arreglo de 100 enteros.
    int *numbers = (int *)arena_alloc(&arena, 100 * sizeof(int));
    for (int i = 0; i < 100; i++) {
        numbers[i] = i * i;
    }

    printf("p = (%d, %d), numbers[42] = %d\n", p->x, p->y, numbers[42]);

    // NADA de free(p), free(numbers): se liberan TODOS con la arena.
    arena_destroy(&arena);

    // Truco típico: reusar el mismo bloque en un bucle.
    // for (int frame = 0; frame < 60; frame++) {
    //     ... reservar todo lo del frame ...
    //     render();
    //     arena_reset(&arena);   // listo para el próximo frame, sin malloc/free
    // }
    return 0;
}
```

Ese es el mínimo viable. Con esto ya podés resolver cosas reales. Lo que sigue son los detalles que lo vuelven "de producción".

12.4 Dos detalles que sí importan

12.4.1 Alineación de memoria

Por qué a la CPU le importa dónde empieza un dato

Los procesadores no leen cualquier dirección igual de rápido, y algunos **exigen** que ciertos tipos empiecen en direcciones "redondas". Si repartís tamaños impares (3 bytes, 5, 1…) el próximo dato puede quedar **desalineado** y provocar lentitud o, en arquitecturas estrictas, un *crash* (comportamiento indefinido).

Pero antes de seguir hay que desarmar una trampa de vocabulario, porque es la raíz de tu duda:

> ***"Múltiplo de su tamaño" es un atajo que solo vale para los primitivos.*** *La regla de verdad es: cada tipo exige que su dirección sea* ***múltiplo de su ALINEACIÓN****, y la alineación* ***no siempre coincide con el tamaño****. Para* `int`*,* `double` *y* `char` *da igual; para un* `struct`*, casi nunca. Eso es exactamente lo que hace que te preguntes si podés mezclar tipos. Podés. Seguí leyendo.*

La alineación es una propiedad del TIPO, no del dato ni de la arena

Una arena **no tiene tipo**: guarda bytes y no sabe qué vas a poner adentro. La alineación no es una propiedad del valor ni de la arena: es una propiedad **del tipo** que vos vas a escribir en ese hueco. El lenguaje ya sabe, para cada tipo, cuánto exige:

Tipo	Tamaño	Alineación típica (x86-64)
`char`	1	1
`short`	2	2
`int`	4	4
`long`, `double`, puntero	8	8
`struct`	según sus miembros (con relleno)	**= la mayor alineación de sus miembros**

La idea que te salva: UNA alineación para todos

Acá está la respuesta a tu primera pregunta. ¿Podés tener `int`, `char`, `double` y structs complejos en la **misma** arena, bien alineados? **Sí, y no hace falta calcular nada por tipo.**

La clave: si **cada** bloque arranca en un múltiplo de 8, entonces arranca también en un múltiplo de 4, de 2 y de 1. Un múltiplo de 8 *es* múltiplo de todos los divisores de 8. Así que alineando **todo** a 8 cubrís a la vez a `char` (1), `int` (4), `double` /puntero (8) y a cualquier struct cuya alineación sea ≤ 8:

```
base+0      base+8      base+16     base+24
  |           |           |           |
  v           v           v           v
[ double  ][ int     ][ char    ][ struct  ... ]
  ^8          ^8          ^8          ^8
  todos múltiplos de 8 -> 8, 4, 2 y 1 quedan satisfechos
```

Por eso el ejemplo de `main` en 12.3 puede pedir un `Point` y después 100 `int` del mismo bloque: no calculás la posición de nadie, **alineás cada reparto al mismo valor** y listo.

> ***¿Qué alineación universal uso?*** `8` *alcanza para* `int`, `double` *y punteros. Para estar cubierto ante cualquier tipo (incluido algún SIMD o* `long double`*), usá* `ARENA_ALIGN = _Alignof(max_align_t)` *(normalmente 16). Es el valor que el estándar considera suficiente para todo. Un poquito más de desperdicio, cero dolores de cabeza.*

Structs: por qué el tamaño NO te dice la alineación

Esta es la parte que confunde. Mirá:

```c
struct Mixed { char tag; int value; };   // sizeof = 8   | _Alignof = 4
struct Pair  { double a; double b; };    // sizeof = 16  | _Alignof = 8
struct Quad  { int a, b, c, d; };        // sizeof = 16  | _Alignof = 4
```

¿Por qué `Mixed` mide 8 y no 5? Porque el compilador **inserta 3 bytes de relleno** (*padding*) después de `tag`, para que el `int` caiga alineado a 4. Y mide 8 (múltiplo de 4) para que, en un arreglo `Mixed items[10]`, el `value` del segundo elemento también quede alineado.

De ahí sale el dato que te ahorra todo el cálculo:

> ***La alineación de un tipo SIEMPRE divide a su tamaño.*** *Es una garantía del estándar de C, y existe precisamente para que los arreglos funcionen. Consecuencia directa: si alineás el bloque a la alineación del struct,* ***todos*** *sus miembros caen alineados solos. Nunca tenés que ubicar* `tag` *ni* `value` *a mano.*

Entonces, tu pregunta "¿cómo calculo, en base al `sizeof`, la posición que debe ocupar un struct complejo?" tiene una respuesta incómoda y liberadora: **no lo calcules desde el `sizeof`.** El `sizeof` te dice *cuánto ocupa*, no *dónde tiene que empezar*. Para saber eso, pedile la alineación al lenguaje:

- `_Alignof(Type)` (C11) te da la alineación en bytes.
- `alignof(Type)` con `<stdalign.h>` (el alias con nombre lindo).
- `_Alignas(16)` en una variable te fuerza *su* alineación.

> ***Cuidado con el `sizeof` como fuente de alineación.*** `struct Pair` *(size 16) pide 8, no 16;* `struct Quad` *(size 16) pide 4, no 16. Si asumís "alineo a* `sizeof` *" vas a desperdiciar y, peor, a equivocarte con structs de tamaño impar (p. ej. size 24 con align 8). La alineación y el tamaño son primos hermanos, no gemelos.*

Ejemplo paso a paso: alineación y tamaño de un struct

Vamos con un caso completo, miembro por miembro:

```
struct Record {
    int    a;   // 4 bytes
    int    b;   // 4 bytes
    double c;   // 8 bytes
    char   d;   // 1 byte
};
```

Paso 1 — la alineación de cada miembro. En x86/x64, lo habitual es:

Miembro	Tipo	`_Alignof`
`a`	`int`	4
`b`	`int`	4
`c`	`double`	8
`d`	`char`	1

Paso 2 — la alineación del struct. Es la **regla del mayor**: el alineamiento de una estructura es el mayor alineamiento de entre todos sus miembros.

```
max(4, 4, 8, 1) = 8
```

Por lo tanto `_Alignof(struct Record)` es **8**.

Paso 3 — acomodar los miembros (y el relleno). El compilador ubica cada miembro en el primer offset que sea múltiplo de *su* alineación:

Miembro	Alineación	Offset donde cae	Ocupa
`a`	4	0	0..3
`b`	4	4	4..7
`c`	8	8	8..15
`d`	1	16	16
relleno final	—	17	17..23 (7 bytes)

Byte a byte queda así:

```
offset   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16 |17 ... 23
       +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+----------+
       |     a (int)   |     b (int)   |           c (double)            | d | relleno |
       +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+----------+
align    ^4              ^4              ^8                                ^1
```

- `b` termina en 8 y `c` pide 8: **entra directo, sin relleno interno**.
- `d` ocupa 1 byte (offset 16), pero el struct completo tiene que medir un múltiplo de 8.

Paso 4 — el `sizeof`. Contenido real: `4 + 4 + 8 + 1 = 17`. Para que sea múltiplo de 8, se agregan **7 bytes de relleno al final**:

```
sizeof(struct Record) = 17 + 7 = 24
```

> ***La trampa del relleno final.*** *El `sizeof` **no** es la suma de los miembros: incluye el relleno. Acá `4 + 4 + 8 + 1 = 17`, pero `sizeof` es **24**, no 16. Esos 7 bytes "fantasma" existen para que, en un arreglo `struct Record items[7]`, cada elemento arranque alineado a 8. Si te olvidás del relleno final, el número no cierra.*

`_Alignof`: ¿operador, macro o función?

`_Alignof` **no es una función** (no se llama en runtime, no existe un símbolo con ese nombre) y **tampoco es, en sí mismo, una macro**. Es un **operador unario**, igual que `sizeof`:

- Lleva el guión bajo (`_Alignof`) porque en C11 entró como **palabra clave reservada**: los identificadores con `_` + mayúscula están reservados al lenguaje, así no choca con código viejo.
- Se evalúa en **tiempo de compilación** y produce un valor de tipo `size_t` (la cantidad de bytes de alineación).
- Su operando es un **tipo** entre paréntesis (`_Alignof(double)`, `_Alignof(struct Record)`), **no** una variable.
- El `_` inicial es feo a propósito. Por eso `<stdalign.h>` define el **macro** `alignof`, que se expande a `_Alignof`. Ahí está tu respuesta fina: `alignof` **es** un macro; `_Alignof` es la palabra clave. En C23 `alignof` pasó a ser palabra clave por derecho propio (y el macro se jubiló).

Cómo usarlo:

```c
#include <stdio.h>
#include <stdalign.h>   // trae el macro `alignof`

typedef struct { int a; int b; double c; char d; } Record;

int main(void) {
    printf("sizeof(int)     = %zu\n", sizeof(int));       // 4
    printf("alignof(double) = %zu\n", alignof(double));   // 8
    printf("sizeof(Record)  = %zu\n", sizeof(Record));    // 24
    printf("alignof(Record) = %zu\n", alignof(Record));   // 8
    return 0;
}
```

Detalle de `printf`. *`_Alignof` y `sizeof` devuelven `size_t`: imprimilos con `%zu`, no con `%d` (si no, el formato no coincide con el tipo y tenés undefined behavior).*

Para la arena. *Este es el número que le pasás a `ARENA_ALIGN`: `_Alignof(max_align_t)` cubre cualquier tipo; `_Alignof(Record)` alcanza para los `Record`. Lo importante: la alineación se **lee del tipo**, no se adivina desde el `sizeof`.*

La fórmula, para tontos

El corazón de todo es redondear **hacia arriba** al múltiplo más cercano:

```c
#define ARENA_ALIGN 8  // alineamos a 8 bytes (potencia de 2)

// Redondea n hacia arriba al múltiplo de ARENA_ALIGN más cercano.
#define ARENA_ALIGN_UP(n) (((n) + (ARENA_ALIGN - 1)) & ~(size_t)(ARENA_ALIGN - 1))
```

La parte `& ~(ARENA_ALIGN - 1)` parece magia negra, pero es solo aritmética de bits. Vamos despacio, con `A = 8`:

1. `A - 1 = 7 = 0b0111`. Es una máscara con los **3 bits de abajo** en 1. Esos 3 bits son justo los que "sobran" dentro de un bloque de 8 (los números 0..7).
2. `n + (A - 1)` = `n + 7`. Sumar 7 "empuja" el número hacia arriba lo suficiente para que, cualquiera sea su resto dentro del bloque de 8, cruce al próximo límite. Si ya estaba justo en un límite, el `+7` se queda adentro del mismo bloque y da lo mismo.
3. `~(A - 1) = ~0b0111 = 0b...11111000`. El `~` invierte: ahora esos 3 bits bajos están en 0 y el resto en 1. Hacer `&` con esto **borra el resto**, deja los bits altos intactos.
4. **Resultado:** el mayor múltiplo de 8 que es ≤ `n+7`, o sea el **menor múltiplo de 8 ≥** `n`.

Tabla, con bits de 4 para que se vea:

`n`	`n + 7`	`n + 7` en binario	`& ~7`	resultado
0	7	`0111`	`0000`	**0**
1	8	`1000`	`1000`	**8**
7	14	`1110`	`1000`	**8**
8	15	`1111`	`1000`	**8**
9	16	`1 0000`	`1 0000`	**16**

Ahí está toda la fórmula: **sumar `A-1` y borrar los bits bajos con `~`**. Nada más.

> ***¿Por qué solo sirve con potencias de 2?*** *Porque `~(A-1)` solo "borra exactamente los bits del resto" cuando `A` es potencia de 2. Para `A = 10` esa máscara no existe y habría que dividir con `/` y `%`. Por suerte **todas** las alineaciones estándar son potencias de 2 (1, 2, 4, 8, 16...), así que el truco siempre aplica.*

Aplicala a la posición, no al tamaño (variante robusta)

El capítulo redondea el **tamaño** y da por hecho que `offset` ya estaba alineado. Eso alcanza mientras el bloque base esté alineado y todos pidan lo mismo. La versión más honesta **redondea el cursor antes de repartir**, que es lo que de verdad querés: que el hueco *nuevo* empiece alineado.

```c
void *arena_alloc(Arena *a, size_t size) {
    // Subo el cursor del offset al múltiplo de 8 más cercano ...
    size_t aligned = ARENA_ALIGN_UP(a->offset);
    // ... y recién ahí veo si entra el pedido exacto.
    if (aligned + size > a->capacity) {
        return NULL;
    }
    void *ptr = a->buffer + aligned;   // puntero al hueco ya alineado
    a->offset = aligned + size;        // el cursor queda donde termina el dato
    return ptr;
}
```

Diferencia con la versión anterior (la del capítulo, que redondea el tamaño): esta **desperdicia menos** (usa `size` exacto y solo paga el *padding* necesario antes del bloque) y expresa mejor la idea "alineá la **dirección** donde arranca el dato".

Ojo: el bloque base también manda

Todo lo anterior asume que `a->buffer` es múltiplo de 8. Como el puntero que devolvés es `base + offset`, si `base` está torcido, **todo** queda torcido y alinear el offset no sirve de nada.

- `malloc` **garantiza** la alineación de cualquier tipo, así que un bloque pedido con `malloc` es seguro.
- En cambio un `unsigned char buffer[4096];` local **puede** quedar alineado a 1, ¡aunque el array sea grande! Si vas a prestar memoria (ejercicio 12.11), forzá la alineación o, mejor, alineá la **dirección absoluta**, que se banca hasta una base torcida:

```c
#include <stdint.h>   // uintptr_t

void *arena_alloc(Arena *a, size_t size) {
    // Alineo la DIRECCIÓN real (base + offset), no solo el offset.
    uintptr_t start   = (uintptr_t)(a->buffer + a->offset);
    uintptr_t aligned = (start + (ARENA_ALIGN - 1)) & ~(uintptr_t)(ARENA_ALIGN - 1);
    size_t    pad     = aligned - start;   // los bytes que "sacrifico" para alinear
```

```
    if (a->offset + pad + size > a->capacity) {
        return NULL;
    }
    a->offset += pad + size;
    return (void *)aligned;
}
```

Ese `pad = aligned - start` **es** la respuesta literal a "cómo calculo la posición que debe ocupar": la posición es `aligned`, y el costo de alinear es `pad`. Simple y sin depender de que `base` haya venido alineado.

Lo que te llevás de alineación

- La alineación es del **tipo**, no del valor ni de la arena.
- Para mezclar tipos **no** calculás posiciones: elegís **una** alineación universal (`_Alignof(max_align_t)`) y alineás cada reparto. Un múltiplo de 8 ya es múltiplo de 8, 4, 2 y 1.
- La alineación de un struct = la mayor de sus miembros, y **siempre divide a su** `sizeof`. Si el bloque arranca alineado, los miembros caen alineados solos.
- La fórmula es `(n + A - 1) & ~(A - 1)`, y solo funciona con `A` potencia de 2.
- El bloque **base** también tiene que estar alineado; lo más robusto es alinear la dirección absoluta, no solo el offset.

12.4.2 ¿Y si no entra lo que pedís?

En este ejemplo simple, `arena_alloc` devuelve `NULL` si la arena está llena. Un diseño más cómodo es que la arena sea una **cadena de regiones** (*chunks*): cuando una se llena, agrega otra y sigue. Eso permite "crecer" sin límite práctico, a costa de más código. Los ejercicios te van a pedir ambas variantes.

12.5 Conceptos que vas a necesitar después

- **Checkpoint / restore (arenas temporales o *scoped*).** Aunque la arena no libera objetos, podés **guardar el offset actual** (`checkpoint`), reservar cosas temporales, y luego **volver** a ese offset (`restore`) para descartar todo lo reservado desde el checkpoint. Es la forma de tener "sub-arenas" que se liberan parcialmente, sin un solo `free` .
- **Helpers tipados.** Escribir `arena_alloc(&a, sizeof(Point))` es tedioso y propenso a error de tipo. Se acostumbra envolverlo en una función o macro que reciba el tipo: por ejemplo `arena_alloc_type(&a, Point)` . Cuidado con las macros: el preprocesador juega sucio.
- **Arena sobre memoria estática (stack/global).** Si sabés que nunca necesitás más de X bytes, podés crear la arena sobre un `unsigned char buffer[X];` local, sin `malloc` **en absoluto**. Rapidísimo y sin fugas. Limitación: no podés devolver punteros a esa memoria más allá de la función.
- **Arena para cadenas.** Como las cadenas son bytes, una operación muy útil es `arena_strdup` : copiar un `const char *` a la arena y devolver tu propia copia. Perfecto para tokens de un parser.
- **Benchmark.** Con frecuencia una arena es **varias veces más rápida** que miles de `malloc` . Medirlo es parte de entender el por qué.

12.6 Errores comunes (leelos, te van a pasar)

- **Guardar punteros después de un `reset` / `destroy` y usarlos.** Eso es *use-after-free* con otra cara. La arena te da velocidad, no inmunidad.
- **Asumir que `arena_alloc` devuelve memoria en cero.** `malloc` no limpia; la arena tampoco. Si necesitás ceros, acordate de inicializar (`memset`).
- **Olvidar la alineación** y sufrir *undefined behavior* con `double` /structs.
- **Creer que podés liberar un objeto del medio.** No se puede. Si lo necesitás, la estructura equivocada es la arena.
- **Reservar la arena *dentro* de la función que la usa y devolverle un puntero a alguien.** El bloque muere al salir de la función (si la creaste sobre el stack) o cuando la destruís.
- **`offset` desbordado.** Si no chequearas `offset + size > capacity` , escribirías fuera del bloque: *buffer overflow* clásico.

12.7 Dato de color: arenas en programas que probablemente uses

No es una curiosidad académica: este truco está metido en software masivo que usás todos los días. Cada caso nombra **qué parte** del programa la usa y **para qué**, como simple información (no hace falta entenderlos a fondo):

- **Google Chrome, Microsoft Edge, Node.js** (los tres corren el motor V8): V8 reserva los nodos del *parser* y del compilador de JavaScript en "zonas". La documentación lo dice tal cual: los bloques "no se pueden liberar por separado… se liberan todos juntos en una sola operación", y las zonas se usan "en todo el parser y los compiladores de V8". Terminado el script, la zona entera se tira de una sola vez.
- **Mozilla Firefox** (motor SpiderMonkey): usa un asignador llamado `LifoAlloc`, que su propia doc define como un *bump allocator*: "asignación rápida, pero incapaz de liberar asignaciones individuales". Se usa mientras se parsea y se compila (JIT) el JavaScript, y trae `mark` / `release` — el mismo *checkpoint* / *restore* del ejercicio 12.8.
- **Juegos hechos con Unity** (miles, y gran parte del mercado móvil): Unity documenta `Allocator.Temp` como un *stack allocator* del que "cada frame, el hilo principal crea un allocator Temp y lo libera por completo al final del frame". Es exactamente el `arena_reset` por frame del ejercicio 12.2, pero corriendo 60 veces por segundo.
- **Python** (CPython): el asignador de objetos chicos usa bloques llamados, literalmente, "arenas" (de 256 KiB). El archivo se llama, sin vueltas, `Objects/obmalloc.c`.
- **Java y Android** (y todo lo que corre encima, de apps móviles a Minecraft): la máquina virtual asigna objetos nuevos casi "empujando" un puntero en la generación joven (el famoso TLAB), sin buscar huecos. El aseo de la basura, otra vez, se hace en bloque.

De los que mencionaste —Word, Excel, Photoshop, Notepad, Paint— no tengo una fuente que afirme "acá usan una arena" y no te lo voy a inventar. Pero el concepto es el mismo: la próxima vez que leas "pool", "zone", "arena" o "region allocator" en el código de un programa grande, ya sabés de qué hablan.

12.8 Ejercicios

> *Dificultad incremental, de **noob** a **pro**. Cada ejercicio declara sus conceptos objetivo. Resolvelos **usando esos conceptos**. Sin código de solución: el libro te da el problema, no la respuesta.*
>
> *Cada ejercicio trae un **escenario de prueba**: una secuencia concreta de operaciones y el **resultado esperado**. Usalo para **probar lo que vas construyendo** (imprimí el resultado real por consola y comparalo). El escenario describe entrada/salida esperada, nunca el algoritmo.*

Ejercicio 12.1 — La arena mínima

Conceptos · `struct`, `malloc`, `free`, punteros, `size_t`

Contexto · Estás escribiendo la base de un asignador para un editor de texto que reserva muchos objetos pequeños de corta vida. Antes de optimizar nada, necesitás la estructura mínima de una arena. Simplificación: ignoramos alineación por ahora.

Consigna · Implementá una estructura `Arena` y las funciones `arena_create`, `arena_alloc` y `arena_destroy`. `arena_alloc` reparte el cursor y devuelve `NULL` si no hay lugar. Después de reservar, **guardá un valor por el puntero** (por ejemplo, un `int`) y **leelo de vuelta**: el espacio sirve solo si podés escribir y recuperar datos en él.

Entrada · Una capacidad en bytes y pedidos de tamaño variable. **Salida** · Punteros válidos dentro del bloque, o `NULL` cuando la arena se llena; y el valor guardado se recupera igual.

Escenario de prueba ·

```
capacity = 16

1. arena_create(16)  -> arena con 16 bytes disponibles
2. int *p = arena_alloc(4)  -> OK (base + 0)  ; *p = 7 ; leer *p -> 7
3. arena_alloc(4)    -> OK   (base + 4)
4. arena_alloc(8)    -> OK   (base + 8)   -> la arena queda llena
5. arena_alloc(1)    -> NULL ("no hay espacio")
6. arena_destroy()   -> el bloque completo se libera de una sola vez
```

(El reuso sin `destroy` se ve en el 12.2.)

Identificadores sugeridos · `Arena`, `arena_create`, `arena_alloc`, `arena_destroy`, `capacity`, `offset`

Criterios de aceptación ·

- `arena_alloc` devuelve direcciones **contiguas** y crecientes.
- Se guarda un valor por el puntero y se recupera intacto (el espacio es **usable**, no solo reservado).
- No se escribe **nunca** fuera del bloque.
- Un único `free` en `arena_destroy` libera todo.
- `arena_alloc` devuelve `NULL` exactamente cuando no hay lugar.

Ejercicio 12.2 — Reset y reutilización

Conceptos · punteros, ciclo de vida de memoria, `for`

Contexto · Un motor de juego reserva, en cada frame, un montón de datos temporales. Reservar y liberar 60 veces por segundo es un desperdicio. Simplificación: no hay gráficos, solo el ciclo de reservas.

Consigna · Agregá `arena_reset` que vuelva el cursor a 0 sin liberar el bloque. Simulá N "frames": en cada uno reservá varios objetos, **escribí valores en ellos y leelos de vuelta**, y recién después reseteá la arena. La dirección **base** del bloque no debe cambiar entre frames.

Entrada · Cantidad de frames y cuántos bytes reserva cada frame. **Salida** · Para cada frame, la dirección del primer objeto reservado (debe repetirse) y el uso total.

Escenario de prueba ·

```
capacity = 16 ; frames = 3 ; cada frame reserva 4 + 4 + 4 = 12

frame 0:
  int *a = alloc(4); *a = 10; leer *a -> 10   (offset 0)
  int *b = alloc(4); *b = 20; leer *b -> 20   (offset 4)
  reset -> offset vuelve a 0
frame 1:
  int *a = alloc(4) -> base + 0   (misma dirección que en el frame 0)
  *a = 30; leer *a -> 30          (el valor viejo ya no está: hueco reutilizado)
frame 2:
  int *a = alloc(4) -> base + 0   (otra vez la misma dirección)
```

Esperado: las tres primeras direcciones son idénticas y el programa hace **un solo** `malloc` en total.

Identificadores sugeridos · `arena_reset` , `frame` , `frame_count`

Restricciones / pistas · Después del reset, `offset` vuelve a 0 pero `buffer` es el mismo. Pensá qué pasa si guardás un puntero de un frame y lo usás en el siguiente.

Criterios de aceptación ·

- El bloque se reserva **una sola vez**.
- La primera dirección de cada frame es idéntica.
- Cada frame guarda y recupera sus valores correctamente del espacio reutilizado.
- No hay ningún `malloc` dentro del bucle de frames.

Ejercicio 12.3 — Alineación correcta

Conceptos · aritmética de bits, alineación, punteros

Contexto · Tu arena guarda mezclados `char`, `int` y `double`. Sin alineación, algunos accesos fallan en ciertas arquitecturas. Simplificación: solo verificamos direcciones múltiplo de la alineación.

Consigna · Modificá `arena_alloc` para que **cada** puntero devuelto quede alineado a un múltiplo de 8. Agregá una función `arena_is_aligned` que diga si una dirección dada es múltiplo de 8.

Entrada · Una secuencia de pedidos de tamaños impares (3, 5, 1, 9…). **Salida** · Las direcciones devueltas y un booleano de alineación para cada una.

Escenario de prueba ·

```
capacity = 32 ; alineación = 8

1. arena_alloc(3)  -> base + 0    (offset 0  -> 8)
   escribir 'A' en base + 0 y leer -> 'A'
2. arena_alloc(1)  -> base + 8    (offset 8  -> 16)
3. arena_alloc(5)  -> base + 16   (offset 16 -> 24)
4. arena_alloc(9)  -> NULL   (9 se redondea a 16 y solo quedan 8 bytes)
5. arena_alloc(8)  -> base + 24   (offset 24 -> 32; arena llena)
6. arena_alloc(1)  -> NULL
```

Además: `arena_is_aligned(base + 0)`, `+ 8`, `+ 16`, `+ 24` -> todos `true` (múltiplo de 8).

Identificadores sugeridos · `ARENA_ALIGN`, `arena_is_aligned`, `aligned`

Restricciones / pistas · Redondeá el tamaño hacia arriba, no la dirección. Usá máscara de bits (asumiendo alineación potencia de 2).

Criterios de aceptación ·

- Toda dirección devuelta es múltiplo de 8.
- Reservar 1 byte consume 8 de capacidad.
- Se escribe y se lee un valor desde un hueco alineado.
- No se pierde memoria fuera de lo necesario (no redondeás de más).

Ejercicio 12.4 — Helper tipado

Conceptos · macros o funciones, `sizeof`, punteros genéricos (`void *`)

Contexto · Escribir `arena_alloc(&a, sizeof(Point))` mil veces invita a errores de tipo. Querés una forma cómoda y segura de pedir "un objeto de tal tipo". Simplificación: solo tipos de tamaño fijo.

Consigna · Ofrecé una abstracción `ARENA_ALLOC_TYPE(arena, Type)` (o una función equivalente) que reserve exactamente `sizeof(Type)` bytes alineados y devuelva un puntero al tipo pedido, **sin cast explícito** en el sitio de uso.

Entrada · Distintos tipos (`int`, `double`, `Point`). **Salida** · Punteros usables directamente como el tipo pedido.

Escenario de prueba ·

```
capacity = 64 ; alineación = 8 ; Point = { int x; int y; } (8 bytes)

1. int    *n = ARENA_ALLOC_TYPE(&a, int);     -> OK (usa bytes 0..7)
2. double *d = ARENA_ALLOC_TYPE(&a, double);  -> OK (usa bytes 8..15)
3. Point  *p = ARENA_ALLOC_TYPE(&a, Point);   -> OK (usa bytes 16..23)
4. p->x = 3; p->y = 4; leer p->x / p->y       -> 3 y 4, sin `cast` en el llamador
```

Esperado: no aparece ningún `(Point *)` ni `sizeof(Point)` en el código que usa la macro.

Identificadores sugeridos · `ARENA_ALLOC_TYPE`, `arena_alloc_type`

Restricciones / pistas · Si usás macro, cuidá los paréntesis y evaluá el argumento una sola vez. Si usás `void *` + cast, explicá por qué el cast es seguro.

Criterios de aceptación ·

- El resultado se usa sin `cast` en el llamador.
- El tamaño reservado es exactamente `sizeof(Type)` (más alineación).
- Funciona con structs, no solo con tipos primitivos.

Ejercicio 12.5 — Una arena, muchos tipos

Conceptos · `void *` , `cast` , `sizeof` , alineación, punteros

Contexto · Un equipo de telemetría junta lecturas de sensores de distinto tipo —temperatura (`double`), cantidad de eventos (`int`), estado (`char`)— y, por velocidad, las guarda **juntas** en un único bloque temporal en lugar de reservar cada una por separado. Simplificación: no hay sensores reales, solo los valores mezclados.

Consigna · Usando **una sola** arena, reservá y guardá tres valores de **tipos distintos** (un `double` , un `int` , un `char`). Escribí cada valor, leelos de vuelta y comprobá que coinciden. Verificá además que cada puntero quede alineado a su tipo.

Entrada · Tres valores de tipos distintos. **Salida** · Los tres valores leídos correctamente desde la **misma** arena, y la comprobación de alineación.

Escenario de prueba ·

```
capacity = 64 ; alineación = 8

1. double *t = arena_alloc(&a, sizeof(double));  -> base + 0    (offset 0  -> 8)
2. int    *c = arena_alloc(&a, sizeof(int));     -> base + 8    (offset 8  -> 16)
3. char   *s = arena_alloc(&a, sizeof(char));    -> base + 16   (offset 16 -> 24)
4. *t = 36.5; *c = 42; *s = 'A';
   leer de vuelta  -> 36.5, 42, 'A'
```

Los tres punteros apuntan a huecos distintos del **mismo** bloque, y cada uno es múltiplo de 8.

Identificadores sugeridos · `temperature` , `eventCount` , `status` , `arena_alloc`

Restricciones / pistas · Un `arena_alloc` por valor, todos de la misma arena. No los envuelvas en un struct: la gracia es ver tipos sueltos conviviendo. Cuidá el `cast` al tipo correcto al leer.

Criterios de aceptación ·

- Los tres valores viven en **una sola** arena.
- Se leen de vuelta con el tipo correcto y coinciden con lo escrito.
- Cada puntero queda alineado a 8 (y por lo tanto a su tipo).
- Un único `arena_destroy` libera todo.

Ejercicio 12.6 — Arena para cadenas: `arena_strdup`

Conceptos · cadenas, `strlen`, `memcpy`, punteros, terminador `'\0'`

Contexto · Un tokenizador lee muchísimos `const char *` de un buffer grande y necesita copias propias que sobrevivan al buffer original. Simplificación: no implementás el tokenizador, solo la copia.

Consigna · Implementá `arena_strdup(Arena *a, const char *src)` que copie la cadena `src` (incluido el `'\0'`) a la arena y devuelva el puntero a la copia. Si no hay lugar, `NULL`.

Entrada · Una o más cadenas de origen. **Salida** · Copias independientes cuyo contenido es igual al original.

Escenario de prueba ·

```
capacity = 16

1. arena_strdup(&a, "arena") -> OK  (6 bytes con terminador -> consumen 8)
2. arena_strdup(&a, "hola")  -> OK  (5 bytes con terminador -> consumen 8)
3. arena_strdup(&a, "x")     -> NULL (la arena está llena)
4. se modifica el original "arena" (por ejemplo, su primer byte)
   -> la copia sigue diciendo "arena"; strcmp(copia, "arena") == 0
```

Identificadores sugeridos · `arena_strdup`, `src`, `dst`

Restricciones / pistas · No olvides el byte del terminador: reservá `strlen(src) + 1`. No uses `strdup` de la librería: la gracia es hacerlo vos.

Criterios de aceptación ·

- La copia es válida aun si el `src` original se modifica o deja de existir.
- Se copia el terminador.
- Devuelve `NULL` si la arena no tiene lugar.

Ejercicio 12.7 — Estadísticas de uso

Conceptos · `struct`, punteros, aritmética, operadores relacionales

Contexto · En desarrollo querés detectar si tu arena de 1 MiB está siempre al borde del desborde, para dimensionarla bien. Simplificación: solo métricas, sin redimensionado.

Consigna · Agregá funciones que reporten: bytes usados, bytes libres, y el **pico histórico** de uso (el `offset` máximo alcanzado, aunque luego haya habido `reset`).

Entrada · Una secuencia de reservas y resets. **Salida** · `used`, `remaining`, `peak` correctos en cada momento.

Escenario de prueba ·

```
capacity = 64
operación          used   remaining   peak
-----------------  -----  ---------   ----
arena_create(64)     0        64         0
arena_alloc(8)       8        56         8
arena_alloc(16)     24        40        24
arena_reset()        0        64        24   <- peak NO baja
arena_alloc(4)       8        56        24
arena_alloc(40)     48        16        48
arena_reset()        0        64        48
```

Esperado: en toda fila `used + remaining == 64`, y `peak` solo puede subir.

Identificadores sugeridos · `arena_used`, `arena_remaining`, `peak_usage`

Restricciones / pistas · `peak` **no** puede bajar con `reset`: es un máximo histórico. Distinguí "usado ahora" de "usado alguna vez".

Criterios de aceptación ·

- `used + remaining == capacity` en todo momento.
- Tras varios resets con usos distintos, `peak` refleja el mayor uso jamás alcanzado.
- Ninguna función estadística reserva ni libera memoria.

Ejercicio 12.8 — Checkpoints (arenas temporales)

Conceptos · punteros, copia de estado, `size_t`, anidamiento lógico

Contexto · Dentro del procesamiento de un documento, cada sección necesita memoria temporal que podés tirar al terminar la sección, **sin** tirar lo acumulado antes. Simplificación: secciones anidadas de profundidad acotada.

Consigna · Implementá `arena_checkpoint(Arena *a)` que devuelva el offset actual, y `arena_restore(Arena *a, size_t checkpoint)` que vuelva el cursor a ese punto (descartando lo reservado después). Usalos para procesar secciones temporales anidadas.

Entrada · Una estructura de secciones que reservan y destruyen memoria temporal. **Salida** · Al restaurar, las reservas posteriores al checkpoint dejan de estar "vivas".

Escenario de prueba ·

```
capacity = 32

1. cp0 = arena_checkpoint(&a)   -> 0
2. int *x = arena_alloc(8); *x = 100; leer *x -> 100   (base + 0, offset 8)
3. arena_alloc(8)               -> base + 8     (offset 16)
4. cp1 = arena_checkpoint(&a)   -> 16
5. int *tmp = arena_alloc(8); *tmp = 999         (base + 16, offset 24)
6. arena_restore(&a, cp1)       -> offset 16    (se descarta `tmp`)
7. leer *x -> 100 (sobrevive al restore); usar `tmp` ahora es inválido
8. arena_alloc(16)              -> base + 16    (offset 32) -> reutiliza el hueco
9. arena_restore(&a, cp0)       -> offset 0     (se descarta todo)
10. arena_alloc(32)             -> base + 0     (offset 32) -> ahora sí entra
```

Ojo: el puntero devuelto en el paso 5 queda inválido después del paso 6.

Identificadores sugeridos · `arena_checkpoint`, `arena_restore`, `checkpoint`

Restricciones / pistas · Este es tu primer "liberar parcialmente" sin `free`. Cuidado: los punteros reservados después del checkpoint quedan inválidos al restaurar.

Criterios de aceptación ·

- Restaurar a un checkpoint rehabilita exactamente la capacidad liberada.
- Lo reservado **antes** del checkpoint conserva su valor tras restaurar.
- Checkpoints anidados funcionan (restaurar al más interno, luego al externo).
- Restaurar es O(1): solo escribe el offset.

Ejercicio 12.9 — Arena que crece (cadena de regiones)

Conceptos · `malloc` , `free` , structs enlazados, manejo de errores

Contexto · No sabés de antemano cuánta memoria necesita un request: a veces 4 KiB, a veces 50 MiB. Una arena de tamaño fijo te obliga a elegir mal. Simplificación: crecimiento por duplicación.

Consigna · Rediseñá la arena como una **cadena de regiones** (`chunk`): cuando una región se llena, se reserva otra (por ejemplo del doble de tamaño) y se sigue repartiendo ahí. `arena_alloc` nunca devuelve `NULL` salvo que la reserva del sistema falle.

Entrada · Una larga secuencia de reservas que superan con creces la capacidad inicial. **Salida** · Todas las reservas exitosas; los punteros viven hasta el `arena_destroy` .

Escenario de prueba ·

```
región inicial = 16 bytes ; crecimiento = x2 (la nueva región nunca es menor a lo pedido)

1. arena_alloc(12)  -> OK  (entra en la región 1; quedan pocos bytes)
2. arena_alloc(12)  -> OK  (no entra en la región 1 -> se crea la región 2, de 32)
3. arena_alloc(100) -> OK  (no entra en ninguna -> se crea la región 3, >= 100)
4. se leen los datos de los pasos 1 y 2 -> siguen intactos
5. arena_destroy()  -> se liberan las 3 regiones (cada una exactamente una vez)
```

Identificadores sugeridos · `Chunk` , `arena_grow` , `next` , `chunk_size`

Restricciones / pistas · Elegí una política de crecimiento y **justificala**. `arena_destroy` debe liberar **todas** las regiones, no solo la actual. Pensá el manejo de error si falla `malloc` .

Criterios de aceptación ·

- Pedir mucho más que la capacidad inicial no falla.
- `arena_destroy` libera cada región exactamente una vez.
- Los punteros ya entregados siguen válidos mientras la arena viva.
- El crecimiento no corrompe datos previos.

Ejercicio 12.10 — Arena como almacén de una colección

Conceptos · structs, arreglos dinámicos, punteros, `arena_alloc`

Contexto · Un sistema de contactos lee N registros de un archivo (nombre + teléfono) y los guarda en memoria temporal para procesarlos. Simplificación: lectura simulada, sin archivos reales todavía.

Consigna · Usando una arena (no un `malloc` por contacto), cargá una lista de contactos y luego calculá: cuántos hay, la longitud total de los nombres, y buscá un nombre. Liberá **todo** con un solo `arena_destroy`.

Entrada · N contactos con nombre (string) y teléfono. **Salida** · Cantidad, suma de longitudes, resultado de la búsqueda, y confirmación de que la memoria se libera de una sola vez.

Escenario de prueba ·

```
capacity = 4096
entrada:
  { "Ana",   "111" }
  { "Bruno", "222" }
  { "Carla", "333" }

salida esperada:
  cantidad                  = 3
  suma de letras de nombres = 3 + 5 + 5 = 13
  buscar "Bruno"            -> encontrado, teléfono 222
  buscar "Zoe"              -> no encontrado
  arena_destroy()           -> TODA la memoria se libera de una sola vez
```

Identificadores sugeridos · `Contact`, `ContactList`, `contacts`, `load_contacts`

Restricciones / pistas · Los nombres también van en la arena (`arena_strdup`). Si el arreglo de contactos puede crecer, decidí entre pedir capacidad fija o usar la arena que crece (12.9).

Criterios de aceptación ·

- **Cero** `malloc` / `free` por contacto: toda la memoria sale de la arena.
- Un único `arena_destroy` limpia todo.
- La búsqueda recorre correctamente los datos guardados en la arena.

Ejercicio 12.11 — Arena sin `malloc` (buffer estático)

Conceptos · arreglos locales, duración de almacenamiento, punteros, límites

Contexto · En un sistema embebido no querés tocar el heap: la fragmentación puede colgar el equipo. Reservás un buffer fijo en memoria y vivís con él. Simplificación: no hay embebido real, solo un buffer estático local.

Consigna · Creá una arena sobre un `unsigned char buffer[4096];` **local** (sin `malloc`). Documentá con claridad por qué **no** podés devolver punteros a esa memoria desde la función que la crea, y demostralo con un caso que **debe fallar** (explicado en comentarios, sin ejecutarlo).

Entrada · Reservas de distinto tamaño dentro de la función. **Salida** · Direcciones dentro del rango `[buffer, buffer + 4096)`.

Escenario de prueba ·

```
unsigned char buffer[4096];  (variable local) ; alineación = 8

1. arena_init_on(&a, buffer, 4096) -> arena sobre memoria prestada
2. char *s = arena_alloc(1000); copiar "hola arena" y leer -> "hola arena"
3. arena_alloc(3000) -> OK (offset 4000)
4. arena_alloc(96)   -> OK (llenó los 4096)
5. arena_alloc(1)    -> NULL
6. arena_destroy()   -> NO libera `buffer` (no es propia)

caso que DEBE fallar (explicado en comentarios, no ejecutado):
  una función crea la arena local y devuelve un puntero a un objeto de `buffer`;
  al volver, `buffer` ya no existe -> puntero colgante (dangling pointer).
```

Identificadores sugeridos · `buffer`, `ARENA_STACK_SIZE`, `arena_init_on`

Restricciones / pistas · La arena apunta al buffer, no lo posee: `arena_destroy` **no** debe hacer `free`. Refactorizá para que la arena funcione tanto con memoria propia como prestada.

Criterios de aceptación ·

- No se usa `malloc` en ningún lado.
- `arena_destroy` distingue "memoria propia" de "memoria prestada" y no libera la prestada.
- Se guarda y se lee una cadena desde el buffer prestado.
- Se explica y se evidencia el peligro de devolver punteros fuera de alcance.

Ejercicio 12.12 — Arena para un árbol sintáctico (pro)

Conceptos · structs con punteros, recursión, arena, cadenas, `enum`

Contexto · Escribir un mini-analizador de expresiones aritméticas (`3 + 4 * (2 - 1)`) donde cada nodo del árbol se reserva en una arena y **todo** se libera de golpe al terminar. Un caso clásico de "muchos nodos pequeños con la misma vida útil".

Consigna · Definí nodos (`number` , `binop`) e implementá el almacenamiento de nodos y de etiquetas de operador usando la arena. Recorré el árbol (recursivo) para imprimir la expresión en notación prefija y calculá el total de nodos y de bytes usados. Liberá con un solo `arena_destroy` .

Entrada · Tokens de una expresión (o la expresión ya tokenizada). **Salida** · La expresión en prefijo, el total de nodos y los bytes de arena consumidos.

Escenario de prueba ·

```
entrada (tokens): 3 + ( 4 - 1 )

salida esperada:
  prefijo: (+ 3 (- 4 1))
  nodos:   5
  bytes:   la suma de los sizeof reservados
           (por ejemplo, si sizeof(Node) == 16 -> 5 * 16 = 80)
```

Sin `malloc` por nodo; un único `arena_destroy` al terminar.

Identificadores sugeridos · `Node` , `NodeKind` , `make_number` , `make_binop` , `print_prefix`

Restricciones / pistas · Cada nodo y cada etiqueta salen de la arena, sin `malloc` . Pensá la vida útil: todos los nodos viven hasta terminar el análisis.

Criterios de aceptación ·

- Ningún nodo se reserva con `malloc` directo.
- El recorrido prefijo es correcto para expresiones anidadas.
- Un solo `arena_destroy` libera todo el árbol.
- Los conteos de nodos y bytes son exactos.

Ejercicio 12.13 — Arena vs `malloc`: medí y decidí (pro)

Conceptos · arena, `malloc` / `free`, medición de tiempo (`clock`), análisis de complejidad

Contexto · Tu equipo duda de si vale la pena migrar a arenas. Nada convence como un número. Simplificación: una medición simple en un solo proceso, sin herramientas externas.

Consigna · Implementá dos versiones del **mismo** workload (por ejemplo, crear y liberar 500.000 objetos pequeños): una con `malloc` / `free` por objeto y otra con una arena. Medí el tiempo de cada una con `clock()` y reportá ambas más la mejora destructiva (cuántas veces más rápida). Además, escribí un **análisis** (en comentarios o en el informe) de **cuándo** conviene cada enfoque, incluyendo un contraejemplo donde la arena **pierde**.

Entrada · Número de objetos y tamaño de cada uno. **Salida** · Tiempos de la versión con `malloc` y con arena, y la relación entre ellos.

Escenario de prueba ·

```
entrada: N = 500.000 objetos de 16 bytes cada uno

salida esperada (los tiempos son orientativos):
  malloc: 0.180 s
  arena:  0.021 s
  la arena es ~8.5x más rápida

contraejemplo (también forma parte del ejercicio):
  reservar 500.000 objetos y luego liberar la mitad "del medio" y seguir trabajando
  -> con malloc es natural; con una arena NO se puede -> la arena es la decisión equivocada
```

Identificadores sugeridos · `bench_malloc`, `bench_arena`, `clock`

Restricciones / pistas · El workload debe ser **idéntico** en ambas versiones para que la comparación valga. Cuidá que el compilador no optimice de más (los "objetos" deben tener efecto observable). El contraejemplo es obligatorio: medir no es lo mismo que entender.

Criterios de aceptación ·

- Ambas versiones hacen el **mismo** trabajo.
- La medición se repite y se reporta de forma honesta (sin descartar resultados incómodos).
- El análisis explica el resultado, no solo lo muestra.
- Se identifica al menos un escenario donde la arena es la decisión **equivocada**.

Ejercicio 12.14 — Registros heterogéneos con unión etiquetada (pro)

Conceptos · `enum` , `union` , punteros, arena, arreglo dinámico, `switch`

Contexto · Un sistema de auditoría registra eventos de distinto tipo —inicio de sesión (con usuario), compra (con monto) y error (con código)— en un mismo flujo temporal. Cada evento tiene una **forma distinta**, pero todos deben vivir juntos y liberarse de una sola vez. Simplificación: los eventos se generan a mano, sin persistencia.

Consigna · Definí un tipo `Event` con un `enum kind` y una `union` de cargas útiles (usuario como cadena, monto como `double` , código como `int`). Creá varios eventos de **distinto tipo** en una **misma** arena (las cadenas también salen de la arena), recorrelos y procesalos según su etiqueta (`switch`), e imprimí un resumen. Liberá todo con un solo `arena_destroy` .

Entrada · Una lista de eventos de tipos variados. **Salida** · Cada evento procesado según su tipo, un total de eventos y un conteo por tipo.

Ejemplo ·

```
entrada:
  LOGIN    usuario = "ana"
  PURCHASE monto   = 19.99
  ERROR    código  = 404

salida esperada:
  LOGIN ana
  PURCHASE 19.99
  ERROR 404
  total eventos = 3
  login = 1, purchase = 1, error = 1
```

Identificadores sugeridos · `Event` , `EventKind` , `USER_MAX` , `make_login` , `make_purchase` , `make_error` , `process_events`

Restricciones / pistas · La `union` ahorra memoria: todos los payloads comparten el mismo espacio y el `enum` te dice cuál es válido en cada momento. Reservá el evento **y** su cadena en la arena. Nunca leas el campo de la `union` que no corresponde al `kind` . Sin `malloc` por evento.

Criterios de aceptación ·

- Un solo `arena_destroy` libera los eventos y sus cadenas.
- El procesamiento respeta la etiqueta (`switch` sobre `kind`), sin leer campos inválidos.
- El conteo por tipo y el total de eventos son exactos.
- No hay `malloc` / `free` por evento.

12.9 Cierre

Una arena no es magia: es **disciplina sobre la duración de los datos**. Si entendés que un conjunto de objetos **nace, vive y muere junto**, ganás velocidad y simplicidad enormes. Si te olvidás de esa condición, la arena te muerde.

Regla mental para llevarte: **una arena por "unidad de trabajo"** (un request, un frame, un documento, un análisis). Cuando la unidad termina, la arena se resetea o se destruye. Nada más.