

Introducción a la Programación

Memoria dinámica

`malloc`, `free`, ownership, ASan y valgrind

Gerson Garrido

Semana 5

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Lo que la pila no puede
2. El heap en el modelo de entornos
3. `malloc`, `sizeof` y `free`
4. Ownership: quién libera cada bloque
5. Errores típicos y cómo leer un informe de ASan
6. Valgrind, y lo que ve cada herramienta
7. `calloc` y `realloc`
8. Imágenes de cualquier tamaño

> **Lectura**
>
> Apuntes de la semana 5. Código en `code/week-05`.

SECCIÓN 01

Lo que la pila no puede

01

Tres límites de la pila

1. **El tamaño se fija al escribir.** `struct image` reserva siempre 64 x 64 píxeles.
2. **La pila es pequeña.** En el entorno del curso, 8 MB (depende del sistema).
3. **Una local muere con su marco.** No se puede devolver su dirección.

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/big_local.c -o build/big_local
$ build/big_local
SUMMARY: AddressSanitizer: stack-overflow errors/big_local.c:6 in main
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g errors/local_address.c -o build/local_address
errors/local_address.c:9:12: warning: function returns address of local variable [-Wreturn-local-addr]
```

Falta una memoria de cualquier tamaño, que se pida al ejecutar y que no dependa de ningún marco.

SECCIÓN 02

El heap

02



El heap *(heap)*

> **Memoria dinámica** ***(dynamic memory)***
> Memoria que se pide al ejecutar, en una zona aparte de la pila: el heap.

- `malloc(n)` reserva un **bloque** de `n` bytes y devuelve su dirección.
- `free(p)` libera el bloque cuyo inicio es `p`.

Un bloque, comparado con una local:

- **No tiene nombre.** Solo se llega a él por un puntero.
- **Su tamaño puede salir de un cálculo** al ejecutar.
- **No depende de ningún marco.** Vive desde `malloc` hasta `free`.

El modelo crece: dos reglas más

Regla	Qué pasa
1 a 4	Llamar, buscar un nombre, asignar y retornar: igual que en la semana 3.
5. Reservar	`malloc` crea un bloque en el heap, sin nombre y con basura adentro, y devuelve su dirección.
6. Liberar	`free(p)` destruye el bloque. El puntero `p` no cambia: sigue guardando la misma dirección, que ya no sirve.

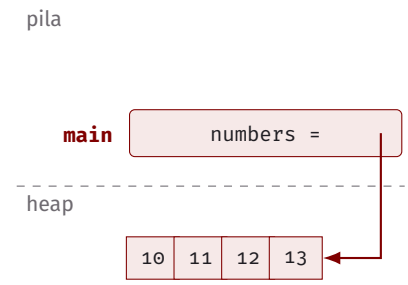
Idea clave

La pila se administra sola y sigue el orden de las llamadas. El heap lo administran ustedes, y un bloque vive exactamente lo que ustedes decidan.

Dibujemos `range_new`

range_block.c

```c
int *range_new(int first, int n) {
    int *block = malloc(n * sizeof *block);
    if (block == NULL) {
        return NULL;
    }
    for (int i = 0; i < n; i++) {
        block[i] = first + i;
    }
    return block;
}
...
int *numbers = range_new(10, 4);
...
free(numbers);
```



Al retornar, el marco desaparece y `block` con él. El bloque sigue ahí.

SECCIÓN 03

malloc y sizeof

03



`malloc` recibe bytes

```
void *malloc(size_t size);
```

- `size_t`: tipo sin signo con que C mide tamaños. Un `int` negativo se vuelve enorme: revisen antes.
- `void *`: dirección sin tipo. Se convierte sola, así que no hace falta un cast.

¿Por qué `int *p = malloc(10);` no reserva 10 enteros?

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/short_block.c -o build/short_block
$ build/short_block
==109==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x502000000018 at pc 0xaaaabe76112c bp 0xffffe8dbcf80 sp 0xffffe8dbcf98
WRITE of size 4 at 0x502000000018 thread T0
    #0 0xaaaabe761128 in main errors/short_block.c:12
0x50200000001a is located 0 bytes after 10-byte region [0x502000000010,0x50200000001a)
```

10 bytes alcanzan para dos enteros y medio: `p[2]` ya se sale.

La forma correcta

reverse_numbers.c

```c
int *values = malloc(n * sizeof *values);
if (values == NULL) {
    fprintf(stderr, "No hay memoria para %d nú
      meros\n", n);
    return 1;
}
```

terminal

```
$ build/sizes
sizeof(int)          = 4
sizeof p  (puntero)  = 8
sizeof *p (un int)   = 4
sizeof q  (puntero)  = 8
sizeof *q (un point) = 8
10 enteros           = 40 bytes
```

`sizeof` solo mira el tipo: no evalúa `*values`.

Error común

`malloc(sizeof p)` reserva lo que mide un puntero. El tamaño del bloque lo guardan ustedes.

Todo `malloc` se revisa

no_memory.c

```c
size_t huge = SIZE_MAX / 2;

char *block = malloc(huge);
if (block == NULL) {
    printf("malloc devolvió NULL: no hay
      memoria\n");
    return 1;
}
```

terminal

```
$ build/no_memory
malloc devolvió NULL: no hay memoria
```

Con sanitizers, ASan detiene el programa ante un pedido imposible, antes de devolver `NULL`.

- `main` puede avisar y terminar.
- Una función de biblioteca libera lo que ya reservó y devuelve `NULL`. Quien llama decide.

Las reglas de `free`

1. Recibe **exactamente** la dirección que dio `malloc`, no otra dentro del bloque.
2. Un bloque se libera **una sola vez**.
3. `free(NULL)` no hace nada. Es legal.
4. No cambia el puntero, ni borra el contenido.

> **Por eso**
>
> Sin sanitizers, un uso después de liberar puede dar un número cualquiera, o el valor de antes, y ningún mensaje. No se puede confiar en que una corrida parezca correcta.

SECCIÓN 04

Ownership

04

¿Quién libera?

Owner

La parte del programa responsable de liberar un bloque. En inglés, la palabra nombra a quien tiene algo como suyo.

- Si nadie libera: **fuga**.
- Si dos liberan: **liberar dos veces**.
- El compilador no lo revisa: `int *` se ve igual si hay que liberar o no.

La regla del curso

Cada bloque tiene exactamente un owner. Puede cambiar, pero hay uno solo a la vez, y la firma, con su comentario, dice quién es. Esa relación es el **ownership**.

Tres firmas, tres respuestas

strings_owner.c

```c
// Quien llama es owner del resultado, y debe liberarlo con free.
char *upper_copy(const char *text) {

// Cambia text en su lugar. El owner de text sigue siendo quien llama.
void upper_in_place(char *text) {

// Devuelve una de las dos cadenas que recibió. No hay memoria nueva: no se libera.
const char *longer_of(const char *a, const char *b) {
```

Función	Qué significa	¿Quién libera?
`upper_copy`	Devuelve memoria nueva	Quien llama, lo devuelto
`upper_in_place`	Toma prestado, y puede cambiarlo	El owner de siempre
`longer_of`	Devuelve algo que ya existía	No se libera lo devuelto

El detalle que más se repite

strings_owner.c

```c
char *copy = malloc(strlen(text) + 1);
if (copy == NULL) {
    return NULL;
}
```

terminal

```
$ build/strings_owner
PARADERO
RUTA
Arequipa
```

- `strlen` no cuenta el `'\0'`. Una cadena de n letras ocupa $n + 1$ bytes.
- Sin el `+ 1`, se escribe un byte fuera del bloque.

Error común

`printf("%s\n", upper_copy("hola"));` funciona y deja una fuga: nadie guardó el puntero.

Un `struct` que es owner de un bloque

student.c

```c
struct student *s = malloc(sizeof *s);
if (s == NULL) {
    return NULL;
}

s->name = malloc(strlen(name) + 1);
if (s->name == NULL) {
    free(s);
    return NULL;
}
```

student.c

```c
void student_free(struct student *s) {
    if (s == NULL) {
        return;
    }
    free(s->name);
    free(s);
}
```

- Dos bloques, y dos `malloc` que pueden fallar. Si falla el segundo, hay que liberar el primero.
- Primero el nombre y después el `struct`. Al revés, leer `s->name` sería un acceso a un bloque liberado.
- Quien usa `student_new` no necesita saber cuántos bloques hay. Solo llama a `student_free`.

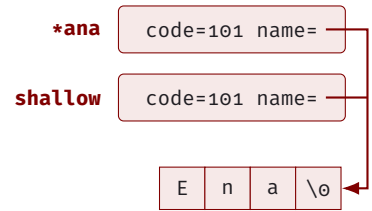
Copiar un `struct` no copia sus bloques

student.c

```c
struct student shallow = *ana;
shallow.name[0] = 'E';
printf("%s %s\n", ana->name, shallow.
      name);
```

terminal

```
$ build/student
Ena Ena
mismo bloque: 1
```



- **Copia superficial:** se copian los campos, y un campo puntero es solo la flecha.
- Dos punteros, un solo bloque. Si cada uno se cree owner, se libera dos veces.
- **Copia profunda:** otro bloque, con el mismo contenido. Cada `struct` es owner del suyo.

SECCIÓN 05

Errores típicos

05

Usar después de liberar: leer el informe

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/use_after_free.c -o build/use_after_free 2>/dev/null
$ build/use_after_free
==132==ERROR: AddressSanitizer: heap-use-after-free on address 0x504000000010 at pc 0xaaaab74b0f34 bp 0xffffed9339b0 sp 0xffffed9339c8
READ of size 4 at 0x504000000010 thread T0
    #0 0xaaaab74b0f30 in main errors/use_after_free.c:13
0x504000000010 is located 0 bytes inside of 40-byte region [0x504000000010,0x504000000038)
freed by thread T0 here:
    #1 0xaaaab74b0ea8 in main errors/use_after_free.c:12
previously allocated by thread T0 here:
    #1 0xaaaab74b0df4 in main errors/use_after_free.c:6
```

1. **Tipo:** `heap-use-after-free`.
2. **Acceso:** `READ of size 4`: leer un `int`.
3. **Línea:** la primera con un archivo de ustedes, `use_after_free.c:13`.
4. **Bloque:** `40-byte region`, al inicio (`0 bytes inside`).
5. **Historia:** liberado en la línea 12, reservado en la línea 6.

Liberar dos veces

errors/double_free.c

```c
void show_and_free(int *values, int n) {
    ...
    free(values);
}
...
show_and_free(values, 3);
free(values);
```

Las dos se creen owners.
Corregir no es borrar un `free` al azar, sino decidir el owner: que libere `main`, que reservó.

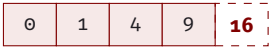
terminal

```
$ build/double_free
==143==ERROR: AddressSanitizer: attempting double-free on 0x502000000010 in thread T0:
    #1 0xaaaadaca10bc in main errors/double_free.c:20
freed by thread T0 here:
    #1 0xaaaadaca106c in show_and_free errors/double_free.c:10
```

Salirse del bloque

errors/off_by_one.c

```c
int n = 4;
int *values = malloc(n * sizeof *values);
...
for (int i = 0; i <= n; i++) {
    values[i] = i * i;
}
```



terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/off_by_one.c -o build/off_by_one
$ build/off_by_one
==154==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x502000000020 at pc 0xaaaacd8b1230 bp 0xffffff5cee6a0 sp 0xffffff5cee6b8
WRITE of size 4 at 0x502000000020 thread T0
    #0 0xaaaacd8b122c in main errors/off_by_one.c:13
0x502000000020 is located 0 bytes after 16-byte region [0x502000000010,0x502000000020)
```

Fuga de memoria *(memory leak)*

errors/leak.c

```c
for (int round = 0; round < 3; round++) {
    int *numbers = range_new(round, 10);
    ...
    printf("%d\n", numbers[9]);
}
```

- El informe llega **al terminar**.
- Dice dónde se reservó, no dónde liberar. El error está en `main`.

terminal

```
$ build/leak
9
10
11
==120==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 120 byte(s) in 3 object(s) allocated from:
    #2 0xaaaac84b124c in main errors/leak.c:18
```

Lo que dice ASan

Error	Qué dice el informe	Qué significa
Usar después de liberar	`heap-use-after-free`	Se accedió a un bloque liberado
Liberar dos veces	`attempting double-free`	`free` sobre un bloque liberado
Salirse del bloque	`heap-buffer-overflow`	Acceso fuera de sus límites
Fuga	`detected memory leaks`	Nadie puede liberar el bloque
`free` con mala dirección	`attempting free on address...`	No es un inicio de bloque

Error común

Corregir el síntoma. Si un bloque se usó después de liberarlo, borrar el `free` hace callar a ASan y deja una fuga. Primero dibujen el bloque, y digan quién era su owner.

SECCIÓN 06

Valgrind

06



Valgrind: sin sanitizers

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -O0 errors/use_after_free.c -o build/use_after_free_vg 2>/dev/null
$ valgrind --leak-check=full build/use_after_free_vg
==42== Invalid read of size 4
==42==    at 0x108868: main (use_after_free.c:13)
==42==  Address 0x4a7f040 is 0 bytes inside a block of size 40 free'd
==42==    at 0x4884F8: free (vg_replace_malloc.c:989)
==42==    by 0x108863: main (use_after_free.c:12)
==42==  Block was alloc'd at
==42==    at 0x488545C: malloc (vg_replace_malloc.c:446)
==42==    by 0x108837: main (use_after_free.c:6)
7
==42== All heap blocks were freed -- no leaks are possible
==42== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

- Ejecuta el programa en una CPU simulada. Se compila con `-g -O0`, sin `-fsanitize`: las dos herramientas no se combinan.
- La misma información: acceso, línea e historia del bloque.
- **Sigue corriendo** después del error, y al final da el total. ASan se detiene en el primero.

Un error que ASan no ve

errors/uninit_read.c

```c
int *counts = malloc(3 * sizeof *counts
    );
...
counts[0] = 5;
counts[2] = 5;
if (counts[1] > 0) {
    printf("hay datos\n");
```

- ASan no reporta nada: el acceso está dentro del bloque.
- Valgrind sabe qué bytes se escribieron alguna vez.
- Avisa al **decidir** con la basura, en el `if`, no al copiarla.

terminal

```
$ valgrind --track-origins=yes build/uninit_read_vg
==69== Conditional jump or move depends on uninitialised value(s)
==69==    at 0x10887C: main (uninit_read.c:13)
==69==  Uninitialised value was created by a heap allocation
==69==    by 0x108837: main (uninit_read.c:6)
```

Un error que valgrind no ve

errors/local_overflow.c

```c
int values[4] = {1, 2, 3, 4};
int total = 0;
for (int i = 0; i <= 4; i++) {
    total = total + values[i];
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -O0 errors/local_overflow.c -o build/local_overflow_vg
$ valgrind build/local_overflow_vg
==78== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/local_overflow.c -o build/local_overflow
$ build/local_overflow
errors/local_overflow.c:8:31: runtime error: index 4 out of bounds for type 'int [4]'
==188==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff87300030 at pc 0xaaaabea81478 bp 0xffffd6a31ac0 sp 0xffffd6a31ad8
```

Valgrind vigila los bloques del heap, no los arreglos locales.

Cuál usar

	ASan	Valgrind
Cómo se usa	Compilar con `-fsanitize=address`	`-g -O0` y correr `valgrind`
Fugas	Sí, al terminar	Sí, con `--leak-check=full`
Bloques del heap	Sí	Sí
Arreglos locales	Sí	No
Valores sin inicializar	No	Sí
Después del primer error	Se detiene	Sigue

Idea clave

Ninguna prueba que un programa está bien. Solo revisan los caminos que **ejecutaron**. Por eso también hay que probar los caminos de error.

SECCIÓN 07

calloc y realloc

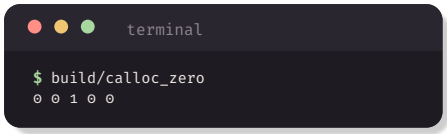
07



`calloc` y `realloc`

`calloc(n, size)`

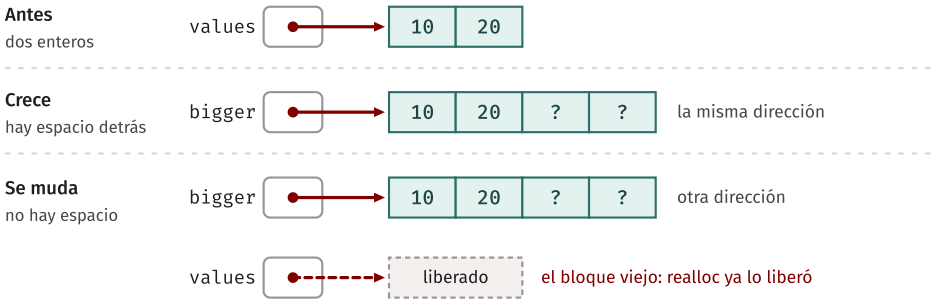
- Reserva `n` elementos, todos en cero.
- Revisa que la multiplicación quepa.



`realloc(p, n)`

- Cambia el tamaño, y conserva el contenido. Lo nuevo trae basura.
- Puede crecer en su lugar, o mudarse: reservar otro bloque, copiar y liberar el viejo.
- Si se muda, los demás punteros al bloque quedan colgantes. Como no se sabe, siempre se asume.
- `realloc(NULL, n)` es igual a `malloc(n)`.

`realloc`: crece o se muda



Como no se sabe cuál de los dos casos ocurrió, se trata como si se hubiera mudado:
un realloc invalida los demás punteros al bloque.

`p = realloc(p, n)` puede causar una fuga

errors/realloc_lost.c

```
1  p = realloc(p, huge);
2  if (p == NULL) {
```

Si falla, `realloc` devuelve `NULL` y **no toca** el bloque viejo. Pero la única copia de su dirección era `p`.

realloc_grow.c

```
1  int *bigger = realloc(values, 4 * sizeof *values
       );
2  if (bigger == NULL) {
3      free(values);
4      return 1;
5  }
6  values = bigger;
```

terminal

```
$ valgrind --leak-check=full build/realloc_lost_vg
realloc devolvió NULL
==87== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==87==    by 0x10887F: main (realloc_lost.c:9)
```

Un arreglo que crece

pseudocódigo

```
algoritmo agregar(arreglo, valor)
    si cuenta = capacidad entonces
        pedir un bloque más grande con
     realloc
        si no se pudo entonces
            devolver falso
        fin si
    fin si
    arreglo[cuenta] ← valor
    cuenta ← cuenta + 1
    devolver verdadero
fin algoritmo
```

¿Cuánto más grande?

terminal

```
$ build/growth_count
de uno en uno: 20000 llamadas
duplicando:    16 llamadas
```

Cada `realloc` puede copiar todo el bloque. Duplicar desperdicia un poco menos de la mitad, a lo sumo, y casi nunca hay que llamarlo.

SECCIÓN 08

Imágenes de cualquier tamaño

08

`struct image` sin límite

lab/image.h

```c
struct image {
    int width;
    int height;
    int max_value;
    int *pixels;
};
```

terminal

```
$ build/image_size
sizeof(struct image) = 24
```

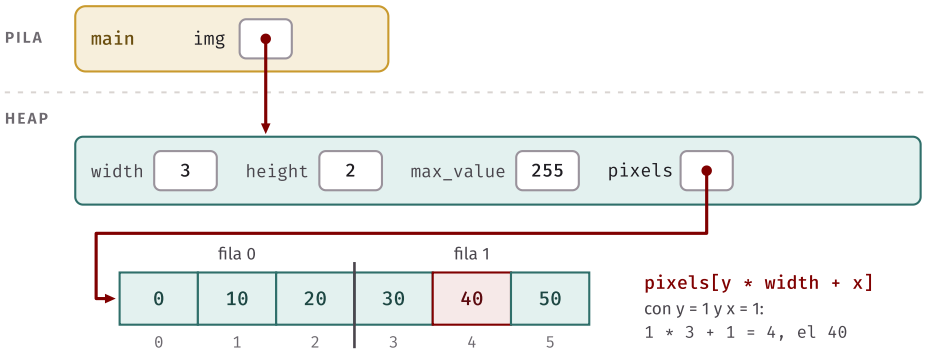
Copiar un `struct image` ya no copia la imagen.

Los píxeles salen del `struct` y pasan a un bloque propio, con todas las filas seguidas: `pixels[y * width + x]`.

Antes, 16396 bytes. Depende del sistema.

Lo que devuelven `image_new` e `image_load` es de quien llama, que lo libera con `image_free`.

Un solo bloque, fila por fila



y * width salta las filas anteriores, y x avanza dentro de la fila

“Sin máximo” no significa sin revisar

lab/image.c

```c
static int valid_size(int width, int height, int max_value) {
    return width >= 1 && height >= 1 && width <= INT_MAX / height
        && max_value >= 1 && max_value <= IMAGE_MAX_VALUE;
}
```

- 50000 x 50000 no cabe en un `int`. Multiplicar sería overflow: se revisa **antes**, con una división.
- El archivo puede mentir: unos pocos bytes pueden decir 40000 x 40000, que son 6.4 GB. Por eso `image_load` no llena la imagen antes de leerla.
- El camino de error importa: si falta un píxel, hay dos bloques que liberar y un archivo que cerrar.

SECCIÓN 09

Laboratorio

09

Laboratorio de esta semana

Sesión 1

1. Cambiar el módulo de imágenes: `image_new`, `image_load` e `image_free`, sin tamaño máximo.
2. Escribir `str_duplicate` y `str_concat`, con pruebas.

Sesión 2

3. Cacería: cinco programas, cinco tipos de error. Explicar cada informe, y corregir.
4. Escribir un arreglo que crece con `realloc`, y usarlo para leer un archivo de cualquier largo.

Comandos

`make lab`, `make hunt` y `make test-lab`. Las pruebas corren con sanitizers: una fuga las hace fallar, aunque el resultado sea correcto.

Para la próxima clase

- Terminen el laboratorio: `make test-lab` debe pasar.
- Sepan responder sin mirar:
  1. ¿Por qué `int *p = malloc(10);` probablemente no reserva 10 enteros?
  2. ¿Qué error tiene `free(p); printf("%d", *p);`? ¿Cómo lo reporta ASan?
  3. Una función devuelve una cadena nueva. ¿Quién debe liberarla? ¿Cómo se lo dicen a quien la usa?
  4. ¿Por qué `p = realloc(p, n);` puede causar una fuga?

Semana 6

Programas en varios archivos y tipos opacos: quien use el módulo de imágenes ya no podrá ver cómo guarda sus píxeles.