

Introducción a la Programación

El modelo de entornos

Marcos, alcance, tiempo de vida e imágenes PGM

Gerson Garrido

Semana 3

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Por qué hace falta un modelo
2. Las cuatro reglas: marcos y nombres
3. Alcance y tiempo de vida. `static` y constantes globales
4. Ver los marcos en `gdb`
5. Paso por valor: `int`, arreglo y `struct`
6. Imágenes PGM y filtros

> **Lectura**
>
> Apuntes de la semana 3. Código en `code/week-03`.

SECCIÓN 01

Por qué un modelo

01

¿Qué imprime?

frames.c

```c
int triple(int n) {
    int result = n * 3;
    return result;
}

int add_triple(int a, int b) {
    int t = triple(b);
    a = a + t;
    return a;
}
int main(void) {
    int a = 2;
    int b = 5;
    int c = add_triple(a, b);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
```

La duda: `a = a + t` cambia `a`. ¿Cambia la `a` que imprime `main`?

terminal

```
$ build/frames
a = 2, b = 5, c = 17
```

Hoy aprendemos a deducir esa salida sin ejecutar.

Por qué no basta con correrlo

- Una corrida dice qué pasó con esos datos, no por qué. No dice qué pasará con otros.
- Un error de memoria puede dar el resultado correcto en una corrida y el incorrecto en otra.
- Antes de escribir un programa no hay nada que correr. Para diseñar, hay que imaginar la ejecución.

Idea clave

Un modelo es útil si predice. **Dibujen primero y ejecuten después.** Si no coinciden, o el programa tiene un bug, o su modelo tiene un error.

SECCIÓN 02

Las reglas del modelo

02

Marco de llamada

Marco de llamada *(stack frame)*

Espacio que crea cada llamada para sus parámetros y sus variables locales.

Regla	Qué pasa
1. Llamar	Se calculan los argumentos. Se crea un marco, y cada parámetro recibe una **copia**, por posición.
2. Buscar un nombre	Entre los ya declarados: en el bloque actual, en los que lo rodean, y fuera de toda función. Nunca en otro marco.
3. Asignar	Cambia solo la casilla que encontró la regla 2.
4. Retornar	Se calcula el valor, el marco desaparece y se sigue en el punto de la llamada.

La regla 2 la revisa el compilador

errors/caller_local.c

```c
void show_total(void) {
    printf("%d\n", total);
}

int main(void) {
    int total = 5;
    show_total();
```

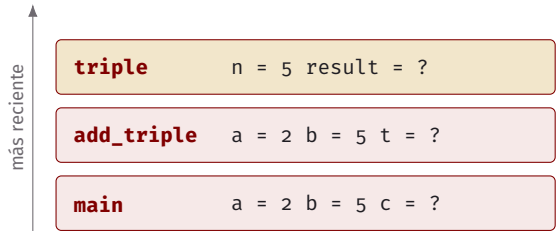
terminal

```
$ gcc -Wall -Wextra -std=c17 errors/caller_local.c -o build/caller_local
errors/caller_local.c:5:20: error: 'total' undeclared (first use in this function)
```

Mientras corre `show_total`, el `total` de `main` existe. Pero una función solo conoce los nombres escritos en su propio texto.

Dibujemos `frames.c`

1. `main` prepara `a` y `b`. `c` espera: ?
2. Se copian 2 y 5. Mismos nombres, otras casillas.
3. `triple(b)`: `b` se busca en el marco actual.
4. `triple` retorna 15. Su marco desaparece.
5. `a = a + t` cambia solo la `a` de arriba.
6. Retorna 17. La `a` de `main` sigue en 2.



La salida: `a = 2, b = 5, c = 17`. Salió de las reglas, sin ejecutar.

¿Qué marcos existen?

Pregunta. `main` llama a `f`, y `f` llama a `g`. ¿Cuáles existen dentro de `g`?

`g`	activo
`f`	esperando
`main`	esperando

- Los **tres**. `main` y `f` no terminaron.
- Solo se pueden usar los nombres de `g`.
- La **pila** *(stack)*: el último marco que entra es el primero que sale.

El límite del modelo

El estándar de C no habla de marcos ni dice hacia dónde crece la pila. Sí garantiza lo que el modelo predice: cada llamada tiene sus propias variables, y al volver se sigue donde se quedó.

SECCIÓN 03

Alcance y tiempo de vida

03

Dos preguntas distintas

Alcance *(scope)*

Propiedad del **texto**: ¿en qué líneas se puede usar este nombre?

Tiempo de vida *(lifetime)*

Propiedad de la **ejecución**: ¿cuánto tiempo existe este dato?

Una variable puede existir sin que su nombre se pueda usar. Mientras corre `triple`, la `t` de `add_triple` está viva y fuera de alcance.

Alcance: los bloques

blocks.c

```c
int x = 1;
printf("afuera: x = %d\n", x);

{
    int x = 2;
    x = x + 10;
    printf("adentro: x = %d\n", x);
}
printf("afuera otra vez: x = %d\n", x);

int total = 0;
for (int i = 1; i <= 3; i++) {
    int square = i * i;
    total = total + square;
}
```

terminal

```
$ build/blocks
afuera: x = 1
adentro: x = 12
afuera otra vez: x = 1
total = 14
```

- Son dos `x`. La de adentro **oculta** a la de afuera.
- `gcc` no avisa ni con `-Wall -Wextra`.
- `square` nace y muere en cada vuelta.

`static`: alcance local, vida completa

static_counter.c

```c
int count_local(void) {
    int calls = 0;
    calls = calls + 1;
    return calls;
}

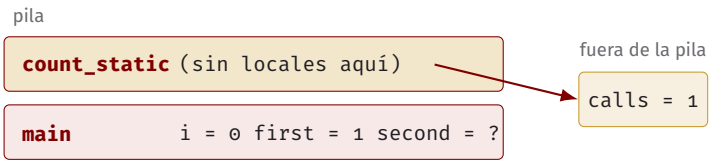
int count_static(void) {
    static int calls = 0;
    calls = calls + 1;
    return calls;
}
```

terminal

```
$ build/static_counter
local: 1, static: 1
local: 1, static: 2
local: 1, static: 3
```

- Existe **una sola vez**, fuera de la pila.
- Se inicializa una vez, **antes** de que el programa empiece.
- El nombre sigue siendo local.

En el dibujo, `static` va aparte



terminal

```
$ gcc -Wall -Wextra -std=c17 errors/static_init.c -o build/static_init
errors/static_init.c:5:24: error: initializer element is not constant
    5 |     static int value = n;
      |                        ^
```

Como se inicializa antes de correr, no puede depender de un parámetro.

Resumen de las variables

Tipo	Alcance	Tiempo de vida	Dónde se dibuja
Parámetro	Toda la función	La llamada	En el marco
Local	Hasta el fin del bloque	La ejecución del bloque	En el marco
Local `static`	Hasta el fin del bloque	Todo el programa	Fuera de la pila
Constante global	Hasta el fin del archivo	Todo el programa	Fuera de la pila

Costo de `static`

Con los mismos argumentos, la función da resultados distintos. Eso complica sus pruebas. Se usa poco.

Constantes globales y `#define`

global_constant.c

```c
#define MAX_STUDENTS 4
const int PASSING_GRADE = 11;
```

- Una constante global, nadie la puede cambiar.
- `#define` reemplaza texto.
- `const int` es una variable.

errors/const_size.c

```c
const int MAX_WIDTH = 64;
struct line {
    int cells[MAX_WIDTH];
};
```

Por eso el módulo de imágenes usa `#define`.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/const_size.c -o build/const_size
errors/const_size.c:7:9: error: variably modified 'cells' at file scope
```

SECCIÓN 04

Los marcos en gdb

04



Comprobar el dibujo

Compilar con `-g -O0` y sin sanitizers.

Comando	Qué hace
`backtrace`	Muestra la pila. `#0` es el marco más reciente
`frame 1`	Selecciona ese marco para mirarlo. No cambia la ejecución
`up`, `down`	Selecciona el marco de quien llamó, o vuelve
`info args`	Muestra los parámetros del marco seleccionado
`info locals`	Muestra las locales del marco seleccionado
`whatis x`	Muestra el tipo de `x`

Los tres marcos de verdad

gdb

```
Breakpoint 1, triple (n=5) at frames.c:5
5       int result = n * 3;
(gdb) backtrace
#0  triple (n=5) at frames.c:5
#1  0x0000aaaaaaaa07e8 in add_triple (a=2, b=5) at frames.c:10
#2  0x0000aaaaaaaa082c in main () at frames.c:18
(gdb) frame 1
#1  0x0000aaaaaaaa07e8 in add_triple (a=2, b=5) at frames.c:10
10      int t = triple(b);
(gdb) info locals
t = 65535
(gdb) frame 2
#2  0x0000aaaaaaaa082c in main () at frames.c:18
18      int c = add_triple(a, b);
(gdb) info locals
a = 2
b = 5
c = 65535
```

- Tres marcos, como el dibujo.
- Donde dibujamos `?`, hay basura: 65535. En su máquina pueden salir otros.
- `frame` no retrocede: el programa sigue detenido en `triple`.
- La línea que se muestra todavía no se ejecutó.

Las direcciones y la basura dependen del entorno.

Alcance y vida en gdb

gdb

```
(gdb) break 11
Breakpoint 1 at 0x7dc: file blocks.c, line 11.
(gdb) run
afuera: x = 1

Breakpoint 1, main () at blocks.c:11
11          printf("adentro: x = %d\n", x);
(gdb) info locals
x = 12
x = 1
total = 2
(gdb) print x
$1 = 12
```

- Las dos `x`, la de adentro primero. `print x` elige esa.
- Aparece `total` con basura, aunque se declara **más abajo**.
- La variable ya existe: su bloque empezó. Su nombre aún no tiene alcance.

Idea clave

Aquí, el tiempo de vida empieza antes que el alcance.

`static` en `gdb`

gdb

```
Breakpoint 1, count_static () at static_counter.c:12
12      calls = calls + 1;
(gdb) continue
Continuing.
local: 1, static: 1

Breakpoint 1, count_static () at static_counter.c:12
12      calls = calls + 1;
(gdb) info locals
calls = 1
(gdb) up
#1  0x0000aaaaaaaa0810 in main () at static_counter.c:19
19          int second = count_static();
(gdb) print count_static::calls
$1 = 1
(gdb) print calls
No symbol "calls" in current context.
```

- Se detiene en la línea 12: la 11 no hace nada en cada llamada.
- En la segunda llamada, `calls` ya vale 1.
- Desde `main`, el dato existe, y el nombre no.
- `count_static::calls` es sintaxis de `gdb`, no de C.

SECCIÓN 05

Paso por valor en el modelo

05

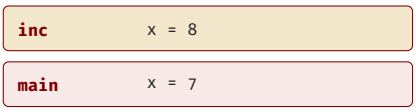
¿Por qué `inc` no cambia nada?

by_value.c

```c
void inc(int x) {
    x++;
    printf("en inc: x = %d\n", x);
}
```

terminal

```
$ build/by_value
en inc: x = 8
en main: x = 7
con return: x = 8
```



- Regla 1: `x` es otra casilla, con una copia.
- Regla 3: `x++` solo cambia esa casilla.
- Regla 4: al retornar, desaparece.

C siempre copia el valor del argumento. La pregunta es qué es “el valor” en cada caso.

Un programa, tres resultados

array_param.c

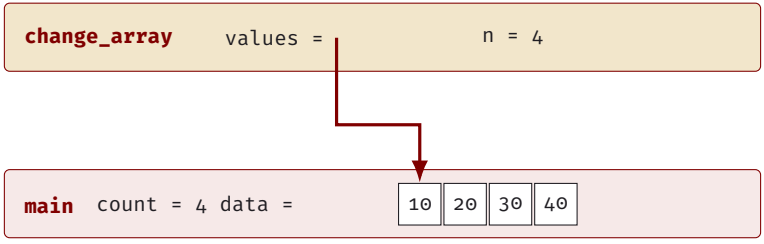
```c
void change_array(int values[], int n) {
...
        values[i] = values[i] * 10;
...
    n = 0;
...
change_int(data[0]);
change_array(data, count);
```

terminal

```
$ build/array_param
en change_int: x = 99
1 2 3 4
en change_array: n = 0
10 20 30 40
count = 4
```

Se pasa	Se copia	Qué toca	¿Lo ve `main`?
`data[0]`	El `int`	Su casilla `x`	No
`count`	El `int`	Su casilla `n`	No
`data`	La dirección del inicio	Los elementos de `main`	Sí

Un parámetro arreglo se dibuja con una flecha



- Hay **un solo arreglo**, en el marco de `main`.
- La función tiene una copia de la dirección. `values[i] = ...` llega a los elementos originales.
- Después del ciclo, `main` ve 10, 20, 30 y 40.

gdb confirma la flecha

gdb

```
Breakpoint 1, change_array (values=0xffffffffa18, n=4) at array_param.c:12
12      for (int i = 0; i < n; i++) {
(gdb) whatis values
type = int *
(gdb) print values
$1 = (int *) 0xffffffffa18
(gdb) print values[0]@4
$2 = {1, 2, 3, 4}
(gdb) up
#1  0x0000aaaaaaaa096c in main () at array_param.c:33
33      change_array(data, count);
(gdb) print sizeof(data)
$3 = 16
(gdb) down
#0  change_array (values=0xffffffffa18, n=4) at array_param.c:12
12      for (int i = 0; i < n; i++) {
(gdb) print sizeof(values)
$4 = 8
```

- Se escribió `int values[]`, pero el parámetro es una dirección: `int *`.
- Ese tipo es un **puntero**: el tema de la próxima semana.
- Por eso se pasa la longitud aparte: `sizeof` no mide el arreglo en la función.

Las direcciones y el 8 dependen del entorno.

`struct`: se copia todo, incluso un arreglo

struct_copy.c

```c
struct row {
    int size;
    int cells[3];
};

void change_copy(struct row r) {
    r.cells[0] = 99;
...
void change_cells(int cells[]) {
    cells[1] = 77;
}
...
struct row a = {3, {1, 2, 3}};
change_copy(a);
...
change_cells(a.cells);
```

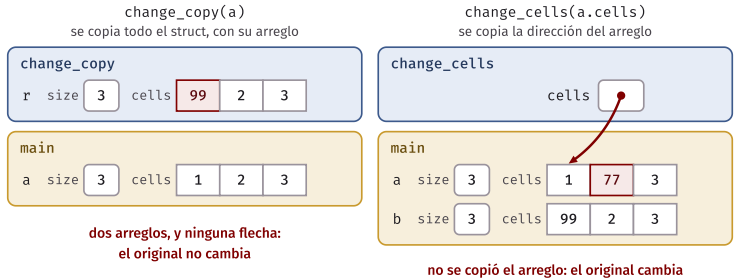
¿Qué pasa con `a` en cada llamada?

terminal

```
$ build/struct_copy
en change_copy: cells[0] = 99
en main: cells[0] = 1
a: 1, b: 99
a.cells[1] = 77
```

- `change_copy(a)`: el argumento es el `struct`. Se copia completo.
- `change_cells(a.cells)`: el argumento es un arreglo. Se copia la dirección.

Dos llamadas parecidas: ¿qué se copia?



Error común

"Los arreglos se pasan por referencia". C no pasa nada por referencia: copia. Pregúntense **qué** se copia.

SECCIÓN 06

Imágenes y filtros

06

PGM de texto

terminal

```
$ cat lab/images/tiny.pgm
P2
# Tres columnas y dos filas, con un comentario.
3 2
255
0 10 20
30 40 50
```

`P2`	Es PGM de texto
`# ...`	Comentario: se ignora
`3 2`	Ancho y alto
`255`	El valor del blanco
`0 10 ...`	Píxeles, fila por fila

Una imagen en escala de grises es una **matriz de números**: 0 es negro, y el máximo es blanco. Los saltos de línea no importan. Las filas salen del ancho.

El módulo: `struct image`

lab/image.h

```c
#define IMAGE_MAX_WIDTH 64
#define IMAGE_MAX_HEIGHT 64
#define IMAGE_MAX_VALUE 255

struct image {
    int width;
    int height;
    int max_value;
    int pixels[IMAGE_MAX_HEIGHT][IMAGE_MAX_WIDTH];
};
```

- El tamaño se reserva completo, aunque la imagen sea de 3 x 2.
- Se usa `pixels[y][x]`: primero la fila.
- Sin punteros, ¿cómo avisa `image_load` que falló? Devuelve una imagen **inválida**, con `width` en 0.
- Cómo se arma un programa en varios archivos: semana 6. Por ahora, el `Makefile` lo hace.

La interfaz del módulo

Función	Qué hace
`image_new(width, height, max_value, value)`	Imagen con todos sus píxeles en `value`
`image_from_values(width, height, max_value, values)`	Imagen desde un arreglo
`image_is_valid(img)`	1 si es válida, 0 si no
`image_equal(a, b)`	1 si son iguales
`image_load(name)`	Lee un archivo PGM
`image_save(img, name)`	Escribe un archivo. 1 si pudo, 0 si no
`image_print(img)`	Escribe los números
`image_show(img)`	Dibuja con letras en la terminal

Predigan con el modelo

image_demo.c

```c
void paint_copy(struct image img) {
    img.pixels[0][0] = 99;
...
void paint_row(int row[], int width) {
...
        row[x] = 99;
...
paint_copy(img);
...
paint_row(img.pixels[0], img.width);
```

terminal

```
$ build/image_demo
sizeof(struct image) = 16396 bytes
Después de paint_copy:
3 x 2, max 255
  0  10  20
 30  40  50
Después de paint_row:
3 x 2, max 255
 99  99  99
 30  40  50
```

`img.pixels[0]` es una fila: un arreglo de `int`.

El costo: cada `struct image` que se pasa o se devuelve copia 16396 bytes en este entorno. La próxima semana: punteros.

Un filtro: recibe copia, devuelve imagen

lab/filters.c

```c
struct image brighten(struct image img, int amount) {
    for (int y = 0; y < img.height; y++) {
        for (int x = 0; x < img.width; x++) {
            int value = img.pixels[y][x] + amount;
            if (value < 0) {
                value = 0;
            }
            if (value > img.max_value) {
                value = img.max_value;
            }
            img.pixels[y][x] = value;
        }
    }
    return img;
}
```

- `img` es una copia propia: se cambia y se devuelve.
- La imagen de quien llama queda intacta.
- Ciclos hasta `img.height` e `img.width`, no hasta el máximo.
- Se usa `max_value`, no un 255 a mano.

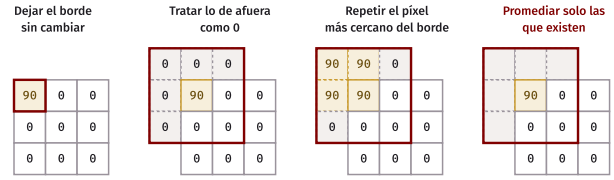
El desenfoque: promedio de una ventana

pseudocódigo

```
algoritmo desenfocar(img, radio)
    resultado ← una copia de img
    para y desde 0 hasta alto - 1 hacer
        para x desde 0 hasta ancho - 1 hacer
            resultado[y][x] ← promedio de img en la
     ventana
        fin para
    fin para
    devolver resultado
fin algoritmo
```

- La ventana es un cuadrado de lado `2 * radius + 1`.
- Debe leer los valores originales: por eso escribe en otra imagen.
- ¿Y en el borde, donde parte de la ventana cae afuera?

El problema de los bordes



Ninguna es la única correcta. El laboratorio usa la última.

Error común

Con `x + 1` igual a `width`, el índice sigue dentro del arreglo de 64: nadie avisa, y se lee un 0 que no es de la imagen. Las herramientas protegen el espacio reservado, no la imagen.

SECCIÓN 07

Laboratorio

07

Laboratorio de esta semana

Sesión 1

1. Dibujar los marcos de tres programas, **primero en papel**.
2. Comprobar en `gdb`: `break`, `backtrace`, `frame`, `info locals`.
3. Leer el módulo de imágenes y responder las preguntas del `README`.

Sesión 2

4. Escribir `invert`, `threshold` y `flip_horizontal`, con pruebas.
5. Escribir `box_blur`, y pensar en los bordes.

Comandos

`make gdb`, `make lab` y `make test-lab`. Las imágenes de prueba, pequeñas y no cuadradas: de 3 x 2, por ejemplo.

Para la próxima clase

- Terminen los filtros: `make test-lab` debe pasar.
- Sepan responder sin mirar:
  1. Si `main` llama a `f` y `f` llama a `g`, ¿qué marcos existen dentro de `g`?
  2. ¿Qué diferencia hay entre alcance y tiempo de vida en una local `static`?
  3. ¿Por qué `void inc(int x) { x++; }` no cambia la variable de quien llama?
  4. ¿Por qué una función sí puede cambiar los elementos de un arreglo que recibe?

Semana 4

Punteros: esa flecha que dibujamos hoy tendrá nombre y tipo, y los filtros dejarán de copiar 16 KB en cada llamada.