

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Programación

Punteros

Paso por puntero, arreglos y punteros, `const`

Gerson Garrido

Semana 4

Hoy

1. Una variable que guarda una dirección: `&` y `*`
2. Paso por puntero y parámetros de salida
3. Arreglos y punteros: `a[i]` es `*(a + i)`
4. Punteros a `struct` y `->`
5. `const`, cadenas y `FILE *`
6. El error clásico: devolver la dirección de una local
7. Imágenes por puntero

> **Lectura**
>
> Apuntes de la semana 4. Código en `code/week-04`.

SECCIÓN 01

Una variable que guarda una dirección

01



La flecha de la semana pasada

Tres problemas pendientes

- Una función no puede cambiar un `int` de quien la llama.
- `return` entrega un solo valor.
- Cada filtro copia 16396 bytes para mirar 6 píxeles.

Recetas por abrir

- El `&` de `scanf`.
- El `FILE *` de `fopen`.
- El `const char s[]` de las cadenas.

Lo que no cambia

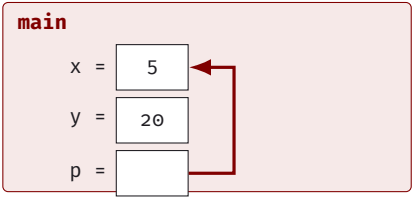
El modelo de entornos sigue igual: C copia cada argumento, y un puntero es un valor más. Lo nuevo es que, desde una casilla, se puede llegar a otra.

Puntero

Puntero ***(pointer)***
Variable que guarda una dirección de memoria. `int *` se lee “dirección de un `int`”.

pointer_basics.c

```c
int x = 5;
int y = 20;
int *p = &x;
```

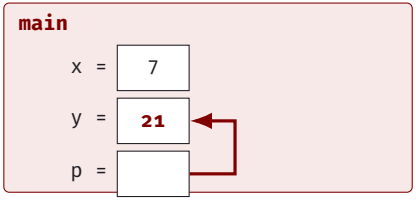


- `&x` da la dirección de `x`. `*p` sigue el puntero: es la casilla a la que apunta.
- Casi nunca importa el número, sino a qué casilla lleva. Por eso se dibuja como flecha.

¿Qué cambia cada asignación?

pointer_basics.c

```c
int *p = &x;
*p = 7;
p = &y;
*p = *p + 1;
```



`*p = *p + 1` lee y escribe en `y`.

La distinción de todo el capítulo

`p = ...` cambia la casilla de `p`. `*p = ...` cambia la casilla a la que apunta.

La salida real

terminal

```
$ build/pointer_basics
x = 5, *p = 5
x = 7, *p = 7
x = 7, y = 21, *p = 21
&x = 0xffffb5e00030
&y = 0xffffb5e00040
 p = 0xffffb5e00040
sizeof(p) = 8, sizeof(*p) = 4
```

- Las direcciones dependen del entorno y de la corrida. Lo que debe coincidir: `p` vale lo mismo que `&y`.
- `%p` siempre va con `(void *)`, como en Introducción a la Computación.
- `p` ocupa 8 bytes y `*p` ocupa 4, en este entorno. El lenguaje no fija ninguno de los dos.

Los tres significados de `*`

Dónde	Ejemplo	Significado
En una declaración	`int *p`	`p` es un puntero a `int`
En una expresión	`*p = 7`	Seguir el puntero
Entre dos valores	`a * b`	Multiplicar

`&` y `*` son operaciones contrarias: si `p` vale `&x`, entonces `*p` es `x`.

gdb confirma la flecha

gdb

```
Breakpoint 1, main () at pointer_basics.c:11
11      *p = 7;
(gdb) print x
$1 = 5
(gdb) print &x
$2 = (int *) 0xfffffffff9f4
(gdb) print p
$3 = (int *) 0xfffffffff9f4
(gdb) print *p
$4 = 5
(gdb) whatis p
type = int *
(gdb) next
12      printf("x = %d, *p = %d\n", x, *p);
(gdb) print x
$5 = 7
```

- `print p` y `print &x` dan el mismo número. Eso significa la flecha.
- Después de `*p = 7`, `x` vale 7.
- Compilen con `-g -O0`, sin sanitizers.

Las direcciones dependen del entorno.

Un puntero tiene un tipo

El tipo dice qué hay en esa dirección: cuántos bytes leer y cómo interpretarlos. Guardar `&d`, con `d` de tipo `double`, en un `int *` no compila:

```
error: initialization of 'int *' from incompatible pointer type 'double *'
```

En `errors/wrong_type.c`. Los apuntes traen el mensaje completo.

Error común

`int *p, q;` declara un puntero `p` y un `int q`. El asterisco pertenece a cada nombre. Una declaración por línea.

`NULL`: ninguna dirección

El **puntero nulo** *(null pointer)* se puede guardar y comparar. No se puede seguir: es comportamiento indefinido.

En `errors/null_deref.c`, `gcc` no dice nada ante `*p = 3`. El error aparece al correr:

terminal, recortado

```
$ build/null_deref_asan 2>&1 | grep -E "runtime error|ERROR|#0"
errors/null_deref.c:8:8: runtime error: store to null pointer of type 'int'
==35==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 ...
    #0 0xaaaacb0b0dd0 in main errors/null_deref.c:8
```

Idea clave

Si no pueden dibujar la flecha, no pueden seguirla. Un puntero sin inicializar vale cualquier cosa.

SECCIÓN 02

Paso por puntero

02

El intento que no funciona

swap.c

```c
void swap_values(int a, int b) {
    int old_a = a;
    a = b;
    b = old_a;
}
...
swap_values(x, y);
```

swap_values	a=2 b=1 old_a=1
main	x=1 y=2

La regla 1 copió los valores. La función intercambió **sus casillas**. Al retornar, el marco desaparece.

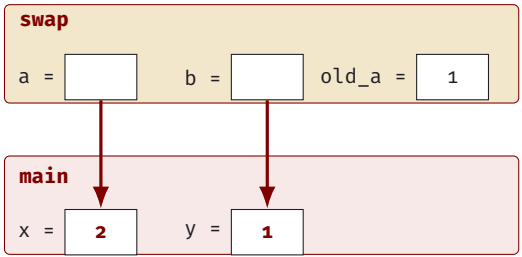
terminal

```
$ build/swap
después de swap_values: x = 1, y = 2
```

Pasar dónde están

swap.c

```c
void swap(int *a, int *b) {
    int old_a = *a;
    *a = *b;
    *b = old_a;
}
...
swap(&x, &y);
```



`*b = old_a` escribe en `y`. Al retornar, el marco se va, y los cambios quedan.

Paso por puntero: pasar por valor una dirección. La llamada avisa que `x` e `y` pueden cambiar.

a y &x: el mismo número, en dos marcos

gdb

```
Breakpoint 1, swap (a=0xffffffffa0c, b=0xffffffffa08) at swap.c:11
11      int old_a = *a;
(gdb) backtrace
#0  swap (a=0xffffffffa0c, b=0xffffffffa08) at swap.c:11
#1  0x0000aaaaaaaa0864 in main () at swap.c:23
(gdb) print *a
$1 = 1
(gdb) finish
Run till exit from #0  swap (a=0xffffffffa0c, b=0xffffffffa08) at swap.c:11
main () at swap.c:24
24      printf("después de swap: x = %d, y = %d\n", x, y);
(gdb) info locals
x = 2
y = 1
(gdb) print &x
$3 = (int *) 0xffffffffa0c
```

- `gdb` muestra los parámetros como direcciones.
- Son los mismos valores de `&x` y `&y` en `main`.
- `finish` corre hasta que la función retorna.

Olvidar el &, y asignar al parámetro

terminal, recortado

```
$ gcc -Wall -Wextra -std=c17 errors/missing_ampersand.c -o build/missing_ampersand
errors/missing_ampersand.c:14:10: error: passing argument 1 of 'swap' makes pointer from integer ...
   14 |     swap(x, y);
errors/missing_ampersand.c:4:16: note: expected 'int *' but argument is of type 'int'
```

El mensaje dice qué esperaba, `int *`, y qué recibió, `int`.

Error común

`p = otra_direccion;` dentro de una función solo cambia la copia de la flecha. Quien llama conserva su puntero. Para cambiar la casilla de llegada se escribe con `*`.

`*p++`: `++` se aplica antes que `*`

errors/star_increment.c

```c
void inc(int *p) {
    *p++;
}
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/star_increment.c -o build/star_increment
errors/star_increment.c:5:5: warning: value computed is not used [-Wunused-value]
    5 |     *p++;
$ build/star_increment
x = 7
```

Lo que avanzó fue el puntero, que es una copia. Lo correcto: `(*p)++` o `*p = *p + 1`. Esa advertencia dice que el programa no hace lo que parece.

Parámetros de salida

Un **parámetro de salida** *(output parameter)* sirve para escribir un resultado.

out_params.c

```c
int divide(int dividend, int divisor, int *quotient, int *remainder) {
    ...
    *quotient = dividend / divisor;
    *remainder = dividend % divisor;
    return 1;
}
```

```c
if (divide(17, 5, &q, &r)) {
    printf("17 / 5: cociente %d, resto %d\n
      ", q, r);
}
```

terminal

```
$ echo 8 | build/out_params
17 / 5: cociente 3, resto 2
17 / 0: no se puede, y q sigue en -1
```

`scanf` ya era esto

```c
if (scanf("%d", &n) != 1) {
```

- `&n` es un **parámetro de salida**: dice dónde escribir.
- Su `return` dice cuántos valores pudo leer.
- `scanf("%19s", word)` no lleva `&`: `word` es un arreglo, y ya da una dirección.

La convención del curso para funciones que pueden fallar

El `return` dice si funcionó: 1 o 0. Los datos salen por parámetros de salida, y si falla no escribe en ellos. `scanf` no cumple todo: devuelve una cuenta, y puede escribir una parte.

SECCIÓN 03

Arreglos y punteros

03



Un arreglo llega como puntero a su primer elemento

En casi toda expresión, el nombre de un arreglo da un puntero a su primer elemento: `a` es `&a[0]`.

Parámetro	Qué es en realidad
`int values[]`	`int *values`
`const int a[]`	`const int *a`
`const char name[]`	`const char *name`

No es una costumbre: la primera forma es otra manera de escribir la segunda. Por eso la longitud sigue haciendo falta.

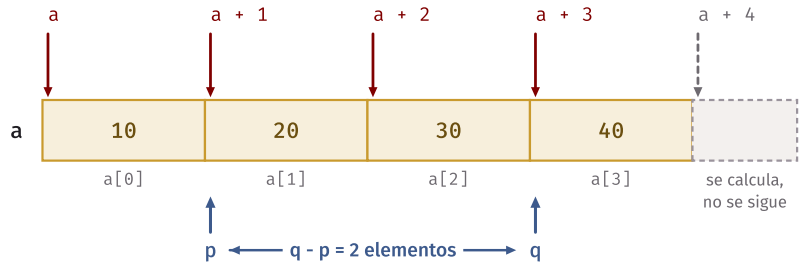
Aritmética de punteros

Aritmética de punteros *(pointer arithmetic)*

`p + 1` apunta al elemento siguiente. El "+ 1" es un elemento, no un byte.

- El lenguaje **define** `a[i]` como `*(a + i)`. Los corchetes son esa cuenta.
- `*(a + 2)` es el elemento de índice 2. Avanza `2 * sizeof(int)` bytes: 8 en este entorno.

`a + i`, y la resta de dos punteros



Cada casilla mide 4 bytes en este entorno: a + 1 está 4 bytes después de a, y q - p no da 8 sino 2.

Lo que escribe el programa

array_pointer.c

```c
printf("a[2] = %d, *(a + 2) = %d\n", a[2], *(
    a + 2));
printf("a    = %p\n", (void *) a);
printf("a + 1 = %p\n", (void *) (a + 1));
printf("a + 2 = %p\n", (void *) (a + 2));

int *p = a + 1;
printf("*p = %d, p[0] = %d, p[1] = %d\n", *p
    , p[0], p[1]);

int *q = &a[3];
printf("q - p = %td elementos\n", q - p);
```

terminal

```
$ build/array_pointer
a[2] = 30, *(a + 2) = 30
a    = 0xffff96100020
a + 1 = 0xffff96100024
a + 2 = 0xffff96100028
*p = 20, p[0] = 20, p[1] = 30
q - p = 2 elementos
```

Las direcciones dependen de la corrida. La distancia no.

- Un puntero también admite corchetes, y los índices cuentan desde donde esté.
- La resta de dos punteros del mismo arreglo da **elementos**, no bytes. Se escribe con `%td`.
- `*(a + 4)` es tan inválido como `a[4]`: son lo mismo.

Recorrer con un puntero, y pasar una parte

array_pointer.c

```c
int sum_with_pointer(const int *a, int n) {
    int total = 0;
    for (const int *p = a; p < a + n; p++) {
        total = total + *p;
    }
    return total;
}
```

- `a + n` es justo después del último: se puede calcular y comparar, no seguir.

```c
printf("desde a[2]: %d\n", sum_with_pointer(a + 2, 2));
```

- Para la función, `a + 2` es el comienzo de un arreglo de 2 elementos. El resultado es 70.

Un arreglo no es un puntero

	`int a[4]`	`int *p = a;`
¿Qué hay en el marco?	Los 4 elementos	Una casilla con una dirección
`sizeof`, en este entorno	16	8
¿Se puede asignar?	No	Sí: `p = otra_direccion;`
`x[i]`	`*(a + i)`	`*(p + i)`

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/array_assign.c -o build/array_assign
errors/array_assign.c:8:7: error: assignment to expression with array type
    8 |     a = b;
```

SECCIÓN 04

Punteros a `struct`

04



El operador ->

struct_pointer.c

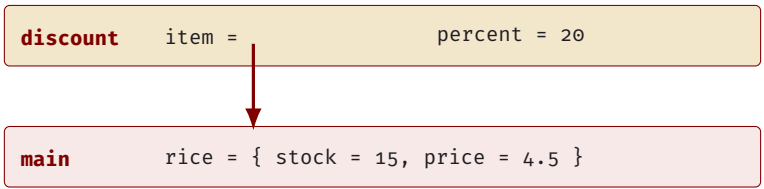
```
1  void restock(struct product *item, int
       amount) {
2      (*item).stock = (*item).stock + amount;
3  }
4
5  void discount(struct product *item, double
       percent) {
6      item->price = item->price * (100 -
       percent) / 100;
7  }
```

- Primero se sigue el puntero. Luego, el campo.
- `p->campo` es **exactamente** `(*p).campo`.
- A la izquierda de `.` va un `struct`, y de `->`, un puntero.

Sin paréntesis, `*p.x` no compila, y `gcc` da la solución:

```
error: 'p' is a pointer; did you mean to use '->'?
```

Tres maneras de pasar lo mismo



Se pasa	Se copia	¿Puede cambiar el original?
`f(rice)`	Todo el `struct`	No
`f(rice.stock)`	Un `int`	No
`f(&rice)`	Una dirección	Sí

SECCIÓN 05

`const`, cadenas y `FILE *`

05



Dos cosas que podrían no cambiar

Con un puntero, la casilla a la que apunta y el puntero mismo. La posición de `const` dice cuál.

Declaración	`*p = 1;`	`p = &b;`	**Para qué**
`int *p`	Sí	Sí	Puntero normal
`const int *p`	No	Sí	Una función que solo lee
`int *const p`	Sí	No	Una flecha que no se mueve
`const int *const p`	No	No	Ninguna de las dos

Se lee de derecha a izquierda desde el nombre. `const int *p`: `p` es un puntero a un `int` constante. `int *const p`: `p` es constante, y es un puntero a un `int`.

Cada intento da un error distinto

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/const_location.c -o build/const_location
errors/const_location.c:6:8: error: assignment of read-only location '*p'
    6 |     *p = 2;
$ gcc -Wall -Wextra -std=c17 errors/const_variable.c -o build/const_variable
errors/const_variable.c:7:7: error: assignment of read-only variable 'p'
    7 |     p = &b;
```

La regla del curso

Si una función no cambia lo que hay detrás de un puntero, ese parámetro lleva `const`.

La promesa es sobre el camino

const_pointers.c

```c
const int *to_const = &a;   // no se escribe a través de él, pero puede apuntar a otro
to_const = &b;
...
b = 20;                     // const promete por el puntero, no por la variable
printf("*to_const = %d\n", *to_const);
```

terminal

```
$ build/const_pointers
*to_const = 2
*to_const = 20
```

Un `int *` se pasa sin problema donde se espera un `const int *`: agregar una promesa no molesta a nadie. Lo contrario pierde la promesa, y `gcc` lo advierte.

`const char *`: cadenas

text_pointers.c

```c
int length(const char *s) {
    const char *p = s;
    while (*p != '\0') {
        p++;
    }
    return (int) (p - s);
}
```

```c
const char *email = "ana@example.com";
...
const char *at = strchr(email, '@');
if (at != NULL) {
    printf("dominio: %s\n", at + 1);
```

terminal

```
$ build/text_pointers
ana@example.com tiene 15 bytes
dominio: example.com
```

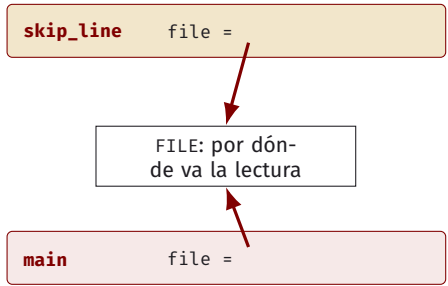
`strchr` devuelve un puntero, y ahora se guarda. Un literal lleva `const char *`.

`FILE *`, sin receta

file_pointer.c

```c
int skip_line(FILE *file) {
    int c = fgetc(file);
    while (c != '\n' && c != EOF) {
        c = fgetc(file);
    }
    return c == '\n';
}
```

- Se usa con las funciones de la biblioteca. Nunca se escribe `*file`.
- El `FILE` sobrevive a `fopen`, hasta `fclose`. La semana 5 explica cómo.



Se copia la dirección, pero la **posición** está en el `FILE`, y hay uno solo. Por eso, al volver, `main` sigue desde donde quedó.

SECCIÓN 06

Devolver la dirección de una local

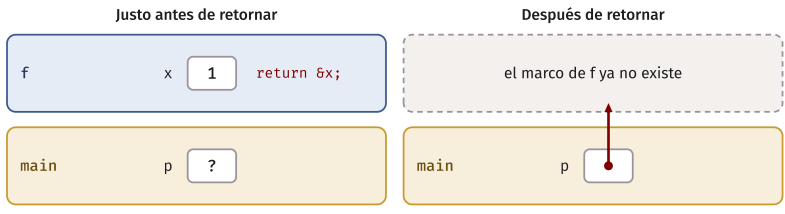
06

El error clásico, con marcos

errors/return_local.c

```c
int *f(void) {
    int x = 1;
    return &x;
}
```

La regla 4: se calcula el valor, que es la dirección de x, y **el marco desaparece**. Queda un puntero **colgante**.



`gcc` lo avisa, si es evidente

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/return_local.c -o build/return_local
errors/return_local.c:6:12: warning: function returns address of local variable [-Wreturn-local-addr]
```

Compila. En este entorno, al correr muere: `gcc` hizo que `f` devolviera un puntero nulo.

El caso peligroso: compila sin una advertencia

errors/dangling.c

```c
int *make_value(void) {
    int value = 42;
    int *p = &value;
    return p;
}
```

```c
int *p = make_value();
printf("primero: %d\n", *p);
other_call();
printf("después: %d\n", *p);
```

terminal, en esta corrida

```
$ build/dangling
primero: 42
después: 65535
```

- Primero dio el resultado correcto. El valor no se borra al morir la variable.
- Luego otras llamadas usaron ese espacio.
- En su máquina puede salir otro número. Es comportamiento indefinido.

ASan lo atrapa en el primer acceso

terminal, recortado

```
$ build/dangling_asan 2>&1 | grep -E "ERROR|^READ|#0|variable"
==21==ERROR: AddressSanitizer: stack-use-after-return on address 0xffffbbf00020 ...
READ of size 4 at 0xffffbbf00020 thread T0
    #0 0xaaaae22711d8 in main errors/dangling.c:18
    #0 0xaaaae2270fb4 in make_value errors/dangling.c:4
    [32, 36) 'value' (line 5) <== Memory access at offset 32 is inside this variable
```

- `stack-use-after-return`: uso de la pila después de retornar.
- El primer `#0` es dónde ocurrió, y el segundo, el marco al que pertenece esa memoria.

Idea clave

Un puntero no debe vivir más que su casilla. Un marco reciente puede usar direcciones de uno antiguo, como `swap`. Al revés, no.

Tres soluciones correctas

Solución	Ejemplo	Por qué funciona
Devolver el valor	`int f(void)`	Se copia antes de que el marco desaparezca
Recibir la dirección de quien llama	`void f(int *result)`	La casilla está en un marco que sigue vivo
Devolver una dirección recibida	`strchr`	Esa cadena ya existía antes de la llamada

La cuarta, memoria que no pertenece a ningún marco, es `malloc`: la próxima semana.

SECCIÓN 07

Imágenes por puntero

07

16396 bytes contra 8

image_demo.c

```c
void paint_copy(struct image img) {
    img.pixels[0][0] = 99;
...
void paint_first(struct image *img) {
    img->pixels[0][0] = 99;
```

- Se copian 8 bytes en este entorno, sea cual sea el tamaño de la imagen.
- Y **ahora el original puede cambiar**.

terminal

```
$ build/image_demo
sizeof(img) = 16396, sizeof(&img) = 8
3 x 2, max 255
  0  10  20
 30  40  50
Después de paint_copy:
3 x 2, max 255
  0  10  20
 30  40  50
Después de paint_first:
3 x 2, max 255
 99  10  20
 30  40  50
```

La nueva interfaz

Semana 3	Semana 4
`struct image image_load(name)`	`int image_load(struct image *img, const char *name)`
`int image_save(img, name)`	`int image_save(const struct image *img, const char *name)`
`struct image image_new(width, ...)`	`int image_new(struct image *img, int width, ...)`
`int image_is_valid(img)`	Ya no hace falta
`void image_print(img)`	`void image_print(const struct image *img)`

- El `const` dice quién cambia la imagen, sin abrir `image.c`.
- `*img` es un parámetro de salida, y el `return` dice si funcionó. Ya no hace falta `width` igual a 0.
- Si falla, `*img` no cambia.

Cómo cumple `image_load` su promesa

lab/image.c

```c
int image_load(struct image *img, const char *name
    ) {
    FILE *file = fopen(name, "r");
    if (file == NULL) {
        return 0;
    }
    struct image loaded;
    int ok = read_image(&loaded, file);
    fclose(file);
    if (ok) {
        *img = loaded;
    }
    return ok;
}
```

- `loaded` es una local. Si el archivo está mal a medias, `*img` no se tocó.
- `&loaded` se usa **mientras su marco existe**. El error era devolverla.
- `*img = loaded` copia todo. Un puntero no prohíbe las copias necesarias.

Error común: `struct image *img;` es una flecha sin destino. Declaren `struct image img;` y pasen `&img`.

Filtros que cambian la imagen

lab/filters.c

```c
void 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;
            ...
            img->pixels[y][x] = value;
        }
    }
}
```

	Por valor	Por puntero
Se copia	Todo	8 bytes
El original	Intacto	Cambia
Componer	Anidando	Línea por línea

lab/filter_demo.c

```c
struct image copy = img;
invert(&copy);
```

Para conservar el original.

Varios resultados: `image_stats`

lab/filters.h

```
1  void image_stats(const struct image *img, int *min, int *max, double *mean);
```

- ¿Con qué valor empiezan el mínimo y el máximo?
- El promedio tiene decimales. ¿Qué pasa con la división entre dos `int`?
- Solo cuentan los píxeles en uso, no todo el espacio reservado.

Cómo probarla

Pongan el extremo en la primera casilla, en la última, y prueben con todos los píxeles iguales y con un promedio que no sea entero.

SECCIÓN 08

Laboratorio

08

Laboratorio de esta semana

Parte 1: dibujar punteros (unos 35 min)

1. Tres programas: **primero en papel**.
2. Comprobar en `gdb` con `print p`, `print &x`, `up`.

Parte 2: filtros (unos 55 min)

3. Reescribir `invert`, `threshold` y `flip_horizontal` por puntero, con pruebas.
4. Escribir `image_stats`, con pruebas.

Comandos

`make gdb`, `make lab` y `make test-lab`. Esta semana una sesión es de laboratorio, y la otra es para la Práctica 1.

Para la próxima clase

- Terminen los filtros: `make test-lab` debe pasar.
- Sepan responder sin mirar:
  1. Dibujen la memoria después de `int x = 5; int *p = &x; *p = 7;`. ¿Cuánto vale `x`?
  2. Expliquen con marcos el error de `int *f(void) { int x = 1; return &x; }`.
  3. ¿Qué diferencia hay entre `const int *p` e `int *const p`?
  4. Si `a` es un arreglo de `int`, ¿qué significa `*(a + 2)`? ¿Cuántos bytes avanza?

Semana 5

Memoria dinámica: imágenes de cualquier tamaño, datos que sobreviven a la función, y la cuarta solución al error de hoy.