

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Programación

Datos en C y cómo leer código ajeno

Arreglos, cadenas, `struct` y archivos · Un método para leer programas

Gerson Garrido

Semana 2

Hoy

1. Repaso rápido: arreglos, cadenas, `struct` y archivos
2. Leer código es una habilidad: un método de cuatro pasos
3. Leer las pruebas como documentación
4. Corregir y extender cambiando poco

> **Lectura**
>
> Apuntes de la semana 2. Código en `code/week-02`. Lo que se olvidó del todo está en Introducción a la Computación: semanas 9, 10, 11 y 12.

SECCIÓN 01

Repaso: los datos

01

Un arreglo tiene dos números



- **Longitud:** cuántos tiene ahora. Cambia, y va en una variable.
- **Capacidad:** cuántos caben. No cambia.
- Antes de escribir, se revisa que quepa.

Pasar un arreglo a una función

array_review.c

```c
int sum(const int a[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total = total + a[i];
    }
    return total;
}
void add_bonus(int a[], int n, int bonus) {
    for (int i = 0; i < n; i++) {
        a[i] = a[i] + bonus;
    }
}
```

terminal

```
$ build/array_review
Suma: 51
Suma de las 2 primeras: 23
Con un punto más: 55
```

- La longitud va aparte: se puede sumar una parte.
- `const`: solo lee. Sin `const`: puede cambiarlo.
- No se copia: viaja la **dirección**.

¿Por qué hay que pasar el tamaño?

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/sizeof_param.c -o build/sizeof_param
errors/sizeof_param.c:5:48: warning: 'sizeof' on array function parameter 'a' will return size of 'int *' [-Wsizeof-array-argument]
$ build/sizeof_param
en main: 32 bytes
en la función: 8 bytes
```

- 8 bytes es el tamaño de una dirección en este entorno. No dice nada del arreglo.
- La función recibe solo dónde empieza: sin un parámetro, no sabe dónde termina.

Error común

`sizeof(a) / sizeof(a[0])` solo sirve donde se declaró el arreglo, no en una función que lo recibe.

Salirse del arreglo

errors/array_overflow.c

```c
for (int i = 0; i <= 4; i++) {
    total = total + grades[i];
}
```

- Compila sin advertencias. Sin sanitizers, corre y escribe un número.
- **Comportamiento indefinido** *(undefined behavior)*: el estándar no exige ningún resultado.

terminal, las direcciones cambian en cada corrida

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/array_overflow.c -o build/array_overflow_san
$ build/array_overflow_san
errors/array_overflow.c:8:31: runtime error: index 4 out of bounds for type 'int [4]'
READ of size 4 at 0xffffb3700030 thread T0
    #0 0xaaaae7e71474 in main errors/array_overflow.c:8
SUMMARY: AddressSanitizer: stack-buffer-overflow errors/array_overflow.c:8 in main
```

Una cadena termina en `'\0'`

`strlen` = 4, hasta el primer nulo

`sizeof` = 8

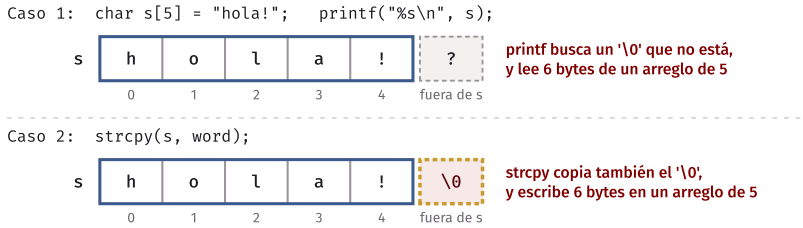


terminal

```
$ build/string_review
sizeof = 8, strlen = 4
Con un nulo en medio: "ho", strlen = 2
strcmp(a, b) = 0
```

- Un texto de `k` bytes necesita `k + 1`.
- Sin `'\0'` no es una cadena: es un arreglo de `char`.
- Para comparar texto: `strcmp`, que da 0 si son iguales. `==` no compara el texto.

¿Qué pasa si `char s[5]` recibe `"hola!"`?



Caso 1: al inicializar, `char s[5] = "hola!";`

- C lo acepta: llena 5 casillas y deja afuera el `'\0'`.
- El problema llega después, cuando `%s` busca un nulo que no está.

Caso 2: al copiar, `strcpy(s, word);`

- Copia hasta el `'\0'` del origen: escribe 6 bytes en 5.
- Es un **buffer overflow**.

Caso 1: lee de más. Caso 2: escribe de más

caso 1: errors/no_terminator.c, las direcciones cambian en cada corrida

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/no_terminator.c -o build/no_terminator_san
$ build/no_terminator_san
READ of size 6 at 0xffff97b00025 thread T0
    #1 0xaaaaac07b0d88 in main errors/no_terminator.c:6
    [32, 37) 's' (line 5) <== Memory access at offset 37 overflows this variable
```

caso 2: errors/copy_overflow.c, las direcciones cambian en cada corrida

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/copy_overflow.c -o build/copy_overflow_san
$ build/copy_overflow_san
WRITE of size 6 at 0xffff7d800025 thread T0
    #1 0xaaaaca960e94 in main errors/copy_overflow.c:8
    [32, 37) 's' (line 6) <== Memory access at offset 37 overflows this variable
```

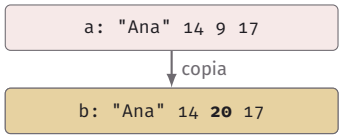
- `gcc` no avisó en ninguno, en esta versión. Sin sanitizers, en esta corrida, parecen funcionar.
- `scanf("%s", s)` es como `strcpy`. Por eso `%s` lleva un límite: `%19s` para 20 bytes.

Un `struct` se copia entero

struct_review.c

```c
struct student {
    char name[NAME_SIZE];
    int grades[N_GRADES];
};

struct student a = {"Ana", {14, 9, 17}};
struct student b = a;
b.grades[1] = 20;
```



- Se copian todos los campos, incluidos los arreglos que tiene adentro.
- Cambiar `b` no toca `a`.

Al pasarlo a una función: ¿qué se copia?

struct_review.c

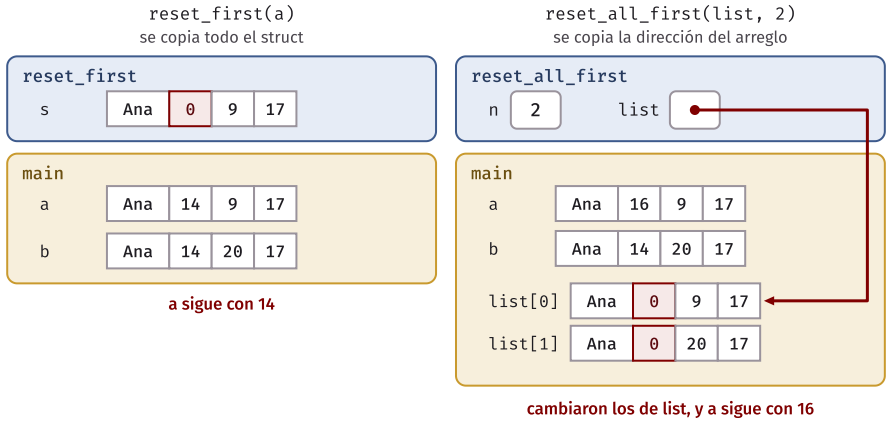
```c
void reset_first(struct student s) {
    s.grades[0] = 0;
    printf("En la función: %d\n", s.grades[0]);
}
struct student with_first(struct student s, int grade) {
    s.grades[0] = grade;
    return s;
}
void reset_all_first(struct student list[], int n) {
    for (int i = 0; i < n; i++) {
        list[i].grades[0] = 0;
    }
}
```

terminal

```
$ build/struct_review
Ana: 14 9 17
Ana: 14 20 17
En la función: 0
Ana: 14 9 17
Ana: 16 9 17
Ana: 0 9 17
Ana: 0 20 17
Ana: 16 9 17
```

- `reset_first`: cambia su copia. `a` sigue con 14.
- `with_first`: devuelve la copia cambiada, y `main` la guarda.
- `reset_all_first`: recibe un arreglo, así que cambia los de `list`.

Copia o dirección



Un archivo: abrir, leer, cerrar

file_review.c

```c
FILE *file = fopen(path, "r");
if (file == NULL) {
    perror(path);
    return 1;
}
while (fgets(line, LINE_SIZE, file) != NULL) {
    int fields = sscanf(line, "%19s %d %1s", name, &
      people, extra);
    if (fields == 2) {
        total = total + people;
    } else {
        fprintf(stderr, "Línea %d: mal formada, se omite\
      n", number);
    }
}
fclose(file);
```

terminal

```
$ echo paraderos.txt | build/file_review
Línea 4: mal formada, se omite
Personas: 92
$ echo no_hay.txt | build/file_review
no_hay.txt: No such file or directory
```

- **Toda lectura revisa su resultado.**
- El ciclo depende de la lectura, no de `feof`.
- `%1s` detecta lo que sobra.

SECCIÓN 02

Cómo leer código

02



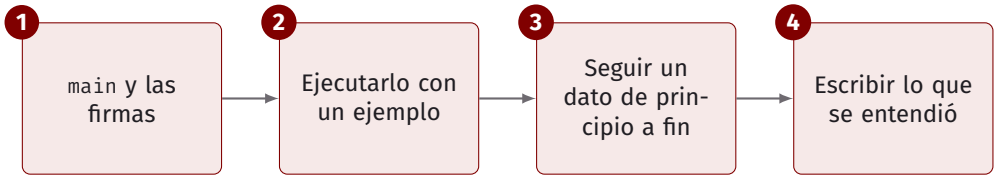
Leer código es una habilidad

- En un trabajo real, casi nunca se empieza de una hoja en blanco: se llega a un programa que ya existe.
- Casi siempre se lee más código del que se escribe.
- Lo que escribieron hace tres meses también les parecerá de otra persona.

Un programa no es un libro

No se ejecuta de la primera línea a la última, así que tampoco se entiende en ese orden. Si empiezan por arriba, se cansan con detalles de los que aún no saben para qué sirven.

El método



- **1.** Ver las piezas y cómo se conectan, sin leer cuerpos.
- **2.** Ver qué hace por fuera, antes de ver cómo lo hace por dentro.
- **3.** Un dato los lleva justo por el camino que importa.
- **4.** Lo que no se puede escribir, no se entendió.

Paso 1: medir y ver las piezas

Una empresa de combis anota sus viajes. `trips.c` resume cada ruta. Hagan como si no lo conocieran.

terminal

```
$ wc -l trips.c
128 trips.c
$ grep -n '^[a-z].*) {$' trips.c
20:int remove_newline(char s[]) {
31:int find_route(const struct route routes[], int n, const char name[]) {
41:int add_trip(struct route routes[], int n, const char name[], int passengers) {
60:int read_trips(const char path[], struct route routes[]) {
84:double average_passengers(struct route r) {
89:int busiest_route(const struct route routes[], int n) {
102:void print_routes(const struct route routes[], int n) {
109:int main(void) {
```

- `grep` escribe las líneas que contienen un patrón. Este patrón encuentra las definiciones de función, en nuestro estilo.
- Una **firma** da el nombre, los parámetros y lo que devuelve. Esta lista ya es un mapa.

Las firmas dicen mucho

Firma	Qué se deduce
`find_route(const struct route ...)`	Busca, y solo lee: lleva `const`
`add_trip(struct route routes[], ...)`	Cambia el arreglo. ¿El `int` es la nueva longitud?
`read_trips(const char path[], ...)`	Llena el arreglo desde un archivo
`average_passengers(struct route r)`	Recibe una copia: no puede cambiar nada
`print_routes(const struct route ...)`	Escribe, y solo lee los datos

Hipótesis

Lo que lleva signo de interrogación se anota, y se comprueba después.

Los datos, y luego `main`

trips.c

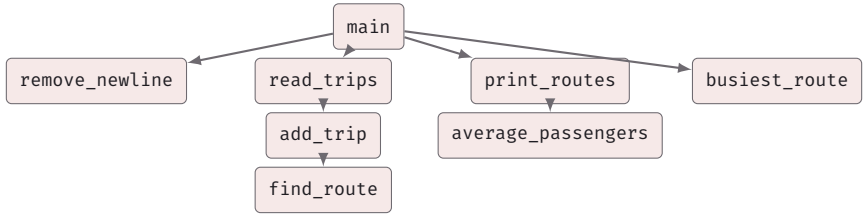
```c
struct route {
    char name[NAME_SIZE];
    int trips;
    int passengers;
};
```

- El tipo principal es lo más importante.
- El manejo de errores se lee una vez y se quita con la vista.
- Queda la historia, en tres llamadas.

trips.c

```c
int main(void) {
    ...
    remove_newline(path);
    struct route routes[MAX_ROUTES];
    int n = read_trips(path, routes);
    if (n < 0) {
        return 1;
    }
    if (n == 0) {
        ...
    }
    print_routes(routes, n);
    printf("Más pasajeros: %s\n", routes[busiest_route(
      routes, n)].name);
    return 0;
}
```

El árbol de llamadas



terminal

```
$ grep -n 'add_trip' trips.c
41:int add_trip(struct route routes[], int n, const char name[], int passengers) {
75:            n = add_trip(routes, n, name, passengers);
```

La línea 75 está dentro de `read_trips`, que empieza en la 60. Así se arma el árbol, de a pocos.

¿Y si son 2 000 líneas?

- No se leen de corrido. Nadie tiene 2 000 líneas en la cabeza a la vez.
- Empiecen por `main` y los tipos principales.
- Bajen solo por la rama que les importa.
- Usen `grep` para saltar: dónde se define algo, y quién lo llama.

> **Idea clave**
> Aprender qué función **no** necesitan leer es parte del método.

Paso 2: ejecutarlo con un ejemplo

terminal

```
$ cat viajes.txt
Chosica 18
Callao 25
Chosica 22
Ate 9
Callao 31
Chosica mucho
Ate 14
Callao 20
```

terminal

```
$ echo viajes.txt | build/trips
Línea 6: mal formada, se omite
Chosica  2 viajes   40 pasajeros  20.00 por viaje
Callao   3 viajes   76 pasajeros  25.33 por viaje
Ate      2 viajes   23 pasajeros  11.50 por viaje
Más pasajeros: Callao
```

- Agrupa por ruta: 8 líneas de datos se vuelven 3 filas.
- En el orden en que cada ruta apareció, no ordenadas.
- Una línea mala no lo detiene: avisa con su número y sigue.
- ¿“Más pasajeros” es el total o el promedio? Callao gana en los dos: este ejemplo no alcanza. Se anota.

Paso 3: seguir un dato. `Callao 31`

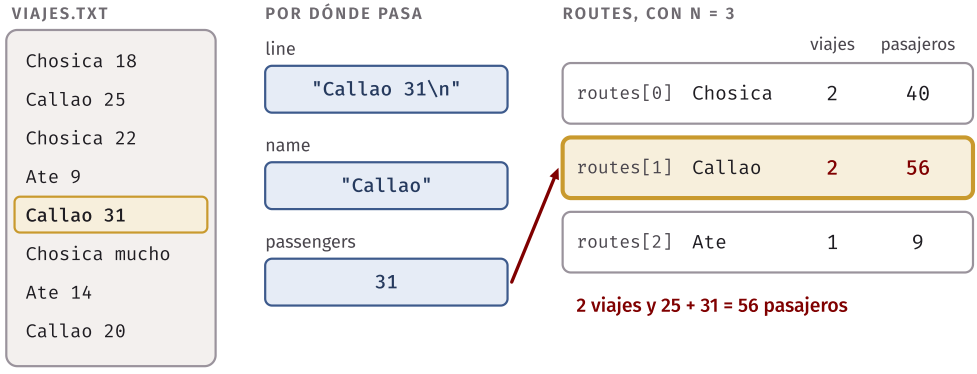
trips.c

```c
int add_trip(struct route routes[], int n, const char name
      [], int passengers) {
    int pos = find_route(routes, n, name);
    if (pos == -1) {
        ...
        pos = n;
        strcpy(routes[pos].name, name);
        routes[pos].trips = 0;
        routes[pos].passengers = 0;
        n = n + 1;
    }
    routes[pos].trips = routes[pos].trips + 1;
    routes[pos].passengers = routes[pos].passengers +
      passengers;
    return n;
}
```

Dónde	Valor
archivo	`Callao 31`
`fgets`	`"Callao 31\n"`
`sscanf`	`"Callao", 31`
`find_route`	devuelve `1`
`add_trip`	`Callao 2 56`

Elegimos la segunda de Callao: la ruta ya existe, y el dato pasa por más código.

`Callao 31`, paso a paso



add_trip le suma un viaje y 31 pasajeros a routes[1]

Cómo crece `routes`: a mano

Línea	`routes[0]`	`routes[1]`	`routes[2]`	`n`
`Chosica 18`	Chosica 1 18			1
`Callao 25`	Chosica 1 18	Callao 1 25		2
`Chosica 22`	Chosica 2 40	Callao 1 25		2
`Ate 9`	Chosica 2 40	Callao 1 25	Ate 1 9	3
`Callao 31`	Chosica 2 40	Callao 2 56	Ate 1 9	3
`Chosica mucho`	no se llama a `add_trip`			3
`Ate 14`	Chosica 2 40	Callao 2 56	Ate 2 23	3
`Callao 20`	Chosica 2 40	Callao 3 76	Ate 2 23	3

La última fila coincide con la salida. Cuando la mano y la máquina coinciden, entienden el programa. Y se comprobó la hipótesis: `add_trip` devuelve la nueva longitud.

Una regla que nadie escribió

trips.c

```c
double average_passengers(struct route r) {
    return (double) r.passengers / r.trips;
}
```

- Divide entre `r.trips`. ¿Puede ser 0?
- En `add_trip`, una ruta nace con 0 viajes, y dos líneas después ya tiene 1.
- Es segura **por cómo se llenan los datos**, no por algo que diga esta función.

> **Idea clave**
>
> Hay reglas que un programa cumple sin escribirlas. Romper una sin darse cuenta es la forma más común de malograrlo al cambiarlo.

Paso 4: escribir lo que entendieron

```
trips.c: resume por ruta los viajes de un archivo.

Datos
route: nombre, viajes y total de pasajeros.
routes[MAX_ROUTES], longitud n. Cada ruta, una vez.
Regla: toda ruta guardada tiene por lo menos un viaje.

Funciones
find_route   busca por nombre; -1 si no está
add_trip     cuenta un viaje; devuelve la nueva n
...

Dudas
¿Una línea más larga que LINE_SIZE?
¿La novena ruta?
```

- Mientras se lee, todo parece claro. Al escribir, salen los huecos.
- Tres cosas: datos, funciones y dudas.
- Las dudas no son una falla: es lo que hay que mirar al cambiar el programa.
- No copien los comentarios.

SECCIÓN 03

Las pruebas como documentación

03



Una prueba se lee como una frase

test_trips.c

```c
void test_add_trip(void) {
    struct route routes[MAX_ROUTES];
    int n = 0;
    n = add_trip(routes, n, "Ate", 9);
    assert(n == 1);
    assert(routes[0].trips == 1);
    assert(routes[0].passengers == 9);
    n = add_trip(routes, n, "Ate", 14);
    assert(n == 1);
    assert(routes[0].trips == 2);
    assert(routes[0].passengers == 23);
    n = add_trip(routes, n, "Callao", 25);
    assert(n == 2);
}
```

- **Cómo se usa:** `n` empieza en 0, y se guarda lo que devuelve.
- **Qué hace:** una ruta repetida acumula y no aumenta `n`.
- En 14 líneas dice lo que nos tomó todo el paso 3.

Consejo

En un programa que no conocen, lean las pruebas **antes** que los cuerpos.

Responden lo que el ejemplo no dijo

test_trips.c

```c
// Una ruta con muchos viajes cortos puede ganarle a una con pocos viajes largos.
void test_busiest_counts_all_trips(void) {
    struct route routes[MAX_ROUTES];
    int n = add_trip(routes, 0, "Callao", 25);
    n = add_trip(routes, n, "Ate", 10);
    n = add_trip(routes, n, "Ate", 10);
    n = add_trip(routes, n, "Ate", 10);
    assert(busiest_route(routes, n) == 1);
}
```

- Ate: 30 pasajeros, 10 por viaje. Callao: 25, en un viaje. Gana Ate: se compara el **total**.
- Y aclaran los bordes: `find_route` con una ruta que no está da `-1`, y no 0, que es una posición válida.

¿Qué da una prueba que no da un comentario?

	Comentario	Prueba
¿Quién lo comprueba?	Nadie	La máquina, en cada corrida
¿Puede quedar desactualizado?	Sí, y sin aviso	No sin que algo falle
¿Qué tan preciso es?	“Busca la ruta”	Esta entrada da esta salida
¿Dice por qué?	Sí, para eso sirve	Casi nunca

Error común

“Todas las pruebas pasan, así que no hay errores.” Solo dicen algo de los casos que prueban. Ninguna de `test_trips.c` mira qué pasa con una novena ruta.

Probar un programa sin correr su `main`

test_trips.c

```c
#define main trips_main
#include "trips.c"
#undef main
```

- Un programa solo puede tener un `main`, y la prueba necesita el suyo.
- Estas líneas cambian el nombre del `main` de `trips.c` mientras lo incluyen.
- **Por ahora es una receta.** En la semana 6 veremos cómo se separa bien un programa en varios archivos.

SECCIÓN 04

Cambiar poco

04

Corregir: una versión de `trips.c` con un bug

terminal

```
$ echo viajes.txt | build/trips_bug
Chosica  2 viajes   22 pasajeros  11.00 por viaje
Callao   3 viajes   20 pasajeros   6.67 por viaje
Ate      2 viajes   14 pasajeros   7.00 por viaje
Más pasajeros: Chosica
```

Están mal los pasajeros, el promedio y la ruta ganadora. ¿Son tres problemas? Corramos las pruebas.

terminal

```
$ build/test_trips_bug
test_trips_bug: errors/test_trips_bug.c:19: test_add_trip: Assertion 'routes[0].passengers == 23' failed.
```

Una ruta recibe 9 y luego 14, y debería tener 23: una entrada que se sigue a mano.

Corregir: hipótesis, y cambiar lo mínimo

1. **Hipótesis antes de tocar nada.** Callao tiene 20 y Ate 14, los de su último viaje: `add_trip` reemplaza en vez de sumar.
2. **Un solo cambio**, en `add_trip`, y volver a correr todo.

terminal

```
$ diff errors/trips_bug.c trips.c
55c55
<     routes[pos].passengers = passengers;
---
>     routes[pos].passengers = routes[pos].passengers + passengers;
$ build/test_trips
Todas las pruebas pasan
```

Tres síntomas, una causa. La cantidad de pruebas que fallan no es la cantidad de bugs. Empiecen por la función más pequeña.

Extender: tres reglas

El pedido nuevo: ver también el total de pasajeros del día.

1. **Partir de pruebas que pasan.** Si algo ya fallaba, no sabrán si lo rompieron ustedes.
2. **Escribir la prueba primero.** Obliga a decidir qué debe hacer, y a pensar en los bordes.
3. **Tocar lo menos posible.** ¿Cuántas funciones existentes hay que cambiar?

Si la respuesta es “muchas”

Paren. Quizás hay un punto por donde pasan todos los datos, y una sola función nueva lo resuelve ahí. Cada parámetro nuevo obliga a cambiar todas las llamadas y sus pruebas.

Extender: una función nueva y una línea

terminal

```
$ diff trips.c trips_total.c
101a102,109
> int total_passengers(const struct route routes[], int n) {
>     int total = 0;
>     for (int i = 0; i < n; i++) {
>         total = total + routes[i].passengers;
>     }
>     return total;
> }
>
126a135
>     printf("Total de pasajeros: %d\n", total_passengers(routes, n));
```

terminal

```
$ build/test_trips_total
Todas las pruebas pasan
$ echo viajes.txt | build/trips_total
Más pasajeros: Callao
Total de pasajeros: 139
```

- Todas las líneas tienen >: se agregaron. No se quitó ni se cambió nada.
- Las pruebas antiguas siguen corriendo: dicen que lo anterior sigue funcionando.
- Si tienen que cambiar una prueba antigua para que pase, deténganse: están cambiando lo que el programa promete.

SECCIÓN 05

El laboratorio

05

El laboratorio: un programa de notas

`code/week-02/lab`: unas 200 líneas que leen un archivo de notas y escriben un reporte y un resumen.

Sesión 1

1. Leerlo sin cambiar nada, y responder por escrito: funciones, llamadas y datos.
2. Corregir los **dos** bugs que muestran las pruebas.

Sesión 2

3. El promedio sin la nota más baja, con pruebas.
4. Filtrar por sección, tocando lo menos posible.

`make test-grades` corre sus pruebas en C. `make test-lab` dice si está terminado: falla hasta que hagan las dos sesiones.

Resumen

- Arreglos: la longitud va aparte, y `sizeof` no la da en una función.
- Cadenas: `k + 1` bytes. `%s` con límite, y `strcmp` para comparar.
- `struct`: se copia entero. Un arreglo de `struct`, no.
- Archivos: toda lectura revisa su resultado.
- Leer: `main` y firmas, ejecutar, seguir un dato, escribir.
- Las pruebas son documentación que se ejecuta.
- Cambiar poco: una hipótesis, un cambio, y correr todo.

Para la próxima clase

- Terminen la sesión de laboratorio que quedó pendiente.
- Lean los apuntes de la semana 2 y hagan los ejercicios.
- Respondan, por escrito:
  1. ¿Qué pasa si el arreglo `char s[5]` recibe la cadena `"hola!"`?
  2. ¿Por qué hay que pasar el tamaño de un arreglo a una función?
  3. ¿Por dónde empezarían a leer un programa de 2 000 líneas que no conocen?
  4. ¿Qué información da una prueba que no da un comentario?

La semana que viene

El modelo de entornos: qué pasa en la memoria durante una llamada. Veremos por qué un arreglo pasado a una función se comporta distinto, con dibujos y con `gdb`.