

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Computación

Biblioteca de cadenas y matrices

`string.h`, sus trampas, y arreglos de dos dimensiones

Gerson Garrido

Semana 11

Hoy

1. `strlen` y el tipo `size_t`: por qué `strlen(s) - 1` es peligroso
2. Comparar: por qué `==` no compara cadenas, y `strcmp`
3. Copiar y agregar: `strncpy`, `strcat`, y dónde queda el `'\0'`
4. Buscar: `strchr` y `strstr`, una dirección o `NULL`
5. Matrices: fila por fila en memoria, recorridos, y pasarlas a una función
6. Sumar y transponer

> **Lectura**
>
> Apuntes de la semana 11. Código en `code/week-11`. Esta semana, la sesión de laboratorio es para la **Práctica 3**.

SECCIÓN 01

La biblioteca de cadenas

01

`string.h`: lo que hicimos a mano, ya escrito

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

Función	Qué hace	Lo que hay que cuidar
`strlen(s)`	cuenta los bytes antes del `'\0'`	devuelve un valor sin signo
`strcmp(a, b)`	compara dos cadenas	solo importa el signo
`strncpy(d, s, n)`	copia como máximo `n` bytes	puede no dejar `'\0'`
`strcat(d, s)`	agrega `s` al final de `d`	no recibe el tamaño de `d`
`strchr(s, c)`	busca un byte	puede devolver `NULL`
`strstr(s, t)`	busca una cadena en otra	puede devolver `NULL`

Ya saben qué hacen por dentro. Hoy importa qué devuelven y qué pasa en los bordes.

`strlen` devuelve `size_t`

lengths.c

```c
char t[8] = "hola";

printf("strlen(t) = %zu\n", strlen(t));
printf("sizeof(t) = %zu\n", sizeof(t));
```

terminal

```
$ build/lengths
strlen(t) = 4
sizeof(t) = 8
strlen("") = 0
strlen("año") = 4
```

`size_t`

Un entero **sin signo**, el mismo tipo que da `sizeof`. Se escribe con `%zu`. Aquí ocupa 8 bytes; depende de la plataforma.

La longitud sigue contando **bytes**: la `ñ` ocupa 2.

Restar 1 a un cero sin signo da la vuelta

terminal, recortado

```
$ gcc -Wall -Wextra -std=c17 errors/strlen_wrap.c -o build/strlen_wrap
errors/strlen_wrap.c:9:27: warning: comparison of unsigned expression in ‘>= 0’ is always true
$ build/strlen_wrap
strlen(empty) - 1 = 18446744073709551615
Esta condición siempre se cumple
```

- No da -1. Da el número más grande que cabe en un `size_t`.
- Aquí `size_t` tiene 64 bits: $2^{64} - 1$. Otra plataforma puede dar otro número.
- Los enteros sin signo dan la vuelta, y eso está definido: lo vimos en la semana 3.
- `gcc` avisa de la condición: un valor sin signo siempre es `>= 0`.
- `strlen(a) - strlen(b) < 0` nunca es verdadero. Se compara: `strlen(a) < strlen(b)`.

El último carácter, y la cadena vacía

errors/last_char.c

```c
char last_char(const char s[]) {
    return s[strlen(s) - 1];
}
```

la corrección

```
algoritmo ultimo(s)
    n ← longitud(s)
    si n = 0 entonces
        devolver NULO
    fin si
    devolver s[n - 1]
fin algoritmo
```

- Con `""` lee antes del arreglo: comportamiento indefinido. Sin aviso de `gcc`.

una corrida, recortada

```
==93592==ERROR: AddressSanitizer: stack-buffer-underflow [...]
    #0 0xaaaacd631108 in last_char errors/last_char.c:6
```

Recorrer usando `strlen`

terminal, recortado

```
$ gcc -Wall -Wextra -std=c17 errors/sign_compare.c -o build/sign_compare
errors/sign_compare.c:8:23: warning: comparison of integer expressions of different signedness [...]
    8 |     for (int i = 0; i < strlen(s); i++) {
```

- Además, `strlen` recorrería toda la cadena **en cada vuelta**.
- Se llama una vez, y se guarda con un cast:

lengths.c

```
1  int n = (int) strlen(t);
2  for (int i = n - 1; i >= 0; i--) {
```

- Con `int`, la cadena vacía da `n - 1` igual a -1, y el ciclo no corre.
- De izquierda a derecha, sigue siendo mejor `s[i] != '\0'`.

SECCIÓN 02

Comparar

02

== compara direcciones

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/equal_arrays.c -o build/equal_arrays
errors/equal_arrays.c: In function ‘main’:
errors/equal_arrays.c:8:11: warning: comparison between two arrays [-Warray-compare]
    8 |     if (a == b) {
      |           ^~
errors/equal_arrays.c:8:11: note: use ‘&a[0] == &b[0]’ to compare the addresses
$ build/equal_arrays
distintas
```

- Los dos arreglos guardan `"hola"`, pero están en lugares distintos.
- Como con los arreglos de la semana 9: `==` no mira el contenido.
- Es como comparar las direcciones de dos casas para saber si tienen los mismos muebles.

Dentro de una función, ni siquiera avisa

errors/equal_params.c

```c
int same_text(const char a[], const char b[]) {
    return a == b;
}
```

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/equal_params.c -o build/equal_params
$ build/equal_params
a con b: 0
a con a: 1
```

- Compila sin ninguna advertencia. Los parámetros ya son direcciones.
- Su respuesta no tiene nada que ver con el texto.
- Solo dice si las dos direcciones son la misma.

Con literales, el resultado no está fijado

una corrida: el resultado puede ser 0 o 1

```
$ gcc -Wall -Wextra -std=c17 errors/equal_literals.c -o build/equal_literals
errors/equal_literals.c:5:27: warning: comparison with string literal results in unspecified behavior
    5 |     printf("%d\n", "hola" == "hola");
$ build/equal_literals
1
```

C no dice si dos literales con el mismo texto se guardan en un solo lugar. Este 1 no prueba nada: `unspecified behavior` es un resultado que el lenguaje no fija.

Idea clave

Para comparar textos, `strcmp`. Que `==` alguna vez dé “iguales” no significa que esté bien.

`strcmp`: solo el signo

`strcmp(a, b)`	**Significa**
negativo	`a` va antes
0	son iguales
positivo	`a` va después

No cuenten con -1 ni con 1. Escriban `< 0`, `== 0`, `> 0`. El orden es por códigos, **no el de un diccionario**.

terminal

```
$ build/compare
"ana" y "ana" son iguales
"ana" va antes que "ano"
"an" va antes que "ana"
"Z" va antes que "a"
"z" va antes que "ñ"
```

Error común

`if (strcmp(a, b))` se lee “si son iguales” y hace lo contrario: iguales da 0, y 0 es falso. Con dos `"hola"`, `errors/strcmp_wrong.c` escribe `distintas`.

SECCIÓN 03

Copiar y agregar

03

`strcpy` y `strncpy`

- Un arreglo no se asigna con `=`. Hay que copiar sus bytes.
- `strcpy(dest, src)`: no recibe el tamaño de `dest`. Es el problema de `%s` sin límite.
- `strncpy(dest, src, n)` recibe un límite, pero **no se comporta como `copy_text`**:

Lo que hace `strncpy`

Si `src` es más corta que `n`: copia y **rellena con `'\0'`** hasta `n` bytes.
Si `src` tiene `n` bytes o más: copia `n` bytes y **no pone ningún `'\0'`**.

`strncpy(dest, src, 5)` con un destino de 5 bytes

`"sol"`	s	o	l	\0	\0	con relleno: una cadena
`"luna"`	l	u	n	a	\0	cabe justo: una cadena
`"estrella"`	e	s	t	r	e	**sin `'\0'`: no es una cadena**

terminal

```
$ build/strncpy_bytes
 115 111 108 0 0
 108 117 110 97 0
 101 115 116 114 101
```

`gcc` puede avisar si ve el texto (`-Wstringop-truncation`). Si llega por un parámetro, con nuestras opciones no avisa.

Si falta el `'\0'`, `printf` sigue de largo

errors/strncpy_no_null.c

```c
void fill(char dest[], const char src[]) {
    strncpy(dest, src, SIZE);
}
```

una corrida, recortada

```
$ build/strncpy_no_null_san
==93595==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffff8ef00025 [...]
READ of size 6 at 0xffff8ef00025 thread T0
    #0 0xffff91d0a4d8 in puts ../../../../src/libsanitizer/sanitizer_common/[...]
    #1 0xaaaad9ee0e28 in main errors/strncpy_no_null.c:15
```

Es el mismo tipo de informe de `char word[4] = "hola"`, la semana pasada. READ: `printf` sigue leyendo en busca de un `'\0'`. Compila sin advertencias.

La forma correcta tiene dos pasos

safe_copy.c

```c
void safe_copy(char dest[], const char src[],
    int size) {
    strncpy(dest, src, size - 1);
    dest[size - 1] = '\0';
}
```

terminal

```
$ build/safe_copy
sol
luna
estr
```

- Copiar uno menos que el tamaño, y poner el `'\0'` a mano.
- El tamaño llega en un parámetro: ahí `sizeof` no lo puede dar.

Idea clave

Después de cualquier `strncpy`, pregúntense: ¿quién puso el `'\0'`?

`strcat`: agregar, sin límite

- `strcat(dest, src)` copia `src` desde el `'\0'` de `dest`. Tampoco recibe el tamaño.
- `errors/strcat_overflow.c`: un `full` de 8 bytes con `"Ana "`, y una palabra leída.
- Compila sin advertencias. Con `Li`, funciona.

una corrida, recortada

```
$ echo Quispe_Huaman | build/strcat_overflow_san
==93609==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xffffa1a00028 [...]
WRITE of size 14 at 0xffffa1a00028 thread T0
    #0 0xffffa46a37a4 in strcat ../../../../src/libsanitizer/asan/asan_interceptors.cpp:515
    #1 0xaaaae88a0f68 in main errors/strcat_overflow.c:13
```

Es el buffer overflow de la semana pasada otra vez: el dato llegó de afuera.

`strncat`: el número es el espacio libre

append.c

```c
void safe_append(char dest[], const char src[],
      int size) {
    int used = (int) strlen(dest);
    strncat(dest, src, size - used - 1);
}
```

terminal

```
$ build/append
Ana Quispe
Ana Qui
```

- Agrega hasta `n` bytes, y **sí pone siempre el `'\0'`**, a diferencia de `strncpy`.
- Por eso `n` es el **espacio libre**, sin contar ese byte. No es el tamaño de `dest`.
- En 8 bytes con `"Ana"`: `used` es 3, y `8 - 3 - 1` da 4. Caben `" Qui"`.

El origen y el destino no pueden pisarse

errors/strcat_overlap.c

```
1  strcat(s, s);
```

una corrida, recortada

```
$ build/strcat_overlap_san
==93598==ERROR: AddressSanitizer: strcat-param-overlap: memory ranges [...] overlap
    #1 0xaaaae8ab0e48 in main errors/strcat_overlap.c:8
```

Comportamiento indefinido

Mientras escribe al final de `s`, cambia el texto que debe leer. Vale para `strcpy`, `strncpy`, `strcat` y `strncat`. Sin sanitizer, en esta corrida dio `holahola`. Eso no lo hace correcto.

SECCIÓN 04

Buscar

04

`strchr` y `strstr`: una dirección o `NULL`

find_text.c

```c
if (strchr(email, '#') == NULL) {
...
if (strstr(text, "Lima") != NULL) {
    printf("Desde Lima: %s\n", strstr(text, "Lima"));
}
...
if (strstr(&text[1], "el") == NULL) {
```

No declaramos variables para guardar direcciones. Con lo que ya sabemos, hay tres usos:

1. **Preguntar si aparece:** comparar con `NULL`.
2. **Usar el resultado como cadena:** `%s` escribe desde esa dirección hasta el `'\0'`. Solo después de revisar.
3. **Empezar desde otro lugar:** `&text[i]` es la dirección del byte `i`, como `&values[i]` en la semana 9.

Lo que sale

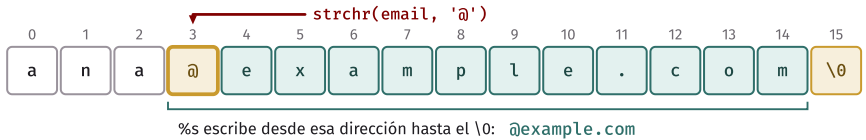
terminal

```
$ build/find_text
Tiene @: @example.com
No tiene #
lima, en minúsculas, no aparece
Desde la ñ: ño pasado llovió en Lima
Desde el byte 3: año pasado llovió en Lima
Desde el byte 1 no hay ninguno
```

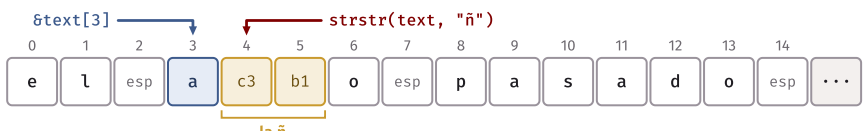
- Distingue mayúsculas, porque compara bytes.
- La `ñ` se busca con `strstr(text, "ñ")`: son 2 bytes. Con `strchr` no se puede.
- `strstr(s, "")` devuelve `s`: una cadena vacía aparece en todas partes.

Una dirección dentro del arreglo

```
char email[] = "ana@example.com";
```




```
char text[] = "el año pasado llovió en Lima";
```




```
Desde el byte 3: año pasado llovió en Lima
Desde la ñ: ño pasado llovió en Lima
```

Una dirección dentro del arreglo es el inicio de otra cadena.

Revisen siempre el `NULL`

errors/strstr_null.c

```c
printf("%s\n", strstr(text, "mar"));
```

en esta corrida, en este entorno

```
$ build/strstr_null; echo "estado: $?"
bash: line 1: 37413 Segmentation fault      build/strstr_null
estado: 139
```

Comportamiento indefinido

`"mar"` no está, y `printf` recibe `NULL`. Compila sin advertencias. Falta `luna` porque esta salida se capturó: aún no se había escrito al morir el programa.

SECCIÓN 05

Matrices

05

Una matriz: filas y columnas

grades_table.c

```c
int grades[STUDENTS][COURSES] = {
    {14, 9, 17, 12},
    {11, 16, 8, 13},
    {18, 10, 15, 20},
};
```

	Curso			
Estudiante	0	1	2	3
0	14	9	17	12
1	11	16	8	13
2	18	10	15	20

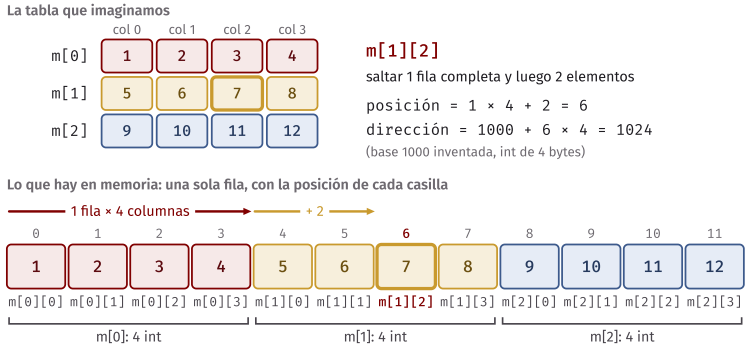
`grades[1][2]` vale 8

Primero las **filas**, luego las **columnas**. Dos índices, que empiezan en 0.

Error común

`m[1, 2]` no es fila 1, columna 2. La coma se queda con el lado derecho: es `m[2]`, la fila entera. `gcc` avisa: `left-hand operand of comma expression has no effect`.

En memoria: fila por fila, sin huecos



Sin huecos: la fila 1 empieza justo después de la fila 0.

`int m[3][4]` es un arreglo de arreglos: cada `m[i]` es una fila de 4 `int`. C guarda **fila por fila**, y lo garantiza el lenguaje (*(row-major order)*). La fórmula `i` × columnas + `j` **usa las columnas, no las filas**.

Comprobémoslo con %p

una corrida: las direcciones cambian

```
$ build/matrix_layout
sizeof(m) = 48
sizeof(m[0]) = 16
sizeof(m[0][0]) = 4
m[0][2] =  3 está en 0xffff90600038
m[0][3] =  4 está en 0xffff9060003c
m[1][0] =  5 está en 0xffff90600040
m[1][1] =  6 está en 0xffff90600044
```

- Un `int` ocupa 4 bytes **en este entorno**: $3 \times 4 \times 4 = 48$.
- Suben de 4 en 4, también de `m[0][3]` a `m[1][0]`.

Si `m` empieza en 1000, ¿dónde está `m[1][2]`?

La posición es $1 \times 4 + 2 = 6$. Con 4 bytes por `int`: $1000 + 6 \times 4 = 1024$.

Salirse de una columna no es “caer en la fila siguiente”

una corrida: el 4 no prueba nada

```
$ echo 2 | build/matrix_bounds_san
m[0][2] = 3
$ echo 3 | build/matrix_bounds_san
errors/matrix_bounds.c:12:38: runtime error: index 3 out of bounds for type 'int [3]'
m[0][3] = 4
```

- `m[0][3]` está dentro de la matriz, pero para C es comportamiento indefinido.
- AddressSanitizer revisa si se sale del objeto completo: **aquí no reporta nada**.
- Lo ve la parte `undefined`, que revisa el índice cuando conoce el tamaño de la fila. Es un aviso, y el programa sigue.

Recorrer por filas: para cada estudiante, sus cursos

pseudocódigo

```
algoritmo promedios_por_estudiante(notas, estudiantes, cursos)
    para i desde 0 hasta estudiantes - 1 hacer
        total ← 0
        para j desde 0 hasta cursos - 1 hacer
            total ← total + notas[i][j]
        fin para
        escribir total, total / cursos
    fin para
fin algoritmo
```

Error común

`total` en 0 fuera del ciclo exterior: cada fila suma también lo anterior. No hay ningún aviso.

Recorrer por columnas: cambian los ciclos, no los índices

pseudocódigo

```
algoritmo promedios_por_curso(notas, estudiantes, cursos)
    para j desde 0 hasta cursos - 1 hacer
        total ← 0
        para i desde 0 hasta estudiantes - 1 hacer
            total ← total + notas[i][j]
        fin para
        escribir total, total / estudiantes
    fin para
fin algoritmo
```

Idea clave

El ciclo exterior dice qué se recorre. Los índices de `notas[i][j]` **no cambian de lugar**: la fila va primero.

Por columnas, en C: el curso 1

grades_table.c

```c
for (int j = 0; j < COURSES; j++) {
    int total = 0;
    for (int i = 0; i < STUDENTS; i++) {
        total = total + grades[i][j];
    }
    ...
}
```

i	grades[i][1]	total
0	9	9

Por columnas, en C: el curso 1

grades_table.c

```c
for (int j = 0; j < COURSES; j++) {
    int total = 0;
    for (int i = 0; i < STUDENTS; i++) {
        total = total + grades[i][j];
    }
    ...
}
```

i	grades[i][1]	total
0	9	9
1	16	25

Por columnas, en C: el curso 1

grades_table.c

```c
for (int j = 0; j < COURSES; j++) {
    int total = 0;
    for (int i = 0; i < STUDENTS; i++) {
        total = total + grades[i][j];
    }
    ...
}
```

i	grades[i][1]	total
0	9	9
1	16	25
2	10	35

Por columnas, en C: el curso 1

grades_table.c

```c
for (int j = 0; j < COURSES; j++) {
    int total = 0;
    for (int i = 0; i < STUDENTS; i++) {
        total = total + grades[i][j];
    }
    ...
}
```

i	grades[i][1]	total
0	9	9
1	16	25
2	10	35

terminal

```
$ build/grades_table
Estudiante 2: suma 63, promedio 15.75
Curso 1: suma 35, promedio 11.67
```

Las dos formas de recorrer

Por filas

para cada estudiante, todos sus cursos

	0	1	2	3	suma
0	14	9	17	12	52
1	11	16	8	13	48
2	18	10	15	20	63

en memoria

14	9	17	12	11	16	8	13	18	10	15	20
1	2	3	4	5	6	7	8	9	10	11	12

orden de visita

Por columnas

para cada curso, todos sus estudiantes

	0	1	2	3
0	14	9	17	12
1	11	16	8	13
2	18	10	15	20

suma

en memoria

14	9	17	12	11	16	8	13	18	10	15	20

orden de visita

El ciclo exterior va sobre las filas: un resultado por fila, y casillas seguidas en memoria.

Las dos formas de recorrer

Por filas

para cada estudiante, todos sus cursos

	0	1	2	3	suma
0	14	9	17	12	52
1	11	16	8	13	48
2	18	10	15	20	63

en memoria

14	9	17	12	11	16	8	13	18	10	15	20
1	2	3	4	5	6	7	8	9	10	11	12

orden de visita

Por columnas

para cada curso, todos sus estudiantes

	0	1	2	3
0	14	9	17	12
1	11	16	8	13
2	18	10	15	20
suma	43	35	40	45

en memoria

14	9	17	12	11	16	8	13	18	10	15	20
1	4	7	10	2	5	8	11	3	6	9	12

orden de visita

Cambia el orden de los ciclos. grades[i][j] sigue igual: i es la fila y j, la columna.

Pasar una matriz a una función

matrix_param.c

```c
int column_sum(int m[][COLS], int rows, int column) {
```

- Primeros corchetes vacíos. Los segundos, **con las columnas**: en este curso, una constante.
- Las filas van aparte. ¿Por qué? `i` × **columnas** + `j`: sin columnas no sabe cuánto saltar.

terminal, recortado

```
$ gcc -Wall -Wextra -std=c17 errors/matrix_no_columns.c -o build/matrix_no_columns
errors/matrix_no_columns.c:4:15: error: array type has incomplete element type ‘int[]’
errors/matrix_no_columns.c:4:15: note: [...] must have bounds [...] except the first
$ gcc -Wall -Wextra -std=c17 errors/matrix_wrong_columns.c -o build/matrix_wrong_columns
errors/matrix_wrong_columns.c:11:26: error: [...] from incompatible pointer type [...]
errors/matrix_wrong_columns.c:4:15: note: expected ‘int (*)[3]’ but argument is of type ‘int (*)[4]’
```

`int (*)[4]` se lee “dirección de una fila de 4 `int`”.

La matriz original cambia

matrix_param.c

```c
void add_bonus(int m[][COLS], int rows,
               int bonus) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < COLS; j++) {
            m[i][j] = m[i][j] + bonus;
        }
    }
}
```

terminal

```
$ build/matrix_param
 14  9 17 12
 11 16  8 13
 18 10 15 20
Suma del curso 0: 43
Suma del curso 0, solo 2 filas: 25
 15 10 18 13
 12 17  9 14
 18 10 15 20
```

`add_bonus(grades, 2, 1)` suma 1 solo a las 2 primeras filas.

`const`: una excepción

La regla de la semana pasada: si no cambia el arreglo, lleva `const`. Con una matriz:

terminal, recortado

```
$ gcc -Wall -Wextra -std=c17 errors/const_matrix.c -o build/const_matrix
$ gcc -Wall -Wextra -std=c17 -Wpedantic errors/const_matrix.c -o build/const_matrix
errors/const_matrix.c:13:26: warning: invalid use of pointers to arrays [...] in ISO C before C23
```

- `const int m[][COLS]` con una matriz normal no está permitido en C17.
- Nuestras opciones no avisan. `-Wpedantic` pide a `gcc` revisar las reglas del estándar.
- C23 ya lo permite, pero el curso usa C17.

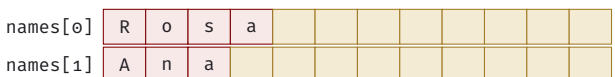
En este curso

Los parámetros matriz van **sin `const`**. La regla sigue igual para arreglos y cadenas.

Un arreglo de cadenas es una matriz de `char`

names.c

```c
char names[COUNT][NAME_SIZE] = {"Rosa", "Ana", "Luis", "Zoila"};
```



- Cada fila ocupa 12 bytes, aunque su texto sea más corto. En oro, los `'\0'`.
- `names[1]` es una cadena: sirve para `%s`, `strlen`, `strcmp`.
- Para cambiar una fila no sirve `=`: `safe_copy(names[1], "...", NAME_SIZE)`.

Usar `strcmp` para decidir el orden

pseudocódigo

```
algoritmo primer_nombre(nombres, n)
    primero ← 0
    para i desde 1 hasta n - 1 hacer
        si comparar(nombres[i], nombres[primero
    ]) < 0 entonces
            primero ← i
        fin si
    fin para
    devolver primero
fin algoritmo
```

names.c

```c
int first_name(char names[][NAME_SIZE],
     int n) {
    int first = 0;
    for (int i = 1; i < n; i++) {
        if (strcmp(names[i], names[first])
      < 0) {
            first = i;
        }
    }
    return first;
}
```

Ordenar solo necesita saber, de dos elementos, cuál va primero: si `strcmp(a, b) > 0`, se cambian de lugar. Se parece a `max_value` de la semana pasada, con `strcmp` en lugar de comparar números.

SECCIÓN 06

Sumar y transponer

06

Sumar y transponer

pseudocódigo

```
algoritmo sumar_matrices(a, b, resultado, filas
    , columnas)
    para i desde 0 hasta filas - 1 hacer
        para j desde 0 hasta columnas - 1 hacer
            resultado[i][j] ← a[i][j] + b[i][j
    ]
        fin para
    fin para
fin algoritmo
```

pseudocódigo

```
algoritmo transponer(a, resultado, filas,
    columnas)
    para i desde 0 hasta filas - 1 hacer
        para j desde 0 hasta columnas - 1 hacer
            resultado[j][i] ← a[i][j]
        fin para
    fin para
fin algoritmo
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

El 6 de `a[1][2]` pasa a `resultado[2][1]`.

Transponer en el mismo arreglo: un error silencioso

errors/transpose_in_place.c

```c
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        int saved = m[i][j];
        m[i][j] = m[j][i];
        m[j][i] = saved;
    }
}
```

terminal

```
$ gcc -Wall -Wextra -std=c17
    errors/transpose_in_place.c
    -o build/transpose_in_place
$ build/transpose_in_place
1 2 3
4 5 6
7 8 9
```

Cada par se intercambia dos veces

Una por `(i, j)` y otra por `(j, i)`. La matriz queda igual. Compila, corre y no dice nada.

SECCIÓN 07

El laboratorio

07

`count_word`: contar con `strstr`

pseudocódigo

```
algoritmo contar_palabra(texto, palabra)
    largo ← longitud(palabra)
    si largo = 0 entonces
        devolver 0
    fin si
    cuenta ← 0
    i ← 0
    mientras palabra aparece en texto desde i hacer
        si palabra aparece justo en la posición i
      entonces
            cuenta ← cuenta + 1
            i ← i + largo
        sino
            i ← i + 1
        fin si
    fin mientras
    devolver cuenta
fin algoritmo
```

`"ana"` en `"banana"`:

`i`	**Desde `i`**	**Qué hace**
0	`banana`	más adelante: `i` a 1
1	`anana`	justo aquí: cuenta 1, `i` a 4
4	`na`	no aparece: termina

- Reglas: byte por byte; `sol` cuenta en `girasol`; no se pisan; vacía da 0.
- Aparece: `strstr(&text[i], word) != NULL`
- Justo en `i`: `strstr(&text[i], word) == &text[i]`. Aquí la pregunta **sí** es sobre lugares.

Promedios y el mejor de cada curso

pseudocódigo

```
algoritmo mejores(notas, estudiantes, cursos, mejor)
    para j desde 0 hasta cursos - 1 hacer
        mejor[j] ← 0
        para i desde 1 hasta estudiantes - 1 hacer
            si notas[i][j] > notas[mejor[j]][j]
      entonces
                mejor[j] ← i
            fin si
        fin para
    fin para
fin algoritmo
```

- `grades.c`: 4 estudiantes, 3 cursos.
- `student_averages` y `course_averages` guardan en un arreglo.
- Empezar con el primero resuelve un curso con puros ceros.
- Con `>`, en un empate gana el de índice menor.
- Las pruebas detectan un `total` que no vuelve a 0 y dividir sin cast.

Transponer y comprobarlo con `assert`

pseudocódigo

```
algoritmo es_transpuesta(a, b, filas, columnas)
    para i desde 0 hasta filas - 1 hacer
        para j desde 0 hasta columnas - 1 hacer
            si b[j][i] ≠ a[i][j] entonces
                devolver 0
            fin si
        fin para
    fin para
    devolver 1
fin algoritmo
```

- `transpose`: `int result[][STUDENTS]`, porque el resultado tiene otra forma.
- Una prueba debe poder fallar: `main` cambia una casilla y exige un 0.

Cómo comprobar

`make test-lab`. Falla hasta que terminen `words.c` y `grades.c`.

Para la próxima clase

- En el laboratorio: **Práctica 3**. `words.c` y `grades.c` son trabajo autónomo.
- Leer los apuntes de la semana 12: `struct` y archivos de texto.
- Comparen con `strcmp`, revisen el `NULL` y pregúntense quién puso el `'\0'`.

Preguntas

¿Por qué `if (a == b)` no compara dos cadenas? ¿Qué compara?
¿Cuántos bytes ocupa `int m[3][4]`, con `int` de 4 bytes? ¿Dónde está `m[1][2]` si empieza en 1000?
¿Por qué una función que recibe una matriz necesita el número de columnas?