

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Computación

`struct` y archivos

Agrupar datos, leer y escribir archivos, y no romperse

Gerson Garrido

Semana 12

Hoy

1. De los arreglos paralelos a `struct`
2. Copiar y pasar un `struct`: siempre una copia completa
3. Arreglos de `struct`
4. Archivos: abrir, leer y cerrar, revisando los errores
5. Escribir: `fprintf` y lo que hace el modo `"w"`
6. Separar campos con `sscanf` y líneas que no caben
7. Un programa de principio a fin

> **Lectura**
>
> Apuntes de la semana 12. Código en `code/week-12`. Los nombres y los precios de los datos son inventados.

SECCIÓN 01

De los arreglos paralelos a `struct`

01



Arreglos paralelos: nada los une

errors/parallel_bug.c

```c
char names[N][NAME_SIZE] = {"Ana", "Bruno", "Carla"};
int grades[N] = {14, 9, 11};

// Bruno se retira: se mueven las notas, y se olvidan los
      nombres.
for (int i = 1; i < n - 1; i++) {
    grades[i] = grades[i + 1];
}
```

terminal

```
$ build/
    parallel_bug
Ana: 14
Bruno: 11
```

Compila sin advertencias, y no se escribe fuera de ningún arreglo. Pero el 11 era de Carla: lo que une `names[1]` y `grades[1]` solo existe en la cabeza de quien programa.

Definir un `struct`

first_struct.c

```c
#define NAME_SIZE 20
#define N_GRADES 3

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

- El tipo se llama `struct student`: las dos palabras.
- Termina en `;` después de la llave.
- Es el molde: no crea ninguna variable. Por eso va afuera de las funciones.

`struct`

Agrupa varios datos bajo un solo nombre. Cada dato es un **campo** (*(field)*), con su propio tipo.

Crear una variable y usar sus campos

first_struct.c

```c
struct student s = {"Ana", {14, 9, 17}};

printf("%s: %d %d %d\n", s.name, s.grades[0], s.
     grades[1], s.grades[2]);
s.grades[1] = 12;
```

terminal

```
$ build/first_struct
Ana: 14 9 17
Después del cambio: 12
```

- Las llaves llevan los valores en el orden de los campos.
- A un campo se llega con un **punto**.

En memoria, un solo bloque



- Los campos quedan en orden, uno tras otro.
- `sizeof(struct student)` da 32 **en este entorno**, con `int` de 4 bytes.

Tres errores de compilación

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/no_struct_word.c -o build/no_struct_word
errors/no_struct_word.c:13:5: error: unknown type name ‘student’; use ‘struct’ keyword to refer to the type
$ gcc -Wall -Wextra -std=c17 errors/assign_name.c -o build/assign_name
errors/assign_name.c:15:12: error: assignment to expression with array type
$ gcc -Wall -Wextra -std=c17 errors/compare_structs.c -o build/compare_structs
errors/compare_structs.c:16:11: error: invalid operands to binary == (have ‘struct student’ and ‘struct student’)
```

- El tipo no es `student`: es `struct student`.
- Un arreglo no se asigna con `=`, tampoco en un `struct`. Para un texto:

first_struct.c

```
1  strncpy(s.name, "Beatriz", NAME_SIZE - 1);
2  s.name[NAME_SIZE - 1] = '\0';
```

- Dos `struct` no se comparan con `==`: campo por campo; cadenas, con `strcmp`.

SECCIÓN 02

Copiar y pasar un `struct`

02

Asignar un `struct` copia todo

struct_copy.c

```c
struct student a = {"Ana", {14, 9, 17}};
struct student b = a;

b.name[0] = 'E';
b.grades[0] = 20;
print_student(a);
print_student(b);
```

terminal

```
$ build/struct_copy
Ana: 14 9 17
Ena: 20 9 17
```

`a` no cambió: `b` tiene su propio nombre y sus propias notas.

Idea clave

Un `struct` es un bloque de bytes con todo adentro, incluidos sus arreglos: al copiarlo, se copian.

Un `struct` llega a la función por valor

struct_param.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_first_array(int grades[]) {
    grades[0] = 0;
}
```

main

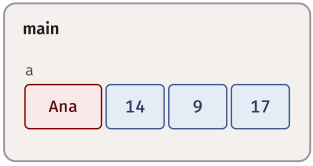
```c
reset_first(a);
a = with_first(a, 16);
reset_first_array(a.grades);
```

terminal

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

Tras cada llamada, `main` escribe `a`.

Las tres llamadas, en memoria



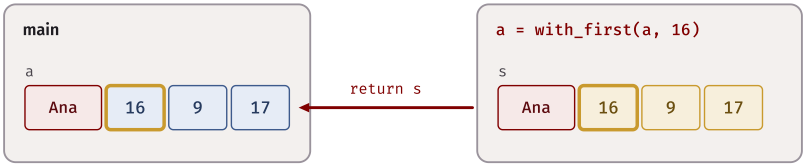


build/struct_param

```
En la función: 0
Ana: 14 9 17
```

reset_first recibe una copia completa.
El cambio se queda en s, y a sigue igual.

Las tres llamadas, en memoria



build/struct_param

```
En la función: 0
Ana: 14 9 17
Ana: 16 9 17
```

with_first cambia su copia y la devuelve.
a cambia porque main guarda lo devuelto.

Las tres llamadas, en memoria



build/struct_param

```
En la función: 0
Ana: 14 9 17
Ana: 16 9 17
Ana: 0 9 17
```

reset_first_array recibe la dirección de a.grades.
Su escritura llega al arreglo original.

La diferencia está en qué se pasa

Se llama con	Qué se copia al parámetro	¿Cambia el original?
`reset_first(a)`	todo el `struct`, con sus arreglos	no

La diferencia está en qué se pasa

Se llama con	Qué se copia al parámetro	¿Cambia el original?
`reset_first(a)`	todo el `struct`, con sus arreglos	no
`reset_first_array(a.grades)`	la dirección donde empieza `a.grades`	sí

La diferencia está en qué se pasa

Se llama con	Qué se copia al parámetro	¿Cambia el original?
`reset_first(a)`	todo el `struct`, con sus arreglos	no
`reset_first_array(a.grades)`	la dirección donde empieza `a.grades`	sí

- Para entregar un cambio: **devolver** el `struct`, y que quien llama lo guarde.
- Un arreglo *dentro* de un parámetro `struct` es una copia. No es como un arreglo parámetro.
- Copiar tiene un costo. Con estos tamaños no importa. Con `struct` muy grandes, en otros cursos se pasa un puntero.

SECCIÓN 03

Arreglos de `struct`

03



Una lista de estudiantes

struct_array.c

```c
struct student list[MAX_STUDENTS] = {
    {"Ana", {14, 9, 17}},
    {"Bruno", {8, 10, 11}},
    {"Carla", {11, 11, 11}},
};
int n = 3;
```

- Es un vector: capacidad `MAX_STUDENTS`, longitud `n`. Los que no aparecen entre las llaves quedan en 0.
- `list[1].grades[2]` se lee de izquierda a derecha: el elemento 1, su campo `grades`, la casilla 2.
- Llega a una función como cualquier arreglo: **viaja su dirección**, y la longitud va aparte.

El promedio de un estudiante

pseudocódigo

```
algoritmo promedio(s)
    total ← 0
    para i desde 0 hasta N_NOTAS - 1 hacer
        total ← total + s.notas[i]
    fin para
    devolver total / N_NOTAS
fin algoritmo
```

struct_array.c

```c
double average(struct student s) {
    int total = 0;
    for (int i = 0; i < N_GRADES; i++) {
        total = total + s.grades[i];
    }
    return (double) total / N_GRADES;
}
```

Recibe una copia. El cast evita la división entera.

El mejor promedio

pseudocódigo

```
algoritmo mejor(lista, n)
    si n ≤ 0 entonces
        devolver -1
    fin si
    mejor_pos ← 0
    para i desde 1 hasta n - 1 hacer
        si promedio(lista[i]) > promedio(lista[
      mejor_pos]) entonces
            mejor_pos ← i
        fin si
    fin para
    devolver mejor_pos
fin algoritmo
```

struct_array.c

```c
int best_index(const struct student list
      [], int n) {
    if (n <= 0) {
        return -1;
    }
    int best = 0;
    for (int i = 1; i < n; i++) {
        if (average(list[i]) > average(
     list[best])) {
            best = i;
        }
    }
    return best;
}
```

Con promedios 13.33, 9.67 y 11.00, devuelve 0.

Eliminar sin olvidar nada

struct_array.c

```c
int remove_at(struct student list[], int n,
              int pos) {
    if (pos < 0 || pos >= n) {
        return n;
    }
    for (int i = pos; i < n - 1; i++) {
        list[i] = list[i + 1];
    }
    return n - 1;
}
```

terminal

```
$ build/struct_array
Ana: 13.33
Bruno: 9.67
Carla: 11.00
Mejor: Ana
Sin Bruno:
Ana: 13.33
Carla: 11.00
Con la primera nota en 0:
Ana: 8.67
Carla: 7.33
```

`list[i] = list[i + 1];` copia el nombre y todas las notas juntos.

Lo último lo hace una función que recibe el arreglo: **sí** cambia los elementos.

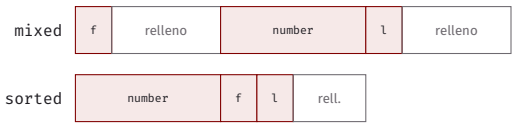
El tamaño no siempre es la suma

struct_size.c

```c
struct mixed {
    char first;
    int number;
    char last;
};
struct sorted {
    int number;
    char first;
    char last;
};
```

terminal

```
$ build/struct_size
sizeof(char) = 1, sizeof(int) = 4
sizeof(struct mixed) = 12
sizeof(struct sorted) = 8
```



En este entorno, el compilador ubica cada `int` en una dirección múltiplo de 4, y deja **relleno** (*(padding)*). C solo promete el orden de los campos y que el primero está al inicio. Usen `sizeof`, no sumen a mano.

SECCIÓN 04

Archivos: abrir, leer y cerrar

04

Un archivo es una secuencia de bytes con nombre

terminal

```
$ wc -c estudiantes.txt
116 estudiantes.txt
$ xxd estudiantes.txt | head -2
00000000: 416e 6120 3134 2039 2031 370a 4272 756e  Ana 14 9 17.Brun
00000010: 6f20 3820 3130 2031 310a 4361 726c 6120  o 8 10 11.Carla
```

- Guardada por el sistema operativo: sigue ahí cuando el programa termina.

Un archivo es una secuencia de bytes con nombre

terminal

```
$ wc -c estudiantes.txt
116 estudiantes.txt
$ xxd estudiantes.txt | head -2
00000000: 416e 6120 3134 2039 2031 370a 4272 756e  Ana 14 9 17.Brun
00000010: 6f20 3820 3130 2031 310a 4361 726c 6120  o 8 10 11.Carla
```

- Guardada por el sistema operativo: sigue ahí cuando el programa termina.
- **Una línea no es nada especial:** lo que hay entre dos bytes `0a`, el salto `\n`.

Un archivo es una secuencia de bytes con nombre

terminal

```
$ wc -c estudiantes.txt
116 estudiantes.txt
$ xxd estudiantes.txt | head -2
00000000: 416e 6120 3134 2039 2031 370a 4272 756e  Ana 14 9 17.Brun
00000010: 6f20 3820 3130 2031 310a 4361 726c 6120  o 8 10 11.Carla
```

- Guardada por el sistema operativo: sigue ahí cuando el programa termina.
- **Una línea no es nada especial:** lo que hay entre dos bytes `0a`, el salto `\n`.
- **El 14 no es un número:** son dos caracteres, `31` y `34`. Es un **archivo de texto**. Para hacer cuentas, hay que convertirlos.

Un archivo es una secuencia de bytes con nombre

terminal

```
$ wc -c estudiantes.txt
116 estudiantes.txt
$ xxd estudiantes.txt | head -2
00000000: 416e 6120 3134 2039 2031 370a 4272 756e  Ana 14 9 17.Brun
00000010: 6f20 3820 3130 2031 310a 4361 726c 6120  o 8 10 11.Carla
```

- Guardada por el sistema operativo: sigue ahí cuando el programa termina.
- **Una línea no es nada especial:** lo que hay entre dos bytes `0a`, el salto `\n`.
- **El 14 no es un número:** son dos caracteres, `31` y `34`. Es un **archivo de texto**. Para hacer cuentas, hay que convertirlos.
- **No hay marca de fin:** ningún `'\0'`. El sistema sabe cuánto mide.

fopen y FILE *

show_file.c

```c
FILE *file = fopen(path, "r");
if (file == NULL) {
    perror(path);
    return 1;
}
```

terminal

```
$ echo no_hay.txt | build/show_file
no_hay.txt: No such file or directory
```

La razón la pone la biblioteca del sistema: puede cambiar con el sistema o su idioma.

- `"r"` abre para leer. Si falla, `fopen` devuelve `NULL`, “ninguna dirección”. **Siempre se revisa.**
- `FILE *` se lee “dirección de un `FILE`”, como `int *` en la semana 10.
- Es una excepción, solo para esto: se guarda, se compara con `NULL` y se pasa a otras funciones. Nunca se usa el `*`.
- Como un número de atención: lo entrega `fopen`, y se muestra en cada trámite con ese archivo.

¿Y si no se revisa?

errors/no_check.c

```c
FILE *file = fopen("no_existe.txt", "r");
while (fgets(line, LINE_SIZE, file) != NULL) {
```

en esta corrida, en este entorno, recortado

```
$ build/no_check; echo "estado: $?"
bash: line 1: 92954 Segmentation fault      build/no_check
estado: 139
$ build/no_check_san
errors/no_check.c:10:12: runtime error: null pointer passed as argument 3, which is declared to never be null
==92961==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0xffff86c5f950 [...])
    #2 0xaaaac5980f04 in main errors/no_check.c:10
```

Comportamiento indefinido. `gcc` no avisa. Lo que debía ser un aviso claro, “ese archivo no existe”, se volvió un programa muerto.

Tres archivos ya abiertos

Nombre	Para qué	Normalmente va a
`stdin`	entrada estándar	el teclado
`stdout`	salida estándar, de `printf`	la pantalla
`stderr`	**salida de errores**, de `perror`	la pantalla

- Por eso `fgets(line, SIZE, stdin)` funcionó en la semana 10 sin `fopen`.
- En pantalla se ven juntas. Pero `>`, de la semana 2, solo desvía `stdout`: el resultado va al archivo y los avisos siguen a la vista.
- `2>` desvía `stderr`.

Leer línea por línea

pseudocódigo

```
algoritmo mostrar(ruta)
    archivo ← abrir(ruta)
    si archivo no se pudo abrir entonces
        escribir la razón
        devolver 1
    fin si
    numero ← 0
    linea ← leer_linea(archivo)
    mientras linea existe hacer
        numero ← numero + 1
        escribir numero, linea
        linea ← leer_linea(archivo)
    fin mientras
    cerrar(archivo)
    devolver 0
fin algoritmo
```

show_file.c

```c
char line[LINE_SIZE];
int number = 0;
while (fgets(line, LINE_SIZE, file) != NULL)
    {
    remove_newline(line);
    number++;
    printf("%d: %s\n", number, line);
}
fclose(file);
```

- `fgets` recuerda dónde iba. Al final, `NULL`.
- `fclose` avisa que terminamos. Los archivos abiertos a la vez son limitados: cada `fopen` que funcionó lleva su `fclose`.

La carpeta actual

terminal

```
$ echo estudiantes_vacio.
    txt | build/show_file
Total: 0 líneas
```

Con un archivo vacío, la primera lectura da `NULL`. No pasa nada malo.

terminal

```
$ cd build; echo estudiantes.txt | ./show_file; cd ..
estudiantes.txt: No such file or directory
$ cd build; echo ../estudiantes.txt | ./show_file | tail -1; cd ..
Total: 8 líneas
```

Un nombre como `estudiantes.txt` se busca desde la **carpeta actual**, no desde la carpeta del programa.

Un límite: `fgets` también devuelve `NULL` si hubo un error de lectura. `ferror(file)` los distingue. Nuestros programas no lo hacen.

El ciclo con `feof`: un error clásico

errors/feof_loop.c

```c
while (!feof(file)) {
    fgets(line, LINE_SIZE, file);
    printf("%s", line);
}
```

terminal

```
$ build/feof_loop
azúcar 3.20 tres
atún 5.90 3
atún 5.90 3
```

El ciclo con `feof`: un error clásico

errors/feof_loop.c

```c
while (!feof(file)) {
    fgets(line, LINE_SIZE, file);
    printf("%s", line);
}
```

terminal

```
$ build/feof_loop
azúcar 3.20 tres
atún 5.90 3
atún 5.90 3
```

Vuelta	¿feof?	Qué hace `fgets`	Qué escribe
5	no	lee hasta el `\n`, y se detiene ahí	`atún 5.90 3`
6	no	intenta, devuelve `NULL`, nadie mira	lo mismo de antes
7	sí	el ciclo termina	

El ciclo con `feof`: un error clásico

errors/feof_loop.c

```c
while (!feof(file)) {
    fgets(line, LINE_SIZE, file);
    printf("%s", line);
}
```

terminal

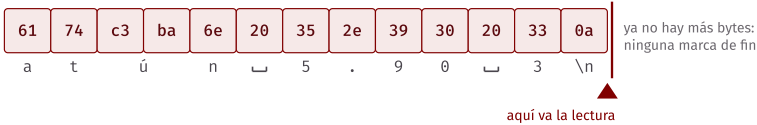
```
$ build/feof_loop
azúcar 3.20 tres
atún 5.90 3
atún 5.90 3
```

Vuelta	¿feof?	Qué hace `fgets`	Qué escribe
5	no	lee hasta el `\n`, y se detiene ahí	`atún 5.90 3`
6	no	intenta, devuelve `NULL`, nadie mira	lo mismo de antes
7	sí	el ciclo termina	

`feof` no anticipa: informa lo que ya pasó. La condición del ciclo es **el resultado de la lectura**.

Dónde queda la lectura

Los últimos bytes de compras.txt



Vuelta 5

¿feof(file) al empezar?	`no`
fgets devuelve	`no es NULL`
line tiene	`atún 5.90 3\n`

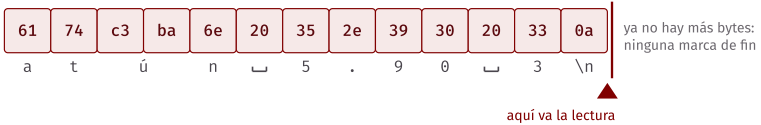
`build/feof_loop, al final`

```
azúcar 3.20 tres
atún 5.90 3
```

fgets se detiene en el \n. No intentó pasar del final, así que feof sigue diciendo no.

Dónde queda la lectura

Los últimos bytes de compras.txt



Vuelta 6

¿feof(file) al empezar?	`no`
fgets devuelve	`NULL`
line tiene	`atún 5.90 3\n`

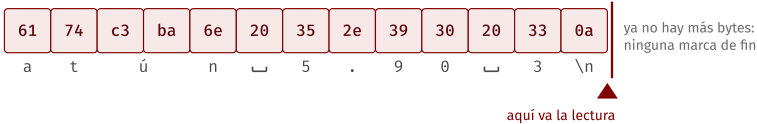
`build/feof_loop, al final`

```
azúcar 3.20 tres
atún 5.90 3
atún 5.90 3
```

fgets intenta, no hay nada, y devuelve NULL. Nadie lo mira: line se escribe otra vez.

Dónde queda la lectura

Los últimos bytes de compras.txt



Vuelta 7

¿feof(file) al empezar?	sí
fgets devuelve	no se llama
line tiene	atún 5.90 3\n

`build/feof_loop, al final`

```
azúcar 3.20 tres
atún 5.90 3
atún 5.90 3
```

Recién ahora feof dice sí, porque ya se intentó leer más allá. El ciclo termina.

SECCIÓN 05

Escribir un archivo

05

Modos y `fprintf`

Modo	Si existe	Si no existe	Para qué
`"r"`	lo lee desde el inicio	`fopen` devuelve `NULL`	leer
`"w"`	lo deja vacío	lo crea	escribir de cero
`"a"`	agrega al final	lo crea	agregar

write_note.c

```c
fprintf(file, "Nota final: %d\n", 15);
```

- `fprintf` es `printf` con un primer parámetro más: dónde escribir.
- `printf(...)` es lo mismo que `fprintf(stdout, ...)`. Nuestros avisos: `fprintf(stderr, ...)`.

El modo "w" borra

terminal

```
$ echo "texto viejo" > build/nota.txt
$ wc -c build/nota.txt
12 build/nota.txt
$ build/write_note
$ cat build/nota.txt
Nota final: 15
$ build/append_note
$ build/append_note
$ cat build/nota.txt
Nota final: 15
Otra línea
Otra línea
```

- El contenido desaparece **al abrir**, antes de escribir nada. Es lo mismo que `>`.
- `"a"`: cada corrida agrega.

Error común

Abrir la salida antes de saber si los datos sirven. Si la entrada no existe, ya borró el resultado anterior. **Primero se lee; la salida se abre al final.**

Cerrar importa más al escribir

en esta corrida: el momento exacto depende de la biblioteca

```
$ build/no_close
El archivo todavía está vacío
El archivo tiene: Nota: 15
```

`errors/no_close.c` escribe una línea y, sin cerrar, mira el archivo. La segunda línea sale después de `fclose`.

write_note.c

```
1  if (fclose(file) != 0) {
```

- Como `printf` en la semana 10, `fprintf` guarda en un buffer. Al cerrar, se escribe lo pendiente.
- Si el disco se llenó, puede ser `fclose` quien lo descubre: al escribir, **se revisa**.
- Si el programa muere antes de cerrar, lo pendiente se pierde.

SECCIÓN 06

Separar campos

06

sscanf: scanf sobre una cadena

parse_fields.c

```c
int fields = sscanf(line, "%19s %d %d %d %1s", name, &a, &b, &c, extra);
```

- El primer parámetro es la cadena. Devuelve **cuántos campos convirtió**.
- `name` no lleva `&`: ya entrega una dirección. `%19s`: el límite para 20 bytes.
- Un espacio en el formato acepta cualquier cantidad de espacios.
- `%1s` es una trampa: si llena `extra`, algo sobraba.

terminal

```
$ build/parse_fields
 4  "Ana 14 9 17"
 3  "Ana 14 9"
 1  "Ana diez 9 17"
 2  "Ana 14.5 9 17"
 5  "Ana 14 9 17 5"
-1  ""
```

Lo que `sscanf` dice y lo que no

Da	Línea	Qué pasó
4	`Ana 14 9 17`	lo que buscamos
3	`Ana 14 9`	falta una nota
1	`Ana diez 9 17`	`%d` no entiende `diez`, y se detiene
2	`Ana 14.5 9 17`	el primer `%d` lee 14; el siguiente no entiende `.5`
5	`Ana 14 9 17 5`	sobra algo: lo recogió `%1s`
1	un nombre de más de 19 bytes	`%19s` guarda 19; lo que queda no es un número
-1	vacía	no había nada: `EOF`

- Para la forma de la línea: **`fields` debe ser exactamente 4**.
- No revisa si tiene sentido: `Ana 25 9 17` da 4. El rango lo revisa el programa.
- No revisa si cabe: un número que no cabe en `int` es comportamiento indefinido en C.

Una línea más larga que el buffer

Después de `fgets`	Byte siguiente	Qué pasó	Qué hacer
hay `\n`	no hace falta mirarlo	la línea cupo	quitar el `\n`
no hay `\n`	`EOF`	última línea, sin salto	usarla
no hay `\n`	`\n`	cupo justo	usarla
no hay `\n`	otro	no cupo	descartar y avisar

`fgetc(file)` es como `getchar`, pero del archivo que le damos. Con un buffer de 16 bytes:

terminal

```
$ build/long_line
Línea 1: "una linea corta"
Línea 2: no cabe en 16 bytes, se descarta
Línea 3: "otra corta"
```

read_line

pseudocódigo

```
algoritmo leer_linea(archivo, linea, tamano)
    si no se pudo leer linea de archivo entonces
        devolver FIN
    fin si
    si linea trae salto entonces
        quitar el salto
        devolver BIEN
    fin si
    c ← siguiente byte de archivo
    si c = salto o c = fin de archivo entonces
        devolver BIEN
    fin si
    mientras c ≠ salto y c ≠ fin de archivo hacer
        c ← siguiente byte de archivo
    fin mientras
    devolver MUY_LARGA
fin algoritmo
```

long_line.c

```c
int read_line(FILE *file, char line[], int
      size) {
    if (fgets(line, size, file) == NULL) {
        return READ_END;
    }
    if (remove_newline(line)) {
        return READ_OK;
    }
    int c = fgetc(file);
    if (c == '\n' || c == EOF) {
        return READ_OK;
    }
    while (c != '\n' && c != EOF) {
        c = fgetc(file);
    }
    return READ_TOO_LONG;
}
```

SECCIÓN 07

De principio a fin

07

Una boleta: el plan, antes del código

pseudocódigo

```
algoritmo procesar(entrada, salida)
    n ← leer_productos(entrada, productos, CAPACIDAD)
    si n < 0 entonces
        devolver 1
    fin si
    si n = 0 entonces
        escribir "no hay productos"
        devolver 1
    fin si
    si no escribir_boleta(salida, productos, n)
      entonces
        devolver 1
    fin si
    escribir "Productos: ", n
    devolver 0
fin algoritmo
```

terminal

```
$ cat compras.txt
arroz 4.50 2
leche 3.80 6
pan 0.50 10
azúcar 3.20 tres
atún 5.90 3
```

La cuarta línea está mal a propósito: debe avisar y seguir.

La salida se abre solo cuando ya sabemos que hay algo que escribir.

Una línea

receipt.c

```c
struct item {
    char name[
      NAME_SIZE];
    double price;
    int quantity;
};
```

receipt.c

```c
int parse_item(const char line[], struct item items[], int pos)
      {
    struct item it;
    char extra[2];
    int fields = sscanf(line, "%19s %lf %d %1s", it.name, &it
      .price, &it.quantity, extra);
    if (fields != 3 || it.price < 0 || it.quantity <= 0) {
        return 0;
    }
    items[pos] = it;
    return 1;
}
```

Recibe el arreglo para poder entregar el resultado: un `struct` suelto llegaría por copia. Solo si todo está bien, copia al arreglo, con una asignación de `struct`.

Todo el archivo

receipt.c

```c
int result = read_line(file, line, LINE_SIZE);
while (result != READ_END) {
    number++;
    if (count == capacity) {
        fprintf(stderr, "Demasiados productos: solo se leen %d\n",
     capacity);
        break;
    }
    if (result == READ_OK && parse_item(line, items, count)) {
        count++;
    } else {
        fprintf(stderr, "Línea %d: mal formada, se omite\n", number
     );
    }
    result = read_line(file, line, LINE_SIZE);
}
```

- `count == capacity` **antes de guardar**: si no, se escribe fuera del arreglo.
- `count`: cuántos hay. `number`: en qué línea vamos.
- `&&` se detiene en el primer falso: si no cupo, no se llama a `parse_item`.

Probarlo con lo bueno y con lo malo

terminal

```
$ printf 'compras.txt\nbuild/boleta.txt\n' | build/receipt
Línea 4: mal formada, se omite
Productos: 4
$ cat build/boleta.txt
arroz 2 x 4.50 = 9.00
leche 6 x 3.80 = 22.80
pan 10 x 0.50 = 5.00
atún 3 x 5.90 = 17.70
Total: S/ 54.50
$ printf 'no_hay.txt\nbuild/boleta.txt\n' | build/receipt; echo "estado: $?"
no_hay.txt: No such file or directory
estado: 1
$ wc -l build/boleta.txt
5 build/boleta.txt
```

Se prueba sobre todo con lo que puede salir mal: el que no existe, el vacío, una línea rota, una larga, más datos de los que caben. Debe **avisar, seguir si puede, y no dejar basura**. Aquí, la boleta anterior sigue intacta.

SECCIÓN 08

El laboratorio

08

Procesar el archivo de estudiantes

- Lea `estudiantes.txt` en un arreglo de `struct`; escriba los aprobados y su promedio. Se aprueba con 11.
- Escriben `read_line`, `parse_student`, `read_students`, `average`, `is_passed`, `write_passed` y `main`.
- Mejor comparar enteros: promedio al menos 11 es lo mismo que suma al menos 33.

Entrada	Qué se espera	Estado	¿Salida?
`estudiantes.txt`	`Leídos: 8. Aprobados: 5.`	0	sí
`estudiantes_malos.txt`	avisos y `Leídos: 3. Aprobados: 2.`	0	sí
`estudiantes_vacio.txt`	`estudiantes_vacio.txt: no hay estudiantes`	1	no se crea
uno que no existe	el mensaje de `perror`	1	no se toca

Cómo comprobar: `make test-lab`. También revisa una línea sin `\n`, una demasiado larga, más de 50 estudiantes, y que se cierren los archivos.

Para la próxima clase

- Hoy, en el laboratorio: `lab/process_students.c`.
- Leer los apuntes de la semana 13: búsqueda, ordenamiento y eficiencia.
- Revisen siempre `fopen`, `fgets` y `sscanf`. Nunca `while (!feof(file))`.

Preguntas

¿Qué ventaja tiene un arreglo de `struct` frente a varios arreglos paralelos?
Si una función recibe un `struct` y cambia un campo, ¿cambia el original? ¿Por qué?
¿Qué hace `fgets` si la línea es más larga que el buffer? ¿Cómo lo detectarían?