Manejo avanzado de archivos en C

Info II
Profesor: Ing. Weber Federico

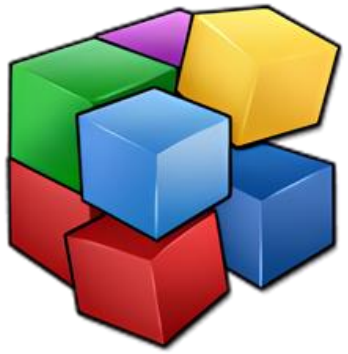
Un poco de historia

En C, los archivos se manejan como streams (flujos de bytes), lo cual refleja una serie de decisiones históricas y técnicas:

- Las cintas magnéticas eran un medio de almacenamiento común en la época en que se diseñó C. Por su naturaleza física, solo permitían acceso secuencial: se debía recorrer la cinta para llegar a un punto específico.
- Los primeros discos duros, aunque admitían acceso aleatorio, eran más eficientes cuando se accedía a los datos de manera secuencial, debido a la disposición física de los sectores y las limitaciones mecánicas de los cabezales de lectura/escritura.
- UNIX, el sistema operativo para el que se diseñó C, adopta una filosofía en la que "todo es un archivo" y muchos dispositivos (terminales, impresoras....) se trataban como streams, es decir, flujos secuenciales de datos.
- Este enfoque unificado, donde tanto los archivos como otros dispositivos se tratan como streams, facilita la portabilidad y la simplicidad del código en C, independientemente del dispositivo subyacente.

Aunque C también proporciona funciones para acceso aleatorio (como fseek y ftell), el modelo general de manejo de archivos está basado en la noción de streams secuenciales, lo cual encaja con las limitaciones tecnológicas y las decisiones de diseño de su época de origen.

Un poco de historia



- La desfragmentación era —y en algunos contextos sigue siendo— una operación importante debido a cómo el acceso secuencial afectaba el rendimiento:
- Cuando un archivo está **fragmentado** (almacenado en bloques dispersos por el disco), el **cabezal de lectura/escritura** debe moverse físicamente entre distintas ubicaciones para acceder a todas las partes. Estos movimientos, conocidos como ***seeks***, son mucho más lentos que una lectura secuencial continua.
- Esto impactaba en el rendimiento:
  1. La **lectura secuencial**, que debería ser la más rápida, se volvía mucho más lenta en presencia de fragmentación.
  2. El sistema operativo debía realizar múltiples operaciones de reposicionamiento del cabezal, lo que se convertía en un cuello de botella.
- La **desfragmentación** reorganizaba los archivos en bloques contiguos, optimizando el disco justamente para el acceso secuencial. Esto beneficiaba directamente el rendimiento de aplicaciones que trabajaban con archivos.
- Con la aparición de los **discos SSD**, que no tienen partes móviles y ofrecen tiempos de acceso casi constantes sin importar la ubicación física de los datos, la desfragmentación perdió gran parte de su relevancia. Esto evidencia que su utilidad estaba directamente relacionada con las **limitaciones físicas de los discos mecánicos** y no al modelo secuencial de acceso.

Estado del arte

¿Acceso secuencial?

SSD Sata

M2 Sata

PCIe tipo placa

M2 NVME

Algunas comparaciones

Tecnología	Latencia	Velocidad	Capacidades típicas	Notas
DDR4 RAM	~10 ns	20–25 GB/s	4–32 GB por módulo	Uso general
DDR5 RAM	~10 ns	40–60 GB/s	8–64 GB por módulo	Más rápida, común en PCs nuevos
SSD NVMe (PCIe 4.0)	25–100 µs	5–7 GB/s	500 GB – 4 TB	M.2 NVMe para usuarios personales
SSD NVMe (PCIe 5.0)	20–80 µs	10–14 GB/s	1–4 TB	Gama alta actual, requiere motherboard moderna
SSD SATA	70–200 µs	500–550 MB/s	120 GB – 2 TB	Compatibilidad universal, más lento
HDD 7200 RPM	4–6 ms	150–200 MB/s	1 – 20 TB	Bajo costo por GB, ideal para backups

Ordenamiento

Si tenemos que ordenar un archivo, qué nos conviene:

¿trabajar directamente sobre el archivo o llevarlo a memoria?

Depende.

- Si el archivo es muy pequeño, la diferencia entre trabajar directamente sobre el archivo o cargarlo en memoria es prácticamente **invisible**. Cualquier método funciona rápido.
- Si el archivo es muy grande y **no entra en memoria**, no queda otra opción: hay que aplicar **algoritmos de ordenamiento externo**, que trabajan directamente sobre el archivo o sobre fragmentos de él.
- Pero en la mayoría de los casos **intermedios**, donde el archivo sí entra en memoria, conviene **llevarlo a memoria y ordenarlo allí**. ¿Por qué?
- Porque la **latencia de acceso en RAM es miles de veces menor** que la de un disco, incluso un SSD moderno.
  Y los algoritmos de ordenamiento —especialmente los más avanzados que veremos más adelante— hacen **muchos accesos aleatorios** a los datos. Si esos accesos son en disco, cada uno tiene un costo. En RAM, ese costo es prácticamente cero.

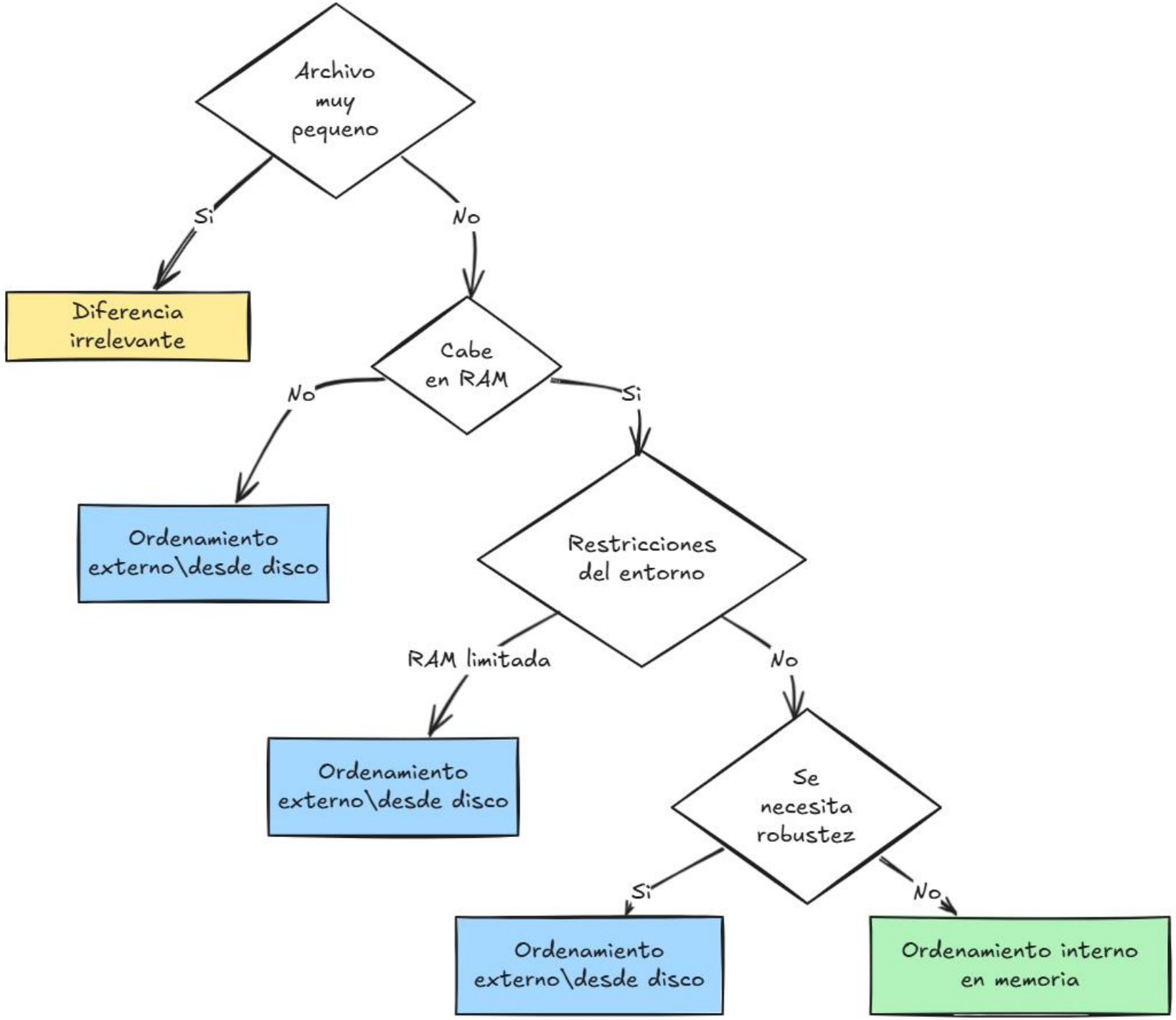
Entonces... ¿siempre es conveniente ordenar nuestros archivos en memoria?

Depende.

- En muchos entornos VPS (como Heroku, DigitalOcean, Contabo, etc.), **la RAM es limitada y compartida** entre usuarios. Si cargás un archivo grande a memoria, podés:
  - Ser penalizado por el sistema (OOM = out of memory).
  - Afectar el rendimiento de otros servicios en tu instancia.
- Si estás manipulando grandes volúmenes de datos en RAM y se corta la luz, se reinicia el sistema o hay un bug... todo el trabajo se pierde. Si trabajás directamente en disco (o con archivos temporales intermedios), podés:
  - Guardar progresos parciales.
  - Retomar desde el último paso exitoso.

¿Entonces?

No queda otra que tomar una decisión



Disclaimer

Unas clases mas adelante vamos a estudiar dos tipos de métodos de ordenamiento:

Basicos	Avanzados
Intercambio	Shell Sort (interno)
Selección	Quick Sort (interno)
Inserción	
Burbujeo	

- Los métodos “avanzados” están diseñados para funcionar eficientemente en memoria RAM, donde el acceso a los datos es muy rápido y no hay penalización por accesos aleatorios.
- En cambio, cuando se trabaja directamente sobre archivos en disco, se necesitan algoritmos específicos llamados **métodos de ordenamiento externo**. Un ejemplo clásico es **Merge Sort externo**, que permite ordenar archivos grandes leyendo y escribiendo por bloques de forma secuencial.
- Estos métodos externos no forman parte de este curso, pero es importante saber que existen y que son los adecuados para ordenar archivos que no caben en memoria.
- Por este motivo, si trabajan sobre archivos directamente, usen únicamente alguno de los **métodos básicos**, y ténganlo en cuenta como una simplificación.
- Si lo bajo a memoria, uso el método avanzado que mas me guste.

Ordenamiento interno con vector auxiliar (en memoria) – ejemplo01.c

Se desea ordenar un archivo de productos en función del precio, de menor a mayor. Cada producto tiene:

- Un **código** (entero)
- Un **nombre** (cadena de hasta 30 caracteres)
- Un **precio** (float)

```
typedef struct {
    int codigo;
    char nombre[31];
    float precio;
} Producto;
```

- No conocemos cuantos productos tiene el archivo

Ordenamiento interno con vector auxiliar (en memoria) – ejemplo01.c

```c
#include <stdio.h>

#define ARCHIVO_IN "productos.dat"
#define ARCHIVO_OUT "productos2.dat"
#define MAX_PRODUCTOS 200

typedef struct {
    int codigo;
    char nombre[31];
    float precio;
} Producto;

void bubbleSort(Producto [], int );
void mostrarArchivo (const char *);
```

Antes del main, ¿qué tenemos?

- Vamos a trabajar con dos archivos solo con fines didácticos, para no sobrescribir el archivo original desordenado y así poder reutilizarlo.
- **#define MAX_PRODUCTOS 200**
  Aún no tenemos una mejor forma (sin usar memoria dinámica) de saber cuántos productos hay hasta leer el archivo, así que reservamos una cantidad que pensamos que alcanza.
- **typedef struct { ... } Producto**;
  Esto es para evitar tener que escribir **struct Producto** cada vez que declaramos una variable del tipo.

Ordenamiento interno con vector auxiliar (en memoria) – ejemplo01.c

```c
#include <stdio.h>

#define ARCHIVO_IN "productos.dat"
#define ARCHIVO_OUT "productos2.dat"
#define MAX_PRODUCTOS 200

typedef struct {
    int codigo;
    char nombre[31];
    float precio;
} Producto;

void bubbleSort(Producto [], int );
void mostrarArchivo (const char *);
```

mostrarArchivo

- Retorna **void**
- recibe un puntero del tipo **char**. **const** lo que hace es que la función no puede modificar el contenido al que se apunta. Es el string con el nombre del archivo

bubbleSort

- Retorna **void**
- recibe un puntero del tipo **producto**. **Producto []** es lo mismo **que Producto *.** Va a recibir la posición inicial del vector auxiliar con toda la información del archivo.

Nota: **const** en punteros ¿Qué cosa es constante?

Declaración	¿Valor apuntado modificable?	¿Puntero modificable?	Significado
const int* ptr	No	Sí	El **dato** es constante
int* const ptr	Sí	No	El **puntero** es constante
const int* const ptr	No	No	**Todo** es constante

```
int a = 10, b = 20;

const int* p1 = &a;   // *p1 no se puede modificar, p1 sí
int* const p2 = &a;   // *p2 sí se puede modificar, p2 no
const int* const p3 = &a; // Nada se puede modificar
```

La palabra reservada **const** afecta a lo que está a su izquierda. Si no hay nada, afecta al dato apuntado.

Ordenamiento interno con vector auxiliar (en memoria) – ejemplo01.c

```c
int main() {

    // Mostrar archivo original
    mostrarArchivo(ARCHIVO_IN);

    Producto productos[MAX_PRODUCTOS];

    FILE *fp = fopen(ARCHIVO_IN, "rb");
    if (!fp) {
        perror("Error al abrir el archivo");
        return 1;
    }

    // Determinar tamaño del archivo en bytes
    fseek(fp, 0, SEEK_END);
    long tam_bytes = ftell(fp);
    int total = tam_bytes / sizeof(Producto);
    fseek(fp, 0, SEEK_SET);
    // Verificar límite
    if (total > MAX_PRODUCTOS) {
        printf("Advertencia: el archivo contier
        total = MAX_PRODUCTOS;
    }
```

- La función **mostrarArchivo** va a imprimir el archivo original antes de empezar.
- Abro el archivo original
- Con **fseek** y **ftell** averiguo el tamaño del archivo y lo guardo en **total**
- Verifico que **total** no sea mayor a mi estimación de máximo. Si es mayor aclaro que solo voy a tomar 200 productos.

```c
    fread(productos, sizeof(Producto), total, fp);
    fclose(fp);
    getchar();

    bubbleSort(productos, total);

    // Reescribir el archivo ordenado
    fp = fopen(ARCHIVO_OUT, "wb");
    if (!fp) {
        perror("Error al escribir el archivo");
        return 1;
    }

    fwrite(productos, sizeof(Producto), total, fp);
    fclose(fp);

    // Mostrar archivo después de ordenar
    printf("\n----- Archivo luego del ordenamiento -----\n");
    mostrarArchivo(ARCHIVO_OUT);
    return 0;
}
```

1. Todo el contenido del archivo lo mando al vector.
2. Cierro el flujo
3. Llamo a la función ordenar vector.
4. Abro el archivo en modo escrituria binaria.
5. Copio el vector en el archivo.
6. Imprimo el archivo ordenado.

```c
void mostrarArchivo(const char *nombreArchivo) {
    FILE *f = fopen(nombreArchivo, "rb");
    if (!f) {
        perror("Error al abrir el archivo");
        return;
    }

    Producto p;
    printf("----- Contenido del archivo '%s' -----\n", nombreArchivo);

    while (fread(&p, sizeof(Producto), 1, f) == 1) {
        printf("Codigo: %3d | Nombre: %-15s | Precio: $%7.2f\n", p.codigo, p.nombre, p.precio);
    }

    fclose(f);
}
```

```
//void bubbleSort(Producto *v, int n)
void bubbleSort(Producto v[], int n) {
    Producto aux;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (v[j].precio > v[j + 1].precio) {
                aux = v[j];
                v[j] = v[j + 1];
                v[j + 1] = aux;
            }
        }
    }
}
```

No voy a explicar el burbujeo, lo deben conocer y además tenemos una clase completa de métodos de ordenamiento.

Observar como manejar el vector de estrucutras

Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

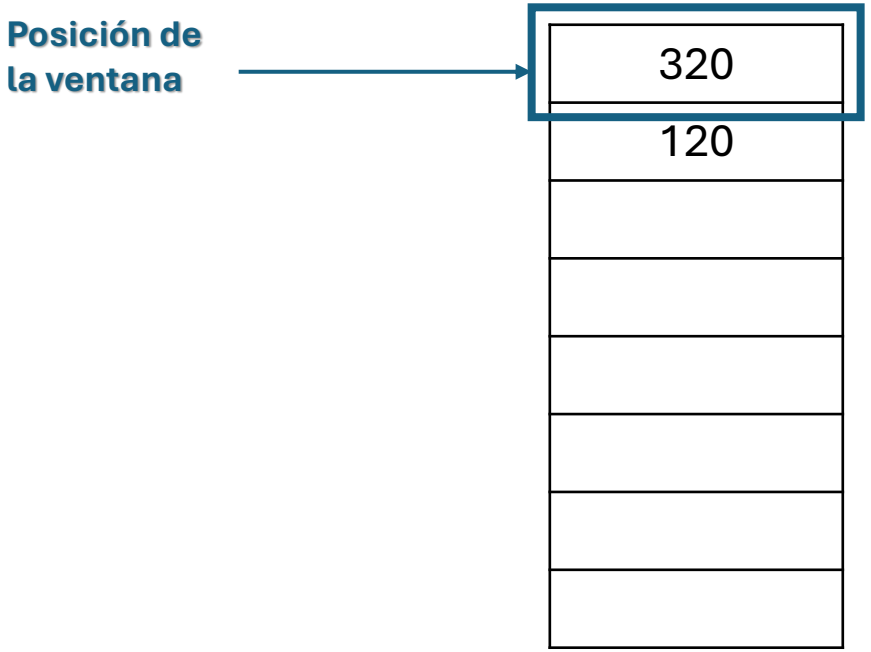
```c
void bubbleSortArchivo(const char *nombreArchivo) {
    FILE *f = fopen(nombreArchivo, "rb+");  // lectura y escritu
    if (!f) {
        perror("Error al abrir el archivo para ordenamiento");
        return;
    }
    fseek(f, 0, SEEK_END);
    long tam_bytes = ftell(f);
    int total = tam_bytes / sizeof(Producto);

    for (int i = 0; i < total - 1; i++) {
        for (int j = 0; j < total - 1 - i; j++) {
            Producto a, b;
            // Leer producto en posición j
            fseek(f, j * sizeof(Producto), SEEK_SET);
            fread(&a, sizeof(Producto), 1, f);
            // Leer producto en posición j+1
            fread(&b, sizeof(Producto), 1, f);
            if (a.precio > b.precio) {
                // Intercambiar: escribir b en j, a en j+1
                fseek(f, j * sizeof(Producto), SEEK_SET);
                fwrite(&b, sizeof(Producto), 1, f);
                fwrite(&a, sizeof(Producto), 1, f);
            }
        }
    }
    fclose(f);
```

1. Determino la cantidad de elementos del archivo
2. Algoritmo de burbujeo, acomodando la ventana de lectura del archivo.

Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

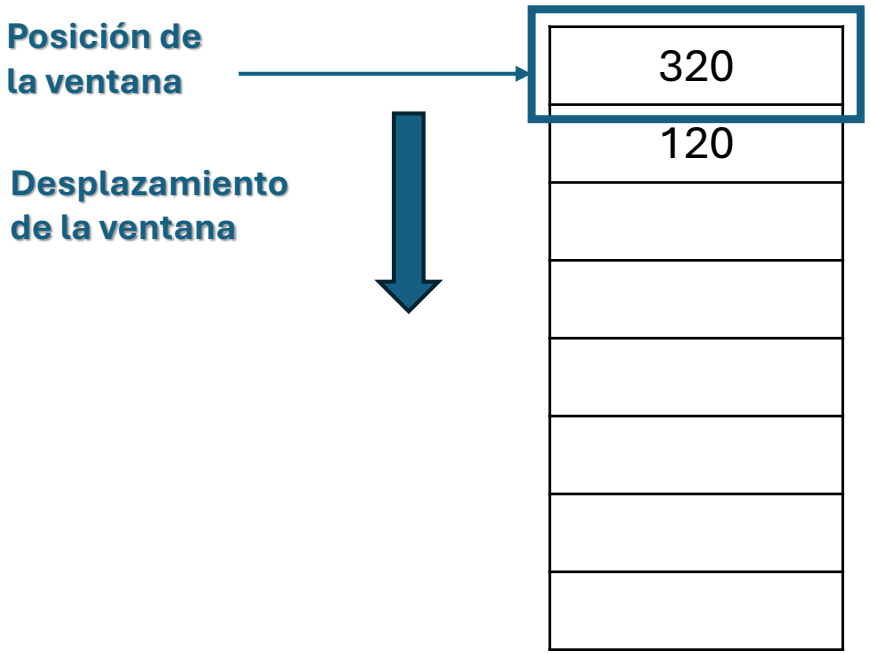
```c
Producto a, b;
for (int i = 0; i < total - 1; i++) {
    for (int j = 0; j < total - 1 - i; j++) {
        // Leer producto en posición j
        fseek(f, j * sizeof(Producto), SEEK_SET);
        fread(&a, sizeof(Producto), 1, f);
        // Leer producto en posición j+1
        fread(&b, sizeof(Producto), 1, f);
        if (a.precio > b.precio) {
            // Intercambiar: escribir b en j, a en j+1
            fseek(f, j * sizeof(Producto), SEEK_SET);
            fwrite(&b, sizeof(Producto), 1, f);
            fwrite(&a, sizeof(Producto), 1, f);
        }
    }
}
```

1



Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

```c
Producto a, b;
for (int i = 0; i < total - 1; i++) {
    for (int j = 0; j < total - 1 - i; j++) {
        // Leer producto en posición j
        fseek(f, j * sizeof(Producto), SEEK_SET); 1
        fread(&a, sizeof(Producto), 1, f); 2
        // Leer producto en posición j+1
        fread(&b, sizeof(Producto), 1, f);
        if (a.precio > b.precio) {
            // Intercambiar: escribir b en j, a en j+1
            fseek(f, j * sizeof(Producto), SEEK_SET);
            fwrite(&b, sizeof(Producto), 1, f);
            fwrite(&a, sizeof(Producto), 1, f);
        }
    }
}
```



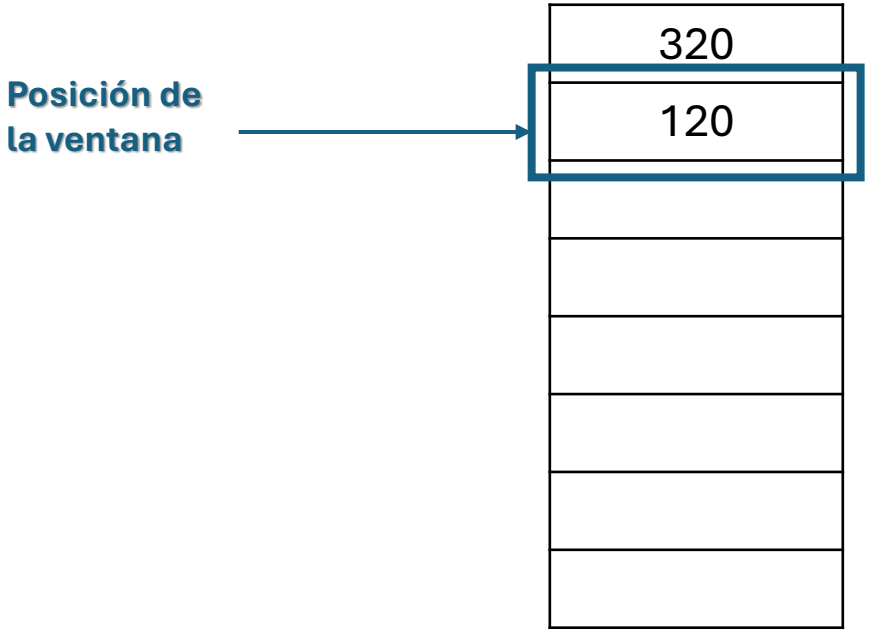
A = 320

Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

```c
Producto a, b;
for (int i = 0; i < total - 1; i++) {
    for (int j = 0; j < total - 1 - i; j++) {
        // Leer producto en posición j
        fseek(f, j * sizeof(Producto), SEEK_SET);
        fread(&a, sizeof(Producto), 1, f);
        // Leer producto en posición j+1
        fread(&b, sizeof(Producto), 1, f);
        if (a.precio > b.precio) {
            // Intercambiar: escribir b en j, a en j+1
            fseek(f, j * sizeof(Producto), SEEK_SET);
            fwrite(&b, sizeof(Producto), 1, f);
            fwrite(&a, sizeof(Producto), 1, f);
        }
    }
}
```

1

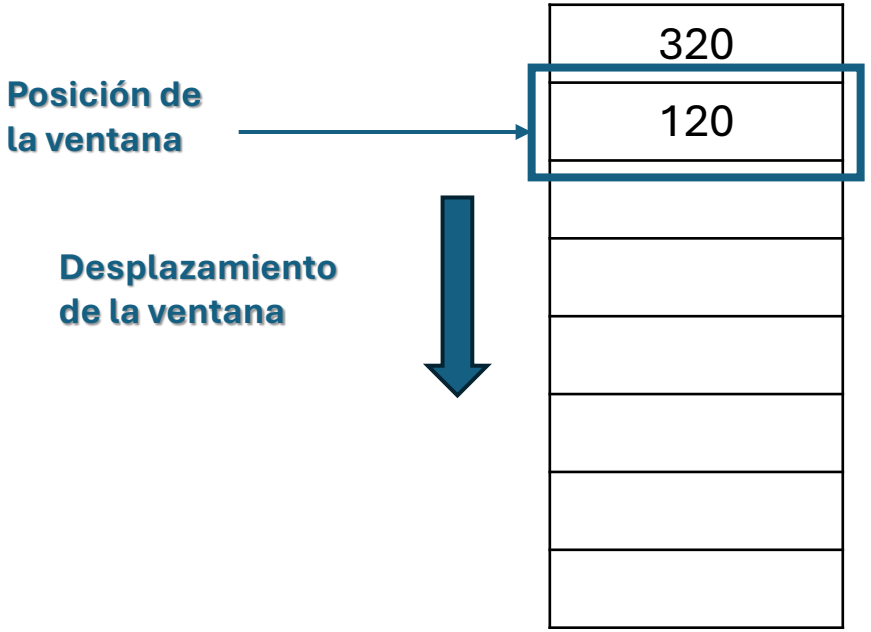
2



A = 320

Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

```c
Producto a, b;
for (int i = 0; i < total - 1; i++) {
    for (int j = 0; j < total - 1 - i; j++) {
        // Leer producto en posición j
        fseek(f, j * sizeof(Producto), SEEK_SET); 1
        fread(&a, sizeof(Producto), 1, f); 2
        // Leer producto en posición j+1
        fread(&b, sizeof(Producto), 1, f); 3
        if (a.precio > b.precio) {
            // Intercambiar: escribir b en j, a en j+1
            fseek(f, j * sizeof(Producto), SEEK_SET);
            fwrite(&b, sizeof(Producto), 1, f);
            fwrite(&a, sizeof(Producto), 1, f);
        }
    }
}
```

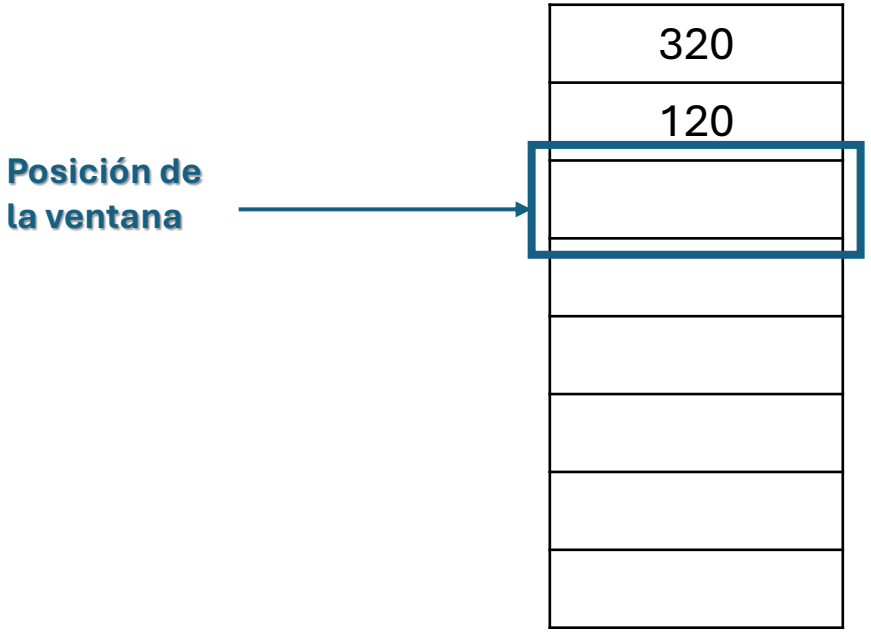


A = 320

B = 120

Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

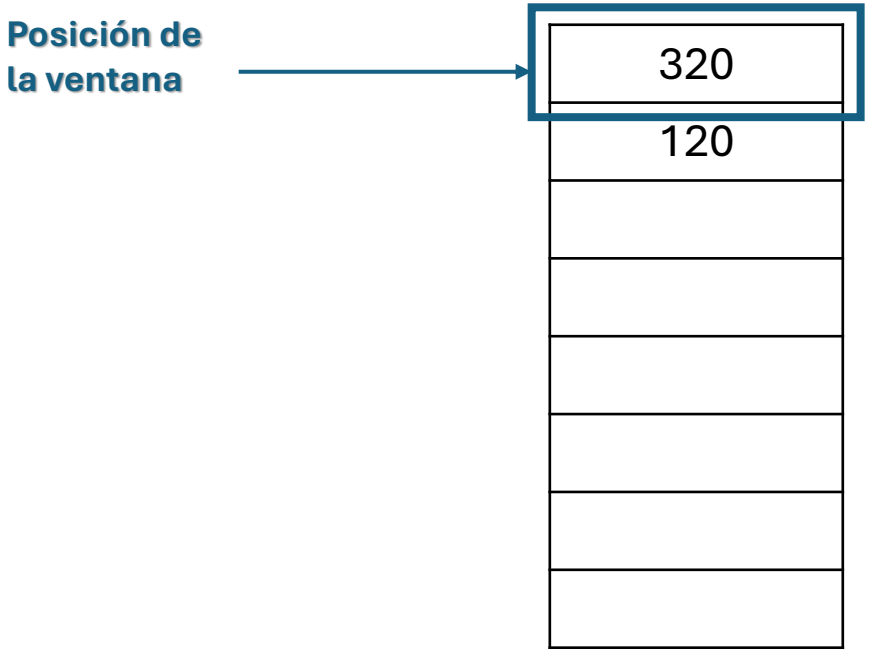
```c
Producto a, b;
for (int i = 0; i < total - 1; i++) {
    for (int j = 0; j < total - 1 - i; j++) {
        // Leer producto en posición j
        fseek(f, j * sizeof(Producto), SEEK_SET); 1
        fread(&a, sizeof(Producto), 1, f); 2
        // Leer producto en posición j+1
        fread(&b, sizeof(Producto), 1, f); 3
        if (a.precio > b.precio) {
            // Intercambiar: escribir b en j, a en j+1
            fseek(f, j * sizeof(Producto), SEEK_SET);
            fwrite(&b, sizeof(Producto), 1, f);
            fwrite(&a, sizeof(Producto), 1, f);
        }
    }
}
```



A = 320
B = 120

Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

```c
Producto a, b;
for (int i = 0; i < total - 1; i++) {
    for (int j = 0; j < total - 1 - i; j++) {
        // Leer producto en posición j
        fseek(f, j * sizeof(Producto), SEEK_SET); 1
        fread(&a, sizeof(Producto), 1, f); 2
        // Leer producto en posición j+1
        fread(&b, sizeof(Producto), 1, f); 3
        if (a.precio > b.precio) {
            // Intercambiar: escribir b en j, a en j+1
            fseek(f, j * sizeof(Producto), SEEK_SET); 4
            fwrite(&b, sizeof(Producto), 1, f);
            fwrite(&a, sizeof(Producto), 1, f);
        }
    }
}
```

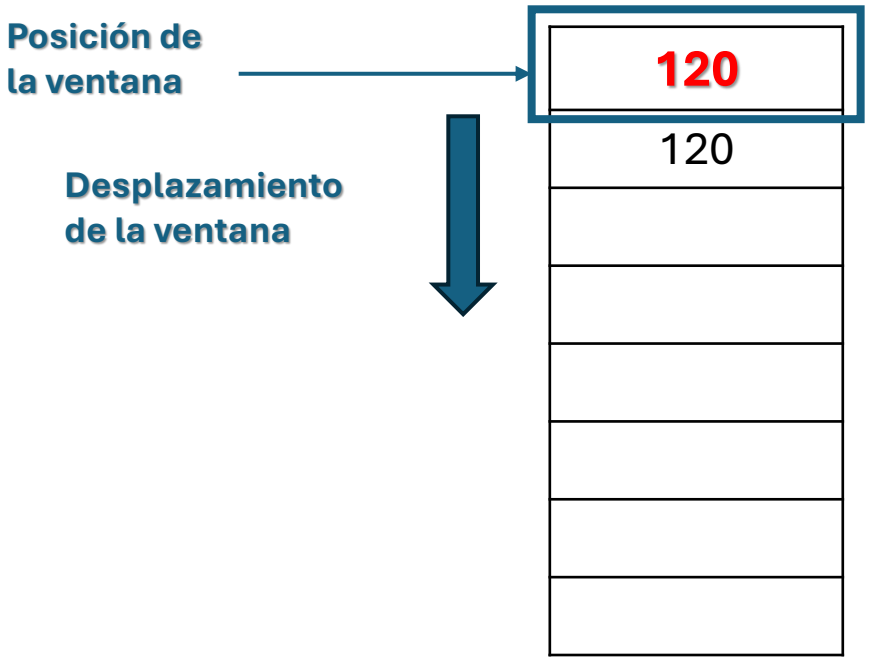


A = 320

B = 120

Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

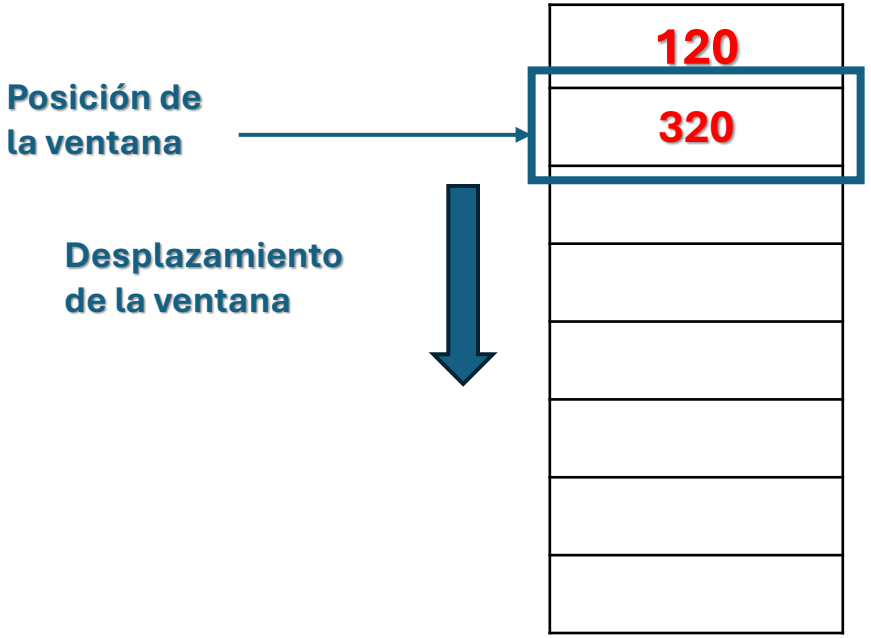
```c
Producto a, b;
for (int i = 0; i < total - 1; i++) {
    for (int j = 0; j < total - 1 - i; j++) {
        // Leer producto en posición j
        fseek(f, j * sizeof(Producto), SEEK_SET); 1
        fread(&a, sizeof(Producto), 1, f); 2
        // Leer producto en posición j+1
        fread(&b, sizeof(Producto), 1, f); 3
        if (a.precio > b.precio) {
            // Intercambiar: escribir b en j, a en j+1
            fseek(f, j * sizeof(Producto), SEEK_SET); 4
            fwrite(&b, sizeof(Producto), 1, f); 5
            fwrite(&a, sizeof(Producto), 1, f);
        }
    }
}
```



A = 320
B = 120

Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

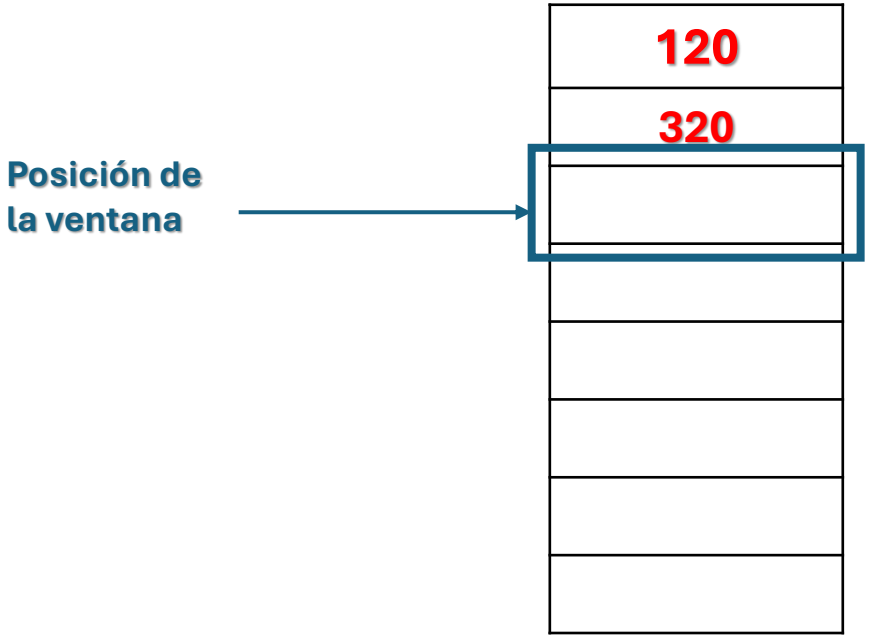
```c
Producto a, b;
for (int i = 0; i < total - 1; i++) {
    for (int j = 0; j < total - 1 - i; j++) {
        // Leer producto en posición j
        fseek(f, j * sizeof(Producto), SEEK_SET); 1
        fread(&a, sizeof(Producto), 1, f); 2
        // Leer producto en posición j+1
        fread(&b, sizeof(Producto), 1, f); 3
        if (a.precio > b.precio) {
            // Intercambiar: escribir b en j, a en j+1
            fseek(f, j * sizeof(Producto), SEEK_SET); 4
            fwrite(&b, sizeof(Producto), 1, f); 5
            fwrite(&a, sizeof(Producto), 1, f); 6
        }
    }
}
```



A = 320
B = 120

Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

```c
Producto a, b;
for (int i = 0; i < total - 1; i++) {
    for (int j = 0; j < total - 1 - i; j++) {
        // Leer producto en posición j
        fseek(f, j * sizeof(Producto), SEEK_SET);
        fread(&a, sizeof(Producto), 1, f);
        // Leer producto en posición j+1
        fread(&b, sizeof(Producto), 1, f);
        if (a.precio > b.precio) {
            // Intercambiar: escribir b en j, a en j+1
            fseek(f, j * sizeof(Producto), SEEK_SET);
            fwrite(&b, sizeof(Producto), 1, f);
            fwrite(&a, sizeof(Producto), 1, f);
        }
    }
}
```

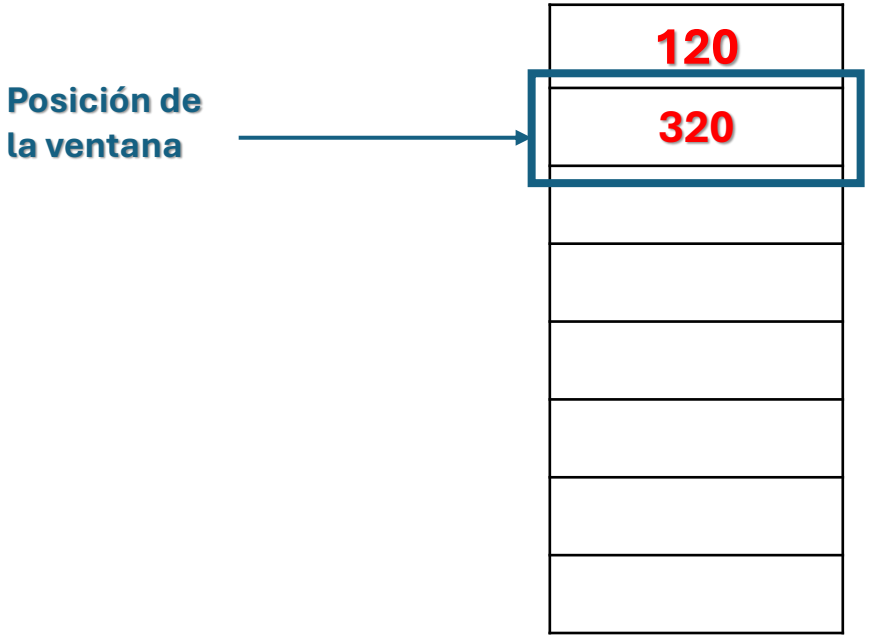


A = 320

B = 120

Ordenamiento externo con burbujeo (sobre el archivo) – ejemplo02.c

```c
Producto a, b;
for (int i = 0; i < total - 1; i++) {
    for (int j = 0; j < total - 1 - i; j++) {
        // Leer producto en posición j
        fseek(f, j * sizeof(Producto), SEEK_SET);
        fread(&a, sizeof(Producto), 1, f);
        // Leer producto en posición j+1
        fread(&b, sizeof(Producto), 1, f);
        if (a.precio > b.precio) {
            // Intercambiar: escribir b en j, a en j+1
            fseek(f, j * sizeof(Producto), SEEK_SET);
            fwrite(&b, sizeof(Producto), 1, f);
            fwrite(&a, sizeof(Producto), 1, f);
        }
    }
}
```



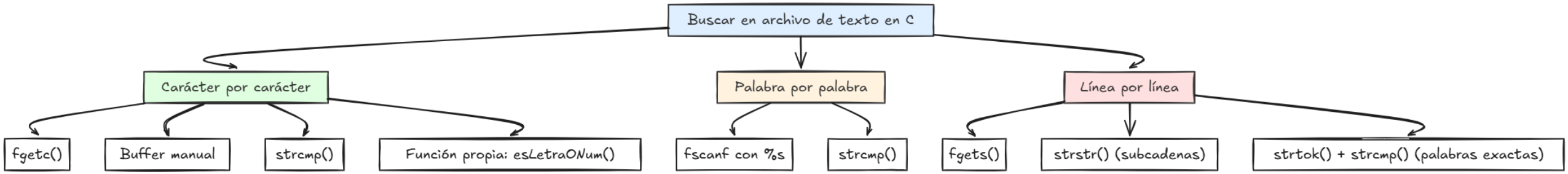
A = 320
B = 120

BUSQUEDA

Nuevamente: ¿En memoria o directo en el archivo?

- No hay mucho mas que agregar.
- Va a depender de cada caso.
- En esta clase solo vamos a ver como buscar directamente en el archivo.

Búsqueda en un archivo de texto



Búsqueda en un archivo de texto – carácter x carácter

Tenemos que considerar algunas reglas. Nosotros vamos a considerar que las palabras están formadas únicamente por letras o números, y separadas por algún símbolo.
Por ejemplo:
pepe lola casa 123
pepe;lola;casa;123

En ambas líneas podría llegar a encontrar, por ejemplo la palabra lola. Entonces necesito una función que determina si el carácter leído es letra o numero.

```
int esLetraONum(char c) {
    return ((c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z') ||
            (c >= '0' && c <= '9'));
}
```

Caracteres ASCII imprimibles

32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Búsqueda en un archivo de texto – carácter x carácter

ejemplo03.c

```c
const char objetivo[] = "error";
char buffer[100];
int idx = 0;
int c, contador = 0;

while ((c = fgetc(fp)) != EOF) {
    if (esLetraONum(c)) {
        if (idx < (sizeof(buffer) - 1)) {
            buffer[idx] = c;
            idx++;
            buffer[idx] = '\0';
        }
    } else {
        if (idx > 0) {
            if (strcmp(buffer, objetivo) == 0) {
                contador++;
            }
            idx = 0;
            buffer[0] = '\0';
        }
    }
}
```

Búsqueda en un archivo binario– referencia a través de un puntero - ejemplo04.c

```c
typedef struct {
    int codigo;
    char nombre[31];
    float precio;
} Producto;

int buscarProd(const char*, int , Producto*);

int main() {

    Producto p;
    if (buscarProd(ARCHIVO, 45, &p)) {
        printf("Producto %d: %s, precio=%.2f\n", p.codigo, p.nombre, p.precio);
    } else {
        printf("Producto con CODIGO 45 no encontrado.\n");
    return 0;
    }
}
```

Estoy enviando una dirección de memoria. En la función voy a poder afectar al valor apuntado por esa dirección de memoria.


COPIA DE ARCHIVOS

Copia de archivos

- Copiar archivos es una operación fundamental.
- El enfoque correcto en C es leer y escribir en bloques grandes de bytes, en lugar de byte a byte, para aprovechar la eficiencia de I/O.
- Usaremos **fread** y **fwrite** con un búfer intermedio.
- Este mismo proceso puede aprovecharse para transformar/modificar datos durante la copia, por ejemplo filtrando o alterando el contenido.

Ejemplo005.c: copia básica de un archivo binario

- Se aplica igualmente a texto, usando modo binario para evitar conversiones.
- El siguiente código copia un archivo origen en un destino, leyendo en bloques de tamaño fijo de 4kb.
- En la mayoría de los sistemas operativos modernos (Linux, Windows, macOS), el tamaño estándar de una página de memoria es de 4 KB.
- Esto significa que leer o escribir en bloques de 4 KB está alineado con cómo el sistema gestiona la memoria y el almacenamiento.
- Redondeamos decimos que 4kb es un valor típico eficiente.

Ejemplo06.c: modificación durante la copia

- Podemos insertar lógica dentro del bucle para alterar el contenido leído antes de escribirlo al destino.
- Por ejemplo, convertir a mayúsculas un archivo de texto, encriptar datos, cambiar delimitadores, etc.
- A continuación, modificamos el ejemplo para convertir todos los caracteres ‘,‘ en ‘;‘ al copiar (imaginemos un CSV donde queremos cambiar comas por punto y coma):

FIN DE LA CLASE