# Archivos Binarios, struct y Registros de Longitud Fija

.UBAfiuba
FACULTAD DE INGENIERÍA

**Fundamentos de Programación / Algoritmos y Programación I**

**Semana 8 · Clase Teórica · Prof. Servetto**

- Módulo struct: empaquetado/desempaquetado, endianness, calcsize
- Registros de longitud fija: acceso directo con seek por offset
- Mantenimiento de archivos (ABM): free list encadenada y bitmap
- Holístico: bloques del SO, fragmentación, buffer cache, lseek

# Agenda de la clase

| | | |
|---|---|---|
| 1 | **¿Por qué archivos binarios?** | Motivación; conexión con Semanas 2, 5, 6, 7 |
| 2 | **El módulo struct** | pack/unpack, cadenas de formato, endianness, calcsize |
| 3 | **Registros de longitud fija** | Diseño del registro, acceso directo con seek, in situ |
| 4 | **ABM con free list encadenada** | Baja lógica, reutilización LIFO, cabecera del archivo |
| 5 | **Registros de longitud variable** | Log append-only y slotted pages (mención conceptual) |
| 6 | **Sistema de archivos del SO [Holístico]** | Bloques, fragmentación interna, salvedad SSD, buffer cache |
| 7 | **Serialización y cierre multicapa** | struct / JSON Lines / JSON / pickle por granularidad |

# 1. ¿Por qué archivos binarios?

Tres limitaciones del texto que el binario resuelve simultáneamente.

## Limitaciones de los archivos de texto

### Tamaño variable

El número 42 ocupa 2 caracteres; 65535 ocupa 5. No se puede calcular dónde empieza el k-ésimo registro.

### Conversión costosa

Cada lectura requiere convertir caracteres a número (parseo). Cada escritura, número a caracteres (formateo).

### Sin acceso directo

Para llegar al k-ésimo dato hay que recorrer todos los anteriores: O(k). El acceso directo es imposible.

## Lo que aporta el archivo binario

Almacenan los bytes *tal cual están en memoria*. Un entero de 4 bytes ocupa siempre 4 bytes, sea el valor 0, 42 o 2 147 483 647.

Esa regularidad es la llave que abre el **acceso directo**: el k-ésimo registro empieza en el byte `k × tamaño_registro`, alcanzable con una sola llamada al SO.

### 🔗 Conexión con semanas previas

Semana 2: representación binaria de enteros (complemento a dos) y reales (IEEE 754) — esos bytes son exactamente lo que ahora vamos a escribir al disco. Semana 5: búsqueda binaria sobre arreglos ordenados — ahora sobre archivos binarios. Semana 6: archivos de texto — el contraste explica por qué necesitamos otra cosa. Semana 7: punteros y modelo de memoria — los registros tendrán "punteros" en forma de índices físicos.

# 2. Texto vs. Binario: misma información, distinta forma

Tres enteros — 42, 1000, 65535 — guardados en cada formato.

## Archivo de texto

*Tamaño variable — delimitadores explícitos*




- 14 bytes en total
- Tamaño de cada número: variable (2, 4 y 5 dígitos)
- Para leer el tercer número, recorrer todo lo anterior
- Acceso secuencial — O(n)

## Archivo binario (3 × int32 little-endian)

*Tamaño fijo — sin delimitadores*




- 12 bytes en total (3 × 4 bytes)
- Tamaño de cada número: fijo (4 bytes, siempre)
- El tercer número comienza en el byte 8
- Acceso directo — O(1) con `seek(8)`

**Sin separadores y con tamaño fijo:** el tercer entero comienza en el offset 2 × 4 = 8, alcanzable con una sola operación. Esa es la propiedad clave.

# 3. El módulo struct

La traductora entre valores Python y bytes crudos.

```python
import struct

# Empaquetar dos enteros como 8 bytes
datos = struct.pack('<ii', 42, 1000)
print(datos.hex())          # 2a000000e8030000
# Desempaquetar de vuelta
a, b = struct.unpack('<ii', datos)
print(a, b)                 # 42 1000
```

### Cadenas de formato más usadas

| Carácter | Tipo C / Python | Tamaño |
|---|---|---|
| `b / B` | int8 / uint8 | 1 byte |
| `h / H` | int16 / uint16 | 2 bytes |
| `i / I` | int32 / uint32 | 4 bytes |
| `q / Q` | int64 / uint64 | 8 bytes |
| `f / d` | float / double IEEE 754 | 4 / 8 B |
| `Ns` | cadena de N bytes (rellena con 0) | N bytes |

**Endianness: el orden de los bytes**

El entero `0x12345678` ocupa 4 bytes. ¿En qué orden se almacenan? `'<i'` little-endian **78 56 34 12** (x86, ARM) · `'>i'` big-endian **12 34 56 78** (POWER) · `'!i'` network byte order = big-endian (TCP/IP).

**Regla práctica:** siempre prefijar `'<'` o `'>'` para que el archivo sea portable entre arquitecturas.

`TAM_REGISTRO = struct.calcsize(FORMATO)` — única fuente de verdad: ningún literal "92" debe aparecer hardcodeado en el código.

# 4. Registros de longitud fija

El archivo se comporta como un arreglo en disco: acceso al k-ésimo en O(1).

### Registro de agenda — formato

*FORMATO = '<i32s16s40s'  →  TAM_REGISTRO = 92 bytes*




### El archivo como arreglo de registros — acceso directo con seek




**`archivo.seek(2 * TAM_REGISTRO)`**

*El cursor salta al Registro 2 sin leer los anteriores — costo O(1).*

```
def leer_contacto(archivo, k):                # O(1) sin importar el tamaño del archivo
    archivo.seek(k * TAM_REGISTRO)            # posición del k-ésimo
    datos = archivo.read(TAM_REGISTRO)        # exactamente 92 bytes
    return desempaquetar_contacto(datos)
```

# 5. ABM con free list encadenada

Baja lógica + reutilización del espacio liberado en O(1).

**Formato extendido: `'<Bi i32s16s40s'` → 97 bytes**

*Se agregan dos campos al inicio: activo (1 byte) + siguiente_libre (4 bytes), antes del id, nombre, teléfono y email.*

**Estado del archivo tras: 5 altas, baja(1), baja(3)**

| Reg 0 | Reg 1 | Reg 2 | Reg 3 | Reg 4 |
|---|---|---|---|---|
| activo = 1 | activo = 0 | activo = 1 | activo = 0 | activo = 1 |
| *(datos)* | siguiente_libre = -1 | *(datos)* | siguiente_libre = 1 | *(datos)* |

**`cabeza_libre = 3`**

**Free list = pila LIFO (Last In - First Out) de huecos disponibles**

**`baja(k)`**: el registro k se inserta al frente de la lista (push). **`alta(...)`**: si la lista no está vacía, se reutiliza el primer registro libre (pop) — O(1) sin recorrer el archivo. La próxima `alta` del ejemplo escribirá Margaret Hamilton en el slot 3 (no en el 1).

**Patrón: el archivo como estructura de datos persistente.**
La cabeza de la lista vive en la cabecera del archivo (entre ejecuciones); cada nodo guarda el índice del siguiente libre dentro del propio registro borrado. Mismo concepto que las listas enlazadas de Semana 10 — aquí el "puntero" es un índice físico al archivo.

# 5.5. Alternativa: bitmap de registros libres

La técnica que efectivamente usan ext4, NTFS y APFS para sus bloques.

| Aspecto | Free list encadenada | Bitmap auxiliar |
|---|---|---|
| **Costo estructural** | 4 bytes por registro (siguiente_libre) | 1 bit por registro (~0.125 B) |
| **Alta con reuso** | O(1) — consume la cabeza | O(1) amortizado con cursor |
| **Baja** | O(1) | O(1) |
| **Estadísticas** | Recorrer la cadena — O(libres) | bin(byte).count('1') — O(n/64) |
| **Riesgo** | Pérdida de cabeza ⇒ pérdida de huecos | Desincronización de archivos |
| **Uso real** | Sistemas pequeños, ad-hoc | ext4, NTFS, APFS, motores DBMS |

```
# Operaciones sobre bitmap (registro k está en byte k//8, posición k%8)

def bit_libre(bitmap, k):    return (bitmap[k // 8] >> (k % 8)) & 1 == 1

def marcar_libre(bitmap, k):  bitmap[k // 8] |= (1 << (k % 8))

def marcar_ocupado(bitmap, k):  bitmap[k // 8] &= ~(1 << (k % 8))
```

# 6. Registros de longitud variable: dos escenarios

El problema se bifurca según el patrón de uso. Los implementaremos como desafíos.

*Datos transaccionales*

## Log append-only

Cada registro se agrega al final, nunca se modifica ni se borra. Lleva un **prefijo de longitud** que indica cuántos bytes siguen.

**Casos de uso reales:**

- WAL (Write-Ahead Log) de PostgreSQL y SQLite (manejadores de bases de datos)
- Segmentos de Apache Kafka (event streaming)
- Event stores en CQRS (Command Query Responsibility Segregation)
- Logs estructurados de servicios

**Ventajas:** altas O(1) al final, sin compactación, sin fragmentación interna.

*Datos maestros con ABM*

## Slotted pages

El archivo se divide en **bloques de tamaño fijo** (4 KiB típico). Dentro de cada bloque conviven los registros de longitud variable y un directorio de ranuras (slots).




**Geometría del bloque:**

- Cabecera con campos de control: cantidad de registros almacenados en el Bloque, offsets a comienzo de espacio libre (para agregar registros) y a fin de espacio libre (para agregar slots)
- **Datos** (registros de longitude variable) crecen **desde el inicio →**
- **Espacio libre**
- **Directorio** (offset Ri, longitud Ri) crece **← desde el final**

**Casos de uso reales:** PostgreSQL, SQL Server, InnoDB (MySQL).

Para elegir bloque sin abrirlos todos: **FSM** (Free Space Map) auxiliar.

**Tratamiento conceptual en clase.** Implementación completa propuesta como desafíos extracurriculares grupales.

# 7. Serialización: una taxonomía por granularidad

La pregunta verdadera no es "binario vs texto", sino "objeto completo vs colección de registros".

| | Pregunta A — Objeto completo | Pregunta B — Colección de registros |
|---|---|---|
| **Texto** | **`JSON plano`**<br>Configuraciones, diccionarios, listas que se cargan enteras.<br>Legible, portable, seguro. | **`JSON Lines`**<br>Un JSON por línea. Streaming, escalable.<br>ELK, CloudWatch, datasets ML. |
| **Binario** | **`pickle`**<br>Cualquier objeto Python: instancias, árboles, grafos.<br>⚠ Riesgo de seguridad. | **`struct + reg. fijos`**<br>Acceso directo O(1), máximo rendimiento.<br>El núcleo de esta semana. |

**Nunca:** JSON plano ni pickle para bases de datos que crecen indefinidamente — la herramienta equivocada falla en producción.

# 8. Sistema de archivos del SO [Holístico]

Bloques, fragmentación, buffer cache y la salvedad SSD.

### Bloques de 4 KiB y fragmentación interna

El SO transfiere bloques completos (4096 bytes en ext4/NTFS/APFS), nunca bytes individuales.

Un archivo de 10 000 bytes ocupa 12 288 bytes en disco (3 bloques): los 2 288 bytes finales son **fragmentación interna** inevitable.

Implicancia: leer 10 bytes o 4000 bytes del mismo bloque cuesta lo mismo. Cruzar la frontera entre dos bloques duplica el costo.

### La salvedad de 2026: la mayoría de los discos son SSD

La narrativa clásica de "fragmentación externa = degradación de performance" se construyó sobre HDD con cabezales móviles, donde los seeks largos eran caros. En SSD *(memoria flash NAND, sin partes móviles)* acceder al bloque 10 o al 10 000 000 cuesta prácticamente lo mismo: la fragmentación externa **ya no degrada la performance** y desfragmentar un SSD es contraproducente.

**Lo que sí queda:** la fragmentación *interna* (desperdicio del último bloque) sobrevive por igual en HDD y SSD — afecta a nuestros archivos binarios.

### Buffer cache

archivo.write() escribe primero a RAM (page cache del kernel). El vuelco al disco es asíncrono. fsync() lo fuerza.

### lseek() syscall

archivo.seek() de Python se traduce a la syscall POSIX lseek: sólo mueve el cursor, no transfiere datos. La I/O sucede en el read/write siguiente.

### Network byte order

TCP/IP define big-endian para sus cabeceras (htonl/ntohl). El prefijo '!' de struct produce datos compatibles con la red.

# Un patrón en cuatro capas: free lists everywhere

La estructura que implementamos hoy se repite en todos los niveles del sistema.

| Capa | Entidad administrada | Dónde vive el "puntero al siguiente libre" |
|---|---|---|
| **Nuestra agenda (Sec. 5)** | Registros de 97 bytes | Campo siguiente_libre dentro del registro borrado |
| **Heap del proceso (Sem. 7)** | Bloques liberados por el GC | Estructura interna del allocator de CPython |
| **Sistema de archivos** | Bloques de disco de 4 KiB | Mapa de bits o lista enlazada (ext4 usa bitmap) |
| **Memoria virtual del kernel** | Marcos de página (4 KiB) | Listas de marcos libres por zona |

### 🔗 El mismo patrón a distintas escalas

En cada capa el problema es el mismo: **mantener un conjunto de unidades disponibles para asignar y devolver al conjunto las unidades liberadas**. La elección específica del mecanismo (free list, mapa de bits, árbol de huecos por tamaño) depende del patrón de uso, pero la estructura conceptual es siempre la misma.

Quien implementó a mano la free list de la agenda y entendió por qué funciona tiene en sus manos la idea esencial de cómo el SO gestiona la memoria, cómo el sistema de archivos gestiona los bloques de disco, y cómo una base de datos gestiona sus páginas. ***Mismo patrón, distintas escalas.***

# Tabla resumen: del tipeo al silicio

Seis capas entre el nombre tipeado y los electrones en una celda flash.

| Capa | Qué hace | Resultado intermedio |
|---|---|---|
| **Usuario** | Tipea nombre, teléfono, email en la interfaz | Cadenas Python en RAM |
| **Nuestro código Python** | struct.pack('<Bi i32s16s40s', ...) | 97 bytes empaquetados |
| **Intérprete CPython** | archivo.seek(...) y archivo.write(bytes) | Llamada a la libc |
| **Biblioteca C / Kernel** | syscalls lseek y write | Datos en buffer cache (RAM kernel) |
| **Kernel del SO** | Vuelca buffers asíncronamente; actualiza metadatos | Bloque listo para descenso al medio |
| **Controlador SSD** | FTL traduce dirección lógica → celdas NAND; wear leveling | Electrones atrapados en celdas flash |

**Tres principios fundamentales del trabajo con archivos binarios:**

**1.** El archivo es la única verdad persistente; las representaciones en RAM son transitorias. **2.** Las constantes de formato son fuente única de verdad (calcsize, no literales).

**3.** El acceso eficiente al disco es por bloques, no por bytes — el diseño debe respetar esa granularidad.

# Visión de conjunto



## Semanas 1–7

Representación binaria de enteros y reales (S2), búsqueda binaria (S5), archivos de texto (S6), modelo de memoria con punteros y referencias (S7) — todo eso es la base sobre la que construimos hoy los archivos binarios.



## Semana 8 (hoy)

struct, registros de longitud fija, acceso directo por offset, ABM con free list o bitmap, registros variables (log append-only / slotted pages), serialización por taxonomía de granularidad, conexión holística con el sistema de archivos del SO.



## Semana 9

Recursividad y manejo de la pila de ejecución; introducción a diccionarios y tablas hash. La free list que implementamos hoy reaparecerá conceptualmente al ver listas enlazadas en la Semana 10.



## Semanas 10–16

Estructuras de datos (Semana 10), técnicas algorítmicas avanzadas (Semanas 11–12) y proyecto integrador final (Semanas 13–16), donde los archivos binarios volverán como medio natural de persistencia.

*« El archivo binario no es un detalle de implementación: es el puente entre la representación interna que conocemos desde la Semana 2 y la realidad física del medio de almacenamiento. »*