

# Tema 3: Ficheros

## Programación 2

---

Grado en Ingeniería Informática  
Universidad de Alicante  
Curso 2025-2026



1. Introducción
2. Ficheros de texto
3. Ficheros binarios

# Introducción

---

## Qué es un fichero (1/3)

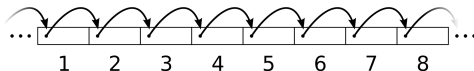
- Todos los datos con los que hemos trabajado hasta ahora se almacenan en la memoria principal del ordenador (*RAM*)
- El tamaño de la memoria principal es bastante limitado (unos pocos Gigabytes)
- Todos los datos se borran cuando el programa termina (*memoria volátil*)

## Qué es un fichero (2/3)

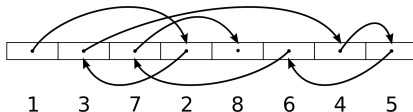
- Los *ficheros* (o *archivos*) son la forma en la que C++ permite acceder a la información almacenada en disco (memoria secundaria)
- Los ficheros son estructuras dinámicas: su tamaño puede variar durante la ejecución del programa según los datos que almacena
- Existen dos tipos de ficheros, en función de cómo se guarda dentro la información: *ficheros de texto* y *ficheros binarios*

## Qué es un fichero (3/3)

- Hay dos formas de acceder a un fichero:
  - *Acceso secuencial*: leemos/escribimos los elementos del fichero en orden, empezando por el principio y uno detrás de otro



- *Acceso directo (o aleatorio)*: nos situamos en cualquier posición del fichero y lo leemos/escribimos directamente, sin pasar por los anteriores



## Ficheros de texto

---

## Definición (1/2)

- Los ficheros de texto también se denominan *ficheros con formato*
- Guardan la información en forma de secuencias de caracteres, tal como se mostrarían por pantalla
- Por ejemplo, el valor entero 19 se guardará en fichero como los caracteres 1 y 9
- Ejemplos de ficheros de texto: un código fuente en C++, una página web (HTML) o un fichero creado con el bloc de notas
- El modo de lectura/escritura más habitual en ficheros de texto es el acceso secuencial



## Definición (2/2)

- Son ficheros que contienen solamente caracteres imprimibles: aquellos cuyo código ASCII es mayor o igual que 32
- El código ASCII es un código que asigna a cada carácter un número para su almacenamiento en memoria:

Dec	Hex	Car	Dec	Hex	Car	Dec	Hex	Car	Dec	Hex	Car
0	00	NUL	32	20	SPC	64	40	@	96	60	'
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	END	36	24	\$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

# Declaración de variables

- Los ficheros son un tipo de dato más en C++
- Hay que incluir la librería `fstream` en nuestro código para poder trabajar con ellos:

```
#include <fstream>
```

- Existen tres tipos de datos básicos para trabajar con ficheros, dependiendo de lo que queramos hacer con ellos:\*

```
ifstream ficheroLec; // Leer de fichero  
ofstream ficheroEsc; // Escribir en fichero  
fstream ficheroLecEsc; // Leer y escribir en fichero
```

\*Es poco habitual usar el tipo `fstream` con ficheros de texto

## Apertura y cierre (1/4)

- Una variable de tipo fichero (*fichero lógico*) se ha de asociar a un fichero real en el sistema (*fichero físico*) para poder leer/escribir en él
- Para establecer esta relación entre la variable y el fichero físico hay que hacer la apertura del fichero mediante `open`:

```
ifstream fichero; // Vamos a leer del fichero
fichero.open("miFichero.txt");
// Ahora ya podemos leer de "miFichero.txt"
```

- El nombre del fichero se puede pasar como un array de caracteres o como un `string`:\*

```
char nombreFichero[]="miFichero.txt";
fichero.open(nombreFichero);
```

\*Solo se puede usar `string` a partir de la versión 2011 de C++

## Apertura y cierre (2/4)

- A `open` se le puede pasar un segundo parámetro que indica el *modo de apertura* del fichero:
  - Lectura: `ios::in`
  - Escritura: `ios::out`
  - Lectura/escritura: `ios::in | ios::out`
  - Añadir al final: `ios::out | ios::app`

```
ifstream ficheroLec;  
ofstream ficheroEsc;  
// Abrimos solo para leer  
ficheroLec.open("miFichero.txt",ios::in);  
// Abrimos para añadir información al final  
ficheroEsc.open("miFichero.txt",ios::out|ios::app);
```

- Si abrimos un fichero que ya existe para escritura (`ios::out`) se borrará todo su contenido
- Si abrimos con `ios::app` no borrará su contenido, sino que irá añadiendo la nueva información al final
- Si el fichero no existe, se creará uno nuevo con tamaño inicial 0

## Apertura y cierre (4/4)

- Por defecto, el tipo `ifstream` se abre para lectura y el `ofstream` para escritura
- Se puede abrir el fichero en el momento de declararlo:

```
ifstream fl("miFichero.txt"); // Por defecto ios::in
ofstream fe("miFichero.txt"); // Por defecto ios::out
```

- Antes de leer/escribir, se debe comprobar con `is_open` si el fichero se ha abierto correctamente (`true`) o no (`false`)
- Al terminar de usar el fichero, se debe liberar con `close`:

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
    // Ya podemos trabajar con el fichero
    ...
    fl.close(); // Cerramos el fichero
}
else // Mostrar error de apertura
```

## Lectura con el operador >> (1/3)

- La lectura de fichero permite recuperar información guardada en disco para ponerla en memoria y poder trabajar con ella
- Podemos utilizar el operador >> para leer de fichero igual que hacíamos con `cin` para leer de teclado
- Bucle para leer un fichero carácter a carácter:

```
ifstream fl("miFichero.txt")
if(fl.is_open()){
    char c; // Podríamos leer int, float, ...
    while(fl >> c){ // Lee mientras queden caracteres
        cout << c;
    }
    fl.close()
}
else{
    cout << "Error al abrir el fichero" << endl;
}
```

## Lectura con el operador >> (2/3)

- Al usar el operador >> descartamos los blancos, al igual que sucedía al leer de `cin`
- Podemos usar la función `get` para leer carácter a carácter sin descartar blancos:

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
    char c;
    while(fl.get(c)){
        cout << c;
    }
    fl.close();
}
else{
    cout << "Error al abrir el fichero" << endl;
}
```



## Lectura con el operador >> (3/3)

- También se puede usar el operador >> para leer ficheros que contengan distintos tipos de datos
- Por ejemplo, si tenemos un fichero que contiene en cada línea una cadena y dos enteros (ej. Hola 1032 124):

```
ifstream fl("miFichero.txt");  
if(fl.is_open()){  
    string s;  
    int num1,num2;  
    while(fl >> s){ // Lee el string  
        fl >> num1; // Lee el primer número  
        fl >> num2; // Lee el segundo número  
        cout << s << "," << num1 << "," << num2 << endl;  
    }  
    fl.close()  
}  
...
```

- Podemos usar la función `getline` para leer una línea completa de fichero, al igual que hacíamos al leer de `cin`:

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
    string s;
    while(getline(fi,s)){
        cout << s << endl;
    }
    fl.close();
}
else{
    cout << "Error al abrir el fichero" << endl;
}
```

# Detección de final de fichero

- El método `eof` nos indica si se ha alcanzado el final de fichero
- Esta circunstancia se da cuando no quedan más datos por leer:

```
ifstream fl;  
...  
while(!fl.eof()){  
    // Leemos utilizando algunos de los métodos vistos  
}
```

- Cuando se intenta leer datos que quedan fuera del fichero, el método devuelve `true`
- Después de haber leído el último dato válido el método sigue devolviendo `false`
- Es necesario hacer una lectura más para provocar que `eof` devuelva `true`

## Escritura con el operador <<

- Podemos utilizar el operador << para escribir en fichero igual que hacíamos con cout para escribir por pantalla:

```
ofstream fe("miFichero.txt");
if(fe.is_open()){
    int num=10;
    string s="Hola, mundo";
    fe << "Un numero entero: " << num << endl;
    fe << "Un string: " << s << endl;
    fe.close();
}
else{
    cout << "Error al abrir el fichero" << endl;
}
```

### Ejercicio 1

Implementa un programa que lea un fichero `fichero.txt` e imprima por pantalla las líneas del fichero que contienen la cadena `Hola`.

### Ejercicio 2

Haz un programa que lea un fichero `fichero.txt` y escriba en otro fichero `FICHERO.TXT` el contenido del fichero de entrada con todas las letras en mayúsculas.

Ejemplo:

<code>fichero.txt</code>	<code>FICHERO.TXT</code>
Hola, mundo.	HOLA, MUNDO.
Como estamos?	COMO ESTAMOS?
Adios, adios...	ADIOS, ADIOS...

## Ejercicios (3/6)

### Ejercicio 3

Haz un programa que lea dos ficheros de texto, `f1.txt` y `f2.txt`, y escriba por pantalla las líneas que sean distintas en cada fichero, con `<` delante si la línea corresponde a `f1.txt` y con `>` si corresponde a `f2.txt`.

Ejemplo:

f1.txt	f2.txt
hola, mundo.	hola, mundo.
como estamos?	como vamos?
adios, adios...	adios, adios...

La salida debe ser:

```
< como estamos?  
> como vamos?
```

### Ejercicio 4

Diseña una función `finFichero` que reciba dos parámetros: el primero debe ser un número entero positivo `n` y el segundo el nombre de un fichero de texto. La función debe mostrar por pantalla las `n` últimas líneas del fichero.

Ejemplo:

```
finFichero(3,"cadenas.txt")
```

```
with several words
```

```
unapalabra
```

```
muuuuchas palabras, muchas, muchas...
```



### Ejercicio 4 (continuación)

Hay dos soluciones:

1. A lo bestia: leer el fichero para contar las líneas que tiene y volver a leer el fichero para escribir las  $n$  líneas finales.  
Problema: ¿y si el fichero tiene 1000000000000000 de líneas?
2. Utilizar un array de `string` de tamaño  $n$  que almacene en todo momento las  $n$  últimas líneas leídas (aunque al principio tendrá menos de  $n$  líneas)

## Ejercicios (6/6)

### Ejercicio 5

Tenemos dos ficheros de texto, `f1.txt` y `f2.txt`, en los que cada línea es una serie de números separados por `:`. Cada línea está ordenada por el primer número, de menor a mayor, en los dos ficheros. Haz un programa que lea los dos ficheros, línea por línea, y escriba en un fichero `f3.txt` las líneas comunes a ambos ficheros.

Ejemplo:

<code>f1.txt</code>	<code>f2.txt</code>	<code>f3.txt</code>
10:4543:23	10:334:110	10:4543:23:334:110
15:1:234:67	12:222:222	15:1:234:67:881:44
17:188:22	15:881:44	20:111:22:454:313
20:111:22	20:454:313	

# Ficheros binarios

---

## Definición (1/2)

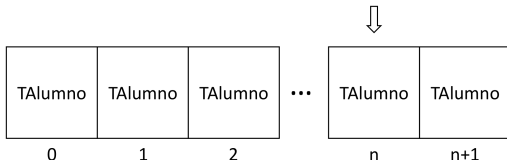
- También se le denominan *ficheros sin formato*
- Guardan la información tal y como se almacena en la memoria principal del ordenador
- Por ejemplo, el valor entero 19 se guardará en fichero como la secuencia 00010011
- Para leer y escribir se utilizan funciones diferentes a las de los ficheros de texto
- Con los ficheros binarios se suele emplear tanto acceso secuencial como directo
- Las lecturas y escrituras son más rápidas que con los ficheros de texto (no hay que convertir a carácter)
- Generalmente ocupan menos espacio en disco que sus equivalentes ficheros de texto

## Definición (2/2)

- Es muy habitual que cada elemento del que se quiere guardar información se almacene en un *registro* (`struct`):

```
struct TAlumno{  
    char nombre[100];  
    int grupo;  
    float notaMedia;  
};
```

- Mediante acceso directo, se puede acceder directamente al elemento  $n$  del fichero sin tener que leer los  $n-1$  anteriores



- Se declaran igual que en los ficheros de texto:

```
#include <fstream> // Siempre que se trabaja con ficheros  
  
ifstream ficheroLec; // Leer de fichero  
ofstream ficheroEsc; // Escribir en fichero  
fstream ficheroLecEsc; // Leer y escribir en fichero
```

# Apertura y cierre

- Al abrir el fichero se debe indicar que es binario mediante el modo de apertura `ios::binary`:
  - Lectura: `ios::in | ios::binary`
  - Escritura: `ios::out | ios::binary`
  - Lectura/escritura: `ios::in | ios::out | ios::binary`
  - Añadir al final: `ios::out | ios::app | ios::binary`

```
ifstream ficheroLec;  
ofstream ficheroEsc;  
// Abrimos solo para leer en modo binario  
ficheroLec.open("miFichero.dat",ios::in | ios::binary);  
// Abrimos solo para escribir en modo binario  
ficheroEsc.open("miFichero.dat",ios::out | ios::binary);  
// Forma abreviada  
fstream ficheroLecEsc("miFichero.dat",ios::binary)
```

- Igual que con los ficheros de texto, se puede comprobar si está abierto con `is_open` y se cierra con `close`

## Lectura (1/3)

- Para leer de fichero binario utilizamos la función `read`
- Esta función recibe dos parámetros: el primero indica dónde se guardará la información leída de fichero y el segundo la cantidad de información (número de bytes) que se va a leer:\*

```
TAlumno alumno;
ifstream fichero;

fichero.open("miFichero.dat",ios::in | ios::binary);
if(fichero.is_open()){
    // En cada iteración leemos un registro TAlumno
    while (fichero.read((char *)&alumno, sizeof(TAlumno))){
        // Mostramos el nombre y la nota de cada alumno
        cout << alumno.nombre << ": " << alumno.nota << endl;
    }
    fichero.close();
}
```

\*Para saber el número de bytes que ocupa una variable se puede usar la función `sizeof`



## Lectura (2/3)

- Se puede leer directamente un elemento  $n$  del fichero sin tener que leer los  $n-1$  anteriores (acceso directo)
- La función `seekg` permite situarse en un punto específico del fichero
- Recibe dos parámetros: el primero indica cuántos bytes nos queremos saltar, mientras que el segundo indica el punto de referencia para hacer ese salto

```
// Tenemos un fichero con registros de tipo TAlumno  
ifstream fichero("miFichero.dat",ios::binary);  
TAlumno alumno;  
  
...  
  
// Podemos leer directamente el tercer registro  
// Nos saltamos los dos primeros registros  
fichero.seekg(2*sizeof(TAlumno),ios::beg);  
// Ya podemos leer el tercer registro  
fichero.read((char *)&alumno,sizeof(alumno));  
  
...
```

- Puntos de referencia posibles:
  - `ios::beg`: contando desde el principio del fichero
  - `ios::cur`: contando desde la posición actual
  - `ios::end`: contando desde el final del fichero
- Si el primer parámetro de `seekg` es un número negativo, la ventana de lectura se mueve hacia el principio del fichero:

```
ifstream fichero("miFichero.dat",ios::binary);
TAlumno alumno;
...
fichero.seekg(-1*sizeof(TAlumno),ios::end);
// Leemos el último registro del fichero
fichero.read((char *)&alumno,sizeof(TAlumno));
...
```

## Escritura (1/3)

- Para escribir en fichero binario utilizamos la función `write`
- Esta función recibe dos parámetros: el primero indica dónde está guardada la información que queremos escribir y el segundo la cantidad de información (número de bytes) que se van a escribir
- La sintaxis es muy parecida a la de `read`:

```
ofstream fichero("miFichero.dat", ios::binary);
TAlumno alumno;

if(fichero.is_open())
{
    strcpy(alumno.nombre, "Pepe Pi");
    alumno.notaMedia=7.8;
    alumno.grupo=5;

    fichero.write((const char *)&alumno, sizeof(TAlumno));
    fichero.close();
}
```

## Escritura (2/3)

- Al igual que para la lectura, se puede escribir directamente en el registro  $n$  sin tener que escribir los  $n-1$  anteriores
- La función `seekp` permite posicionarse para escritura (`seekg` es para lectura)
- Los parámetros son los mismos que para `seekg`:

```
ofstream fichero("miFichero.dat",ios::binary);
TAlumno alumno;
...
// Nos posicionamos para escribir el tercer registro
fichero.seekp(2*sizeof(TAlumno),ios::beg);
fichero.write((const char *)&alumno,sizeof(TAlumno));
...
```

- Si la posición a la que se va con `seekp` no existe en el fichero, éste se "alarga" para que se pueda escribir en él

- Para almacenar cadenas de caracteres en un fichero binario se deben usar arrays de caracteres, nunca `string`
- El problema de `string` es que es un dato de tamaño variable, por lo que no podemos tener registros que tengan todos el mismo tamaño
- Puede ser necesario recortar la cadena para que quepa en el registro antes de guardarlo en fichero:

```
const int TAM=20;
char cad[TAM];
string s;
...
strncpy(cad,s.c_str(),TAM-1); // Máximo 19 caracteres
cad[TAM-1]='\0';
```

- La posición actual (en bytes) de la ventana de lectura se puede obtener mediante la función `tellg` y la de escritura mediante `tellp`
- Se puede usar, por ejemplo, para calcular el número de registros de un fichero:

```
ifstream fichero("miFichero.dat",ios::binary);  
// Colocamos la ventana de lectura al final del fichero  
fichero.seekg(0,ios::end);  
// Calculamos el numero de registros TAlumno del fichero  
cout << fichero.tellg()/sizeof(TAlumno) << endl;
```

### Ejercicio 6

Tenemos un fichero binario `alumnos.dat` que tiene registros de alumnos con la siguiente información:

- `dni`: array de 10 caracteres
- `apellidos`: array de 40 caracteres
- `nombre`: array de 20 caracteres
- `grupo`: entero

Haz un programa que imprima por pantalla el DNI de todos los alumnos del grupo 7.

Ampliación: haz un programa que intercambie los alumnos de los grupos 4 y 8 (los grupos van del 1 al 10).

### Ejercicio 7

Dado el fichero `alumnos.dat` del ejercicio anterior, haz un programa que pase a mayúsculas el nombre y los apellidos del quinto alumno del fichero, volviéndolo a escribir en él.

### Ejercicio 8

Diseña un programa que cree el fichero `alumnos.dat` a partir de un fichero de texto `alu.txt` en el que cada dato (dni, apellidos, etc.) está en una línea distinta. Ten en cuenta que en el fichero de texto el dni, nombre y apellidos pueden ser más largos que los tamaños especificados para el fichero binario, en cuyo caso se deberán recortar.



### Ejercicio 9

Escribe un programa que se encargue de la asignación automática de alumnos en 10 grupos de prácticas. A cada alumno se le asignará el grupo correspondiente al último número de su DNI (a los alumnos con DNI acabado en 0 se les asignará el grupo 10). Los datos de los alumnos están en un fichero `alumnos.dat` con la misma estructura que en los ejercicios anteriores.

La asignación de grupos debe hacerse leyendo el fichero una sola vez y sin almacenarlo en memoria. En cada paso se leerá la información correspondiente a un alumno, se calculará el grupo que le corresponde y se guardará el registro en la misma posición.