

# Ficheros Binarios



Eva Lucrecia Gibaja Galindo

Dpto. Informática y Análisis Numérico

# Introducción

- **Ejemplo.** Supongamos una aplicación que debe guardar en un fichero una serie de  $n$  números enteros positivos (representados como `unsigned int`). Supongamos que  $n=5$  y se genera la siguiente secuencia:

12345, 34, 4567345, 8845, 98367485

- Esta serie puede almacenarse en formato texto o binario

# Introducción

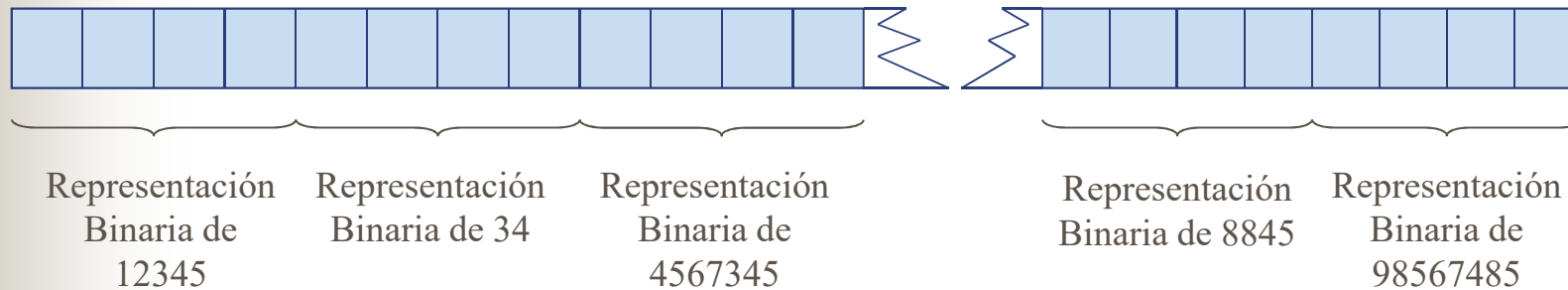
1. **Texto.** Almacenar cada entero en una línea del fichero o separados por un espacio o tabulador (indiferente)
  - a) Si `sizeof(unsigned int)` es 4, el máximo entero sin signo que puede utilizarse es 4294967295 → 10 cifras (`limits.h`)
  - b) Si los datos se generan de forma uniforme podemos tener valores desde 1 a 10 cifras con la misma probabilidad a priori
  - c) Cada número ocupa tanto espacio como cifras tiene
  - d) El tamaño del fichero es la suma del total de cifras de los números y separadores separadores: **31 bytes: 26 cifras y 5 separadores**

'1'	'2'	'3'	'4'	'5'	'\n'	'3'	'4'	'\n'	'4'	'5'	'6'	'7'	'3'	'4'	'5'
-----	-----	-----	-----	-----	------	-----	-----	------	-----	-----	-----	-----	-----	-----	-----

'\n'	'8'	'8'	'4'	'5'	'\n'	'9'	'8'	'5'	'6'	'7'	'4'	'8'	'5'	'\n'
------	-----	-----	-----	-----	------	-----	-----	-----	-----	-----	-----	-----	-----	------

# Introducción

2. **Binario.** Cada valor se almacena según su *representación interna* (copia exacta de como está en memoria)
- a) Cada valor se almacenaría exactamente con 4 bytes
  - b) No se necesitan separadores para delimitar cada valor
  - c) El tamaño del fichero será de **20 bytes: 4 bytes/dato 5 datos**



**Salvo en circunstancias muy extrañas, la adopción del formato binario para estos ficheros de datos resulta más ventajosa**



# Introducción

- **Flujo binario.** Sucesión de *bytes*. Más general que los flujos de texto
- **Registro activo.** Aquel que va a ser procesado en la siguiente operación con el fichero
- **Cursor.** Marca interna que siempre apunta al registro activo en cada momento. El apuntador se incrementa automáticamente cada vez que se procesa un registro (se lee o se escribe)
- **Clave.** Valor (o un conjunto de valores) de un registro que lo identifica unívocamente. Es diferente para todos los registros

# Creación y apertura de ficheros binarios

**FILE\* fopen (const char\* nombre, const char\*modo);**

- Abre el fichero físico cuyo nombre está referenciado por `nombre` y devuelve un puntero a `FILE` ó `NULL` si hay algún problema durante su apertura. La cadena `modo` puede ser:

Modo	Acciones	Cursor	Si existe	Si no existe
rb	Lectura	Inicio	Abre	Código error
wb	Escritura	Inicio	Borra contenido	Crea
ab	Adición	Final	Abre. Agrega al final	Crea
r+b	L/E	Inicio	Abre. Agrega al inicio, sobrescribiendo	Código error
w+b	L/E	Inicio	Borra contenido	Crea
a+b	L/Adición	Final	Abre. Agrega al final	Crea

# E/S de ficheros binarios

`size_t fread(void* ptr, size_t tam, size_t num, FILE* f);`

- Lee un máximo de `num` objetos de tamaño `tam` del fichero referenciado por `f` y los deja en el *buffer* referenciado por `ptr`
- Devuelve el número de objetos leídos, que puede ser menor que `num` si hay algún error o se alcanza el fin del fichero
- El cursor del fichero avanza los *bytes* leídos

`size_t fwrite(const void* ptr, size_t tam, size_t num, FILE* f);`

- Escribe en el fichero referenciado por `f` un máximo de `num` objetos de tamaño `tam` tomados del *buffer* referenciado por `ptr`
- Devuelve el número de objetos escritos, que puede ser menor que `num` si hay algún error
- El cursor del fichero avanza los *bytes* escritos

# El operador sizeof

- En muchos programas es necesario conocer el tamaño (cantidad de *bytes*) que ocupa una variable
- Este tamaño depende del compilador que se use
- Para mantener la portabilidad utilizaremos el operador llamado *sizeof*, que devuelve el tamaño en *bytes* de un tipo de dato
  - *sizeof(int) sizeof(char) sizeof(long double)* , etc.



# E/S de ficheros binarios. Ejemplo

```
#include <stdio.h>

#define TAM_BUF 1024 //Tamaño del buffer en bytes

typedef char byte;

int main()
{
    FILE *fOrigen, *fDestino;

    char nomOrigen[]="origen.doc", nomDestino[]="destino.doc";
    long leidos;
    byte buffer[TAM_BUF];

    if((fOrigen=fopen(nomOrigen, "rb") )==NULL)
    {
        printf("\nError: no pudeo abrir %s", nomOrigen);exit(-1);}

    if((fDestino=fopen(nomDestino, "wb") )==NULL)
    {
        printf("\nError: no pudeo crear %s", nomDestino); exit(-1);}
```

Lectura según el tamaño de un *buffer*

# E/S de ficheros binarios. Ejemplo

//En cada iteracion intentamos leer TAM\_BUF objetos

//de 1 byte. Escribimos lo que hemos leído

```
while((leidos=fread(buffer, sizeof(byte), TAM_BUF, fOrigen))>0)
```

```
{
```

```
    fwrite(buffer, 1, leidos, fDestino);
```

```
}
```

```
fclose(fOrigen);
```

```
fclose(fDestino);
```

```
return(0);
```

```
}
```

*nº de elementos*

*tamaño*

*sizeof() nos dice el tamaño de un tipo de dato*

# Funciones de posicionamiento

```
int fseek (FILE* f, long desp, int origen) ;
```

- Devuelve: *cierto* ( $\neq 0$ ) si se produce algún error
- Establece la posición actual del cursor
  - Ficheros binarios: la nueva posición se establece como un desplazamiento de *desp bytes* a partir de *origen*, que puede ser:
    - SEEK\_SET / 0 : inicio del fichero
    - SEEK\_CUR / 1: posición actual del cursor
    - SEEK\_END / 2 : fin del fichero
  - Ficheros de texto: *desp* debe ser:
    - 0L
    - Un valor devuelto por `ftell()`. En este caso, *origen* será SEEK\_SET

```
long ftell (FILE* f) ;
```

- Devuelve:
  - El número de *bytes* desde el principio hasta la posición actual del cursor
  - -1L si hay error

*Para ficheros de cualquier tipo!!*

# Tamaño de un fichero

```
#include <stdio.h>
int main()
{ FILE* f;
  long tam;
  char nombre[] = "origen.doc";
  if((f=fopen(nombre, "rb"))==NULL)
  {
    fprintf(stderr, "\nError: no puedo abrir <%s>", nombre);exit(-1);
  }
  if(fseek(f, 0L, SEEK_END))
  {
    fprintf(stderr, "\nError: no puedo usar <%s>", nombre); exit(-1);
  }
  tam = ftell(f);
  printf("\n<%s>: %ld bytes %5.2fKbytes",nombre,tam, tam/1024.0);
  fclose(f);
  return(0);
}
```

*ftell() devuelve el numero de bytes desde el principio del fichero hasta la posición actual*

*1Kbyte =  $2^{10}$  bytes = 1024 bytes*



# Numero de registros de un fichero binario

*Hay que conocer el tamaño de los registros*

$$n^{\circ} \text{ registros} = \frac{\text{bytes en fichero}}{\text{bytes / registro}}$$

```
void tamano(char* nombreFichero)
{
    FILE* f;
    long tam;

    if((f=fopen(nombreFichero, "rb") )==NULL)
    {
        fprintf(stderr, "\nError: no puedo abrir el fichero <%=s>", nombreFichero);
        exit(-1);
    }
    if(fseek(f, 0L, SEEK_END))
    {
        fprintf(stderr, "\nError: no puedo usar el fichero <%=s>", nombreFichero);
        exit(-1);
    }
    tam = ftell(f);
    printf("\nFichero:<%=s>%ld bytes%5.2f Kbytes", nombreFichero, tam, tam/1024.0);
    printf("\n\tContenido: %d registros", tam/sizeof(struct cliente));
    fclose(f);
}
```

# Otras Funciones

**int fflush (FILE\* f);**

- Fuerza a que el flujo *f* se libere: se vacía el *buffer* asociado al flujo de salida. El efecto está indefinido para flujos de entrada. Devuelve EOF si hay algún error de escritura y cero en otro caso
  - fflush(NULL) libera todos los flujos de salida
  - fflush(stdout) limpia el buffer del dispositivo de salida
  - fflush(stdin) su efecto está indefinido

**int remove (const char\* nombre);**

- Borra el fichero llamado nombre
  - Devuelve un valor distinto de cero en caso de error

**int rename (const char\* viejo, const char\* nuevo);**

- Cambia el nombre del fichero llamado *viejo* por el de *nuevo*
  - Devuelve un valor distinto de cero en caso de error

# Procesamiento de un fichero

- Listar registros de un fichero
- Fichero a vector
- Vector a fichero
- Añadir datos a un fichero
- Posicionamiento del apuntador
- Tamaño del fichero
- Número de registros
- Ver registro i-ésimo
- Consulta de datos (búsqueda en un fichero)
- Modificación de los datos de un registro
- Borrado de registros (físico y lógico)

Lectura elemento a elemento.

# Listar fichero

```
void listaFicheroClientes(char* nombreFichero)
{ FILE* f;
  int cuenta =0;
  struct cliente clienteAux;

  //abre el fichero en modo lectura binaria
  if((f=fopen(nombreFichero, "rb") )==NULL)
  { //Equivale a fprintf(stdin, "\nNo se puede abrir <%s>", nombreFichero);
    fprintf(stderr, "\nNo se puede abrir <%s>", nombreFichero);
  }
  else
  { while(fread(&clienteAux, sizeof(struct cliente), 1, f)==1)
    { //Si el registro no está marcado como borrado, lo escribimos
      if(strcmp(clienteAux.DNI, "") !=0)
        escribeCliente(clienteAux);
    }
    fclose(f);
  }
}
```

```
struct cliente
{ char DNI[MAX_DNI];
  int cuenta;
  char nombre[MAX_NOMBRE];
  float saldo;
};
```

*Si el campo tiene la marca de borrado (dni es ""), no se muestra*



# Fichero a vector (1)

Leemos  
MAX\_CLIENTES en  
una sola operación

```
void clientesAVector(char* nombreFichero, struct cliente Clientes[],
    int* tope)
{
    FILE* f;
    if((f=fopen(nombreFichero, "rb") )==NULL)
    {
        fprintf(stderr, "\nNo se puede abrir <%s>", nombreFichero);
    }
    else
    {
        //Al ser los vectores estáticos, como mucho podremos almacenar
        //MAX_CLIENTES
        *tope = fread(Clientes, sizeof(struct cliente), MAX_CLIENTES, f);
        fclose(f);
    }
}
```

## Fichero a vector (2)

Ojo con no salirnos de la memoria reservada

```
void clientesAVector2(char* nombreFichero, struct cliente Clientes[],
    int* tope)
{
    FILE* f;
    int cuenta =0;
    struct cliente clienteAux;
    if((f=fopen(nombreFichero, "rb"))==NULL)
    {fprintf(stderr, "\nNo se puede abrir <%s>", nombreFichero);}
    else
    { *tope=0;
        while(((cuenta = fread(&clienteAux, sizeof(struct cliente), 1,
            f))==1) && (*tope<MAX_CLIENTES))
        { if(strcmp(clienteAux.DNI, "")!=0)
            { Clientes[*tope]=clienteAux;
                *tope = *tope+cuenta;
            }
        }
        fclose(f);}}
```

# Vector a fichero

## Casos:

1. El fichero está recién creado  $\Rightarrow$  No tenemos que preocuparnos por el apuntador, situado en la cabecera del fichero
2. El fichero está creado y contiene datos  $\Rightarrow$  Para conservar la información existente hay que añadir los nuevos datos al final

# Vector a fichero. Sobrecribir

```
void clientesAFichero(char* nombreFichero, struct cliente
    Clientes[], int tope)
{
    FILE* f;
    if((f=fopen(nombreFichero, "wb") )==NULL)
    {
        fprintf(stderr, "\nError: no se puede abrir <%s>",
            nombreFichero);
    }
    else
    {
        //Escribe todos los clientes
        fwrite(Clientes, sizeof(struct cliente), tope, f);
        printf("\n\tFichero <%s> salvado a disco", nombreFichero);
        fclose(f);
    }
}
```

Escribimos todo el vector  
con un solo fwrite



Esta función es igual que clientesAFichero(), sólo cambia el modo de apertura

## Vector a fichero. Añadir

```
void addClientes(char* nombreFichero, struct cliente Clientes[],
    int tope)
{
    FILE* f;
    if((f=fopen(nombreFichero, "ab"))==NULL)
    {
        fprintf(stderr, "\nNo se puede abrir <%s>", nombreFichero);
    }
    else
    {
        //Escribe todos los clientes válidos del vector
        fwrite(Clientes, sizeof(struct cliente), tope, f);
        fclose(f);
    }
}
```

Escribimos todo el vector con un solo fwrite

# Búsqueda en un fichero

```
void listaSaldoSuperior(char* nombreFichero, float saldoTope)
{
    FILE* f;
    struct cliente cliAux;
    if((f=fopen(nombreFichero, "rb") )==NULL)
    {
        fprintf(stderr, "Error: no se puede abrir <%s>", nombreFichero);
    }
    else
    {
        printf("\n\nMODULO DE LISTADO DE CLIENTES CON SALDO >%d", saldoTope);
        while(fread(&cliAux, sizeof(struct cliente), 1, f)==1)
        {
            if((cliAux.saldo>=saldoTope) && (strcmp(cliAux.DNI, "") !=0))
            {escribeCliente(cliAux); }
        }
        fclose(f);
    }
}
```

# Búsqueda en un fichero

```
void consultaSaldo(char* nombreFichero, char* dniCliente)
{ FILE* f;
  struct cliente cliAux;
  int encontrado=0;

  if((f=fopen(nombreFichero, "rb"))==NULL)
  {fprintf(stderr, "Error: no se puede abrir <%s>", nombreFichero);}
  else
  { printf("\n\nMODULO DE CONSULTA DE SALDO:");
    while((fread(&cliAux, sizeof(struct cliente), 1, f)==1)&&(!encontrado))
    {
      if(strcmp(cliAux.DNI, dniCliente)==0)
      { escribeCliente(cliAux);
        encontrado = 1;
      }
    }
    fclose(f);
    if(!encontrado)
    {printf("\nNo existe el cliente: <%s>", dniCliente);}
  }
}
```

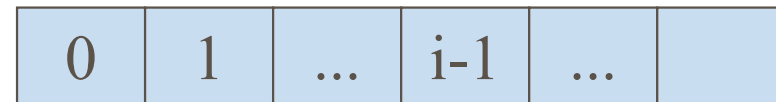
# Ver registro i-ésimo

```
struct cliente verRegistro_i(char *nombreFichero, int i)
{FILE *f;
 struct cliente cliAux;
 strcpy(cliAux.DNI, "");

 if(i>tamano(nombreFichero))
  return(cliAux);

 f = fopen(nombreFichero, "rb");
 fseek(f, i*sizeof(struct cliente), SEEK_SET);
 fread(&cliAux, sizeof(struct cliente), 1, f);

 fclose(f);
 //devuelve el registro leido
 return cliAux;
}
```



$Reg_0$   $Reg_1$  ...  $Reg_{i-1}$  ...



# Modificar un registro

1. Localizar en el fichero el registro a modificar
  1. Situar el apuntador al principio del fichero
    - Abriendo el fichero
    - Con una función específica
  2. Leer registro a registro el fichero y almacenándolos en una variable auxiliar
  3. Comparar con el contenido de algún campo (generalmente el campo clave)
2. Una vez encontrado el registro, realizar las modificaciones sobre la variable auxiliar
3. Escribir la variable auxiliar en el fichero en la misma posición en que estaba
  - Para ello hay que retroceder el apuntador, pues la lectura ha avanzado el cursor

# Modificar un registro

*1. Localizar el registro*



nombre: fulanito
cod: 5
saldo: 2000

*copiar en registro auxiliar (avanza el puntero!!)*

*2. Modificar registro auxiliar*



nombre: fulanito
cod: 5
<b>saldo: 2050</b>

*3. Retoceder el puntero*



nombre: fulanito
cod: 5
saldo: 2050

*4. Escribir registro*

# Modificar un registro

```
void modificaSaldo(char* nombreFichero, char* dniCliente, float nuevoSaldo)
{ FILE* f;
  struct cliente cliAux;
  int encontrado=0;

  if((f=fopen(nombreFichero, "r+b")!=NULL)
  {fprintf(stderr, "Error: no se puede abrir <%s>", nombreFichero);}
  else{
    while((fread(&cliAux, sizeof(struct cliente), 1, f)==1)&&(!encontrado)){
      if(strcmp(cliAux.DNI, dniCliente)==0){
        cliAux.saldo = nuevoSaldo; //Actualizamos registro local
        fseek(f, -(int)sizeof(struct cliente), SEEK_CUR); //Retrocedemos posicion
        fwrite(&cliAux, sizeof(struct cliente), 1, f); //Escribimos registro actualizado
        fflush(f); //el estandar requiere fflush para hacer fread despues de write
        encontrado = 1;
      }
    }
    fclose(f);
    if(!encontrado) {printf("\nNo existe el cliente: <%s>", dniCliente);}
  }
}
```

# Borrado físico vs. borrado lógico

- **Borrado lógico.** No es un borrado real del registro. Es una **actualización** que marca, de alguna forma los registros que ya no son válidos y utilizar esos huecos para futuras inserciones
- Formas de marcar:
  - Si existe un campo clave dándole un valor nulo: NULL, cadena vacía, etc. Problema si no es posible definir un valor nulo
  - Añadir a cada registro un campo adicional de tipo lógico borrado tal que si su valor es cierto, entendemos que el registro está borrado. Desventaja, cada registro tiene un campo adicional que aumenta el tamaño del fichero



# Borrado físico vs. borrado lógico

- **Borrado físico.** Eliminar un registro de un fichero para que el espacio que éste ocupa en el disco quede libre. Dos formas:
  1. Para borrar el registro de la posición  $i$  de un fichero de  $n$  registros, ir desplazando los registros de las posiciones  $i+1$ ,  $i+2$ , ...  $n$  a una posición menos, de manera que el de la posición  $i+1$  quede grabado sobre el de la posición  $i$ , etc. Muy costoso en tiempo. Si queremos borrar varios registros hay que repetir el proceso para todos
  2. Copiar en un nuevo fichero sólo los registros que se desee conservar. Inconveniente, necesidad de espacio en disco, pero es más eficiente
- Se realiza cuando los sistemas de la empresa no están siendo utilizados (durante la noche)
- Se borran a la vez todos los registros marcados para borrar

# Borrado lógico

```
void borradoLogico(char* nombreFichero, char* dniCliente)
{
    FILE* f;
    struct cliente cliAux;
    int encontrado=0;

    if((f=fopen(nombreFichero, "r+b"))==NULL)
    {
        fprintf(stderr, "\nNo se puede abrir <%s>", nombreFichero);
    }
    else
    {
        printf("\n\nMODULO DE BORRADO LOGICO POR DNI:");
        printf("\n-----");
    }
}
```

# Borrado lógico

```
while((fread(&cliAux, sizeof(struct cliente), 1, f)==1)&&(!encontrado))
{
    if(strcmp(cliAux.DNI, dniCliente)==0)
    { printf("\nDNI: %s", cliAux.DNI);
      printf("\n\tnombre: %s", cliAux.nombre);
      printf("\n\tcuenta: %d", cliAux.cuenta);
      printf("\n\tsaldo: %f ", cliAux.saldo);
      strcpy(cliAux.DNI, ""); //cadena vacia
      printf("\n\tdniNuevo: %s ", cliAux.DNI);
      fseek(f, -(int)sizeof(struct cliente), SEEK_CUR);
      fwrite(&cliAux, sizeof(struct cliente), 1, f);
      fflush(f);
      encontrado = 1;
    }
}
fclose(f);
if(!encontrado)
{ printf("\nNo existe el cliente: <%s>", dniCliente);}
}
```

*retroceso del cursor*



# Borrado físico

*El borrado físico requiere dos  
ficheros*

```
void borradoFisico(char* nombreFichero)
{
    FILE* f;
    FILE* faux;
    struct cliente cliAux;
    int encontrado=0;

    if( ((f=(FILE*) fopen(nombreFichero, "rb") )==NULL) ||
        ((faux=(FILE*) fopen("tmp"      , "wb") )==NULL) )
    {
        fprintf(stderr, "Error: no se puede abrir <%s> o <aux.tmp>",
            nombreFichero);
    }
    else
    {
        printf("\n\nMODULO DE BORRADO FISICO:");
        printf("\n-----");
    }
}
```



# Borrado físico

```
while(fread(&cliAux, sizeof(struct cliente), 1, f)==1)
{ if(strcmp(cliAux.DNI, "")!=0)
  {fwrite(&cliAux, sizeof(struct cliente), 1, faux);}
  else
  { printf("\nBORRADO: %s", cliAux.nombre);
    encontrado=1;
  }
}
//while
fclose(f);
fclose(faux);
if(!encontrado)
{ printf("\nNo hay registros para borrar");
  remove("tmp");
}
else
{ remove(nombreFichero);
  rename("tmp", nombreFichero);
}
}
//else
}
```

# Resumen

<b>Ficheros de texto</b>	L/E de caracteres	fgetc, getc, fputc, putc
	E/S Líneas	fscanf, fprintf, fgets, gets, fputs, puts
<b>Ficheros binarios</b>	L/E	fread, fwrite
	Posicionamiento	fseek-ftell
	Fin de fichero	feof (en texto es más complejo utilizarla)
	Otras	sizeof
<b>Ficheros de texto y binarios</b>	Apertura	fopen
	Cierre	fclose
	Renombrar y borrar	remove, rename
	Otras	fflush, rewind, freopen, ferror, tmpfile, tmpnam, fsetpos-fgetpos

# Resumen

## ■ Lectura de un fichero binario

### ■ Leer todo el fichero de una vez

```
*tope = fread(Clientes, sizeof(struct cliente), MAX_CLIENTES, f);
```

### ■ Leer elemento a elemento

```
while(fread(&clienteAux, sizeof(struct cliente), 1, f)==1){...}
```

### ■ Leer bloques de elementos

```
while((leidos=fread(buffer, sizeof(byte), TAM_BUF, fOrigen))>0){...}
```

### ■ Con *feof*

#### ■ Requiere lectura anticipada

- En ficheros de texto, si el fichero no termina en '\n', al salir del bucle *while*, tendremos que escribir la última línea leída. **Con ficheros de texto no utilizaremos *feof***

```
fread(...)
while(!feof())
{
    ... //Procesamiento
    fread()
}
```

# Material adicional





# La Función `fEOF()`

```
int fEOF (FILE* f) ;
```

- Devuelve:

- *falso* (0) si el cursor no ha sobrepasado el final de fichero
- *cierto* ( $\neq 0$ ) si el cursor ha sobrepasado el fin del fichero

- Requiere realizar **lectura anticipada** lo veremos en los ejemplos

- Una forma más general de comprobar si se ha alcanzado el fin del fichero

# Fichero a vector (3). Con feof()

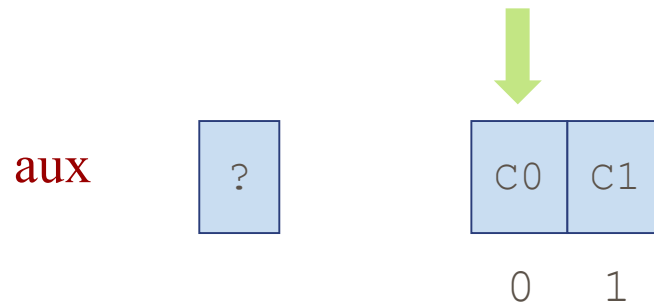
```
void clientesAVector3(char* nombreFichero, struct cliente Clientes[], int* tope)
{ FILE* f;
  int cuenta =0;
  struct cliente clienteAux;

  if((f=fopen(nombreFichero, "rb"))==NULL)
  {fprintf(stderr, "\nNo se puede abrir <%s>", nombreFichero);}
  else
  { *tope=0;
    cuenta = fread(&clienteAux, sizeof(struct cliente), 1, f);
    while((!feof(f))&&(*tope<MAX_CLIENTES))
    { //El cliente se pasa al vector si no tiene marca de borrado
      if(strcmp(clienteAux.DNI, "")!=0)
      { Clientes[*tope]=clienteAux;
        *tope = *tope+cuenta;}
      //Leemos el siguiente cliente
      cuenta = fread(&clienteAux, sizeof(struct cliente), 1, f);
    }
    fclose(f);
  }
}
```

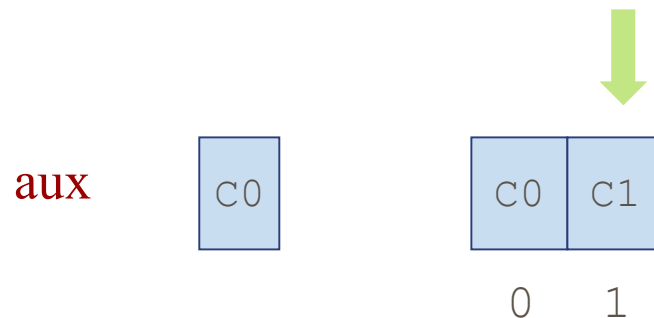
*lectura anticipada*

# Fichero a vector (3). Con feof()

1. Apertura del fichero en modo lectura



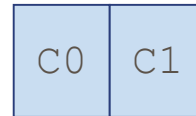
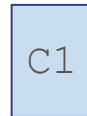
2. Lectura anticipada, leemos el cliente C0



# Fichero a vector (3). Con feof()

3. Leemos el cliente C1

aux



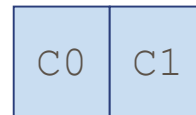
0 1



feof devuelve 0 porque aun no está activo el indicador de EOF

4. Intento de lectura, se activa el indicador de fin de fichero

aux



0 1



Al intentar hacer la lectura se activa el indicador de EOF y feof devuelve  $\neq 0$



# Búsqueda en un fichero. Con feof()

```
void listaSaldoSuperior2(char* nombreFichero, float saldoTope)
{ FILE* f;
  struct cliente cliAux;

  if((f=fopen(nombreFichero, "rb"))==NULL)
  {fprintf(stderr, "Error: no se puede abrir <%s>", nombreFichero);}
  else
  {
    printf("\n\nMODULO DE LISTADO DE CLIENTES CON SALDO >%d", saldoTope);
    fread(&cliAux, sizeof(struct cliente), 1, f);
    while(!feof(f))
    {
      if((cliAux.saldo>=saldoTope)&&(strcmp(cliAux.DNI, "")!=0))
      {escribeCliente(cliAux); }
      //Leemos el siguiente cliente
      fread(&cliAux, sizeof(struct cliente), 1, f);
    }
    fclose(f);
  }
}
```

*lectura anticipada*

# Búsqueda en un fichero. Con feof()

```
void consultaSaldo2(char* nombreFichero, char* dniCliente)
{ FILE* f;
  struct cliente cliAux;
  int encontrado=0;

  if((f=fopen(nombreFichero, "rb"))==NULL)
  {fprintf(stderr, "Error: no se puede abrir <%s>", nombreFichero);}
  else
  { printf("\n\nMODULO DE CONSULTA DE SALDO:");
    fread(&cliAux, sizeof(struct cliente), 1, f);
    while(!feof(f))
    { if(strcmp(cliAux.DNI, dniCliente)==0)
      { escribeCliente(cliAux);
        encontrado = 1;
      }
      fread(&cliAux, sizeof(struct cliente), 1, f);
    }
    fclose(f);
    if(!encontrado)
    {printf("\nNo existe el cliente: <%s>", dniCliente);}
  }
}
```

*lectura anticipada*

# Modificar un registro. Con feof()

```
void modificaSaldo2(char* nombreFichero, char* dniCliente, float nuevoSaldo)
```

```
{ FILE* f;
  struct cliente cliAux;
  int encontrado=0;

  if((f=fopen(nombreFichero, "r+b"))==NULL)
  {fprintf(stderr, "Error: no se puede abrir <%s>", nombreFichero);}
  else{
    fread(&cliAux, sizeof(struct cliente), 1, f);
    while(!feof(f) && (!encontrado)) {
      if(strcmp(cliAux.DNI, dniCliente)==0){
        cliAux.saldo = nuevoSaldo;
        fseek(f, -(int)sizeof(struct cliente), SEEK_CUR);
        fwrite(&cliAux, sizeof(struct cliente), 1, f);
        fflush(f); encontrado = 1;
      }
      fread(&cliAux, sizeof(struct cliente), 1, f);
    }
    fclose(f);
    if(!encontrado)
    {printf("\nNo existe el cliente: <%s>", dniCliente);}}}
```

*lectura anticipada*

# Borrado lógico. Con feof()

```
void borradoLogico2(char* nombreFichero, char* dniCliente)
{
    FILE* f;
    struct cliente cliAux;
    int encontrado=0;

    if((f=fopen(nombreFichero, "r+b"))==NULL)
    {
        fprintf(stderr, "\nNo se puede abrir <%s>", nombreFichero);
    }
    else
    {
        printf("\n\nMODULO DE BORRADO LOGICO POR DNI:");
        printf("\n-----");
    }
}
```



# Borrado lógico. Con feof()

```
fread(&cliAux, sizeof(struct cliente), 1, f);
while ((!feof(f)) && (!encontrado)) {
    if (strcmp(cliAux.DNI, dniCliente) == 0)
    { printf("\nDNI: %s", cliAux.DNI);
      printf("\n\t nombre: %s", cliAux.nombre);
      printf("\n\t cuenta: %d", cliAux.cuenta);
      printf("\n\t saldo: %f ", cliAux.saldo);
      strcpy(cliAux.DNI, ""); //cadena vacia
      printf("\n\t dniNuevo: %s ", cliAux.DNI);
      fseek(f, -(int)sizeof(struct cliente), SEEK_CUR);
      fwrite(&cliAux, sizeof(struct cliente), 1, f);
      fflush(f);
      encontrado = 1;
    }
    fread(&cliAux, sizeof(struct cliente), 1, f);
}
fclose(f);
if (!encontrado) { printf("\nNo existe el cliente: <%s>", dniCliente); }
}
```

*lectura anticipada*

*retroceso del cursor*

# Ejemplo. Ficheros de texto y feof()

```
void leeVersion2(char* nombreFichero)
{
    FILE* fich;
    char cadena[30];

    if((fich=fopen(nombreFichero, "r"))==NULL)
        printf("\nNo se ha podido abrir el fichero <%s>", nombreFichero);
    else
    {
        fgets(cadena, 30, fich);
        while(!feof(fich))
        {
            if(cadena[strlen(cadena)-1]=='\n')
                cadena[strlen(cadena)-1]='\0';
            printf("<%s>\n", cadena);
            fgets(cadena, 30, fich);
        }
        if(cadena[0]!='\n')
            printf("<%s>\n", cadena);
        fclose(fich);
    }
}
```

*Lectura anticipada*

*Si el fichero no termina en '\n' la ultima lectura contiene la última línea del fichero*

# Ejemplo.

```
void fileCopy(FILE* destino, FILE* fuente)
```

```
{ int c;
```

```
while((c=getc(fuente)) != EOF)
```

```
{putc(c, destino);}
```

```
}
```

---

```
void fileCopy(FILE* destino, FILE* fuente)
```

```
{ int c;
```

```
c=getc(fuente); Lectura anticipada
```

```
while(!feof(fuente))
```

```
{ putc(c, destino); //Escritura en destino
```

```
c=getc(fuente); //Nueva lectura
```

```
}
```

```
}
```

Esta función sólo es válida para ficheros de texto.

Esta función es válida para ficheros de cualquier tipo, ya que la condición de terminación se evalúa utilizando feof().

# Otras funciones de posicionamiento

```
void rewind (FILE* f);
```

- La traducción de `rewind` es *rebobinar*, esto es, *colocarse al principio del fichero*

- Equivalente a `fseek(f, 0L, SEEK_SET);`

```
int fgetpos (FILE* f, fpos_t* ptr);
```

- Asigna la posición actual del fichero referenciado por `f` a `*ptr`. El tipo `fpos_t` es adecuado para registrar tales valores. Devuelve un valor distinto de cero si hay error

```
int fsetpos (FILE* f, const fpos_t* ptr);
```

- Establece la posición actual del fichero referenciado por `f` con el valor dado por `*ptr`. Devuelve un valor distinto de cero si hay error
- Las funciones `fgetpos` y `fsetpos` son interfaces alternativas equivalentes a `ftell` y `fseek` (con el origen puesto a `SEEK_SET`). Utilizar mejor `ftell` y `fseek`



# Otras Funciones

**FILE\* freopen(const char\* nbre, const char\* modo, FILE\* f);**

- Cierra el fichero asociado con *f* y abre el fichero llamado *nbre* en el modo especificado por *modo* y lo asocia con el flujo *f*. Devuelve FILE\* ó NULL

**int ferror (FILE\* f);**

- Devuelve un valor distinto de 0 si se ha producido un error durante la última operación sobre el archivo

**FILE\* tmpfile();**

- Crea un fichero temporal (en modo “wb+”) que se borra al ser cerrado o cuando termina el programa normalmente. Devuelve FILE \* si todo va bien ó NULL en caso de error

**char\* tmpnam(char s[L\_tmpnam]);**

- Genera una cadena de caracteres que es un nombre válido para ficheros y que no es igual al nombre de un fichero existente. La función *tmpnam* genera una cadena diferente cada vez que es llamada