Tema 1. Fundamentos de C
Parte II

Diseño de Algoritmos
2º I.T.I. Electricidad
E.U. Politécnica

M. Carmen Aranda Garrido
Despacho: I-307

Departamento Lenguajes y Ciencias de la Computación.
Universidad de Málaga

Índice

10. Tipos Enumerados

- En **C**, un **Tipo Enumerado** consiste en una sucesión de **constantes enteras con nombre** que especifica todos los valores válidos para cierta variable.
  - Simplifican la programación haciendo que los programas sean más legibles, ya que usarían los símbolos (los nombres válidos).
  - **Sintaxis**: `enum nombre_tipo {lista_nombres} variables;`
  - **Ejemplo**: `enum semana {L, M, X, J, V, S, D} dia;`
    - Con ese ejemplo es correcto usar:

```
dia = S;
if (dia == X) ...
```


10. Tipos Enumerados (II)

- Cada **símbolo** de la enumeración tiene un **valor entero:** Se empieza en cero y para cada símbolo se suma 1: `L` vale `0`, `M` vale `1`, `X` vale `2`...
- Esos valores pueden alterarse:

```
enum semana {L, M, X, J, V, S=10, D} dia;
```

  - Así, `S` vale `10` y `D` vale `11`: El orden de los valores se mantiene.
- Los símbolos **NO pueden escribirse** directamente como si fueran cadenas de caracteres: **Son constantes enteras**.
- Usualmente el sistema **NO genera error** si no se respetan las restricciones de un tipo enumerado: Es responsabilidad del programador usarlo correctamente.

11. Clases de Almacenamiento de Variables

- **Clases de almacenamiento en C:** En una declaración la clase de almacenamiento puede especificarse antes de la declaración:
  - **`auto`, Variables Automáticas:** Son las **variables normales**, que se crean cuando se declaran y se destruyen cuando se termina la función en la que están declaradas (o el programa si son globales).
    - Es equivalente declarar `int i;` que `auto int i;`
  - **`extern`, Variables Externas:** Se refiere a que se trata de variables declaradas en un **módulo (fichero) distinto** a donde aparece la declaración con `extern`. •**Ej.:** `extern int i; /*` **Variable ya declarada en otro módulo** `*/`

11. Clases de Almacenamiento de Variables (II)

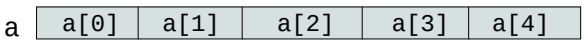
- **`register`, Variables Registro:** Son las variables que intentan ser almacenadas en algún **registro del procesador**, para que las operaciones sobre ellas sean más rápidas. Se usará en variables sobre las que recaigan múltiples operaciones (control de bucles...). No puede abusarse de este tipo de variables ya que, en tal caso, algunas variables no se almacenarán en registros. •**Ej.:** `register int i;`
- **`static`, Variables Estáticas:** Variables que se crean la primera vez que se ejecuta su declaración y no se destruyen al finalizar su función. Son como variables globales, aunque sólo son vistas en la función en la que se declaran.
  - La segunda vez (y las siguientes) que se llame a una función con una variable estática, la función "recordará" el valor que obtuvo la variable en la ejecución anterior, ya que la variable NO se destruyó al finalizar dicha ejecución anterior.
    - •**Ej.:** `static int i=9; /* Valor inicial */`
  - Una variable estática siempre debe **inicializarse** al declararse.
  - Si se usa `static` en una **var. global** será vista sólo en ese fichero.

12.1. Arrays

- DECLARACIÓN de un array unidimensional:

```
tipo_elemento  nombre_array [num_elementos];
```

EJEMPLO: `int a[5];`



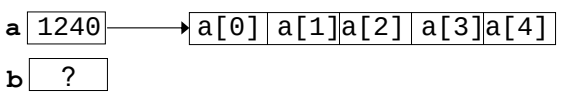
- Los elementos de un array se almacenan contiguos en memoria.
- El nombre de un array es un **"puntero constante"** que tiene la dirección del primer elemento del array. Su valor es constante, no se puede cambiar.

12.1. Arrays y Punteros

- Hay una fuerte relación entre **arrays y punteros**, pero los **arrays** y los **punteros** NO son equivalentes:

```
int a[5];

int *b;
```



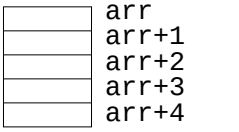
- **DIFERENCIAS:**
  - Declarar un array reserva memoria para **TODOS** los elementos.
  - Declarar un puntero reserva memoria **SÓLO** para el puntero.
  - El nombre de un array almacena una dirección fija (**constante**) que apunta al principio de este espacio. NO se puede cambiar el valor de la **constante**.
    - **El nombre de un array es la dirección de memoria del primer elemento:** `a ≡ &a[0]`
  - Una variable puntero almacena una dirección de memoria que puede ser modificada. La variable puntero NO está inicializada para apuntar a ningún espacio existente, por lo que inicialmente tiene cualquier valor no válido. Para indicar que un puntero no apunta a ningún sitio se le puede asignar el valor **NULL**.

12.1. Arrays y Punteros

- Los arrays y los punteros usan diferentes notaciones de indexación.
- Así, las siguientes expresiones son **equivalentes:** `float arr[5];`

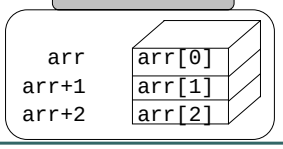
Usando índices	Usando punteros
`arr[0]`	`*(arr)`
`arr[1]`	`*(arr+1)`
`arr[2]`	`*(arr+2)`
`arr[3]`	`*(arr+3)`
`arr[4]`	`*(arr+4)`
`arr[índice]` ≡	**`*(arr+índice)`**

Direcciones

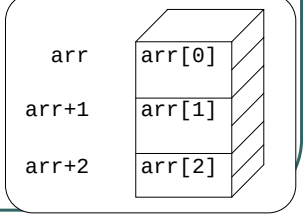


El número de posiciones que se incrementa un puntero depende del tipo de datos del array.
Suponemos que un **`int`** ocupa 2 bytes y un **`char`** 1 byte.

`char arr[3];`

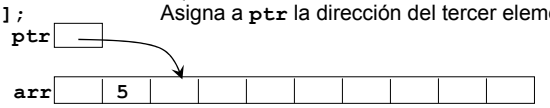


`int arr[3];`



12.1. Arrays y Punteros

Ejemplos	Descripción
`int arr[10];`	Declara el array **`arr`** con 10 elementos.
`int *ptr;`	Declara un puntero a entero.
`arr[1]=5;`	Asigna 5 al segundo elemento del array **`arr`**.
`*(arr+1)=5;`	Equivalente a la instrucción anterior.
`ptr=&arr[2];`	Asigna a **`ptr`** la dirección del tercer elemento:



`ptr`	Equivale a **`arr+2`** y a **`&arr[2]`**
`*ptr`	Equivale a **`arr[2]`** y a **`*(arr+2)`**
`ptr[0]`	Equivale a **`arr[2]`**
`ptr + 6`	Equivale a **`arr+8`** o **`&arr[8]`**
`*ptr + 6`	Equivale a **`arr[2] + 6`**
`*(ptr + 6)`	Equivale a **`arr[8]`**
`ptr[-1]`	Equivale a **`arr[1]`**
`ptr[9]`	Equivale a **`arr[11]`**: ¡Fuera del array **`arr`**!

Acceder fuera de las posiciones de un array es un **fallo muy grave** del que no avisa el compilador. Es responsabilidad del programador usar los arrays correctamente.

12.1. Arrays: Acceso con índices y punteros

- Los dos bucles siguientes ponen a 0 los elementos de un array:

Con índices:

```
int arr[10], i;

for(i=0; i<10; i++)
    arr[i]=0;
```

Se realizan multiplicaciones cuando se ejecuta.

Con punteros:

```
int arr[10], *ip;

for(ip=arr; ip<arr+10; ip++)
    *ip=0;
```

Se realizan sumas cuando se ejecuta.

- **El acceso con punteros es frecuentemente más rápido que con índices**.
- La mejora en rapidez es máxima cuando se accede al array de forma secuencial (a todos los elementos uno tras otro).
- Si el acceso es aleatorio (a cualquier elemento) con punteros no es más rápido que con índices.
- **DESVENTAJA: La notación se puede complicar usando punteros**, y la claridad del programa puede empeorar.

12.1. Arrays como argumentos.

- Si el argumento es un array se pasa **un puntero al primer elemento** del array.
- El paso de un array SIEMPRE tiene un comportamiento como si fuera un paso por REFERENCIA**:**
  - NO se puede modificar la dirección del array.
  - La función SI puede **modificar** los elementos del array.
- Por tanto, NO es necesario poner el **`&`** si se quiere modificar un array dentro de una función. Si se pone da ERROR.
- Ejemplo: PROTOTIPO de función que usa un array:

```
float  maximo(float arr[], int tama);
```

- Es imposible que la función determine el tamaño del array.
  `sizeof(arr)` da el tamaño del puntero, NO del array.
- Si se necesita el tamaño en la función se tiene que pasar como un argumento explícito (como **`tama`** en el ejemplo).

12.1. Arrays multidimensionales

Un array es **MULTIDIMENSIONAL** si tiene más de una dimensión.

```
tipo_elemento   nombre_array [a][b][c]...[z];
```

- Una **matriz** es un array de 2 dimensiones, es decir un array unidimensional de arrays unidimensionales.
- En general, un array de dimensión **`n`** es un array unidimensional de arrays de dimensión **`n-1`**.

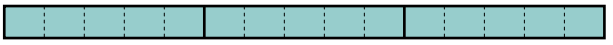
EJEMPLOS:

`int m[6][10];` Array bidimensional de 6 ×10 enteros (**matriz**)

`float arr[3][2][5];` Array tridimensional de 3×2×5 reales (**cubo**)

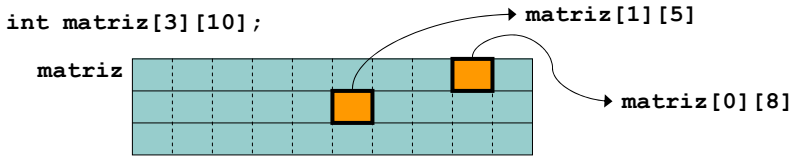
- **Los elementos se almacenan CONTIGUOS en memoria.**

`int array[3][5];` Son 15 enteros

array 

12.1. Arrays multidimensionales

- Para identificar un elemento de un array multidimensional, se debe dar **un índice para cada dimensión**, en el mismo orden que en la declaración.
- Cada índice se encierra en sus propios corchetes.



- Hay dos modos de escribir la lista de inicializaciones:

1. Todos los valores seguidos: `int matriz[2][3]={0,1,2,10,11,12};`

2. Por partes:

```
int   matriz[2][3]={
      { 0, 1, 2 },
      { 10, 11, 12 }};
```

12.1. Arrays multidimensionales

```
/* Programa que  muestra los valores almacenados en
   un array bidimensional 2 por 3 */

#include <stdio.h>
#define FILAS     2
#define COLUMNAS  3
int main() {
 float matriz[FILAS][COLUMNAS]=
  {{1,2,3},{4,5,6}};
 int fil,col;
 for(fil=0;fil<FILAS;fil++)
   for(col=0;col<COLUMNAS;col++)
     printf("El valor de [%d][%d] es %d\n",
        fil,col,matriz[fil][col]);
 return 0;
}
```

SALIDA EN PANTALLA

```
El valor de [0][0] es 1
El valor de [0][1] es 2
El valor de [0][2] es 3
El valor de [1][0] es 4
El valor de [1][1] es 5
El valor de [1][2] es 6
```


12.1. Arrays multidimensionales

- Un array **multidimensional** puede pasarse como argumento a una función.
- Se pasa un PUNTERO AL PRIMER ELEMENTO del array.
  - Recuerde que ese primer elemento es un array (de 1 dimensión menos).
- Se define la longitud de todas las dimensiones excepto la primera.
- Si varía el array dentro de la función también varía en la llamada.

EJEMPLO:

- Función que devuelve el valor máximo de un array tridimensional de reales de dimensión **`tx3x5`**(**`t`** indica el tamaño de la primera dimensión).

```
float max(float d[][3][5],int t);
```

Observe que esa función sirve para arrays de diversos tamaños: 4x3x5, 29x3x5 .....

12.2. Cadenas de Caracteres

- Una cadena de caracteres es un **array unidimensional de caracteres**.
- VENTAJA: Existen librerías con funciones para realizar la mayor parte de las operaciones básicas sobre cadenas.
- El último carácter visible de la cadena debe estar seguido del carácter nulo que se representa por **'\0'** (código ASCII 0).
- Este carácter marca el final de la cadena de caracteres.

DECLARACIÓN:

```
char cadena[20];
char cadena[10]="Hola";
char cadena[]="Adios";
```


12.2. Cadenas de Caracteres

- Una cadena de caracteres puede ser leída y asignada a un array utilizando **`scanf()`** con el modificador **`%s`**, o con **`gets()`** .
  - Con `scanf` se lee hasta el primer espacio, tabulador o retorno de carro.
  - Si se pone un tamaño `T` en el especificador de formado (`%Ts`) (ej. %5s), se lee como máximo ese número de caracteres.
  - Con `gets` se lee hasta el primer retorno de carro.
  - Para leer una frase (con espacios) de un tamaño concreto, se puede usar la función **`fgets()`**, cuyo prototipo es:

```
char *fgets(char s[], int n, stdin);
```

Lee caracteres hasta que lee n-1 o hasta que lee el carácter RETURN '\n', añadiendo el carácter '\0' al final. Si hay más de n-1 caracteres, quedan sin leer y podrán leerse después.

- Se escribe con **printf()** con el modificador **%s** o con **puts()**.

La cadena debe contener el carácter '\0' para ser escrita correctamente.

12.2. Cadenas de Caracteres

- Algunas funciones de manejo de cadenas (**`string.h`**)

`int strlen(char str[])`

- Determina la longitud de una cadena.

`int strcmp(char str1[],char str2[])`

- Compara dos cadenas.
- Devuelve 0 si son iguales, un número negativo si **`str1`** es menor que **`str2`** o un número positivo si **`str1`** es mayor que **`str2`**.

`char *strcpy(char str1[],char str2[])`

- Copia **`str2`** en **`str1`** (carácter a carácter).

`char *strcat(char str1[],char str2[])`

- Añade **`str2`** al final de **`str1`** (carácter a carácter).

12.3. Estructuras

- Una **estructura** es un grupo de información relacionada que se usa para almacenar datos acerca de un tema o actividad.
- Las estructuras están divididas en **CAMPOS**.
- Un campo es una variable de un tipo determinado.

SINTAXIS:

```
struct nombre {    /* Define el tipo de datos struct nombre. */
    campo1;        /* Los campos se definen como una variable,
    campo2; ...       indicando el tipo del campo y su nombre. */
    campoN;
};
 /* Posteriormente se pueden declarar variables de tipo struct nombre */
struct nombre var1 , ..., varN;
```

12.3. Estructuras: Acceso a los Campos

- Para acceder a cada campo se utiliza el operador punto (**.**)
- Se pone el nombre de una **variable** de tipo estructura seguida de un **punto** y luego el nombre del **campo**.

EJEMPLO:

```
struct trabajador{
  char nombre[20];
  char apellidos[40];
  int edad;
  char puesto[10];
};
struct trabajador fijo, temporal;
```

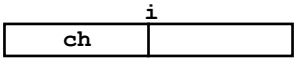

12.3. Estructuras. Ejemplo

```
void visualizar(struct trabajador);
main(){ /* Rellenar y visualizar */
  struct trabajador fijo;
  printf("Nombre: ");
  scanf("%s",fijo.nombre);
  printf("\nApellidos: ");
  scanf("%s",fijo.apellidos);
  printf("\nEdad: ");
  scanf("%d",&fijo.edad);
  printf("\nPuesto: ");
  scanf("%s",fijo.puesto);
  visualizar(fijo);
}
void visualizar(struct trabajador datos){
  printf("Nombre: %s",datos.nombre);
  printf("\nApellidos: %s",datos.apellidos);
  printf("\nEdad: %d",datos.edad);
  printf("\nPuesto: %s",datos.puesto);
}
```

12.4. Uniones

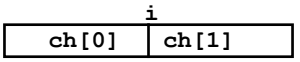
- **Uniones:** Es una forma de hacer que diferentes variables compartan la misma zona de la memoria.
  - Su declaración tiene la misma sintaxis que las estructuras, excepto que se usa la palabra **`union`** en vez de **`struct`**.
  - **Ejemplo:**

```
union tipo_union{
    int i;
    char ch;
} V;
```



  - Si a **`V.i`** se le asigna un valor, el primer byte de ese entero puede ser visto desde **`V.ch`**.
  - **Ejemplo:** Para acceder de forma individual al segundo byte de **`V.i`** podríamos declarar **`ch`** como un array de dos posiciones:

```
union tipo_union{
    int i;
    char ch[2];
} V;
V.i=258; /* En binario: 00000001 00000010 */
printf("%i-%i", V.ch[1], V.ch[0]);// Escribe: 1-2.
```



12.4. Características de las Uniones

- Ocupan tanto espacio en **Memoria** como el **mayor** de sus campos.
  - Las estructuras ocupan tanta memoria como la suma de todos sus campos.
- Una vez definida una **unión**, se pueden **declarar variables** de forma similar a las estructuras.
  - **Ejemplo:** `union tipo_union tu1, tu2;`
- Para **acceder a los campos** se accede, igual que en las estructuras, usando el operador punto (si tenemos una unión) y el operador flecha (si tenemos el puntero a una unión).
- Igual que ocurre con los campos de bits, el código puede ser **dependiente de la máquina:**
  - Existen máquinas que representan los datos de izquierda a derecha y de derecha a izquierda.
- Una unión puede tener **más de dos campos:** Todos los campos comparten la memoria.
  - Esos campos pueden ser de cualquier tipo, incluyendo arrays, estructuras, campos de bits...

12.5. Creación de Tipos de Datos

Se crean con la sentencia **typedef**.

```
typedef   definicion_tipo   nombre_tipo;
```

EJEMPLOS:

```
typedef int entero;
entero a, b=3;
typedef int tablero[8][8]; /* define un nuevo tipo de datos
                    llamado tablero que es una matriz 8x8 */
tablero t1,t2;   /* declaración de variables de ese tipo */
typedef struct {   /* Se crea un tipo de datos llamado
   double real;       complejo */
   double imag;
} complejo;
complejo c; /* Se declara una variable de ese tipo */
complejo x[10]; /* array unidimensional de 10 complejos */
```


12.5. Creación de Tipos de Datos

```
#include <stdio.h>
typedef struct {
   double real, imag;
} complejo;
void pedir_complejo(complejo& c);
void escribir_complejo (complejo c);
complejo suma_complejo(complejo c1, complejo c2);
int main(){
  complejo a,b,s;
  pedir_complejo(a); pedir_complejo(b);
  s=suma_complejo(a,b);
  escribir_complejo(s); return 0;
}
void pedir_complejo(complejo& c) {
  printf("Parte real? "); scanf("%lf", &(c.real));
  printf("Parte imaginaria? "); scanf("%lf", &(c.imag)); }
```

13. E/S por Ficheros:Operaciones Básicas

- Para operar con un **Fichero o Archivo** hay que seguir **3** pasos**:**
  - **1. Abrir el Fichero:** Prepara un fichero para poder utilizarlo.
    - La **función `fopen()`** (de **`stdio.h`**) devuelve un puntero a un fichero (**`FILE *`**), que es una estructura con la información que necesita el sistema para acceder al fichero (nombre, estado, posición actual...).
      - **`fopen()`** devuelve **`NULL`** si no se puede abrir el fichero (no existe, disco lleno, disco protegido...).
    - El **primer argumento de `fopen()`** es una cadena de caracteres con el **nombre** (y la ruta) del fichero que se desea abrir.

13. E/S por Ficheros:Operaciones Básicas

- El **segundo argumento de `fopen()`** es una cadena de caracteres con el **modo** con el que se desea abrir un fichero**:**

`r`	Abre un fichero ya existente sólo para lectura (***read***).
`w`	Crea un fichero para escritura (***write***). Si ya existe, lo borra.
`a`	Abre un fichero para añadir (escribir) datos al final (***append***). Si no existe, lo crea.
`r+`	Abre un fichero existente para actualización (lectura y escritura).
`w+`	Crea un fichero para actualización (lectura y escritura). Si ya existe, lo borra.
`a+`	Abre un fichero para actualización (lectura y escritura) al final. Si no existe, lo crea.

- Puede añadirse una **`'t'`** para indicar que el fichero es de **texto** y una **`'b'`** para indicar que el fichero es **binario**.
  - Los **ficheros de texto** ocupan más espacio que los **binarios**, pero pueden visualizarse (y modificarse) con cualquier editor de texto.

13. E/S por Ficheros:Operaciones Básicas

- **2. Operar con el Fichero:**
  - Las **operaciones típicas** con un fichero son **Leer y Escribir** y para estas operaciones hay muchas funciones que veremos a continuación.
  - Existen **otras operaciones** que pueden resultar útiles y que también veremos a continuación. Estas operaciones son, por ejemplo, situarnos en una posición concreta del fichero (para leer o escribir allí), ver si hemos llegado al final del fichero, situarnos al principio del fichero...

13. E/S por Ficheros:Operaciones Básicas

- **3. Cerrar el Fichero:** Finaliza la utilización de un fichero. Si no se hace se pueden perder los datos del fichero.
  - La función **`fclose()`** tiene como argumento un puntero a fichero (**`FILE *`**) y cierra un fichero previamente abierto con **`fopen()`**.

- **Ej.:**

```
FILE *fich; /* fich es un puntero a fichero */
fich = fopen("datos.txt", "rt");
if (fich==NULL)
   puts("\n- No se puede abrir el fichero.");
else { /* Operaciones de E/S en el fichero */
       ...
       fclose(fich);
}
```

13. E/S por Ficheros: Funciones de E/S (stdio.h)

- `int putc(int c, FILE *fich);`
  - Escribe el carácter `c` en el fichero `fich`.
  - Devuelve el carácter escrito, o bien, `EOF` si hubo algún error.
    - `EOF` (*End Of File*, Fin De Fichero) es una constante simbólica definida también en `stdio.h`.
- `int getc(FILE *fich);`
  - Lee un carácter del fichero `fich`.
  - Devuelve el carácter leído, o bien, `EOF` si hubo algún error (fin de fichero...).
- `int fprintf(FILE *fich, const char *format [, args]);`
  - Similar a la función `printf()` pero el resultado lo escribe en el fichero `fich`.
  - Ejemplo: Para escribir un texto con una variable real en el fichero `salida`: `fprintf(salida,"Jamones %f", peso);`

13. E/S por Ficheros: Funciones de E/S (stdio.h)

- `int fscanf(FILE *fich, const char *format [, direcs]);`
  - Similar a `scanf()` pero el origen de la lectura es el fichero `fich`.
- `int fputs(const char *cadena, FILE *fich);`
  - Escribe la cadena de caracteres en el fichero, sin añadir el carácter de nueva línea.
  - Devuelve el último carácter escrito o `EOF` si hubo error.
- `char *fgets(char *s, int n, FILE *fich);`
  - Lee caracteres del fichero y los almacena en `s`. Para de leer cuando lee `n-1` caracteres, el carácter de nueva línea o el carácter `EOF`.
  - Devuelve la cadena leída o `NULL` si hubo error.

13. E/S por Ficheros: Funciones de E/S (stdio.h)

- `size_t fwrite(void *p, size_t size,size_t n, FILE *f);`
  - Escribe, en un fichero **BINARIO**, `n` elementos de tamaño `size` (en bytes). Se supone que esos elementos están en la memoria a la que `p` apunta.
  - Permite la escritura de un bloque de datos apuntado por `p`.
  - El tipo `size_t` es `unsigned`.
  - Se escribirán un total de (`size*n`) bytes en **BINARIO**.
  - El puntero `p` puede ser la dirección de una variable, o bien, un bloque de memoria asignado dinámicamente (con `malloc()`, por ejemplo).
  - Ejemplo: Para escribir un array `V` de 8 enteros en el fichero `f`, usar:

```
fwrite(V, sizeof(int), 8, f);
```

- `size_t fread(void *p, size_t size,size_t n, FILE *f);`
  - Lee, de un fichero **BINARIO**, `n` elementos de tamaño `size` (en bytes). Los almacena en la memoria a la que `p` apunta (suponiendo que caben).

13. E/S por Ficheros: Funciones Útiles (stdio.h)

- `int feof(FILE *fich);`
  - Devuelve un valor distinto de cero si se alcanzó ya el final del fichero (EOF, End Of File) en la última lectura.
  - Devuelve cero si no se llegó al final.
- `int ferror(FILE *fich);`
  - Devuelve un valor distinto de cero si se produjo algún error en la última operación con el fichero.
  - Se debe llamar a esta función después de cada operación para garantizar que el proceso se realiza correctamente.
- `void rewind(FILE *fich);`
  - Inicializa el indicador de posición del archivo al principio del mismo (rebobinar).

13. E/S por Ficheros: Funciones Útiles (stdio.h)

- `int fseek(FILE *fich, long desplazamiento, int donde);`
  - Desplaza el indicador de posición del fichero tantos bytes como indique `desplazamiento`, a partir de la posición indicada por el argumento `donde`, que puede tomar 3 valores: `SEEK_SET` (a partir del inicio), `SEEK_CUR` (a partir de la posición actual) y `SEEK_END` (a partir del final del fichero).
  - Devuelve 0 si el desplazamiento se efectuó con éxito.
  - En ficheros de texto puede producir errores.
- `long ftell(FILE *fich);`
  - Devuelve el valor actual del indicador de posición de un archivo binario. Este valor puede usarse en la función `fseek()` para volver a la posición anterior.
  - Devuelve `-1L` (`-1` como `long`) si se produce un error.
  - Es similar a la función `fgetpos()`, la cual tiene una función inversa que es `fsetpos()`.

13. E/S por Ficheros: Funciones Útiles (stdio.h)

- `int fflush(FILE *fich);`
  - Si el fichero es un flujo (*stream*) abierto para escritura, entonces `fflush()` escribe en el fichero el contenido del *buffer*.
  - Si el fichero está abierto para lectura, entonces `fflush()` borra el contenido del *buffer*.
  - Devuelve cero si la operación se realizó correctamente y `EOF` si se produjo algún error.
  - Si se aplica sobre `stdin` borra el *buffer* de teclado.
- `int flushall(void);`
  - Limpia los *buffers* de todos los ficheros abiertos, devolviendo el número de ficheros abiertos (incluyendo los abiertos automáticamente).
- `int fcloseall(void);`
  - Cierra todos los ficheros excepto los abiertos automáticamente.

Ejemplo.

```
#include <stdio.h>
#include <conio.h>
struct registro
{
char nombre[40];
char edad[3];
}
main()
{

int op;
char opcion[5];
```

```
do{
 clrscr();
 printf("1. alta\n");
 printf("2. baja\n");
 printf("3. consulta\n");
 printf("4. listado\n");
 printf("5. modificación\n");
 printf("6. salir\n");
 gets(opcion);
 op=atoi(opcion);
 if (op==1)
   altas();
 if (op==4)
   listado();
}while (op!=6);
```


Ejemplo.

```
altas(){
 char fichero[40];
 FILE *f;
 struct registro persona;
 clrscr();
 strcpy(fichero,"DATOS.DAT");
 f=fopen(fichero,"a");
 if (f==NULL)
   f=fopen(fichero,"w");

 printf("Nombre: ");
 gets(persona.nombre);
 printf("Edad: ");
 gets(persona.edad);
 fwrite(&persona,sizeof(persona),1,f);
 fclose(f);
}
```

Ejemplo.

```
listado(){
 char fichero[40];
 FILE *f;
 struct registro persona;

 clrscr();
 strcpy(fichero,"DATOS.DAT");
 f=fopen(fichero,"r");
 while (!feof(f)){
   clrscr();
   fread(&persona,sizeof(persona),1,f);
   printf("Nombre: %s",persona.nombre);
   printf("Edad: %s",persona.edad);
   getche();
 }
 fclose(f);
}
```