1 Fundamentos de C

Contenido

1. **Estructura de un programa.**
2. **La función main.**
3. **Tipos de datos.**
4. **Declaración de funciones.**
5. **Declaración de variables.**
6. **Sentencias de control.**
7. **Directiva #define.**
8. **Ficheros.**

1. Estructura de un programa.

Todo programa en **C** consta de una o más funciones, una de las cuales se llama **main**. El programa comienza en la función main, desde la cual es posible llamar a otras funciones.

Cada función estará formada por la cabecera de la función, compuesta por el nombre de la misma y la lista de argumentos (si los hubiese), la declaración de las variables a utilizar y la secuencia de sentencias a ejecutar.

Ejemplo:

```
declaraciones globales

main( ) {
        variables locales
        bloque
}

funcion1( ) {
        variables locales
        bloque
}
……..
……..
```

1.1.- Comentarios

A la hora de programar es conveniente añadir comentarios (cuantos más mejor) para poder saber que función tiene cada parte del código, en caso de que no lo utilicemos durante algún tiempo. Además facilitaremos el trabajo a otros programadores que puedan utilizar nuestro archivo fuente.

Para poner comentarios en un programa escrito en **C** usamos los símbolos **/*** y ***/**:

```
/* Este es un ejemplo de comentario */

/* Un comentario también puede
estar escrito en varias líneas */
```

El símbolo **/*** se coloca al principio del comentario y el símbolo ***/** al final.

El comentario, contenido entre estos dos símbolos, no será tenido en cuenta por el compilador.

1.2.- Palabras clave.

Existen una serie de indicadores reservados, con una finalidad determinada, que no podemos utilizar como identificadores.

A continuación vemos algunas de estas palabras clave:

char	**int**	**float**	**double**
else	**do**	**while**	**for**
switch	**short**	**long**	**extern**
static	**default**	**continue**	**break**
register	**sizeof**	**typedef**	**if**

1.3.- Identificadores.

Un identificador es el nombre que damos a las variables y funciones. Está formado por una secuencia de letras y dígitos, aunque también acepta el caracter de subrayado **_**. Por contra no acepta los acentos ni la **ñ**/**Ñ**.

El primer caracter de un identificador no puede ser un número, es decir que debe ser una letra o el símbolo **_**.

Se diferencian las mayúsculas de las minúsculas, así **num**, **Num** y **nuM** son distintos identificadores.

A continuación vemos algunos ejemplos de identificadores válidos y no válidos:

Válidos	**No válidos**
_num	1num
var1	número2
fecha_nac	año_nac

2. La función main.

La función main() puede tomar argumentos como cualquier otra función. Lo mejor del asunto es que los argumentos pueden ser pasados a main(), exteriormente, una

vez que el programa ha sido compilado y linkeado desde el sistema operativo, en tiempo de ejecución.

Los argumentos o parámetros que main(int argc,char *argv[]) puede recibir desde el exterior se trata de igual forma que los argumentos de cualquier otra función.

El parámetro argc recoge automáticamente el número de items presente en la llamada desde el DOS al programa ejecutable. Así, una llamada a 'pepe.exe' del tipo:

Pepe archivo1 archivo2

hará que argc tome 3 como valor.

El parámetro argv[] es un array de punteros de tipo char,que apunta a los items o cadenas de caracteres, recibidas desde el exterior.

En este caso, las cadenas apuntadas por cada puntero de argv[] son:

argv[0]	"pepe"
argv[1]	"archivo1"
argv[2]	"archivo2"
argv[3]	puntero NULL

Tradicionalmente, argc y argv[] se designan por estos nombre, pero pueden utilizar cualquier identificador o nombre de variable que se ajuste a los tipos descritos.

El que argv[] se declare de tipo char, no significa que usted esté limitado a transferir a main() datos de tipo char. Recuerde que en C existen funciones de transformación de tipos e datos.

3. Tipos de datos.

3.1.- Tipos

En 'C' existen básicamente cuatro tipos de datos, aunque como se verá después, podremos definir nuestros propios tipos de datos a partir de estos cuatro. A continuación se detalla su nombre, el tamaño que ocupa en memoria y el rango de sus posibles valores.

TIPO	Tamaño	Rango de valores
char	1 byte	-128 a 127
int	2 bytes	-32768 a 32767
float	4 bytes	3'4 E-38 a 3'4 E+38
double	8 bytes	1'7 E-308 a 1'7 E+308

3.2.- Calificadores de tipo

Los calificadores de tipo tienen la misión de modificar el rango de valores de un determinado tipo de variable. Estos calificadores son cuatro:

- **signed**

  Le indica a la variable que va a llevar signo. Es el utilizado por defecto.

	tamaño	rango de valores
signed char	1 byte	-128 a 127
signed int	2 bytes	-32768 a 32767

- **unsigned**

  Le indica a la variable que no va a llevar signo (valor absoluto).

	tamaño	rango de valores
unsigned char	1 byte	0 a 255
unsigned int	2 bytes	0 a 65535

- **short**

  Rango de valores en formato corto (limitado). Es el utilizado por defecto.

	tamaño	rango de valores
short char	1 byte	-128 a 127
short int	2 bytes	-32768 a 32767

- **long**

  Rango de valores en formato largo (ampliado).

	tamaño	rango de valores
long int	4 bytes	-2.147.483.648 a 2.147.483.647
long double	10 bytes	-3'36 E-4932 a 1'18 E+4932

También es posible combinar calificadores entre sí:

signed long int = long int = long	
unsigned long int = unsigned long	**4 bytes**

(0 a 4.294.967.295 (El mayor entero permitido en 'C'))

4. Declaración de funciones.

4.1.- Tiempo de vida de los datos

Según el lugar donde son declaradas puede haber dos tipos de variables.

Globales: las variables permanecen activas durante todo el programa. Se crean al iniciarse éste y se destruyen de la memoria al finalizar. Pueden ser utilizadas en cualquier función.

Locales: las variables son creadas cuando el programa llega a la función en la que están definidas. Al finalizar la función desaparecen de la memoria.

Si dos variables, una global y una local, tienen el mismo nombre, la local prevalecerá

sobre la global dentro de la función en que ha sido declarada.

Dos variables locales pueden tener el mismo nombre siempre que estén declaradas en funciones diferentes.

4.2.- Funciones

Las funciones son bloques de código utilizados para dividir un programa en partes más pequeñas, cada una de las cuáles tendrá una tarea determinada.

Su sintaxis es:

```
tipo_función nombre_función (tipo y nombre de argumentos)
{
        bloque de sentencias
}
```

tipo_función: puede ser de cualquier tipo de los que conocemos. El valor devuelto por la función será de este tipo. Por defecto, es decir, si no indicamos el tipo, la función devolverá un valor de tipo entero (**int**). Si no queremos que retorne ningún valor deberemos indicar el tipo vacío (**void**).

nombre_función: es el nombre que le daremos a la función.

tipo y nombre de argumentos: son los parámetros que recibe la función. Los argumentos de una función no son más que variables locales que reciben un valor. Este valor se lo enviamos al hacer la llamada a la función. Pueden existir funciones que no reciban argumentos.

bloque de sentencias: es el conjunto de sentencias que serán ejecutadas cuando se realice la llamada a la función.

Las funciones pueden ser llamadas desde la función **main** o desde otras funciones. Nunca se debe llamar a la función **main** desde otro lugar del programa. Por último recalcar que los argumentos de la función y sus variables locales se destruirán al finalizar la ejecución de la misma.

4.3.- Declaración de las funciones

Al igual que las variables, las funciones también han de ser declaradas. Esto es lo que se conoce como prototipo de una función. Para que un programa en C sea compatible entre distintos compiladores es imprescindible escribir los prototipos de las funciones.

Los prototipos de las funciones pueden escribirse antes de la función **main** o bién en otro fichero. En este último caso se lo indicaremos al compilador mediante la directiva **#include**.

En el ejemplo adjunto podremos ver la declaración de una función (prototipo). Al no recibir ni retornar ningún valor, está declarada como **void** en ambos lados. También vemos que existe una variable global llamada *num*. Esta variable es reconocible en todas las funciones del programa. Ya en la función **main** encontramos una variable local llamada *num*. Al ser una variable local, ésta tendrá preferencia sobre la global. Por tanto la función escribirá los números 10 y 5.

```
/* Declaración de funciones. */

#include <stdio.h>

void funcion(void); /* prototipo */
int num=5; /* variable global */
main() /* Escribe dos números */
{
        int num=10; /* variable local */
        printf("%d\n",num);
        funcion(); /* llamada */
}

void funcion(void)
{
        printf("%d\n",num);
}
```

4.4.- Paso de parámetros a una función

Como ya hemos visto, las funciones pueden retornar un valor. Esto se hace mediante la instrucción **return**, que finaliza la ejecución de la función, devolviendo o no un valor.

En una misma función podemos tener más de una instrucción return. La forma de retornar un valor es la siguiente:

```
return ( valor o expresión );
```

El valor devuelto por la función debe asignarse a una variable. De lo contrario, el valor se perderá.

En el ejemplo puedes ver lo que ocurre si no guardamos el valor en una variable.

Fíjate que a la hora de mostrar el resultado de la suma, en el **printf**, también podemos llamar a la función.

```
#include <stdio.h>

int suma(int,int); /* prototipo */
main() /* Realiza una suma */
{
        int a=10,b=25,t;
        t=suma(a,b); /* guardamos el valor */
        printf("%d=%d",suma(a,b),t);
        suma(a,b); /* el valor se pierde */
}

int suma(int a,int b)
{
        return (a+b);
}
```

5. Declaración de variables.

5.1.- Las variables

Una variable es un tipo de dato, referenciado mediante un identificador (que es el nombre de la variable). Su contenido podrá ser modificado a lo largo del programa.

Una variable sólo puede pertenecer a un tipo de dato. Para poder utilizar una variable, primero tiene que ser declarada:

```
        [calificador] <tipo> <nombre>
```

Es posible inicializar y declarar más de una variable del mismo tipo en la misma sentencia:

```
[calificador] <tipo> <nombre1>,<nombre2>=<valor>

#include <stdio.h>

main() /* Suma dos valores */
{
        int num1=4,num2,num3=6;
        printf("El valor de num1 es %d",num1);
```

```
        printf("\nEl valor de num3 es %d",num3);
        num2=num1+num3;
        printf("\nnum1 + num3 = %d",num2);
}
```

5.2.- ¿ Dónde se declaran ?

Las variables pueden ser de dos tipos según el lugar en que las declaremos: *globales* o *locales*.

La variable global se declara antes de la **main( )**. Puede ser utilizada en cualquier parte del programa y se destruye al finalizar éste.

La variable local se declara después de la **main( )**, en la función en que vaya a ser utilizada. Sólo existe dentro de la función en que se declara y se destruye al finalizar dicha función.

El identificador (nombre de la variable) no puede ser una **palabra clave** y los caracteres que podemos utilizar son las letras: **a-z** y **A-Z** (ojo! la **ñ** o **Ñ** no está permitida), los números: **0-9** y el símbolo de subrayado **_**. Además hay que tener en cuenta que el primer caracter no puede ser un número.

```
#include <stdio.h>

int a;
main() /* Muestra dos valores */
{
        int b=4;
        printf("b es local y vale %d",b);
        a=5;
        printf("\na es global y vale %d",a);
}
```

5.3.- Constantes

Al contrario que las **variables**, las constantes mantienen su valor a lo largo de todo el programa.

Para indicar al compilador que se trata de una constante, usaremos la directiva **#define**:

```
        #define <identificador> <valor>
```

Observa que no se indica el punto y coma de final de sentencia ni tampoco el tipo de dato.

La directiva **#define** no sólo nos permite sustituir un nombre por un valor numérico, sinó también por una cadena de caracteres.

El valor de una constante no puede ser modificado de ninguna manera

```
#include <stdio.h>
#define pi 3.1416
#define escribe printf
main() /* Calcula el perímetro */
{
        int r;
        escribe("Introduce el radio: ");
        scanf("%d",&r);
        escribe("El perímetro es: %f",2*pi*r);
}
```

5.4.- Secuencias de escape

Ciertos caracteres no representados gráficamente se pueden representar mediante lo que se conoce como secuencia de escape.

A continuación vemos una tabla de las más significativas:

\n	salto de línea
\b	retroceso
\t	tabulación horizontal
\v	tabulación vertical
\\	contrabarra
\f	salto de página
\'	apóstrofe
\"	comillas dobles
\0	fin de una cadena de caracteres

5.5.- Inclusión de ficheros

En la programación en C es posible utilizar funciones que no esten incluídas en el propio programa. Para ello utilizamos la directiva **#include**, que nos permite añadir librerías o funciones que se encuentran en otros ficheros a nuestro programa.

Para indicar al compilador que vamos a incluir ficheros externos podemos hacerlo de dos maneras (siempre antes de las declaraciones).

1. Indicándole al compilador la ruta donde se encuentra el fichero.

   #include "misfunc.h"
   #include "c:\includes\misfunc.h"
2. Indicando que se encuentran en el directorio por defecto del compilador.

   #include <misfunc.h>

6. Sentencias de control.

Este tipo de sentencias permiten variar el flujo del programa en base a unas determinadas condiciones.

Existen varias estructuras diferentes:

6.1.- Estructura IF...ELSE

Sintaxis:

if (condición) sentencia;

La sentencia solo se ejecuta si se cumple la condición. En caso contrario el programa sigue su curso sin ejecutar la sentencia.

Otro formato:

if (condición) sentencia1;
else sentencia2;

Si se cumple la condición ejecutará la **sentencia1**, sinó ejecutará la **sentencia2**. En cualquier caso, el programa continuará a partir de la **sentencia2**.

6.2.- Estructura SWITCH

Esta estructura se suele utilizar en los menús, de manera que según la opción seleccionada se ejecuten una serie de sentencias.

Su sintaxis es:

```
switch (variable){
        case contenido_variable1:
                sentencias;
                break;
        case contenido_variable2:
                sentencias;
                break;
        default:
                sentencias;
}
```

Cada case puede incluir una o más sentencias sin necesidad de ir entre llaves, ya que se ejecutan todas hasta que se encuentra la sentencia **BREAK**. La variable evaluada sólo puede ser de tipo **entero** o **caracter**. **default** ejecutará las sentencias que incluya, en caso de que la opción escogida no exista.

```
/* Uso de la sentencia condicional SWITCH. */

#include <stdio.h>

main() /* Escribe el día de la semana */
{
  int dia;
  printf("Introduce el día: ");
  scanf("%d",&dia);
  switch(dia){
    case 1: printf("Lunes"); break;
    case 2: printf("Martes"); break;
    case 3: printf("Miércoles"); break;
    case 4: printf("Jueves"); break;
    case 5: printf("Viernes"); break;
    case 6: printf("Sábado"); break;
    case 7: printf("Domingo"); break;
  }
}
```

6.3. Operadores lógicos.

Los operadores lógicos básicos son tres:

&&	AND
||	OR
!	NOT (El valor contrario)

Estos operadores actúan sobre expresiones lógicas. Permiten unir expresiones lógicas simples formando otras más complejas.

V = Verdadero F = Falso

```
/* Uso de los op. lógicos AND,OR,NOT. */

#include <stdio.h>

main() /* Compara un número introducido */
{
  int numero;
  printf("Introduce un número: ");
  scanf("%d",&numero);
  if(!(numero>=0))
    printf("El número es negativo");
  else if((numero<=100)&&(numero>=25))
    printf("El número está entre 25 y 100");
  else if((numero<25)||(numero>100))
    printf("El número no está entre 25 y 100");
}
```

6.4.- Sentencia WHILE

Los bucles son estructuras que permiten ejecutar partes del código de forma repetida mientras se cumpla una condición.

Esta condición puede ser simple o compuesta de otras condiciones unidas por operadores lógicos.

Su sintaxis es:

while (condición) sentencia;

Con esta sentencia se controla la condición antes de entrar en el bucle. Si ésta no se cumple, el programa no entrará en el bucle.

Naturalmente, si en el interior del bucle hay más de una sentencia, éstas deberán ir entre

llaves para que se ejecuten como un bloque.

```
/* Uso de la sentencia WHILE. */

#include <stdio.h>

main() /* Escribe los números del 1 al 10 */
{
  int numero=1;
  while(numero<=10)
  {
    printf("%d\n",numero);
    numero++;
  }
}
```

6.5.- Sentencia DO...WHILE

Su sintaxis es:

```
do{
        sentencia1;
        sentencia2;
}while (condición);
```

Con esta sentencia se controla la condición al final del bucle. Si ésta se cumple, el programa vuelve a ejecutar las sentencias del bucle.

La única diferencia entre las sentencias while y do...while es que con la segunda el cuerpo del bucle se ejecutará por lo menos una vez.

```
/* Uso de la sentencia DO...WHILE. */

#include <stdio.h>

main() /* Muestra un menú si no se pulsa 4 */
{
  char seleccion;
  do{
    printf("1.- Comenzar\n");
    printf("2.- Abrir\n");
    printf("3.- Grabar\n");
    printf("4.- Salir\n");
    printf("Escoge una opción: ");
```

```
    seleccion=getchar();
    switch(seleccion){
            case '1':printf("Opción 1");
              break;
            case '2':printf("Opción 2");
              break;
            case '3':printf("Opción 3");
            }

  }while(seleccion!='4');
}
```

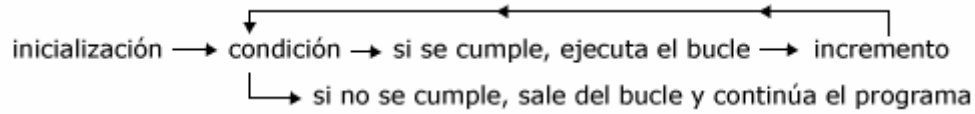
6.6.- Sentencia FOR

Su sintaxis es:

```
for (inicialización;condición;incremento){
        sentencia1;
        sentencia2;
}
```

La inicialización indica una variable (variable de control) que condiciona la repetición del bucle. Si hay más, van separadas por comas:

```
for (a=1,b=100;a!=b;a++,b- -){
```



El flujo del bucle **FOR** transcurre de la siguiente forma:

```
/* Uso de la sentencia FOR. */

#include <stdio.h>

main() /* Escribe la tabla de multiplicar */
{
  int num,x,result;
  printf("Introduce un número: ");
  scanf("%d",&num);
  for (x=0;x<=10;x++){
```

```
    result=num*x;
    printf("\n%d por %d = %d\n",num,x,result);
  }
}
```

6.7.- Sentencia BREAK

Esta sentencia se utiliza para terminar la ejecución de un bucle o salir de una sentencia **SWITCH**.

6.8.- Sentencia CONTINUE

Se utiliza dentro de un bucle. Cuando el programa llega a una sentencia **CONTINUE** no ejecuta las líneas de código que hay a continuación y salta a la siguiente iteración del bucle.

Y aquí termina el capítulo dedicado a los bucles. Existe otra sentencia, **GOTO**, que permite al programa saltar hacia un punto identificado con una etiqueta, pero el buen programador debe prescindir de su utilización. Es una sentencia muy mal vista en la programación en 'C'.

/* Uso de la sentencia CONTINUE. */

```
#include <stdio.h>

main() /* Escribe del 1 al 100 menos el 25 */
{
  int numero=1;
  while(numero<=100)
  {
    if (numero==25)
    {
            numero++;
            continue;
    }
  printf("%d\n",numero);
  numero++;
  }
}
```

7. Directiva #define.

La directiva #define se utiliza comúnmente para asociar identificadores con constantes, teclas, y usa generalmente sentencias o expresiones.

Llamaremos constantes manifiestas a los identificadores que representan constantes, y macros a los identificadores que representan expresiones o sentencias.

Una vez definido un identificador, éste no puede ser redefinido con diferentes valores sin (anteriormente) destruir la definición original.Sin embargo, la directiva #undef NombreMacro anula la decisión original y permite redefinir el identificador.

La definición deuna constante manifiesta se realiza:

#define IDENTIFICADOR valor

Ejemplo:

```
#define MÁXIMO 100
#define MÍNIMO 10
#define PI 3.141592

main()
{
char cadena[MÁXIMO];
char cad[MINIMO];
.............
.............
}
```

En el ejemplo anterio, se declaran tres constantes manifiestas: MÁXIMO, MINIMO y PI. La aparición posterior en el programa de estas constantes hará que el procesador las sustituya por los valores situados a su derecha.

En la práctica, existen dos tipos de macros: macros sin parámetros y macros con parámetros. Los dos formatos que se describen a continuación se correspnden con estos dos tipo:

#define identificador Sentencias/Funciones C
#define identificador(lista de parámetros) Sentencias/FuncionesC

La directiva #define sustituye identificador por Sentencias/FuncionesC en todas las posteriores apariciones de Identificador en su programa fuente.

Identificador es reemplazado solamente cuando forma un conjunto de c aracteres reconocibles por el compilador, y debe ser una sola palabra sin espacios en blanco intercalados.

Identificador no es reemplazado cuando aparece dentro de una cadena de caracteres o forma parte de un identificador de mayor longitud.

Si detrás del identificador aparece una lista de parámetros, la directiva #define reemplaza, cada vez que lo encuentre,identificados(lista de parámetros) por Sentencias/FuncionesC, sustituyendo los argumentos actuales por parámetros formales.

Sentencias/FuncionesC consiste en una serie de agrupación de caracteres reconocibles por el compilador,caracteres del teclado, constantes o sentencias7funciones. Entre identificador y Sentencias/FuncionesC debe haber uno o más espacios en blanco que no serán tenidos en cuenta en la sustitución.

Si el texto de Sentencias/FuncionesC no cabe en una sola línea, puede ser continuado en la siguiente, colocando al final de la línea, antes de pulsar la tecla [ENTER], el carácter de barra izquierda (\).

ListaParámetros es opcional. Consiste en uno o más parámetros formales separados por comas. Cada nombre de la lista debe ser único, y la l ista debe de estar entre paréntesis, no pudiendo existir espacios entre Identificador y el paréntesis de apertura.

Ejemplo:

```
#define cuadrado(x) x*x
#define cubo(x) cuadrado(x)*x
#define ABS(X) ( (x>=0) ? x:0-x)
#define producto(x,y) x*y
#define IMPRIME(X) printf("\n"#x)
```

8. Ficheros.

Ahora veremos la forma de almacenar datos que podremos recuperar cuando deseemos. Estudiaremos los distintos modos en que podemos abrir un fichero, así como las funciones para leer y escribir en él.

8.1.- Apertura

Antes de abrir un fichero necesitamos declarar un puntero de tipo **FILE**, con el que trabajaremos durante todo el proceso. Para abrir el fichero utilizaremos la función

fopen( ).

Su sintaxis es:

```
FILE *puntero;
puntero = fopen ( nombre del fichero, "modo de apertura" );
```

donde **puntero** es la variable de tipo **FILE**, **nombre del fichero** es el nombre que daremos al fichero que queremos crear o abrir. Este nombre debe ir encerrado entre comillas. También podemos especificar la ruta donde se encuentra o utilizar un array que contenga el nombre del archivo (en este caso no se pondrán las comillas). Algunos ejemplos:

```
puntero=fopen("DATOS.DAT","r");
puntero=fopen("C:\\TXT\\SALUDO.TXT","w");
```

Un archivo puede ser abierto en dos modos diferentes, en modo texto o en modo binario. A continuación lo veremos con más detalle.

Modo texto

w	crea un fichero de escritura. Si ya existe lo crea de nuevo.
w+	crea un fichero de lectura y escritura. Si ya existe lo crea de nuevo.
a	abre o crea un fichero para añadir datos al final del mismo.
a+	abre o crea un fichero para leer y añadir datos al final del mismo.
r	abre un fichero de lectura.
r+	abre un fichero de lectura y escritura.

Modo binario

wb	crea un fichero de escritura. Si ya existe lo crea de nuevo.
w+b	crea un fichero de lectura y escritura. Si ya existe lo crea de nuevo.
ab	abre o crea un fichero para añadir datos al final del mismo.
a+b	abre o crea un fichero para leer y añadir datos al final del mismo.
rb	abre un fichero de lectura.
r+b	abre un fichero de lectura y escritura.

La función **fopen** devuelve, como ya hemos visto, un puntero de tipo **FILE**. Si al intentar abrir el fichero se produjese un error (por ejemplo si no existe y lo estamos abriendo en modo lectura), la función **fopen** devolvería **NULL**. Por esta razón es mejor controlar las posibles causas de error a la hora de programar. Un ejemplo:

```
FILE *pf;
pf=fopen("datos.txt","r");
if (pf == NULL) printf("Error al abrir el fichero");
```

freopen( )

Esta función cierra el fichero apuntado por el puntero y reasigna este puntero a un fichero que será abierto. Su sintaxis es:

```
freopen(nombre del fichero,"modo de apertura",puntero);
```

donde **nombre del fichero** es el nombre del nuevo fichero que queremos abrir, luego el **modo de apertura**, y finalmente el puntero que va a ser reasignado.

8.2.- Cierre

Una vez que hemos acabado nuestro trabajo con un fichero es recomendable cerrarlo. Los ficheros se cierran al finalizar el programa pero el número de estos que pueden estar abiertos es limitado. Para cerrar los ficheros utilizaremos la función **fclose( );**.

Esta función cierra el fichero, cuyo puntero le indicamos como parámetro. Si el fichero se cierra con éxito devuelve **0**.

```
fclose(puntero);
```

Un ejemplo ilustrativo aunque de poca utilidad:

```
FILE *pf;
pf=fopen("AGENDA.DAT","rb");
if ( pf == NULL ) printf ("Error al abrir el fichero");
else fclose(pf);
```

8.3.- Escritura y lectura

A continuación veremos las funciones que se podrán utilizar dependiendo del dato que queramos escribir y/o leer en el fichero.

Un caracter

```
fputc( variable_caracter , puntero_fichero );
```

Escribimos un caracter en un fichero (abierto en modo escritura). Un ejemplo:

```
        FILE *pf;
        char letra='a';
        if (!(pf=fopen("datos.txt","w"))) /* otra forma de controlar si se produce
un error */
        {
                printf("Error al abrir el fichero");
                exit(0); /* abandonamos el programa */
        }
        else fputc(letra,pf);
        fclose(pf);^b

        fgetc( puntero_fichero );
```

Lee un caracter de un fichero (abierto en modo lectura). Deberemos guardarlo en una variable. Un ejemplo:

```
        FILE *pf;
        char letra;
        if (!(pf=fopen("datos.txt","r"))) /* controlamos si se produce un error */
        {
                printf("Error al abrir el fichero");
                exit(0); /* abandonamos el programa */
        }
        else
        {
                letra=fgetc(pf);
                printf("%c",letra);
                fclose(pf);
        }
```

Un número entero

```
        putw( variable_entera, puntero_fichero );
```

Escribe un número entero en formato binario en el fichero. Ejemplo:

```
        FILE *pf;
        int num=3;
        if (!(pf=fopen("datos.txt","wb"))) /* controlamos si se produce un error
*/
```

```
        {
                printf("Error al abrir el fichero");
                exit(0); /* abandonamos el programa */
        }
        else
        {
                fputw(num,pf); /* también podíamos haber hecho directamente:
fputw(3,pf); */
                fclose(pf);
        }

        getw( puntero_fichero );
```

Lee un número entero de un fichero, avanzando dos bytes después de cada lectura. Un ejemplo:

```
        FILE *pf;
        int num;
        if (!(pf=fopen("datos.txt","rb"))) /* controlamos si se produce un error */
        {
                printf("Error al abrir el fichero");
                exit(0); /* abandonamos el programa */
        }
        else
        {
                num=getw(pf);
                printf("%d",num);
                fclose(pf);
        }
```

Una cadena de caracteres

```
        fputs( variable_array, puntero_fichero );
```

Escribe una cadena de caracteres en el fichero. Ejemplo:

```
        FILE *pf;
        char cad="Me llamo Vicente";
        if (!(pf=fopen("datos.txt","w"))) /* controlamos si se produce un error */
        {
                printf("Error al abrir el fichero");
                exit(0); /* abandonamos el programa */
        }
        else
```

```
{
        fputs(cad,pf); /* o también así: fputs("Me llamo Vicente",pf); */
        fclose(pf);
}
```

fgets(variable_array, variable_entera, puntero_fichero);

Lee una cadena de caracteres del fichero y la almacena en variable_array. La variable_entera indica la longitud máxima de caracteres que puede leer. Un ejemplo:

```
FILE *pf;
char cad[80];
if (!(pf=fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else
{
        fgets(cad,80,pf);
        printf("%s",cad);
        fclose(pf);
}
```

Con formato

fprintf(puntero_fichero, formato, argumentos);

Funciona igual que un **printf** pero guarda la salida en un fichero. Ejemplo:

```
FILE *pf;
char nombre[20]="Santiago";
int edad=34;
if (!(pf=fopen("datos.txt","w"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else
{
        fprintf(pf,"%20s%2d\n",nombre,edad);
        fclose(pf);
}
```

```
fscanf( puntero_fichero, formato, argumentos );
```

Lee los argumentos del fichero. Al igual que con un **scanf**, deberemos indicar la dirección de memoria de los argumentos con el símbolo **&** (ampersand). Un ejemplo:

```
FILE *pf;
char nombre[20];
int edad;
if (!(pf=fopen("datos.txt","rb"))) /* controlamos si se produce un error */
{
        printf("Error al abrir el fichero");
        exit(0); /* abandonamos el programa */
}
else
{
        fscanf(pf,"%20s%2d\",nombre,&edad);
        printf("Nombre: %s  Edad: %d",nombre,edad);
        fclose(pf);
}
```

Estructuras

fwrite(*buffer, tamaño, nº de veces, puntero_fichero);

Se utiliza para escribir bloques de texto o de datos, **estructuras**, en un fichero. En esta función, ***buffer** será la dirección de memoria de la cuál se recogerán los datos; **tamaño**, el tamaño en bytes que ocupan esos datos y **nº de veces**, será el número de elementos del tamaño indicado que se escribirán.

fread(*buffer, tamaño, nº de veces, puntero_fichero);

Se utiliza para leer bloques de texto o de datos de un fichero. En esta función, ***buffer** es la dirección de memoria en la que se almacenan los datos; **tamaño**, el tamaño en bytes que ocupan esos datos y **nº de veces**, será el número de elementos del tamaño indicado que se leerán.

Otras funciones para ficheros

rewind(puntero_fichero);

Sitúa el puntero al principio del archivo.

fseek(puntero_fichero, long posicion, int origen);

Sitúa el puntero en la **posicion** que le indiquemos. Como **origen** podremos poner:

0 o **SEEK_SET**, el principio del fichero
1 o **SEEK_CUR**, la posición actual
2 o **SEEK_END**, el final del fichero

rename(nombre1, nombre2);

Su función es exactamente la misma que la que conocemos en **MS-DOS**. Cambia el nombre del fichero **nombre1** por un nuevo nombre, **nombre2**.

remove(nombre);

Como la función del DOS **del**, podremos eliminar el archivo indicado en **nombre**.

Detección de final de fichero

feof(puntero_fichero);

Siempre deberemos controlar si hemos llegado al final de fichero cuando estemos leyendo, de lo contrario podrían producirse errores de lectura no deseados. Para este fin disponemos de la función **feof( )**. Esta función retorna **0** si no ha llegado al final, y un valor diferente de **0** si lo ha alcanzado.

Pues con esto llegamos al final del tema. Espero que no haya sido muy pesado. No es necesario que te aprendas todas las funciones de memoria. Céntrate sobre todo en las funciones **fputs( )**, **fgets( )**, **fprintf( )**, **fwrite( )** y **fread( )**. Con estas cinco se pueden gestionar los ficheros perfectamente.

APÉNDICE

Concepto de estructura

Una estructura es un conjunto de una o más variables, de distinto tipo, agrupadas bajo un mismo nombre para que su manejo sea más sencillo.

Su utilización más habitual es para la programación de bases de datos, ya que están especialmente indicadas para el trabajo con registros o fichas.

La sintaxis de su declaración es la siguiente:

```
struct tipo_estructura
{
        tipo_variable nombre_variable1;
        tipo_variable nombre_variable2;
        tipo_variable nombre_variable3;
};
```

Donde **tipo_estructura** es el nombre del nuevo tipo de dato que hemos creado. Por último, **tipo_variable** y **nombre_variable** son las variables que forman parte de la estructura.

Para definir variables del tipo que acabamos de crear lo podemos hacer de varias maneras, aunque las dos más utilizadas son éstas:

Una forma de definir la estructura:

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

Otra forma:

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

En el primer caso declaramos la estructura, y en el momento en que necesitamos las variables, las declaramos. En el segundo las declaramos al mismo tiempo que la estructura. El problema del segundo método es que no podremos declarar más variables de este tipo a lo largo del programa. Para poder declarar una variable de tipo estructura, la estructura tiene que estar declarada previamente. Se debe declarar antes de la función **main**.

El manejo de las estructuras es muy sencillo, así como el acceso a los campos (o variables) de estas estructuras. La forma de acceder a estos campos es la siguiente:

variable.campo;

Donde **variable** es el nombre de la variable de tipo *estructura* que hemos creado, y **campo** es el nombre de la variable que forma parte de la estructura. Lo veremos mejor con un ejemplo basado en la estructura del capítulo 13.1:

temporal.edad=25;

Lo que estamos haciendo es almacenar el valor 25 en el campo **edad** de la variable **temporal** de tipo **trabajador**.

Otra característica interesante de las estructuras es que permiten pasar el contenido de una estructura a otra, siempre que sean del mismo tipo naturalmente:

fijo=temporal;

Al igual que con los otros tipos de datos, también es posible inicializar variables de tipo **estructura** en el momento de su declaración:

struct trabajador fijo={"Pedro","Hernández Suárez", 32, "gerente"};

Si uno de los campos de la estructura es un **array** de números, los valores de la inicialización deberán ir entre llaves:

```
struct notas
{
        char nombre[30];
        int notas[5];
};

struct notas alumno={"Carlos Pérez",{8,7,9,6,10}};
```

Estructuras y funciones

Podemos enviar una estructura a una función de las dos maneras conocidas:

1.- Por valor: su declaración sería:

void visualizar(struct trabajador);

Después declararíamos la variable **fijo** y su llamada sería:

visualizar(fijo);

Por último, el desarrollo de la función sería:

void visualizar(struct trabajador datos)

```
/* Paso de una estructura por valor. */

#include <stdio.h>

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

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);
}
```

Arrays de estructuras

Es posible agrupar un conjunto de elementos de tipo estructura en un array. Esto se conoce como *array de estructuras*:

```
struct trabajador
{
        char nombre[20];
        char apellidos[40];
        int edad;
};

struct trabajador fijo[20];
```

Así podremos almacenar los datos de 20 trabajadores. Ejemplos sobre como acceder a los campos y sus elementos: para ver el nombre del cuarto trabajador, **fijo[3].nombre;**. Para ver la tercera letra del nombre del cuarto trabajador, **fijo[3].nombre[2];**. Para inicializar la variable en el momento de declararla lo haremos de esta manera:

```
struct trabajador fijo[20]={{"José","Herrero Martínez",29},{"Luis","García Sánchez",46}};
```

Typedef

Es posible agrupar un conjunto de elementos de tipo estructura en un array. Esto se conoce como *array de estructuras*: El lenguaje 'C' dispone de una declaración llamada **typedef** que permite la creación de nuevos tipos de datos. Ejemplos:

```
typedef int entero; /* acabamos de crear un tipo de dato llamado entero */
entero a, b=3;              /* declaramos dos variables de este tipo */
```

Su empleo con estructuras está especialmente indicado. Se puede hacer de varias formas:

Una forma de hacerlo:

```
struct trabajador
{
        char nombre[20];
        char apellidos[40];
        int edad;
```

```
};

typedef struct trabajador datos;
datos fijo,temporal;
```

Otra forma:

```
typedef struct
{
        char nombre[20];
        char apellidos[40];
        int edad;
}datos;

datos fijo,temporal;
```