INDICE

1.− CARACTERISTICAS DEL LENGUAJE C

El lenguaje C se conoce como un lenguaje compilado. Existen dos tipos de lenguaje: interpretados y compilados. Los interpretados son aquellos que necesitan del código fuente para funcionar (P.ej: Basic). Los compilados convierten el código fuente en un fichero objeto y éste en un fichero ejecutable. Este es el caso del lenguaje C.

Podemos decir que el lenguaje C es un lenguaje de nivel medio, ya que combina elementos de lenguaje de alto nivel con la funcionalidad del lenguaje ensamblador. Es un lenguaje estructurado, ya que permite crear procedimientos en bloques dentro de otros procedimientos. Hay que destacar que el C es un lenguaje portable, ya que permite utilizar el mismo código en diferentes equipos y sistemas informáticos: el lenguaje es independiente de la arquitectura de cualquier máquina en particular.

Por último solo queda decir que el C es un lenguaje relativamente pequeño; se puede describir en poco espacio y aprender rápidamente. Este es sin duda el objetivo de éste curso. No pretende ser un completo manual de la programación, sinó una base útil para que cualquiera pueda introducirse en este apasionante mundo.

Aunque en principio cualquier compilador de C es válido, para seguir este curso se recomienda utilizar el compilador **Turbo C/C++** o bien el **Borland C++ 5.0.**

2.− ESTRUCTURA DE UN PROGRAMA EN C

2.1.− Estructura

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

}
```

2.2.− 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.

2.3.− 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 if
```

else do while for switch

short long extern static default

continue break register sizeof typedef

2.4.− 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

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')

3.3.− 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>,<nombre3>=<valor>,<nombre4>

Ejemplo:

```
/* Uso de las variables */

#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);

}
```

3.4.− ¿ 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.

Ejemplo:

```
/* Declaración de variables */

#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);
```

}

3.5.− 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.

Ejemplo:

```
/* Uso de las constantes */

#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);

}
```

3.6.− 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

Ejemplo:

```
/* Uso de las secuencias de escape */

#include <stdio.h>

main() /* Escribe diversas sec. de escape */

{

printf("Me llamo \"Nemo\" el grande");

printf("\nDirección: C\\ Mayor 25");

printf("\nHa salido la letra \'L\'");

printf("\nRetroceso\b");

printf("\n\tEsto ha sido todo");

}
```

3.7.− 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>
```

4.− OPERADORES ARITMETICOS Y DE ASIGNACION

A continuación se explican los tipos de operadores (aritméticos y de asignación) que permiten realizar

operaciones matemáticas en lenguaje C.

4.1.− Operadores aritméticos

Existen dos tipos de operadores aritméticos:
Los binarios:

+ Suma

− Resta

***** Multiplicación

/ División

% Módulo (resto)

y los unarios:

++ Incremento (suma 1)

− − Decremento (resta 1)

− Cambio de signo

Su sintaxis es:

binarios:

<variable1><operador><variable2>

unarios:

<variable><operador> y al revés, **<operador><variable>.**

Ejemplo:

```
/* Uso de los operadores aritméticos */

#include <stdio.h>

main() /* Realiza varias operaciones */

{

int a=1,b=2,c=3,r;

r=a+b;

printf("%d + %d = %d\n",a,b,r);

r=c−a;
```

```
printf("%d − %d = %d\n",c,a,r);

b++;

printf("b + 1 = %d",b);

}
```

4.2.− Operadores de asignación

La mayoría de los operadores aritméticos binarios explicados en el capítulo anterior tienen su correspondiente operador de asignación:

= Asignación simple

+= Suma

−= Resta

***=** Multiplicación

/= División

%= Módulo (resto)

Con estos operadores se pueden escribir, de forma más breve, expresiones del tipo:

n=n+3 se puede escribir **n+=3**
k=k*(x−2) lo podemos sustituir por **k*=x−2**

ejemplo:

```
/* Uso de los operadores de asignación */

#include <stdio.h>

main() /* Realiza varias operaciones */

{

int a=1,b=2,c=3,r;

a+=5;

printf("a + 5 = %d\n",a);

c−=1;

printf("c − 1 = %d\n",c);

b*=3;
```

```
printf("b * 3 = %d",b);

}
```

4.3.− Jerarquía de los operadores

Será importante tener en cuenta la precedencia de los operadores a la hora de trabajar con ellos:

( ) Mayor precedencia

++, − −

***, /, %**

+, − Menor precendencia

Las operaciones con mayor precedencia se realizan antes que las de menor precedencia.

Si en una operación encontramos signos del mismo nivel de precedencia, dicha operación se realiza de izquierda a derecha. A continuación se muestra un ejemplo sobre ello:

a*b+c/d−e

1. a*b resultado = x

2. c/d resultado = y

3. x+y resultado = z

4. z−e

Fijarse que la multiplicación se resuelve antes que la división ya que está situada más a la izquierda en la operación. Lo mismo ocurre con la suma y la resta.

Ejemplo:

```
/* Jerarquía de los operadores */

#include <stdio.h>

main() /* Realiza una operación */

{

int a=6,b=5,c=4,d=2,e=1,x,y,z,r;

x=a*b;

printf("%d * %d = %d\n",a,b,x);

y=c/d;

printf("%d / %d = %d\n",c,d,y);
```

```
z=x+y;

printf("%d + %d = %d\n",x,y,z);

r=z−e;

printf("%d = %d",r,a*b+c/d−e);

}
```

5.− SALIDA / ENTRAD

5.1.− Sentencia printf( )

La rutina printf permite la aparición de valores numéricos, caracteres y cadenas de texto por pantalla.
El prototipo de la sentencia *printf* es el siguiente:

printf(control,arg1,arg2...);

En la cadena de control indicamos la forma en que se mostrarán los argumentos posteriores. También podemos introducir una cadena de texto (sin necesidad de argumentos), o combinar ambas posibilidades, así como **secuencias de escape.**

En el caso de que utilicemos argumentos deberemos indicar en la cadena de control tantos modificadores como argumentos vayamos a presentar.

El modificador está compuesto por el caracter **%** seguido por un caracter de conversión, que indica de que tipo de dato se trata.

Ejemplo:

```
/* Uso de la sentencia printf() 1. */

#include <stdio.h>

main() /* Saca por pantalla una suma */

{

int a=20,b=10;

printf("El valor de a es %d\n",a);

printf("El valor de b es %d\n",b);

printf("Por tanto %d+%d=%d",a,b,a+b);

}
```

Los **modificadores** más utilizados son:

%c Un único caracter

%d Un entero con signo, en base decimal

%u Un entero sin signo, en base decimal

%o Un entero en base octal

%x Un entero en base hexadecimal

%e Un número real en coma flotante, con exponente

%f Un número real en coma flotante, sin exponente

%s Una cadena de caracteres

%p Un puntero o dirección de memoria

Ejemplo:

```
/* Uso de la sentencia printf() 2. */

#include <stdio.h>

main() /* Modificadores 1 */

{

char cad[]="El valor de";

int a=-15;

unsigned int b=3;

float c=932.5;

printf("%s a es %d\n",cad,a);

printf("%s b es %u\n",cad,b);

printf("%s c es %e o %f",cad,c,c);

}
```

El formato completo de los modificadores es el siguiente:

% [signo] [longitud] [.precisión] [l/L] conversión

Signo: indicamos si el valor se ajustará a la izquierda, en cuyo caso utilizaremos el signo menos, o a la derecha (por defecto).
Longitud: especifica la longitud máxima del valor que aparece por pantalla. Si la longitud es menor que el número de dígitos del valor, éste aparecerá ajustado a la izquierda.

Precisión: indicamos el número máximo de decimales que tendrá el valor.
l/L: utilizamos l cuando se trata de una variable de tipo long y L cuando es de tipo double.

Ejemplo:

```
/* Uso de la sentencia printf() 3. */

#include <stdio.h>

main() /* Modificadores 2 */

{

char cad[ ]="El valor de";

int a=25986;

long int b=1976524;

float c=9.57645;

printf("%s a es %9d\n",cad,a);

printf("%s b es %ld\n",cad,b);

printf("%s c es %.3f",cad,c);

}
```

5.2.− Sentencia scanf( )

La rutina scanf permite entrar datos en la memoria del ordenador a través del teclado.
El prototipo de la sentencia *scanf* es el siguiente:

```
scanf(control,arg1,arg2...);
```

En la cadena de control indicaremos, por regla general, los modificadores que harán referencia al tipo de dato de los argumentos. Al igual que en la sentencia *printf* los **modificadores** estarán formados por el caracter **%** seguido de un caracter de conversión. Los argumentos indicados serán, nuevamente, las variables.

La principal característica de la sentencia *scanf* es que necesita saber la posición de la memoria del ordenador en que se encuentra la variable para poder almacenar la información obtenida. Para indicarle esta posición utilizaremos el símbolo *ampersand* (**&**), que colocaremos delante del nombre de cada variable. (Esto no será necesario en los arrays).

Ejemplo:

```
/* Uso de la sentencia scanf(). */

#include <stdio.h>

main() /* Solicita dos datos */

{

char nombre[10];
```

```
int edad;

printf("Introduce tu nombre: ");

scanf("%s",nombre);

printf("Introduce tu edad: ");

scanf("%d",&edad);

}
```

6.− OPERADORES RELACIONALES

Los operadores relacionales se utilizan para comparar el contenido de dos variables.
En C existen seis operadores relacionales básicos:

> Mayor que

< Menor que

>= Mayor o igual que

<= Menor o igual que

== Igual que

!= Distinto que

El resultado que devuelven estos operadores es **1** para Verdadero y **0** para Falso.
Si hay más de un operador se evalúan de izquierda a derecha. Además los operadores == y **!=** están por debajo del resto en cuanto al orden de precedencia.

Ejemplo:

```
/* Uso de los operadores relacionales. */

#include <stdio.h>

main() /* Compara dos números entre ellos */

{

int a,b;

printf("Introduce el valor de A: ");

scanf("%d",&a);

printf("Introduce el valor de B: ");

scanf("%d",&b);
```

```
if(a>b)

printf("A es mayor que B");

else if(a<b)

printf("B es mayor que A");

else

printf("A y B son iguales");

}
```

7.− SENTENCIAS CONDICIONALES

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

Existen varias estructuras diferentes:

7.1.− 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**.

Ejemplo:

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

#include <stdio.h>

main() /* Simula una clave de acceso */

{

int usuario,clave=18276;

printf("Introduce tu clave: ");

scanf("%d",&usuario);
```

```
if(usuario==clave)

printf("Acceso permitido");

else

printf("Acceso denegado");

}
```

Otro formato:

```
if (condición) sentencia1;

else if (condición) sentencia2;

else if (condición) sentencia3;

else sentencia4;
```

Con este formato el flujo del programa únicamente entra en una de las condiciones. Si una de ellas se cumple, se ejecuta la sentencia correspondiente y salta hasta el final de la estructura para continuar con el programa.

Existe la posibilidad de utilizar llaves para ejecutar más de una sentencia dentro de la misma condición.

Ejemplo:

```
/* Uso de la sentencia condicional ELSE...IF. */

#include <stdio.h>

main() /* Escribe bebé, niño o adulto */

{

int edad;

printf("Introduce tu edad: ");

scanf("%d",&edad);

if (edad<1)

printf("Lo siento, te has equivocado.");

else if (edad<3) printf("Eres un bebé");

else if (edad<13) printf("Eres un niño");

else printf("Eres adulto");

}
```

7.2.− 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.

Ejemplo:

```
/* 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;

}

}

8.− OPERADORES LOGICOS

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.

OPERANDOS		AND	OR
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

V = Verdadero F = Falso

ejemplo:

/* 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");

}
```

9.− BUCLES

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.

9.1.− Sentencia WHILE

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.

Ejemplo:

```
/* 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++;

}
```

}

9.2.− 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.

Ejemplo:

```
/* 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');

}
```

9.3.− Sentencia FOR

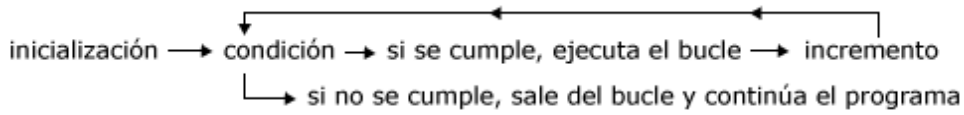
Su sintaxis es:

```
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:



Ejemplo:

```
/* 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);

}

}
```

9.4.− Sentencia BREAK

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

9.5.− 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'.

Ejemplo:

```
/* 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++;
```

```
}

}
```

9.4.− Sentencia BREAK

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

9.5.− 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'.

ejemplo:

```
/* 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++;

}

}
```

10.− FUNCIONES

10.1.− 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.

Ejemplo:

```
/* Variables globales y locales. */

#include <stdio.h>

int num1=1;

main() /* Escribe dos cifras */

{

int num2=10;

printf("%d\n",num1);

printf("%d\n",num2);

}
```

10.2.− 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.

10.3.− 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.

Ejemplo:

```
/* 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);

}
```

10.4.− 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.

Ejemplo:

```
/* Paso de parámetros. */

#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);

}
```

Ahora veremos lo que se conoce como paso de parámetros.

Existen dos formas de enviar parámetros a una función:

Por valor: cualquier cambio que se realice dentro de la función en el argumento enviado, **NO** afectará al valor original de las variables utilizadas en la llamada. Es como si trabajaramos con una *copia*, no con el *original*. No es posible enviar por valor **arrays**, deberemos hacerlo por referencia.

Por referencia: lo que hacemos es enviar a la función la dirección de memoria donde se encuentra la variable o dato. Cualquier modificación **SI** afectará a las variables utilizadas en la llamada. Trabajamos directamente con el *original*

Ejemplo:

```
/* Paso por valor. */
```

```
#include <stdio.h>

void intercambio(int,int);

main() /* Intercambio de valores */

{

int a=1,b=2;

printf("a=%d y b=%d",a,b);

intercambio(a,b); /* llamada */

printf("a=%d y b=%d",a,b);

}

void intercambio (int x,int y)

{

int aux;

aux=x;

x=y;

y=aux;

printf("a=%d y b=%d",x,y);

}
```

Para enviar un valor por referencia se utiliza el símbolo **&** (ampersand) delante de la variable enviada. Esto le indica al compilador que la función que se ejecutará tendra que obtener la dirección de memoria en que se encuentra la variable.

Vamos a fijarnos en los ejemplos. En el ejemplo anterior podrás comprobar que antes y después de la llamada, las variables mantienen su valor. Solamente se modifica en la función **intercambio** (*paso por valor*).

En el siguiente ejemplo podrás ver como las variables intercambian su valor tras la llamada de la función (*paso por referencia*).

Las variables con un ***** son conocidas como **punteros**, el único dato en 'C' que puede almacenar una dirección de memoria.

Ejemplo:

```
/* Paso por referencia. */

#include <stdio.h>

void intercambio(int *,int *);
```

```
main() /* Intercambio de valores */

{

int a=1,b=2;

printf("a=%d y b=%d",a,b);

intercambio(&a,&b); /* llamada */

printf("a=%d y b=%d",a,b);

}

void intercambio (int *x,int *y)

{

int aux;

aux=*x;

*x=*y;

*y=aux;

printf("a=%d y b=%d",*x,*y);

}
```

.− Los argumentos de la función main

Ya hemos visto que las funciones pueden recibir argumentos. Pues bién, la función **main** no podía ser menos y también puede recibir argumentos, en este caso desde el exterior.

Los argumentos que puede recibir son:

argc: es un contador. Su valor es igual al número de argumentos escritos en la línea de comandos, contando el nombre del programa que es el primer argumento.

argv: es un puntero a un array de cadenas de caracteres que contiene los argumentos, uno por cadena.

En este ejemplo vamos a ver un pequeño programa que escribirá un saludo por pantalla. El programa **FUNCION6.EXE**.

ejemplo:

```
/* Argumentos de la main. */

#include <stdio.h>

main(int argc,char *argv[]) /* argumentos */

{

printf("\nCurso de Programación en C − Copyright (c) 1997−2001, Sergio Pacho\n");
```

```
printf("Programa de ejemplo.\n\n");

if (argc<2)

{

printf("Teclee: funcion6 su_nombre");

exit(1); /* fin */

}

printf("Hola %s",argv[1]);

}
```

11.− ARRAYS

Un array es un identificador que referencia un conjunto de datos del mismo tipo. Imagina un tipo de dato **int**; podremos crear un conjunto de datos de ese tipo y utilizar uno u otro con sólo cambiar el índice que lo referencia. El índice será un valor entero y positivo. En **C** los arrays comienzan por la posición **0**.

11.1.− Vectores

Un vector es un array *unidimensional*, es decir, sólo utiliza un índice para referenciar a cada uno de los elementos. Su declaración será:

tipo nombre [tamaño];

El tipo puede ser cualquiera de los ya conocidos y el tamaño indica el número de elementos del vector (se debe indicar entre corchetes **[ ]**). En el ejemplo puedes observar que la variable **i** es utilizada como índice, el primer **for** sirve para rellenar el vector y el segundo para visualizarlo. Como ves, las posiciones van de **0** a **9** (total 10 elementos).

Ejemplo:

```
/* Declaración de un array. */

#include <stdio.h>

main() /* Rellenamos del 0 − 9 */

{

int vector[10],i;

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

for (i=0;i<10;i++) printf(" %d",vector[i]);

}
```

Podemos *inicializar* (asignarle valores) un vector en el momento de declararlo. Si lo hacemos así no es necesario indicar el tamaño. Su sintaxis es:

tipo nombre []={ valor 1, valor 2...}

Ejemplos:

```
int vector[]={1,2,3,4,5,6,7,8};

char vector[]="programador";

char vector[]={'p','r','o','g','r','a','m','a','d','o','r'};
```

Una particularidad con los vectores de tipo **char** (cadena de caracteres), es que deberemos indicar en que elemento se encuentra el fin de la cadena mediante el caracter nulo (**\0**). Esto no lo controla el compilador, y tendremos que ser nosotros los que insertemos este caracter al final de la cadena.

Por tanto, en un vector de 10 elementos de tipo **char** podremos rellenar un máximo de 9, es decir, hasta **vector[8]**. Si sólo rellenamos los 5 primeros, hasta **vector[4]**, debemos asignar el caracter nulo a **vector[5]**. Es muy sencillo: **vector[5]='\0';** .

Ahora veremos un ejemplo de como se rellena un vector de tipo **char**.

Ejemplo:

```
/* Vector de tipo char. */

#include <stdio.h>

main() /* Rellenamos un vector char */

{

char cadena[20];

int i;

for (i=0;i<19 && cadena[i−1]!=13;i++)

cadena[i]=getche( );

if (i==19) cadena[i]='\0';

else cadena[i−1]='\0';

printf("\n%s",cadena);

}
```

Podemos ver que en el **for** se encuentran dos condiciones:

1.− Que no se hayan rellenado todos los elementos (**i<19**).

2.− Que el usuario no haya pulsado la tecla ENTER, cuyo código ASCII es **13**. (**cadena[x−i]!=13**).

También podemos observar una nueva función llamada **getche( )**, que se encuentra en **conio.h**. Esta función permite la entrada de un caracter por teclado. Después se encuentra un **if**, que comprueba si se ha

rellenado todo el vector. Si es cierto, coloca el caracter nulo en el elemento nº20 (**cadena[19]**). En caso contrario tenemos el **else**, que asigna el caracter nulo al elemento que almacenó el caracter ENTER.

En resumen: al declarar una cadena deberemos reservar una posición más que la longitud que queremos que tenga dicha cadena.

.− Llamadas a funciones con arrays

Como ya se comentó en el tema anterior, los arrays únicamente pueden ser enviados a una función por referencia. Para ello deberemos enviar la dirección de memoria del primer elemento del array. Por tanto, el argumento de la función deberá ser un puntero.

Ejemplo:

```
/* Envío de un array a una función. */

#include <stdio.h>

void visualizar(int []); /* prototipo */

main() /* rellenamos y visualizamos */

{

int array[25],i;

for (i=0;i<25;i++)

{

printf("Elemento nº %d",i+1);

scanf("%d",&array[i]);

}

visualizar(&array[0]);

}

void visualizar(int array[]) /* desarrollo */

{

int i;

for (i=0;i<25;i++) printf("%d",array[i]);

}
```

En el ejemplo se puede apreciar la forma de enviar un array por referencia. La función se podía haber declarado de otra manera, aunque funciona exactamente igual:

declaración o prototipo

void visualizar(int *);

desarrollo de la función

void visualizar(int *array)

11.2.− Matrices

Una matriz es un array *multidimensional*. Se definen igual que los vectores excepto que se requiere un índice por cada dimensión.

Su sintaxis es la siguiente:

tipo nombre [tamaño 1][tamaño 2]...;

Una matriz *bidimensional* se podría representar gráficamente como una tabla con filas y columnas.

La matriz *tridimensional* se utiliza, por ejemplo, para trabajos gráficos con objetos **3D**.

En el ejemplo puedes ver como se rellena y visualiza una matriz *bidimensional*. Se necesitan dos bucles para cada una de las operaciones. Un bucle controla las filas y otro las columnas.

Ejemplo:

```
/* Matriz bidimensional. */

#include <stdio.h>

main() /* Rellenamos una matriz */

{

int x,i,numeros[3][4];

/* rellenamos la matriz */

for (x=0;x<3;x++)

for (i=0;i<4;i++)

scanf("%d",&numeros[x][i]);

/* visualizamos la matriz */

for (x=0;x<3;x++)

for (i=0;i<4;i++)

printf("%d",numeros[x][i]);

}
```

Si al declarar una matriz también queremos inicializarla, habrá que tener encuenta el orden en el que los

valores son asignados a los elementos de la matriz. Veamos algunos ejemplos:

int numeros[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};

quedarían asignados de la siguiente manera:

numeros[0][0]=1 numeros[0][1]=2 numeros[0][2]=3 numeros[0][3]=4

numeros[1][0]=5 numeros[1][1]=6 numeros[1][2]=7 numeros[1][3]=8

numeros[2][0]=9 numeros[2][1]=10 numeros[2][2]=11 numeros[2][3]=12

También se pueden inicializar cadenas de texto:

char dias[7][10]={"lunes","martes","miércoles","jueves","viernes","sábado","domingo"};

Para referirnos a cada palabra bastaría con el primer índice:

printf("%s",dias[i]);

12.− PUNTEROS

Un puntero es una variable que contiene la dirección de memoria de otra variable. Se utilizan para pasar información entre una función y sus puntos de llamada.

12.1.− Declaración

Su sintaxis es la siguiente:

tipo *nombre;

Donde nombre es, naturalmente, el nombre de la variable, y tipo es el tipo del elemento cuya dirección almacena el puntero.

12.2.− Operadores

Existen dos operadores especiales para trabajar con punteros: **&** y *****.
El primero devuelve la dirección de memoria de su operando. Por ejemplo, si queremos guardar en el puntero **x** la dirección de memoria de la variable **num**, deberemos hacer lo siguiente:

x=#

El segundo devuelve el valor de la variable cuya dirección es contenida por el puntero. Este ejemplo sitúa el contenido de la variable apuntada por **x**, es decir **num**, en la variable **a**:

a=*x;

12.3.− Asignación

Los punteros se asignan igual que el resto de las variables. El programa ejemplo mostrará las direcciones contenidas en **p1** y **p2**, que será la misma en ambos punteros.

Ejemplo:

```
/* Asignaciones de punteros. */

#include <stdio.h>

main() /* Asignamos direcciones */

{

int a;

int *p1,*p2;

p1=&a;

p2=p1;

printf("%p %p",p1,p2);

}
```

12.4.− Aritmética de direcciones

Es posible desplazar un puntero recorriendo posiciones de memoria. Para ello podemos usar los operadores de suma, resta, incremento y decremento (+, −, ++, − −). Si tenemos un puntero (*p1*) de tipo **int** (*2 bytes*), apuntando a la posición 30000 y hacemos: **p1=p1+5;** el puntero almacenará la posición 30010, porque apunta 5 enteros por encima (*10 bytes más*).

13.− ESTRUCTURAS

13.1.− 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 definida anteriormente:

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}};

13.2.− 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)

Ejemplo:

/* 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);
```

}

2.− Por referencia: su declaración sería:

void visualizar(struct trabajador *);

Después declararemos la variable **fijo** y su llamada será:

visualizar(&fijo);

Por último, el desarrollo de la función será:

void visualizar(struct trabajador *datos)

Fíjate que en la función **visualizar**, el acceso a los campos de la variable **datos** se realiza mediante el operador −>, ya que tratamos con un puntero. En estos casos siempre utilizaremos el operador −>. Se consigue con el signo *menos* seguido de *mayor que*.

Ejemplo:

/* Paso de una estructura por referencia. */

#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);

}
```

13.3.− 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}};

13.4.− Typedef

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;
```

14.− 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.

14.1.− 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.

14.2.− 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);
```

14.3.− 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);
```

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.

Puedes encontrar ejemplos sobre la apertura y cierre de ficheros, así como de la lectura y escritura de datos, en el archivo **IMAGECAT.C**. Se trata de un programa que crea un catálogo en formato HTML a partir de las imágenes que se encuentran en un directorio determinado.

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.

15.− GESTION DINAMICA DE MEMORIA

15.1.− Funciones

Como veremos después, la gestión dinámica memoria se realiza mediante estructuras dinámicas de datos. Fíjate que se repite la palabra **dinámica**. Estas estructuras se diferencian de las **estáticas** (arrays y estructuras), en que no tienen un tamaño fijo, es decir, no tenemos que indicar su tamaño al declararlas, sino que podremos aumentarlo o disminuirlo en **tiempo de ejecución**, cuando se esté ejecutando la aplicación. Como puedes ver, las estructuras dinámicas son de gran utilidad. A continuación veremos las funciones que se encargan de reservar y liberar memoria durante la ejecución, que se encuentran en la librería **alloc.h**:

malloc(tamaño);

Esta función reserva en memoria una zona de **tamaño** bytes, y devuelve un puntero al inicio de esa zona. Si no hubiera suficiente memoria retornaría **NULL**. Más adelante veremos algunos ejemplos.

free(puntero);

Esta función libera de la memoria la zona que habíamos reservado anteriormente con la función **malloc**. También podremos ver algún ejemplo en la página siguiente.

15.2.− Estructuras dinámicas de datos

En función de la forma en que se relacionan existen varios tipos de estructuras de datos. Este tipo de estructuras son autorreferenciadas, es decir, contienen entre sus campos un puntero de su mismo tipo. Las más utilizadas son:

− pilas

− colas

− listas

Las pilas

Este tipo de estructuras se caracteriza porque todas las operaciones se realizan en el mismo lado. Es de tipo **LIFO** (**L**ast **I**n **F**irst **O**ut), el último elemento en entrar es el primero en salir.

Ejemplo:

```
/* Ejemplo de una pila. */

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <alloc.h>

void insertar(void);

void extraer(void);

void visualizar(void);

struct pila

{

char nombre[20];

struct pila *ant;

}*CAB=NULL,*AUX=NULL;

main() /* Rellenar, extraer y visualizar */

{

char opc;

do
```

```
{
clrscr(); /* borramos la pantalla */
gotoxy(30,8); /* columna 30, fila 8 */
printf("1.− Insertar");
gotoxy(30,10);
printf("2.− Extraer");
gotoxy(30,12);
printf("3.− Visualizar la pila");
gotoxy(30,14);
printf("4.− Salir");
opc=getch( );
switch(opc)
{
case '1':
insertar( );
break;
case '2':
extraer( );
break;
case '3':
visualizar( );
}
}while (opc!='4');
}
void insertar(void)
{
```

```
AUX=(struct pila *)malloc(sizeof(struct pila));
clrscr();
printf("Nombre: ");
gets(AUX−>nombre);
if (CAB==NULL)
{
CAB=AUX;
AUX−>ant=NULL;
}
else
{
AUX−>ant=CAB;
CAB=AUX;
}
}
void extraer(void)
{
if (CAB==NULL) return;
AUX=CAB;
CAB=CAB−>ant;
free(AUX);
}
void visualizar(void)
{
if (CAB==NULL) return;
clrscr();
```

```
AUX=CAB;

while (AUX!=NULL)

{

printf("Nombre: %s\n",AUX−>nombre);

AUX=AUX−>ant;

}

getch( );

}
```

La estructura tipo que utilizaremos será ésta:

```
struct pila

{

tipo variables;

struct pila *ant;

}*CAB=NULL,*AUX=NULL;
```

donde **tipo variables** serán las diferentes variables que guardaremos en la estructura, **struct pila *ant** es un puntero que apunta al elemento de tipo **pila** introducido anteriormente, ***CAB** será donde guardaremos el último elemento insertado en la pila y ***AUX** nos servirá para guardar elementos temporalmente y para recorrer la pila al visualizarla.

Antes de insertar un elemento, deberemos comprobar si la pila está vacía o no. Si lo estuviera deberemos insertar el primer elemento:

```
CAB=AUX;

CAB−>ant=NULL;
```

Si ya hubiera algún elemento crearemos uno nuevo apuntado por **AUX** y haremos que **AUX−>ant** apunte a **CAB**, que en este momento contiene la dirección del elemento insertado anteriormente. Tras esto haremos que **CAB** apunte al último elemento insertado, que será la nueva cabeza de la pila:

```
AUX−>ant=CAB;

CAB=AUX;
```

Para extraer un elemento de la pila deberemos hacer que **AUX** apunte a la misma dirección que **CAB**, después haremos que **CAB** apunte a **CAB−>ant**, con lo que el elemento anterior pasará a ser la cabeza de la pila. Tras esto, solo queda liberar la memoria de la zona apuntada por **AUX**. No olvides controlar si existe algún elemento (si **CAB** es igual a **NULL** la pila está vacía):

if (CAB==NULL) return;

AUX=CAB;

CAB=CAB−>ant;

free(AUX);

Por último, para visualizar los elementos de la pila, haremos que el puntero auxiliar **AUX** apunte a la cabeza de la pila, o sea, a **CAB**. Tras esto iremos visualizando el contenido de la pila, haciendo que **AUX** tome la dirección de **AUX−>ant**, mientras **AUX** sea distinto de **NULL**. También es importante controlar que la pila no esté vacía.

if (CAB==NULL) return;

AUX=CAB;

while (AUX!=NULL)

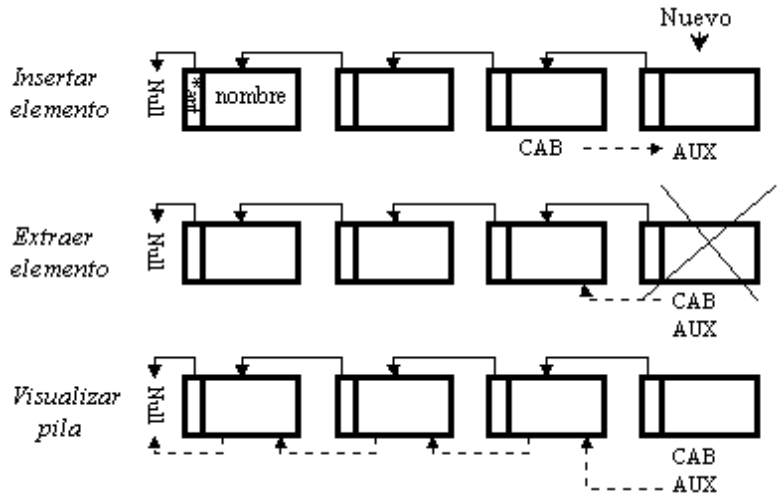
{

printf("%s",AUX−>nombre);

AUX=AUX−>ant;

};

Estructura gráfica de una pila:



Las colas

Este tipo de estructuras se caracteriza porque insertamos los elementos por un lado y los extraemos por el otro lado. Es de tipo **FIFO** (**F**irst **I**n **F**irst **O**ut), el primer elemento en entrar es el primero en salir. Para gestionar la cola utilizaremos **3** punteros (para la pila solo eran necesarios **2**).

Ejemplo:

```
/* Ejemplo de una cola. */

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <alloc.h>

void insertar(void);

void extraer(void);

void visualizar(void);

struct cola

{

char nombre[20];

struct cola *sig;

}*CAB=NULL,*AUX=NULL,*FIN=NULL;

main() /* Rellenar, extraer y visualizar */

{

char opc;

do

{

clrscr();

gotoxy(30,8);

printf("1.− Insertar");

gotoxy(30,10);

printf("2.− Extraer");

gotoxy(30,12);

printf("3.− Visualizar la cola");

gotoxy(30,14);
```

```
printf("4.− Salir");
opc=getch( );
switch(opc)
{
case '1':
insertar( );
break;
case '2':
extraer( );
break;
case '3':
visualizar( );
}
}while (opc!='4');
}
void insertar(void)
{
AUX=(struct cola *)malloc(sizeof(struct cola));
clrscr();
printf("Nombre: ");
gets(AUX−>nombre);
AUX−>sig=NULL;
if (FIN==NULL)
FIN=CAB=AUX;
else
{
```

```
FIN−>sig=AUX;
FIN=AUX;
}
}
void extraer(void)
{
if (CAB==NULL) return;
AUX=CAB;
CAB=CAB−>sig;
free(AUX);
}
void visualizar(void)
{
if (CAB==NULL) return;
clrscr();
AUX=CAB;
while (AUX!=NULL)
{
printf("Nombre: %s\n",AUX−>nombre);
AUX=AUX−>sig;
}
getch();
}
```

La estructura que utilizaremos será:

```
struct cola
{
```

tipo variables;

struct cola *sig;

}*CAB=NULL,*AUX=NULL,*FIN=NULL;

donde **tipo variables** serán las diferentes variables que guardaremos en la estructura, **struct cola *sig** es un puntero que apunta al elemento de tipo **cola** introducido a continuación, ***CAB** será donde guardaremos el primer elemento insertado en la cola, ***AUX** nos servirá para guardar elementos temporalmente y para recorrer la cola al visualizarla y ***FIN** tomará la dirección del último elemento insertado.

Antes de insertar un elemento, deberemos comprobar si la cola está vacía o no. Si lo está deberemos insertar el primer elemento:

if (FIN==NULL)

CAB=FIN=AUX;

Si ya existiera algún elemento haremos que **FIN−>sig** apunte al elemento de **AUX** y a continuación haremos que **FIN** tome la dirección de **AUX**, con lo que **FIN** apuntará al último elemento insertado.

FIN−>sig=AUX;

FIN=AUX;

Para extraer un elemento de la cola haremos que el puntero auxiliar **AUX** tome la dirección del primer elemento insertado, que hemos guardado en **CAB**. Tras esto haremos que **CAB** apunte a **CAB−>sig**, es decir, que tome la dirección del segundo elemento insertado, que ahora pasará a ser el primero. Luego liberaremos la zona de memoria apuntada por **AUX**:

AUX=CAB; /* Deberemos controlar que no esté vacía: if (CAB==NULL) return; */

CAB=CAB−>sig;

free(AUX);

Para visualizar la cola comprobaremos que existan elementos, esto es, que **FIN** sea distinto de **NULL**. Hecho esto asignaremos a **AUX** la dirección de **CAB** e iremos recorriendo la cola hasta que **AUX** sea igual a **NULL**.

AUX=CAB; /* Deberemos controlar que no esté vacía: if (CAB==NULL) return; */

while(AUX!=NULL)

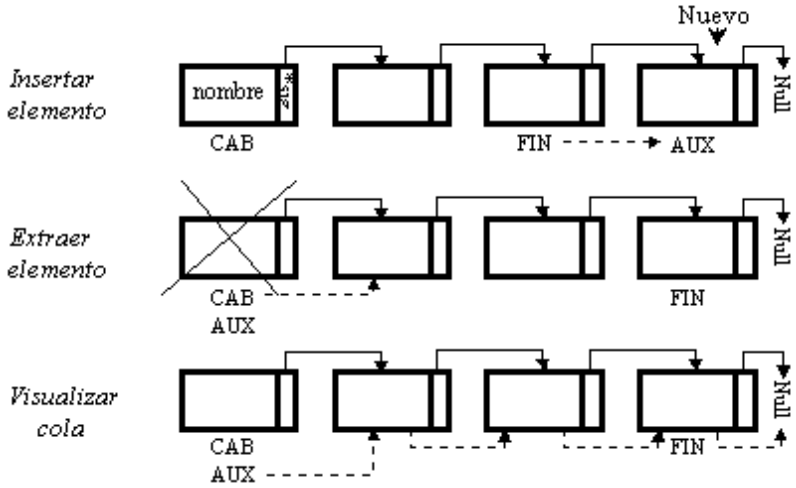
{

printf("%s",AUX−>nombre);

AUX=AUX−>sig;

}

Estructura gráfica de una cola:



Las listas

Este tipo de estructuras se caracteriza porque los elementos están enlazados entre sí, de manera que además de las acciones habituales de insertar, extraer y visualizar también podremos buscar un elemento. Para gestionar la lista utilizaremos **4** punteros.

Ejemplo:

```
/* Ejemplo de una lista. */

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <alloc.h>

void insertar(void);

void extraer(void);

void visualizar(void);

struct lista

{

int num;

struct lista *sig;

}*CAB=NULL,*AUX=NULL,*F=NULL,*P=NULL;

main() /* Rellenar, extraer y visualizar */

{
```

```
char opc;
do
{
clrscr( );
gotoxy(30,8);
printf("1.− Insertar");
gotoxy(30,10);
printf("2.− Extraer");
gotoxy(30,12);
printf("3.− Visualizar la lista");
gotoxy(30,14);
printf("4.− Salir");
opc=getch( );
switch(opc)
{
case '1':
insertar( );
break;
case '2':
extraer( );
break;
case '3':
visualizar( );
}
}while (opc!='4');
}
```

```
/* A continuación insertaremos el elemento que
vamos a crear en la posición que le corresponda,
teniendo en cuenta que la lista deberá quedar
ordenada de menor a mayor. El puntero P comprueba
si el campo num de un elemento es menor que el
campo num del elemento introducido. El puntero
F se quedará apuntando al elemento de la posición
anterior al elemento que hemos insertado */
void insertar(void)
{
AUX=(struct lista *)malloc(sizeof(struct lista));
clrscr( );
printf("Introduce un número: ");
scanf("%d",&AUX->num);
AUX->sig=NULL;
if (CAB==NULL)
CAB=AUX;
else if (CAB->num > AUX->num)
{
AUX->sig=CAB;
CAB=AUX;
}
else
{
P=F=CAB;
while (P->num < AUX->num && P!=NULL)
```

```
{
if (P==CAB) P=P−>sig;
else
{
P=P−>sig;
F=F−>sig;
}
}
AUX−>sig=F−>sig;
F−>sig=AUX;
}
}
void extraer(void)
{
int var;
if (CAB==NULL) return;
clrscr( );
printf("Introduce el número a extraer: ");
scanf("%d",&var);
if (CAB−>num==var)
{
P=CAB;
CAB=CAB−>sig;
free(P);
}
else
```

```
{
P=F=CAB;
while (P−>num != var && P!=NULL)
{
if (P==CAB) P=P−>sig;
else
{
P=P−>sig;
F=F−>sig;
}
}
if (P==NULL) return;
F−>sig=P−>sig;
free(P);
}
}
void visualizar(void)
{
if (CAB==NULL) return;
clrscr( );
AUX=CAB;
while (AUX!=NULL)
{
printf("Número: %d\n",AUX−>num);
AUX=AUX−>sig;
}
```

getch( );

}

La estructura que utilizaremos será:

struct lista

{

tipo variables;

struct lista *sig;

}*CAB=NULL,*AUX=NULL,*F=NULL,*P=NULL;

donde **tipo variables** serán las variables que guardaremos en la estructura, **struct lista *sig** es un puntero que apunta al elemento de tipo **lista** introducido a continuación, ***CAB** será donde guardaremos el primer elemento de la lista, ***AUX** nos servirá para guardar elementos temporalmente y para recorrer la lista al visualizarla, ***P** para comparar los valores introducidos y ordenarlos, y ***F**, que apuntará al elemento anterior al último introducido.

Antes de insertar un elemento, deberemos comprobar si la lista está vacía o no. Si lo está deberemos insertar el primer elemento:

if (CAB==NULL) CAB=AUX;

Si ya existiera algún elemento haremos que **P** y **F** apunten al primero de la lista. Si el elemento introducido fuera menor que el primero de la lista, haríamos que el nuevo elemento pasara a ser el primero, y el que hasta ahora era el primero, pasaría a ser el segundo.

if (AUX−>num < CAB−>num){

AUX−>sig=CAB;

CAB=AUX;

}

Para extraer un elemento de la lista solicitaremos un número, si el número introducido se corresponde con el campo num de uno de los elementos, éste será extraído de la lista. Deberemos controlar que la lista no esté vacía y que el elemento con el número solicitado exista.

Fíjate en el ejemplo, en la función extraer. Si **CAB** es igual a **NULL**, será que la lista está vacía, y si **P** es igual a **NULL** al salir del **while** significará que no se ha encontrado ningún elemento que contenga el número introducido.

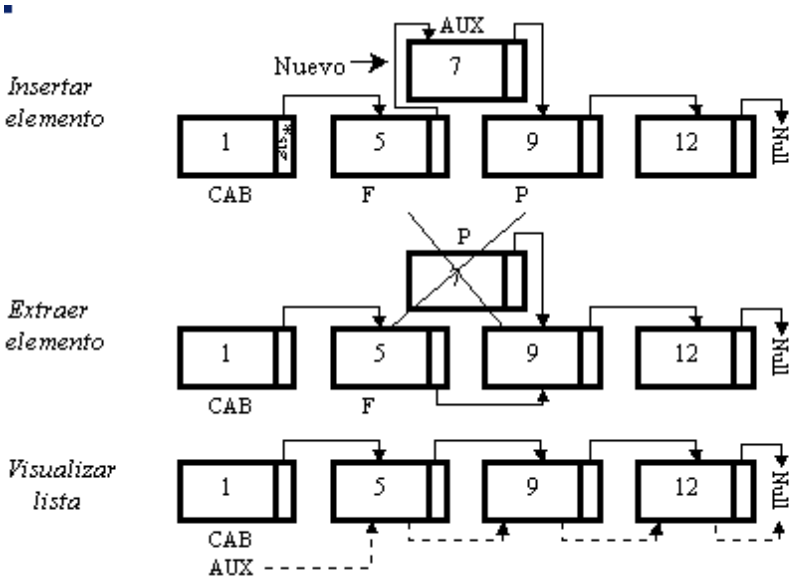
Para visualizar la lista comprobaremos que existan elementos, es decir, que **CAB** sea distinto de **NULL**. Hecho esto asignaremos a **AUX** la dirección de **CAB** e iremos recorriendo la lista mientras **AUX** sea distinto de **NULL**.

if (CAB==NULL) return;

AUX=CAB;

```
while(AUX!=NULL)

{

printf("%d",AUX->num);

AUX=AUX->sig;

}
```

Estructura gráfica de una lista:



Aquí finaliza el tema de la gestión dinámica de memoria. Es un tema algo complejo hasta que se asimila el concepto y funcionamiento de las diferentes estructuras, pero tras conseguirlo ya no tiene ningún secreto. Si alguna vez no recuerdas su funcionamiento siempre es una buena solución coger papel y lápiz, dibujar una pila, cola o lista gráficamente y simular la introducción de elementos, escribiendo la situación de los punteros en cada momento.

Existen otras estructuras, como las **listas doblemente enlazadas**. La única diferencia con la **lista** que conocemos es que en las primeras cada elemento guarda la dirección del anterior y del posterior. Sería una estructura como esta:

```
struct lista_doble

{

char nombre[20];

struct lista_doble *ant;

struct lista_doble *sig;

};
```

Su funcionamiento es muy similar al de una lista normal. Puedes intentar hacerla tu mismo.
Otras estructuras, como los **árboles** son más complejas y menos utilizadas.

16.− PROGRAMACION GRAFICA

16.1.− Conceptos básicos

El estándar de 'C' no define ninguna función gráfica debido a las grandes diferencias entre las interfaces de los distintos tipos de hardware. Nosotros veremos el conjunto de funciones que utiliza Turbo C. La resolución más habitual del modo gráfico en Turbo C es de **640x480x16**.

Inicialización del modo gráfico

Para poder trabajar en modo gráfico primero deberemos inicializarlo. Las funciones a utilizar son estas.

detectgraph (int *tarjeta , int *modo);

Detecta el tipo de tarjeta que tenemos instalado. Si en el primer argumento retorna **−2** indica que no tenemos ninguna tarjeta gráfica instalada (cosa bastante improbable).
initgraph (int *tarjeta , int *modo , "path");

Inicializa el modo gráfico (primero hay que usar **detectgraph**). En **path** deberemos indicar el directorio donde se encuentra el archivo **EGAVGA.BGI**.
int graphresult( );

Retorna el estado del modo gráfico. Si no se produce ningún error devuelve **0**, de lo contrario devuelve un valor entre **−1** y **−16**.
char grapherrormsg(int error);

Retorna un puntero al mensaje de error indicado por **graphresult**.

Finalización del modo gráfico

closegraph( );

Cierra el modo gráfico y nos devuelve al modo texto.
restorecrtmode( );

Reestablece el modo de video original (anterior a **initgraph**).

Ejemplo:

```
/* Inicialización del modo gráfico. */

#include <graphics.h>

main() /* Inicializa y finaliza el modo gráfico. */

{

int tarjeta, modo, error;
```

```
detectgraph(&tarjeta,&modo);
initgraph(&tarjeta,&modo,"C:\\TC\\BGI");
error=graphresult( );
if (error)
{
printf("%s",grapherrormsg(error));
}
else
{
getch( );
closegraph( );
}
}
```

16.2.− Funciones

int getmaxx( );

Retorna la coordenada máxima horizontal, probablemente **639**. Ej: **hm=getmaxx( );**

int getmaxy( );

Retorna la coordenada máxima vertical, probablemente **479**. Ej: **vm=getmaxy( );**

int getx( );

Retorna la coordenada actual horizontal. Ej: **hact=getx( );**

int gety( );

Retorna la coordenada actual vertical. Ej: **vact=gety( );**

moveto(int x , int y);

Se mueve a las coordenadas indicadas. Ej: **moveto(320,240);**

setcolor(color);

Selecciona el color de dibujo y texto indicado. Ej: **setcolor(1);** o **setcolor(BLUE);**

setbkcolor(color);

Selecciona el color de fondo indicado. Ej: **setbkcolor(4);** o **setbkcolor(RED);**

int getcolor( );

Retorna el color de dibujo y texto actual. Ej: **coloract=getcolor( );**

int getbkcolor( );

Retorna el color de fondo actual. Ej: **fondoact=getbkcolor( );**

int getpixel(int x , int y);

Retorna el color del pixel en x,y. Ej: **colorp=getpixel(120,375);**

cleardevice( );

Borra la pantalla. Ej: **cleardevice( );**

Funciones de dibujo

putpixel(int x , int y , color);

Pinta un pixel en las coordenadas y color indicados. Ej: **putpixel(100,50,9);**

line(int x1 , int y1 , int x2 , int y2);

Dibuja una linea desde x1,y1 a x2,y2. Ej: **line(20,10,150,100);**

circle(int x , int y , int radio);

Dibuja un círculo del radio indicado y con centro en x,y. Ej: **circle(320,200,20);**

rectangle(int x1 , int y1 , int x2 , int y2);

Dibuja un rectángulo con la esquina superior izquierda en x1,y1 y la inferior derecha en x2,y2. Ej: **rectangle(280,210,360,270);**

arc(int x , int y, int angulo1 , int angulo2 , int radio);

Dibuja un arco cuyo centro está en x,y, de radio r, y que va desde angulo1 a angulo2. Ej: **arc(200,200,90,180,40);**

setlinestyle(int estilo, 1 , grosor);

Selecciona el estilo de linea a utilizar. El estilo puede tomar un valor de **0** a **4**. El grosor puede tomar dos valores: **1** = normal y **3** = ancho. Ej: **setlinestyle(2,1,3);**

Funciones de relleno

floodfill(int x , int y , int frontera);

Rellena el area delimitada por el color indicado en frontera comenzando desde x,y. Ej: **floodfill(100,30,12);**

setfillstyle(int pattern , int color);

Selecciona el patrón y el color de relleno. El patrón puede tomar un valor de **0** a **12** Ej: **setfillstyle(1,9);**

bar(int x1 , int y1, int x2 , int y2);

Dibuja una barra (rectángulo) y si es posible la rellena. Ej: **bar(200,200,400,300);**

bar3d(int x1 , int y1, int x2 , int y2 , int profundidad , int tapa);

Dibuja una barra en 3d, son los mismos valores que bar además de la profundidad y la tapa: **0** si la queremos sin tapa y **1** si la queremos con tapa. Ej: **bar3d(100,100,400,150,40,1);**

pieslice(int x , int y , int angulo1 , int angulo2 , int radio);

Dibuja un sector. Hace lo mismo que arc, pero además lo cierra y lo rellena. Ej: **pieslice(250,140,270,320,50);**

Funciones de escritura de texto

outtextxy(int x , int y , char *);

Muestra el texto indicado (puede ser un array o puede escribirse al llamar a la función) en las coordenadas x,y. Ej: **outtextxy(50,50,"Esto es texto en modo gráfico");**

settextstyle(int fuente , int dirección , int tamaño);

Selecciona el estilo del texto. Las fuentes más comunes son las que van de **0** a **4**. La dirección puede ser: **0** = horizontal y **1** = vertical. El tamaño puede tomar un valor de **1** a **10**. Ej: **settextstyle(2,0,5);**

setviewport(int x1 , int y1 , int x2 , int y2 , int tipo);

Define una porción de pantalla para trabajar con ella. La esquina superior izquierda está determinada por x1,y1 y la inferior derecha por x2,y2. Para tipo podemos indicar **1**, en cuyo caso no mostrará la parte de un dibujo que sobrepase los límites del viewport, o distinto de **1**, que sí mostrará todo el dibujo aunque sobrepase los límites. Al activar un viewport, la esquina superior izquierda pasará a tener las coordenadas (0,0). Para volver a trabajar con la pantalla completa, deberemos escribir: **viewport(0,0,639,479,1);**.

clearviewport( );

Borra el contenido del viewport.

Aquí concluye el tema del modo gráfico. Hay algunas funciones más, aunque su complejidad es mayor. Generalmente no se suelen utilizar más que las aquí descritas, pero puedes investigar en la ayuda de Turbo C para conocer alguna otra.

17.− APENDICE

En este capítulo y para finalizar veremos los ficheros de cabecera, donde están declaradas las funciones que

utilizaremos habitualmente.

17.1.− Librería stdio.h

printf
Función: Escribe en la salida estándar con formato.

Sintaxis: printf(formato , arg1 , ...);

scanf
Función: Lee de la salida estándar con formato.
Sintaxis: scanf(formato , arg1 , ...);

puts
Función: Escribe una cadena y salto de linea.
Sintaxis: puts(cadena);

gets
Función: Lee y guarda una cadena introducida por teclado.
Sintaxis: gets(cadena);

fopen
Función: Abre un fichero en el modo indicado.
Sintaxis: pf=fopen(fichero , modo);

fclose
Función: Cierra un fichero cuyo puntero le indicamos.
Sintaxis: fclose(pf);

fprintf
Función: Escribe con formato en un fichero.
Sintaxis: fprintf(pf , formato , arg1 , ...);

fgets
Función: Lee una cadena de un fichero.
Sintaxis: fgets(cadena , longitud , pf);

17.2.− Librería stdlib.h

atof
Función: Convierte una cadena de texto en un valor de tipo float.
Sintaxis: numflo=atof(cadena);

atoi
Función: Convierte una cadena de texto en un valor de tipo entero.
Sintaxis: nument=atoi(cadena);

itoa
Función: Convierte un valor numérico entero en una cadena de texto. La base generalmente será 10, aunque se puede indicar otra distinta.
Sintaxis: itoa(número , cadena , base);

exit
Función: Termina la ejecución y abandona el programa.
Sintaxis: exit(estado); /* Normalmente el estado será 0 */

17.3.− Librería conio.h

clrscr
Función: Borra la pantalla.
Sintaxis: clrscr( );

clreol
Función: Borra desde la posición del cursor hasta el final de la linea.
Sintaxis: clreol( );

gotoxy
Función: Cambia la posición del cursor a las coordenadas indicadas.
Sintaxis: gotoxy(columna , fila);

textcolor
Función: Selecciona el color de texto (0 − 15).
Sintaxis: textcolor(color);

textbackground
Función: Selecciona el color de fondo (0 − 7).
Sintaxis: textbackground(color);

wherex
Función: Retorna la columna en la que se encuentra el cursor.
Sintaxis: col=wherex( );

wherey
Función: Retorna la fila en la que se encuentra el cursor.
Sintaxis: fila=wherey( );

getch
Función: Lee y retorna un único caracter introducido mediante el teclado por el usuario. No muestra el caracter por la pantalla.
Sintaxis: letra=getch( );

getche
Función: Lee y retorna un único caracter introducido mediante el teclado por el usuario. Muestra el caracter por la pantalla.
Sintaxis: letra=getche( );

17.4.− Librería string.h

strlen
Función: Calcula la longitud de una cadena.
Sintaxis: longitud=strlen(cadena);

strcpy
Función: Copia el contenido de una cadena sobre otra.

Sintaxis: strcpy(copia , original);

strcat
Función: Concatena dos cadenas.
Sintaxis: strcat(cadena1 , cadena2);

strcmp
Función: Compara el contenido de dos cadenas. Si **cadena1 < cadena2** retorna un número negativo. Si **cadena1 > cadena2**, un n£mero positivo, y si **cadena1** es igual que **cadena2** retorna **0** (o **NULL**).
Sintaxis: valor=strcmp(cadena1 , cadena2);

17.5.− Librería graphics.h

Además de las que vimos al estudiar la programación gráfica existen otras funciones. Aquí tienes algunas de ellas.

getmaxcolor
Función: Retorna el valor más alto de color disponible.
Sintaxis: mcolor=getmaxcolor( );

setactivepage
Función: En modos de video con varias páginas, selecciona la que recibirá todas las operaciones y dibujos que realicemos.
Sintaxis: setactivepage(página); /* En modo VGA página = 0 ó 1 */

setvisualpage
Función: En modos de video con varias páginas, selecciona la que se visualizará por pantalla.
Sintaxis: setvisualpage(página);

17.6.− Librería dir.h

En esta librería encontraremos una serie de rutinas que nos permitirán realizar operaciones básicas con directorios y unidades de disco.

chdir
Función: Cambia el directorio actual.
Sintaxis: chdir(ruta); /* Podemos indicar la unidad: chdir("a:\\DATOS"); */

getcwd
Función: Lee del sistema el nombre del directorio de trabajo.
Sintaxis: getcwd(directorio,tamañocad) /* Lee el directorio y la unidad */

getdisk
Función: Lee del sistema la unidad actual.
Sintaxis: disk=getdisk( ) + 'A'; /* Retorna un entero: 0 = A: , 1 = B: ... */

mkdir
Función: Crea un directorio.
Sintaxis: mkdir(nombre);

17.7.− Funciones interesantes

fflush(stdin)
Función: Limpia el buffer de teclado.
Sintaxis: fflush(stdin);
Prototipo: stdio.h

sizeof
Función: Operador que retorna el tamaño en bytes de una variable.
Sintaxis: tamaño=sizeof(variable);

cprintf
Función: Funciona como el printf pero escribe en el color que hayamos activado con la función textcolor sobre el color activado con textbackground.
Sintaxis: cprintf(formato , arg1 , ...);
Prototipo: conio.h

kbhit
Función: Espera la pulsación de una tecla para continuar la ejecución.
Sintaxis: while (!kbhit( )) /* Mientras no pulsemos una tecla... */
Prototipo: conio.h

random
Función: Retorna un valor aleatorio entre 0 y num−1.
Sintaxis: valor=random(num); /* También necesitamos la función randomize */
Prototipo: stdlib.h

randomize
Función: Inicializa el generador de números aleatorios. Deberemos llamarlo al inicio de la función en que utilicemos el random. También deberemos utilizar el include **time.h**, ya que randomize hace una llamada a la función **time**, incluída en este último archivo.
Sintaxis: randomize( );
Prototipo: stdio.h

system
Función: Ejecuta el comando indicado. Esto incluye tanto los comandos del sistema operativo, como cualquier programa que nosotros le indiquemos. Al acabar la ejecución del comando, volverá a la linea de código situada a continuación de la sentencia system.
Sintaxis: system(comando); /* p.ej: system("arj a programa"); */
Prototipo: stdlib.h

Aquí finaliza este **Curso de Programación en C**. A lo largo de todas sus páginas he intentado describir los metodos, funciones, sentencias, operadores... para poder programar en 'C'.

Naturalmente el 'C' no se acaba aquí, pero espero que con lo que hayas aprendido puedas comenzar a investigar por tu cuenta, de forma que comprendas el funcionamiento de cualquier código fuente que se te presente.

Animo y... que tengas mucha suerte en tu faceta de programador.

37