Lenguaje C

Enrique Vicente Bonet Esteban

Tema 1 - Introducción.

El lenguaje de programación C fue creado por Brian Kernighan y Dennis Ritchie a mediados de los años 70. La primera implementación del mismo la realizó Dennis Ritchie sobre un computador DEC PDP-11 con sistema operativo UNIX. C es el resultado de un proceso de desarrollo que comenzó con un lenguaje anterior, el BCPL, el cual influyó en el desarrollo por parte de Ken Thompson de un lenguaje llamado B, el cual es el antecedente directo del lenguaje C. El lenguaje C es un lenguaje para programadores en el sentido de que proporciona una gran flexibilidad de programación y una muy baja comprobación de incorrecciones, de forma que el lenguaje deja bajo la responsabilidad del programador acciones que otros lenguajes realizan por si mismos. Así, por ejemplo, C no comprueba que el índice de referencia de un vector (llamado array en la literatura informática) no sobrepase el tamaño del mismo; que no se escriba en zonas de memoria que no pertenecen al área de datos del programa, etc.

El lenguaje C es un lenguaje estructurado, en el mismo sentido que lo son otros lenguajes de programación tales como el lenguaje Pascal, el Ada o el Modula-2, pero no es estructurado por bloques, o sea, no es posible declarar subrutinas (pequeños trozos de programa) dentro de otras subrutinas, a diferencia de como sucede con otros lenguajes estructurados tales como el Pascal. Además, el lenguaje C no es rígido en la comprobación de tipos de datos, permitiendo fácilmente la conversión entre diferentes tipos de datos y la asignación entre tipos de datos diferentes, por ejemplo la expresión siguiente es válida en C:

```
float a; /*Declaro una variable para numeros reales*/
int b; /*Declaro otra variable para numero enteros*/
b=a; /*Asigno a la variable para entera el numero real*/
```

Todo programa de C consta, básicamente, de un conjunto de funciones, y una función llamada ***main*****,** la cual es la primera que se ejecuta al comenzar el programa, llamándose desde ella al resto de funciones que compongan nuestro programa.

Desde su creación, surgieron distintas versiones de C, que incluían unas u otras características, palabras reservadas, etc. Este hecho provoco la necesidad de unificar el lenguaje C, y es por ello que surgió un standard de C, llamado ANSI-C, que declara una serie de características, etc., que debe cumplir todo lenguaje C. Por ello, y dado que todo programa que se desarrolle siguiendo el standard ANSI de C será fácilmente portable de un modelo de ordenador a otro modelo de ordenador, y de igual forma de un modelo de compilador a otro, en estos apuntes explicaremos un C basado en el standard ANSI-C.

El lenguaje C posee un número reducido de palabras reservadas (tan solo 32) que define el standard ANSI-C. Estas palabras reservadas pueden verse en la tabla siguiente:

auto	break	case	char	const	continue	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short

signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

Tabla 1.1: Palabras reservadas del lenguaje C.

Tema 2 - Identificadores, tipos de datos, variables y constantes.

2.1 - Identificadores.

Antes de proceder a explicar los identificadores en C, es necesario resaltar que C es un lenguaje sensible al contexto, a diferencia por ejemplo de Pascal, por lo cual, C diferencia entre mayúsculas y minúsculas, y por tanto, diferencia entre una palabra escrita total o parcialmente en mayúsculas y otra escrita completamente en minúsculas.

En el lenguaje C, un identificador es cualquier palabra no reservada que comience por una letra o por un subrayado, pudiendo contener en su interior letras, números y subrayados. La longitud máxima de un identificador depende del compilador que se este usando, pero, generalmente, suelen ser de 32 caracteres, ignorándose todos aquellos caracteres que compongan el identificador y sobrepasen la longitud máxima. Recuérdese, además, que al ser C sensible al contexto, un identificador escrito como *esto_es_un_ident* y otra vez como *Esto_Es_Un_Ident* será interpretado como dos identificadores completamente distintos.

2.2 - Tipos de datos, modificadores de tipo y modificadores de acceso.

En C, toda variable, antes de poder ser usada, debe ser declarada, especificando con ello el tipo de dato que almacenara. Toda variable en C se declara de la forma:

```
<tipo de dato> <nombre de variable> [, nombre de variable];
```

En C existen cinco tipos de datos según puede verse en la tabla siguiente:

Tipo de dato	Descripción.
`char`	Carácter o entero pequeño (byte)
`int`	Entero
`float`	Punto flotante
`double`	Punto flotante (mayor rango que *float*)
`void`	Sin tipo (uso especial)

Tabla 2.2.1: Tipos de datos en C.

Algunos ejemplos de variables de C serían:

```
float a;
int b,c;
char caracter,otro_caracter;
```

Existen, además, cuatro modificadores de tipo, los cuales se aplican sobre los tipos de datos anteriormente citados. Los modificadores de tipo permiten cambiar el tamaño, etc., de los tipos de datos anteriormente especificados. Estos modificadores, que sintácticamente anteceden a la declaración del tipo de dato, son:

Modificador	Tipos de actuación		Descripción
`signed`	`char`	`int`	Con signo (por defecto)
`unsigned`	`char`	`int`	Sin signo
`long`	`int`	`double`	Largo
`short`	`int`		Corto

Tabla 2.2.2: Modificadores de los tipos de datos en C.

Es por ello, que podemos declarar variables como:

```
unsigned char a;
long double b;
short int i;
```

Es posible, además, aplicar dos modificadores seguidos a un mismo tipo de datos, así, es posible definir una variable de tipo *unsigned long int* (entero largo sin signo). El rango de valores de que permite cada variable depende del sistema operativo sobre el cual se trabaje (MS-DOS, Windows95/98/NT/2000, UNIX/Linux), por lo cual conviene referirse al manual del compilador para conocerlo. De forma general, los sistemas operativos de 16 bits (MS-DOS, Windows 16 bits) poseen un rango y los de 32 bits (Windows 32 bits, UNIX/Linux) otro.

	Rango de valores posibles en (notación matemática)	
Tipo de variable declarada	**16 bits**	**32 bits**
`char / signed char`	[-128 , 127]	[-128 , 127]
`unsigned char`	[0 , 255]	[0 , 255]
`int / signed int`	[-32768 , 32767]	[-2147483647 , 2147483648]
`unsigned int`	[0 , 65535]	[0 , 4294967295]
`short int / signed short int`	[-32768 , 32767]	[-32768 , 32767]
`unsigned short int`	[0 , 65535]	[0 , 65535]
`long int / signed long int`	[-2147483647 , 2147483648]	[-2147483647 , 2147483648]
`unsigned long int`	[0 , 4294967295]	[0 , 4294967295]
`float`	[-3.4E+38 , -3.4E-38], 0 , [3.4E-38 , 3.4E+38]	[-3.4E+38 , -3.4E-38], 0 , [3.4E-38 , 3.4E+38]
`double`	[-1.7E+308 , -1.7E-308], 0 , [1.7E-308 , 1.7E+308]	[-1.7E+308 , -1.7E-308], 0 , [1.7E-308 , 1.7E+308]
`long double`	[-3.4E+4932 , -1.1E-4932], 0 , [3.4E-4932 , 1.1E+4932]	[-3.4E-4932 , -1.1E+4932], 0 , [3.4E-4932 , 1.1E+4932]

Tabla 2.2.3: Rango de valores de las variables en C.

Además de los modificadores de tipo existen modificadores de acceso. Los modificadores de acceso limitan el uso que puede realizarse de las variables declaradas. Los modificadores de acceso anteceden a la declaración del tipo de dato de la variable y son los siguientes:

Modificador	Efecto
`const`	Variable de valor constante
`volatile`	Variable cuyo valor es modificado externamente

Tabla 2.2.4: Modificadores de acceso en C.

La declaración de una variable como *const* permite asegurarse de que su valor no será modificado por el programa, excepto en el momento de su declaración, en el cual debe asignársele un valor inicial. Así, si declaramos la siguiente variable:

```
const int x=237;
```

Cualquier intento posterior de modificar el valor de *x*, tal como *x=x+5;*, producirá un error en tiempo de compilación.

La declaración de una variable como *volatile*, indica al compilador que dicha variable puede modificarse por un proceso externo al propio programa (tal como la hora del sistema), y por ello, que no trate de optimizar dicha variable suponiéndole un valor constante, etc. Ello fuerza a que cada vez que se usa la variable, se realice una comprobación de su valor.

Los modificadores *const* y *volatile* pueden usarse de forma conjunta en ciertos casos, por lo cual no son excluyentes el uno del otro. Ello es posible si se declara una variable que actualizara el reloj del sistema, (proceso externo al programa), y que no queremos pueda modificarse en el interior del programa. Por ello, podremos declarar:

```
volatile const unsigned long int hora;
```

2.3 - Declaración de variables y alcance.

En C, las variables pueden ser declaradas en cuatro lugares del módulo del programa:

- Fuera de todas las funciones del programa, son las llamadas variables globales, accesibles desde cualquier parte del programa.

- Dentro de una función, son las llamadas variables locales, accesibles tan solo por la función en las que se declaran.

- Como parámetros a la función, accesibles de igual forma que si se declararan dentro de la función.

- Dentro de un bloque de código del programa, accesible tan solo dentro del bloque donde se declara. Esta forma de declaración puede interpretarse como una variable local del bloque donde se declara.

Para un mejor comprensión, veamos un pequeño programa de C con variables declaradas de las cuatro formas posibles:

```
#include <stdio.h>
int sum; /* Variable global, accesible desde cualquier parte */
         /* del programa*/
void suma(int x) /* Variable local declarada como parámetro, */
                 /* accesible solo por la función suma*/
{
   sum=sum+x;
   return;
}
void intercambio(int *a,int *b)
{
   if (*a>*b)
   {
      int inter; /* Variable local, accesible solo dentro del */
                 /* bloque donde se declara*/
      inter=*a;
      *a=*b;
      *b=inter;
   }
   return;
}
int main(void) /*Función principal del programa*/
{
   int contador,a=9,b=0; /*Variables locales, accesibles solo */
                         /* por main*/
   sum=0;
   intercambio(&a,&b);
   for(contador=a;contador<=b;contador++) suma(contador);
   printf("%d\n",suma);
   return(0);
}
```

2.4 - Especificadores de almacenamiento de los tipos de datos.

Una vez explicada la declaración de variables y su alcance, vamos a proceder a explicar como es posible modificar el alcance del almacenamiento de los datos. Ello es posible realizarlo mediante los especificadores de almacenamiento. Existen cuatro especificadores de almacenamiento. Estos especificadores de almacenamiento, cuando se usan, deben preceder a la declaración del tipo de dato de la variable. Estos especificadores de almacenamiento son:

Especificador de almacenamiento	Efecto
`auto`	Variable local (por defecto)
`extern`	Variable externa
`static`	Variable estática
`register`	Variable registro

Tabla 2.4.1: Especificadores de almacenamiento en C.

El especificador *auto* se usa para declarar que una variable local existe solamente mientras estemos dentro de la subrutina o bloque de programa donde se declara, pero, dado que por defecto toda variable local es *auto*, no suele usarse.

El especificador *extern* se usa en el desarrollo de programas compuestos por varios módulos. El modificador *extern* se usa sobre las variables globales del módulo, de forma que si una variable global se declara como *extern*, el compilador no crea un almacenamiento para ella en memoria, sino que, tan solo tiene en cuenta que dicha variable ya ha sido declarada en otro modulo del programa y es del tipo de dato que se indica.

El especificador *static* actúa según el alcance de la variable:

- Para variables locales, el especificador *static* indica que dicha variable local debe almacenarse de forma permanente en memoria, tal y como si fuera una variable global, pero su alcance será el que correspondería a una variable local declarada en la subrutina o bloque. El principal efecto que provoca la declaración como *static* de una variable local es el hecho de que la variable conserva su valor entre llamadas a la función.

- Para variables globales, el especificador *static* indica que dicha variable global es local al módulo del programa donde se declara, y, por tanto, no será conocida por ningún otro módulo del programa.

El especificador *register* se aplica solo a variables locales de tipo *char* e *int*. Dicho especificador indica al compilador que, caso de ser posible, mantenga esa variable en un registro de la CPU y no cree por ello una variable en la memoria. Se pueden declarar como *register* cuantas variables se deseen, pues el compilador ignorara dicha declaración caso de no poder ser satisfecha. El uso de variables con especificador de almacenamiento *register* permite colocar en registros de la CPU variables muy frecuentemente usadas, tales como contadores de bucles, etc.

Algunos ejemplos de uso de los especificadores de almacenamiento son:

```
register unsigned int a;
static float b;
extern int c;
static const unsigned long int d;
```

2.5 - Constantes.

En C, las constantes se refieren a los valores fijos que el programa no puede alterar. Algunos ejemplos de constantes de C son:

Tipo de dato	Constantes de ejemplo		
`char`	‘a’	‘9’	'Q'
`int`	1	-34	21000
`long int`	-34	67856L	456
`short int`	10	-12	1500

`unsigned int`	45600U	345	3
`float`	12.45	4.34e-3	-2.8e9
`double`	-34.657	-2.2e-7	1.0e100

Tabla 2.5.1: Tipos de constantes en C.

Existen, además, algunos tipos de constantes, distintos a los anteriores, que es necesario resaltar de forma particular. Estos tipos de constantes son las constantes hexadecimales y octales, las constantes de cadena, y las constantes de barra invertida.

Las constantes hexadecimales y octales son constantes enteras, pero definidas en base 16 (constantes hexadecimales) o en base 8 (constantes octales). Las constantes de tipo hexadecimal comienzan por los caracteres *0x* seguidas del número deseado. Las constantes de tipo octal comienzan por un cero (*0*). Por ejemplo, son constantes hexadecimales *0x34* (52 decimal), *0xFFFF* (65535 decimal); y constantes octales *011* (9 decimal), *0173* (123 decimal)

Las constantes de cadena son conjuntos de caracteres que se encierran entre comillas dobles. Por ejemplo, son constantes de cadena *"Esto es una constante de cadena"*, *"Estos son unos apuntes de C"*, etc.

Las constantes de caracteres de barra invertida se usan para introducir caracteres que es imposible introducir por el teclado (tales como retorno de carro, etc.). Estas constantes son proporcionadas por C para que sea posible introducir dichas caracteres como constantes en los programas en los cuales sea necesario. Estas constantes de caracteres de barra invertida son:

Código	Significado
`\b`	Retroceso
`\f`	Alimentación de hoja
`\n`	Nueva línea
`\r`	Retorno de carro
`\t`	Tabulador horizontal
`\"`	Doble comilla
`\'`	Simple comilla
`\0`	Nulo
`\\`	Barra invertida
`\v`	Tabulador vertical
`\a`	Alerta
`\o`	Constante octal
`\x`	Constante hexadecimal

Tabla 2.5.2: Constantes de barra invertida en C.

El uso de las constantes de barra invertida es igual que el de cualquier otro carácter, así, si *ch* es una variable de tipo *char*, podemos hacer: *ch='\t'*, o *ch='\x20'* (carácter espacio), etc., de igual forma que realizaríamos con cualquier otra constante de carácter. Además, las constantes de barra invertida pueden usarse en el interior de constantes de cadena como un carácter más, por ello, podemos poner escribir la constante de cadena: *"Esto es una linea\n"*.

Tema 3 - Operadores aritméticos, relaciónales y lógicos; operador asignación; operador sizeof y operadores avanzados (operadores sobre bits y operador ?).

3.1 - Operadores aritméticos.

Los operadores aritméticos existentes en C son, ordenados de mayor a menor precedencia:

Operador		**Operador**		**Operador**	
++	Incremento	--	Decremento		
-	Menos unario				
*	Multiplicación.	/	División	%	Módulo
+	Suma	-	Resta		

Tabla 3.1.1: Operadores aritméticos en C.

Los operadores ++, -- y % solo pueden usarse con datos de tipo *int* o *char*. El operador incremento (++), incrementa en una unidad el valor de la variable sobre la que se aplica, el operador decremento (--), decrementa en una unidad el valor de la variable, y el operador módulo (%), calcula el resto de una división de dos variables de tipo entero o carácter.

Un aspecto que conviene explicar es el hecho de que los operadores incremento y decremento pueden preceder o posceder a su operando, lo cual permite escribir, si *x* es una variable de tipo *int*, las expresiones ++*x* o *x*++. Usado de forma aislada no presenta ninguna diferencia, sin embargo, cuando se usa en una expresión existe una diferencia en el orden de ejecución del mismo. Cuando el operador incremento (o decremento) precede al operando, C primero realiza el incremento (o decremento), y después usa el valor del operando, realizándose la operación al contrario si el operador poscede al operando. Así, considérense el siguiente código de un programa:

```
int var1=10,var2;
var2=++var1; /* Pone 11 en var2, pues primero incrementa var1,*/
             /* y luego asigna su valor a var2 */
```

Mientras que el siguiente código funciona de forma distinta:

```
int var1=10,var2;
var2=var1++; /* Pone 10 en var2, pues primero asigna su valor */
             /* a var2, y luego incrementa var1 */
```

3.2 - Operadores relaciónales y lógicos.

Los operadores relaciónales y lógicos de C, ordenados de mayor a menor prioridad son:

Operador		**Operador**		**Operador**		**Operador**	
!	Not						
>	Mayor que	>=	Mayor o igual que	<	Menor que	<=	Menor o igual que
==	Igual	!=	No igual				
&&	And						
||	Or						

Tabla 3.2.1: Operadores relacionales y lógicos en C.

Los operadores relaciónales y lógicos tiene menor prioridad que los operadores aritméticos antes descritos, así , escribir *10>3+9* es equivalente a escribir *10>(3+9)*.

3.3 - Operador asignación.

El lenguaje C, a diferencia de otros lenguajes tales como Pascal, no diferencia la asignación de cualquier otro operador del lenguaje. Para C, la asignación es un operador, el llamado operador asignación (=), el cual posee la prioridad más baja de todos los operadores. Es por ello que en C podemos escribir expresiones del tipo:

```
if ((c=a*b)<0) /* if es la comprobación condicional de C, que */
               /* se vera con posterioridad */
```

Esta expresión asigna a la variable *c* el valor de *a*b* y devuelve su valor para compararlo con el valor constante *0*. Los paréntesis son necesarios pues el operador asignación tiene la prioridad mas baja de todos los operadores.

3.4 - Operador sizeof.

El operador *sizeof* es un operador en tiempo de compilación. El operador *sizeof* devuelve el tamaño de una variable o tipo de dato durante la compilación, no durante la ejecución del programa. Veamos algunos ejemplos:

sizeof(int) devuelve el valor 2 en los sistemas operativos de 16 bits y 4 en los de 32 bits.

Si tenemos *char a[20]*, *sizeof(a)* devuelve el valor 20, y si tenemos *float a[6]*, *sizeof(a)* devuelve el valor 24 (4*6).

3.5 - Operadores sobre bits.

El lenguaje C posee operadores que actúan a nivel de bits sobre los datos, estos operadores son:

Operador	Nombre	Operación
~	Not	Complemento a uno (NOT)
<<	Desplazamiento izquierda	Desplazamiento izquierda
>>	Desplazamiento derecha	Desplazamiento derecha
&	And	Y
^	Xor	O exclusivo (XOR)
|	Or	O

Tabla 3.5.1: Operadores sobre bits en C.

Los operadores &, | y ^ actúan sobre dos operandos, mientras que ~ ,<< **y** >> actúan sobre un solo operando. Veamos su actuación sobre dos valores cualquiera:

Operador	Operando 1	Operando 2	Resultado
~	0xB2		0x4D
<<3	0xB2		0x90
>>2	0xB2		0x2C
&	0xB2	0x79	0x30
^	0xB2	0x79	0xCB
|	0xB2	0x79	0xFB

Tabla 3.5.2: Ejemplos de operaciones sobre bits en C.

Donde los números que acompañan a los operadores << y >> indican cuantas posiciones se desplaza el operando. La prioridad de los operadores sobre bits es:

- El operador ~ tiene la misma prioridad que los operadores ++ y .--.
- Los operadores << y >> tienen la prioridad situada entre los operadores aritméticos y los operadores relaciónales y lógicos.
- Los operadores &, ^ y | tienen la prioridad situada entre los operadores relaciónales y los operadores lógicos (&& y ||).

3.6 - El operador ?.

El operador ? se usa para reemplazar las sentencias *if/else* (que veremos con posterioridad) de formato general:

```
if (condición)
   expresión;
else
   expresión;
```

Donde expresión debe ser una expresión sencilla y no otra sentencia de C.

El operador ? es un operador ternario cuyo formato general es:

```
Exp1 ? Exp2 : Exp3;
```

Donde *Exp1*, *Exp2* y *Exp3* son expresiones. El operador ? evalúa la expresión *Exp1*, si es cierta se evalúa *Exp2* y si es falsa se evalúa *Exp3*. Veamos algunos ejemplos:

```
int x,y;
y=(x>10) ? 100 : 200;
```

Asignara el valor 100 a *y* si *x* es mayor de 10, y el valor 200 en caso contrario.

```
int t;
(t) ? f1(t)+f2() : printf("t vale cero");
```

Ejecutara las funciones *f1()* y *f2()* si *t* es distinto de cero, y la función *printf* si *t* vale cero.

Tema 4 - Conversión de tipos de datos.

4.1 - Conversión automática de tipos de datos.

El lenguaje C permite que en una misma expresión aparezcan diferentes tipos de datos, encargándose el compilador de realizar las operaciones de forma correcta. El compilador del lenguaje C, cuando en una misma expresión aparecen dos o más tipos de datos, convierte todos los operandos al tipo del operando más grande existente de acuerdo con las dos reglas siguientes:

- Todos los *char* y *short int* se convierten a *int*. Todos los *float* a *double*.

- Para todo par de operandos, lo siguiente ocurre en secuencia:
  - Si uno de los operandos es un *long double*, el otro se convierte en *long double*.
  - Si uno de los operandos es *double*, el otro se convierte a *double*.
  - Si uno de los operandos es *long*, el otro se convierte a *long*.
  - Si uno de los operandos es *unsigned*, el otro se convierte a *unsigned*.

Después de que el compilador aplique estas reglas de conversión, cada par de operandos será del mismo tipo, y el resultado será del tipo de los operandos. Veamos un ejemplo:

```
char ch;
int i;
float f;
double d;

(  ch  /  i  )  +  (  f  *  d  )  -  (  f  +  i  );
  char   int        float double     float  int
```

Debido a que en la operación existen diferentes tipos se aplica la primera conversión:

```
ch de char pasa a int.
f de float pasa a double.

(  ch  /  i  )  +  (  f  *  d  )  -  (  f  +  i  );
  int    int       double double     double  int
```

Al ser los dos operandos de igual tipo, realizamos la primera operación, *(ch / i)*, y el resultado es de tipo *int*. De igual forma, para la segunda operación, *(f * d)*, y el resultado es de tipo *double*.

Para la tercera operación, y dado que las variables no son del mismo tipo, se aplica la segunda regla, convirtiéndose el *int* en *double*, realizándose la suma *(f + i)* como dos datos de tipo *double*, y siendo por tanto el resultado un *double*.

Ahora procedemos a realizar la suma de los dos primeros resultados *(ch / i) + (f * d)*, como uno de ellos es de tipo *int***,** y el otro de tipo *double*, el *int* se convierte en *double* por la segunda regla, y el resultado es un *double*.

Y por último, realizamos la resta final, siendo los dos operandos de tipo *double* y el resultado final, por tanto, de tipo *double*.

4.2 - Conversión forzada de tipos datos.

En C, existe, además, de la conversión automática de tipos de datos, la posibilidad de forzar la conversión de un tipo de datos en otro tipo de datos. Esta conversión de un tipo de datos en otro se llama “casts”, y su sintaxis es:

```
(tipo)expresión
```

Su utilidad queda claramente expresada en el ejemplo siguiente:

```
int a=3,b=2;
float c;
c=a/b;
```

La operación asigna a *c* el valor *1.0* en vez de el valor *1.5*, ello se debe a que al ser *a* y *b* variables de tipo entero, se realiza una división entre enteros, y el resultado de *3/2* es *1*. A continuación ese valor *1* se convierte a un valor en coma flotante para realizar la asignación (valor *1.0*), y se asigna a *c*. Si lo que se desea es que la división se realice en punto flotante, debe escribirse la operación de la siguiente forma:

```
c=(float)a/b;
```

Esta conversión forzada obliga a convertir la variable *a* en *float*, y entonces, aplicando las reglas de conversión automática de tipos, se realiza la división en coma flotante. Este proceso da lugar a que el resultado de la operación sea *1.5*, y dicho valor sea el asignado a la variable *c*.

Tema 5 - Sentencias de control y bucles.

5.1 - Sentencia de control if.

Antes de empezar a explicar las sentencias de control del lenguaje C, conviene explicar los conceptos de verdadero/falso y de sentencia que posee el lenguaje C.

El lenguaje C posee un concepto muy amplio de lo que es verdadero. Para C, cualquier valor que sea distinto de cero es verdadero, siendo por tanto falso solo si el valor cero. Es por ello que una expresión del tipo *if(x)* será verdad siempre que el valor de la variable **x** sea distinto de cero, sea cual sea el tipo de la variable *x*.

El concepto de sentencia en C es igual que el de otros muchos lenguajes. Por sentencia se entiende en C cualquier instrucción simple o bien, cualquier conjunto de instrucciones simples que se encuentren encerradas entre los caracteres { y }, que marcan respectivamente el comienzo y el final de una sentencia.

La forma general de la sentencia *if* es:

```
if (condición
   sentencia;
else
   sentencia;
```

Siendo el *else* opcional. Si la *condición* es verdadera se ejecuta la *sentencia* asociada al *if*, en caso de que sea falsa la *condición* se ejecuta la *sentencia* asociada al *else* (si existe el *else*). Veamos algunos ejemplos de sentencias *if*:

```
int a,b;

if (a>b)
{
   b--;
   a=a+5;
}
else
{
   a++;
   b=b-5;
}
if (b-a!=7)
   b=5;
```

Las sentencias de control *if* pueden ir anidadas. Un *if* anidado es una sentencia *if* que es el objeto de otro *if* o *else*. Esta anidación de *if/else* puede presentar la problemática de decidir que *else* va asociado a cada *if*. Considerese el siguiente ejemplo:

```
if (x)
   if (y) printf("1");
```

```
   else printf("2");
```

¿A que *if* se refiere el *else*?. C soluciona este problema asociando cada *else* al *if* más cercano posible y que todavía no tiene ningún *else* asociado. Es por ello que en este caso el *if* asociado al *else* es el *if(y)*. Si queremos que el *else* este asociado al *if(x)*, deberíamos escribirlo de la siguiente forma:

```
if (x)
{
   if (y)
      printf("1");
}
else
   printf("2");
```

5.2 - Sentencia de control switch.

La forma general de la sentencia *switch* es:

```
switch(variable)
{
   case const1:
      sentencia;
      break;
   case const2:
      sentencia;
      break;
   ...
   default:
      sentencia;
}
```

Donde *variable* debe ser de tipo *char* o *int*, y donde *const1*, *const2*, **...**, indican constantes de C del tipo de datos de la *variable*. Dichas constantes no pueden repetirse dentro del *switch*. El *default* es opcional y puede no aparecer, así como los *break* de los *case*. La sentencia *switch* se ejecuta comparando el valor de la variable con el valor de cada una de las constantes, realizando la comparación desde arriba hacia abajo. En caso de que se encuentre una constante cuyo valor coincida con el valor de la variable, se empieza a ejecutar las sentencias hasta encontrar una sentencia *break*. En caso de que no se encuentre ningún valor que coincida, se ejecuta el *default* (si existe)**.** Veamos algunos ejemplos:

```
int valor;
switch(valor)
{
   case 0: cont++;
           break;
   case 5: cont--;
           break;
   default: cont=-10; /* Se ejecuta si valor no es 0 o 5 */
}

char d;
```

```
int cont=0;
switch(d)
{
   case ‘\r’: cont++; /* Si d es un retorno de carro, se */
                      /* ejecuta este cont++ y el siguiente*/
                      /* al no aparecer un break */
   case ‘\x1B’: cont++;
                break;
   default: cont=-1;
}
```

Las sentencias *switch* pueden aparecer unas dentro de otras, igual que sucedía con las sentencias *if*. Veámoslo con un ejemplo:

```
char d,e;
switch(d)
{
   case ‘a’:
   case ‘A’: switch(e)
             {
                case ‘1’: d=‘z’;
                          e=‘+’;
                          break;
                case ‘2’: d=‘Z’;
                          e=‘-’;
             }
             break;
   case ‘b’:
   case ‘B’: switch(e)
             {
                case ‘0’: d=‘2’;
                default: e=‘+’;
             }
}
```

5.3 - Bucle for.

La sintaxis del bucle *for* es:

```
for(inicialización,condición,incremento) sentencia;
```

En primer lugar, conviene destacar el hecho de la gran flexibilidad del bucle *for* de C. En C, el bucle *for* puede no contener *inicialización*, *condición* o *incremento*, o incluso pueden no existir dos e incluso las tres expresiones del bucle. El bucle *for* se ejecuta siempre que la *condición* sea verdadera, es por ello que puede llegar a no ejecutarse.

Veamos algunos ejemplos de bucles *for*:

```
int i,suma=0;
for(i=1;i<=100;i++)
   suma=suma+i;

int i,j;
```

```
for(i=0,j=100;j>i;i++,j--)
{
   printf("%d\n",j-i);
   printf("%d\n",i-j);
}

float a=3e10;
for(;a>2;a=sqrt(a)) /* sqrt() calcula la raíz cuadrada */
   printf("%f",a);

char d;
for(;getc(stdin)!='\x1B';); /* Bucle que espera hasta que se */
                            /* pulsa la tecla Esc */

char d;
for(;;)
{
   d=getc(stdin);
   printf("%c",d);
   if (d=='\x1B')
      break;
}
```

Como se observa en este último ejemplo, el bucle *for* no posee ninguna expresión. Para salir de el se usa la sentencia *break*, dicha sentencia (ya vista junto con la sentencia de control *switch*) se explicara mas detalladamente con posterioridad.

5.4 - Bucle while.

La sintaxis del bucle *while* es:

```
while (condición) sentencia;
```

Donde la *sentencia* puede no existir (sentencia vacía), pero siempre debe existir la *condición*. El bucle *while* se ejecuta mientras la condición sea verdad. Veamos algunos ejemplos de bucles *while*:

```
int i=1,suma=0;
while (i<=100)
{
   suma=suma+i;
   i++;
}

while (getc(stdin)!='\x1B'); /* Bucle que espera hasta que se */
                             /* pulse la tecla Esc */

while (1) /* Recordar que en C lo que no es cero es verdad */
{
   d=getc(stdin);
   printf("%c",d);
   if (d=='\x1B')
      break;
}
```

5.5 - Bucle do/while.

Al contrario que los bucles *for* y *while* que comprueban la condición en lo alto de la misma, el bucle *do/while* comprueba la condición en la parte baja del mismo, lo cual provoca que el bucle se ejecute como mínimo una vez. La sintaxis del bucle *do/while* es:

```
do
   sentencia;
while (condición);
```

El bucle *do/while* se ejecuta mientras la *condición* sea verdad. Veamos algunos ejemplos de bucle *do/while*:

```
int num;
do
   scanf("%d",&num);
while (num>100);

int i,j;
do
{
   scanf("%d",&i);
   scanf("%d",&j);
}
while (i<j);
```

5.6 - Sentencias de control break y continue.

Las sentencias de control *break* y *continue* permiten modificar y controlar la ejecución de los bucles anteriormente descritos.

La sentencia *break* provoca la salida del bucle en el cual se encuentra y la ejecución de la sentencia que se encuentra a continuación del bucle.

La sentencia *continue* provoca que el programa vaya directamente a comprobar la condición del bucle en los bucles *while* y *do/while*, o bien, que ejecute el incremento y después compruebe la condición en el caso del bucle *for*.

Veamos algunos ejemplos de uso de *break* y de *continue*:

```
int x;
for(x=0;x<10;x++)
{
   for(;;)
      if (getc(stdin)=='\x1B')
         break;
   printf("Salí del bucle infinito, el valor de x es: %d\n",x);
}
int x;
for(x=1;x<=100;x++) /* Esta rutina imprime en pantalla los */
{                   /*números pares */
```

```
   if (x%2)
      continue;
   printf(“%d\n”,x);
}
```

Tema 6 - Arrays, cadenas y punteros.

6.1 – Arrays y cadenas.

En C, un array unidimensional se declara como:

```
tipo nombre[tamaño];
```

En C, el primer elemento de un array es el que posee el índice 0, por lo tanto, un array de 20 elementos posee sus elementos numerados de 0 a 19. Veamos unos ejemplos de declaración y manejo de algunos arrays:

```
int x[100],i;
for(i=0;i<100;i++)
   x[i]=i;

char letras[256];
int i;
for(i=0;i<256;i++)
   letras[i]=i;

int x[10],i,suma;
for(i=0;i<10;i++)
{
   printf("Introducir un número: %d: ",i);
   scanf("%d",&x[i]);
}
for(suma=0,i=0;i<10;i++)
   suma=suma+x[i];
printf("La suma es: ",suma);
```

Sin embargo, el lenguaje C no comprueba el tamaño de los arrays, por lo cual, es posible construir una rutina como la siguiente, la cual ocasionara un incorrecto funcionamiento del programa:

```
float a[10];
int i;
for(i=0;i<100;i++) /* Este bucle es incorrecto */
   a[i]=i;
```

Es por ello, que es misión del programador comprobar que no se produzca el desbordamiento de los arrays.

Una cadena, también llamada string, es un tipo especial de array unidimensional. Una cadena es un array de caracteres (*char*) que termina con un carácter especial (el carácter *'\0'*). Es por ello, que la declaración de una cadena de caracteres se realiza exactamente igual que la declaración de un array unidimensional de caracteres:

```
char cadena[tamaño];
```

Como toda cadena debe terminar en el carácter *'\0'*, es por ello que si se quiere usar una cadena de 20 caracteres, debe declararse de tamaño 21 (20 caracteres + carácter terminador).

Por lo demás, puede usarse una cadena como si fuera un array unidimensional, pues se puede referenciar uno cualquiera de sus elementos, etc. Para manejar las cadenas, existen un gran número de funciones de biblioteca que proporciona el standard ANSI-C, para más información referirse al apendice A o a cualquier libro de C.

La declaración de arrays de más de una dimensión se realiza de forma parecida a la de una dimensión, la sintaxis de la declaración de un array multidimensional es:

```
tipo nombre[tam1][tam2]...[tamN];
```

Y su indexación, etc., se realiza de forma similar al array unidimensional. Veamos un ejemplo:

```
float matriz[2][3];
int i,j;
for(i=0;i<2;i++)
   for(j=0;j<3;j++)
   {
      printf("M[%d][%d]: ",i,j);
      scanf("%f",&matriz[i][j]);
   }
```

Además, es posible inicializar los arrays en el momento de declararlos. Su sintaxis es:

```
tipo nombre[tam1][tam2]...[tamN]={lista_de_valores};
```

Por lo cual, podemos escribir:

```
float vector[3]={-3.0,5.7,-7.5};
```

También es posible inicializar arrays sin ponerles el tamaño, el compilador cuenta el número de caracteres de inicialización y reserva el tamaño necesario de forma automática. Por ejemplo:

```
float vector[]={-3.0,5.7,-7.5};
char cadena[]="Esto es una cadena";
```

6.2 – Punteros.

Los punteros son una de las poderosas herramientas que ofrece el lenguaje C a los programadores, sin embargo, son también una de las más peligrosas, el uso de punteros sin inicializar, etc., es una fuente frecuente de errores en los programas de C, y además, suele producir fallos muy difíciles de localizar y depurar.

Un puntero es una variable que contiene una dirección de memoria. Normalmente esa dirección es una posición de memoria de otra variable, por lo cual se suele decir que el puntero "apunta" a la otra variable.

La sintaxis de la declaración de una variable puntero es:

```
tipo *nombre;
```

El tipo base de la declaración sirve para conocer el tipo de datos al que pertenece la variable a la cual apunta la variable de tipo puntero. Esto es fundamental para poder leer el valor que almacena la zona de memoria apuntada por la variable puntero y para poder realizar ciertas operaciones aritméticas sobre los mismos.

Algunos ejemplos de declaración de variables puntero son:

```
int *a;
char *p;
float *f;
```

Existen dos operadores especiales de los punteros, el operador * y el operador &.

El operador & es un monario que devuelve la dirección de una variable de memoria. Así, si declaramos:

```
int *a,b;
```

Y hacemos:

```
a=&b;
```

La variable puntero *a* contendrá la dirección de memoria de la variable *b*.

El operador * es un operador monario que devuelve el valor de la variable situada en la dirección que sigue. Veámoslo con un ejemplo:

```
int *a,b,c;
```

Si hacemos:

```
b=15;
a=&b;
c=*a;
```

Entonces la variable *c* contendrá el valor *15*, pues **a* devuelve el valor de la dirección que sigue (a la que "apunta" la variable puntero), y con anterioridad hemos hecho que *a* contenga la dirección de memoria de la variable *b* usando para ello el operador &.

Con las variables de tipo puntero es posible realizar algunas operaciones:

- Asignación de punteros. Es posible asignar el valor de una variable de tipo puntero a otra variable de tipo puntero. Por ejemplo:

```
int *a,*b,c;
a=&c;
b=a;
```

Entonces *b* contiene el valor de *a*, y por ello, *b* también "apunta" a la variable *c*.

- Aritmética de punteros. Sobre las variables de tipo puntero es posible utilizar los operadores +, -, ++ y --. Estos operadores incrementan o decrementan la posición de memoria a la que "apunta" la variable puntero. El incremento o decremento se realiza de acuerdo al tipo base de la variable de tipo puntero, de ahí la importancia del tipo del que se declara la variable puntero. Veamos esto con la siguiente tabla:

		Operación	++	--	+9	-5
Variable	**Dirección actual**	**Nueva dirección**				
int *a;	3000		3002	2998	3018	2990
float *b	3000		3004	2996	3036	2980

Tabla 6.2.1: Ejemplos de aritmética de punteros en C.

Por lo tanto, si tenemos:

```
tipo *a;
```

Y hacemos:

```
a=a+num;
```

La posición a la que apunta *a* se incrementa en *num*sizeof(tipo)*. Para la resta se decrementa de igual forma en *num*sizeof(tipo)*. Los operadores ++ y -- son equivalentes a realizar *num=1*, y con ello quedan obviamente explicados.

- Comparaciones de punteros. Sobre las variables de tipo puntero es posible realizar operaciones de comparación entre ellas. Veamos un ejemplo:

```
int *a,*b;
if (a<b)
   printf("a apunta a una dirección más baja que b");
```

Existe una estrecha relación entre los punteros y los arrays. Consideremos el siguiente fragmento de código:

```
char str[80],*p;
p=str;
```

Este fragmento de código pone en la variable puntero p la dirección del primer elemento del array *str*. Entonces, es posible acceder al valor de la quinta posición del array mediante *str[4]* y *(p+4)* (recuérdese que los índices de los arrays empiezan en 0). Esta estrecha relación entre los arrays y los punteros queda más evidente si se tiene

en cuenta que el nombre del array sin índice es la dirección de comienzo del array, y, si además, se tiene en cuenta que un puntero puede indexarse como un array unidimensional, por lo cual, en el ejemplo anterior, podríamos referenciar ese elemento como *p[4]*.

Es posible obtener la dirección de un elemento cualquiera del array de la siguiente forma:

```
int str[80],*p;
p=&str[4];
```

Entonces, el puntero *p* contiene la dirección del quinto elemento del array *str*.

Hasta ahora hemos declarado variables puntero aisladas. Es posible, como con cualquier otro tipo de datos, definir un array de variables puntero. La declaración para un array de punteros *int* de tamaño *10* es:

```
int *a[10];
```

Para asignar una dirección de una variable entera, llamada *var*, al tercer elemento del array de punteros, se escribe:

```
x[2]=&var;
```

Y para encontrar el valor de *var*:

```
*x[2];
```

Dado, además, que un puntero es también una variable, es posible definir un puntero a un puntero. Supongamos que tenemos lo siguiente:

```
int a,*b,**c;
b=&a;
c=&b;
```

Y entonces, ***c* tiene el valor de la variable *a*, pues *c* es un puntero a una variable que ya es de tipo puntero.

Este concepto de puntero a puntero podría extenderse a puntero a puntero a puntero, etc., pero no nos ocuparemos de ello. Además, existe el concepto de puntero a una función, al cual nos referiremos en el tema dedicado a las funciones.

Tema 7 - Funciones.

El formato general de una función de C es:

```
tipo nombre(lista de parámetros)
{
   cuerpo de la función
}
```

Las funciones son simulares a las de cualquier otro lenguaje, pero, tal y como citamos en la introducción, al no ser un lenguaje estructurado por bloques, no es posible declarar funciones dentro de otras funciones.

7.1 - La sentencia return.

Antes de empezar la explicación de las funciones, conviene explicar la sentencia *return*. La sentencia *return* permite, en primer lugar, salir de la función desde cualquier punto de la misma, y en segundo lugar, devolver un valor del tipo de la función, si ello es necesario (no se devuelve ningún valor si la función es de tipo *void*). Veamos un ejemplo:

```
int Comparacion(int a,int b)
{
   if (a>b) return 1;  /* a es mayor que b */
   if (a<b) return -1; /* a es menor que b */
   return 0;           /* a y b son iguales */
}
```

Como se observa en el ejemplo, una función puede contener más de una sentencia *return*. Ello permite, la posibilidad de poder salir de la función desde distintos puntos de la misma. Un aspecto que conviene resaltar es el hecho de que una función también termina su ejecución si llega al final de la misma sin encontrar ninguna sentencia *return*. Ello es posible en toda función de tipo *void*. Veamos un ejemplo:

```
void A(int *a)
{
   *a=5;
}
```

Esa función es equivalente a otra que tuviera como última línea una sentencia *return*, y funcionaría de igual forma.

7.2 - Argumentos de las funciones, llamada por valor y por "referencia".

Una vez conocido el uso de la función *return*, podemos introducirnos en la explicación de las funciones. En primer lugar, si una función usa argumentos, es

necesario declarar variables que acepten los argumentos de la función. Veamos un ejemplo:

```
int EstaEn(char *cad,char c) /* Devuelve 1 si el carácter c */
{                            /* esta en el string cad */
   while (*cad!='\0')
   {
      if (*cad==c)
         return 1;
      cad++;
   }
   return 0;
}
```

Esta función, podría ser llamada desde otra función de la siguiente forma:

```
char cadena[]="Esta es una cadena de prueba";
if (EstaEn(cadena,'a'))
   printf("Esta");
else
   printf("No esta");
```

A diferencia de otros lenguaje, el lenguaje C, solo permite pasar parámetros a las funciones por valor. Si se desea que los cambios efectuados en una función sobre una variable afecten fuera del alcance de la función, es posible simular un paso por referencia mediante el uso de punteros. En efecto, si a una función le pasamos como argumento la dirección de una variable, cualquier modificación que se realice en esa dirección, afectara, lógicamente, al valor que tiene la variable original, y con ello, conseguiremos el mismo efecto que un paso por referencia. Veámoslo con un ejemplo:

```
#include <stdio.h>
void Alfa(int *val,float pos)
{
   *val=5;
   pos=7.7;
   return;
}
void Beta(int val,float *pos)
{
   val=10;
   *pos=14.7;
}

int main(void)
{
   int a=6;
   float b=9.87;
   printf("Al principio valen a=%d b=%f\n",a,b);
   Alfa(&a,b);
   printf("Después de Alfa valen a=%d b=%f\n",a,b);
   Beta(a,&b);
   printf("Después de Beta valen a=%d b=%f\n",a,b);
}
```

Este programa mostrara en pantalla:

```
Al principio valen a=6 b=9.87
Después de Alfa  valen a=5 b=9.87
Después de Beta valen a=5 b=14.7
```

Ello es, pues a *Alfa* se le pasa la variable *a* por "referencia" (se le pasa *&a*, o sea, un puntero a la variable *a*), y la variable *b* por valor, mientras que en *Beta* sucede al revés.

7.3 - Arrays como argumentos de funciones.

Un aspecto a tener muy en cuenta es que C no permite el paso de un array por valor a una función, un array es siempre pasado por "referencia", pues en la llamada, lo que se pasa es la dirección del primer elemento del array (recuérdese que el nombre de un array es un puntero al primer elemento). Por valor tan solo es posible pasar por valor elementos individuales del array, pero no el array completo. Veámoslo en un ejemplo:

```
#include <stdio.h>
void PasoValorReferencia(int *array,int valor)
{
   array[5]=-8.6;
   valor=4;
}

int main(void)
{
   int array[10]={0,0,0,0,0,0,0,0,0,0};
   PasoValorReferencia(array,array[3]);
   printf("Array[5] vale: %d y array[3] vale:%d\n",array[5],array[3]);
   return 0;
}
```

Colocara en pantalla en el mensaje:

```
Array[5] vale: 8.6 y array[3] vale: 0
```

7.4 - Argumentos de la función main.

La función *main()*, como toda función de C, acepta argumentos. Los argumentos que acepta la función *main()* son un entero (*int argc*), un array de punteros a strings (*char *argv[]*), y otro array de punteros a strings (*char *env[]*). Aunque los nombres de dichos argumentos no tienen porque ser *argc*, *argv*, y *env*, en toda la literatura de C se usan dichos nombres, y aquí los respetaremos. El significado de los parámetros *argc*, *argv* y *env* es el siguiente:

- El parámetro *argc* contiene el número de argumentos en la línea de ordenes de la llamada al programa.

- El parámetro *argv* contiene un puntero a cada uno de los argumentos (strings) de la línea de ordenes de la llamada al programa.

- El parámetro *env* contiene un puntero a cada una de las variables de ambiente (strings) del sistema operativo.

Veamos un ejemplo de programa que use argumentos en la línea de ordenes:

```
#include <stdio.h>
int main(int argc,char *argv[],char *env[])
{
   int i;

   printf("El valor de argc es: %d\n",argc);
   for(i=0;i<argc;i++)
      printf("El argumento %d es: %s\n",i,argv[i]);
   for(i=0;env[i]!=NULL;i++)
      printf("La variable de ambiente %d es: %s\n",i,env[i]);
   return 0;
}
```

Supongamos que el programa lo hemos llamado *prueba.exe*, entonces, llamando al programa con la siguiente línea:

```
prueba.exe   Este_es_el_argumento_1   Este_es_el_argumento_2
```

Escribirá en pantalla:

```
El valor de argc es: 3
El argumento 0 es: prueba.exe
El argumento 1 es: Este_es_el_argumento_1
El argumento 2 es: Este_es_el_argumento_2
La variable de ambiente 0 es: COMSPEC=C:\DOS\COMMAND.COM
La variable de ambiente 1 es: TEMP=C:\WINDOWS\TEMP
La variable de ambiente 2 es: PROMPT=$P$G
```

Como se puede observar, existen 3 argumentos, numerados de 0 a 2, siendo el argumento 0, siempre, el nombre del programa, y siendo el resto de argumentos los argumentos del programa. El número y valor de las variables de ambiente depende, tanto de que sistema operativo se trate (MS-DOS, UNIX, etc.), como de la configuración, etc., del procesador de comandos de dicho sistema operativo.

7.5 - Recursividad.

Una función de C puede llamarse a si misma. Este proceso recibe el nombre de recursividad. Los ejemplos de recursividad abundan, siendo uno de los mas habituales la función factorial:

```
unsigned Factorial(unsigned num)
{
   if (num==0) return 1;

   return num*Factorial(num-1);
}
```

La recursividad es una poderosa herramienta de programación, sin embargo, presenta dos problemas:

- La velocidad de ejecución de un algoritmo programado de forma recursiva es mucho mas lento que el programado de forma iterativa.

- La recursividad, si es excesiva, puede ocasionar el desbordamiento de la pila, y con ello, el fallo en la ejecución del programa.

Sin embargo, el uso de la recursividad es frecuente en campos como la inteligencia artificial, etc., y en la implementación de ciertos algoritmos tales como el algoritmo de ordenación "QuickSort", muy difícil de implementar de forma iterativa, pero relativamente sencillo de forma recursiva.

7.6 - Punteros a funciones.

Al igual que cualquier otro tipo de dato, una función ocupa una dirección de memoria, y por tanto, puede ser apuntada por un puntero. La declaración de un puntero a una función es:

```
tipo de dato (*nombre de la variable)(prototipo);
```

Veamos algunos ejemplos:

```
int (*a)(int,float);
void (*b)(void);
```

Generalmente, los punteros a funciones se usan en la programación de bajo nivel, tal como modificación de interrupciones, creación de controladores de dispositivos, etc.

7.7 - El modificador de almacenamiento static aplicado a funciones.

Al igual que en el caso de las variables globales, es posible aplicar delante de una función el modificador de almacenamiento *static*. Dicho modificador hace que la función sobre la que se aplica sea local al módulo donde se encuentra, y no pueda ser conocida por los restantes módulos del programa, de igual forma a como sucedía con las variables globales. Esta modificación del alcance de una función permite realizar un mejor encapsulado del código y simplificar la programación en proyectos de gran envergadura.

Tema 8 - Estructuras, campos de bit, uniones y enumeraciones.

8.1 - Estructuras.

Una estructura es un conjunto de variables que se referencian bajo el mismo nombre. La sintaxis de la declaración de una estructura en lenguaje C es:

```
struct nombre_estructura{
   tipo nombre_variable;
   tipo nombre_variable;
   ...
   tipo nombre_variable;
}variables_estructura;
```

Es posible no poner el nombre de la estructura (*nombre_estructura*), o bien, no crear en el momento de declarar la estructura ninguna variable de la estructura (*variables_estructura*), pero no es posible eliminar la aparición de ambos elementos. Veamos algunos ejemplos de declaración de estructuras:

```
struct LISTA{
   int tam;
   char cadena[50];
}var_lista;

struct DATO{
   int tam;
   float vector[3];
   struct DATO *siguiente;
};

struct {
   float a,b;
   unsigned long i,j;
   char cadena[5];
}memo[10];

struct ALFA{
   int a;
   float b;
};
struct BETA{
   struct ALFA alfa;
   float c,d;
}variable;
```

Para referenciar un elemento de una estructura se realiza de la siguiente forma:

```
variables_estructura.nombre_variable;
```

Así, podíamos referenciar los elementos de las estructuras anteriores de la siguiente forma:

```
var_lista.tam;
var_list.cadena;
var_list.cadena[7];
memo[2].a;
memo[6].cadena[3];
variable.alfa.a;
variable.c;
```

Un aspecto que es necesario aclarar es el paso de estructuras como parámetros a las funciones. A una función es posible pasarle un elemento de los que componen la estructura, una estructura entera, e incluso, un array de estructuras. En caso de pasarle un elemento de la estructura, el paso se hace siguiendo las reglas del tipo del cual sea ese elemento; en el caso de una estructura entera, C la pasa, a no ser que se le indique lo contrario, por valor, y en el caso de un array de estructuras, como todo array, lo pasara por "referencia". Conviene aclarar en este momento que si la estructura posee en su interior un array de elementos, la estructura puede ser pasada por valor a una función, pero el array será pasado siempre por referencia. En concreto a la función se le pasara por valor un puntero al primer elemento del array. Veamos todo esto en un ejemplo:

```
struct ALFA{
   int a;
   char b[20];
};

void PasoDeElementos(int val,char *cadena)
{
   val=15;
   cadena[7]='a';
}

void PasoDeLaEstructuraPorValor(struct ALFA a)
{
   a.val=14;
   a.cadena[6]='b';
}

void PasoDeLaEstructuraPorReferencia(struct ALFA *a)
{
   *(a.val)=13;
   *(a.cadena)[5]='c';
}

void PasoDeUnArrayDeEstructuras(struct ALFA *a)
{
   a[4].val=12;
   a[5].cadena[4]='d';
}

int main(void)
{
   struct ALFA a,b[10];
```

```
   PasoDeElementos(a.val,a.b);
   PasoDeLaEstructuraPorValor(a);
   PasoDeLaEstructuraPorReferencia(&a);
   PasoDeUnArrayDeEstructuras(b);
   return 0;
}
```

En el paso de una estructura por referencia, se ha usado una construcción **(variable_estructura.nombre_variable)*, esta construcción asigna el valor que se desea a esa variable de la estructura, pues *variable_estructura.nombre_variable* es un puntero a la variable. El uso de los paréntesis es necesario, pues el operador . tiene menor prioridad que el operador *. Es por ello que C introduce un nuevo operador, el operador ->. Este operador es equivalente al otro, pero más cómodo y fácil de escribir y de usar. Entonces, podríamos haber escrito la función de paso de una estructura por referencia de la forma siguiente:

```
void PasoDeLaEstructuraPorReferencia(struct ALFA *a)
{
   a->val=13;
   a->cadena[5]='c';
}
```

8.2 - Campos de bit.

Un campo de bit es un método predefinido por C para poder acceder a un bit de un byte. Aunque dicho acceso siempre es posible mediante operaciones el uso de los operadores sobre bits, explicados con anterioridad, el uso de campos de bit puede añadir claridad al programa.

El método de declaración de un campo de bit se basa en la estructura, pues un campo de bit no es mas que un tipo especial de estructura. El formato de declaración de un campo de bit es:

```
struct nombre_campo_bit{
   tipo nombre1 : longitud;
   tipo nombre2 : longitud;
   ...
   tipo nombreN : longitud;
}variables_campo_bit;
```

El *tipo* de un campo de bit debe declararse como *unsigned int* o *signed int*. Veamos un ejemplo de declaración de un campo de bit:

```
struct ALFA{
   unsigned a : 1;
   signed b : 2;
   unsigned : 4;
   unsigned c : 1;
}campo;
```

En dicho ejemplo, se declara un campo de bit de tamaño 4 al cual no se le da nombre, eso es valido, y su efecto es que esos cuatro bits no podrían ser referenciados.

Es posible mezclar en la declaración elementos normales de estructura con elementos de campo de bit. Veamos un ejemplo:

```
struct EMP{
   char nombre[20],apellido[2][20];
   float sueldo;
   unsigned vacaciones:1;
   unsigned enfermo:1;
};
```

8.3 - Uniones.

En C una unión es una posición de memoria que se usa por varias variables similares, que pueden ser de tipos diferentes. La definición de unión es:

```
union nombre_union{
   tipo nombre1;
   tipo nombre2;
...
   tipo nombreN;
}var_union;
```

Como puede observarse su declaración, etc., es parecida a la declaración de una estructura. Sin embargo, en una unión, todos los tipos de datos comparten la misma dirección de memoria. Así, si declaramos:

```
union ALFA{
   int a;
   char b;
}alfa;
```

Tendremos:

```
<------alfa.a------>
   Byte0    Byte1
<-alfa.b->
```

Por lo tanto, *b* tendrá en común con *a* el byte más bajo.

Un ejemplo mas útil de una unión es el siguiente:

```
union BETA{
   unsigned short a;
   char b[2];
};beta
```

Entonces *beta.b[0]* contendrá el byte bajo de *beta.a*, y *beta.b[1]* contendrá el byte alto de *beta.a*. Ello permite acceder a la parte alta o baja de dicho *unsigned short* sin necesidad de usar operadores sobre bits.

8.4 - Enumeraciones.

Una enumeración es un conjunto de constantes enteras con nombre y especifica todos los valores legales que pueden tener unas variables. Las enumeraciones se declaran de la siguiente forma:

```
enum nombre_enum{lista_de_enumeración} lista_de_variables;
```

Donde, al igual que en las estructuras, puede no aparecer *nombre_enum* o *lista_de_variables*. Veamos un ejemplo de enumeración:

```
enum MONEDAS{ peseta, duro, diez, cinco, cincuenta, cien, doscientas,
quinientas} monedas_espana;
```

Las enumeraciones asignan una constante entera a cada uno de los símbolos de la enumeración, empezando por el valor *0*. Esto puede modificarse colocando en la declaración el valor que deseamos tengan los elementos a partir de uno dado. Esto se realiza de la siguiente forma:

```
enum CURSO{ primero, segundo, tercero, cuarto_t=100, quinto_t,
cuarto_e=200, quinto_e};
```

En este caso, las constantes *primero*, *segundo* y *tercero* tienen los valores *0,1* y *2*, las constantes cuarto_t y quinto_t los valores *100* y 101, y las constantes *cuarto_e* y *quinto_e* los valores *200* y *201* respectivamente.

8.5 - La palabra reservada typedef.

El lenguaje C permite mediante el uso de la palabra reservada *typedef* definir nuevos nombres para los tipos de datos existentes, esto no debe confundirse con la creación de un nuevo tipo de datos. La palabra clave *typedef* permite solo asignarle un nuevo nombre a un tipo de datos ya existente. La sintaxis general de uso de *typedef* es:

```
typedef tipo nombre;
```

Donde *tipo* es cualquier tipo de datos permitido, y *nombre* es el nuevo nombre que se desea tenga ese tipo. Veamos algunos ejemplos:

```
typedef int entero;

typedef struct{
   unsigned codigo;
   char nombre[40];
   char apellido[40];
}cliente;
```

Y entonces podrían crearse nuevas variables de la forma:

```
entero a;
cliente b,*c;
```

Tema 9 - El preprocesador.

En un programa escrito en C, es posible incluir diversas instrucciones para el compilador dentro del código fuente del programa. Estas instrucciones dadas al compilador son llamadas directivas del preprocesador y, aunque realmente no son parte del lenguaje C, expanden el ámbito del entorno de programación de C.
El preprocesador, definido por el standard ANSI de C, contiene las siguientes directivas:

`#if`	`#ifdef`	`#ifndef`	`#else`
`#elif`	`#endif`	`#include`	`#define`
`#undef`	`#line`	`#error`	`#pragma`

Tabla 9.1: Directivas del preprocesador en C.

9.1 - Directiva #define.

La directiva define se usa para definir un identificador y una cadena que el compilador sustituirá por el identificador cada vez que se encuentre en el archivo fuente. El standard ANSI llama al identificador "nombre de macro" y al proceso de sustitución "sustitución de macro". Por ejemplo:

```
#define TRUE 1
#define FALSE 0
```

El compilador, cada vez que vea el identificador *TRUE*, lo sustituirá por el valor *1*, e igual con *FALSE*. El uso mas común de la directiva *#define* es la definición de valores constantes en el programa, tamaños de arrays, etc.

Una característica que posee la directiva *#define* es que el "nombre de macro" puede contener argumentos. Cada vez que el compilador encuentra el "nombre de macro", los argumentos reales encontrados en el programa reemplazan los argumentos asociados con el nombre de macro. Veamos un ejemplo:

```
#define MIN(a,b) (a<b) ? a : b
```

Si tenemos ahora en el programa:

```
printf("El valor mínimo es: %d\n",MIN(10,20));
```

El compilador sustituye el "nombre de macro" y sus argumentos en tiempo de compilación, y ello equivale a haber escrito el código:

```
printf("El valor mínimo es: %d\n",(10<20) ? 10 : 20);
```

9.2 - Directiva #undef.

La directiva *#undef* permite quitar una definición de "nombre de macro" que se realizo con anterioridad. Veamos un ejemplo:

```
#define TAM 10
.......
#undef TAM
```

A partir de *#undef TAM*, el "nombre de macro" *TAM* deja de existir, ello permite localizar los "nombre de macro" donde sea necesario.

9.3 - Directiva #error.

La directiva *#error* fuerza a parar la compilación del programa, a la vez que muestra un mensaje de error. El mensaje de error no aparecerá entre comillas dobles. Veamos un ejemplo:

```
#error Detenida compilación
```

Su principal uso viene asociado a detener la compilación en ciertas condiciones en asociación con las directivas *#if*, etc., explicadas con posterioridad.

9.4 - Directiva #include.

La directiva *#include* fuerza al compilador a incluir otro archivo fuente en el archivo que tiene la directiva *#include*, y a compilarlo. El nombre del archivo fuente a incluir se colocara entre comillas dobles o entre paréntesis de ángulo. Por ejemplo:

```
#include <stdio.h>
#include "stdio.h"
```

Los archivos incluidos mediante *#include* pueden a su vez poseer otras directivas *#include*. La diferencia existente entre encerrar el archivo entre paréntesis de ángulo o entre comillas dobles, es que, en el primer caso, se busca el archivo en los directorios de la linea de ordenes de compilación, y, después en los directorios standard de C, pero nunca en el directorio de trabajo; y en el segundo caso el primer sitio donde se busca el archivo a incluir es en el directorio actual de trabajo, pasándose, caso de no haber sido encontrado, a buscar en los mismos sitios que el caso anterior.

9.5 - Directivas #if, #ifdef, #ifndef, #else, #elif y #endif.

Las directivas *#if*, *#ifdef*, *#ifndef*, *#else*, *#elif* y *#endif* son directivas condicionales de compilación. Estas directivas permiten decirle al compilador que partes del programa debe compilar bajo distintas condiciones.

La idea general de la directiva *#if* es que si es verdad la expresión que se encuentra después del *#if*, se compilara el código que figura entre el *#if* y el *#endif* se compilara. La directiva *#else* funciona de igual forma que el *else* del lenguaje C. La directiva *#elif* funciona como el escalonado de *if* del lenguaje C. La definición formal es:

```
#if expresión1
   secuencia de sentencias
#elif expresión2
   secuencia de sentencias
......
#else
   secuencia de sentencias
#endif
```

Veamos algunos ejemplos:

```
#define MEM 200

#if MEM>100
   printf("MEM es mayor de 100");
#endif

#define VALOR 0

#if VALOR==0
   c=a*b/(VALOR+1);
#else
   c=a*b/VALOR;
#endif
```

Compilara el código para el caso de *VALOR==0*.

```
#define VALOR 15

#if VALOR<0
   b=b/(-VALOR);
#elif VALOR==0
   b=b/(VALOR+1);
#else
   b=b/VALOR;
#endif
```

Compilara el código para el último caso.

La directivas *#ifdef* y *#ifndef* se usan también para compilación condicional, solo que no evalúan expresión alguna, solo comprueba si esta definido (*#ifdef*) o si no esta definido (*#ifndef*) algún nombre de macro. Su sintaxis general es:

```
#ifdef nombre de macro
   secuencia de sentencias
#else
   secuencia de sentencias
#endif
```

E igual para *#ifndef*. Veamos algunos ejemplos:

```
#define VAL 10

#ifdef VAL
   printf("VAL definido");
#else
   printf("VAL no definido");
#endif

#ifndef NOVAL
   printf("NOVAL no definido");
#endif
```

Compilara el código para el caso de *VAL* definido y, además, compilara el código de *NOVAL*, al no estar definida dicha macro. Como se observa no se comprueba el valor de *VAL*, o el de *NOVAL*, solo se comprueba si están definidos o no.

9.6 - Directiva #line.

La directiva *#line* permite cambiar la cuenta de lineas del compilador y el nombre del archivo. Su sintaxis es:

```
#line número ["nombre de archivo"]
```

Veamos un ejemplo:

```
#line 100 /* Inicializa el contador de lineas a 100 */
      /* 10 lineas */
#error Detenida compilación
```

Indicara el mensaje de error en la linea *110* del programa, y no en la que suceda realmente.

9.7 - Directiva #pragma.

La directiva *#pragma* es una directiva que permite dar instrucciones al compilador sobre como debe realizar la compilación del código fuente. Su sintaxis es:

```
#pragma nombre
```

Tema 10 - Entrada y salida.

Antes de empezar a explicar la entrada y salida en C, es necesario realizar dos pequeños comentarios:

En primer lugar, para el correcto funcionamiento de la entrada y salida en C, y dado que las funciones de E/S, estructuras de datos usadas por esas funciones, etc., se encuentran declaradas en el archivo de cabecera <*stdio.h*>, es necesario incluir dicho archivo, mediante la directiva del preprocesador *#include*, para que la E/S funcione correctamente, pues en caso contrario, puede funcionar de forma incorrecta, e incluso, puede llegar a dar errores de compilación.

En segundo lugar, aparte de la E/S por consola y la E/S de fichero mediante buffer intermedio, que serán explicadas en este tema, existe una E/S de fichero sin buffer intermedio, proveniente de la primitiva implementación de C en máquinas UNIX., y que el standard ANSI de C no ha estandarizado, por lo cual, no es recomendable su uso. Por este motivo, y dada su similitud en la mayoría de apartados con el sistema de E/S de fichero mediante buffer intermedio, no será explicado en el presente tema.

10.1 - Entrada y salida desde consola.

La entrada y salida desde consola se refiere a las operaciones que se producen en el teclado y la pantalla del ordenador. Dichos dispositivos son automáticamente abiertos y cerrados al comenzar y terminar el programa, por lo cual, no deben ser abiertos ni cerrados por el propio programa. Existen, básicamente, seis funciones de entrada y salida desde consola, tres de entrada y tres de salida. Veámoslas:

La función *getchar()*, lee un carácter desde el teclado. Dicha función se define como:

```
int getchar(void);
```

Dicha función lee caracteres, de uno en uno, desde el teclado, esperando, para leer los caracteres, la pulsación de un retorno de carro. Es por ello que es posible escribir varios caracteres antes de que se ninguno de ellos sea leído. La función *getchar()* hace eco en pantalla del carácter leído. En caso de error devuelve *EOF*.

La función *putchar()* escribe un carácter a la pantalla del ordenador. Dicha función se define como:

```
int putchar(int c);
```

La función *putchar()*, si sucede de forma correcta, devuelve el carácter escrito. En caso de error devuelve el carácter *EOF*.

Veamos un ejemplo de uso de *getchar()* y *putchar()*:

```
#include <stdio.h>

int main(void)
{
   char ch;

   do
   {
      ch=getchar();
      putchar(ch);
   }
   while (ch!='e' && ch!='E');

   return 0;
}
```

Este programa lee todas las teclas pulsadas en el teclado, y las coloca en pantalla, hasta leer una *'e'* o una *'E'*. Obsérvese que solo lee las teclas después de pulsar un retorno de carro.

La función *gets()* lee un string desde el teclado. La función se define como:

```
char *gets(char *s);
```

La función *gets()* lee un string desde el teclado hasta que se pulsa un retorno de carro. El string es almacenado en la variable *s*, y el retorno de carro leído desde el teclado es, automáticamente, reemplazado por un carácter de final de string (*'\0'*). Devuelve un puntero a la variable *s* si sucede de forma correcta, y *NULL* en caso contrario. La función *gets()* permite corregir errores de teclado usando la tecla de retroceso antes de pulsar el retorno de carro.

La función *puts()* escribe un string en pantalla. La función se define como:

```
int puts(const char *s);
```

La función *puts()* escribe en pantalla el string almacenado en *s*, y añade al final un retorno de carro. Devuelve un entero no negativo si sucede de forma correcta, y *EOF* en caso de error.

Veamos un ejemplo de uso de *gets()* y *puts()*:

```
#include <stdio.h>

#define TAM 100

int main(void)
{
   char cadena[TAM];

   puts("Introduce una cadena:");
   gets(cadena);

   return 0;
}
```

La función *scanf()* se usa para leer cualquier tipo de dato predefinido desde el teclado, y convertirlo, de forma automática, al formato interno adecuado. La función se define como:

```
int scanf(const char *formato[,dirección,...]);
```

El string *formato* es la cadena de control que indica los datos a leer. Dicha cadena de control consta de tres clases de caracteres:

- Especificadores de formato.
- Caracteres de espacio en blanco.
- Caracteres que no sean espacios en blanco.

Los especificadores de formato están precedidos por el signo *%*, y dicen a la función que tipo de datos van a ser leídos a continuación. Los especificadores de formato validos son:

Especificador	**Descripción.**
%c	Leer un único carácter.
%d	Leer un entero decimal.
%i	Leer un entero decimal.
%e	Leer un número en punto flotante.
%f	Leer un número en punto flotante.
%g	Leer un número en punto flotante.
%o	Leer un número octal.
%s	Leer una cadena de caracteres.
%x	Leer un número hexadecimal.
%p	Leer un puntero.
%n	Recibe un valor igual al número de carácter leídos.
%u	Leer un entero sin signo.

Tabla 10.1.1: Especificadores de formato de la función scanf().

Además, es posible utilizar los modificadores *h* (*short*), *l* (*long*) y *L*. El modificador *h* se puede aplicar a los tipo *d*, *i*, *o*, *u* y *x*, e indica que el tipo de dato es *short int* o *unsigned short int* según el caso. El modificador *l* se puede aplicar a los casos anteriores, indicando que el tipo de dato es *long int* o *unsigned long int*, pero, además, se puede aplicar a los tipos *e*, *f* y *g*, indicando, en tal caso, que el tipo de dato es *double*. Por último, el modificador *L* se puede aplicar a los tipos *e*, *f* y *g*, e indica que el tipo de dato es *long double*.

Los caracteres de espacio en blanco en la cadena de control dan lugar a que *scanf()* lea y salte sobre cualquier número (incluido cero) de espacios en blanco. Un espacio en blanco es, además del carácter espacio, un tabulador o un salto de línea.

Un carácter que no sea espacio en blanco da lugar a que *scanf()* lea y elimine el carácter asociado. Por ejemplo, *%d:%d* da lugar a que *scanf()* lea primero un *int*,

después lea, y descarte, los dos puntos, y luego lea otro *int*. Si el carácter especificado no se encuentra, *scanf()* termina su ejecución.

Todas las variables utilizadas para recibir valores (si son necesarias), deben ser pasadas por "referencia", o sea, por sus direcciones. Esto supone que los argumentos deben ser punteros a las variables.

La presencia del signo * después del signo % y antes del código del formato produce que *scanf()* lea, pero no asigne el valor leído a ninguna variable. Por ejemplo:

```
int x,y;

scanf("%d%*c%d",&x,&y);
```

Provoca que, si la entrada es *10/20*, se le asigne el valor *10* a la variable *x*, se lea, y se descarte el signo */*, y después se asigne el valor *20* a la variable *y*.

La función *scanf()* devuelve un número igual al de campos que han sido asignados correctamente, este número no incluye los campos que fueron leídos, pero no asignados, utilizando el modificador * para eliminar la asignación. En caso de error devuelve *EOF*.

La función *printf()* se usa para escribir cualquier tipo de dato a la pantalla. Su formato es:

```
int printf(const char *formato[,argumento,...]);
```

La cadena apuntada por formato consta de dos tipos de elementos. El primer tipo esta constituido por los caracteres que se mostraran en pantalla. El segundo tipo contiene las ordenes de formato que describen la forma en que se muestran los argumentos. Las ordenes de formato están precedidas por el signo *%* y le sigue el código de formato. Estas ordenes de formato son:

Especificador	**Descripción**
%c	Carácter.
%d	Enteros decimales con signo.
%i	Enteros decimales con signo.
%e	Punto flotante en notación científica (e minúscula).
%E	Punto flotante en notación científica (E mayúscula).
%f	Punto flotante.
%g	Usar el más corto de %e y %f.
%G	Usar el más corto de %E y %f.
%o	Octal sin signo.
%s	Cadena de caracteres.
%u	Enteros decimales sin signo.
%x	Hexadecimales sin signo (letras minúsculas).
%X	Hexadecimales sin signo (letras mayúsculas).
%p	Mostrar un puntero.

%n	El argumento asociado es un puntero a un entero, el cual recibirá el número de caracteres escritos.
%%	Imprimir el signo %.

Tabla 10.1.2: Especificadores de formato de la función printf().

Además, e igual que con la función *scanf()*, existen los modificadores *h*, *l* y *L*. Para su uso consultar la función *scanf()*.

La función *printf()* devuelve el número de carácteres escritos. En caso de error devuelve el valor *EOF*.

Veamos un ejemplo de uso de las funciones *scanf()* y *printf()*:

```
#include <stdio.h>

int main(void)
{
   int a,b;

   printf("\nIntroduce el valor de a: ");
   scanf("%d",&a);
   printf("\nIntroduce el valor de b: ");
   scanf("%d",&b);

   if (b!=0)
      printf("\nEl valor de %d dividido %d es: %f\n",
              a,b,(float)a/b);
   else
      printf("\nError, b vale 0\n");

   return 0;
}
```

10.2 - Entrada y salida desde fichero.

Antes de explicar la entrada y salida desde fichero, conviene explicar el tipo de dato *FILE **. Dicho tipo de dato es el "puntero de fichero", y es, realmente, una estructura que contiene la información sobre el nombre del fichero abierto, su modo de apertura (lectura, escritura, etc.), estado, etc. Dicho "puntero de fichero", por tanto, especifica el fichero que se esta usando y es, la forma que poseen las funciones de entrada y salida desde fichero de conocer sobre que archivo actúan.

Sobre un archivo es necesario, antes de poder usarlo, realizar una operación, la apertura del mismo; una vez terminado su uso, es necesaria otra operación, cerrar el archivo. De esto se encargan dos funciones de C. Dichas funciones son *fopen()* y *fclose()*. Veámoslas con detalle:

La función *fopen()* se encarga de abrir un archivo. Su definición es:

```
FILE *fopen(char *nombre,char *modo);
```

Donde *nombre* es un string que contiene el nombre del archivo que queremos leer y *modo* es otro string que contiene el modo de apertura deseado. Dichos modos de apertura son:

Modo	**Descripción**
r	Abrir un archivo para lectura.
w	Crear un archivo para escritura.
a	Abrir un archivo para añadir.
rb	Abrir un archivo binario para lectura.
wb	Crear un archivo binario para escritura.
ab	Abrir un archivo binario para añadir.
rt	Abrir un archivo de texto para lectura.
wt	Crear un archivo de texto para escritura.
at	Abrir un archivo de texto para añadir.
r+	Abrir una archivo para lectura/escritura.
w+	Crear un archivo para lectura/escritura.
a+	Abrir un archivo para leer/añadir.
r+b	Abrir un archivo binario para lectura/escritura.
w+b	Crear un archivo binario para lectura/escritura.
a+b	Abrir un archivo binario para leer/añadir.
r+t	Abrir un archivo de texto para lectura/escritura.
w+t	Crear un archivo de texto para lectura/escritura.
a+t	Abrir un archivo de texto para leer/añadir.

Tabla 10.2.1: Modos de apertura de un fichero con la función fopen().

En todos los casos de añadir, si el archivo especificado no existe, se procede a crearlo.

Si no se especifica en *modo* si la apertura se realiza para un archivo binario o texto, dependerá de la configuración del sistema que la apertura sea en binario o en texto, siendo en la mayoría de los casos en modo texto. La diferencia fundamental entre modo texto y modo binario es que en modo texto, secuencias de lectura tales como retorno de carro/alimentación de línea se traducen en un único carácter nueva línea, mientras que en modo texto eso no sucede; el efecto contrario sucede en escritura.

La función *fopen()* devuelve un puntero de tipo *FILE* a la estructura que representa el archivo abierto. En caso de que no pueda abrir o crear el archivo especificado, se devuelve un puntero *NULL*, por lo cual, siempre que se abra un archivo, deberá comprobarse que el valor devuelto no es *NULL*, y entonces, el código deberá ser:

```
FILE *fp;

if ((fp=fopen("prueba","w"))==NULL)
{
   puts("\nNo puedo abrir el fichero\n");
   exit(1);
}
```

La función *fclose()* cierra un archivo. Su definición es:

```
int fclose(FILE *fp);
```

Donde *fp* es el puntero al fichero abierto mediante la función *fopen()*.

La función *fclose()* cierra el archivo, lo cual da lugar a que el buffer de archivo existente en memoria se libere, escribiéndose en el fichero si es necesario, además, libera el bloque de control de archivo, lo cual lo hace disponible para otro archivo (el sistema operativo limita el número de ficheros abiertos simultáneamente).

Un valor devuelto de cero indica que el archivo fue cerrado con éxito. Cualquier valor distinto de cero indica un error.

Veamos un ejemplo de uso de *fopen()* y *fclose()*:

```
#include <stdio.h>

int main(int argc,char *argv[])
{
   FILE *fp;

   if (argc!=2)
   {
      puts("Nombre del fichero no pasado");
     return 0;
   }

   if ((fp=fopen(argv[1],"r"))==NULL)
   {
      printf("Error abriendo el fichero: %s\n",argv[1]);
      return 0;
   }

   if (fclose(fp))
   {
      puts("Error cerrando el fichero");
      return 1;
   }

   return 0;
}
```

Una vez abierto un archivo, y hasta que se proceda a cerrarlo es posible leer, escribir, etc., en el, según se indique en el modo de apertura. Las principales funciones de lectura y escritura sobre un archivo son:

```
int getc(FILE *fp);
int putc(int ch,FILE *fp);
char *fgets(char *str,int n,FILE *fp);
int fputs(const char *str,FILE *fp);
int fscanf(FILE *fp,const char *formato[,dirección,...]);
int fprintf(FILE *fp,const char *formato[,argumento,...]);
int fread(void *memoria,int num,int cont,FILE *fp);
int fwrite(void *memoria,int num,int cont,FILE *fp);
```

La función *getc()* lee caracteres del archivo asociado a *fp*. Devuelve *EOF* cuando se alcanza el final del archivo.

La función *putc()* escribe el carácter *ch* en el archivo asociado a *fp*. Devuelve el carácter escrito si funciona de forma correcta, y *EOF* en caso de error.

La función *fgets()* funciona de igual forma que la función *gets()*, solo que, además de leer del fichero asociado a *fp*, el parámetro *n* indica el número máximo de caracteres que se pueden leer. Existe, además, una sutil diferencia, la función *fgets()* no elimina el retorno de carro (si se lee) de la cadena de entrada, sino que lo conserva en la misma,. añadiendo a continuación de dicho retorno de carro, y de forma automática, el carácter de fin de cadena (*'\0'*).

La función *fputs()* funciona igual que la función *puts()*, solo que, además de escribir en el fichero asociado a *fp*, no añade al final del string un retorno de carro, tal y como hacia la función *puts()*.

Las funciones *fscanf()* y *fprintf()* funcionan de forma similar a sus equivalentes sobre la consola *scanf()* y *printf()*, solo que leen o escriben del archivo asociado a *fp*.

Las función *fread()* permite leer un bloque de datos. Su declaración es:

```
int fread(void *memoria,int num,int cont,FILE *fp);
```

Donde *memoria* es un puntero a la zona de memoria donde se almacenaran los datos leídos, *num* es el tamaño (en bytes) de cada uno de los bloques a leer, *cont* es el número de bloques (cada uno de *num* bytes de tamaño) a leer, y *fp* es el puntero al fichero desde donde se lee.

La función *fread()* devuelve el número de bloques (no bytes) realmente leídos.

La función *fwrite()* es la función dual a *fread()*. La función *fwrite()* permite escribir un bloque de datos. Su declaración es:

```
int fwrite(void *memoria,int num,int cont,FILE *fp);
```

Donde *memoria* es un puntero a la zona de memoria donde se encuentran los datos a escribir, *num* es el tamaño (en bytes) de cada uno de los bloques a escribir, *cont* es el número de bloques (cada uno de *num* bytes de tamaño) a escribir, y *fp* es el puntero al fichero desde donde se escribe.

La función *fwrite()* devuelve el número de bloques (no bytes) realmente escritos.

Un aspecto a resaltar de las funciones *fread()* y *fwrite()* es el hecho de que no realizan ningún tipo de conversión con los datos leídos o escritos, así , la secuencia retorno de carro/alimentación de línea, no es convertida en el carácter nueva línea en la escritura, y viceversa para la lectura. Es por ello, que dichas funciones son, generalmente, usadas con archivos abiertos en modo binario.

Veamos un ejemplo de uso de *fread()* y *fwrite()*:

```
#include <stdio.h>

#define TAM 1000

int main(int argc,char *argv[])
{
   FILE *f_inp,*f_out;
   char buffer[TAM];
   int num;

   if (argc!=3)
      return 0;

   if ((f_inp=fopen(argv[1],"rb"))==NULL)
      return 0;
   if ((f_out=fopen(argv[2],"wb"))==NULL)
      exit(1);
   while ((num=fread(buffer,sizeof(char),TAM,f_inp))!=0)
      fwrite(buffer,sizeof(char),num,f_out);
   if (fclose(f_inp) || fclose(f_out)
      exit(1);

   return 0;
}
```

Además de las funciones de entrada y salida de datos desde archivo, descritas con anterioridad, existen tres funciones que no son de entrada y salida de datos y que conviene explicar. Dichas funciones son:

```
int ferror(FILE *fp);
void rewind(FILE *fp);
int fseek(FILE *fp,long num,int origen);
```

La función *ferror()* devuelve si durante la última operación realizada sobre el archivo asociado a *fp* se produjo o no un error. Devuelve el valor cero si no se produjo error, y un valor distinto de cero si se produjo error.

La función *rewind()* posiciona el indicador de posición del archivo *fp* al principio del mismo.

La función *fseek()* se usa para operaciones de entrada y salida de acceso aleatorio. La función *fseek()* desplaza el indicador de posición del archivo *fp* un tamaño *num* desde la posición especificada por *origen*. Los valores validos para *origen* son:

Origen	Nombre de la constante	Valor
Comienzo del archivo	SEEK_SET	0
Posición actual	SEEK_CUR	1
Final del archivo	SEEK_END	2

Tabla 10.2.2: Valores del origen en la función fseek().

La función *fseek()* devuelve un valor de cero si funciona correctamente. Un valor distinto de cero indica un error en la última operación de posicionamiento en el fichero.

La función *fseek()* solo funciona correctamente en archivos abiertos en modo binario, pues, dadas las conversiones que se realizan en ciertas transacciones de caracteres en los archivos abiertos en modo texto, se producirían errores en el posicionamiento en el fichero al usar dicha función. Veamos un ejemplo de uso de *fseek()*:

```
#include <stdio.h>

int LeeCaracter(FILE *fp,long pos,int origen)
{
   if (fseek(fp,pos,origen))
      return(EOF);
   return(getc(fp));
}
```

Antes de terminar este tema, es necesario comentar la existencia de tres ficheros que son abiertos de forma automática al comenzar la ejecución del programa, y cerrados, también de forma automática, al terminar la misma. Estos archivos son la entrada standard (*stdin*), la salida standard (*stdout*) y la salida standard de error (*stderr*). Normalmente estas ficheros están asociados a la consola, pero pueden redireccionarse a cualquier otro dispositivo. Además, dado que son exactamente igual que ficheros, pueden usarse sus nombres en los mismos lugares que se usan las variables de tipo *FILE* *, por lo cual, cualquier función de fichero puede usarse con la consola usando estos archivos standard abiertos al comenzar el programa. Es por ello, que podemos leer, por ejemplo, una cadena desde el teclado de la siguiente forma:

```
char cadena[100];
fgets(cadena,100,stdin);
```

Y escribir dicha cadena, por ejemplo en la salida standard de error, de la forma:

```
fputs(cadena,stderr);
```

Tema 11 - Asignación dinámica de memoria.

Antes de empezar con el desarrollo del tema, es necesario aclarar que el mismo no pretende explicar las estructuras de datos dinámicas, sino tan solo dar unas ligeras nociones básicas sobre la posibilidad de asignar memoria de forma dinámica, esto es, en tiempo de ejecución, y por tanto de crear nuevas variables.

Las funciones que realizan un manejo activo de la memoria del sistema requieren todas ellas para su correcto funcionamiento la inclusión, mediante la directiva del prepocesador *#include* del archivo de cabecera <*stdlib.h*>.

11.1 - Reserva dinámica de memoria.

En C, la reserva dinámica de memoria se realiza mediante el uso de funciones, existen varias funciones de reserva de memoria (ver apéndice A), pero aquí solo explicaremos la reserva dinámica de memoria mediante la función *malloc()*. La función *malloc()* tiene la forma:

```
void *malloc(unsigned num_bytes);
```

Siendo *num_bytes* el número de bytes que se desean reservar. La función *malloc()* devuelve un puntero al tipo de datos *void* (sin tipo). Dicho puntero puede ser asignado a una variable puntero de cualquier tipo base mediante una conversión forzada de tipo de datos (casts). Veamos un ejemplo:

```
int *a;
a=(int *)malloc(sizeof(int));
```

Y ahora podríamos realizar la siguiente asignación:

```
*a=3;
```

La función *malloc()*, y en general, cualquier función de reserva dinámica de memoria, devuelve un puntero nulo (*NULL*) si la reserva de memoria no puede realizarse, generalmente por falta de memoria disponible. Por ello, antes de usar un puntero devuelto por la función *malloc()* o por cualquier otra función de reserva dinámica de memoria es imprescindible, con el fin de evitar posibles fallos en la ejecución del programa, comprobar que dicho puntero no es nulo (*NULL*). Veamos algunos ejemplos de reserva dinámica de memoria:

```
float *a;
a=(float *)malloc(sizeof(float));
if (a==NULL) exit(0); /* Salimos del programa */

unsigned long int*b;
if ((b=(unsigned long int)malloc(sizeof(unsigned long int)))==NULL)
   exit(0); /* Salimos del programa */

struct ALFA{
```

```
   unsigned a;
   float b;
   int *c;
}*d;
if ((d=(struct ALFA *)malloc(sizeof(struct ALFA)))==NULL)
   exit(0); /Salimos del programa */
```

11.2 - Liberación dinámica de memoria.

La memoria dinámica reservada es eliminada siempre al terminar la ejecución del programa por el propio sistema operativo. Sin embargo, durante la ejecución del programa puede ser interesante, e incluso necesario, proceder a liberar parte de la memoria reservada con anterioridad y que ya ha dejado de ser necesario tener reservada. Esto puede realizarse mediante la función *free()*. La función *free()* tiene la forma:

```
void free(void *p);
```

Donde *p* es la variable de tipo puntero cuya zona de memoria asignada de forma dinámica queremos liberar. Veamos un ejemplo de liberación de memoria:

```
int *a;
if ((a=(int *)malloc(sizeof(int)))==NULL)
   exit(0); /* Salimos del programa */
......
free(a);
```

Un aspecto a tener en cuenta es el hecho de que el puntero a liberar no debe apuntar a nulo (*NULL*), pues en tal caso se producirá un fallo en el programa. Es por ello que cobra aún más sentido la necesidad de comprobar al reservar memoria de forma dinámica que la reserva se ha realizado de forma correcta, tal y como se explico en el apartado anterior.

11.3 - Ejemplo de asignación y liberación dinámica de memoria.

Vamos a ver un sencillo ejemplo práctico de como asignar y liberar memoria. Para ello construiremos las funciones necesarias para crear, manejar y liberar de forma dinámica una lista ligada.

En primer lugar, definiremos la estructura de datos necesaria para ello. Esta estructura de datos es:

```
struct LISTA{
   tipo dato;
   struct LISTA *sig;
};
```

Donde *tipo* es cualquier tipo de datos valido (*float*, *int*, *long int*, etc.)

Las variables necesarias para crear la lista son las siguientes:

```
struct LISTA *cabeza=NULL,*p;
tipo dato;
```

El código de la función de creación de la lista, con inserción por la cabeza:

```
struct LISTA *CrearLista(struct LISTA *cabeza,tipo dato)
{
   struct LISTA *p;

   if ((p=(struct LISTA *)malloc(sizeof(struct LISTA)))==NULL)
      exit(0); /* Salimos del programa */
   p->dato=dato;
   p->sig=cabeza;
   return p;
}
```

Siendo la llamada para la creación de la forma:

```
cabeza=CrearLista(cabeza,dato);
```

La función para obtener un elemento de la lista es:

```
struct LISTA *BuscarLista(struct LISTA *p,tipo dato)
{
   while (p!=NULL && p->dato!=dato)
      p=p->sig;
   return p;
}
```

Siendo la llamada de la forma:

```
if ((p=BuscarLista(cabeza,dato))!=NULL)
   /* El elemento ha sido encontrado */
```

Y por último, la función para liberar un elemento de la memoria es:

```
struct LISTA *LiberarLista(struct LISTA *cabeza,tipo dato)
{
   struct LISTA p,q;

   if (cabeza!=NULL)
   {
      p=cabeza;
      if (cabeza->dato==dato)
         cabeza=cabeza->sig;
      else
         while (p!=NULL && p->dato!=dato)
         {
            q=p;
            p=p->sig;
         }
      if (p!=NULL)
      {
```

```
            q->sig=p->sig;
            free(p);
         }
      }
      return cabeza;
}
```

Siendo la llamada:

```
cabeza=LiberarLista(cabeza,dato);
```

Apéndice A - Funciones de biblioteca del standard ANSI de C.

Antes de comenzar a describir las funciones de biblioteca del standard ANSI de C, unos pequeños comentarios:

Existen muchas mas funciones de las aquí descritas, pero este pequeño conjunto de funciones es lo suficientemente amplio como para que puedan realizarse todas las operaciones necesarias.

Las funciones se encuentran clasificadas de acuerdo a la función que realizan (entrada/ salida de datos, etc.), con el fin de facilitar su uso.

Las funciones se presentan de la siguiente forma:

Nombre de la función: *fclose*
Fichero de includes donde se encuentra su prototipo: *#include <stdio.h>*
Formato de la función: *int fclose(FILE *f);*
Breve descripción de la función.

A.1 - Funciones de entrada y salida de datos.

fclose

```
#include <stdio.h>
int fclose(FILE *f);
```

La función *fclose()* cierra el archivo asociado a la variable *f* y vuelca su buffer al disco. Después de un *fclose()*, la variable *f* queda desligada del archivo y cualquier buffer que tuviera asignado se libera. Si *fclose()* se ejecuta de forma correcta, devuelve el valor cero, en cualquier otro caso devuelve un valor distinto de cero.

feof

```
#include <stdio.h>
int feof(FILE *f);
```

La función *feof()* comprueba el indicador de posición del archivo para determinar si se ha alcanzado el final del archivo asociado a *f*. Un valor distinto de cero supone que el indicador de posición del archivo esta en el final del mismo, en caso contrario se devuelve el valor cero.

ferror

```
#include <stdio.h>
int ferror(FILE *f);
```

La función *ferror()* comprueba si existen errores en alguna operación realizada sobre el archivo asociado a *f***.** Un valor devuelto de cero indica que no hay errores, un valor distinto de cero indica la existencia de errores. Los indicadores de error asociados a *f* permanecen activos hasta que se cierra el archivo o se llama a las funciones *rewind()* o *perror()*.

fflush

```
#include <stdio.h>
int fflush(FILE *f);
```

La función *fflush()* vacía el buffer asociado a la variable *f*. Si el archivo ha sido abierto para escritura, *fflush()* vacía el contenido del buffer de salida en el archivo. Si el archivo ha sido abierto para lectura, *fflush()* tan solo vacía el contenido del buffer de entrada. Un valor de cero indica que el buffer se ha vaciado de forma correcta, un valor distinto de cero indica un error. Todos los buffers se vuelcan automáticamente cuando un programa termina de forma correcta, cuando están llenos, o cuando se cierra el archivo (ver *fclose()*).

fgetc

```
#include <stdio.h>
int fgetc(FILE *f);
```

La función *fgetc()* devuelve el carácter del archivo de entrada asociado a *f*, e incrementa el indicador de posición del archivo. El carácter se lee como *unsigned char* y se convierte a *int*, por lo cual no existe ningún problema en asignarle el valor devuelto por *fgetc()* a una variable de tipo carácter (*char*).

fgets

```
#include <stdio.h>
char *fgets(char *cad,int num,FILE *f);
```

La función *fgets()* lee como máximo hasta *num-1* caracteres del archivo asociado a *f* y los sitúa en el array apuntado por *cad*. Los caracteres se leen hasta que se recibe un carácter de salto de linea, un *EOF* (fin de fichero) o hasta que se llega al limite especificado. Después de leídos los caracteres, se pone automáticamente un carácter de nulo inmediatamente después del ultimo carácter leído. Si se termina la lectura por un carácter de salto de linea, dicho carácter se guarda como parte de *cad*. Si tiene éxito, *fgets()* devuelve un puntero a la dirección de *cad*. En caso de error devuelve un puntero nulo (*NULL*).

fopen

```
#include <stdio.h>
FILE *fopen(const char *nombre,const char *modo);
```

La función *fopen()* abre un archivo cuyo nombre viene dado por *nombre* y devuelve un puntero a una estructura de tipo *FILE* que se le asocia en la apertura. El

tipo de operaciones permitidas en el archivo están definidas por el valor de *modo*. Los valores permitidos de *modo* son:

Modo	**Descripción**
r	Abrir un archivo para lectura.
w	Crear un archivo para escritura.
a	Abrir un archivo para añadir.
rb	Abrir un archivo binario para lectura.
wb	Crear un archivo binario para escritura.
ab	Abrir un archivo binario para añadir.
rt	Abrir un archivo de texto para lectura.
wt	Crear un archivo de texto para escritura.
at	Abrir un archivo de texto para añadir.
r+	Abrir una archivo para lectura/escritura.
w+	Crear un archivo para lectura/escritura.
a+	Abrir un archivo para leer/añadir.
r+b	Abrir un archivo binario para lectura/escritura.
w+b	Crear un archivo binario para lectura/escritura.
a+b	Abrir un archivo binario para leer/añadir.
r+t	Abrir un archivo de texto para lectura/escritura.
w+t	Crear un archivo de texto para lectura/escritura.
a+t	Abrir un archivo de texto para leer/añadir.

Tabla A.1.1: Modos de apertura de un fichero con la función fopen().

Si *fopen()* tiene éxito en abrir el archivo, devuelve un puntero de tipo *FILE*, en caso contrario devuelve un puntero nulo (*NULL*).

fprintf

```
#include <stdio.h>
int fprintf(FILE *f,const char *formato,...);
```

La función *fprintf()* escribe en el archivo asociado a *f* los valores de los argumentos que componen su lista de argumentos según se especifica en la cadena *formato*. Devuelve un número que indica el número de caracteres escritos. Si se produce un error se devuelve un valor negativo. Para una explicación sobre sus argumentos consúltese la función *printf()*.

fputc

```
#include <stdio.h>
int fputc(int c,FILE *f);
```

La función *fputc()* escribe el carácter especificado por *c* en el archivo especificado por *f* a partir de la posición actual del archivo, y entonces incrementa el indicador de posición del archivo. Aunque *c* tradicionalmente se declara de tipo *int*, es convertido por *fputc()* a *unsigned char*, por lo cual en lugar de un *int* se puede usar como argumento un *char* o *unsigned char*. Si se utiliza un *int*, la parte alta del mismo

será ignorada y no se escribirá. Si se ejecuta de forma correcta, *fputc()* devuelve el valor *c*, en caso de error devuelve el valor *EOF*.

fputs

```
#include <stdio.h>
int fputs(const char *cad,FILE *f);
```

La función *fputs()* escribe el contenido de la cadena de caracteres apuntada por *cad* en el archivo especificado por *f*. El carácter nulo de terminación de la cadena no es escrito. En caso de error *fputs()* devuelve el valor *EOF*.

fread

```
#include <stdio.h>
int fread(void *buf,size_t tam,sise_t cuenta, FILE *f);
```

La función *fread()* lee *cuenta* numero de elementos, cada uno de ellos de *tam* bytes de longitud, del archivo asociado a la variable *f*, y los sitúa en el array apuntado por *buf*. El indicador de posición del archivo se incrementa en el número de bytes leídos. La función *fread()* devuelve el número de elementos realmente leídos. Si se leen menos elementos de los pedidos en la llamada se produce un error. La función *fread()* funciona de forma correcta en archivos abiertos en modo binario; en archivos abiertos en modo texto, pueden producirse ciertos cambios de caracteres (salto de carro seguido de salto de linea se convierte en salto de linea, etc.).

fscanf

```
#include <stdio.h>
int fscanf(FILE *f,const char *formato,...);
```

La función *fscanf()* lee del archivo asociado a la variable *f* de igual forma que la función *scanf()* lo realiza del teclado. Devuelve el numero de argumentos a los que realmente se asigna valores. Este número no incluye los campos ignorados. Si se produce un error antes de realizar la primera asignación se devuelve el valor *EOF*. Para mas información consultar la función *scanf()*.

fseek

```
#include <stdio.h>
#int fseek(FILE *f,long desp,int origen);
```

La función *fseek()* coloca el indicador de posición del archivo asociado a la variable *f* de acuerdo a los valores dados por *origen* y *desp*. Su objetivo es dar soporte a las operaciones de E/S de acceso directo. El valor de *origen* debe ser una de estas constantes, definidas en *stdio.h*:

Origen	Nombre de la constante	Valor
Comienzo del archivo	SEEK_SET	0
Posición actual	SEEK_CUR	1

Final del archivo	SEEK_END	2

Tabla A.1.2: Valores del origen en la función fseek().

La función *fseek()* devuelve un valor de cero si sucede correctamente, en caso contrario el valor devuelto es distinto de cero.

Puede utilizarse *fseek()* para mover el indicador de posición en el archivo a cualquier lugar del mismo, incluso mas alla del final del mismo, pero es un error intentar situarse antes del comienzo del archivo.

ftell

```
#include <stdio.h>
long ftell(FILE *f);
```

La función *ftell()* devuelve el valor actual del indicador de posición del archivo asociado a la variable *f*. Para archivos binarios, el valor devuelto es el número de bytes desde el principio del archivo. Para archivos de texto solo debe usarse como argumento para la función *fseek()*, ya que, debido a que secuencias de caracteres como retorno de carro y salto de línea pueden sustituirse por un salto de línea, el tamaño aparente del archivo puede variar. Si falla la función *ftell()* devuelve el valor *-1L*.

fwrite

```
#include <stdio.h>
int fwrite(const void *buf,size_t tam,size_t cuenta,FILE *f);
```

La función *fwrite()* escribe *cuenta* numero de elementos, cada uno de ellos de *tam* bytes de longitud, del array apuntado por *buf* al archivo asociado a la variable *f***.** El indicador de posición del archivo se incrementa en el número de bytes escritos. La función *fwrite()* devuelve el número de elementos realmente escritos. Si se escriben menos elementos de los pedidos en la llamada se produce un error. La función *fwrite()* funciona de forma correcta en archivos abiertos en modo binario; en archivos abiertos en modo texto, pueden producirse ciertos cambios de caracteres (salto de carro seguido de salto de linea se convierte en salto de linea, etc.).

getc

```
#include <stdio.h>
int getc(FILE *f);
```

La función *getc()* devuelve del archivo de entrada asociado a la variable *f* el siguiente carácter desde la posición actual e incrementa el indicador de posición del archivo. El carácter se lee como *unsigned char* y se transforma en un *int*. Si se alcanza el final de archivo devuelve el carácter *EOF*. Debido a que *EOF* es un valor valido para archivos abiertos en modo binario, debe utilizarse la función *feof()* para comprobar el final del fichero en dichos archivos.

gets

```
#include <stdio.h>
char *gets(char *cad);
```

La función *gets()* lee caracteres desde *stdin* (entrada standard, normalmente el teclado), y los sitúa en el array de caracteres apuntado por *cad*. Se leen caracteres hasta que se recibe un carácter de salto de línea o una marca de *EOF*. El carácter de terminación se transforma, automáticamente, en el carácter nulo para terminar la cadena. Si se ejecuta correctamente, *gets()* devuelve un puntero a *cad*. En caso de error se devuelve un puntero nulo (*NULL*). No existe límite al número de caracteres que leerá *gets()*, por lo cual le corresponde al programador asegurarse de que no se sobrepasa el tamaño del array apuntado por *cad*.

perror

```
#include <stdio.h>
int perror(const char *cad);
```

La función *perror()* convierte el valor de la variable global *errno* en una cadena de caracteres y escribe esta cadena en *stderr* (salida standard de error). Si el valor de *cad* no es nulo (*NULL*), se escribe primero la cadena apuntada por *cad*, seguida de dos puntos y el mensaje de error asociado.

printf

```
#include <stdio.h>
int printf(const char *formato,...);
```

La función *printf()* escribe en *stdout* (salida standard, generalmente la pantalla), los argumentos que componen la lista de argumentos bajo el control de la cadena apuntada por *formato*. La cadena apuntada por *formato* consta de dos tipos de elementos. El primer tipo esta constituido por los caracteres que se mostraran en pantalla. El segundo tipo contiene las ordenes de formato que describen la forma en que se muestran los argumentos. Una orden de formato comienza por el signo %, y le sigue el código de formato. Las ordenes de formato son:

Especificador	Descripción
%c	Carácter.
%d	Enteros decimales con signo.
%i	Enteros decimales con signo.
%e	Punto flotante en notación científica (e minúscula).
%E	Punto flotante en notación científica (E mayúscula).
%f	Punto flotante.
%g	Usar el más corto de %e y %f.
%G	Usar el más corto de %E y %f.
%o	Octal sin signo.
%s	Cadena de caracteres.
%u	Enteros decimales sin signo.
%x	Hexadecimales sin signo (letras minúsculas).

%X	Hexadecimales sin signo (letras mayúsculas).
%p	Mostrar un puntero.
%n	El argumento asociado es un puntero a un entero, el cual recibirá el número de caracteres escritos.
%%	Imprimir el signo %.

Tabla A.1.3: Especificadores de formato de la función printf().

Existen ademas los modificadores *h* (*short*), *l* (*long*) y *L*. El modificador *h* (*short*) se puede aplicar a los tipos *d*, *i*, *o*, *u*., *x* y *X*, y le dice que el tipo de datos es *short int* o *unsigned short int* según el caso. El modificador *l* (*long*), se puede aplicar a los casos anteriores, significando que el tipo de datos es *long int* o *unsigned long int*, pero, además, se puede aplicar a los tipos *e*, *E*, *f* y *g*, indicando que el tipo de datos es *double*. El modificador *L* se puede aplicar a los tipos *e*, *E*, *f* y *g*, y dice que el tipo de datos es *long double*.

La función *printf()* devuelve el número de caracteres realmente escritos. Un valor negativo indica que se ha producido un error.

putc

```
#include <stdio.h>
int putc(int c,FILE *f);
```

La función *putc()* escribe el carácter contenido en el byte menos significativo de **c** en el archivo apuntado por *f*. Dado que los argumentos de tipo *char* son transformados en argumentos de tipo *int* en el momento de la llamada, se pueden utilizar variables de tipo *char* para el argumento *c* de *putc()*. La función *putc()* devuelve el carácter escrito. En caso de error devuelve *EOF*, y, dado que *EOF* es un valor valido en archivos abiertos en modo binario, se recomienda en dicho tipo de archivos el uso de la función *ferror()* para la comprobación de errores.

puts

```
#include <stdio.h>
int puts(char *cad);
```

La función *puts()* escribe la cadena apuntada por *cad* en el dispositivo de salida standard. El carácter nulo de terminación de cadena se transforma en un carácter de salto de línea. Si tiene éxito, se devuelve un valor no negativo. En caso de error se devuelve el valor *EOF*.

rewind

```
#include <stdio.h>
void rewind(FILE *f);
```

La función *rewind()* mueve el indicador de posición del archivo apuntado por *f* al principio del mismo. La función *rewind()* inicializa también los indicadores de error y de fin de archivo asociados a la variable *f*. Ningún valor es devuelto.

scanf

```
#include <stdio.h>
int scanf(const char *formato,...);
```

La función *scanf()* es una rutina de propósito general que lee de *stdin* (dispositivo standard de entrada, normalmente el teclado). Puede leer los tipos de datos que haya y transformarlos en el formato interno adecuado. Es la inversa de la función *printf()*. La cadena de control especificada por *formato* consiste en tres clases de caracteres:

- Especificadores de formato.
- Caracteres de espacio en blanco.
- Caracteres que no sean espacios en blanco.

Los especificadores de formato de entrada están precedidos por el signo %, y dicen a *scanf()* que tipo de datos van a ser leídos a continuación. Los especificadores de formato validos son:

Especificador	**Descripción.**
%c	Leer un único carácter.
%d	Leer un entero decimal.
%i	Leer un entero decimal.
%e	Leer un número en punto flotante.
%f	Leer un número en punto flotante.
%g	Leer un número en punto flotante.
%o	Leer un número octal.
%s	Leer una cadena de caracteres.
%x	Leer un número hexadecimal.
%p	Leer un puntero.
%n	Recibe un valor igual al número de carácter leídos.
%u	Leer un entero sin signo.

Tabla A.1.4: Especificadores de formato de la función scanf().

Además, es posible utilizar los modificadores *h* (*short*), *l* (*long*) y *L* de igual forma que en la función *printf()*.

Un espacio en blanco en la cadena de control da lugar a que *scanf()* salte sobre uno o mas espacios de la cadena de entrada, un espacio en blanco puede ser un espacio, un tabulador o un salto de línea. Además, un espacio en blanco da lugar, también, a que *scanf()* lea, pero no guarde cualquier número de espacios en blanco, incluso cero.

Un carácter que no sea espacio en blanco, da lugar a que *scanf()* lea y elimine el carácter asociado. Por ejemplo, *%d:%d* da lugar a que *scanf()* lea primero un *int*, después lea, y descarte, los dos puntos, y luego lea otro *int*. Si el carácter especificado no se encuentra, *scanf()* termina su ejecución.

Todas las variables utilizadas para recibir valores a través de *scanf()* deben ser pasadas por referencia, o sea, por sus direcciones. Esto supone que los argumentos deben ser punteros a las funciones.

La presencia del signo * después del signo *%* y antes del código del formato, produce que *scanf()* lea, pero no asigne el valor leído a ninguna variable, por ejemplo:

*scanf("%d%*c%d",&x,&y);*

Provoca, si la entrada es *10/20*, que se le asigne el valor *10* a la variable *x*, se lea y descarte el signo */*, y después se asigne el valor *20* a la variable *y*.

La función *scanf()* devuelve un número igual al de campos que han sido asignados correctamente, este número no incluye los campos que fueron leídos, pero no asignados, utilizando el modificador * para eliminar la asignación.

setbuf

```
#include <stdio.h>
void setbuf(FILE *f, char *buf);
```

La función *setbuf()* se utiliza para determinar el buffer del archivo asociado a la variable *f* que se utilizara, o bien, si se llama con *buf* a nulo (*NULL*), para desactivar el buffer. Si un buffer va a ser definido por el programador, su tamaño debe ser *BUFSIZ*, siendo *BUFSIZ* una constante definida en el archivo *stdio.h*. La función *setbuf()* no devuelve ningún valor.

setvbuf

```
#include <stdio.h>
int setvbuf(FILE *f,char *buf,int modo,.size_t tam);
```

La función *setvbuf()* permite al programador especificar el buffer, el tamaño y el modo para el archivo asociado a la variable *f*. El array de caracteres apuntado por *buf* se utiliza como buffer de *f* para las operaciones de entrada y salida. Si *buf* es nulo (*NULL*), *setvbuf()* creara su propio buffer, de tamaño *tam*, mediante una llamada a la función *malloc()*. El tamaño del buffer se fija mediante el valor de *tam*, que debe ser siempre mayor de cero. La variable *modo* determina el uso del buffer. Los valores legales de *modo*, definidos en *stdio.h*, son:

Modo	Descripción
_IOFBF	Modo normal, el buffer se vacía solo cuando se llena (en escritura), o bien, cuando ya se ha leído todo su contenido (en lectura).
_IOLBF	Igual al anterior, solo que el buffer también se vacía cuando se lee o escribe un retorno de carro.
_IONBF	Desactiva el buffer.

Tabla A.1.5: Valores del modo en la función setvbuf().

La función *setvbuf()* devuelve un valor de cero si se ejecuta con éxito. En caso de error, un valor distinto de cero será devuelto.

sprintf

```
#include <stdio.h>
int sprintf(char *buf,const char *formato,...);
```

La función *sprintf()* es idéntica a la función *printf()*, excepto que la salida generada se sitúa en el array apuntado por *buf*. El valor devuelto por la función es igual al número de caracteres realmente situados en el array. Para una mayor explicación refiérase a la función *printf()*.

sscanf

```
#include <stdio.h>
int sscanf(const char *buf,const char *formato,...);
```

La función *sscanf()* es idéntica a la función *scanf()*, excepto que los datos son leídos del array apuntado por *buf*. El valor devuelto por la función es igual al número de campos que hubieran sido realmente asignados. Este número no incluye los campos que fueron saltados al utilizar el modificador de ordenes de formato *. Para más detalles vea la función *scanf()*.

ungetc

```
#include <stdio.h>
int ungetc(int c,FILE *f);
```

La función *ungetc()* devuelve el carácter especificado por el byte menos significativo de *c* al archivo especificado por *f*. Este carácter será devuelto en la siguiente operación de lectura sobre el archivo. Una llamada a *fflush()* o a *fseek()* deshace una operación *ungetc()* y deshecha el carácter previamente devuelto a la secuencia de entrada. No se debe usar *ungetc()* sobre una marca de *EOF*. El valor devuelto por la función es igual a *c*, si la función ha tenido éxito, e igual a *EOF*, si ha fallado.

A.2 -Funciones de caracteres.

isalnum

```
#include <ctype.h>
int isalnum(int ch);
```

La función *isalnum()* devuelve un valor distinto de cero si *ch* es una letra del alfabeto o un dígito. En caso contrario, se devuelve un valor distinto de cero.

isalpha

```
#include <ctype.h>
int isalpha(int ch);
```

La función *isalpha()* devuelve un valor distinto de cero si *ch* es una letra del alfabeto, en cualquier otro caso devuelve cero.

iscntrl

```
#include <ctype.h>
int iscntrl(int ch);
```

La función *iscntrl()* devuelve un valor distinto de cero si *ch* se encuentra entre *0* y *0x1F* o si *ch* es igual a *0x7F* (tecla DEL), en cualquier otro caso devuelve cero.

isdigit

```
#include <ctype.h>
int isdigit(int ch);
```

La función *isdigit()* devuelve un valor distinto de cero si *ch* es un dígito (*0..9*), en cualquier otro caso devuelve el valor cero.

isgraph

```
#include <ctype.h>
int isgraph(int ch);
```

La función *isgraph()* devuelve un valor distinto de cero si *ch* es cualquier carácter imprimible distinto del espacio, en cualquier otro caso devuelve cero.

islower

```
#include <ctype.h>
int islower(int ch);
```

La función *islower()* devuelve un valor distinto de cero si *ch* es una carácter minúscula, en cualquier otro caso devuelve cero.

isprint

```
#include <ctype.h>
int isprint(int ch);
```

La función *isprintf()* devuelve un valor distinto de cero si *ch* es cualquier carácter imprimible, incluyendo el espacio, en cualquier otro caso devuelve cero.

ispunct

```
#include <ctype.h>
```

```
int ispunct(int ch);
```

La función *ispunct()* devuelve un valor distinto de cero si *ch* es un carácter de puntuación, excluyendo el espacio, en cualquier otro caso devuelve el valor cero.

isspace

```
#include <ctype.h>
int isspace(int ch);
```

La función *isspace()* devuelve un valor distinto de cero si *ch* es un espacio, tabulador, o carácter de salto de línea, en cualquier otro caso devuelve el valor cero.

isupper

```
#include <ctype.h>
int isupper(int ch);
```

La función *isupper()* devuelve un valor distinto de cero si *ch* es una letra mayúscula, en cualquier otro caso devuelve cero.

isxdigit

```
#include <ctype.h>
int isxdigit(int ch);
```

La función *isxdigit()* devuelve un valor distinto de cero si *ch* es un dígito hexadecimal, en cualquier otro caso devuelve cero. Un dígito hexadecimal está en uno de estos rangos: *0* hasta *9*, *A* hasta *F* y *a* hasta *f*.

tolower

```
#include <ctype.h>
int tolower(int ch);
```

La función *tolower()* devuelve el equivalente en minúscula de *ch*, si *ch* es una letra mayúscula, en cualquier otro caso se devuelve *ch* sin modificar.

toupper

```
#include <ctype.h>
int toupper(int ch);
```

La función *toupper()* devuelve el equivalente en mayúsculas de *ch*, si *ch* es una letra minúscula, en cualquier otro caso se devuelve *ch* sin modificar.

A.3 - Funciones de cadenas.

memchr

```
#include <string.h>
void *memchr(const void *buffer,int ch,size_t cuenta);
```

La función *memchr()* busca en *buffer* la primera ocurrencia de *ch* en los primeros *cuenta* caracteres. La función devuelve un puntero a la primera ocurrencia del carácter *ch* en *buffer*. Si no encuentra ch, devuelve un puntero nulo (*NULL*).

memcmp

```
#include <string.h>
int memcmp(const void *buf1,const void *buf2,size_t cuenta);
```

La función *memcmp()* compara los primeros cuenta caracteres de los arrays apuntados por *buf1* y *buf2*. La comparación se hace lexicográficamente. La función devuelve un entero que es interpretado según se indica a continuación:

Valor devuelto	Descripción
Menor que cero	buf1 es menor que buf2
Igual a cero	buf1 es igual a buf2
Mayor que cero	buf1 es mayor que buf2

Tabla A.3.1: Interpretación de los valores devueltos por la función memcmp().

memcpy

```
#include <string.h>
void *memcpy(void *hacia,const void *desde,size_t cuenta);
```

La función *memcpy()* copia *cuenta* caracteres del array apuntado por *desde* en el array apuntado por *hacia*. Si los arrays se superponen, el comportamiento de *memcpy()* queda indefinido. La función devuelve un puntero a *hacia*.

memset

```
#include <string.h>
void *memset(void *buf,int ch,size_t cuenta);
```

La función *memset()* copia el byte menos significativo de *ch* en los primero *cuenta* caracteres del array apuntado por *buf*. Devuelve *buf*. Su uso más común es inicializar una región de memoria con algún valor conocido.

strcat

```
#include <string.h>
char *strcat(char *cad1,const char *cad2);
```

La función *strcat()* concatena una copia de *cad2* en *cad1*, y añade al final de *cad1* un carácter nulo (*'\0'*). El carácter nulo de terminación, que originalmente tenía *cad1*, es sustituido por el primer carácter de *cad2*. La cadena *cad2* no se modifica en esta operación. La función *strcat()* devuelve *cad1*.

strchr

```
#include <string.h>
char *strchr(char *cad,int ch);
```

La función *strchr()* devuelve un puntero a la primera ocurrencia del byte menos significativo de *ch* en la cadena apuntada por *cad*. Si no sucede, devuelve un puntero nulo (*NULL*).

strcmp

```
#include <string.h>
int strcmp(const char *cad1,const char *cad2);
```

La función *strcmp()* compara lexicográficamente dos cadenas que finalizan con el carácter nulo, y devuelve un entero que se interpreta de la siguiente forma:

Valor devuelto	Descripción
Menor que cero	cad1 es menor que cad2
Igual a cero	cad1 es igual a cad2
Mayor que cero	cad1 es mayor que cad2

Tabla A.3.2: Interpretación de los valores devueltos por la función strcmp().

strcpy

```
#include <string.h>
char *strcpy(char *cad1,const char *cad2);
```

La función *strcpy()* se utiliza para copiar el contenido de *cad2* en *cad1*. El elemento *cad2* debe ser un puntero a una cadena que finalice con un carácter nulo. La función devuelve un puntero a *cad1*.

strlen

```
#include <string.h>
unsigned int strlen(char *cad);
```

La función *strlen()* devuelve el número de caracteres de la cadena apuntada por *cad* que finaliza con un carácter nulo. El carácter nulo no se contabiliza.

strtok

```
#include <string.h>
char *strtok(char *cad1,const char *cad2);
```

La función *strtok()* devuelve un puntero a la siguiente palabra de la cadena apuntada por *cad1*. Los caracteres que constituyen la cadena apuntada por *cad2* son los delimitadores que identifican la palabra. Devuelve un puntero nulo (*NULL*) cuando no existe ninguna palabra que devolver.

La primera vez que se llama a *strtok()* se utiliza realmente *cad1* en la llamada. Las llamadas posteriores utilizan un puntero nulo (*NULL*) como primer argumento.

La función *strtok()* modifica la cadena apuntada por *cad1*, pues, cada vez que se encuentra una palabra, se pone un carácter nulo donde esta el delimitador. De esta forma *strtok()* puede continuar avanzando por la cadena.

A.4 - Funciones matemáticas.

acos

```
#include <math.h>
double acos(double arg);
```

La función *acos()* devuelve el arcocoseno de *arg*. El argumento de *acos()* debe estar en el rango de *-1* a *1*, en cualquier otro caso se produce un error de dominio.

asin

```
#include <math.h>
double asin(double arg);
```

La función *asin()* devuelve el arcoseno de *arg*. El argumento de *asin()* debe estar en el rango de *-1* a *1*, en cualquier otro caso se produce un error de dominio.

atan

```
#include <math.h>
double atan(double arg);
```

La función *atan()* devuelve el arcotangente de *arg*.

atan2

```
#include <math.h>
double atan2(double y,double x);
```

La función *atan2()* devuelve el arcotangente de *y/x*. Utiliza el signo de sus argumentos para obtener el cuadrante del valor devuelto.

ceil

```
#include <math.h>
double ceil(double num);
```

La función *ceil()* devuelve el menor entero mayor o igual que *num* y lo representa como *double*. Por ejemplo, dado *1.02*, *ceil()* devuelve *2.0*, dado *-1.02*, *ceil()* devuelve *-1.0*.

cos

```
#include <math.h>
double cos(double arg);
```

La función *cos()* devuelve el coseno de *arg*. El valor de *arg* debe venir dado en radianes.

cosh

```
#include <math.h>
double cosh(double arg);
```

La función *cosh()* devuelve el coseno hiperbólico de *arg*. El valor de *arg* debe venir dado en radianes.

exp

```
#include <math.h>
double exp(double arg);
```

La función *exp()* devuelve el número *e* elevado a la potencia de *arg*.

fabs

```
#include <math.h>
double fabs(double num);
```

La función *fabs()* devuelve el valor absoluto de *num*.

floor

```
#include <math.h>
double floor(double num);
```

La función *floor()* devuelve el mayor entero, representado como *double*, que no es mayor que *num*. Por ejemplo, dado *1.02*, *floor()* devuelve *1.0*, dado *-1.02*, *floor()* devuelve *-2.0*.

fmod

```
#include <math.h>
double fmod(double x,double y);
```

La función *fmod()* devuelve el resto de la división entera *x/y*.

log

```
#include <math.h>
double log(double num);
```

La función *log()* devuelve el logaritmo neperiano de *num*. Se produce un error de dominio si *num* es negativo, y un error de rango si el argumento es cero.

log10

```
#include <math.h>
double log10(double num);
```

La función *log10()* devuelve el logaritmo en base *10* de *num*. Se produce un error de dominio si *num* es negativo, y un error de rango si el argumento es cero.

pow

```
#include <math.h>
double pow(double base,double exp);
```

La función *pow()* devuelve *base* elevada a *exp*. Se produce un error de dominio si *base* es cero y *exp* es menor o igual a cero. También puede ocurrir si *base* es negativo y *exp* no es entero. Un desbordamiento produce un error de rango.

sin

```
#include <math.h>
double sin(double arg);
```

La función *sin()* devuelve el seno de *arg*. El valor de *arg* debe venir dado en radianes.

sinh

```
#include <math.h>
double sinh(double arg);
```

La función *sinh()* devuelve el seno hiperbólico de *arg*. El valor de *arg* debe venir dado en radianes.

sqrt

```
#include <math.h>
double sqrt(double num);
```

La función *sqrt()* devuelve la raíz cuadrada de *num*. Si se llama con un número negativo, se produce un error de dominio.

tan

```
#include <math.h>
double tan(double arg);
```

La función *tan()* devuelve la tangente de *arg*. El valor de *arg* debe venir dado en radianes.

tanh

```
#include <math.h>
double tanh(double arg);
```

La función *tanh()* devuelve la tangente hiperbólica de *arg*. El valor de *arg* debe venir dado en radianes.

A.5 - Asignación dinámica de memoria.

calloc

```
#include <stdlib.h>
void *calloc(size_t num,size_t tam);
```

La función *calloc()* asigna memoria para un array de *num* objetos, cada uno de los cuales tiene tamaño *tam*. La memoria asignada es inicializada con el valor cero. La función *calloc()* devuelve un puntero al primer byte de la región asignada. Si no existe memoria libre suficiente para satisfacer la petición, se devuelve un puntero nulo (*NULL*).

free

```
#include <stdlib.h>
void free(void *ptr);
```

La función *free()* libera la memoria apuntada por *ptr*, haciendo que dicha memoria este disponible para futuras asignaciones. Solo se debe llamar a *free()* con un puntero que haya sido previamente asignado utilizando alguna función de asignación dinámica.

malloc

```
#include <stdlib.h>
void *malloc(size_t tam);
```

La función *malloc()* devuelve un puntero al primer byte de una región de memoria de tamaño *tam* que se encuentra libre. Si no existe memoria suficiente para satisfacer la petición, se devuelve un puntero nulo (*NULL*).

realloc

```
#include <stdlib.h>
void *realloc(void *ptr,size_t tam);
```

La función *realloc()* cambia el tamaño de la memoria apuntada por *ptr* al que esta especificado por *tam*. El valor de *tam* puede ser mayor o menor que el original. Devuelve un puntero al nuevo bloque de memoria, ya que puede ser necesario que *realloc()* traslade el bloque de posición al incrementar su tamaño. Si esto sucede, el contenido del antiguo bloque se copia en el nuevo bloque, por lo cual, la información no se pierde.

Si *ptr* es un puntero nulo (*NULL*), *realloc()* simplemente asigna *tam* bytes de memoria y devuelve un puntero a dicha memoria. Si *tam* es cero, la memoria asignada se libera. Si no existe memoria suficiente para satisfacer la petición, *realloc()* devuelve un puntero nulo (*NULL*), y el bloque de memoria original se deja sin cambiar.

A.6 - Funciones varias.

abs

```
#include <stdlib.h>
int abs(int num);
```

La función *abs()* devuelve el valor absoluto del entero dado por *num*.

atof

```
#include <stdlib.h>
double atof(const char *cad);
```

La función *atof()* convierte la cadena apuntada por *cad* en un valor de tipo *double*. La cadena debe contener un número valido en coma flotante. En caso contrario el valor devuelto es indefinido.

El número puede terminar por cualquier carácter que no pueda formar parte de un número válido en coma flotante. Esto incluye espacios en blanco, signos de puntuación distintos del punto, y caracteres que no sean *E* o *e*. Así si se llama a *atof()* con la cadena *"100.00HOLA"*, devolverá el valor *100.00*.

atoi

```
#include <stdlib.h>
int atoi(const char *cad);
```

La función *atoi()* convierte la cadena apuntada por *cad* en un valor de tipo *int*. La cadena debe contener un número entero valido. Si no es este el caso, el valor devuelto es indefinido, aunque, la mayoría de implementaciones de la función devuelven el valor cero.

El número puede acabar con cualquier carácter que no pueda formar parte de un número entero. Esto incluye espacios en blanco, signos de puntuación, y cualquier carácter que no sea la *E* o la *e*. Esto supone que si se llama a *atoi()* con la cadena *"123.23"*, devolverá el valor *123*.

atol

```
#include <stdlib.h>
long int atol(const char *cad);
```

La función *atol()* convierte la cadena apuntada por *cad* en un valor de tipo *long int*. Para más información consultar la función *atoi()*.

exit

```
#include <stdlib.h>
void exit(int estado);
```

La función *exit()* da lugar inmediatamente a la terminación normal de un programa. El valor de *estado* se pasa al proceso que llamo a este programa, normalmente el sistema operativo, si el entorno lo soporta. Por convenio, si el valor de *estado* es cero, se supone que se ha producido una terminación normal del programa. Un valor distinto de cero puede utilizarse para indicar un error definido por la implementación.

labs

```
#include <stdlib.h>
long labs(long num);
```

La función *labs()* devuelve el valor absoluto de *num*.

system

```
#include <stdlib.h>
int system(const char *cad);
```

La función *system()* pasa la cadena apuntada por *cad* como una orden al procesador de ordenes del sistema operativo. Si se llama a *system()* con un puntero nulo (*NULL*), devuelve un valor distinto de cero si está presente un procesador de

ordenes, en otro caso, se devuelve un valor distinto de cero. Si *cad* no es un puntero nulo (*NULL*), *system()* devuelve el valor cero si la orden ha sido correctamente ejecutada, y un valor distinto de cero en caso contrario.

Apéndice B: Ejemplos de programas en C.

En este apéndice se incluyen algunos programas de ejemplo escritos en lenguaje C. Los programas han sido realizados de forma que puedan ser compilados en la mayoría de compiladores existentes para los sistemas operativos MS-DOS y UNIX sin que exista la necesidad de realizar ningún tipo de cambio en los mismos.

B.1 - palindro.c.

```
/* Programa que calcula si una palabra es palindroma, esto es, se lee
igual de derecha a izquierda que de izquierda a derecha. */

#include <stdio.h>
#include <string.h>

#define TAM 100

/* Rutina que calcula si una palabra es palindroma.
Parametros: char *cadena Puntero al string con la palabra.
Return: int 0 no palindroma, <>0 palindroma. */

int Palindroma(char *cadena)
{
   register int i,j;

   i=0;
   j=strlen(cadena)-1;
   while (i<j && cadena[i]==cadena[j])
   {
      i++;
      j--;
   }
   return (i>=j);
}

int main(void)
{
   char cadena[TAM];

   printf("\nIntroduce la palabra\n");
   gets(cadena);
   printf("La palabra: %s %s palindroma.\n",cadena,
      (Palindroma(cadena)) ? "es" : "no es");
   return 0;
}
```

B.2 - matriz.c.

```
/* Programa que calcula el producto de dos matrices. */

#include <stdio.h>
```

```
/* Definicion del tamaño maximo */
#define TAM 10

/* Definicion de los codigos de error */
#define OK 0
#define ERROR 1

/* Definicion de la estructura de datos */
struct MATRIZ
{
   unsigned fila,columna;
   float matriz[TAM][TAM];
};

/* Rutina que muestra un menu y pide una opcion del menu.
Parametros: Ninguno.
Return: char Opcion del menu elegida. */

char Menu(void)
{
   register char d;

   printf("\nElige la opcion deseada:\n");
   printf("\t0 -- Salir del programa.\n");
   printf("\t1 -- Cambiar la matriz A.\n");
   printf("\t2 -- Cambiar la matriz B.\n");
   printf("\t3 -- Calcular A*B\n");
   printf("\t4 -- Calcular B*A\n");
   while ((d=getchar())<'0' || d>'4');
   return d;
}

/* Rutina que pide el numero de filas o de columnas de una matriz.
Parametros: char *cadena Puntero al string a mostrar.
Return: unsigned Numero de filas o de columnas. */

unsigned PedirTamano(const char *cadena)
{
   unsigned valor;

   do
   {
      printf("%s",cadena);
      scanf("%u",&valor);
   }
   while (valor==0 || valor>TAM);
   return valor;
}

/* Rutina que cambia una matriz.
Parametros: struct MATRIZ *a Puntero a la matriz que vamos a cambiar.
Return: Ninguno. */

void PedirMatriz(struct MATRIZ *a)
{
   register unsigned i,j;
   float valor;
```

```
   a->fila=PedirTamano("\nNumero de filas de la matriz: ");
   a->columna=PedirTamano("\nNumero de columnas de la matriz: \n");
   for(i=0;i<a->fila;i++)
      for(j=0;j<a->columna;j++)
      {
         printf("M[%u][%u]: ",i,j);
         scanf("%f",&valor);
         a->matriz[i][j]=valor;
      }
}

/* Rutina que multiplica dos matrices. Las matrices se pasan por
puntero pues ello es mas rapido, aunque no se modifican en toda la
funcion.
Parametros: struct MATRIZ *a Puntero a la estructura con la primera
matriz a multiplicar.
            struct MATRIZ *b Puntero a la estructura con la segunda
matriz a multiplicar.
            struct MATRIZ *res Puntero a la estructura que contendra
el resultado.
Return: int Codigo de error. */

int Multiplicar(const struct MATRIZ *a,const struct MATRIZ *b,struct
MATRIZ *res)
{
   register unsigned i,j,k;

   if (a->columna!=b->fila)
      return ERROR;
   res->fila=a->fila;
   res->columna=b->columna;
   for(i=0;i<a->fila;i++)
      for(j=0;j<b->columna;j++)
      {
         res->matriz[i][j]=0;
         for(k=0;k<a->fila;k++)
            res->matriz[i][j]+=a->matriz[i][k]*b->matriz[k][j];
      }
   return OK;
}

/* Rutina que muestra en pantalla el resultado de la operacion.
Parametros: struct MATRIZ *res Puntero a la estructura con el
resultado.
Return: Ninguno. */

void Mostrar(const struct MATRIZ *res)
{
   register unsigned i,j;

   for(i=0;i<res->fila;i++)
   {
      for(j=0;j<res->columna;j++)
         printf("Res[%u][%u]= %f\n",i,j,res->matriz[i][j]);
      printf("\nPulsa Enter para continuar.\n");
      getchar();
   }
}
```

```
int main(void)
{
   struct MATRIZ a,b,res;
   char d;

   a.fila=a.columna=b.fila=b.columna=1;
   a.matriz[0][0]=b.matriz[0][0]=1.0;
   do
      switch(d=Menu())
      {
         case '0':break;
         case '1':PedirMatriz(&a);
            break;
         case '2':PedirMatriz(&b);
            break;
         case '3':
            if (Multiplicar(&a,&b,&res)==ERROR)
               printf("\nNo es posible multiplicar A*B\n");
            else
               Mostrar(&res);
            break;
         case '4':
            if (Multiplicar(&b,&a,&res)==ERROR)
               printf("\nNo es posible multiplicar B*A\n");
            else
               Mostrar(&res);
            break;
      }
   while (d!='0');
   return 0;
}
```

B.3 - ordenar.c.

```
/* Programa que ordena un fichero de cualquier tamaño mediante el
algoritmo QuickSort. El fichero contiene como primer elemento un
unsigned con el numero de elementos del fichero, y a continuacion
figuran todos los elementos a ordenar */

#include <stdio.h>
#include <stdlib.h>

/* Rutina que lee el fichero de datos y devuelve un puntero al array
de la memoria reservada.
Parametros: char *nombre Nombre del fichero a leer.
            unsigned *num Puntero al unsigned que contendra el numero
de elementos del array.
Return: float * Puntero al array de float, NULL si sucede un error. */

float *LeerFichero(const char *nombre, unsigned *num)
{
   FILE *fp;
   float *p;
   register unsigned i;
```

```
   if ((fp=fopen(nombre,"rt"))==NULL)
   {
      printf("\nError, no puedo abrir el fichero: %s\n",nombre);
      return NULL;
   }
   fscanf(fp,"%u\n",num);
   if ((p=(float *)calloc(*num,sizeof(float)))==NULL)
   {
      printf("\nError, memoria insuficiente.\n");
      fclose(fp);
      return NULL;
   }
   for(i=0;i<*num;i++)
      fscanf(fp,"%f\n",&p[i]);
   fclose(fp);
   return p;
}

/* Rutina que escribe el fichero de datos ordenado.
Parametros: char *nombre Nombre del fichero donde guardar los datos.
            unsigned num Numero de elementos del array.
            float *p Puntero al array ordenado.
Return: Ninguno. */

void GuardarFichero(const char *nombre,const unsigned num,const float
*p)
{
   FILE *fp;
   register unsigned i;

   if ((fp=fopen(nombre,"wt"))==NULL)
   {
      printf("\nError, no puedo crear el fichero: %s\n",nombre);
      return;
   }
   fprintf(fp,"%u\n",num);
   for(i=0;i<num;i++)
      fprintf(fp,"%f\n",p[i]);
   fclose(fp);
}

/* Rutina que ordena un array segun el algoritmo Quick-Sort.
Parametros:   float *p Puntero al array a ordenar.
              unsigned izq Elemento de la izquierda a ordenar.
              unsigned der Elemento de la derecha a ordenar.
Return:   Ninguno. */

void QuickSort(float *p,unsigned izq,unsigned der)
{
   register unsigned i=izq,j=der;
   float val,inter;

   val=p[(i+j)/2];
   do
   {
      while (p[i]<val) i++;
      while (p[j]>val) j--;
      if (i<=j)
```

```
      {
         inter=p[i];
         p[i]=p[j];
         p[j]=inter;
         i++;
         j--;
      }
   }
   while (i<=j);
   if (izq<j) QuickSort(p,izq,j);
   if (i<der) QuickSort(p,i,der);
   return;
}

int main(int argc,char *argv[])
{
   float *p;
   unsigned num;

   if (argc!=3)
   {
      printf("\nModo de uso: %s <fichero1> <fichero2>\n",argv[0]);
      return(1);
   }
   if ((p=LeerFichero(argv[1],&num))==NULL)
      return 1;
   QuickSort(p,0,num-1);
   GuardarFichero(argv[2],num,p);
   free(p);
   return 0;
}
```

B.4 - fichero.c.

```
/* Programa que maneja una pequeña base de datos directamente sobre el
fichero */

#include <stdio.h>
#include <string.h>

/* Definicion de las constantes del programa */
#define TAM 30
#define TAM_BUFFER 10

/* Definicion de los codigos de error */
#define OK 0
#define ERROR 1

/* Definicion de las estructuras de datos del programa */
struct FICHA
{
   unsigned long dni;
   char nombre[TAM];
   char apellido[2][TAM];
};
```

```
/* Rutina que muestra un menu en pantalla.
Parametros: Ninguno.
Return: char Opcion elegida. */

char Menu(void)
{
   register char d;

   printf("\nElige una opcion:\n");
   printf("\t0 -- Salir del programa.\n");
   printf("\t1 -- Insertar un nuevo elemento.\n");
   printf("\t2 -- Buscar un elemento por su dni.\n");
   printf("\t3 -- Buscar un elemento por su apellido.\n");
   while ((d=getchar())<'0' || d>'3');
   return d;
}

/* Rutina que muestra un elemento en pantalla.
Parametros: struct FICHA *ficha Puntero a la estructura con los datos
a mostrar.
Return: Ninguno. */

void Mostrar(const struct FICHA *ficha)
{
   printf("\n\nDNI: %lu\n",ficha->dni);
   printf("NOMBRE: %s\n",ficha->nombre);
   printf("PRIMER APELLIDO: %s\n",ficha->apellido[0]);
   printf("SEGUNDO APELLIDO: %s\n",ficha->apellido[1]);
   printf("\nPulsa Enter para continuar\n");
   getchar();
}

/* Rutina que busca un elemento dado su dni.
Parametros: FILE *fichero Puntero al fichero de trabajo.
            unsigned long dni Numero de dni a buscar.
            char opcion Opcion de ejecucion, 1 mostrar, 0 no mostrar.
Return: int Codigo de error. */

int BuscarDni(FILE *fichero,const unsigned long dni,const char opcion)
{
struct FICHA ficha;

   fseek(fichero,0L,SEEK_SET);
   while (fread(&ficha,sizeof(struct FICHA),1,fichero)==1)
      if (dni==ficha.dni)
      {
         if (opcion)
            Mostrar(&ficha);
         Return OK;
      }
   return ERROR;
}

/* Rutina que busca por apellidos.
Parametros: FILE *fichero Puntero al fichero de trabajo.
            char *apellido Apellido a buscar.
Return: int Codigo de error.*/
```

```
int BuscarApellido(FILE *fichero,char *apellido)
{
   struct FICHA ficha;
   char encontrado=0;

   fseek(fichero,0L,SEEK_SET);
   while (fread(&ficha,sizeof(struct FICHA),1,fichero)==1)
      if (!strcmp(apellido,ficha.apellido[0]) ||
         !strcmp(apellido,ficha.apellido[1]))
      {
         Mostrar(&ficha);
         encontrado=1;
      }
   return (encontrado) ? OK : ERROR;
}

/* Rutina que inserta un nuevo elemento en el fichero.
Parametros: FILE *fichero Puntero al fichero de trabajo.
            struct FICHA *ficha Puntero a la ficha a insertar.
Return: int Codigo de error. */

int Insertar(FILE *fichero,const struct FICHA *ficha)
{
   if (BuscarDni(fichero,ficha->dni,0)!=ERROR)
      return ERROR;
   fseek(fichero,0L,SEEK_END);
   fwrite(ficha,sizeof(struct FICHA),1,fichero);
   return OK;
}

/* Rutina que pide los datos de una ficha.
Parametros: struct FICHA *ficha Puntero a la ficha que contendra los
datos.
            char opcion Opcion de ejecucion (0..2).
Return: struct FICHA * Puntero a la ficha que contiene los datos. */

struct FICHA *PedirDatos(struct FICHA *ficha,
const char opcion)
{
   switch(opcion)
   {
      case 0: printf("\nDNI: ");
         scanf("%lu",&ficha->dni);
         fflush(stdin);
         break;
      case 1: fflush(stdin);
         printf("APELLIDO: ");
         strupr(gets(ficha->apellido[1]));
         break;
      case 2: printf("\nDNI: ");
         scanf("%lu",&ficha->dni);
         fflush(stdin);
         printf("NOMBRE: ");
         strupr(gets(ficha->nombre));
         printf("PRIMER APELLIDO: ");
         strupr(gets(ficha->apellido[0]));
         printf("SEGUNDO APELLIDO: ");
         strupr(gets(ficha->apellido[1]));
```

```
         break;
   }
   return ficha;
}

int main(int argc,char *argv[])
{
   FILE *fichero;
   struct FICHA ficha;
   register char d;

   if (argc!=2)
   {
      printf("\nModo de uso: %s <fichero>\n",argv[0]);
      return 1;
   }
   if ((fichero=fopen(argv[1],"a+b"))==NULL)
   {
      printf("\nError creando el fichero: %s\n",argv[1]);
      return 1;
   }
   if (setvbuf(fichero,NULL,_IOFBF,
      TAM_BUFFER*sizeof(struct FICHA))!=0)
   {
      printf("\nError creando el buffer para %d elementos.\n",
         TAM_BUFFER);
      fclose(fichero);
      return 1;
   }
   do
      switch(d=Menu())
      {
         case '0':break;
         case '1':if (Insertar(fichero,PedirDatos(&ficha,2))==ERROR)
                     printf("\nNumero de dni duplicado.\n");
            break;
         case '2':PedirDatos(&ficha,0);
            if (BuscarDni(fichero,ficha.dni,1)==ERROR)
               printf("\nDni no existente.\n");
            break;
         case '3':PedirDatos(&ficha,1);
            if (BuscarApellido(fichero,ficha.apellido[1])==ERROR)
               printf("\nApellido inexistente.\n");
            break;
      }
   while (d!='0');
   fclose(fichero);
   return 0;
}
```

B.5 - arbol.c.

```
/* Programa que lee las palabras de un fichero y las almacena en un
arbol binario */

#include <stdio.h>
```

```
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

/* Definicion de la longitud maxima de una palabra */
#define TAM 30

/* Definicion de las estructuras de datos del programa */
struct ARBOL
{
   char pal[TAM+1];
   struct ARBOL *izq,*der;
};

/* Rutina que lee una palabra del fichero.
Parametros: FILE *fichero Puntero al fichero de donde se leen las
palabras.
            char *cadena Array de caracteres donde almacenar las
palabras.
Return: char * Puntero a la cadena con la palabra leida, NULL si
error. */

char *LeerPalabra(FILE *fichero,char *cadena)
{
   register char d,i=0;

   while ((d=fgetc(fichero))!=EOF && !isalpha(d));
   if (d==EOF)
      return NULL;
   do
      cadena[i++]=d;
   while (i<TAM && (isalpha(d=fgetc(fichero)) || isdigit(d) ||
      d=='_'));
   cadena[i]='\0';
   return cadena;
}

/* Rutina que crea el arbol binario, leyendo para ello el fichero.
Parametros: char *nombre Nombre del fichero a leer.
Return: struct ARBOL * Puntero a la raiz del arbol creado, NULL si
error. */

struct ARBOL *LeerFichero(char *nombre)
{
   FILE *fichero;
   char cadena[TAM+1],insertado;
   int val;
   struct ARBOL *cab=NULL,*p,*q;

   if ((fichero=fopen(nombre,"rt"))==NULL)
   {
      printf("\nError, no puedo leer el fichero: %s\n",nombre);
      return(NULL);
   }
   while (LeerPalabra(fichero,cadena)!=NULL)
   {
      if ((q=(struct ARBOL *)malloc(sizeof(struct ARBOL)))==NULL)
      {
```

```
         printf("\nError reservando memoria.\n");
         fclose(fichero);
         return NULL;
      }
      strcpy(q->pal,cadena);
      q->izq=q->der=NULL;
      if (cab==NULL)
         cab=q;
      else
      {
         p=cab;
         insertado=0;
         while (!insertado)
            if ((val=strcmp(cadena,p->pal))<0)
               if (p->izq==NULL)
               {
                  p->izq=p;
                  insertado=1;
               }
               else
                  p=p->izq;
            else
               if (val>0)
                  if (p->der==NULL)
                  {
                     p->der=q;
                     insertado=1;
                  }
                  else
                     p=p->der;
               else
                  insertado=1;
      }
   }
   fclose(fichero);
   return cab;
}

/* Rutina que muestra por pantalla el arbol ordenado a la vez que
libera la memoria.
Parametros: struct ARBOL *p Puntero al nodo a mostrar.
            unsigned *cont Puntero al contador de elementos para
permitir parar la visualizacion.
Return: Ninguno.
*/

void Mostrar(struct ARBOL *p,unsigned *cont)
{
   if (p->izq!=NULL)
      Mostrar(p->izq,cont);
   puts(p->pal);
   if (++*cont>21)
   {
      *cont=1;
      printf("\nPulsa Enter para continuar.\n");
      getchar();
   }
   if (p->der!=NULL)
```

```
      Mostrar(p->der,cont);
   free(p);
}

int main(int argc,char *argv[])
{
   struct ARBOL *p;
   unsigned cont=1;

   if (argc!=2)
   {
      printf("\nModo de uso: %s <fichero>\n",argv[0]);
      return 1;
   }
   if ((p=LeerFichero(argv[1]))==NULL)
      return 1;
   printf("\n\n\n\n\n\n");
   Mostrar(p,&cont);
   return 0;
}
```