CURSO BASICO DE PROGRAMACION EN C

Apoyo a Investigación C. P. D.
Servicios Informáticos U. C. M.

1 INDICE

2 INTRODUCCION

El lenguaje C fue inventado e implementado por primera vez por Dennis Ritchie en un DEC PDP-11 en Bell Laboratories.

Es el resultado de un proceso de desarrollo comenzado con un lenguaje anterior denominado B, inventado por Ken Thompson. En los años 70 el lenguaje B llevó al desarrollo del C. En 1978, Brian Kernighan y Dennis Ritchie publicaron el libro The C Programming Language que ha servido hasta la actualidad como definición eficiente de este lenguaje.

Durante muchos años el estándar de C fue la versión proporcionada con la versión cinco del sistema operativo UNIX. En 1983, el instituto de estándares americanos estableció un estándar que definiera el lenguaje C, conocido como ANSI C. Hoy día, todos los principales compiladores de C llevan implementado el estándar ANSI.

El lenguaje C se denomina como un lenguaje de nivel medio, puesto que combina elementos de lenguajes de alto nivel (Fortran, Pascal, Basic…) con el funcionalismo del lenguaje ensamblador.

C permite la manipulación de bits, bytes y direcciones (los elementos básicos con que funciona la computadora).

Otras características del C es que posee muy pocas palabras clave (32, donde 27 fueron definidas en la versión original y cinco añadidas por el comité del ANSI, *enum*, *const*, *signed*, *void* y *volatile*). Todas las palabras clave de C están en minúsculas (C distingue entre las mayúsculas y minúsculas). En la siguiente tabla se muestran las 32 palabras clave:

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*

Los programas en C consisten en una o más funciones. La única función que debe estar absolutamente presente es la denominada *main*, siendo la primera función que es llamada cuando comienza la ejecución del programa. Aunque *main* no forma técnicamente parte del lenguaje C, hay que tratarla como si lo fuera, pues si se emplea para nombrar una variable, probablemente confundirá al compilador.

La forma general de un programa en C es:

```
instrucciones del preprocesador
declaraciones globales
tipo_devuelto main(lista de parámetros)
{
        secuencia de sentencias
}
```

```
tipo_devuelto función_1(lista de parámetros)
{
        secuencia de sentencias
}
tipo_devuelto función_2(lista de parámetros)
{
        secuencia de sentencias
}

…..

…..
tipo_devuelto función_n(lista de parámetros)
{
        secuencia de sentencias
}
```

El programa así escrito se denomina programa fuente y puede estar escrito en uno o varios ficheros.

Para que el programa pueda ser ejecutado se debe compilar y enlazar (linkar) con todas aquellas funciones de la biblioteca que se necesiten.

El proceso de compilar consiste en traducir el programa fuente a código o lenguaje máquina.

El proceso de linkaje (enlazado) consiste en añadir rutinas (propias o bibliotecas existentes en el mercado) que también están en código máquina, es decir, están en objeto.

Una vez enlazado el programa objeto, tenemos un programa ejecutable que se puede ejecutar en el ordenador.

Estos procesos son realizados por un programa llamado compilador.

El compilador en las máquinas Alpha del C. P. D. es el DEC OSF/1 Versión 4.0. Para compilar y enlazar un programa con este compilador basta con hacer

```
cc nombre_del_programa.c
```

para crear, si no hay errores, un ejecutable (a.out). Existen múltiples opciones en el compilador que se pueden comprobar con el comando de ayuda de los sistemas operativos.

Los ejemplos del curso siguen la sintaxis aceptada por el estándar ANSI, con lo que son portables con cualquier otro compilador que lo lleve implementado.

3 ELEMENTOS

3.1 Comentarios

Los comentarios son textos que no son procesados por el compilador. Sirven como información al programador.

Para que un texto sea comentario debe estar entre los símbolos /* (marca el comienzo) y */ (marca el final de comentario).

3.2 Identificadores

Se usan para referenciar las variables, las funciones, las etiquetas y otros objetos definidos por el usuario. La longitud del identificador puede variar entre uno o varios caracteres (se recomienda no más de 31 y si el identificador está envuelto en el proceso de enlazado al menos los seis primeros deben ser significativos).

El primer carácter debe ser una letra o un símbolo subrayado y los caracteres siguientes pueden ser letras, números o símbolos de subrayado. Las minúsculas y las mayúsculas se tratan como distintas.

Un identificador no puede ser igual a una palabra clave de C y no debe tener el mismo nombre que una función ya escrita o que se encuentre en la biblioteca de C.

3.3 Constantes

Las constantes son expresiones con un significado invariable.

La representación más simple de un concepto de este lenguaje son las constantes.

Pueden ser:

- Números enteros: Su formato es “signo dígitos marcadores”. El signo puede ser “-“ (negativo) o “+” (positivo, por defecto). Los dígitos se pueden escribir en notación decimal, octal (base 8, un 0 seguido de una secuencia de números del 0 al 7) o en hexadecimal (base 16, un 0 seguido por una x (o X) y una secuencia de dígitos del 0 al 9 y de la A B a la F). Los marcadores definen el tipo de entero (ver capítulo siguiente), la ‘l’ (o L) asocia un entero long y la “u” (o U) de tipo unsigned. Por ejemplo 1234lu.

- Números reales (con parte decimal): Su formato es “signo dígitos e signo_exponente exponente marcador”. El signo indica el signo de la mantisa. Dígitos indica una secuencia de números que pueden llevar un punto separando la parte entera y la decimal. e indica el comienzo del valor del exponente de base 10. Exponente es una constante entera decimal. Marcador es una (f o F) y/o (l o L), donde las primeras indican una constante float y las segundas una doble precisión. Por ejemplo –13.13e-17f (es –13.13 por 10 a la –17).

- Caracteres: Su forma es ‘carácter’ (carácter entre apostrofos). El carácter puede ser escribible (es imprimible desde teclado) o de escape en los que hay que poner un carácter especial (\) para avisar al compilador. Todos los caracteres escribibles se pueden poner en forma de escape con el código octal correspondiente (‘a’ es igual a ‘\141’). El carácter nulo (NULL) se puede representar como ‘\0’. Este carácter es puesto siempre por el compilador al final de cualquiera cadena de caracteres.

- Cadenas de caracteres: Es una secuencia de caracteres (escribibles o de escape) encerrada entre dobles comillas.

Para asignar un identificador a una constante se realiza con la directiva *#define* (ver capítulo 12).

3.4 Variables

Una variable es una posición de memoria con nombre que se usa para mantener un valor que puede ser modificado en el programa. Todas las variables deben ser declaradas antes de poder usarlas. Una variable puede ser fijada a una constante con la sintaxis *const tipo identificador = valor* (por ejemplo *const int a=10*). También existe otro modificador del tipo de acceso (*volatile*) que permite cambiar el valor de una variable por medios no explícitamente especificados por el programa, por ejemplo la dirección de una variable global que apunta a un puerto externo (*volatile unsigned char *puerto=0x30;*).

3.5 Operadores

Los operadores son palabras o símbolos que hacen que un programa actúe sobre las variables.

En C existen seis tipos de operadores. Aritméticos, relacionales, de asignación, lógico, de dirección y de movimiento.

3.6 Sentencias

Una sentencia es una expresión en C donde se esperan unas consecuencias, normalmente son asignaciones, operaciones, llamadas a funciones, etc.

3.7 Macros del preprocesador

Una macro es una codificación de instrucciones que implican una o varias acciones. El preprocesador toma como entrada el programa fuente en C antes que el compilador y ejecuta todas las macros que encuentra.

4 TIPOS

Cuando en C, se dice que un objeto es de un tipo, se quiere decir que ese objeto pertenece a un conjunto específico de valores con los cuales se pueden realizar un conjunto de operaciones también determinadas.

Existen cinco tipos básicos: carácter, entero, coma flotante, coma flotante de doble precisión y void.

Los demás tipos se basan en alguno de estos tipos básicos. El tamaño y el rango de estos tipos de datos varían con cada tipo de procesador y con la implementación del compilador de C.

El tipo void, o bien declara explícitamente una función como que no devuelve valor alguno, o bien crea punteros genéricos.

La siguiente tabla muestra todas las combinaciones que se ajustan al estándar ANSI junto con sus rangos mínimos y longitudes aproximadas en bits.

Tipo	Tamaño en bits	Rango
char	8	-127 a 127
unsigned char	8	0 a 255
signed char	8	-127 a 127
int	16	-32767 a 32767
unsigned int	16	0 a 65535
signed int	16	-32767 a 32767
short int	16	-32767 a 32767
unsigned short int	16	0 a 65535
signed short int	16	-32767 a 32767
long int	32	-2147483647 a 2147483647
signed long int	32	-2147483647 a 2147483647
unsigned long int	32	0 a 4294967295
float	32	seis dígitos de precisión
double	64	diez dígitos de precisión
long double	64	diez dígitos de precisión

Nota: En el capítulo 3 del manual del compilador DEC C existe una tabla con sus tamaños y rangos.

C utiliza unos tipos de elementos creados, como son las enumeraciones, estructuras, uniones y estructuras y tipos definidos por el usuario.

Las **enumeraciones** son listas de constantes enteras con nombre. Para crear una enumeración se utiliza la palabra reservada **enum**. La sintaxis es la siguiente:

enum identificador {lista de nombres};

En una enumeración, el primer valor tiene, por defecto, el valor 0; el segundo nombre 1, y así sucesivamente. Este valor se puede cambiar por el programador, por ejemplo:

```
enum pajaros {gorrion=10, petirrojo=20,aguila=30};
enum pajaros pajaro_1;
```

Estructuras: Es una colección de variables que se referencia bajo un único nombre, proporcionando un medio eficaz de mantener junta la información relacionada. Una definición de estructura forma una plantilla que puede utilizarse para crear variables de estructura. En general cada elemento de la estructura está relacionado lógicamente con los otros. La palabra clave **struct** indica al compilador que se está definiendo una plantilla de estructura. Por ejemplo:

```
struct dir {
        char nombre[30];
        char calle[40];
        char ciudad [20];
            };
```

Para referir a un elemento individual de la estructura se utiliza el operador punto (.), por ejemplo, *dir.nombre = 'Pedro';*

Uniones: es una posición de memoria que es compartida por dos o más variables diferentes, generalmente de distinto tipo, en distintos momentos. La definición es similar a la de la estructura. La palabra clave es **union**:

```
union etiq {
        int i;
        char ch
            };
```

En una variable que este definida del tipo etiq, tanto el entero i como el carácter ch comparten la misma posición de memoria.

Tipos definidos: C permite definir explícitamente un nuevo nombre de tipo de dato usando la palabra clave **typedef**. Realmente no se crea una nueva clase de datos, sino que se define un nuevo nombre para un tipo existente. La forma general es:

```
typedef tipo nombre;
```

Por ejemplo, *typedef float balance;*

5 DECLARACIONES

La declaración de objetos en C tiene como finalidad dar a conocer el tipo y propiedades de los identificadores.

En general la forma de una declaración es:

(durabilidad) tipo identificador (=expresión de inicialización);

Por ejemplo, *static int n=10;*.

Todos las variables deben ser declaradas. En las declaraciones es obligado especificar el tipo.

De cada objeto en un programa C se puede establecer tres propiedades que le afectan en su relación: el alcance, la visibilidad y la durabilidad.

5.1 Alcance

El **alcance** sirve para saber en qué región del código una declaración de un objeto está activa, es decir, el objeto existe.

Si la declaración es realizada en un bloque de código entre llaves, el alcance es la región que va entre las llaves. Si se declara en la parte de arriba del fichero (normalmente) o en una parte que no va entre llaves, el alcance se establece en todo el fichero. Los identificadores establecidos con la sentencia *#define* tienen alcance durante todo el fichero o hasta que lo elimina la sentencia *#undef*. Las etiquetas de sentencia (ver capítulo 7) tienen como alcance el cuerpo de la función donde se han establecido.

5.2 Visibilidad

La **visibilidad** nos indica en qué región del código un objeto está activo. La diferencia con el alcance es que en una misma región pueden estar dos objetos con el mismo identificador, ocultando un objeto a otro.

5.3 Durabilidad

La **durabilidad** es el tiempo de ejecución del programa donde el objeto existe en la memoria. La durabilidad puede ser:

- Estática: El objeto perdura desde la compilación hasta el final. Esta durabilidad la tienen todas las funciones declaradas, las variables no declaradas en ningún cuerpo de función (incluido *main*) y las variables declaradas con **static**.

- Local: El objeto es creado en la entrada de un bloque y es borrado a la salida. Esta durabilidad la tienen los argumentos formales y las variables declaradas con **auto**

(es la declaración por defecto y no se suele poner). Las variables declaradas en un bloque son (a menos que se especifique) variables locales.

Existen otros dos tipos de durabilidad:

extern: Los objetos especificados tienen durabilidad **static** e informa al enlazador de programas para que realice las unificaciones pertinentes entre ficheros. Es decir, convierten su alcance al total del programa. La declaración del mismo objeto en otros ficheros se toma como referencia y debe tener obligatoriamente el especificador **extern**. Si se pone un valor de inicialización, se debe inicializar en la declaración de la variable que no lleva **extern**.

Register: Se puede utilizar para variables locales y argumentos de funciones. Establece una durabilidad local pero informa de un uso intensivo de memoria, haciendo que el compilador le asocie un acceso de memoria rápido.

6 OPERADORES

C es un lenguaje muy rico en operadores. Se definen seis tipos de operadores aritméticos, relacionales, de asignación, lógicos, de dirección y de movimiento.

Existe otro tipo de operador denominado molde que su función es hacer posible que una expresión sea de un tipo determinado utilizando la sintaxis

(tipo) expresión;

Siendo tipo uno de los tipos estándar de C (ver capítulo 4). Por ejemplo, si se quiere asegurar que la expresión x/2 se evalúe de tipo float, se puede escribir: *(float) x/2;*.

6.1 Operadores aritméticos

OPERADORES ARITMETICOS	
OPERADOR	**SIGNIFICADO**
+	Suma
-	Resta
*	Producto
/	Cociente de una división
%	Resto de una división

6.2 Operadores lógicos

OPERADORES LOGICOS	
OPERADOR	**SIGNIFICADO**
!	Not (no lógico)
&&	And (y lógico)
||	Or (ó lógico)

6.3 Operadores relacionales

OPERADORES RELACIONALES	
OPERADOR	**SIGNIFICADO**
==	Igual a
!=	No igual a
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

6.4 Operadores de asignación

OPERADORES DE ASIGNACION		
OPERADOR	**SENTENCIA ABREVIADA**	**SENTENCIA NO ABREVIADA**
=	=	=
++	m++	m=m+1
--	m--	m=m-1
+=	m+=n	m=m+n
-=	m-=n	m=m-n
*=	m*=n	m=m*n
/=	m/=n	m=m/n
%=	m%=n	m=m%n

Los operadores de asignación ++ y – pueden ir antes o delante de una expresión formando una nueva expresión. Estas expresiones se denominan post-incrementos o pre-incrementos (decrementos si el operador es --) y son expresiones compuestas, normalmente son del tipo *y=x++;* (*y=++x;*).

Existen una diferencia entre el post-incremento y el pre-incremento. El post-incremento primero suma uno a la variable (x en el ejemplo) y luego asigna ese valor (y en el ejemplo), mientras con el pre-incremento, la asignación es anterior.

Por ejemplo:

```
int x=1, y;
y=x++;  /*  y es 1 y x vale 2  */

int x=1, y;
y=++x;  /*  x vale 2 e y también vale 2  */
```

6.5 Operadores de dirección

OPERADORES DE DIRECCION	
OPERADOR	**SIGNIFICADO**
*	Operador de contenido de apuntado u operador de indirección
&	Operador de dirección

6.6 Operadores de movimiento

Existe un último tipo de operadores, no comentado hasta el momento, los operadores de movimiento (<<, movimiento a la izquierda y >>, a la derecha). Su función es desplazar los bits de la palabra de memoria dada tantos espacios como se le indiquen a derecha o izquierda. La forma general es:

expresion1 << expresion2

Los dos operandos deben ser del tipo entero, y mueve los bits de la expresion1 tantas posiciones como se lo indique la expresion2 (en este caso hacia la izquierda).

Por ejemplo, sea x un entero con representación interna:
00010001110011000101010111111111
x<<4; da como resultado:
00011100110001010101111111110000

6.7 Prioridad y asociatividad de los operadores

Operador	Asociatividad
() []	De izquierda a derecha
- ++ -- ! ~ * & sizeof(tipo)	De derecha a izquierda
* / %	De izquierda a derecha
+ -	De izquierda a derecha
<< >>	De izquierda a derecha
< <= > >=	De izquierda a derecha
== !=	De izquierda a derecha
&	De izquierda a derecha
&&	De izquierda a derecha
||	De izquierda a derecha
?:	De derecha a izquierda
= *= /= %= += -= &= <<= >>=	De derecha a izquierda
,	De izquierda a derecha

7 SENTENCIAS

Una sentencia es cualquier expresión en C que se espera que tenga alguna consecuencia. Pueden ser asignaciones, operaciones, llamadas a funciones o combinaciones de ellas.

Toda sentencia termina con un ";".

Una sentencia simple consiste en una expresión acabada en un punto y coma (;).

7.1 Etiquetas de sentencia

Sirven para etiquetar una sentencia de forma que el control del programa pueda ser transferido a ella. Se separan de la sentencia por dos puntos ":".

La sintaxis es:

etiqueta: sentencia;

Por ejemplo, *etiq100: x++;*.

7.2 Sentencias compuestas

Es un conjunto de sentencia simples que se encierran entre los símbolos "{" y "}" para formar un bloque de código.

Pueden aparecer en cualquier sitio en el que podría aparecer una sentencia simple.

Pueden contener declaraciones de nuevos objetos (el alcance de la declaración afecta al bloque).

La sintaxis es:

```
{
  sentencia;
  sentencia;
    ….
  sentencia;
}
```

7.3 Sentencias de selección

Existen dos tipos, *if* y *switch*. Además, el operador "?" es una alternativa para *if* en ciertas situaciones.

IF

La forma general es:

```
if (expresion) sentencia;
else sentencia;
```

Donde sentencia puede ser una sentencia simple, un bloque de sentencias o nada (en el caso de sentencias vacías). La cláusula *else* es opcional. Si la expresión del *if* es cierta (cualquier valor que no sea 0), se ejecuta la sentencia o el bloque de sentencias que constituye el objetivo del *if*; en cualquier otro caso se ejecuta la sentencia o bloque de sentencias que constituye el objetivo del *else*, si existe. Siempre se ejecuta el código asociado al *if* o al *else*, nunca ambos.

Un *if* anidado es un *if* que es el objeto de otro *if* o *else*. Son muy comunes en la programación. Un sentencia *else* siempre se refiere al *if* más próximo que esté en el mismo bloque que el *else* y que no esté asociado con un *if*. Por ejemplo:

```
if(i) {
        if(j) sentencia 1;
        if(k) sentencia 2; /* este if esta */
        else sentencia 3; /* asociado con este else */
    }
```

El estándar ANSI específica que al menos se deben permitir 15 niveles de anidamiento. En la práctica, la mayoría de los compiladores permiten bastantes más.

La alternativa ?

Se puede usar el operador "*?*" para reemplazar las sentencias *if-else* con la forma general:

```
if(condición) Expresión1;
else Expresión2;
```

la *?* es un operador ternario, ya que necesita tres operandos y tiene la forma general:

```
Condición ? Expresión1 : Expresión2;
```

Donde *Condición*, *Expresión1* y *Expresión2* son expresiones. El valor de una expresión con *?* se determina de esta forma, se evalúa *Condición*, si es cierta se evalúa *Expresión1* y se convierte en el valor de la expresión completa. Si *Condición* es falsa, se evalúa *Expresión2* y su valor se convierte en el valor de la expresión completa. Por ejemplo:

```
x = 10;
y = x >9 ? 100 : 200;
```

En el ejemplo a "*y*" se le esta asignando el valor 100.

SWITCH

Es una sentencia de selección múltiple, que compara sucesivamente el valor de una expresión con una lista de constantes enteras o de caracteres. Cuando se encuentra una

correspondencia, se ejecutan las sentencias asociadas con la constante. La forma general es:

```
switch (expresión) {
        case constante1:
                secuencia de sentencias;
                break;
        case constante1:
                secuencia de sentencias;
                break;
        case constante1:
                secuencia de sentencias;
                break;
        …..
        …..
        default:
                secuencia de sentencias;
        }
```

Se comprueba el valor de la expresión, por orden, con los valores de las constantes especificadas en las sentencias *case*. Cuando se encuentra una correspondencia, se ejecuta la secuencia de sentencias asociada con ese *case*, hasta que se encuentra la sentencia *break* o el final de la sentencia *switch*. Si no se incluye la sentencia *break*, sigue buscando más correspondencias en las siguientes sentencias *case*. La sentencia *default* se ejecuta si no se ha encontrado ninguna correspondencia. La sentencia *default* es opcional, y si no está presente, no se ejecuta ninguna acción al fallar todas las comprobaciones.

El estándar ANSI específica que una sentencia *switch* debe permitir al menos 257 sentencias *case*. En la práctica el número empleado es menor por razones de eficiencia. Aunque *case* es una sentencia de etiqueta, no tiene calidad por sí misma fuera de un *switch*.

La sentencia *switch* se diferencia de la sentencia *if* en que sólo puede comparar la igualdad, mientras que *if* puede evaluar expresiones relacionales o lógicas.

No puede haber dos constantes *case* en el mismo *switch* que tengan los mismos valores (por supuesto que una sentencia *switch* contenida en otra sentencia *switch* pude tener constantes *case* que sean iguales).

Si se utilizan constantes de tipo carácter en la sentencia *switch*, se convierten automáticamente a sus valores enteros.

7.4 Sentencias de iteración

También denominadas bucles. Permiten realizar un conjunto de instrucciones hasta que se alcance una cierta condición (que puede estar predefinida como en el bucle *for*; o no haber final predeterminado, como en los bucles *while* y *do-while*).

FOR

El formato general es:

for (inicialización; condición; incremento) sentencia;

La inicialización normalmente es una sentencia de asignación que se utiliza para iniciar la variable de control del bucle.

La condición es una expresión relacional que determina cuando finaliza el bucle.

El incremento define como cambia la variable de control cada vez que se repite el bucle.

Estas tres secciones principales deben ser separadas por punto y coma (";"). El bulce *for* continua ejecutándose mientras que la condición sea cierta. Una vez que la condición es falsa, la ejecución del programa sigue por la sentencia siguiente al *for*.

No es obligatoria ninguna de las tres expresiones, por ejemplo, se puede realizar un bucle infinito de la forma:

for(;;) printf("este bucle estará siempre ejecutándose.\n");

WHILE

Su forma general es:

while (condición) sentencia;

La condición puede ser cualquier expresión, cualquier valor distinto de 0 es cierto. El bucle itera mientras la condición sea cierta. Cuando la condición se hace falsa, el control del programa pasa a la línea siguiente al código del bucle.

DO-WHILE

A diferencia de los bucles *for* y *while*, que analizan la condición del bucle al principio del mismo, el bucle *do-while* analiza la condición al final del bucle. Esto significa que le bucle *do-while* siempre se ejecuta al menos una vez. La forma general es:

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

Aunque las llaves no son necesarias cuando sólo hay una sentencia, se utilizan normalmente para evitar confusiones al programador con el *while*.

7.5 Sentencias de salto

C tiene cuatro sentencias que llevan a cabo un salto incondicional (además de ***goto***, pero su uso no está bien visto por sus programadores): ***return***, ***break***, ***exit()*** y ***continue***.

RETURN

Se usa para volver de una función. Se trata de una sentencia de salto porque hace que la ejecución vuelva al punto en que se hizo la llamada a la función. Si hay algún valor asociado con ***return***, se trata del valor de vuelta de la función. Si no se especifica un valor de vuelta, se asume que devuelve un valor sin sentido. La forma general es:

return expresión;

Donde *expresión* es opcional. Se pueden usar tantas sentencias ***return*** como se quiera en una función. Sin embargo, la función termina al encontrar el primero.

BREAK

Tiene dos usos: para finalizar un *case* en una sentencia *switch* y para forzar la terminación inmediata de un bucle, saltando la evaluación condicional normal del ciclo. Cuando se encuentra la sentencia ***break*** dentro de un bucle, el bucle finaliza inmediatamente y el control sigue en la sentencia posterior al bucle.

EXIT()

Igual que se puede interrumpir un bucle, se puede salir anticipadamente de un programa usando la función ***exit()*** de la biblioteca estándar. Esta función da lugar a la terminación inmediata del programa, forzando la vuelta al sistema operativo. La forma general de la función ***exit()*** es:

void exit (int código_de_vuelta);

El valor de *código_de_vuelta* es el que se devuelve al proceso de llamada, que normalmente es el sistema operativo. Generalmente se usa un cero como código de vuelta para indicar que se trata de una terminación normal del programa. Se utiliza otros argumentos para indicar algún tipo de error.

CONTINUE

Funciona de forma similar a *break*. Sin embargo, en vez de forzar la terminación, ***continue*** fuerza una nueva iteración del bucle y salta cualquier código que exista entremedias. Para el bucle *for*, ***continue*** hace que se ejecuten las partes de prueba condicional y de incremento del bucle. Para los bucles *while* y *do-while*, el control del programa pasa a la prueba condicional.

8 FUNCIONES

Las funciones son los bloques constructores de C y el lugar donde se da toda la actividad del programa.

8.1 Definición

La forma general de definición de una función es:

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

El tipo especifica el tipo de valor que devuelve la sentencia *return* de la función. El valor puede ser cualquier tipo válido; si no se especifica ninguno, se asume un resultado entero.

La lista de parámetros es la lista de nombres de variables separados por comas con sus tipos asociados que reciben los valores de los argumentos cuando se llama a la función. Una función puede no tener parámetros, en cuyo caso la lista de parámetros está vacía; sin embargo, los paréntesis son necesarios.

8.2 Declaración

Cada función debe ser declarada. Su forma general es:

```
tipo nombre_función (lista de tipos (y nombres) de los argumentos);
```

Si una función va usar argumentos, debe declarar variables que acepten los valores de los argumentos. Estas variables se llaman parámetros formales de la función y se comportan como variables locales dentro de la función, creándose al entrar en la función y destruyéndose al salir. La declaración de parámetros aparece después del nombre de la función al definirla.

Los parámetros formales tienen que ser del mismo tipo que los argumentos usados al llamar una función (el compilador no dará error pero los resultados serán inesperados).

Al igual que con variables locales, se pueden hacer asignaciones a los parámetros de una función o usarlos en cualquier expresión válida. Se pueden usar como cualquier otra variable.

Una función es visible para ella misma y otras funciones desde el momento en que se define. Es visible para el propio cuerpo de la función, es decir, la recursividad esta permitida.

El código de una función es privado a esa función y sólo se puede acceder a él mediante una llamada a esa función. Las variables definidas dentro de una función son locales (a no ser que las definamos globales) por lo que no conservan su valor de una llamada a otra (excepto si se declaran como *static*, entonces el compilador no las destruye y almacena su valor para la próxima llamada, aunque la variable tiene limitado el ámbito al interior de la función).

En C, todas las funciones están al mismo nivel, es decir, no se puede definir una función dentro de otra función. Esto es por lo que C no es técnicamente un lenguaje estructurado por bloques.

8.3 Llamadas a funciones

Las funciones son llamadas para su ejecución desde cualquier parte del código, teniendo en cuenta que antes deben haber sido declaradas (y por supuesto definidas).

La llamada de una función se produce mediante el uso de su nombre en una sentencia, pasando una lista de argumentos que deben coincidir en número y tipo con los especificados en la declaración (en otro caso se produciría una conversión de tipos o resultados inesperados).

Llamadas por valor y por referencia

En general, se pueden pasar argumentos a las funciones de dos formas, por valor y por referencia.

La llamada por valor copia el valor de un argumento en el parámetro formal de la función. De esta forma, los cambios en los parámetros de la función no afectan a las variables que se usan en la llamada (es la llamada más usual, es decir, en general no se pueden alterar las variables usadas para llamar a la función).

La llamada por referencia copia la dirección del argumento en el parámetro. Dentro de la función se usa la dirección para acceder al argumento usado, significando que los cambios hechos a los parámetros afectan a la variable usada en la llamada.

Es posible simular una llamada por referencia pasando un puntero al argumento, entonces, al pasar la dirección, es posible cambiar el valor de la variable usada en la llamada.

9 ARRAYS Y CADENAS

Un array es una colección de variables del mismo tipo que se referencian por un nombre común. A un elemento específico de un array se accede mediante un índice. En C todos los arrays constan de posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la más alta al último. Los arrays pueden tener una o varias dimensiones. El array más común en C es la cadena, que simplemente es un array de caracteres terminado por uno nulo.

9.1 Arrays unidimensionales

Los arrays unidimensionales son listas de información del mismo tipo que se guardan en posiciones contiguas de memoria según el orden del índice.

La forma general de declaración es:

tipo nombre_variable [tamaño];

Los arrays tienen que declararse implícitamente para que el compilador reserve espacio en memoria para ellos. El tipo declara el tipo de los elementos del array, el tamaño indica cuántos elementos mantendrá el array.

Para declarar un array de 10 elementos denominado *p* y de tipo carácter, se escribe:

char p[10];

En este caso hemos declarado un array que tiene diez elementos, desde p[0] hasta p[9]. En C todos los arrays tienen el 0 como índice de su primer elemento.

C no comprueba los límites de los arrays. Se puede pasar cualquier extremo de un array y escribir en alguna otra variable de datos e incluso en el código del programa.

9.2 Cadenas

El uso más común de los arrays unidimensionales es como cadenas de caracteres. En C una cadena se define como un array de caracteres que termina en un carácter nulo (‘\0’). Para declarar un array de caracteres es necesario un carácter más que la cadena más larga que pueda contener, para dejar sitio para el carácter nulo del final de la cadena.

Una constante de cadena es una lista de caracteres encerrada entre dobles comillas. Por ejemplo:

“hola, que tal”

No es necesario añadir explícitamente el carácter nulo al final de las constantes de cadena, el compilador lo hace automáticamente.

9.3 Arrays multidimensionales

C permite arrays de más de una dimensión. La forma general de declaración de un array multidimensional es:

Tipo nombre [a] [b] [c] ….. [z];

Los arrays de tres o más dimensiones no se utilizan a menudo por la cantidad de memoria que se requiere para almacenarlos, ya que el almacenamiento requerido aumenta exponencialmente con el número de dimensiones.

9.4 Inicialización de arrays

C permite la inicialización de arrays en el momento de declararlos, como en cualquier variable. La forma general es:

tipo nombre [tamaño1] … [tamaño n]={lista de valores};

la lista de valores es una lista de constantes separadas por comas cuyo tipo es compatible con el tipo especificado en la declaración del array. Por ejemplo:

```
int i[10] = {1,2,3,4,5,6,7,8,9,10};
```

Con las cadenas o arrays de caracteres se hace igual, aunque permite una inicialización abreviada. Estas dos sentencias producen el mismo resultado:

```
char cad[11] = “Me gusta C”;
char cad[11] = {‘M’,’e’,’ ‘,’g’,’u’,’s’,’t’,’a’,’ ‘,’C’,’\0’};
```

Los arrays multidimensionales se inicializan del mismo modo que los unidimensionales.

10 PUNTEROS

Un puntero es una variable que contiene una dirección de memoria. Normalmente, esa dirección es la posición de otra variable de memoria. Si una variable contiene la dirección de otra variable, entonces se dice que la primera variable apunta a la segunda.

Si una variable va a contener un puntero, entonces tiene que declararse como tal. Una declaración de un puntero consiste en un tipo base, un * y el nombre de la variable. La forma general es:

*tipo *nombre;*

Donde *tipo* es cualquier tipo válido y *nombre* es el nombre de la variable puntero. El tipo base del puntero define el tipo de variables a las que puede apuntar. Técnicamente, cualquier tipo de puntero puede apuntar a cualquier dirección de la memoria, sin embargo, toda la aritmética de punteros esta hecha en relación a sus tipos base, por lo que es importante declarar correctamente el puntero.

Existen dos operadores especiales de punteros: *&* y *. El operador de dirección (&) devuelve la dirección de memoria de su operando. El operador de indirección (*) devuelve el contenido de la dirección apuntada por el operando.

Después de declarar un puntero, pero antes de asignarle un valor, éste contiene un valor desconocido; si en ese instante lo intenta utilizar, probablemente se estrellará, no sólo el programa sino también el sistema operativo. Por convenio, se debe asignar el valor nulo a un puntero que no este apuntando a ningún sitio, aunque ésto tampoco es seguro.

10.1 Asignación de punteros

Como en el caso de cualquier otra variable, un puntero puede utilizarse a la derecha de una declaración de asignación para asignar su valor a otro puntero. Por ejemplo:

```
int x;
int *p1,*p2;
p1=&x;
p2=p1;
```

Tanto p1 como p2 apuntan a x.

10.2 Aritmética de punteros

Existen sólo dos operaciones aritméticas que se puedan usar con punteros: la suma y la resta.

Cada vez que se incrementa un puntero, apunta a la posición de memoria del siguiente elemento de su tipo base. Cada vez que se decrementa, apunta a la posición del elemento anterior. Con punteros a caracteres parece una aritmética normal, sin embargo,

el resto de los punteros aumentan o decrecen la longitud del tipo de datos a los que apuntan.

Por ejemplo, si asumimos que los enteros son de dos bytes de longitud y p1 es un puntero a entero con valor actual 2000. Entonces, después de la expresión *p1++;* p1 contiene el valor 2002, no 2001.

No pueden realizarse otras operaciones aritméticas sobre los punteros más allá de la suma y resta de un puntero y un entero. En particular, no se pueden multiplicar o dividir punteros y no se puede sumar o restar el tipo *float* o el tipo *double* a los punteros.

10.3 Punteros y arrays

Existe una estrecha relación entre los punteros y los arrays. Considérese el siguiente fragmento:

```
char cad[80], *p1;
p1=cad;
```

Aquí, p1 ha sido asignado a la dirección del primer elemento del array *cad*. Para acceder al quinto elemento de cad se escribe *cad[4]* o **(p1+4)*.

Un nombre de array sin índice devuelve la dirección de comienzo del array, que es el primer elemento. El compilador traduce la notación de arrays en notación de punteros. Es decir, al crear un array se genera un puntero (en realidad una constante de puntero) con el mismo nombre que apunta a la dirección del primer elemento del array.

10.4 Arrays de punteros

Los punteros pueden estructurarse en arrays como cualquier otro tipo de datos. La declaración, por ejemplo, para un array de punteros a enteros de tamaño 10 es:

```
int *x[10];
```

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

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

Se puede encontrar el valor de *var* de la forma:

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

Si se quiere pasar un array de punteros a una función, se puede utilizar el mismo método que se utiliza para otros arrays: llamar simplemente a la función con el nombre del array sin índices. Así se pasa el puntero que apunta al array.

No se pasa un puntero a enteros, sino un puntero a un array de punteros a enteros.

10.5 Indirección múltiple

Se puede hacer que un puntero apunte a otro puntero que apunte a un valor de destino. Esta situación se denomina indirección múltiple o punteros a punteros.

Una variable que es puntero a puntero tiene que declararse como tal. Esto se hace colocando un * adicional en frente del nombre de la variable. Por ejemplo, la siguiente declaración inicial indica al compilador que ptr es un puntero a puntero de tipo *float*:

```
float **ptr;
```

10.6 Funciones de asignación dinámica, malloc() y free()

Los punteros proporcionan el soporte necesario para el potente sistema de asignación dinámica de memoria de C. La asignación dinámica es la forma en la que un programa puede obtener memoria mientras se está ejecutando.

Como ya se ha visto, a las variables globales se les asigna memoria en tiempo de compilación y las locales usan la pila. Sin embargo, durante la ejecución no se pueden añadir variables globales o locales, pero existen ocasiones en las que un programa necesita usar cantidades de memoria variables.

El centro del sistema de asignación dinámica está compuesto por las funciones (existentes en la biblioteca *stdlib.h*) *malloc()*, que asigna memoria; y *free()* que la devuelve.

El prototipo de la función *malloc()* es:

```
void *malloc(size_t número de bytes);
```

Tras una llamada fructífera, *malloc()* devuelve un puntero, el primer byte de memoria dispuesta. Si no hay suficiente memoria libre para satisfacer la petición de *malloc()*, se da un fallo de asignación y devuelve un nulo. El fragmento de código que sigue asigna 1000 bytes de memoria:

```
char *p;
p = (char *) malloc(1000);
```

Después de la asignación, *p* apunta al primero de los 1000 bytes de la memoria libre. El siguiente ejemplo dispone espacio para 50 enteros. Obsérvese el uso de *sizeof* para asegurar la portabilidad:

```
int *p;
p= (int *) malloc(50*sizeof(int));
```

La función *free()* es la opuesta de *malloc()* porque devuelve al sistema la memoria previamente asignada. Una vez que la memoria ha sido liberada, puede ser reutilizada en una posterior llamada a *malloc()*. El prototipo de la función *free()* es:

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

11 ENTRADA Y SALIDA

En C no existe ninguna palabra clave para realizar la entrada y salida de datos (E/S). Se realizan a través de funciones de biblioteca (concretamente, la biblioteca *stdio.h*, ver capítulo 13).

11.1 E/S por consola

Las funciones principales que realizan la entrada y salida sin formato son:

getchar(): Lee un carácter del teclado. Espera hasta que se pulsa una tecla y entonces devuelve su valor.
putchar(): Imprime un carácter en la pantalla en la posición actual del cursor.
gets(): Lee una cadena de caracteres introducida por el teclado y la sitúa en una dirección apuntada por su argumento de tipo puntero a carácter.
puts(): Escribe su argumento de tipo cadena en la pantalla seguida de un carácter de salto de línea.

El siguiente fragmento de código lee un carácter del teclado y lo muestra por pantalla. A continuación lee una cadena (de 10 caracteres incluido el carácter nulo) y también la muestra por pantalla:

```
#include <stdio.h>
main()
{
  char cadena[10];
  int i;
  i=getchar();
  putchar(i);
  gets(cadena);
 puts(cadena);
}
```

Las funciones principales que realizan la entrada y salida con formato, es decir, se pueden leer y escribir en distintas formas controladas, son:

printf(): Escribe datos en la consola con el formato especificado.
scanf(): Función de entrada por consola con el formato especificado.

Sus prototipos son los siguientes:

int printf (" caracteres de transmisión y escape", lista de argumentos);
int scanf (" caracteres de transmisión y escape", lista de argumentos);

En la función *printf()* (con *scanf()* no), entre las comillas se pueden poner rótulos literales mezclados con los caracteres de transmisión.

Los **caracteres de transmisión** son precedidos de un % para distinguirlos de los normales:

Caracteres de transmisión	Argumento que transmite
%c	Int: un carácter simple
%Ns	Char *: una cadena de caracteres
%Nd %Ni	Int: un número decimal
%o	Int: octal sin signo
%x %X	Int: hexadecimal sin signo
%Nu	Int: decimal sin signo
%N.Df	Float o double con D decimales, en notación fija
%N.De %N.DE	Float o double con D decimales, en notación científica
%N.Dg %N.DG	Float o double en notación científica si el exponente es menor de diez a la menos cuatro, o fija en caso contrario.
%p	Void *: escribe el número que corresponde al puntero
%%	Escribe un signo de %

Donde aparecen las letras N.D o no se pone nada o serán en realidad dos números que dicen que la transmisión total del valor al menos ocupará N posiciones (si el número necesita más de N las tomará, si usa menos las dejara en blancos, a menos que se quiera rellenar con ceros, entonces se pone 0N) y que la parte decimal tendrá como máximo las D posiciones después de un punto.

Normalmente el número se ajusta por la derecha para el campo de N posiciones que le hemos dicho que utilice; si deseamos el ajuste por la izquierda, se añade un signo menos precediendo al valor N (-N).

Una *l* precediendo a N (p.e. *%l5d*) significa que transmitiremos un *long int*: si, por el contrario, es una *h* significa que transmitiremos un *short int*.

Existe otro tipo de carácter especial, los **caracteres de escape**, que tienen un significado especial. Los caracteres de escape son los siguientes:

Caracteres de escape	Significado
\n	Nueva línea
\t	Tabulador
\b	Espacio atrás
\r	Retorno de carro
\f	Comienzo de página
\a	Pitido sonoro
\’	Comilla simple
\”	Comilla doble
\\	Barra invertida
\xdd	Código ASCII en notación hexadecimal (cada d representa un dígito)
\ddd	Código ASCII en notación octal (cada d representa un dígito)

La **lista de argumentos** estará separada por comas. Debe existir una correspondencia biyectiva entre los caracteres de transmisión (aquellos que comienzan con un %) y la lista de argumentos a transmitir.

Cabe destacar una diferencia en la lista de argumentos entre las funciones *printf()* y *scanf()*. En está ultima función (*scanf()*), la lista de argumentos va precedida por el operador de dirección(&), puesto que *scanf()* requiere que los argumentos sean las direcciones de las variables, en lugar de ellas mismas.

11.2 E/S por archivos

En C un archivo puede ser cualquier cosa, desde un archivo de disco a un terminal o una impresora. Se asocia una secuencia con un archivo específico realizando una operación de apertura, una vez que está abierto, la información puede ser intercambiada entre éste y el programa. El puntero a un archivo es el hilo que unifica el sistema de E/S con buffer. Un puntero a un archivo es un puntero a una información que define varias cosas sobre él, incluyendo el nombre, el estado y la posición actual del archivo. En esencia, el puntero a un archivo identifica un archivo en disco específico y utiliza la secuencia asociada para dirigir el funcionamiento de las funciones de E/S con buffer. Para obtener una variable de tipo puntero a archivo se debe utilizar una sentencia como la siguiente:

```
FILE *punt;
```

La función *fopen()* abre una secuencia para que pueda ser utilizada y le asocia a un archivo. Su prototipo es:

```
FILE *fopen(const char *nombre_archivo, const char *modo);
```

Donde *nombre_archivo* es un puntero a una cadena de caracteres que representan un nombre válido del archivo y puede incluir una especificación de directorio. La cadena que apunta *modo* determina cómo se abre el archivo. Los modos son los siguientes:

r: Abre un archivo de texto para lectura.
w: Crea un archivo de texto par escritura
a: Abre un archivo de texto para añadir
r+: Abre un archivo de texto para lectura/escritura
w+: Crea un archivo de texto para lectura/escritura
a+: Añade o crea un archivo de texto para lectura/escritura

por ejemplo, si desea abrir un archivo llamado prueba para escritura, escribir:

```
FILE *fp;
fp=fopen("prueba","w");
```

La función *fclose()* cierra una secuencia que fue abierta mediante una llamada a *fopen()*. Escribe toda la información que todavía se encuentre en el buffer del disco y realiza un cierre formal del archivo a nivel del sistema operativo. También libera el bloque de control de archivo asociado con la secuencia, dejándolo libre para su reutilización. A

veces es necesario cerrar algún archivo para poder abrir otro, debido a la existencia de un límite del sistema operativo en cuanto al número de archivos abiertos. Su prototipo es:

*int fclose(FILE *fp);*

La función *putc()* escribe caracteres en un archivo que haya sido abierto previamente para operaciones de escritura, utilizando la función *fopen()*. Su prototipo es:

*int putc(int car, FILE *pf);*

La función *getc()* escribe caracteres en un archivo que haya sido abierto, en modo lectura, mediante *fopen()*. Su prototipo es:

*int getc(FILE *pf);*

La función *fputs()* escribe la cadena en la secuencia especificada. Su prototipo es:

*int fputs(const char *cad, FILE *pf);*

La función *fgets()* lee una cadena de la secuencia especificada hasta que se lee un carácter de salto de línea o hasta que se han leído longitud-1 caracteres. Su prototipo es:

*int fgets(char *cad, FILE *pf);*

La función *rewind()* inicia el indicador de posición al principio del archivo indicado por su argumento. Su prototipo es:

*void rewind(FILE *pf);*

Existen otras muchas funciones en la biblioteca estándar de C (ver capítulo 13) como pueden ser

remove(): Borra el archivo especificado.
fflush(): Vacía el contenido de una secuencia de salida.
fread(): Lee tipos de datos que ocupan más de un byte. Permiten la lectura de bloques de cualquier tipo de datos.
fwrite(): Escribe tipos de datos que ocupan más de un byte. Permiten la escritura de bloques de cualquier tipo de datos.
fprintf(): Hace las funciones de *printf()* sobre un fichero.
fscanf(): Hace las funciones de *scanf()* sobre un fichero.
feof(): Detecta el final de un fichero.
ferror(): Detecta un error en la lectura/escritura de un fichero.

12 PREPROCESADOR

Se pueden incluir varias instrucciones dirigidas al compilador en el código fuente de un programa en C. Se llaman directivas de preprocesamiento y, aunque no son realmente parte del lenguaje C, amplían el ámbito del entorno de programación en C.

Se denomina preprocesador a un programa que procesa macros. Una macro es una codificación de instrucciones que implican una o varias acciones. El preprocesador toma como entrada el programa fuente en C antes que el compilador y ejecuta todas las macros que encuentra.

Los comandos para el procesador se ponen entre las líneas de código fuente pero se distinguen porque comienzan con el símbolo "#". La lista de comandos normalizados del procesador son:

Comando	Acción
#define	Define una macro. P.e.: #define CIERTO 1
#error	Fuerza al compilador a parar la compilación, mostrando un mensaje de error P.e.: #error mensaje_de_error
#include	Incluye otro archivo fuente. P.e.: #include <stdio.h>
#undef	Elimina una definición de macro previa. P.e.: #undef CIERTO
#if	Permite la inclusión de texto en función del valor de una expresión test
#endif	Marca el final de un bloque #if. P.e.: #if expresión-constante secuencia sentencias #endif
#else	Incluye un texto si el test establecido en el comando #if, o #ifdef o #ifndef que le precede ha dado resultado falso.
#elif	Significa "else if".
#ifdef y #ifndef	Permite la inclusión de texto en función de si ha sido definida o no previamente un nombre de macro respectivamente.
#line	Cambia los contenidos de _LINE_ y _FILE_, que son identificadores del compilador predefinidos. _LINE_ contiene el número de línea que se está compliando actualmente. _FILE_ es una cadena que contiene el nombre del archivo fuente que se está compilando. P.e.: #line número "nombre de archivo"
#pragma	Definida por la implementación que permite se den varias instrucciones al compilador.

La línea completa que comienza con "#" es una línea para el procesador. Si se desea que la siguiente línea del fichero sea continuación de la anterior, esta debe acabarse con " \".

En la composición de macros se pueden utilizar todos los elementos básicos del lenguaje.

Las expresiones que se pueden poner en los comandos del preprocesador cuando ponemos la fórmula “expresión”, debe ser una expresión que diese los mismos resultados que si la escribiésemos en una línea de C. El resultado debe ser una constante.

13 LIBRERIAS

El estándar ANSI C define un conjunto de funciones, así como tipos relacionados y macros, que son proporcionados para la implementación.

Todas las librerías son declaradas en un fichero cabecera. Para que sea visible al programa, se añade el comando del preprocesador *#include.* Por ejemplo

#include <stddef.h>;

Cada fichero de cabecera se denomina librería. Las librerías estándar son:

Librería	Descripción
assert.h	Contiene una macro para el diagnóstico dentro de los programas.
ctype.h	Contiene varias funciones para comprobación de tipos y transformación de caracteres.
errno.h	Contiene varias macros usadas para informar de errores.
limits.h	Contienen varias macros que definen constantes para el tamaño de tipo enteros.
float.h	Contienen varias macros que definen constantes para el tamaño de tipo flotante.
locale.h	Contienen varias macros, funciones y tipos para unidades locales, como unidad monetaria, tiempo, dígitos, etc.
math.h	Contiene una macro y varias funciones matemáticas.
setjmp.h	Contienen declaraciones que proporcionan una forma de evitar la secuencia normal de llamada y regreso de funciones.
signal.h	Contiene un tipo, dos funciones y varias macros para manejar condiciones excepcionales que aparecen durante la ejecución, tal como una señal de interrupción de una fuente externa o un error en la ejecución.
stdarg.h	Contiene un tipo y tres macros que proporcionan recursos para recorrer una lista de argumentos de función de tamaño y tipo desconocido.
stddef.h	Contiene varios tipos y macros que también están definidas en otras librerías, como size_t.
stdio.h	Contiene tipos, macros y funciones para la realización de tareas de E/S.
stdlib.h	Contiene tipos, macros y funciones para la conversión numérica, generación de números aleatorios, búsquedas y ordenación, gestión de memoria y tareas similares.
string.h	Contiene tipos, macros y funciones para la manipulación de cadenas de caracteres.
time.h	Contiene tipos, macros y funciones para la la manipulación de información sobre fechas y horas.

Para una descripción con más detalle, ver el manual del compilador DEC C.

14 EJERCICIOS

1. Escriba un programa que imprima el mensaje "Primer programa".

```
/* Primer programa  */

#include <stdio.h>

main()
{
 printf("Primer programa\n");
}
```

2. Escriba un programa que lea y escriba un carácter.

```
 /* Leer y escribir un caracter  */

#include <stdio.h>

main ()
{
char car;
printf("Escriba un caracter: ");
car=getchar();
printf("\nEl caracter introducido es %c.\n",car);
}
```

3. Escriba un programa que imprima una lista de amigos guardados en una agenda (tipo estructura).

```
  /* Lista los campos de una agenda construida en el programa  */

#include <stdio.h>
#define N 3
main()
{
 struct agenda
 {
  char nombre[25];
  char telefono[10];
  int edad;
 };
 struct agenda
amigos[N]={{"Pepe","913472314",18},{"Juan","915547623",19},
                         {"Rosa","917456778",21}};
 int i;
 for (i=0; i<N; ++i)
 {
  printf("\nAmigo %s\t telefono %s\t edad %d",amigos[i].nombre,
         amigos[i].telefono,amigos[i].edad);
 }
 printf("\n");
```

```
}
```

4. Escriba un programa para ver las longitudes y valores máximos y mínimos en bytes de los tipos básicos de programación en C.

```
 /* Muestra las longitudes en longitudes en bytes
    y los valores maximos y minimos de los tipos basicos */

#include <stdio.h>
#include <limits.h>
#include <float.h>

main()
{
 char a;
 short int b;
 int c;
 long int d;
 unsigned char e;
 unsigned short int f;
 unsigned int g;
 unsigned long int h;
 float i;
 double j;
 long double k;

 printf ("Longitud de cada uno de los tipos basicos \n\n");

 printf ("La longitud de char a= %d\n",sizeof(a));
 printf ("La longitud de short int b= %d\n",sizeof(b));
 printf ("La longitud de int c= %d\n",sizeof(c));
 printf ("La longitud de long int d= %d\n",sizeof(d));
 printf ("La longitud de unsigned char e= %d\n",sizeof(e));
 printf ("La longitud de unsigned short int f= %d\n",sizeof(f));
 printf ("La longitud de unsigned int g= %d\n",sizeof(g));
 printf ("La longitud de unsigned long int h= %d\n",sizeof(h));
 printf ("La longitud de float i= %d\n",sizeof(i));
 printf ("La longitud de double j= %d\n",sizeof(j));
 printf ("La longitud de long double k= %d\n",sizeof(k));

 printf("\nValores minimos y maximos de cada uno de los tipos\n\n");

 printf ("Minimo y maximo de char a= %d\t\t%d\n",CHAR_MIN,CHAR_MAX);
 printf ("Minimo y maximo de short int b=
%d\t\t%d\n",SHRT_MIN,SHRT_MAX);
 printf ("Minimo y maximo de int c= %d\t\t%d\n",INT_MIN,INT_MAX);
 printf ("Minimo y maximo de long int d=
%d\t\t%d\n",LONG_MIN,LONG_MAX);
 printf ("Maximo de unsigned char e= %d\n",UCHAR_MAX);
 printf ("Maximo de unsigned short int f= %d\n",USHRT_MAX);
 printf ("Maximo de unsigned int g= %d\n",UINT_MAX);
 printf ("Maximo de unsigned long int h= %d\n",ULONG_MAX);
 printf ("Minimo y maximo de float i= %d\t\t%d\n",FLT_MIN,FLT_MAX);
 printf ("Minimo y maximo de double j= %d\t\t%d\n",DBL_MIN,DBL_MAX);
 printf ("Minimo y maximo de long double k=
%d\t\t%d\n",LDBL_MIN,LDBL_MAX);
}
```

5. Escriba un programa que te imprima un mensaje de presentación, te pregunte como te llamas y te salude.

```
/*  Saludo   */

#include <stdio.h>

main()
{
char nombre[20];
printf("Hola, me llamo Ordenador, y tu?\n");
scanf("%s",&nombre[0]);
printf("Me alegro de conocerte %s\n",nombre);
}
```

6. Escriba el ejercicio de presentación (ejercicio número 5) y que te salude con nombre y apellidos.

```
/* Saludo con Nombre y apellido (utilizacion de gets en lugar de
scanf) */

#include <stdio.h>
void contestar (char []);
main()
{
char nombre[20];
printf("Hola, me llamo Iagoba, y tu?\n");
gets(nombre);
contestar(nombre);
}
void contestar(char *nombre)
{
printf("Me alegro de conocerte %s\n",nombre);
}
```

7. Escriba un programa que calcule el área de un triángulo rectángulo, dada la altura y la base.

```
  /*   Area de un triangulo rectangulo    */

#include <stdio.h>

main()
{
 float altura, base;
 double area;
 printf("\nBase del triangulo = ");
 scanf("%f",&base);
 printf("\nAltura del triangulo = ");
 scanf("%f",&altura);
 area= 0.5 * (double) altura * base;
 printf("\nArea = %g\n",area);
}
```

8. Escriba un programa que halle el menor de dos números pedidos al usuario.

```
  /* Calculo del menor de dos numeros dados */

#include <stdio.h>

main()
{
 int menor, numero1, numero2;
 printf("\nEscriba el primer numero y pulso INTRO: ");
 scanf("%d",&numero1);
 printf("\nEscriba el segundo numero y pulso INTRO: ");
 scanf("%d",&numero2);
 if (numero1 < numero2) menor=numero1;
 else menor=numero2;

 /* la sentencia if-else es equivalente a:
 menor=numero1<numero2 ? numero1 : numero2;  */

 printf("\nEl menor de %d y %d es %d\n",numero1, numero2, menor);
}
```

9. Escriba un programa que pida al usuario un carácter y un número de repeticiones. Luego imprima el carácter el número de veces especificado.

```
  /* Repetir un caracter un numero de veces  */

main()
{
 char ch;
 int num_rep;
 printf("\nEscriba el caracter a repetir: ");
 scanf("%c",&ch);
 printf("\nEscriba el numero de repeticiones: ");
 scanf("%d",&num_rep);
 while (num_rep>0)
 {
  printf("%c",ch);
  --num_rep;
 }
 printf("\n");
}
```

10. Escriba un programa que imprima una tabla con las cuatro primeras potencias de los números 1 a 10.

```
  /*  Imprime la tabla de las primera 4 potencias del 1 al 10 */

#include <stdio.h>

main()
{
 int n;
```

```
 puts("numero\t exp2\t exp3\t exp4");
 puts("------\t-----\t-----\t-----");
 for (n=0; n<=10; ++n)
printf("%2d\t%5d\t%5d\t%5d\n",n,n*n,n*n*n,n*n*n*n);
}
```

11. Escriba un programa que calcule el factorial de un número.

```
  /* Factorial de un numero  */

#include <stdio.h>

main()
{
 int i, numero, factorial=1;
 printf("\nEscriba un numero entero para calcular su factorial: ");
 scanf("%d",&numero);
 for (i=numero; i>1; --i) factorial *= i;
 printf("\n%d! = %d\n",numero,factorial);
}
```

12. Escriba un programa que muestre una tabla de conversiones de temperatura de grados Farenheit a grados Celsius, de 0ºF a 300ºF de 20 en 20ºF.

```
/* Sistema de temperaturas  */

#include <stdio.h>

main()
{
int i;
float c;
for (i=0;i<=300;i+=20)
  {
  c=(5./9)*(i-32);
  printf("%3d grados Farenheit equivalen a %4.4f grados
Celsius\n",i,c);
  }
}
```

13. Escriba un programa que muestre la tabla ASCII.

```
/*  Tabla ASCII   */

#include <stdio.h>
#define INI 33
#define FIL 15
#define COL 20

main()
{
 int k, kk, i;
 for (k=1; k<FIL; k++)
 {
```

```
  if (k==1) printf ("\t\t\t\t TABLA ASCII\n\n");
  else printf("\n");
  for (i=1; i<=COL; i++)
  {
   kk=INI+(k-1)*COL+i;
   if (kk>255) break;
   printf("   %c",kk);
  }
  printf("\n");
  for (i=1; i<=COL; i++)
  {
   kk=INI+(k-1)*COL+i;
   if (kk>255) return(1);
   if (kk<100) printf("   %d",kk);
  }
 }
}
```

14. Escriba un programa que dándole el importe exacto de una cantidad te indica el mínimo número de monedas que podrías tener. Las monedas son de 1, 5, 10, 25, 50, 100, 200 y 500 pesetas.

```
  /*  Cambio optimo de monedas  */

#include <stdio.h>
#define LIM 8
int monedas[LIM]= {500, 200, 100, 50, 25, 10, 5, 1};

main()
{
 int num, cantidad, numonedas;
 printf ("Introduzca el importe exacto: ");
 scanf ("%d", &cantidad);
 printf ("El cambio optimo es el siguiente: \n");
 for (num=0; num<LIM; num++)
 {
  numonedas=cantidad/monedas[num];
  if (numonedas != 0) printf ("%d de %d.\n", numonedas, monedas[num]);
  cantidad= cantidad % monedas[num];
 }
}
```

15. Escriba un programa que ordene un vector (de dimensión máxima 15) por el método de la burbuja.

```
/*   Algoritmo de la burbuja (ordenacion de un vector)  */

#include <stdio.h>
#define LIM 15

main ()
{

 int i=0, j, tamanyo, aux;
 int lista[LIM];
```

```
 printf ("Longitud del vector a ordenar ");
 scanf ("%d",&tamanyo);
 if (tamanyo > LIM)
 {
  printf("El limite del vector se ha excedido");
  printf(", por favor cambie el limite en el codigo\n");
  return;
 }
 while (i<tamanyo)
 {
  i++;
  printf("Numero %d: ",i);
  scanf("%d",&lista[i-1]);
 }
 printf("\n El vector introducido es el siguiente:\n");
 for (i=0; i<tamanyo; i++) printf("%d ",lista[i]);
 for (i=0; i<tamanyo-1; i++)
 {
  for (j=i+1; j<tamanyo; j++)
  {
   if (lista[i] > lista[j])
   {
    aux=lista[i];
    lista[i]=lista[j];
    lista[j]=aux;
   }
  }
 }
 printf("\n El vector ordenado es el siguiente:\n");
 for (i=0; i<tamanyo; i++) printf("%d ",lista[i]);
 printf("\n");
}
```

16. Escriba un programa que compruebe la diferencia entre una variable global, una variable local y un argumento de función.

```
  /* Ejemplo de variable global, local y argumento de funcion  */

#include <stdio.h>
int vglobal=100;  /* Se ve en todo el fichero y debe ir con extern
dentro
                de la declaracion en las funciones  */
void suma1(int x);
void suma2(int x);

main()
{
 extern int vglobal;
 int vlocal=200;
 printf("\nLa variable vglobal en main vale = %d\n", vglobal);
 printf("\nLa variable vlocal en main vale = %d\n", vlocal);
 suma1(vlocal);

/* En salida no se ha modificado su valor pese a que en suma1
   se hagan operaciones con vlocal (argumento) */

 printf("\nLa variable vglobal despues de suma1 vale = %d\n",
vglobal);
 printf("\nLa variable vlocal despues de suma1 vale = %d\n", vlocal);
```

```
 suma2(vlocal);
 printf("\nLa variable vglobal despues de suma2 vale = %d\n",
vglobal);
 printf("\nLa variable vlocal despues de suma2 vale = %d\n", vlocal);
}

void suma1(int x)
{
 extern int vglobal;
 ++x;
 ++vglobal;
 printf("\nLa variable vglobal dentro de suma1 vale = %d\n", vglobal);
 printf("\nEl argumento (vlocal) dentro de suma1 vale = %d\n", x);
 return;
}

void suma2(int x)
{
 extern int vglobal;
 ++x;
 ++vglobal;
 printf("\nLa variable vglobal dentro de suma2 vale = %d\n", vglobal);
 printf("\nEl argumento (vlocal) dentro de suma2 vale = %d\n", x);
 return;
}
```

17. Hacer el ejercicio de presentación (ejercicio 5) con una llamada a una función.

```
   /* Saludo con funcion  */

#include <stdio.h>
void contestar (char []);
main()
{
char nombre[20];
printf("Hola, me llamo Ordenador, y tu?\n");
scanf("%s",nombre);
contestar(nombre);
}
void contestar(char *nombre)
{
printf("Me alegro de conocerte %s\n",nombre);
}
```

18. Escriba un programa que calcule el cuadrado de un número mediante una macro.

```
  /* Calculo del cuadrado de un numero  */

#include <stdio.h>
#define CUAD(x) (x*x)  /* Definicion de macros */

main()
{
 float a;
 printf("\nEscriba un numero: ");
```

```
 scanf("%f",&a);
 printf("\nSu cuadrado es: %f\n",CUAD(a));
}
```

19. Escriba un programa que imprima un mensaje rodeado por un borde, utilizando funciones para dibujar los elementos del borde.

```
  /*  Mensaje rodeado por un borde   */

#include <stdio.h>
#define ANCHO 77
#define FILAS 4

void linea();
void lados();

main()
{
 printf("\n\n\n\n");
 linea();
 lados();
 printf("   *\t\t\t\t Bienvenido a C \t\t\t\t*\n");
 lados();
 linea();
 printf("\n\n\n\n");
}

void linea()
{
 int x;
 printf("   ");
 for (x=0; x<=ANCHO; ++x) putchar('*');
 putchar('\n');
}

void lados()
{
 int y;
 for (y=0; y<=FILAS; ++y) printf("   *\t\t\t\t\t\t\t\t\t\t*\n");
}
```

20. Escriba un programa que imprima una tabla con las áreas del círculo y de la esfera para un radio en el rango de 0 hasta 2 en incrementos de 0.2.

```
/* Area del circulo y de la esfera para un radio desde 0 hasta 2
   con incremento de 0.2  */

#include <stdio.h>
#define MAX_RADIO 2.0
#define PI 3.141592

double AreaCirculo(double radio);
double AreaEsfera(double radio);

main()
```

```
{
 double radio;
 puts("\n\t=== Tabla de Areas ===\n");
 puts("\tRadio\tCirculo\tEsfera");
 puts("\t-----\t-------\t------");
 for (radio=0.0; radio <= (double) MAX_RADIO; radio +=0.2)

printf("\t%6.2lf\t%6.3lf\t%6.3lf\n",radio,AreaCirculo(radio),AreaEsfer
a(radio));
}

double AreaCirculo(double radio)
{
 double area;
 area = PI * (radio * radio);
 return(area);
}

double AreaEsfera(double radio)
{
 double area;
 area = 4.00 * PI * (radio * radio);
 return(area);
}
```

21. Escriba un programa con una función que borre la pantalla emitiendo una serie de caracteres de salto de línea.

```
    /*  Limpia Pantalla  */

#include <stdio.h>

void LimpiaPantalla(int n);

main()
{
 int numlin=30;
 LimpiaPantalla(numlin);
}

void LimpiaPantalla(int n)
{
 while (n-- >0) putchar('\n');
}
```

22. Escriba un programa que calcule la potencia entera de un número entero de forma iterativa y de forma recursiva.

```
  /*  Potencia entera. Por iteracion y por recursividad  */

#include <stdio.h>

long poten_iter(int base, int expo);
long poten_recu(int base, int expo);
```

```
main()
{
 long potencia;
 int b, e;
 printf("\nEscriba una base entera para la operacion b= ");
 scanf("%d",&b);
 printf("\nEscriba un exponente entero para la operacion e= ");
 scanf("%d",&e);

 /*  Forma iterativa  */

 potencia=poten_iter(b,e);
 printf("\nEl resultado de la potencia en forma iterativa
=%ld\n",potencia);

 /*  Forma recursiva  */

 potencia=poten_recu(b,e);
 printf("\nEl resultado de la potencia en forma recursiva
=%ld\n",potencia);

}

long poten_iter(int base, int expo)
{
 long p=1;
 int i;
 for (i=1; i<=expo; i++) p*=base;
 return(p);
}

long poten_recu(int base, int expo)
{
 long p;

 /* Condiciones de finalizacion de la recursividad  */

 if (base==0) return(0);
 if (expo==0) return(1);

 /* Desarrollo de la recursividad  */

 p=base*poten_recu(base,expo-1);
 return(p);
}
```

23. Escriba un programa que inicialice un array de enteros. Calcule e imprima su suma, media, mínimo y máximo.

```
  /* Calculo de la suma, media, minimo y maximo de un vector de
enteros */

#include <stdio.h>
#define MIN(a,b) ((a < b) ? a : b)
#define MAX(a,b) ((a > b) ? a : b)

main()
{
```

```
 int valores[] = {10,1,3,4,15,6,7,8,9,10};
 int i, tam, suma=0, minimo=valores[0], maximo=valores[0];
 float media;
 tam=sizeof(valores)/sizeof(int);
 for (i=0; i<tam;++i)
 {
  printf(" %d ",valores[i]);
  minimo=MIN(minimo,valores[i]);
  maximo=MAX(maximo,valores[i]);
  suma+=valores[i];
 }
 media=(float) suma / tam;

 printf("\nSuma= %d; Media= %f\nMinimo= %d; Maximo=
%d\n",suma,media,minimo,
        maximo);
}
```

24. Escriba un programa que contenga dos arrays y utilizando punteros genere un tercer array con la suma de los otros dos.

```
  /*  Suma de arrays por punteros  */

#include <stdio.h>
#define FILAS 4
#define COLS 5

main()
{
 int j,k;
 int arr1 [FILAS] [COLS] = { {13,15,17,19,21},
                        {20,22,24,26,28},
                        {31,33,35,37,39},
                        {40,42,44,46,48} };
 int arr2 [FILAS] [COLS] = { {10,11,12,13,14},
                        {15,16,17,18,19},
                        {20,21,22,23,24},
                        {25,26,27,28,29} };
 int arr3 [FILAS] [COLS];
 for (j=0; j<FILAS; j++)
 {
  for (k=0; k<COLS; k++)
  {
   *(*(arr3+j)+k) = *(*(arr1+j)+k) + *(*(arr2+j)+k);
   printf("%d ", *(*(arr3+j)+k));
  }
  printf("\n");
 }
}
```

25. Escriba un programa que utilice una función para intercambiar dos valores. Hacerlo para dos funciones, una con llamada por valor y otra por referencia.

```
   /* Intercambio de valores usando llamada por valor y por referencia
*/
```

```
#include <stdio.h>

void IntercambioValor(int v1, int v2);           /*  Utiliza variables
*/
void IntercambioReferencia(int *pv1, int *pv2);  /*  Utiliza punteros
*/

main()
{
 int val1=10, val2=20;
 printf("Valores iniciales:\n\tval1 = %d; val2 = %d\n",val1,val2);

  /*  Paso de parametros por valor  */

 IntercambioValor(val1, val2);
 printf("\nPaso de parametros por valor:\n\tval1 = %d; val2 =
%d\n",val1,val2);

  /*  Paso de parametros por referencia  */

 IntercambioReferencia(&val1, &val2);
 printf("\nPaso de Parametros por Referencia:\n\tval1 = %d; val2 =
%d\n",
         val1,val2);
}

void IntercambioValor(int v1, int v2)
{
 int tmp;
 tmp = v1;
 v1 = v2;
 v2 = tmp;
}

void IntercambioReferencia(int *pv1, int *pv2)
{
 int tmp;
 tmp = *pv1;
 *pv1 = *pv2;
 *pv2 = tmp;
}
```

26. Escriba un programa que pida una cadena por el teclado y la imprima después de convertir el primer carácter en mayúscula y el resto en minúsculas.

```
  /* Primera letra mayuscula y demas minuscula  */

#include <stdio.h>
#include <ctype.h>
#define MAXCADENA 20

char *ConversionLetra(char *cadena);

main()
{
 char tmp[MAXCADENA+1];  /* Se suma 1 para el byte NUL */
```

```
 printf("\nEscriba una cadena (de %d caracteres maximo) y teclee
INTRO:\n",
         MAXCADENA);
 gets(tmp);
 printf("\n%s\n", ConversionLetra(tmp));
}

char *ConversionLetra(char *cadena)
{
 char *pc;
 pc=cadena;
 *pc=toupper(*pc);
 ++pc;
 while (*pc != '\0')
 {
  *pc = tolower(*pc);
  ++pc;
 }
 return(cadena);
}
```

27. Escriba un programa que lea una cadena desde el teclado y cuente el número de caracteres de tres categorías: letras (a-z y A-Z), dígitos (0-9) y otros caracteres. Utilice las funciones isdigit() e isalpha() definidas en la librería ctype.h.

```
  /*  Contar letras de (a-z y A-Z), digitos (0-9) y otros caracteres
        Funciones isdigit() e isalpha()  */

#include <stdio.h>
#include <ctype.h>
#define MAXCAD 80

main()
{
 char linea[MAXCAD], *pc=linea;
 int digitos=0, letras=0, otros=0;
 printf("\nEscriba una cadena (<%d caracteres):\n", MAXCAD);
 gets(linea);
 while (*pc != '\0')
 {
  if (isdigit(*pc)) ++digitos;
  else if (isalpha(*pc)) ++letras;
       else ++otros;
  ++pc;
 }
 printf("\n\tDigitos = %d\n\tLetras = %d\n\tOtros =
%d\n",digitos,letras,otros);
}
```

28. Escriba un programa que lea una cadena y busque un carácter en ella.

```
/*   Buscar un caracter en una cadena  */

#include <stdio.h>
```

```
#include <string.h>

main()
{
 char caract, cadena[80], *ptr;
 printf("Introduzca la cadena donde se va a buscar:\n");
 gets(cadena);
 printf("Escriba el caracter a buscar:\n");
 caract=getchar();
 ptr=strchr(cadena, caract);
 if (ptr==0) printf("El caracter %c no se encuentra en la
cadena.\n",caract);
 else printf("La posicion del caracter es %d.\n", ptr-cadena+1);
}
```

29. Escriba un programa que inserte un carácter en una determinada posición de una cadena.

```
  /* Insertar caracter en una posicion en una cadena  */

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

void insertar(char *cadena, char ca, int n);

main()
{
 char car, cadena[81];
 int posicion;
 printf("\nEscriba una cadena [Intro], caracter [Intro], posicion\n");
 gets(cadena);
 scanf("%c\n%d",&car,&posicion);
 insertar(cadena,car,posicion);
 puts(cadena);
}

void insertar(char *cadena, char ca, int n)
{
 char temporal[81];
 strcpy(temporal,&cadena[n-1]);
 cadena[n-1]=ca;
 strcpy(&cadena[n], temporal);
}
```

30. Escriba un programa que copie un fichero a otro.

```
   /* Copia de un fichero a otro con otro nombre  */

#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
 FILE *origen, *destino;
```

```
 char car, aux1, aux2;
 int i=0, igu=1;
 if (argc != 3)
 {
  printf ("Error. La instruccion es:\n copia fichero_origen
fichero_destino\n");
  return;
 }
 do
 {
  aux1=*(argv[1]+i);
  aux2=*(argv[2]+i);
  if (aux1 != aux2)
  {
   igu=0;
   break;
  }
  i++;
 } while (aux1 != '\0' || aux2 != '\0');
 if (igu == 1)
 {
  printf ("El fichero destino es el mismo que el fichero origen\n");
  return;
 }
 origen=fopen(argv[1],"r");
 destino=fopen(argv[2],"w");
 if (origen==NULL || destino ==NULL)
 {
  printf ("El fichero de origen no existe o no hay espacio en el
disco\n");
  return;
 }
 while ( (car=getc(origen)) != EOF ) putc(car,destino);
 fclose(origen);
 fclose(destino);
}
```

31. Escriba un programa que lea y muestre en pantalla el contenido de un fichero.

```
  /*  Muestra en pantalla el contenido de un fichero de texto */

#include <stdio.h>
#define MAXVIA 64
#define MAXLINEA 256

main()
{
 int car;            /* caracter de entrada  */
 FILE *pf;           /* puntero a fichero  */
 char via_acceso[MAXVIA];  /* buffer para el nombre del fichero  */
 char linea[MAXLINEA];     /* buffer de linea para fgets()  */
 printf("\nNombre de fichero: ");
 gets(via_acceso);
 if (*via_acceso == '\0') return; /* No se ha introducido ningun
nombre */
 pf = fopen(via_acceso, "r");
 if (pf == NULL)
 {
  printf("\nEL fichero no existe o la ruta no es valida.\n");
```

```
  return;
 }
 while (fgets(linea,MAXLINEA,pf) != NULL) fputs(linea,stdout);
 fclose(pf);
}
```

15 BIBLIOGRAFIA

DIGITAL 1994. **DEC C. User's Guide for OpenVMS Systems**. Digital Equipment Corporation.

DIGITAL. 1996. **Digital UNIX DEC C Language Reference**. Digital Equipment Corporation.

HANSEN, A. 1988. **¡Aprenda C ya!**. Anaya.

KERNIGHAN, B. y RITCHIE, D. 1978. **The C Programming Language**. Prentice-Hall Software Series.

LAFORE, R. 1990. **Programación en MICROSOFT C. Introducción y técnicas avanzadas de programación**. Anaya.