4. Funciones y Procedimientos en Pascal

Un programa en Pascal consiste de uno o más módulos, los cuales a su vez también pueden estar constituídos por otros módulos, y así sucesivamente. Estos módulos serán llamados, en términos generales, SUBPROGRAMAS, y en particular ***PROCEDIMIENTOS y FUNCIONES***.

Un subprograma es un bloque de programa que realiza una tarea determinada, y que al llamársele o invocársele puede necesitar que se le pasen PARAMETROS.

Los parámetros son identificadores que proveen un mecanismo para pasar información hacia los subprogramas invocados.

Los parámetros que se utilizan en la declaración de subprogramas se llaman PARAMETROS FORMALES.
Un subprograma puede ser invocado desde cualquier punto del programa, por medio de una instrucción INVOCADORA, la cual puede contener una lista de parámetros llamados PARAMETROS ACTUALES.

Procedimientos

Un procedimiento es un subprograma que realiza alguna de las tareas del programa, y que no devuelve ningún valor al subprograma que lo invocó.

Un procedimiento está compuesto de un grupo de sentencias a las que se asigna un nombre (*identificador*) y constituye una unidad del programa. La tarea asignada al procedimiento se ejecuta siempre que Pascal encuentra el nombre del procedimiento.

En Turbo Pascal resulta obligatorio declarar los procedimientos antes de ser referenciados en el cuerpo del programa.

. Declaración de un procedimiento
La sintaxis para *declarar* un procedimiento es :

Formato 1 :

```
procedure nombre_procedimiento;
 declaraciones locales
begin
 cuerpo del procedimiento
end;
```

Formato 2:

```
procedure nombre_procedimiento(parámetros formales);
 declaraciones locales
begin
 cuerpo del procedimiento
end;
```

La existencia de parámetros_formales dependerá de la naturaleza del procedimiento, esto es, de la tarea que va a realizar y de la forma en que lo ha estructurado su creador.

Invocación al procedimiento

Para *invocar* a un procedimiento, la sintaxis es :

<nombre_de_procedimiento> (*parámetros_actuales*) ;

donde la existencia de *parámetros_actuales* dependerá de que en la declaración del procedimiento se hayan utilizado parámetros formales.

Para ejemplificar el uso de procedimientos, diseñaremos un programa que resuelva el problema de SUMAR y MULTIPLICAR MATRICES.

PSEUDOCODIGO

```
{PROGRAMA PARA SUMAR Y MULTIPLICAR MATRICES}
INICIO
 IMPRIME encabezado.
 LEE las dimensiones de las matrices A y B.
 SI las matrices no son compatibles para la suma,
 ENTONCES
  IMPRIME mensaje_1.
 SI las matrices no son compatibles para la mult.,
 ENTONCES
  IMPRIME mensaje_2.
 SI son compatibles para la suma o para la mult. ,
 ENTONCES
  INICIO
   LEE las matrices A y B.
   IMPRIME las matrices A y B.
   SI son compatibles para la suma, ENTONCES
    INICIO
     SUMA las matrices A y B.
     IMPRIME la matriz resultado C.
    FIN
   SI son compatibles para la multiplicacion, ENTONCES
    INICIO
     MULTIPLICA las matrices A y B.
    IMPRIME la matriz resultado D.
    FIN
  FIN
FIN.
```

CODIFICACION :

```
Program Opera_matrices;
 {Programa para sumar y multiplicar matrices de orden
  hasta de dim_max por dim_max }
Uses Crt;
Const
  dim_max = 10;
Type
  mat = array [1..dim_max , 1..dim_max] of real;
Var
mat_a,mat_b,mat_c       : mat;
bandera_suma,bandera_mult:boolean;
ren_a,ren_b,col_a,col_b  :integer;
procedure inicio;
Var
  contador : integer;
begin ClrScr;
  gotoxy(23,2);
  WriteLn('SUMA Y MULTIPLICACION DE MATRICES');
  for contador := 1 to 80 do
   Write('=');
```

```
  WriteLn
end;
{Lee las dimensiones de las matrices}
procedure dim ;
begin
  WriteLn('DIMENSIONES DE LA MATRIZ A');
  WriteLn;
  Write('Número de renglones ==> ');
  ReadLn(ren_a);
  Write('Numero de columnas  ==> ');
  ReadLn(col_a);
  WriteLn;
  WriteLn;
  WriteLn('DIMENSIONES DE LA MATRIZ B');
  WriteLn;
  Write('Número de renglones ==> ');
  ReadLn(ren_b);
  Write('Número de columnas  ==> ');
  ReadLn(col_b)
end;
{Verifica la compatibilidad para la suma}
procedure compat_suma (ren_f_a,ren_f_b,col_f_a,col_f_b:integer;
                       Var bandera_f:boolean);
begin
  if ((ren_f_a <> ren_f_b) or (col_f_a <> col_f_b)) then
    begin
      WriteLn;
      WriteLn('Las matrices A y B son incompatibles para la suma');
      bandera_f :=false
    end
  else
     bandera_f :=true
end;
{Verifica la compatibilidad para la multiplicación}
procedure compat_mult(ren_f_a,ren_f_b,col_f_a,col_f_b:integer;
                      Var bandera_f:boolean);
begin
  if col_f_a <> ren_f_b then
    begin
      WriteLn;
      WriteLn('Las matrices A y B son icompatibles para la multiplicación');
      bandera_f := false
    end
  else
     bandera_f := true
end;
{Lee una matriz}
procedure lee(nmat:char;Var mat_f:mat;ren_max,col_max:integer);
Var
  ren,col : integer;
begin
  WriteLn;
  WriteLn('ELEMENTOS DE LA MATRIZ : ',nmat);
  WriteLn;
  for ren := 1 to ren_max do
    for col := 1 to col_max do
      begin
        Write('Elemento [ ',ren,',',col,'] = ');
        ReadLn(mat_f[ren,col])
```

```
    end
end;
{Suma dos matrices}
procedure suma( mat_f_a,mat_f_b:mat;Var mat_f_c:mat;
            ren_f, col_f :integer);
Var
  ren,col : integer;
begin
  WriteLn;
  WriteLn('La suma de A y B es :');
  for ren := 1 to ren_f do
   for col := 1 to col_f do
     mat_f_c[ren,col] := mat_f_a[ren,col] + mat_f_b[ren,col]
end;
{Multiplica dos matrices}
procedure multiplica( mat_f_a, mat_f_b: mat ;Var mat_f_c : mat ;
                  ren_f_a, col_f_a, col_f_b :integer);
Var
  ren, acol, bcol : integer ;
  acum          : real ;
begin
  WriteLn;
  WriteLn('El producto de A y B es :');
  for ren := 1 to ren_f_a do
    for bcol := 1 to col_f_b do
    begin
      acum := 0.0 ;
      for acol := 1 to col_f_a do
        acum := acum + mat_f_a[ren,acol] * mat_f_b[acol,bcol];
      mat_f_c[ren,bcol] := acum
    end
end;
{Imprime una matriz}
procedure imprime(nmat : char ; mat_f : mat ;
              ren_f, col_f : integer) ;
Var
  ren, col : integer;
begin WriteLn;
  WriteLn('MATRIZ ',nmat);
  for ren := 1 to ren_f do
    for col := 1 to col_f do begin
      Write(mat_f[ren,col]:6:1,' ');
       WriteLn
    end;
  WriteLn;
  Write('Oprima una tecla para continuar..... ');
  ReadKey;
  WriteLn
end;
{Módulo Principal}
begin
 inicio;
 dim;
 compat_suma(ren_a,ren_b,col_a,col_b,bandera_suma);
 compat_mult(ren_a,ren_b,col_a,col_b,bandera_mult);
 if bandera_suma or bandera_mult  then
   begin
```

```
    lee('A',mat_a,ren_a,col_a);
    lee('B',mat_b,ren_b,col_b);
    imprime('A',mat_a,ren_a,col_a);
    imprime('B',mat_b,ren_b,col_b);
    if bandera_suma then
      begin
        suma(mat_a,mat_b,mat_c,ren_a,col_a);
        imprime('C',mat_c,ren_a,col_b)
      end;
    if bandera_mult then begin
      multiplica(mat_a,mat_b,mat_c,ren_a,col_a,col_b);
      imprime('D', mat_c, ren_a, col_b)
      end
  end
end.
```

Observe que el MÓDULO PRINCIPAL está formado por un bloque de instrucciones que invocan a procedimientos.

Funciones

La diferencia principal entre un procedimiento y una función es que el identificador de la función asume un valor, y cuando la función termina su tarea, devuelve ese valor al módulo que la invocó; mientras que el procedimiento no devuelve ningún valor.

Puesto que el nombre de la función toma un valor, dicho nombre debe tener asociado un tipo de dato.

Declaración de funciones
La declaración de una función tiene la siguiente forma :

```
function Nombre (p1,p2,...):tipo
 {declaraciones locales y subprogramas}
begin
  <cuerpo de la función>
Nombre := valor de la función
end;
```

p1,p2,... lista de parámetros formales
tipo tipo de dato del resultado que devuelve la función

Ejemplos :

```
function verifica : boolean ;
{ Función sin parámetros formales}
function cambia(Var valor_1, valor_2: real):real;
function potencia( base, exponente : real ):real;
```

Invocación de funciones
Las funciones se invocan de la siguiente manera :

<nombre_función> (*parámetros_locales*) ;

donde :
parámetros_locales es una lista de variables y/o constantes separadas por comas. La existencia de parámetros_locales dependerá de que existan *parámetros formales* en la declaración de la función.

Por ejemplo, resolvamos el problema de cacular la raíz cuadrada de un número, utilizando el algoritmo de Newton:

x(i+1) = x(i) + 0.5 (a/x(i) -x(i))

La codificación del programa sería :

```
Program Raiz_cuadrada; {El siguiente programa calcula la raíz  cuadrada de un número}
Uses Crt;
Var
  raiz, numero : real;
{Declaración de la función raiz_cuad}
function raiz_cuad( a : real ) : real ;
Var
  c,x : real ;
begin
  x := 1E-9 ;
  c := 1.0 ;
  while Abs (c-x)>1E-9 do
    begin
      c := x ;
      x := x + 0.5 * ( a/x -x )
    end;
   raiz_cuad := x
    {El resultado se asigna a nombre_función}
end;
begin
ClrScr;
  Write('La raíz cuadrada de : ');
  ReadLn(numero) ;
  raiz:=raiz_cuad(numero);
  {Invoca a la función raiz_cuad}
  WriteLn ;
  WriteLn('Es igual a : ',raiz:6:8);
  ReadKey;
  ClrScr
end.
```

Ambito de variables

Las variables se clasifican en ***LOCALES y GLOBALES***. Una variable LOCAL es una variable declarada dentro de un subprograma, y el significado de dicha variable se limita a ese subprograma y a los módulos que éste contiene. Cuando otro subprograma utiliza el mismo nombre de variable, se crea una variable diferente en otra posición de la memoria. Por eso, si un subprograma asigna un valor a una de las variables locales, tal valor no es accesible a los otros subprogramas.

Cuando se desea que otros subprogramas tengan acceso al valor de una variable, ésta debe declararse como GLOBAL, lo cual se logra declarándola en el módulo que abarca a dichos subprogramas.

Para tener la seguridad de que una variable va a tener un alcance GLOBAL, conviene declararla en el MODULO PRINCIPAL.

Los conceptos anteriores se aclaran por medio de la siguiente figura :

Program anidado;

```
Const
    M=230;
Var
   i,j : real;

procedure A (Var i: real);
   Var
     r,s : boolean;

   function B(g : real):boolean;
      Var
        m,n : char;
      begin
          .
      end;

   begin
       B(5.0)
   end;
begin
    A(i)
end;
```

Referencias de identificadores válidos :

Bloque	Identificador	Significado de cada identificador
anidado	M	constante global
	i,,j	varibles globales
	A	procedimiento declarado en anidado
A	i	parámetros de A
	r,s	variables locales
	B	función local
	j	variable declarada en anidado
	B	función declarado en anidado
	M	constante global
B	g	parámetros de B
	m,n	variables locales
	r,s	variable declarada en A
	i	parámetro de A
	y	variable declarada en anidado
	A	procedimiento declarado en anidado
	B	función declarada en anidado
	M	constante global

Paso de parámetros

Al invocar a un subprograma se le pueden pasar parámetros, los cuales pueden consistir de valores de variables declaradas en el módulo invocador. El paso de tales parámetros puede hacerse de dos maneras :

- Por valor
- Por referencia

Paso por valor

El paso de parámetros por valor consiste en enviar una COPIA del valor de la variable al módulo invocado. De esta manera se asegura que el valor de la variable sólo puede ser modificado por el módulo que la declaró.

Si la palabra Var no aparece delante del *parámetro formal* en un procedimiento, Turbo Pascal supone que el parámetro formal es un *parámetro por valor*.

Ejemplo:

```
Program Suma_por_Valor;
 {El siguiente programa realiza
  la suma de dos numeros }
 Uses Crt; Var
 A,B,C:integer;

procedure suma(A,B,C :integer);
begin
 C := A + B
end;

begin
 ClrScr;C:=10; A:=10; B:=10;

 suma(,B,C); WriteLn(A,',',B,',',C); ReadKey;
```

```
ClrScr end.
```

El resultado de la ejecución del programa sería :10,10,10

El valor de C no se modifica puesto que es un parámetro por valor.

Paso por referencia

En el caso de que se requiera que el valor de una variable sea modificado por el módulo invocado, debe hacerse el paso de parámetro por referencia, por medio del cual el módulo invocado tiene acceso a la dirección en que se guarda el valor a modificar.

Para aclarar los conceptos de *paso por valor* y *paso por referencia*, analicemos el programa de suma y multiplicación de matrices dado en la sección anterior.

Observamos que las invocaciones a subprogramas son similares en los casos de paso por valor y paso por referencia.

Por ejemplo, las invocaciones a los procedimientos imprime y suma quedan de la siguiente manera :

```
imprime('C',mat_c,ren_a,col_b);
suma(mat_a,mat_b,mat_c,ren_a,
col_a); y sus respectivas
declaraciones son :
procedure imprime(nmat:char;mat_f:mat;ren_f,col_f:integer);
procedure suma(mat_f_a,mat_f_b:mat;Var mat_f_c:mat;ren_f,col_f:integer);
```

Vemos que en la declaración del procedimiento suma existe la parte: Var mat_f_c : mat la cual significa lo siguiente :

"La variable mat_f_c contiene la dirección de la variable correspondiente en la invocación (mat_c) , y es de tipo mat "

Esto significa que el paso del parámetro mat_c se hizo *por referencia*, y que el procedimiento invocado (suma) puede modificar el valor de mat_c . Ejemplo:

```
Program Suma_por_Referencia;  {El siguiente programa realiza la suma de dos
números } Uses Crt;
Var
  A,B,C:integer;
procedure suma(A,B:integer;Var C:integer);
begin
  C := A + B
  end; begin ClrScr;
  C:=10;
  A:=10; B:=10; suma(A,B,C);
  WriteLn(A,',',B,',',C); ReadKey;
  ClrScr end.
```

El resultado de la ejecución del programa sería :10,10,20

El valor de C se modifica puesto que es un parámetro por referencia.

Nota : En Turbo Pascal no se permite el paso de procedimientos y funciones como parámetros

4. Funciones y Procedimientos en Lenguaje C

Como se explicó en la Sección 1.3, una ***función*** es una parte de código independiente del programa principal y de otras funciones, que puede ser llamada enviándole unos datos (o sin enviarle nada), para que realice una determinada tarea y/o proporcione unos resultados. Las funciones son una parte muy importante del lenguaje C. En los apartados siguientes se describen los aspectos más importantes de las funciones.

Utilidad de las funciones

Parte esencial del correcto diseño de un programa de ordenador es su *modularidad*, esto es su división en partes más pequeñas de finalidad muy concreta. En C estas partes de código reciben el nombre de ***funciones***. Las funciones facilitan el desarrollo y mantenimiento de los programas, evitan errores, y ahorran memoria y trabajo innecesario. Una misma función puede ser utilizada por diferentes programas, y por tanto no es necesario reescribirla. Además, una función es una parte de código independiente del programa principal y de otras funciones, manteniendo una gran independencia entre las variables respectivas, y evitando errores y otros efectos colaterales de las modificaciones que se introduzcan.

Mediante el uso de funciones se consigue un código limpio, claro y elegante. La adecuada división de un programa en funciones constituye un aspecto fundamental en el desarrollo de programas de cualquier tipo. Las funciones, ya compiladas, pueden guardarse en ***librerías***. Las librerías son conjuntos de funciones compiladas, normalmente con una finalidad análoga o relacionada, que se guardan bajo un determinado nombre listas para ser utilizadas por cualquier usuario.

Definición de una función

La ***definición de una función*** consiste en la definición del código necesario para que ésta realice las tareas para las que ha sido prevista. La definición de una función se debe realizar en alguno de los ficheros que forman parte del programa. La forma general de la definición de una función es la siguiente:

```
tipo_valor_de_retorno  nombre_funcion(lista de argumentos con tipos)
{
     declaración de variables y/o de otras funciones
     codigo ejecutable
     return (expresión);          // optativo
}
```

La primera línea recibe el nombre de ***encabezamiento*** (header) y el resto de la definición –encerrado entre llaves– es el ***cuerpo*** (***body***) de la función. Cada función puede disponer de sus propias variables, *declaradas* al comienzo de su código. Estas variables, por defecto, son de tipo ***auto***, es decir, sólo son visibles dentro del bloque en el que han sido definidas, se crean cada vez que se ejecuta la función y permanecen ocultas para el resto del programa. Si estas variables de definen como ***static***, conservan su valor entre distintas llamadas a la función. También pueden hacerse visibles a la función *variables globales* definidas en otro fichero (o en el mismo fichero, si la definición está por debajo de donde se utilizan), declarándolas con la palabra clave ***extern***.

El *código ejecutable* es el conjunto de instrucciones que deben ejecutarse cada vez que la función es llamada. La *lista de argumentos con tipos*, también llamados ***argumentos formales***, es una lista de declaraciones de variables, precedidas por su *tipo* correspondiente y separadas por comas (,). Los *argumentos formales* son la forma más natural y directa para que la función reciba valores desde el programa que la llama, correspondiéndose en número y tipo con otra lista de argumentos -los ***argumentos actuales***- en el programa que realiza la llamada a la función. Los *argumentos formales* son declarados en el encabezamiento de la función, pero no pueden ser inicializados en él.

Cuando una función es ejecutada, puede devolver al programa que la ha llamado un valor (el *valor de retorno*), cuyo tipo debe ser especificado en el encabezamiento de la función (si no se especifica, se supone por defecto el tipo *int*). Si no se desea que la función devuelva ningún valor, el *tipo del valor de retorno* deberá ser ***void***.

La sentencia ***return*** permite devolver el control al programa que llama. Puede haber varias sentencias *return* en una misma función. Si no hay ningún *return*, el control se devuelve cuando se llega al final del *cuerpo* de la función. Le palabra clave *return* puede ir seguida de una *expresión*, en cuyo caso ésta es evaluada y el valor resultante devuelto al programa que llama como *valor de retorno* (si hace falta, con una conversión previa al *tipo* declarado en el encabezamiento). Los paréntesis que engloban a la *expresión* que sigue a *return* son optativos.

El valor de retorno es un valor único: *no puede ser un vector o una matriz*, aunque sí un *puntero* a un vector o a una matriz. Sin embargo, *el valor de retorno sí puede ser una estructura*, que a su vez puede contener vectores y matrices como elementos miembros.

Como ejemplo supóngase que se va a calcular a menudo el *valor absoluto* de variables de tipo *double*. Una solución es definir una función que reciba como argumento el valor de la variable y devuelva ese valor absoluto como valor de retorno. La definición de esta función podría ser como sigue:

```
double valor_abs(double x)
{
     if (x < 0.0)
          return -x;
     else
          return x;
}
```

Declaración y llamada de una función

De la misma manera que en C es necesario declarar todas las variables, también *toda función debe ser declarada* antes de ser utilizada en la función o programa que realiza la llamada. De todas formas, ahora se verá que aquí hay una mayor flexibilidad que en el caso de las variables.

En C la declaración de una función se puede hacer de tres maneras:

a) Mediante una ***llamada*** a la función. En efecto, cuando una función es llamada sin que previamente haya sido *declarada* o *definida*, esa llamada sirve como declaración suponiendo *int* como tipo del valor de retorno, y el tipo de los argumentos actuales como tipo de los argumentos formales. Esta práctica es muy peligrosa (es fuente de numerosos errores) y debe ser evitada.

b) Mediante una ***definición*** previa de la función. Esta práctica es segura si la definición precede a la *llamada*, pero tiene el inconveniente de que si la definición se cambia de lugar, la propia llamada pasa a ser declaración como en el caso a).

c) Mediante una ***declaración*** explícita, previa a la *llamada*. Esta es la práctica más segura y la que hay que tratar de seguir siempre. La declaración de la función se hace mediante el ***prototipo*** de la función, bien fuera de cualquier bloque, bien en la parte de declaraciones de un bloque.

C++ es un poco más restrictivo que C, y obliga a declarar explícitamente una función antes de llamarla.

La forma general del *prototipo* de una función es la siguiente:

```
tipo_valor_de_retorno  nombre_funcion(lista de tipos de argumentos);
```

Esta forma general coincide sustancialmente con la primera línea de la definición -el encabezamiento-, con dos pequeñas diferencias: en vez de la lista de argumentos formales o parámetros, *en el prototipo basta incluir los tipos de dichos argumentos*. Se pueden incluir también identificadores a continuación de los tipos, pero son ignorados por el compilador. Además, una segunda diferencia es que el *prototipo* termina con un carácter (;). Cuando no hay argumentos formales, se pone entre los paréntesis la palabra ***void***, y se pone también ***void*** precediendo al nombre de la función cuando no hay valor de retorno.

Los *prototipos* permiten que el compilador realice correctamente la conversión del tipo del valor de retorno, y de los *argumentos actuales* a los tipos de los *argumentos formales*. *La declaración de las funciones mediante los prototipos suele hacerse al comienzo del fichero, después de los* **#define** *e* **#include**. En muchos casos –particularmente en programas grandes, con muchos ficheros y muchas funciones–, se puede crear un fichero (con la extensión **.h**) con todos los prototipos de las funciones utilizadas en un programa, e incluirlo con un **#include** en todos los ficheros en que se utilicen dichas funciones.

La ***llamada a una función*** se hace incluyendo su *nombre* en una expresión o sentencia del programa principal o de otra función. Este nombre debe ir seguido de una lista de *argumentos* separados por comas y encerrados entre paréntesis. A los argumentos incluidos en la llamada se les llama ***argumentos actuales***, y pueden ser no sólo variables y/o constantes, sino también *expresiones*. Cuando el programa que llama encuentra el nombre de la función, evalúa los *argumentos actuales* contenidos en la llamada, los convierte si es necesario al tipo de los *argumentos formales*, y ***pasa copias de dichos valores*** a la función junto con el control de la ejecución.

El número de *argumentos actuales* en la llamada a una función debe coincidir con el número de *argumentos formales* en la definición y en la declaración. Existe la posibilidad de definir funciones con un *número variable o indeterminado* de argumentos. Este número se concreta luego en el momento de llamarlas. Las funciones ***printf()*** y ***scanf()***, que se verán en la sección siguiente, son ejemplos de funciones con número variable de argumentos.

Cuando se llama a una función, después de realizar la conversión de los argumentos actuales, se ejecuta el código correspondiente a la función hasta que se llega a una sentencia ***return*** o al final del cuerpo de la función, y entonces se devuelve el control al programa que realizó la llamada, junto

con el *valor de retorno* si es que existe (convertido previamente al *tipo* especificado en el *prototipo*, si es necesario). Recuérdese que el valor de retorno puede ser un valor numérico, una dirección (un puntero), o una estructura, pero no una matriz o un vector.

La llamada a una función puede hacerse de muchas formas, dependiendo de qué clase de tarea realice la función. Si su papel fundamental es *calcular un valor de retorno* a partir de uno o más argumentos, lo más normal es que sea llamada incluyendo su nombre seguido de los argumentos actuales en una *expresión aritmética* o de otro tipo. En este caso, la llamada a la función hace el papel de un operando más de la expresión. Obsérvese cómo se llama a la función *seno* en el ejemplo siguiente:

```
a = d * sin(alpha) / 2.0;
```

En otros casos, *no existirá valor de retorno* y la llamada a la función se hará incluyendo en el programa una sentencia que contenga solamente el nombre de la función, siempre seguido por los argumentos actuales entre paréntesis y terminando con un carácter (;). Por ejemplo, la siguiente sentencia llama a una función que multiplica dos matrices (nxn) **A** y **B**, y almacena el resultado en otra matriz **C**. Obsérvese que en este caso no hay valor de retorno (un poco más adelante se trata con detalle la forma de pasar vectores y matrices como argumentos de una función):

```
prod_mat(n, A, B, C);
```

Hay también *casos intermedios* entre los dos anteriores, como sucede por ejemplo con las funciones de entrada/salida que se verán en la próxima sección. Dichas funciones tienen valor de retorno, relacionado de ordinario con el número de datos leídos o escritos sin errores, pero es muy frecuente que no se haga uso de dicho valor y que se llamen al modo de las funciones que no lo tienen.

La declaración y la llamada de la función **valor_abs()** antes definida, se podría realizar de la forma siguiente. Supóngase que se crea un fichero ***prueba.c*** con el siguiente contenido:

```
// fichero prueba.c
#include <stdio.h>
double valor_abs(double);            // declaración

void main (void)
{
     double z, y;

     y = -30.8;
     z = valor_abs(y) + y*y;         // llamada en una expresion
}
```

La función **valor_abs()** recibe un valor de tipo *double*. El valor de retorno de dicha función (el valor absoluto de **y**), es introducido en la expresión aritmética que calcula **z**.

La declaración `(double valor_abs(double))` no es estrictamente necesaria cuando la definición de la función está en el mismo archivo ***buscar.c*** que **main()**, y dicha definición está antes de la llamada.

Paso de argumentos por valor y por referencia

En la sección anterior se ha comentado que en la llamada a una función los ***argumentos actuales*** son evaluados y se pasan ***copias*** de estos valores a las variables que constituyen los ***argumentos***

formales de la función. Aunque los argumentos actuales sean variables y no expresiones, y haya una correspondencia biunívoca entre ambos tipos de argumentos, los cambios que la función realiza en los argumentos formales no se trasmiten a las variables del programa que la ha llamado, precisamente porque lo que la función ha recibido son *copias*. El modificar una copia no repercute en el original. A este mecanismo de paso de argumentos a una función se le llama ***paso por valor***. Considérese la siguiente función para permutar el valor de sus dos argumentos **x** e **y**:

```
void permutar(double x, double y)               // funcion incorrecta
{
     double temp;
     temp = x;
     x = y;
     y = temp;
}
```

La función anterior podría ser llamada y comprobada de la siguiente forma:

```
#include <stdio.h>

void main(void)
{
     double a=1.0, b=2.0;
     void permutar(double, double);

     printf("a = %lf, b = %lf\n", a, b);
     permutar(a, b);
     printf("a = %lf, b = %lf\n", a, b);
}
```

Compilando y ejecutando este programa se ve que **a** y **b** siguen teniendo los mismos valores antes y después de la llamada a **permutar()**, a pesar de que en el interior de la función los valores sí se han permutado (es fácil de comprobar introduciendo en el código de la función los **printf()** correspondientes). La razón está en que *se han permutado los valores de las copias* de **a** y **b**, pero no los valores de las propias variables. Las variables podrían ser permutadas si se recibieran sus direcciones (en realidad, *copias* de dichas direcciones). Las direcciones deben recibirse en *variables puntero*, por lo que los argumentos formales de la función deberán ser punteros. Una versión correcta de la función **permutar()** que pasa direcciones en vez de valores sería como sigue:

```
void permutar(double *x, double *y)
{
     double temp;
     temp = *x;
     *x = *y;
     *y = temp;
}
```

que puede ser llamada y comprobada de la siguiente forma:

```
#include <stdio.h>

void main(void)
{
     double a=1.0, b=2.0;
     void permutar(double *, double *);

     printf("a = %lf, b = %lf\n", a, b);
```

```
     permutar(&a, &b);
     printf("a = %lf, b = %lf\n", a, b);
}
```

Al mecanismo de paso de argumentos mediante direcciones en lugar de valores se le llama ***paso por referencia***, y deberá utilizarse siempre que la función deba devolver argumentos modificados. Un caso de particular interés es el paso de *arrays* (vectores, matrices y cadenas de caracteres). Este punto se tratará con más detalle un poco más adelante. Baste decir ahora que como ***los nombres de los arrays son punteros*** (es decir, direcciones), dichos datos *se pasan por referencia*, lo cual tiene la ventaja adicional de que no se gasta memoria y tiempo para pasar a las funciones copias de cantidades grandes de información.

Un caso distinto es el de las ***estructuras***, y conviene tener cuidado. Por defecto *las **estructuras se pasan por valor***, y pueden representar también grandes cantidades de datos (pueden contener *arrays* como miembros) de los que se realizan y transmiten copias, con la consiguiente pérdida de eficiencia. Por esta razón, *las estructuras se suelen pasar de modo explícito por referencia, por medio de punteros a las mismas*.

La función main() con argumentos

Cuando se ejecuta un programa desde ***MS-DOS*** tecleando su nombre, existe la posibilidad de pasarle algunos datos, tecleándolos a continuación en la misma línea. Por ejemplo, se le puede pasar algún valor numérico o los nombres de algunos ficheros en los que tiene que leer o escribir información. Esto se consigue por medio de argumentos que se pasan a la función **main()**, como se hace con otras funciones.

Así pues, a la función **main()** se le pueden pasar argumentos y también puede tener valor de retorno. El primero de los argumentos de **main()** se suele llamar **argc**, y es una variable *int* que contiene el número de palabras que se teclean a continuación del nombre del programa cuando éste se ejecuta. El segundo argumento se llama **argv**, y es un *vector de punteros a carácter* que contiene las direcciones de la primera letra o carácter de dichas palabras. A continuación se presenta un ejemplo:

```
int main(int argc, char *argv[])
{
     int cont;
     for (cont=0; cont<argc; cont++)
          printf("El argumento %d es: %s\n", cont, argv[cont]);
     printf("\n");
     return 0;
}
```

Funciones para cadenas de caracteres

En C, existen varias funciones útiles para el manejo de cadenas de caracteres. Las más utilizadas son: **strlen()**, **strcat()**, **strcmp()** y **strcpy()**. Sus prototipos o declaraciones están en el fichero **string.h**, y son los siguientes (se incluye a continuación una explicación de cómo se utiliza la función correspondiente).

FUNCIÓN STRLEN()

El prototipo de esta función es como sigue:

```
unsigned strlen(const char *s);
```

Explicación: Su nombre proviene de ***string length***, y su misión es contar el número de caracteres de una cadena, sin incluir el '\0' final. El paso del argumento se realiza *por referencia*, pues como argumento se emplea un puntero a la cadena (tal que el valor al que apunta es constante para la función; es decir, ésta no lo puede modificar), y devuelve un entero sin signo que es el número de caracteres de la cadena.

La palabra ***const*** impide que dentro de la función lac cadena de caracteres que se pasa como argumento sea modificada.

FUNCIÓN STRCAT()

El prototipo de esta función es como sigue:

```
char *strcat(char *s1, const char *s2);
```

Explicación: Su nombre proviene de ***string concatenation***, y se emplea para unir dos cadenas de caracteres poniendo **s2** a continuación de **s1**. El valor de retorno es un puntero a **s1**. Los argumentos son los punteros a las dos cadenas que se desea unir. La función almacena la cadena completa en la primera de las cadenas. ¡PRECAUCIÓN! Esta función no prevé si tiene sitio suficiente para almacenar las dos cadenas juntas en el espacio reservado para la primera. Esto es responsabilidad del programador.

FUNCIONES STRCMP() Y STRCOMP()

El prototipo de la función **strcmp()** es como sigue:

```
int strcmp(const char *s1, const char *s2)
```

Explicación: Su nombre proviene de ***string comparison***. Sirve para comparar dos cadenas de caracteres. Como argumentos utiliza punteros a las cadenas que se van a comparar. La función devuelve cero si las cadenas son iguales, un valor menor que cero si **s1** es menor –en orden alfabético– que **s2**, y un valor mayor que cero si **s1** es mayor que **s2**. La función **strcomp()** es completamente análoga, con la diferencia de que no hace distinción entre letras mayúsculas y minúsculas).

FUNCIÓN STRCPY()

El prototipo de la función **strcpy()** es como sigue:

```
char *strcpy(char *s1, const char *s2)
```

Explicación: Su nombre proviene de ***string copy*** y se utiliza para copiar cadenas. Utiliza como argumentos dos punteros a carácter: el primero es un puntero a la cadena copia, y el segundo es un puntero a la cadena original. El valor de retorno es un puntero a la cadena copia **s1**.

Es muy importante tener en cuenta que en C no se pueden copiar cadenas de caracteres directamente, por medio de una sentencia de asignación. Por ejemplo, sí se puede asignar un texto a una cadena en el momento de la declaración:

```
char s[] = "Esto es una cadena";          // correcto
```

Sin embargo, sería ilícito hacer lo siguiente:

```
char s1[20] = "Esto es una cadena";
char s2[20];
...
// Si se desea que s2 contenga una copia de s1
s2 = s1;           // incorrecto: se hace una copia de punteros
strcpy(s2, s1);    // correcto:   se copia toda la cadena
```

Punteros como valor de retorno

A modo de resumen, recuérdese que una función es un conjunto de instrucciones C que:

- Es llamado por el programa principal o por otra función.
- Recibe datos a través de una lista de argumentos, o a través de variables *extern*.
- Realiza una serie de tareas específicas, entre las que pueden estar cálculos y operaciones de lectura/escritura en el disco, en teclado y pantalla, etc.
- Devuelve resultados al programa o función que la ha llamado por medio del *valor de retorno* y de los *argumentos* que hayan sido *pasados por referencia* (punteros).

El utilizar *punteros como valor de retorno* permite superar la limitación de devolver un único valor de retorno. Puede devolverse un puntero al primer elemento de un vector o a la dirección base de una matriz, lo que equivale a devolver múltiple valores. El valor de retorno ***puntero a void*** (***void **) es un puntero de tipo indeterminado que puede asignarse sin *casting* a un puntero de cualquier tipo. Los *punteros a void* son utilizados por las funciones de reserva dinámica de memoria **calloc()** y **malloc()**, como se verá más adelante.

Paso de arrays como argumentos a una función

Para considerar el paso de ***arrays*** (vectores y matrices) como argumentos de una función, hay que recordar algunas de sus características, en particular su relación con los ***punteros*** y la forma en la que las matrices se almacenan en la memoria. Este tema se va a presentar por medio de un ejemplo: una función llamada **prod()** para realizar el producto de matriz cuadrada por vector ([a]{x}={y}).

Para que la definición de la función esté completa es necesario dar las dimensiones de la matriz que se le pasa como argumento (excepto la 1ª, es decir, excepto el nº de filas), con objeto de poder reconstruir la fórmula de direccionamiento, en la que interviene el número de columnas pero no el de filas. El encabezamiento de la definición sería como sigue:

```
void prod(int n, double a[][10], double x[], double y[])
{...}
```

Dicho encabezamiento se puede también establecer en la forma:

```
void prod(int n, double (*a)[10], double *x, double *y)
{...}
```

donde el paréntesis (*a) es necesario para que sea "puntero a vector de tamaño 10", es decir, puntero a puntero. Sin paréntesis sería "vector de tamaño 10, cuyos elementos son punteros", por la mayor prioridad del operador [ ] sobre el operador *.

La declaración de la función **prod()** se puede hacer en la forma:

```
void prod(int, double a[][10], double x[], double y[]);
```

o bien,

```
void prod(int n, double (*a)[10], double *x, double *y);
```

Para la llamada basta simplemente utilizar los nombres de los argumentos:

```
double a[10][10], x[10], y[10];
...
prod(nfilas, a, x, y);
...
```

En todos estos casos **a** es un *puntero a puntero*, mientras que **x** e **y** son *punteros*.

Punteros a funciones

De modo similar a como el nombre de un array en C es un puntero, también el nombre de una función es un puntero. Esto es interesante porque permite pasar como argumento a una función el nombre de otra función. Por ejemplo, si **pfunc** es un puntero a una función que devuelve un entero y tiene dos argumentos que son punteros, dicha función puede declararse del siguiente modo:

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

El primer paréntesis es necesario pues la declaración:

```
int *pfunc(void *, void *);      // incorrecto
```

corresponde a una función llamada ***pfunc*** que devuelve un puntero a entero. Considérese el siguiente ejemplo para llamar de un modo alternativo a las funciones ***sin()*** y ***cos(x)***:

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

void main(void){
     double (*pf)(double);

     *pf = sin;
     printf("%lf\n", (*pf)(3.141592654/2));
     *pf = cos;
     printf("%lf\n", (*pf)(3.141592654/2));
}
```

Obsérvese cómo la función definida por medio del puntero tiene la misma “signature” que las funciones seno y coseno. La ventaja está en que por medio del puntero ***pf*** las funciones seno y coseno podrían ser pasadas indistintamente como argumento a otra función.

FUNCIONES DE ENTRADA/SALIDA

A diferencia de otros lenguajes, *C no dispone de sentencias de entrada/salida*. En su lugar se utilizan funciones contenidas en la librería estándar y que forman parte integrante del lenguaje.

Las funciones de entrada/salida (Input/Output) son un conjunto de funciones, incluidas con el compilador, que permiten a un programa recibir y enviar datos al exterior. Para su utilización es necesario incluir, al comienzo del programa, el archivo **stdio.h** en el que están definidos sus prototipos:

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

donde ***stdio*** proviene de ***standard-input-output***.

Función printf()

La función **printf()** imprime en la unidad de salida (el monitor, por defecto), el texto, y las constantes y variables que se indiquen. La forma general de esta función se puede estudiar viendo su *prototipo*:

```
int printf("cadena_de_control",  tipo arg1, tipo arg2, ...)
```

Explicación: La función **printf()** imprime el texto contenido en **cadena_de_control** junto con el valor de los otros argumentos, de acuerdo con los *formatos* incluidos en **cadena_de_control**. Los puntos suspensivos (...) indican que puede haber un número variable de argumentos. Cada formato comienza con el carácter (%) y termina con un *carácter de conversión*.

Considérese el ejemplo siguiente,

```
int    i;
double tiempo;
float  masa;

printf("Resultado nº: %d. En el instante %lf la masa vale %f\n",
     i, tiempo, masa);
```

en el que se imprimen 3 variables (**i**, **tiempo** y **masa**) con los formatos (**%d**, **%lf** y **%f**), correspondientes a los tipos (***int***, ***double*** y ***float***), respectivamente. La cadena de control se imprime con el valor de cada variable intercalado en el lugar del formato correspondiente.

Tabla 8.1. Caracteres de conversión para la función **printf()**.

Carácter	Tipo de argumento	Carácter	Tipo de argumento
d, i	int decimal	o	octal unsigned
u	int unsigned	x, X	hex. unsigned
c	char	s	cadena de char
f	float notación decimal	e, g	float not. científ. o breve
p	puntero (void *)		

Lo importante es considerar que debe haber correspondencia uno a uno (el 1º con el 1º, el 2º con el 2º, etc.) entre los formatos que aparecen en la **cadena_de_control** y los otros argumentos (constantes, variables o expresiones). Entre el carácter % y el *carácter de conversión* puede haber, por el siguiente orden, uno o varios de los elementos que a continuación se indican:

- Un número entero positivo, que indica la *anchura* mínima del campo en caracteres.
- Un signo (-), que indica *alineamiento* por la izda (el defecto es por la dcha).
- Un punto (.), que separa la anchura de la *precisión*.
- Un número entero positivo, la *precisión*, que es el nº máximo de caracteres a imprimir en un *string*, el nº de decimales de un *float* o *double*, o las cifras mínimas de un *int* o *long*.
- Un *cualificador*: una (h) para *short* o una (l) para *long* y *double*

Los caracteres de conversión más usuales se muestran en la Tabla 8.1.

A continuación se incluyen algunos ejemplos de uso de la función **printf()**. El primer ejemplo contiene sólo texto, por lo que basta con considerar la **cadena_de_control**.

```
printf("Con cien cañones por banda,\nviento en popa a toda vela,\n");
```

El resultado serán dos líneas con las dos primeras estrofas de la famosa poesía. *No es posible partir **cadena_de_control** en varias líneas con caracteres **intro***, por lo que en este ejemplo podría haber problemas para añadir más estrofas. Una forma alternativa, muy sencilla, clara y ordenada, de escribir la poesía sería la siguiente:

```
printf("%s\n%s\n%s\n%s\n",
     "Con cien cañones por banda,",
     "viento en popa a toda vela,",
     "no cruza el mar sino vuela,",
     "un velero bergantín.");
```

En este caso se están escribiendo 4 cadenas constantes de caracteres que se introducen como argumentos, con formato %s y con los correspondientes saltos de línea. Un ejemplo que contiene una constante y una variable como argumentos es el siguiente:

```
printf("En el año %s ganó %ld ptas.\n", "1993", beneficios);
```

donde el texto **1993** se imprime como cadena de caracteres (%s), mientras que **beneficios** se imprime con formato de variable ***long*** (%ld). Es importante hacer corresponder bien los formatos con el tipo de los argumentos, pues si no los resultados pueden ser muy diferentes de lo esperado.

La función **printf()** *tiene un valor de retorno de tipo int*, que representa el número de caracteres escritos en esa llamada.

Función scanf()

La función **scanf()** es análoga en muchos aspectos a **printf()**, y se utiliza para leer datos de la entrada estándar (que por defecto es el teclado). La forma general de esta función es la siguiente:

```
int scanf("%x1%x2...", &arg1, &arg2, ...);
```

donde x1, x2, ... son los *caracteres de conversión*, mostrados en la Tabla 8.2, que representan los formatos con los que se espera encontrar los datos. La función **scanf()** devuelve como valor de retorno el número de conversiones de formato realizadas con éxito. La cadena de control de **scanf()** puede contener caracteres además de formatos. Dichos caracteres se utilizan para tratar de detectar la presencia de caracteres idénticos en la entrada por teclado. Si lo que se desea es leer variables numéricas, esta posibilidad tiene escaso interés. A veces hay que comenzar la cadena de control con un espacio en blanco para que la conversión de formatos se realice correctamente.

En la función ***scanf()*** *los argumentos que siguen a la* ***cadena_de_control*** *deben ser* ***pasados por referencia***, ya que la función los lee y tiene que trasmitirlos al programa que la ha llamado. Para ello, dichos argumentos deben estar constituidos por las *direcciones de las variables* en las que hay que depositar los datos, y no por las propias variables. Una excepción son las *cadenas de caracteres*, cuyo nombre es ya de por sí una dirección (un puntero), y por tanto no debe ir precedido por el *operador* (&) en la llamada.

Tabla 8.2. Caracteres de conversión para la función **scanf()**.

carácter	caracteres leídos	argumento
c	cualquier carácter	char *
d, i	entero decimal con signo	int *
u	entero decimal sin signo	unsigned int
o	entero octal	unsigned int
x, X	entero hexadecimal	unsigned int
e, E, f, g, G	número de punto flotante	float
s	cadena de caracteres sin ''	char
h, l	para short, long y double	
L	modificador para long double	

Por ejemplo, para leer los valores de dos variables *int* y *double* y de una cadena de caracteres, se utilizarían la sentencia:

```
int    n;
double distancia;
char   nombre[20];
scanf("%d%lf%s", &n, &distancia, nombre);
```

en la que se establece una correspondencia entre **n** y **%d**, entre **distancia** y **%lf**, y entre **nombre** y **%s**. Obsérvese que **nombre** no va precedido por el operador (&). La lectura de cadenas de caracteres se detiene en cuanto se encuentra un espacio en blanco, por lo que para leer una línea completa con varias palabras hay que utilizar otras técnicas diferentes.

En los formatos de la cadena de control de **scanf()** pueden introducirse *corchetes* [...], que se utilizan como sigue. La sentencia,

```
scanf("%[AB \n\t]", s);   // se leen solo los caracteres indicados
```

lee caracteres hasta que encuentra uno diferente de `('A','B',' ','\n','\t')`. En otras palabras, se leen sólo los caracteres que aparecen en el corchete. Cuando se encuentra un carácter distinto de éstos se detiene la lectura y se devuelve el control al programa que llamó a **scanf()**. Si los corchetes contienen un carácter (^), se leen todos los caracteres distintos de los caracteres que se encuentran dentro de los corchetes a continuación del (^). Por ejemplo, la sentencia,

```
scanf(" %[^\n]", s);
```

lee todos los caracteres que encuentra hasta que llega al carácter ***nueva línea*** '\n'. Esta sentencia puede utilizarse por tanto para leer líneas completas, con blancos incluidos. Recuérdese que con el formato ***%s*** la lectura se detiene al llegar al primer delimitador (carácter blanco, tabulador o nueva línea).

Macros getchar() y putchar()

Las macros[6] **getchar()** y **putchar()** permiten respectivamente leer e imprimir *un sólo carácter* cada vez, en la entrada o en la salida estándar. La macro **getchar()** recoge un carácter introducido por teclado y lo deja disponible como valor de retorno. La macro **putchar()** escribe en la pantalla el carácter que se le pasa como argumento. Por ejemplo:

```
putchar('a');
```

escribe el carácter **a**. Esta sentencia equivale a

```
printf("a");
```

mientras que

```
c = getchar();
```

equivale a

```
scanf("%c", &c);
```

Como se ha dicho anteriormente, **getchar()** y **putchar()** son *macros* y no *funciones*, aunque para casi todos los efectos se comportan como si fueran funciones. El concepto de ***macro*** se verá con más detalle en la siguiente sección. Estas macros están definidas en el fichero **stdio.h**, y su código es sustituido en el programa por el *preprocesador* antes de la compilación.

Por ejemplo, *se puede leer una línea de texto completa* utilizando **getchar()**:

```
int i=0, c;
char name[100];
while((c = getchar()) != '\n')  // se leen caracteres hasta el '\n'
    name[i++] = c;              // se almacena el carácter en Name[]
name[i]='\0';                   // se añade el carácter fin de cadena
```

Otras funciones de entrada/salida

Las funciones **fprintf()** y **fscanf()** se emplean de modo análogo a **printf()** y **scanf()**, pero en lugar de leer y escribir en la salida y en la entrada estándar, lo hacen en un *archivo de disco*. Dicho archivo deberá haber sido abierto previamente mediante la función **fopen()**, que funciona como se explica a continuación.

En primer lugar hay que declarar un *puntero a archivo* (se trata del *tipo* FILE, un tipo derivado que está predefinido en el fichero *stdio.h*), que servirá para identificar el archivo con el que se va a trabajar. Después se abre el archivo con la función **fopen()**, que devuelve como valor de retorno un puntero al archivo abierto. A **fopen()** hay que pasarle como argumentos el *nombre del archivo* que se quiere abrir y el *modo* en el que se va a utilizar dicho archivo. A continuación se presenta un ejemplo:

```
FILE *pf;          // declaración de un puntero a archivo

pf = fopen("nombre", "modo");
```

---

[6] Una ***macro*** representa una sustitución de texto que se realiza antes de la compilación por medio del preprocesador. Para casi todos los efectos, estas macros pueden ser consideradas como funciones. Más adelante se explicarán las macros con algunos ejemplos.

donde *nombre* es el nombre del archivo y *modo* es un carácter que indica el modo en el que el fichero se desea abrir:

```
r    sólo lectura (read)
w    escritura desde el comienzo del archivo (write)
a    escritura añadida al final del archivo (append)
r+   lectura y escritura
w+   escritura y lectura
rb   sólo lectura (archivo binario)
wb   escritura desde el comienzo del archivo (archivo binario)
ab   escritura añadida al final del archivo (archivo binario)
```

El puntero **pf** devuelto por **fopen()** será NULL (=0) si por alguna razón no se ha conseguido abrir el fichero en la forma deseada.

Los prototipos de las funciones **fscanf()** y **fprintf()** tienen la forma:

```
int fprintf(FILE *fp, "cadena de control", tipo arg1, tipo arg2, ...);
int fscanf(FILE *fp, "cadena de control", &arg1, &arg2, ...);
```

y la forma general de llamar a dichas funciones, que es completamente análoga a la forma de llamar a **printf()** y **scanf()**, es la siguiente:

```
fprintf(fp, "cadena_de_control", otros_argumentos);
fscanf(fp, "cadena_de_control", otros_argumentos);
```

Los argumentos de **fscanf()** se deben pasar *por referencia*. Una vez que se ha terminado de trabajar con un fichero, es conveniente cerrarlo mediante la función **fclose()**, en la forma:

```
fclose(fp);
```

Existen también unas funciones llamadas **sprintf()** y **sscanf()** que son análogas a **fprintf()** y **fscanf()**, sustituyendo el *puntero a fichero* por un *puntero a una cadena de caracteres* almacenada en la memoria del ordenador. Así pues, estas funciones leen y/o escriben en una cadena de caracteres almacenada en la memoria.

Hay otras funciones similares a **fprintf()** y **fscanf()**, las funciones **fread()** y **fwrite()**, que leen y escriben en disco *archivos binarios*, respectivamente. Como no se realizan conversiones de formato, estas funciones son mucho más eficientes que las anteriores en tiempo de CPU.

Las "funciones" **putc(c, fp)** y **getc(fp**) son asimismo ***macros*** definidas en **stdio.h** que escriben o leen un carácter de un fichero determinado por un puntero a fichero (FILE *fp).

Existen también funciones para acceder a un fichero de *modo directo*, es decir, *no secuencial*. El acceso secuencial lee y escribe las líneas una detrás de otra, en su orden natural. El acceso directo permite leer y escribir datos en cualquier orden.

Las funciones **remove()** y **rename()** se utilizan para borrar o cambiar el nombre a un fichero desde un programa.

Existen en C más funciones y macros análogas a las anteriores. Para más detalles sobre la utilización de todas estas funciones, consultar un manual de C.

EL PREPROCESADOR

El ***preprocesador*** del lenguaje C permite sustituir ***macros*** (sustitución en el programa de constantes simbólicas o texto, con o sin parámetros), realizar compilaciones condicionales e incluir archivos, todo ello antes de que empiece la compilación propiamente dicha. El preprocesador de C reconoce los siguientes comandos:

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

Los comandos más utilizados son: **#include**, **#define**.

Comando #include

Cuando en un archivo ***.c*** se encuentra una línea con un **#include** seguido de un nombre de archivo, el preprocesador la sustituye por el contenido de ese archivo.

La sintaxis de este comando es la siguiente:

```
#include "nombre_del_archivo"
#include <nombre_del_archivo>
```

La diferencia entre la primera forma (con comillas "...") y la segunda forma (con los símbolos <...>) estriba en el directorio de búsqueda de dichos archivos. En la forma con comillas se busca el archivo en el directorio actual y posteriormente en el directorio estándar de librerías (definido normalmente con una variable de entorno del MS-DOS llamada INCLUDE, en el caso de los compiladores de Microsoft). En la forma que utiliza los símbolos <...> se busca directamente en el directorio estándar de librerías. En la práctica, los archivos del sistema (***stdio.h***, ***math.h***, etc.) se incluyen con la segunda forma, mientras que los archivos hechos por el propio programador se incluyen con la primera.

Este comando del preprocesador se utiliza normalmente para incluir archivos con los ***prototipos*** (declaraciones) de las funciones de librería, o con módulos de programación y prototipos de las funciones del propio usuario. Estos archivos suelen tener la extensión ***.h,** aunque puede incluirse cualquier tipo de archivo de texto.

Comando #define

El comando **#define** establece una ***macro*** en el código fuente. Existen dos posibilidades de definición:

```
#define NOMBRE texto_a_introducir
#define NOMBRE(parámetros) texto_a_introducir_con_parámetros
```

Antes de comenzar la compilación, el preprocesador analiza el programa y cada vez que encuentra el identificador NOMBRE lo sustituye por el texto que se especifica a continuación en el comando **#define**. Por ejemplo, si se tienen las siguientes líneas en el código fuente:

```
#define E 2.718281828459
...
void main(void) {
     double a;
     a= (1.+1./E)*(1.-2./E);
     ...
}
```

al terminar de actuar el preprocesador, se habrá realizado la sustitución de **E** por el valor indicado y el código habrá quedado de la siguiente forma:

```
void main(void) {
     double a;
     a= (1.+1./2.718281828459)*(1.-2./2.718281828459);
     ...
}
```

Este mecanismo de sustitución permite definir constantes simbólicas o valores numéricos (tales como **E**, **PI**, **SIZE**, etc.) y poder cambiarlas fácilmente, a la vez que el programa se mantiene más legible.

De la forma análoga se pueden definir ***macros con parámetros***: Por ejemplo, la siguiente macro calcula el cuadrado de cualquier variable o expresión,

```
#define CUAD(x) ((x)*(x))

void main() {
     double a, b;
     ...
     a = 7./CUAD(b+1.);
     ...
}
```

Después de pasar el preprocesador la línea correspondiente habrá quedado en la forma:

```
a = 7./((b+1.)*(b+1.));
```

Obsérvese que los paréntesis son necesarios para que el resultado sea el deseado, y que en el comando **#define** no hay que poner el carácter (;). Otro ejemplo de ***macro*** con dos parámetros puede ser el siguiente:

```
#define C_MAS_D(c,d) (c + d)

void main() {
     double a, r, s;
     a = C_MAS_D(s,r*s);
     ...
}
```

con lo que el resultado será:

```
     a = (s + r*s);
```

El resultado es correcto por la mayor prioridad del operador (*) respecto al operador (+). Cuando se define una *macro con argumentos* conviene ser muy cuidadoso para prever todos los posibles resultados que se pueden alcanzar, y garantizar que todos son correctos. En la definición de una *macro* pueden utilizarse *macros* definidas anteriormente. En muchas ocasiones, *las macros son más eficientes que las funciones*, pues realizan una sustitución directa del código deseado, sin perder tiempo en copiar y pasar los valores de los argumentos.

Es recomendable tener presente que el comando **#define**:

– No define variables.

– Sus parámetros no son variables.

– En el preprocesamiento no se realiza una revisión de tipos, ni de sintaxis.

– Sólo se realizan sustituciones de código.

Es por estas razones que los posibles errores señalados por el compilador en una línea de código fuente con **#define** se deben analizar con las substituciones ya realizadas. Por convención entre los programadores, ***los nombres de las macros se escriben con mayúsculas***.

Existen también muchas *macros predefinidas a algunas variables internas del sistema*. Algunas se muestran en la Tabla 9.1.

Tabla 9.1. Algunas macros predefinidas.

`__DATE__`	Fecha de compilación
`__FILE__`	Nombre del archivo
`__LINE__`	Número de línea
`__TIME__`	Hora de compilación

Se puede definir una *macro* sin *texto_a_sustituir* para utilizarla como ***señal*** a lo largo del programa. Por ejemplo:

```
#define COMP_HOLA
```

La utilidad de esta línea se observará en el siguiente apartado.

Comandos #ifdef, #ifndef, #else, #endif, #undef

Uno de los usos más frecuentes de las *macros* es para establecer bloques de compilación opcionales. Por ejemplo:

```
#define COMP_HOLA

void main() {
     // si está definida la macro llamada COMP_HOLA
     #ifdef COMP_HOLA
          printf("hola");
     #else                         // si no es así
          printf("adios");
     #endif
}
```

El código que se compilará será **printf("hola")** en caso de estar definida la *macro* **COMP_HOLA**; en caso contrario, se compilará la línea **printf("adios")**. Esta compilación condicional se utiliza con frecuencia para desarrollar *código portable* a varios distintos tipos de computadores; según de qué computador se trate, se compilan unas líneas u otras. Para eliminar una *macro* definida previamente se utiliza el comando ***#undef***:

```
#undef COMP_HOLA
```

De forma similar, el comando **#ifndef** pregunta por la *no-definición* de la *macro* correspondiente. Un uso muy importante de los comandos **#ifdef** y **#ifndef** es para evitar comandos **#include** del mismo fichero repetidos varias veces en un mismo programa.

OTROS ASPECTOS DEL LENGUAJE C

Typedef

Esta palabra reservada del lenguaje C sirve para la creación de nuevos nombres de tipos de datos. Mediante esta declaración es posible que el usuario defina una serie de tipos de variables propios, no incorporados en el lenguaje y que se forman a partir de tipos de datos ya existentes. Por ejemplo, la declaración:

```
typedef int ENTERO;
```

define un tipo de variable llamado **ENTERO** que corresponde a *int*.

Como ejemplo más completo, se pueden declarar mediante **typedef** las siguientes estructuras:

```
#define MAX_NOM     30
#define
MAX_ALUMNOS 400

struct s_alumno {                // se define la estructura
         s_alumno char           nombre[MAX_NOM];
         short
         edad
           ;
};
typedef struct s_alumno ALUMNO;  // ALUMNO es un nuevo tipo de
variable typedef struct s_alumno *ALUMNOPTR;

struct clase {
         ALUMNO alumnos[MAX_ALUMNOS];
         char   nom_profesor[MAX_NOM];
};
typedef struct clase CLASE;
typedef struct clase *CLASEPTR;
```

Con esta definición se crean las cuatro palabras reservadas para tipos, denominadas **ALUMNO** (una estructura), **ALUMNOPTR** (un puntero a una estructura), **CLASE** y **CLASEPTR**. Ahora podría definirse una función del siguiente modo:

```
int anade_a_clase(ALUMNO un_alumno, CLASEPTR clase)
{
     ALUMNOPTR otro_alumno;
     otro_alumno = (ALUMNOPTR) malloc(sizeof(ALUMNO));
     otro_alumno->edad = 23;
     ...
     clase->alumnos[0]=alumno;
     ...
     return 0;
}
```

El comando **typedef** ayuda a parametrizar un programa contra problemas de portabilidad. Generalmente se utiliza **typedef** para los tipos de datos que pueden ser dependientes de la instalación. También puede ayudar a documentar el programa (es mucho más claro para el programador el tipo **ALUMNOPTR**, que un tipo declarado como un puntero a una estructura complicada), haciéndolo más legible.