CAPÍTULO 2
FUNCIONES Y PROCEDIMIENTOS

1. MODULO

En programación un módulo es una porción de un programa de computadora. De las varias tareas que debe realizar un programa para cumplir con su función u objetivos, un módulo realizará una de dichas tareas.

En general un módulo recibe como entrada la salida que haya proporcionado otro módulo o los datos de entrada al sistema (programa) si se trata del módulo principal de éste; y proporcionará una salida que, a su vez, podrá ser utilizada como entrada de otro módulo o bien contribuirá directamente a la salida final del sistema (programa), si se retorna al módulo principal.

En programación, los módulos suelen estar organizados jerárquicamente en niveles, de forma que hay un módulo principal que realiza las llamadas oportunas a los módulos de nivel inferior.

Cuando un módulo es convocado, recibe como entrada los datos proporcionados por otro del mismo nivel o de nivel superior, el que ha hecho la llamada; luego realiza su tarea. A su vez este módulo convocado puede llamar a otro u otros módulos de nivel inferior si fuera necesario; cuando ellos finalizan su tarea, devuelven la salida pertinente al módulo inmediato llamador, en secuencia reversa, finalmente se continúa con la ejecución del módulo principal.

En un lenguaje de programación un modulo se puede llamar procedimiento o función.

2. PROCEDIMIENTOS Y FUNCIONES

Las funciones y los procedimientos son conjuntos de instrucciones que realizar una tarea en particular y permiten crear programas complejos, mediante un reparto de tareas que permite construir el programa de forma estructurada y modular.

Desde un punto de vista académico, se entiende por **procedimiento** el conjunto de sentencias a las que se asocia un identificador (un nombre), y que realiza una tarea que se conoce por los cambios que ejerce sobre el conjunto de variables.

Y entendemos por **función** el conjunto de sentencias a las que se asocia un identificador (un nombre) y que genera un valor nuevo, calculado a partir de los argumentos que recibe.

Los elementos que componen un procedimiento o función son:

1. Un **identificador**, que es el nombre que sirve para invocar a esa función o a ese procedimiento.
2. Una **lista de parámetros**, que es el conjunto de variables que se facilitan al procedimiento o función para que realice su tarea modularizada. Al hacer la abstracción del sistema, y modularlo en partes más accesibles, hay que especificar los parámetros formales que permiten la comunicación y definen el dominio (tipo de dato) de los datos de entrada. Esa lista de parámetros define el modo en que podrán comunicarse el programa que utiliza la función y la función usada.
3. Un **cuerpo** o **conjunto de sentencias**. Las necesarias para poder realizar la tarea para la que ha sido definida la función o el procedimiento. Dentro de las sentencias también se encuentran las declaraciones de variables locales.
4. Un **entorno**. Entendemos por entorno el conjunto de variables globales al procedimiento o función, que pueden ser usadas y modificadas dentro del ámbito de la función. Esas variables, por ser globales y por tanto definidas en un ámbito más amplio al ámbito local de la función, no necesitan ser explicitadas en la lista de parámetros de la función.

Es una práctica desaconsejable trabajar con el entorno de la función desde el ámbito local de la función. Hacerlo lleva consigo que esa función deja de ser independiente de ese entorno y, por tanto, deja de ser exportable. Perderíamos entonces el valor de la independencia funcional, que es una de las propiedades de la programación por módulos.

Podemos pues concluir que el uso de variables globales dentro del cuerpo de un procedimiento o función es altamente desaconsejable.

En el lenguaje C no se habla habitualmente de procedimientos, sino sólo de funciones. Pero existen de las dos tipos de módulos. Procedimientos serían, por ejemplo, la función printf no se invoca para calcular valores nuevos, sino para realizar una tarea sobre las variables. Más claro se ve con la función scanf que, efectivamente, realiza una tarea que se conoce por los cambios que ejerce sobre una variable concreta. Y funciones serían, por ejemplo, la función strlen, que a partir de una cadena de caracteres que recibe como parámetro de entrada calcula un valor, que es la longitud de esa cadena; o la función sin, que a partir de un ángulo que recibe como valor de entrada, calcula el seno de ese ángulo como valor de salida.

En definitiva, una función o procedimiento es una porción de código, identificada con un nombre concreto (su identificador), que realiza una tarea concreta, que puede ser entendida de forma independiente al resto del programa, y que tiene muy bien determinado cómo se hace uso de ella, con qué parámetros se la invoca y bajo qué condiciones puede ser usada, cuál es la tarea que lleva a cabo, y cuál es el valor que calcula y devuelve.

Tanto los procedimientos como las funciones pueden ser vistos como cajas negras: un código del que desconocemos sus sentencias, al que se le puede suministrar unos datos de entrada y obtener modificaciones para esos valores de entrada y/o el cálculo de un nuevo valor, obtenido a partir de los valores que ha recibido como entrada.

Con eso se consigue programas más cortos; que el código pueda ser usado más de una vez; mayor facilidad para gestionar un correcto orden de ejecución de sentencias; que las variables tengan mayor carácter local, y no puedan ser manipuladas fuera del ámbito para el que han sido creadas.

3. FUNCIONES EN C

3.1 QUE ES UNA FUNCIÓN

Una función, en C, es un segmento independiente de código fuente, diseñado para realizar una tarea específica.

Las funciones son los elementos principales de un programa en C. Cada una de las funciones de un programa constituye una unidad, capaz de realizar una tarea determinada. Quizá se podría decir que un programa es simplemente un conjunto de definiciones de distintas funciones, empleadas luego de forma estructurada.

La primera función que aparece en todo programa C es la función principal, o función main. Todo programa ejecutable tiene una, y sólo una, función main. Un programa sin función principal no genera un ejecutable. Y si todas las funciones se crean para poder ser utilizadas, la función principal es la única que no puede ser usada por nadie: nadie puede invocar a la función principal de un programa. Tampoco puede llamarse a sí misma.

Además de la función principal, en un programa se pueden encontrar otras funciones: o funciones creadas y diseñadas por el programador para esa aplicación, o funciones ya creadas e implementadas y compiladas en librerías: de creación propia o adquirida.

También se pueden crear las propias funciones en C. Así, una vez creadas y definidas, ya pueden ser invocadas tantas veces como se quiera. Y así, podemos ir creando nuestras propias bibliotecas de funciones.

Siempre que hemos hablado de funciones hemos utilizado dos verbos, uno después del otro: creación y definición de la función. Y es que en una función hay que distinguir entre su declaración o prototipo (creación de la función), su definición (el cuerpo de código que recoge las sentencias que debe ejecutar la función para lograr llevar a cabo su tarea) y, finalmente, su invocación o llamada: una función creada y definida sólo se ejecuta si otra función la invoca o llama. Y en definitiva, como la única función que se ejecuta sin ser invocada (y también la única función que no permite ser invocada) es la función main, cualquier función será ejecutada únicamente si es invocada por la función main o por alguna función que ha sido invocada por la función main o tiene en su origen, en una cadena de invocación, una llamada desde la función main.

3.2 DECLARACIÓN DE UNA FUNCIÓN

La declaración de una función se realiza a través de su prototipo. Un prototipo tiene la forma:

tipo_funcion nombre_funcion([tipo1 [var1] [,… tipoN [varN]]);

Donde **tipo_funcion** declara de qué tipo es el valor que devolverá la función. Una función puede devolver valores de cualquier tipo de dato válido en C, tanto primitivo como diseñado por el programador. Si no devuelve ningún valor, entonces se indica que es de tipo void.

Donde **nombre_función** es el nombre (identificador) que le daremos a la función.

Donde tipo1,…, tipoN declara de qué tipo es cada uno de los valores que la función recibirá como parámetros al ser invocada. En la declaración del prototipo es opcional indicar el nombre que tomarán las variables que recibirán esos valores y que se comportarán como variables locales de la función.

Al final de la declaración viene el punto y coma. Y es que la declaración de una función es una sentencia en C. Una sentencia que se consigna fuera de cualquier función. La declaración de una función tiene carácter global dentro de programa donde se declara. No se puede declarar, ni definir, una función dentro de otra función: eso siempre dará error de compilación.

Toda función que quiera ser definida e invocada debe haber sido previamente declarada. El prototipo de la función presenta el modo en que esa función debe ser empleada. Es como la definición de su interface, de su forma de comunicación: qué valores, de qué tipo y en qué orden debe recibir la función como argumentos al ser invocada. El prototipo permite localizar cualquier conversión ilegal de tipos entre los argumentos utilizados en la llamada de la función y los tipos definidos en los parámetros, entre los paréntesis del prototipo. Además, controla que el número de argumentos usados en una llamada a una función coincida con el número de parámetros de la definición.

Existe una excepción a esa regla: cuando una función es de tipo int, puede omitirse su declaración. Pero es recomendable no hacer uso de esa excepción. Si en una expresión, en una sentencia dentro del cuerpo de una función, aparece un nombre o identificador que no ha sido declarado previamente, y ese nombre va seguido de un paréntesis de apertura, el compilador supone que ese identificador corresponde al nombre de una función de tipo int.

Todas las declaraciones de función deben preceder a la definición del cuerpo de la función main.

3.3 DEFINICIÓN DE LA FUNCIÓN

Ya tenemos la función declarada. Con el prototipo ha quedado definido el modo en que podemos utilizarla: cómo nos comunicamos nosotros con ella y qué resultado nos ofrece.

Ahora queda la tarea de definirla.

Hay que escribir el código, las sentencias, que van a realizar la tarea para la que ha sido creada la función.

La forma habitual que tendrá la definición de una función la conocemos ya, pues hemos visto ya muchas: cada vez que hacíamos un programa, y escribíamos la función principal, estábamos definiendo esa función main. Esa forma es:

```
tipo_funcion nombre_funcion([tipo1 var1][,… tipoN varN])
{
        [declaración de variables locales]
        [cuerpo de la función: grupo de sentencias]
        [return(parámetro);]
}
```

Donde el **tipo_función** debe coincidir con el de la declaración, lo mismo que nombre_función y lo mismo que la lista de parámetros. Ahora, en la definición, los parámetros de la función siguen recogiendo el tipo de dato y el nombre de la variable: pero ahora ese nombre NO es opcional. Debe ponerse, porque esos nombres serán los identificadores de las variables que recojan los valores que se le pasan a la función cuando se la llama o invoca. A esas variables se las llama parámetros formales: son variables locales a la función: se crean cuando la función es invocada y se destruyen cuando se termina la ejecución de la función.

La lista de parámetros puede ser una lista vacía porque no se le quiera pasar ningún valor a la función: eso es frecuente. En ese caso, tanto en el prototipo como en la definición, entre los paréntesis que siguen al nombre de la función se coloca la palabra clave void.

```
tipo_función nombre_función(void); // declaración del prototipo
```

Si la función no devuelve valor alguno, entonces se indica como de tipo void, al igual que ya se hizo en la definición del prototipo. Una función declarada como de tipo void no puede ser usada como operando en una expresión de C, porque esa función no tiene valor alguno. Una función de tipo void puede mostrar datos por pantalla, escribir o leer ficheros, etc.

El bloque de la función tiene tres partes: la declaración de las variables locales, el cuerpo de la función, donde están las sentencias que llevarán a cabo la tarea para la que ha sido creada y definida la función, y la sentencia return..

El bloque de la función viene recogido entre llaves. Aunque la función tenga una sola sentencia, es obligatorio recoger esa sentencia única entre las llaves de apertura y de cerrado.

Las variables creadas en el cuerpo de la función serán locales a ella. Se pueden usar identificadores idénticos para nombrar distintas variables de diferentes funciones, porque cada variable de cada función pertenece a un ámbito completamente disjunto al ámbito de otra función, y no hay posibilidad alguna de confusión. Cada variable tendrá su dirección y su ámbito distintos.

Aunque ya se ha dicho anteriormente, recordamos que todas las funciones en C, sin excepción alguna, están en el mismo nivel de ámbito, es decir, no se puede declarar ninguna función dentro de otra función, y no se puede definir una función como bloque interno en el cuerpo de otra función.

3.4 LLAMADA A LA FUNCIÓN

La llamada a una función es una sentencia habitual en C. Ya la hemos usado con frecuencia, invocando hasta el momento únicamente funciones de biblioteca. Pero la forma de invocar es la misma para cualquier función.

nombre_función([argumento1][, …, argumentoN]);

La sentencia de llamada está formada por el nombre de la función y sus argumentos (los valores que se le pasan) que deben ir recogidos en el mismo orden que la secuencia de parámetros del prototipo y entre paréntesis. Si la función no recibe parámetros (porque así esté definida), entonces se coloca después de su nombre los paréntesis de apertura y cerrado sin ninguna información entre ellos. Si no se colocan los paréntesis, se produce un error de compilación.

El paso de parámetros en la llamada exige una asignación para cada parámetro. El valor del primer argumento introducido en la llamada a la función queda asignado en la variable del primer parámetro formal de la función; el segundo valor de argumento queda asignado en el segundo parámetro formal de la función; y así sucesivamente. Hay que asegurar que el tipo de dato de los parámetros formales es compatible en cada caso con el tipo de dato usado en lista de argumentos en la llamada de la función. El compilador de C no dará error si se fuerzan cambios de tipo de dato incompatibles, pero el resultado será inesperado totalmente.

La lista de argumentos estará formada por nombres de variables que recogen los valores que se desean pasar, o por literales. No es necesario (ni es lo habitual) que los identificadores de los argumentos que se pasan a la función cuando es llamada coincidan con los identificadores de los parámetros formales.

Las llamadas a las funciones, dentro de cualquier función, pueden realizarse en el orden que sea necesario, y tantas veces como se quiera, independientemente del orden en que hayan sido declaradas o definidas. Incluso se da el caso, bastante frecuente como veremos más adelante, que una función pueda llamarse a sí misma.

Si la función debe devolver un valor, con cierta frecuencia interesará que la función que la invoca almacene ese valor en una variable local suya. En ese caso, la llamada a la función será de la forma:

variable = nombre_función([argumento1][, …, argumentoN]);

Aunque eso no siempre se hace necesario, y también con frecuencia encontraremos las llamadas a las funciones como partes de una expresión.

3.5 LA SENTENCIA RETURN

Hay dos formas ordinarias de terminar la ejecución de una función.

1. Llegar a la última sentencia del cuerpo, antes de la llave que cierra el bloque de esa función.

2. Llegar a una sentencia return. La sentencia return fuerza la salida de la función, y devuelve el control del programa a la función que la llamó, en la sentencia inmediatamente posterior a la de la llamada a la función.

Si la función es de un tipo de dato distinto de void, entonces en el bloque de la función debe escribirse, al menos, una sentencia return. En ese caso, además, en esa sentencia y a continuación de la palabra return, deberá ir el valor que devuelve la función: o el identificador de una variable o un literal, siempre del mismo tipo al tipo de la función o de otro tipo compatible.

Una función tipo void no necesariamente tendrá la sentencia return. En ese caso, la ejecución de la función terminará con la sentencia última del bloque. Si una función de tipo void hace uso de sentencias return, entonces en ningún caso debe seguir a esa palabra valor alguno: si así fuera, el compilador detectará un error y no compilará el programa.

La sentencia return puede encontrarse en cualquier momento del código de una función. De todas formas, no tendría sentido recoger ninguna sentencia más allá de una sentencia return que no estuviera condicionada, pues esa sentencia jamás llegaría a ejecutarse.

En resumen, la sentencia return realiza básicamente dos operaciones:

1. Fuerza la salida inmediata del cuerpo de la función y se vuelve a la siguiente sentencia después de la llamada.

2. Si la función no es tipo void, entonces además de terminar la ejecución de la función, devuelve un valor a la función que la llamó. Si esa función llamante no recoge ese valor en una variable, el valor se pierde, con todas las variables locales de la función abandonada.

La forma general de la sentencia return es:

return [expresión];

Si el tipo de dato de la expresión del return no coincide con el tipo de la función entonces, de forma automática, el tipo de dato de la expresión se convierte en el tipo de dato de la función.

Ha llegado el momento de ver algunos ejemplos.

Ejemplo 1:
Veamos primero una función de tipo void (Procedimiento): una que muestre un mensaje por pantalla.

Declaración: void mostrar(short);

Definición:

```
void mostrar(short x)
{
    printf("El valor recibido es %hd.", x);
}
```

Llamada:

```
mostrar(10);
```

que ofrece la siguiente salida por pantalla:

```
El valor recibido es 10.
```

Ejemplo 2:
Una función que reciba un entero y devuelva el valor de su cuadrado.

Declaración: unsigned long int cuadrado(short);

Definición:

```
unsigned long int cuadrado(short x)
{
    return x * x;
}
```

Una posible llamada:

```
printf("El cuadrado de %hd es %ld.\n", a, cuadrado(a));
```

Ejemplo 3:
Ahora con dos sentencias return: una función que reciba como parámetros formales dos valores enteros y devuelve el valor del mayor de los dos:

Declaración: short mayor(short, short);

Definición:

```
short mayor(short x, short y)
{
    if(x > y) return x;
    else return y;
}
```

Desde luego la palabra else podría omitirse, porque jamás se llegará a ella si se ejecuta el primer return, y si la condición del if es falsa, entonces se ejecuta el segundo return.

Otra posible definición:

```
short mayor(short x, short y)
{
    x > y ? return(x) : return(y);
}
```

Llamada:

```
A = mayor(a,b);
```

Donde la variable A guardará el mayor de los dos valores entre a y b.

3.6 ÁMBITO Y VIDA DE LAS VARIABLES

Ya conocemos el concepto de ámbito de la variable. Y ahora que ya sabemos algo de las funciones, es conveniente presentar cuándo se puede acceder a cada variable, cuándo diremos que está viva, etc.

Veamos un programa ya conocido, el del cálculo del factorial de un entero, resuelto ahora mediante funciones:

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

```
long Factorial(short);

int main(void)
{
  short n;
  printf("Introduzca el valor de n ... ");
  scanf("%hd",&n);
  printf("El factorial de %hd ",n);
  printf("es %ld",Factorial(n));
  getch();
  return 0;
}
long Factorial(short a)
{
  long F = 1;
  while(a)
  {
    F = F * a;
    a = a - 1;
  }
  return F;
}
```

En este programa, la función principal main tiene definida una variable de tipo short, a la que hemos llamado n. En esa función, esa variable es local, y podemos recoger sus características.

La variable, de tipo short, n se almacena en una dirección de memoria y guardará el valor que reciba de la función scanf.

La función main invoca a la función Factorial. En la llamada se pasa como parámetro el valor de la variable n. En esa llamada, el valor de la variable **n** se copia en la variable **a** de Factorial:

Desde el momento en que se produce la llamada a la función Factorial, abandonamos el ámbito de la función main. En este momento, la variable n está fuera de ámbito y no puede, por tanto hacerse uso de ella. No ha quedado eliminada: estamos en el ámbito de Factorial pero aún no han terminado todas las sentencias de main. En el cálculo dentro de la función Factorial se ve modificado el valor de la variable local a. Pero esa modificación para nada influye en la variable n, que está definida en otra posición de memoria distinta.

Cuando se termina la ejecución de la función Factorial, el control del programa vuelve a la función main. La variable a y la variable F mueren, pero el valor de la variable F ha sido recibido como parámetro en la función printf, y así podemos mostrarlo por pantalla. Ahora, de nuevo en la función principal, volvemos al ámbito de la variable n, de la que podríamos haber hecho uso si hubiera sido necesario.

Veamos ahora otro ejemplo, con un programa que calcule el máximo común divisor de dos enteros. De nuevo, resolvemos el problema mediante funciones:

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

long euclides(long, long);

int main(void)
```

```
{
    long n1, n2;
    do
    {
        printf("Introduzca el valor de n1 ... ");
        scanf("%ld", &n1);
        printf("Introduzca el valor de n2 ... ");
        scanf("%ld", &n2);
        if(n2 != 0)
            printf("\nEl mcd de %ld y %ld ",n1, n2);
            printf("es %ld\n", euclides(n1,n2));

    }while(n2 != 0);
    return 0;
}

long euclides(long a, long b)
{
    static short cont = 0;
    long mcd;
    while(b)
    {
        mcd = b;
        b = a % b;
        a = mcd;
    }
    printf("Invocaciones a la función ... %hd\n", ++cont);
    return mcd;
}
```

En esta ocasión, además, hemos incluido una variable static en la función euclides. Esta variable nos informará de cuántas veces se ha ejecutado la función.

Las variables n1 y n2, de main, dejan de estar accesibles cuando se invoca a la función euclides. En ese momento se copian sus valores en las variables a y b que comienzan a existir precisamente en el momento de la invocación de la función. Además de esas variables locales, y de la variable local mcd, se ha creado otra, llamada cont, que es también local a euclides pero que, a diferencia de las demás variables locales, no desaparecerá cuando se ejecute la sentencia return y se devuelva el control de la aplicación a la función main: es una variable declarada static. Cuando eso ocurra, perderá la variable cont su ámbito, y no podrá ser accedida, pero en cuanto se invoque de nuevo a la función euclides, allí estará la variable, ya creada, accesible para cuando la función la requiera.

3.7 LLAMADAS POR VALOR Y LLAMADAS POR REFERENCIA

Estos dos nuevos conceptos son tradicionales al hablar de funciones. Y muy importantes. Hacen referencia al modo en que la función recibe los parámetros.

Hasta ahora, en todos los ejemplos previos presentados, hemos trabajado haciendo llamadas “por valor”. Decimos que una función es llamada por valor cuando se copia el valor del argumento en el parámetro formal de la función. Una variable está en la función que llama; y otra variable, distinta, es la que recibe el valor en la función llamada. La función llamada no puede alterar el valor del argumento original de la función que llama. Únicamente puede cambiar el valor de su variable local que ha recibido por asignación el valor de esa variable en el momento en que se realizó la llamada a

la función. Así, en la función llamada, cada argumento es efectivamente una variable local inicializada con el valor con que se llamó a la función.

Pero supongamos que necesitamos en nuestro programa realizar con mucha frecuencia la tarea de intercambiar el valor de dos variables. Ya sabemos cómo se hace, y lo hemos visto resuelto tanto a través de una variable auxiliar como gracias al operador or exclusivo. Sería muy conveniente disponer de una función a la que se le pudieran pasar, una y otra vez, el par de variables de las que deseamos intercambiar sus valores. Pero ¿cómo lograr hacer ese intercambio a través de una función si todo lo que se realiza en la función llamada muere cuando termina su ejecución? ¿Cómo lograr que en la función que invoca ocurra realmente el intercambio de valores entre esas dos variables?

La respuesta no es trivial: cuando invocamos a la función (que llamaremos en nuestro ejemplo intercambio), las variables que deseamos intercambiar dejan de estar en su ámbito y no llegamos a ellas. Toda operación en memoria que realice la función intercambio morirá con su última sentencia: su único rastro será, si acaso, la obtención de un resultado, el que logra sobrevivir de la función gracias a la sentencia return.

Y aquí llegamos a la necesidad de establecer otro tipo de llamadas a funciones, las llamadas “por referencia”. En este tipo de llamada, lo que se transfiere a la función no es el valor del argumento, sino la dirección de memoria de la variable argumento. Se copia la dirección del argumento en el parámetro formal, y no su valor.

Evidentemente, en ese caso, el parámetro formal deberá ser de tipo puntero. En ese momento, la variable argumento quedará fuera de ámbito, pero a través del puntero correspondiente en los parámetros formales podrá llegar a ella, y modificar su valor.

La función intercambio podría tener el siguiente prototipo:

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

Y su definición podría ser la siguiente:

```
void intercambio(long*a,long*b)
{
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}
```

O también

```
void intercambio(long*a,long*b)
{
    short aux;
    aux = *b;
    *b = *a;
    *a = aux;
}
```

Supongamos que la función que llama a la función intercambio lo hace de la siguiente forma:

```
intercambio(&x,&y);
```

Donde lo que le pasa son las direcciones (no los valores) de las dos variables de las que se desea intercambiar sus valores.

En la función llamante tenemos:
y
En la función intercambio tenemos:
y
Es decir, dos variables puntero cuyos valores que se le van asignar serán las posiciones de memoria de cada una de las dos variables usadas como argumento, y que son con las que se desea realizar el intercambio de valores.

La función trabaja sobre los contenidos de las posiciones de memoria apuntadas por los dos punteros. Y cuando termina la ejecución de la función, efectivamente, mueren las dos variables puntero a y b creadas. Pero ya han dejado hecha la faena en las direcciones que recibieron al ser creadas: en ahora queda codificado el valor; y en queda codificado el valor . Y en cuanto termina la ejecución de intercambio regresamos al ámbito de esas dos variables x e y: y nos las encontramos con los valores intercambiados.

Muchos son los ejemplos de funciones que, al ser invocadas, reciben los parámetros por referencia. La función scanf recibe el parámetro de la variable sobre la que el usuario deberá indicar su valor con una llamada por referencia. También lo hemos visto en la función gets, que recibe como parámetro la dirección de la cadena de caracteres donde se almacenará la cadena que introduzca el usuario.