Tema 5.- Funciones

Introducción

Una de las formas más adecuadas de resolver un problema de programación consiste en descomponerlo en subproblemas. A cada uno de ellos se le asocia una función que lo resuelve, de tal modo que la solución del problema se obtiene por medio de llamadas a funciones. A su vez, cada función puede descomponerse en subfunciones que realicen tareas más elementales, intentando conseguir que cada función realice una y sólo una tarea.

En Lenguaje C una función se define de la siguiente forma:

```
tipo NombreFunción (parámetros formales)
{
    ...
    cuerpo de la función
    ...
}
```

El **tipo** es el tipo de dato que devuelve la función por medio de la sentencia **return** cuyo estudio se adelantó en la página 8. Cuando no se especifica un tipo, se asume que el tipo devuelto es **int**. El **NombreFunción** es un identificador válido en C. Es el nombre mediante el cual se invocará a la función desde **main()** o desde otras funciones. Los **parámetros formales** son las variables locales mediante las cuales la función recibe datos cuando se le invoca. Deben ir encerrados entre paréntesis. Incluso si no hay parámetros formales los paréntesis deben aparecer. La siguiente función devuelve el mayor de dos números enteros:

```
int mayor (int x, int y)
{
    int max;

    if (x > y) max = x;
    else max = y;

    return max;
}
```

Al manejar funciones en C debemos tener en cuenta que a una función sólo se puede acceder por medio de una llamada. Nunca se puede saltar de una función a otra mediante una sentencia **goto**. Tampoco está permitido declarar una función dentro de otra.

En C, a diferencia de otros lenguajes, como Pascal, no existe distinción entre funciones y procedimientos. Por ello, una función puede estar o no dentro de una expresión. Si la función devuelve un valor, son válidas sentencias del tipo:

```
a = b * funcion (c, d);
```

Pero también podemos encontrar funciones solas en una línea de programa,

```
funcion (a, b, c);
```

incluso aunque la función devuelva algún valor. En ese caso la función realizaría la tarea encomendada y el valor devuelto se perdería. Esto no provoca ningún error.

Por último, hemos de tener en cuenta que, aunque una función puede formar parte de una expresión compleja, nunca se le puede asignar un valor. Por tanto, es incorrecto escribir sentencias como

funcion () = variable;

que sería tan improcedente como escribir

5 = variable;

El único método para que una función reciba valores es por medio de los parámetros formales.

Argumentos de funciones

Los parámetros formales son variables locales que se crean al comenzar la función y se destruyen al salir de ella. Al hacer una llamada a la función los parámetros formales deben coincidir en tipo y en número con las variables utilizadas en la llamada, a las que denominaremos **argumentos de la función**. Si no coinciden, puede no detectarse el error. Esto se evita usando los llamados *prototipos de funciones* que estudiaremos más adelante, en este capítulo.

Los argumentos de una función pueden ser:

- valores (llamadas por valor)
- direcciones (llamadas por referencia)

Llamadas por valor

En las llamadas por valor se hace una copia del valor del argumento en el parámetro formal. La función opera internamente con estos últimos. Como las variables locales a una función (y los parámetros formales lo son) se crean al entrar a la función y se destruyen al salir de ella, cualquier cambio realizado por la función en los parámetros formales no tiene ningún efecto sobre los argumentos. Aclaremos esto con un ejemplo.

```
#include <stdio.h>

void main ()
{
    int a = 3, b = 4;

    intercambia (a, b);
    printf ("\nValor de a: %d - Valor de b: %d", a, b);
}

void intercambia (int x, int y)
{
    int aux;

    aux = x;
    x = y;
    y = aux;
}
```

La salida de este programa es:

Valor de a: 3 - Valor de b: 4

es decir, NO intercambia los valores de las variables **a** y **b**. Cuando se hace la llamada

intercambia (a, b);

se copia el valor de **a** en **x**, y el de **b** en **y**. La función trabaja con **x** e **y**, que se destruyen al finalizar, sin que se produzca ningún proceso de copia a la inversa, es decir, de **x** e **y** hacia **a** y **b**.

Llamadas por referencia

En este tipo de llamadas los argumentos contienen direcciones de variables. Dentro de la función la dirección se utiliza para acceder al argumento real. En las llamadas por referencia cualquier cambio en la función tiene efecto sobre la variable cuya dirección se pasó en el argumento. Esto es así porque se trabaja con la propia dirección de memoria, que es única, y no hay un proceso de creación/destrucción de esa dirección.

Aunque en C todas las llamadas a funciones se hacen por valor, pueden simularse llamadas por referencia utilizando los operadores **&** (dirección) y ***** (en la dirección). Mediante **&** podemos pasar direcciones de variables en lugar de valores, y trabajar internamente en la función con los contenidos, mediante el operador *****.

El siguiente programa muestra cómo se puede conseguir el intercambio de contenido en dos variables, mediante una función.

```
#include <stdio.h>

void main ()
{
    int a = 3, b = 4;

    intercambio (&a, &b);
    printf ("\nValor de a: %d - Valor de b: %d", a, b);
}

void intercambio (int *x, int *y)
{
    int aux;

    aux = *x;
    *x = *y;
    *y = aux;
}
```

Este programa SÍ intercambia los valores de **a** y **b**, y muestra el mensaje

Valor de a: 4 - Valor de b: 3

Además, a diferencia del de la página 83, trabaja con direcciones. Como veremos en el siguiente capítulo, en la declaración

void intercambio (int *x, int *y);

los parámetros **x** e **y** son unas variables especiales, denominadas **punteros**, que almacenan direcciones de memoria. En este caso, almacenan direcciones de enteros.

Valores de retorno de una función

Como ya vimos, de una función se puede salir de dos formas distintas: bien porque se "encuentra" la llave **}** que cierra la función, bien porque se ejecuta una sentencia **return**. En este último caso la forma de la sentencia puede ser cualquiera de las siguientes:

return *constante***;**
return *variable***;**
return *expresión***;**

donde *expresión* puede, por claridad, ir entre paréntesis. El valor devuelto por **return** debe ser compatible con el tipo declarado para la función.

Como vimos al principio del capítulo, si a una función no se le asigna ningún tipo devuelve un valor de tipo **int**, por lo que debe tener una sentencia **return** de salida que devuelva un entero. Sin embargo, la ausencia de esta sentencia sólo provoca un **Warning** en la compilación. De cualquier manera, si una función no devuelve ningún valor (no hay sentencia **return**) lo correcto es declararla del tipo **void**.

Tampoco es necesario declarar funciones de tipo **char**, pues C hace una conversión limpia entre los tipos **char** e **int**.

Prototipos de funciones

Cuando una función devuelve un tipo no entero, antes de utilizarla, hay que "hacérselo saber" al resto del programa usando los **prototipos de funciones**. Un prototipo de función es algo que le indica al compilador que existe una función y cuál es la forma correcta de llamarla. Es una especie de "plantilla" de la función, con dos cometidos:

- identificar el tipo devuelto por la función.
- identificar el tipo y número de argumentos que utiliza la función.

La forma de definir un prototipo de función es:

tipo NombreFunción (lista de tipos);

Entre paréntesis se ponen los tipos de los argumentos separados por comas (no es necesario poner el nombre) en el mismo orden en el que deben estar en la llamada. Así, un prototipo como el siguiente

```
float Ejemplo (int, char);
```

indica que el programa va a utilizar una función llamada **Ejemplo** que devuelve, mediante una sentencia **return**, un valor de tipo **float**, y recibe dos valores en sus argumentos: el primero es un valor entero y el segundo un carácter. También es un prototipo válido

```
float Ejemplo (int x, char y);
```

En un programa, los prototipos de funciones se sitúan antes de la función **main()**. A continuación se muestra un programa que define un prototipo de función.

```
#include <stdio.h>

float multiplica (float, float);          /* Prototipo de la función multiplica() */

main ()
{
    float a, b, total;

    printf ("\nTeclee dos números: ");
    scanf ("%f %f", &a, &b);
    total = multiplica (a, b);
    printf ("\nEl producto de ambos es %f", total);
}

float multiplica (float m, float n)
{
    return m * n;
}
```

En este programa si en lugar de la línea

total = multiplica (a, b);

escribiésemos

total = multiplica (a);

el compilador, alertado por el prototipo, informaría del error.

Tanto en los prototipos como en las declaraciones de funciones, las referencias a cadenas de caracteres se realizan mediante la expresión **char ***. El significado de esta expresión se entenderá más claramente en el próximo capítulo. Así, el prototipo de una función que devuelve una cadena de caracteres y tiene como único argumento una cadena de caracteres es

char *Funcion (char *);

Cuando una función tiene un número variable de argumentos, se especifica por medio de 3 puntos suspensivos. Este es el caso de la función **printf()** cuyo prototipo es:

int printf (const char *, ...);

La palabra reservada **const** aquí significa que la cadena referida con **char *** puede ser, bien una variable, como en

printf (cadena);

o bien una constante, como en

printf ("Cadena constante");

Para indicar wque una función no tiene argumentos se pone la palabra reservada **void** entre los paréntesis. Del mismo modo, se indica con **void** que una función no devuelve ningún valor (no hay sentencia return). Por ejemplo, la función de prototipo

void funcion (void);

no devuelve nada ni recibe valores.

Cuando deseamos que una función devuelva una cadena de caracteres, el prototipo se escribe

char *funcion (lista de parámetros);

En el próximo capítulo estudiaremos más profundamente el significado de la expresión **char ***.

Todos los programas que utilicen funciones de la biblioteca estándar deben incluir sus prototipos. Estos prototipos y otras definiciones usadas por las funciones están en los archivos de cabecera como **stdio.h**, por lo que el programa deberá tener las sentencias **#include** necesarias.

Recursividad

La recursividad es un concepto de lógica matemática que consiste en definir una función en términos de sí misma. Por ejemplo, la definición del factorial de un número es una definición recursiva:

$$n! = n \cdot (n - 1)!$$
$$0! = 1$$

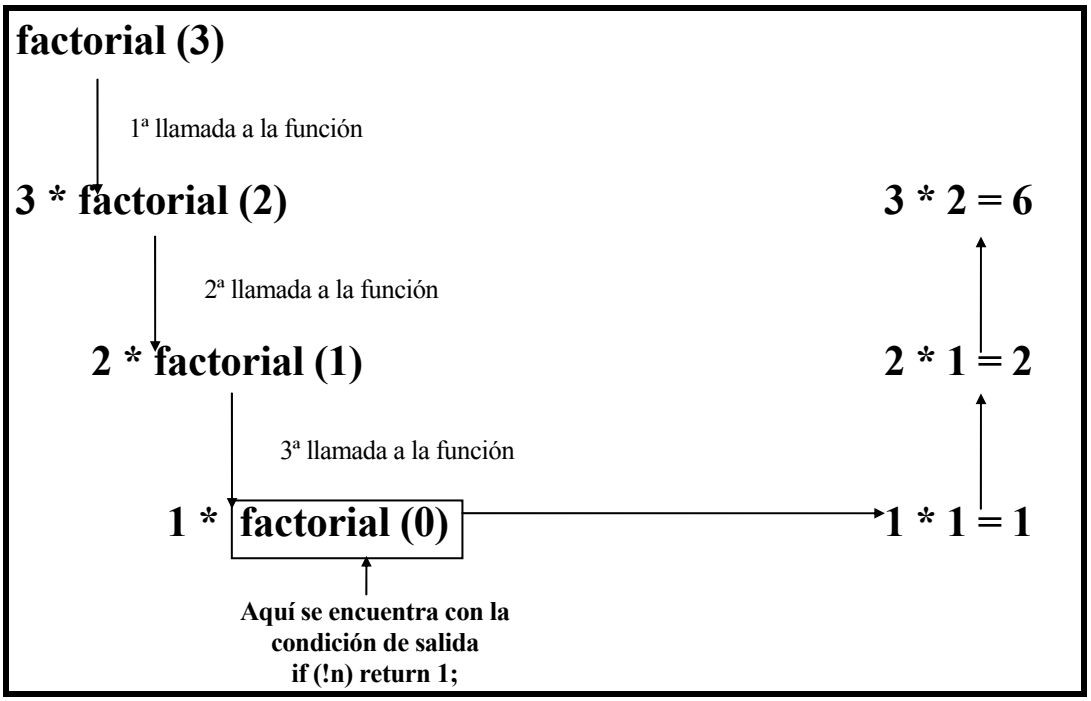
En programación una función es recursiva si puede llamarse a sí misma. No todos los lenguajes permiten la recursividad. Una definición recursiva de una función requiere dos partes:

- Una **condición de salida** o **cláusula de escape**.
- Un paso en el que los valores restantes de la función se definen en base a valores definidos anteriormente.

La recursividad es una alternativa a la iteración. Sin embargo, en general, una solución recursiva es menos eficiente, en términos de tiempo, que una solución iterativa. Además, las soluciones recursivas utilizan mucha memoria de pila, pues cada llamada a la función hace que se copie en la pila (cada vez) todo el juego de argumentos. Esto puede crear problemas cuando se llega a niveles profundos de recursión. Cómo decidir cuándo un problema se resuelve recursivamente o mediante iteraciones depende del problema en sí. La solución recursiva suele ser la más sencilla, y debe elegirse si no hay grandes requerimientos de velocidad del proceso ni problemas con el tamaño de la memoria. En otro caso debemos elegir la solución iterativa, mucho más rápida, aunque a menudo más compleja. La siguiente función resuelve recursivamente el cálculo del factorial

```
long factorial (int n)
    {
      if (!n) return 1;
      return n * factorial (n - 1);
    }
```

Representaremos ahora en un esquema el proceso de recursión al llamar, por ejemplo, a **factorial(3)**.



Se puede comparar este proceso con el que genera la solución iterativa:

```
long factorial (int n)
{
    register int i;
    long total = 1;

    for (i = 1; i <= n; i++) total *= i;
    return total;
}
```

En el primer caso se hacen varias llamadas a la función, situando en la pila los valores correspondientes y "dejando pendientes" los cálculos intermedios, hasta que se llega a la condición de salida. En ese momento comienza el proceso inverso, sacando de la pila los datos previamente almacenados y solucionando los cálculos que "quedaron sin resolver". Obviamente esta solución es más lenta que la iterativa en la que el resultado se obtiene con una sola llamada a la función y con menos requerimientos de memoria.

Sin embargo, mientras que en el caso del cálculo del factorial de un número parece más aconsejable la solución iterativa, hay otros ejemplos en lo que ocurre lo contrario. Por ejemplo, la función de Ackerman se define del siguiente modo:

$$Ack(s, t) = \begin{cases} t + 1 \\ Ack(s - 1, 1) \\ Ack(s - 1, Ack(s, t - 1)) \end{cases} \text{ si } \begin{cases} s = 0 \\ s \neq 0 \text{ y } t = 0 \\ s \neq 0 \text{ y } t \neq 0 \end{cases}$$

El diseño recursivo de esta función consiste, simplemente, en aplicar la definición

```
int Ack (int s, int t)
{
```

```
    if (!s) return t + 1;
    else if (!t) return Ack (s - 1, 1);
    else return Ack (s - 1, Ack (s, t - 1));
}
```

La solución no recursiva de la función de Ackerman es mucho más complicada.

La biblioteca de funciones

Hay una serie de funciones y macros definidas por el estándar ANSI que realizan tareas que facilitan enormemente la labor del programador. Además, cada fabricante de software incluye sus propias funciones y macros, generalmente relacionadas con el mejor aprovechamiento de los ordenadores personales y del MS-DOS. Respecto a esto, debe tenerse en cuenta que la compatibilidad del código sólo está asegurado si se utilizan exclusivamente funciones pertenecientes al estándar ANSI. Como ya quedó dicho, los prototipos de estas funciones, así como declaraciones de variables, macros y tipos de datos utilizados por ellas, están definidos en los archivos de cabecera ***.h**. Por ello, es necesario incluirlos mediante sentencias **#include** cuando el programa hace uso de ellas.

La tabla de la página siguiente muestra los archivos de cabecera estándar usados por Turbo C++. En el Capítulo 13 se muestran algunas de las funciones de la biblioteca estándar.

LENGUAJE	***ARCHIVO DE CABECERA***	***DESCRIPCIÓN***
	ALLOC.H	*Define funciones de asignación dinámica de memoria*
ANSI C	*ASSERT.H*	*Declara la macro de depuración **assert***
C++	*BCD.H*	*Define la clase **bcd***
	BIOS.H	*Define funciones utilizadas en rutinas de ROM-BIOS*
C++	*COMPLEX.H*	*Define las funciones matemáticas complejas*
C++	*CONIO.H*	*Define varias funciones utilizadas en las llamadas a rutinas de E/S por consola en DOS*
ANSI C	*CTYPE.H*	*Contiene información utilizada por las macros de conversión y clasificación de caracteres*
	DIR.H	*Contiene definiciones para trabajar con directorios.*
	DOS.H	*Declara constantes y da las declaraciones necesarias para llamadas específicas del 8086 y del DOS*
ANSI C	*ERRNO.H*	*Declara mnemónicos constantes para códigos de error*
	FCNTL.H	*Declara constantes simbólicas utilizadas en conexiones con la biblioteca de rutinas **open()***
ANSI.C	*FLOAT.H*	*Contiene parámetros para rutinas de coma flotante*
C++	*FSTREAM.H*	*Define los flujos de C++ que soportan E/S de archivos*
C++	*GENERIC.H*	*Contiene macros para declaraciones de clase genéricas*
C++	*GRAPHICS.H*	*Define prototipos para las funciones gráficas*
	IO.H	*Declaraciones de rutinas de E/S tipo UNIX*
C++	*IOMANIP.H*	*Define los gestores de flujos de E/S de C++ y contiene macros para creación de gestores de parámetros*
C++	*IOSTREAM.H*	*Define rutinas básicas de flujo de E/S de C++ (v2.0)*
ANSI C	*LIMITS.H*	*Parámetros y constantes sobre la capacidad del sistema*
ANSI C	*LOCALE.H*	*Define funciones sobre el país e idioma*
ANSI C	*MATH.H*	*Define prototipos para las funciones matemáticas*

	MEM.H	*Define las funciones de gestión de memoria*
	PROCESS.H	*Contiene estructuras y declaraciones para las funciones* ***spawn()***, ***exec()***
ANSI C	*SETJMP.H*	*Declaraciones para dar soporte a saltos no locales*
	SHARE.H	*Parámetros utilizados en funciones que utilizan arhivos-compartidos*
ANSI C	*SIGNAL.H*	*Declara constantes y declaraciones para utilizarlos en funciones* ***signal()*** *y* ***raise()***
ANSI C	*STDARG.H*	*Soporte para aceptar un número variable de argumentos*
ANSI C	*STDDEF.H*	*Declara varios tipos de datos y macros de uso común*
ANSI C	*STDIO.H*	*Declara tipos y macros para E/S estándar*
C++	*STDIOSTR.H*	*Declara las clases de flujo para utilizar con estructuras del archivo* ***stdio.h***
ANSI C	*STDLIB.H*	*Define algunas de las rutinas comúnmente utilizadas*
C++	*STREAM.H*	*Define las clases de flujo de C++ para utilizarlas con arrays de bytes en memoria*
ANSI C	*STRING.H*	*Define varias rutinas de manipulación de cadenas y de memoria*
	SYS\STAT.H	*Declara constantes simbólicas utilizadas para abrir y crear archivos*
	SYS\TIMEB.H	*Define la función* ***ftime()*** *y la estructura* ***timeb***
C++	*SYS\TYPES.H*	*Define el tipo* ***time_t***
ANSI C	*TIME.H*	*Estructuras y prototipos para funciones de tiempo*
	VALUES.H	*Declara constantes dependientes de la máquina*

Ejercicios

1. El seno de un ángulo puede calcularse mediante la serie

$$\text{sen}\,(x) = x - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \cdots$$

donde **x** se expresa en radianes (π radianes = 180 grados). Escribe un programa que calcule y muestre en pantalla el seno de un ángulo mediante la serie anterior. Finaliza el cálculo en el término de la serie cuyo valor sea menor o igual a10^{-3}. Para el cálculo de $\mathbf{x^n}$ crea una función no recursiva de prototipo

double potencia (duble x, int n);

De igual modo, para el cálculo de **n!** crea una función no recursiva de prototipo

double factorial (int n);

El ángulo debe ser un valor comprendido entre 0 y 360. En caso se enviará un mensaje a pantalla.

2. Escribe un programa que calcule $\mathbf{x^n}$, siendo **x** y **n** dos enteros positivos que se introducen por teclado. Para el cálculo crea una función recursiva de prototipo

long potencia (int x, int n);

que solucione el cálculo.

3. Escribe un programa que muestre los **N** primeros términos de la sucesión de Fibonacci, utilizando una función recursiva

int Fibonacci (int N);

que devuelva el elemento **N**. El valor **N** se leerá del teclado.

4. Escribe un programa que muestre en pantalla la información del equipo proporcionada por las funciones de biblioteca **biosequip()** y **biosmemory()**.

5. Por medio de la función **bioskey()** construye un programa que muestre en pantalla el estado de pulsación del teclado con el siguiente formato:

Mayúsculas derecha:	SI/NO	
Mayúsculas izquierda:	SI/NO	
Tecla Control:	SI/NO	
Tecla Alt:		SI/NO
Tecla Bloq Despl:		SI/NO
Tecla Bloq Num:		SI/NO
Tecla Bloq Mayús:		SI/NO
Tecla Ins:		SI/NO

NOTA: Para evitar el efecto que produce el cursor en la presentación de la pantalla (hay que estar continuamente imprimiendo la información en la pantalla) elimínalo usando la función **_setcursortype()** de la biblioteca estándar.

6. Escribe una función de prototipo

char *Intro (int f, int c, int tam, char *cad);

que utilice la función **cgets()** para capturar en la fila **f**, columna **c**, una cadena de caracteres de longitud máxima **tam**, y la almacene en **cad**. Devuelve **cad**.

7. Escribe una función de prototipo

int strdigit (char *cad);

que inspeccione la cadena **cad** y devuelva **1** si **cad** está compuesta por dígitos numéricos, y **0** si algún carácter de **cad** no es numérico. Utiliza la función **isdigit()**.

8. Escribe una función de prototipo

char *Format_fecha (char *fecha, int tipo, char *format);

que recibe en la cadena **fecha** una fecha con formato DDMMAAAA y pone en la cadena **format** esta fecha en un formato indicado por **tipo**. Los valores permitidos por **tipo** y sus formatos correspondientes se muestran en la siguiente tabla:

tipo	***format***
0	*DD/MM/AA*
1	*DD/MM/AAAA*
2	*DD de MMMMMMMMMM de AAAA*
3	*diasemana, DD de MMMMMMMMMM de AAAA*

Si **fecha** no es una fecha válida o **tipo** está fuera de rango, **format** será la cadena nula. Devuelve **format**. Utiliza la función **sprintf()**.

9. Escribe una función de prototipo

int Valida_fecha (char *fecha);

que recibe en la cadena **fecha** una fecha en formato DDMMAAAA y devuelve un valor de **0** a **6** si la fecha es correcta, indicando con los valores 0 a 6 el día de la semana a que corresponde **fecha** (0=domingo, 1=lunes, ...). Si la fecha no es correcta, devolverá **-1**. Utiliza la función **intdos** para hacer las llamadas a las funciones 2Ah y 2Bh de la INT 21h.

10. Escribe una función de prototipo

int Tecla (int *scan);

que captura una pulsación de tecla y devuelve en **scan** el código de exploración, y mediante **return**, su código ASCII. Utiliza la función **bioskey()**.

11. Escribe una función de prototipo

int Mensaje (int st, int fil, int col, char *cad);

que presenta en pantalla la cadena **cad** en la fila **fil**, columna **col**. Si **st** vale **1**, espera a que se pulse una tecla, devolviendo mediante **return** su código ASCII si se pulsó una tecla normal, y 1000+código de exploración si se pulsó una tecla especial. Si **st** vale **0**, no espera a que se pulse ninguna tecla.