Funciones en C 03

En esta unidad aprenderás a:

- Comprender el desarrollo de un programa utilizando funciones
- Establecer la diferencia entre la definición y declaración de una función
- Conocer y utilizar el paso de valores entre funciones y la devolución de valores de una función
- Establecer la diferencia entre el paso de argumentos por valor y por referencia
- Conocer la clasificación de las variables de acuerdo a su ámbito y clase de almacenamiento

3.1 Introducción

En la unidad 2 hemos visto que una función es un fragmento de código que realiza una tarea bien definida. Por ejemplo, la función *printf* imprime por la salida estándar los argumentos que le pasamos. Al igual que esta función, existen otras funciones que realizan diversas tareas ya definidas en el estándar ANSI C y que pueden ser utilizadas por el programador. Este tipo de funciones predefinidas son denominadas **funciones de biblioteca.** Sin embargo, cada programador puede definir sus propias funciones de acuerdo a sus necesidades. Las funciones que define el programador son conocidas como **funciones de usuario.**

La utilización de funciones nos permite dividir un programa extenso en pequeños segmentos que realizan tareas concretas. Probablemente, dentro de un mismo programa se realicen las mismas tareas varias veces, lo que se facilita mediante la utilización de funciones. Sin embargo, es probable que ciertas funciones no sean reutilizables, pero al usarlas se mejora la legibilidad del programa.

La filosofía en la que se base el diseño de C es el empleo de funciones. Por esta razón, un programa en C contiene al menos una función, la función *main*. Esta función es particular dado que la ejecución del programa se inicia con las instrucciones contenidas en su interior. Una vez iniciada la ejecución del programa, desde la función *main* se puede llamar a otras funciones y, posiblemente, desde estas funciones a otras. Otra particularidad de la función *main* es que se llama directamente desde el sistema operativo y no desde ninguna otra función. De esta manera, un programa en C sólo puede contener una función *main*.

Con el propósito de permitir un manejo eficiente de los datos, las funciones en C no se pueden anidar. En otras palabras, una función no se puede declarar dentro de otra función, por lo que todas las funciones son globales o externas, lo que hace que puedan llamarse desde cualquier parte de un programa.

Se puede acceder (llamar) a una determinada función desde cualquier parte de un programa. Cuando se llama a una función, se ejecutan las instrucciones que constituyen dicha función. Una vez que se ejecutan las instrucciones de la función, se devuelve el control del programa a la siguiente instrucción (si existe) inmediatamente después de la que provocó la llamada a la función.

Cuando se accede a una función desde un determinado punto del programa, se le puede pasar información mediante unos identificadores especiales conocidos como **argumentos** (también denominados parámetros). Una vez que la función procesa esta información, devuelve un valor mediante la instrucción *return*.

La estructura general de una función en C es la siguiente:

```
tipo_de_retorno nombre_de_la_función (lista_de_parámetros)
{
      cuerpo_de_la_función
      return expresión
}
```

Donde:

- **tipo_de_retorno:** es el tipo del valor devuelto por la función, o, en caso de que la función no devuelva valor alguno, la palabra reservada *void*.
- **nombre_de_la_función:** es el nombre o identificador asignado a la función.
- **lista_de_parámetros:** es la lista de declaración de los parámetros que son pasados a la función. Éstos se separan por comas. Debemos tener en cuenta que pueden existir funciones que no utilicen parámetros.
- **cuerpo_de_la_función:** está compuesto por un conjunto de sentencias que llevan a cabo la tarea específica para la cual ha sido creada la función.
- **return expresión:** mediante la palabra reservada *return*, se devuelve el valor de la función, en este caso representado por *expresión*.

Vamos a suponer que queremos crear un programa para calcular el precio de un producto basándose en el precio base del mismo y el impuesto aplicable. A continuación mostramos el código fuente de dicho programa:

```
#include <stdio.h>

float precio(float base, float impuesto); /* declaración */

main()
{
      float importe = 2.5;
      float tasa = 0.07;

      printf("El precio a pagar es: %.2f\n", precio(importe, tasa));
      return 0;
}

float precio(float base, float impuesto) /* definición */
{
      float calculo;
      calculo = base + (base * impuesto);
      return calculo;
}
```

El ejemplo anterior se compone de dos funciones, la función requerida *main* y la función creada por el usuario *precio,* que calcula el precio de un producto tomando como parámetros su precio base y el impuesto aplicable. La función *precio* calcula el precio de un producto sumándole el impuesto correspondiente al precio base y devuelve el valor calculado mediante la sentencia *return*.

Por otra parte, en la función *main* declaramos dos variables de tipo *float* que contienen el precio base del producto y el impuesto aplicable. La siguiente sentencia dentro de la función *main* es la llamada a la función de biblioteca *printf,* que recibe como parámetro una llamada a la función *precio,* que devuelve un valor de tipo *float*. De esta manera, la función *printf* imprime por la salida estándar el valor devuelto por la función *precio*. Es importante tener en cuenta que las variables *importe* y *tasa* (argumentos) dentro de la función *main* tienen una correspondencia con las variables *base* e *impuesto* (parámetros) dentro de la función *precio* respectivamente.

En el ejemplo anterior, justo antes de la función *main*, hemos declarado la función *precio*. La intención es que la función *main* sea capaz de reconocerla. Sin embargo, la definición de dicha función aparece después de la función *main*. Las definiciones de función pueden aparecer en cualquier orden dentro de uno o más ficheros fuentes. Más adelante, en esta unidad, veremos en detalle la declaración y definición de funciones. Por otra parte, hemos añadido la sentencia *return 0* al final de la función *main*, puesto que se trata de una función como otra cualquiera y puede devolver un valor a quien le ha llamado, en este caso el entorno en el que se ejecuta el programa. Generalmente, el valor *0* implica un fin de ejecución normal, mientras que otro valor diferente implica un final de ejecución inusual o erróneo.

3.2 Declaración de funciones

Antes de empezar a utilizar una función debemos **declararla.** La declaración de una función se conoce también como **prototipo** de la función. En el prototipo de una función se tienen que especificar los parámetros de la función, así como el tipo de dato que devuelve.

Los prototipos de las funciones que se utilizan en un programa se incluyen generalmente en la cabecera del programa y presentan la siguiente sintaxis:

```
tipo_de_retorno nombre_de_la_función(lista_de_parámetros);
```

En el prototipo de una función no se especifican las sentencias que forman parte de la misma, sino sus características. Por ejemplo:

```
int cubo(int numero);
```

En este caso se declara la función *cubo* que recibe como parámetro una variable de tipo entero *(numero)* y devuelve un valor del mismo tipo. En ningún momento estamos especificando qué se va a hacer con la variable *numero*, sólo declaramos las características de la función *cubo*.

Cabe señalar que el nombre de los parámetros es opcional y se utiliza para mejorar la comprensión del código fuente. De esta manera, el prototipo de la función *cubo* podría expresarse de la siguiente manera:

```
int cubo(int);
```

Los prototipos de las funciones son utilizados por el compilador para verificar que se accede a la función de la manera adecuada con respecto al número y tipo de parámetros, y al tipo de valor de retorno de la misma. Veamos algunos ejemplos de prototipos de funciones:

```
int potencia(int base, int exponente);
double area_rectangulo (float base, float altura);
int mayor(int, int);
struct direccion entrada(void);
```

Las funciones de biblioteca se declaran en lo que se conocen como **ficheros de cabecera** o **ficheros .h** (del inglés *headers*, cabeceras). Cuando deseamos utilizar alguna de las funciones de biblioteca, debemos especificar el fichero .h en que se encuentra declarada la función, al inicio de nuestro programa. Por ejemplo, si deseamos utilizar la función *printf* en nuestro programa, debemos incluir el fichero stdio.h que contiene el prototipo de esta función.

3.3 Definición de funciones

Tras declarar una función, el siguiente paso es implementarla. Generalmente, este paso se conoce como **definición.** Es precisamente en la definición de una función donde se especifican las instrucciones que forman parte de la misma y que se utilizan para llevar a cabo la tarea específica de la función. La definición de una función consta de dos partes, el **encabezado** y el **cuerpo de la función.** En el encabezado de la función, al igual que en el prototipo de la misma, se tienen que especificar los parámetros de la función, si los utiliza y el tipo de datos que devuelve, mientras que el cuerpo se compone de las instrucciones necesarias para realizar la tarea para la cual se crea la función. La sintaxis de la definición de una función es la siguiente:

```
tipo_de_retorno nombre_de_la_función(lista_de_parámetros)
{
      sentencias;
}
```

El *tipo_de_retorno* representa el tipo de dato del valor que devuelve la función. Este tipo debe ser uno de los tipos simples de C, un puntero a un tipo de C o bien un tipo *struct*. De forma predeterminada, se considera que toda función devuelve un tipo entero *(int)*. En otras palabras, si en la declaración o en la definición de una función no se especifica el *tipo_de_retorno*, el compilador asume que devuelve un valor de tipo *int*. El *nombre_de_la_función* representa el nombre que se le asigna a la función.

Se recomienda que el nombre de la función esté relacionado con la tarea que lleva a cabo. En caso de que la función utilice parámetros, éstos deben estar listados entre paréntesis a continuación del nombre de la función, especificando el tipo de dato y el nombre de cada parámetro. En caso de que una función no utilice parámetros, se pueden dejar los paréntesis vacíos o incluir la palabra *void*, que indica que la función no utiliza parámetros. Después del encabezado de la función, debe aparecer, delimitado por llaves (*{* y *}*), el cuerpo de la función compuesto por las sentencias que llevan a cabo la tarea específica de la función. Veamos la definición de la función *cubo* declarada en el apartado anterior:

```
int cubo(int base)
{
      int potencia;
      potencia = base * base * base;
      return potencia;
}
```

Como ya hemos visto, a los argumentos que recibe la función también se les suele llamar **parámetros.** Sin embargo, algunos autores consideran como parámetros a la lista de variables entre paréntesis utilizada en la declaración o en la definición de la función, y como argumentos los valores utilizados cuando se llama a la función. También se utilizan los términos **argumentos formales** y **argumentos reales,** respectivamente, para hacer esta distinción.

Cuando un programa utiliza un número elevado de funciones, se suelen separar las declaraciones de función de las definiciones de las mismas. Al igual que con las funciones de biblioteca, las declaraciones pasan a formar parte de un fichero cabecera (extensión *.h*), mientras que las definiciones se almacenan en un fichero con el mismo nombre que el fichero .h, pero con la extensión *.c*. En algunas ocasiones, un programador no desea divulgar el código fuente de sus funciones. En estos casos, se suele proporcionar al usuario el fichero de cabecera, el fichero compilado de las definiciones (con extensión *.o,* de objeto) y una documentación de las mismas. De esta manera, cuando el usuario desea hacer uso de cualquiera de las funciones, sabe qué argumentos pasarle y qué tipo de datos devuelve, pero no tiene acceso a la definición de las funciones.

Ejemplo práctico

1 El siguiente programa calcula el cubo de los números del 1 al 5 utilizando una función definida por el usuario.

```
#include <stdio.h>

int cubo(int base);

main()
{
      int numero;
      for(numero=1; numero<=5; numero++){
        printf("El cubo del número %d es %d\n", numero, cubo(numero));
      }
      return 0;
}

int cubo(int base)
{
      int potencia;
      potencia = base * base * base;
      return potencia;
}
```

La salida es:

```
      El cubo del número 1 es 1
      El cubo del número 2 es 8
      El cubo del número 3 es 27
      El cubo del número 4 es 64
      El cubo del número 5 es 125
```

3.4 Devolución de valores

Una función en C sólo puede devolver un valor. Para devolver dicho valor, se utiliza la palabra reservada *return* cuya sintaxis es la siguiente:

```
return <expresión>;
```

Donde *<expresión>* puede ser cualquier tipo de dato salvo un array o una función. Además, el valor de la expresión debe coincidir con el tipo de dato declarado en el prototipo de la función. Por otro lado, existe la posibilidad de devolver múltiples valores mediante la utilización de punteros o estructuras. Dentro de una función pueden existir varios *return* dado que el programa devolverá el control a la sentencia que ha llamado a la función en cuanto encuentre la primera sentencia *return*. Si no existen *return*, la ejecución de la función continúa hasta la llave del final del cuerpo de la función *(})*. Hay que tener en cuenta que existen funciones que no devuelven ningún valor. El tipo de dato devuelto por estas funciones puede ser *void*, considerado como un tipo especial de dato. En estos casos, la sentencia *return* se puede escribir como *return* o se puede omitir directamente. Por ejemplo:

```
void imprime_cabecera();
      {
        printf("esta función sólo imprime esta línea");
        return;
      }
```

equivale a:

```
void imprime_cabecera();
      {
        printf("esta función sólo imprime esta línea");
      }
```

3.5 Acceso a una función

Para que una función realice la tarea para la cual fue creada, debemos acceder o llamar a la misma. Cuando se llama a una función dentro de una expresión, el control del programa se pasa a ésta y sólo regresa a la siguiente expresión de la que ha realizado la llamada cuando encuentra una instrucción *return* o, en su defecto, la llave de cierre al final de la función.

Generalmente, se suele llamar a las funciones desde la función *main*, lo que no implica que dentro de una función se pueda acceder a otra función.

Cuando queremos acceder a una función, debemos hacerlo mediante su nombre seguido de la lista de argumentos que utiliza dicha función encerrados entre paréntesis. En caso

de que la función a la que se quiere acceder no utilice argumentos, se deben colocar los paréntesis vacíos.

Cualquier expresión puede contener una llamada a una función. Esta llamada puede ser parte de una expresión simple, como una asignación, o puede ser uno de los operandos de una expresión más compleja. Por ejemplo:

```
a = cubo(2);
calculo = b + c / cubo(3);
```

Debemos recordar que los argumentos que utilizamos en la llamada a una función se denominan **argumentos reales.** Estos argumentos deben coincidir en el número y tipo con los **argumentos formales** o **parámetros** de la función. No olvidemos que los argumentos formales son los que se utilizan en la definición y/o declaración de una función. Los argumentos reales pueden ser variables, constantes o incluso expresiones más complejas. El valor de cada argumento real en la llamada a una función se transfiere a dicha función y se le asigna al argumento formal correspondiente.

Generalmente, cuando una función devuelve un valor, la llamada a la función suele estar dentro de una expresión de asignación, como operando de una expresión compleja o como argumento real de otra función. Sin embargo, cuando la función no devuelve ningún valor, la llamada a la función suele aparecer sola. Veamos un ejemplo:

```
z = potencia( a, b);
imprime_valores (x, y, z);
```

Ejemplo práctico

2 Vamos a acceder a las funciones *primera* y *segunda* desde la función *main*.

```
#include <stdio.h>

void primera(void);
void segunda(void);

main()
{
      printf("La primera función llamada, main\n");
      primera();
      segunda();
      printf("Final de la función main\n");
      return 0;
}

void primera(void)
{
      printf("Llamada a la función primera\n");
      return;
}
```

```
void segunda(void)
{
      printf("Llamada a la función segunda\n");
      return;
}
```

La salida es:

```
      La primera función llamada, main
      Llamada a la función primera
      Llamada a la función segunda
      Final de la función main
```

3.6 Ámbito y clases de almacenamiento

Como hemos visto en unidades anteriores, en C, las variables se pueden clasificar de acuerdo a su tipo de dato. Por ejemplo, una variable puede ser de tipo entero *(int)* o de tipo carácter *(char)*. Sin embargo, las variables también pueden clasificarse de acuerdo a su ámbito, es decir, la parte del programa en la que la variable es reconocida. De acuerdo con su ámbito, las variables pueden ser locales o globales. Por otro lado, existen los modificadores de tipo o clases de almacenamiento que permiten modificar el ámbito y la permanencia de una variable dentro de un programa. Existen cuatro modificadores de tipo, automático, externo, estático y registro, que se corresponden con las palabras reservadas *auto, extern, static* y *register*, respectivamente.

3.7 Variables locales

Cuando declaramos variables dentro de la función principal del programa, es decir, dentro de la función *main*, están únicamente asociadas a esta función, en otras palabras, son variables locales de la función *main* y no se puede acceder a ellas a través de ninguna otra función.

Al igual que sucede con las variables declaradas dentro de la función *main*, cualquier variable que declaremos dentro de una función, es local a esa función, es decir, su ámbito esta confinado a dicha función. Esta situación permite que existan variables con el mismo nombre en diferentes funciones y que no mantengan ninguna relación entre sí.

Debemos tener en cuenta que cualquier variable declarada dentro de una función se considera como una variable automática *(auto)* a menos que utilicemos algún modificador

de tipo. Una variable se considera **automática** porque cuando se accede a la función se le asigna espacio en la memoria automáticamente y se libera dicho espacio tan pronto se sale de la función. En otras palabras, una variable automática no conserva un valor entre dos llamadas sucesivas a la misma función. Con el propósito de garantizar el contenido de las variables automáticas, éstas deben inicializarse al entrar a la función para evitar que su valor sea indeterminado.

Todas las variables que hemos utilizado en los ejemplos vistos hasta ahora son **variables automáticas.** La utilización de la palabra reservada *auto* es opcional, aunque normalmente no se utiliza, por ejemplo:

```
auto int contador;
```

equivale a

```
int contador;
```

Ejemplo práctico

3 Utilización del mismo identificador de variable en diferentes funciones mostrando su localidad.

```
#include <stdio.h>

void imprimeValor();
main()
{
	int contador = 0;
	contador++;
	printf("El valor de contador es: %d\n", contador);
	imprimeValor();
	printf("Ahora el valor de contador es: %d\n", contador);
return 0;

}

void imprimeValor()
{
	int contador = 5;
	printf("El valor de contador es: %d\n", contador);
}
```

La salida es:

```
	El valor de contador es: 1
	El valor de contador es: 5
	Ahora el valor de contador es: 1
```

3.8 Variables globales

A diferencia de las variables locales cuyo ámbito estaba confinado a la función donde estaban declaradas, el ámbito de las variables globales se extiende desde el punto en el que se definen hasta el final del programa. En otras palabras, si definimos una variable al principio del programa, cualquier función que forme parte de éste podrá utilizarla simplemente haciendo uso de su nombre.

La utilización de variables globales proporciona un mecanismo de intercambio de información entre funciones sin necesidad de utilizar argumentos. Por otra parte, las variables globales mantienen el valor que se les ha asignado dentro de su ámbito, incluso después de finalizar las funciones que modifican dicho valor. Debemos tener en cuenta que el uso de variables globales para el intercambio de informaciones entre funciones puede resultar útil en algunas situaciones (como cuando se desea transferir más de un valor desde una función), pero su utilización podría llevarnos a programas de difícil interpretación y complejos de depurar.

Ejemplo práctico

4 Utilización de variables globales como mecanismo de intercambio de información entre funciones.

```
#include <stdio.h>

void unaFuncion();
void otraFuncion();
int variable;

main()
{
	variable = 9;
	printf("El valor de variable es: %d\n", variable);
	unaFuncion();
	otraFuncion();
	printf("Ahora el valor de variable es: %d\n", variable);
	return 0;

}

void unaFuncion()
{
	printf("En la función unaFuncion, variable es: %d\n", variable);
}

void otraFuncion()
{
	variable++;
```

```
        printf("En la función otraFuncion, variable es: %d\n",variable);
}
```

La salida es:

```
El valor de variable es: 9
En la función unaFuncion, variable es: 9
En la función otraFuncion, variable es: 10
Ahora el valor de variable es: 10
```

Cuando trabajamos con variables globales debemos distinguir entre la definición de una variable global y su declaración. Cuando definimos una variable global, lo hacemos de la misma forma en que se declara una variable ordinaria. La definición de una variable global se realiza fuera de cualquier función. Además, las definiciones de variables globales suelen aparecer antes de cualquier función que desee utilizar dicha variable. La razón es que una variable global se identifica por la localización de su definición dentro del programa. Cuando se define una variable global automáticamente, se reserva memoria para el almacenamiento de ésta. Además, podemos asignarle un valor inicial a la misma.

Si una función desea utilizar una variable global previamente definida, basta con utilizar su nombre sin realizar ningún tipo de declaración especial dentro de la función. Sin embargo, si la definición de la función aparece antes de la definición de la variable global, se requiere incluir una declaración de la variable global dentro de la función.

Para declarar una variable global se utiliza la palabra reservada *extern*. Al utilizar *extern,* le estamos diciendo al compilador que el espacio de memoria de esa variable está definido en otro lugar. Es más, en la declaración de una variable externa *(extern)* no se puede incluir la asignación de un valor a dicha variable. Por otro lado, el nombre y el tipo de dato utilizados en la declaración de una variable global debe coincidir con el nombre y el tipo de dato de la variable global definida fuera de la función.

Debemos recordar que sólo se puede inicializar una variable global en su definición. El valor inicial que se le asigne a la variable global debe expresarse como una constante y no como una expresión. Es importante señalar que si no se asigna un valor inicial a la variable global, automáticamente se le asigna el valor cero (0). De esta manera, las variables globales siempre cuentan con un valor inicial.

Cabe señalar que la declaración de una variable global puede hacer referencia a una variable que se encuentra definida en otro fichero. Por esta razón, podemos decir que el especificador de tipo *extern* hace referencia a una variable que ha sido definida en un lugar distinto al punto en el que se está declarando y donde se va a utilizar.

En aplicaciones grandes compuestas de varios ficheros, es común que las definiciones de variables globales estén agrupadas y separadas del resto de ficheros fuente. Cuando se desea utilizar cualquiera de las variables globales en un fichero fuente, se debe incluir el fichero en el que están definidas las variables mediante la directiva de precompilación *#include*.

Ejemplo práctico

5 Utilización del modificador de tipo *extern*.

```
#include <stdio.h>

void unaFuncion();
void otraFuncion();


main()
{
	extern variable;
	variable = 9;
	printf("El valor de variable es: %d\n", variable);
	unaFuncion();
	printf("Ahora el valor de variable es: %d\n", variable);
	return 0;

}

void unaFuncion()
{
	extern variable;
	printf("En la función unaFunción, variable es: %d\n", variable);
}

int variable;
```

Su salida es:

```
	El valor de variable es: 9
	En la función unaFuncion, variable es: 9
	Ahora el valor de variable es: 9
```

3.9 Variables estáticas

Otro tipo de almacenamiento son las variables estáticas identificadas por la palabra reservada *static*. Las variables estáticas pueden ser tanto locales como globales. Una variable estática local, al igual que una variable automática, está únicamente asociada a la función en la que se declara con la salvedad de que su existencia es permanente.

En otras palabras, su contenido no se borra al finalizar la función, sino que mantiene su valor hasta el final del programa. Por ejemplo, en el siguiente programa declaramos la variable *contador* como estática dentro de la función *imprimeValor* y desde la función *main* llamamos a esta función varias veces:

```
#include <stdio.h>
void imprimeValor();
main()
{
      imprimeValor();
      imprimeValor();
      imprimeValor();
      imprimeValor();
      return 0;
}
void imprimeValor()
{
      static int contador = 0;
      printf("El valor de contador es: %d\n", contador);
      contador++;
}
```

La primera vez que se llama a la función *imprimeValor* se inicializa su valor a cero, y tras imprimir su valor se incrementa éste. En las sucesivas llamadas, el valor de la variable *contador* se mantiene y el resultado es el siguiente:

```
El valor de contador es: 0
El valor de contador es: 1
El valor de contador es: 2
El valor de contador es: 3
```

Como hemos visto, el valor de la variable *contador* se mantiene de una llamada a otra de la función *imprimeValor*. Esto quiere decir que las variables locales estáticas proporcionan un medio privado de almacenamiento permanente en una función.

Por otro lado, la aplicación del calificador *static* a variables globales hace que a éstas sólo se pueda acceder desde el fichero fuente en el que se definieron y no desde ningún otro fichero fuente. Por ejemplo, si definimos las variables globales *tiempo* y *reloj* en el siguiente fichero fuente:

```
static int tiempo;
static int reloj;

main()
{
      ...
}
void funcion1()
{
      ...
}
```

cualquier otra función que forme parte de la aplicación y que no forme parte de este fichero fuente no podrá disponer de acceso a *tiempo* ni a *reloj*. Es más, se pueden utilizar los mismos nombres para definir variables en funciones en otros ficheros sin ningún tipo de problema.

Al igual que las variables globales, las funciones son objetos externos cuyos nombres, generalmente, se desea que se conozcan de manera global. Sin embargo, en algunas situaciones resulta deseable limitar dicho acceso al fichero en el que se declara la función. En estos casos, se utiliza la palabra reservada *static* y su mecanismo de aplicación es similar al caso de las variables, como podemos apreciar en el siguiente ejemplo:

```
static int cuadrado (int numero)
{
      int calculo=0;
      calculo = numero * numero;
      return calculo;
}
```

Limitar el acceso tanto a variables globales como a funciones, mediante su declaración como estáticas, resulta útil en algunas situaciones en las que se quiere evitar que entren en conflicto con otras variables o funciones, incluso inadvertidamente.

3.10 Variables de registro

Cuando declaramos una variable como variable de registro, le estamos diciendo al compilador que queremos que la variable se almacene en un lugar de rápido acceso, generalmente en los registros de la unidad central de procesamiento del ordenador. Para declarar una variable como variable de registro, debemos colocar la palabra reservada *register* antes de la declaración de la variable. Veamos algunos ejemplos:

```
register int contador:
register char a;
```

Sólo se pueden declarar variables como *register* si son variables automáticas (locales) o argumentos formales de una función. Veamos un ejemplo de la utilización de *register* en los argumentos formales de una función:

```
funcion_A (register int entrada, register int numero)
{
      ...
}
```

Cuando se utiliza una variable de tipo *register*, no está garantizado que su valor se almacene en un registro. Para que el valor de dicha variable se almacene en un registro, debe haber alguno disponible, de lo contrario C ignora el calificador *register* y crea la variable localmente como ya hemos visto.

Debido a restricciones de hardware, existen ciertas limitaciones al utilizar variables *register*. Por esta razón, sólo un número determinado de variables pueden declararse como *register* dentro de una función. También existen limitaciones en cuanto al tipo de las variables. Es el compilador el que se encarga de ignorar el calificador *register* si existe un número excesivo de declaraciones o los tipos utilizados no están permitidos.

3.11 Paso de argumentos y punteros

En C todos los argumentos que se pasan a una función se pasan por valor. En otras palabras, se pasa una copia del valor del argumento y no el argumento en sí (por ello, este procedimiento se conoce en algunas ocasiones como **paso por copia**). Al pasar una copia del argumento original a la función, cualquier modificación que se realice sobre esta copia no tendrá efecto sobre el argumento original utilizado en la llamada de la función. Se puede considerar un argumento pasado por valor como una variable local de la función a la que se ha pasado, de tal modo que los cambios que se realicen sobre ésta tendrán efecto sólo dentro de la función.

Veamos un ejemplo del paso por valor de argumentos a una función:

```
#include <stdio.h>

void modificar(int variable);
main()
{
      int i = 1;
      printf("\ni=%d antes de llamar a la función modificar", i);
      modificar(i);
      printf("\ni=%d después de llamar a la función modificar", i);
}

void modificar(int variable)
{
      printf("\nvariable = %d dentro de modificar", variable);
      variable = 9;
      printf("\nvariable = %d dentro de modificar", variable);
}
```

Dado que lo que se pasa a la función *modificar* es una copia de la variable *i*, el valor de ésta en la función *main* no se ve alterado cuando dentro de la función *modificar* se cambia el valor de variable. De ahí, la salida del ejemplo anterior es la siguiente:

```
i=1 antes de llamar a la función modificar
variable = 1 dentro de modificar
variable = 9 dentro de modificar
i=1 después de llamar a la función modificar
```

Como ya hemos visto, cuando se pasa un argumento por valor, realmente se pasa una copia de éste, y si esta copia se modifica el argumento original no se ve alterado. Sin embargo, en muchas ocasiones lo que queremos es que una función cambie los valores de los argumentos que le pasamos. Para lograrlo se utiliza lo que se conoce como **paso de argumentos por referencia.** En estos casos, no se pasa una copia del argumento, sino el argumento mismo.

Cuando realizamos un paso de argumentos por referencia en C, realmente lo que estamos pasando son direcciones de memoria. En otras palabras, lo que le pasamos a la fun-

ción son las direcciones de memoria de los argumentos. Como hemos visto en la unidad anterior, esta operación se logra mediante la utilización de punteros. De este modo, cuando llamamos a una función, lo que realmente le pasamos son punteros a los argumentos que deseamos modificar. Veamos el ejemplo anterior utilizando el paso de argumentos por referencia:

```
#include <stdio.h>

void modificar(int *variable);
main()
{
      int i = 1;
      printf("\ni=%d antes de llamar a la función modificar", i);
      modificar(&i);
      printf("\ni=%d después de llamar a la función modificar", i);
      return 0;
}

void modificar(int *variable)
{
      printf("\nvariable = %d dentro de modificar", *variable);
      *variable = 9;
      printf("\nvariable = %d dentro de modificar", *variable);
}
```

La salida de este ejemplo sería:

```
i=1 antes de llamar a la función modificar
variable = 1 dentro de modificar
variable = 9 dentro de modificar
i=9 después de llamar a la función modificar
```

Como se puede observar, el valor de *i* ha cambiado puesto que la función *modificar* ha utilizado la dirección de memoria de esta variable en la sentencia de asignación **variable = 9*. Analicemos detenidamente este ejemplo. Lo primero que tenemos que tener en cuenta es que la premisa de que en C todos los argumentos se pasan por valor sigue siendo cierta. Lo que sucede es que en este caso estamos pasando el valor de la dirección de memoria de la variable *i* y no el valor de su contenido *(1)*. Para pasar la dirección de memoria de una variable se utiliza el operador *&*. Al finalizar la función, el valor de dicha dirección permanece igual y lo que se ha modificado es el contenido de esa dirección de memoria. Dentro de la función se utilizan los punteros para trabajar con las direcciones de memoria *(*variable)*.

Dado que el paso de argumentos por referencia es común en C, conviene que en este punto ampliemos el concepto de puntero. Consideremos las siguientes declaraciones:

```
int dato;
int *puntero;
```

La primera de las declaraciones reserva memoria para almacenar una variable de tipo entero *(int)* mientras que la segunda declaración reserva memoria para almacenar una

dirección. A pesar de que apunta a una variable de tipo entero, lo que se va a almacenar es una dirección. Como en el fondo un puntero puede apuntar a cualquier cosa, C permite la declaración de punteros tipo *void*. Supongamos que el compilador reserva la dirección en hexadecimal *bfffff120* para la variable *dato* y la dirección en hexadecimal *0012fed4* para puntero.

En la Figura 3.1 se muestra gráficamente la representación de la declaración de las variables anteriores

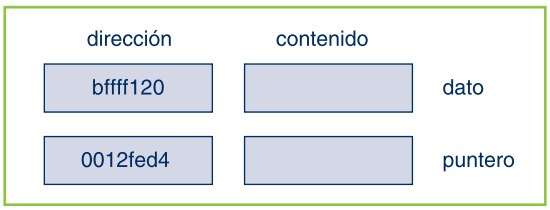

Fig. 3.1. *Declaración de variables.*

Si a continuación asignamos un valor a la variable *dato*, éste se almacenaría en la dirección de memoria *bfffff120*. En el caso de *puntero*, si le asignamos un contenido (la dirección de la variable *dato*), la dirección en memoria de la variable *dato* se almacenaría en la dirección *0012fed4*. Supongamos que realizamos las siguientes asignaciones:

```
dato = 99;
puntero = &dato;
```

El resultado en la memoria se muestra en la Figura 3.2. Debemos recordar que mediante el operador *&* obtenemos la dirección de una variable.

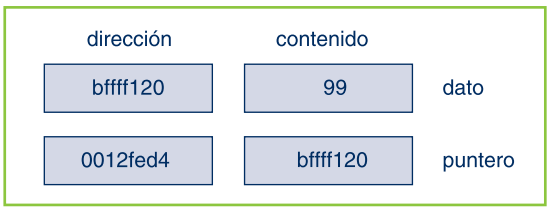

Fig. 3.2. *Asignación de valores.*

Una vez realizada la asignación anterior, disponemos de dos maneras de acceder al contenido de la variable *dato*. La primera forma de acceder es mediante el nombre de la variable y la segunda mediante el puntero que apunta a dicha variable. Para acceder a la variable dato mediante puntero, se utiliza el operador *.

Una utilidad del paso de argumentos por referencia se relaciona con la devolución de valores desde una función. Como ya sabemos, en C una función sólo puede devolver un único valor. Sin embargo, a menudo se hace necesario que una función devuelva más de un valor, y es aquí donde podemos utilizar el paso de argumentos por referencia.

Además, la utilización del paso de argumentos por referencia nos permite ahorrar tiempo y espacio. En el caso que deseáramos pasar algo a una función cuyo tamaño sea grande,

por ejemplo una estructura, sería conveniente pasarla por referencia, ya que si lo hacemos por valor, se tiene que realizar una copia de la misma y colocarla en la pila, lo que implica consumo de tiempo y espacio.

3.12 Recursividad

Las funciones en C pueden ser recursivas, en otras palabras, pueden llamarse a sí mismas directa o indirectamente. La recursividad directa es el proceso mediante el que una función se llama a sí misma desde el propio cuerpo de la función, mientras que la recursividad indirecta implica más de una función.

Un proceso recursivo tiene que tener una condición de finalización, ya que de lo contrario podría continuar infinitamente.

Un ejemplo típico de aplicación de la recursividad es el cálculo del factorial de un número entero. Recordemos que el factorial de un número entero (n!) se calcula de la siguiente manera:

```
n! = n * (n-1) * (n-2) * ... * 2 * 1
```

En principio, la solución a este problema podría realizarse sin tener que utilizar la recursividad con el siguiente programa:

```
#include <stdio.h>

int factorial(int numero);

main()
{
      int valor = 4;
      int resultado;

      resultado = factorial(valor);
      printf("El factorial de %d es %d \n", valor, resultado);
      return 0;
}

int factorial(int numero)
{
      int i;
      int devuelve = 1;
      for(i = 1; i <= numero; i++)
{
        devuelve = devuelve * i;
      }
      return devuelve;
}
```

Sin embargo, resulta más intuitivo dada la definición de número factorial utilizar una función recursiva como la siguiente:

```
int factorial(int numero)
{

      if(numero == 1)
        return 1;
      else
        return (numero * factorial(numero-1));

}
```

En la función anterior, en el caso de que el argumento utilizado en la llamada sea *1*, ésta devuelve *1*, y en caso contrario se calcula un producto que involucra a la variable *numero* y una nueva llamada a la función cuyo argumento es menor en una unidad *(numero -1)*.

El funcionamiento de una función recursiva se realiza almacenando las llamadas pendientes, con sus argumentos, en la **pila en tiempo de ejecución.** Veamos un ejemplo: imagina que utilizamos el valor *4* como argumento de la función que calcula el factorial, es decir, *factorial(4),* el proceso de llamadas será el siguiente:

```
Llamada # 1:
     numero = 4
     numero != 1 entonces ejecutamos la siguiente sentencia
     return ( 4 * (realizamos la segunda llamada))

     Llamada # 2:
         numero = 3
         numero != 1 entonces ejecutamos la siguiente sentencia
         return ( 3 * (realizamos la tercera llamada))

         Llamada # 3:
             numero = 2
             numero != 1 entonces ejecutamos la siguiente sentencia
             return ( 2 * (realizamos la cuarta llamada))

             Llamada # 4:
                 numero = 1
                 numero == 1 entonces se ejecuta la sentencia del if:
                 return 1

             Fin Llamada # 4 -> DEVUELVE 1

             return ( 2 * 1)

         Fin Llamada # 3 -> DEVUELVE 2

         return ( 3 * 2)

     Fin Llamada # 2 -> DEVUELVE 6
```

```
        return ( 4 * 6)

    Fin Llamada #1 -> DEVUELVE 24
```

En la Figura 3.3 podemos ver ilustrado el proceso descrito anteriormente.

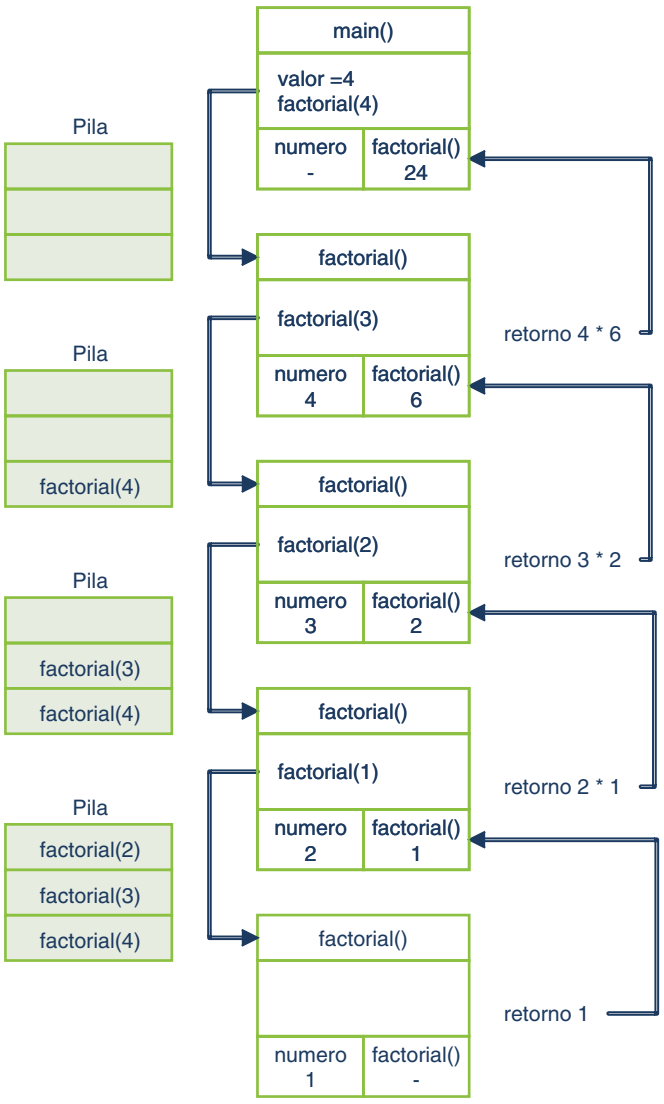

Fig. 3.3. *Llamadas recursivas.*

En muchas ocasiones, la resolución de un problema mediante una función recursiva resulta conceptualmente más clara que la resolución mediante una función iterativa. Tal es el caso de algunas estructuras de datos como los árboles binarios, cuyo manejo es sencillo mediante una función recursiva. Sin embargo, la función iterativa resulta algo más compleja. Es evidente que hay tareas que se pueden resolver mediante funciones recursivas o funciones iterativas, aunque es el programador el que tiene que optar por una solución u otra.

3.13 Funciones de biblioteca

C ofrece un conjunto de funciones estándar que dan soporte a las operaciones que se utilizan con más frecuencia. Estas funciones están agrupadas en bibliotecas. Para utilizar cualquiera de las funciones que forman parte de las bibliotecas estándar de C, sólo hace falta realizar una llamada a dicha función.

Las funciones que forman parte de la biblioteca estándar de C, funciones estándar o predefinidas, están divididas en grupos. Todas las funciones que pertenecen a un mismo grupo se definen en el mismo fichero de cabecera.
Los nombres de los ficheros cabeceras de C se muestran en la siguiente tabla:

assert	ctype	errno	float
limits	locale	math	setjmp
signal	stdarg	stddef	stdio
stdlib	string	time	

Tabla 3.1. *Nombre de los ficheros cabecera.*

Cuando deseamos utilizar cualquiera de las funciones estándar de C, primero debemos utilizar la directiva de precompilación *#include* para incluir los ficheros cabecera en nuestro programa. Por otra parte, antes de utilizar una función, primero debemos conocer las características de dicha función, es decir, el número y tipo de datos de sus argumentos y el tipo de valor que devuelve. Esta información es proporcionada por los prototipos de función.

Los grupos de funciones estándar más comunes son:

- entrada/salida estándar
- matemáticas
- de conversión
- diagnóstico
- de manipulación de memoria
- control de proceso
- ordenación
- directorios
- fecha y hora
- búsqueda
- manipulación de cadenas
- gráficos
- etcétera

Podemos incluir tantos ficheros de cabecera como nos haga falta, incluyendo los ficheros de cabecera que hemos creado y donde hemos definido nuestras funciones. En el resto de este apartado veremos algunas de las funciones de la entrada/salida estándar más utilizadas.

3.14 Entrada/salida estándar

Hasta ahora hemos utilizado la función *printf* para escribir datos en la salida estándar del ordenador (consola). Sin embargo, no es la única función para la salida de datos en C. De la misma manera, al igual que existen funciones para la salida de datos, existen funciones para la entrada de los mismos, entre las que se encuentra la función *scanf*.

La función *printf* es la salida genérica por consola que es utilizada por cualquier compilador de C. Por otra parte, la entrada estándar, que generalmente es por teclado, se realiza mediante la función *scanf*. Tanto la función *printf* como la función *scanf* permiten especificar el formato en que se van a escribir o leer los datos. Esto se conoce como **entrada/salida formateada.**

C proporciona otros mecanismos de entrada/salida menos sofisticados que permiten leer o escribir simplemente un carácter por la entrada/salida estándar. Esto se realiza mediante las funciones *getchar* y *putchar*.

A. La función *printf*

Como ya hemos visto a lo largo de esta unidad y de unidades anteriores, la función *printf* básicamente imprime una cadena de caracteres sobre la pantalla del ordenador. Tanto esta función como otras funciones de entrada/salida están definidas en la biblioteca *stdio*. Por esta razón, cada vez que deseamos hacer uso de la misma tenemos que incluir el fichero stdio.h mediante la directiva de precompilación *#include <stdio.h>*.

La sintaxis de la función *printf* es la siguiente:

```
printf("cadena de control", lista de argumentos);
```

La cadena de control contiene los especificadores de formato de los argumentos. Éstos le indican a *printf* cómo han de visualizarse los argumentos por pantalla y su número. Después de la cadena de control, aparecen los argumentos de la función separados por comas.

El la Tabla 3.2 mostramos algunos de los especificadores de formato utilizados en función *printf*.

Tabla 3.2. *Especificadores de formato.*

Especificador	Tipo de argumento
%c	Carácter
%d	Decimal
%e	Notación científica
%f	Decimal en coma flotante

Especificador	Tipo de argumento
%g	Usar *%e* o *%f*, el que resulte más corto
%o	Octal
%s	Cadena de caracteres
%u	Decimal sin signo (unsigned)
%i	Entero
%x ó %X	Hexadecimal

La cadena de control puede contener tanto caracteres a imprimir como especificadores de formato de los argumentos de la función. Los especificadores de formato pueden aparecer en cualquier parte de la cadena de control.

Cuando se llama a la función *printf*, lo primero que hace es analizar la cadena de control. Aquellos caracteres imprimibles que no estén precedidos por un símbolo de tanto por ciento *(%)* se muestran por pantalla directamente. Cuando encuentra un especificador de formato dentro de la cadena de control, utiliza éste para determinar cómo se muestra el argumento correspondiente. Debemos tener en cuenta que los especificadores de formato y los argumentos deben coincidir de izquierda a derecha. Por otra parte, el número de especificadores en la cadena de control le indican a *printf* cuántos argumentos debe esperar.

El símbolo *%* se utiliza para identificar los especificadores de formato. Si queremos imprimir este símbolo dentro de la cadena de control, tendremos que utilizar dos símbolos de tanto por ciento *(%%)*.

Cuando deseamos imprimir un argumento de tipo carácter que utiliza el especificador de formato *%c*, éste debe ir encerrado entre comillas simples, por ejemplo, *'b'*. Por otro lado, si lo que deseamos imprimir es una cadena de caracteres que utiliza el especificador de formato *%s*, ésta debe ir encerrada entre comillas dobles, por ejemplo *"esto es una cadena de caracteres"*.

Por otra parte, es muy común la utilización de secuencias de escape dentro de la función *printf*, por ejemplo, el salto de línea (*\n*). En la unidad anterior hemos visto las secuencias de escape y su utilización en C.

B. La función *scanf*

Un programa puede recibir datos a través de diversas fuentes, por ejemplo el teclado o ficheros almacenados en disco. En este apartado, vamos a considerar la entrada estándar por teclado. La función de entrada más utilizada es la función *scanf* ya que es versátil y permite una entrada con formato.
Al igual que la función *printf*, la función *scanf* está definida en el fichero de cabecera

stdio.h. En términos generales, *scanf* permite leer datos de la entrada estándar, de acuerdo con el formato especificado en el primer argumento y almacenar estos datos en las variables que recibe como argumento. La sintaxis de esta función es la siguiente:

```
scanf ("cadena de control", lista de argumentos);
```

Donde la cadena de control contiene los tipos de datos y, si se lo desea, la anchura de los mismos, mientras que la lista de argumentos son las variables del tipo indicado por los especificadores de formato. Los especificadores de formato más comunes son los que hemos utilizado en la función *printf*, salvo *%h* que se utiliza para indicar el tipo de enteros *short*.

A diferencia de la función *printf* que recibe los argumentos por valor, la función *scanf* recibe los argumentos por referencia. De esta manera, puede rellenarlos con los valores leídos. Por ejemplo, la siguiente sentencia devolverá en la variable *dato* el entero leído del teclado:

```
scanf("%i", &dato);
```

Ejemplo práctico

6 Un programa que pide que se introduzcan dos cantidades, una entera y otra en coma flotante, y los escribe a continuación dentro de una frase.

```
#include <stdio.h>

main ( )
{
      int edad;
      char sexo;

      printf ("Escriba su edad (en años) y sexo (H o M):\n");
      scanf ("%i %c", &edad, &sexo) ;
      printf("Su sexo es %c y su edad %i años\n", sexo, edad);
      return 0;
}
```

Si introducimos por teclado 30 y H, la salida es:

```
      Su sexo es H y su edad 30 años
```

C. Otras funciones de entrada/salida

Existen otras funciones de entrada/salida estándar definidas en el fichero de cabecera stdio.h. Entre estas funciones están las funciones *getchar* y *putchar*.
La función *getchar* nos permite leer caracteres uno a uno. Esta función no requiere argu-

mentos y simplemente devuelve el carácter leído por la entrada estándar en forma de entero sin signo. El prototipo de esta función es:

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

Por otra parte, la función *putchar* es la función simétrica a *getchar*. Esta función recibe un único argumento que es el carácter que se imprimirá por la salida estándar. Su prototipo es:

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

Ejemplo práctico

7 En el siguiente ejemplo se muestra la utilización de las funciones *getchar* y *putchar*.

```
#include <stdio.h>

main ( )

{
      char x;
      printf("Introduzca un carácter:\n");
      x = getchar( ) ;
      printf("El carácter introducido es:\n");
      putchar (x) ;
      return 0;
}
```

La salida es:

```
Introduzca un carácter
6
El carácter introducido es:
6
```

Existen en C otras funciones de entrada/salida por consola que iremos viendo a lo largo del desarrollo de este libro. Entre esas funciones podemos mencionar las siguientes:

- *getch:* Leer un carácter sin que aparezca en pantalla.
- *getche:* Lee un carácter visualizándolo en pantalla.
- *gets:* Leer una cadena de caracteres.
- *puts:* Escribir una cadena de caracteres

Ejemplo práctico

8 El siguiente programa lee 10 caracteres por teclado y luego los imprime en una tabla en orden inverso acompañados de sus respectivos códigos ASCII.

```
#include <stdio.h>

char caracteres[10];
int i;

main()
{

      for(i=0; i<10; i++){
        printf("Introduce un carácter\n");
        caracteres[i]=getchar();
        getchar();  /* se utiliza para leer el retorno de carro */
      }

      /* Impresión del encabezado de la tabla */
      printf("**********************************\n");
      printf("CARÁCTER \tCÓDIGO ASCII\n");

      /* Bucle para imprimir los caracteres leídos en orden inverso */
      for(i=9;i>=0;i—){
        putchar(caracteres[i]);
        printf("\t\t"); /* imprime dos tabulaciones */
        printf("%d",caracteres[i]);
        printf("\n");
      }
        return 0;
}
```

Suponiendo que introducimos las 10 primeras letras del alfabeto, la salida es:

```
**********************************
CARÁCTER  CÓDIGO ASCII
j                 106
i                 105
h                 104
g                 103
f                 102
e                 101
d                 100
c                 99
b                 98
a                 97
```

Ejercicios

1 **Escribe un programa en C que implemente y haga uso de una función que calcule el factorial de un número entero para determinar los posibles podios que pueden darse en una carrera entre 8 corredores. Utiliza la fórmula de variación:**

$$V_m^n = \frac{m!}{(m-n)!}$$

2 **Escribe un programa en C que permita calcular la probabilidad de que aparezcan las figuras al escoger tres cartas de entre las de un palo de la baraja. Utiliza la fórmula de combinación:**

$$C_m^n = \frac{m!}{n!(m-n)!}$$

3 **Calcula el Máximo Común Divisor (MCD) de dos números leídos por teclado utilizando una función *sigprimo(num,comienzo)* que devuelva el siguiente divisor primo de *num* mayor que *comienzo*.**

4 **Calcula el Mínimo Común Múltiplo (MCM) de dos números leídos por teclado utilizando una función *sigprimo(num,comienzo)* que devuelva el siguiente divisor primo de *num* mayor que *comienzo*.**

```
Ejemplo (problemas 3 y 4)

Números: 120, 144

120=2^3x3x5
144=2^4x3^2

mcd= 2^3x3 =   24
mcm= 2^4x3^2x5= 720
```

5 **Escribe un programa que implemente y utilice una función para determinar si un número es positivo o negativo. Lee un número entero por teclado e imprime por pantalla si el número leído es positivo o negativo haciendo uso de la función definida.**

6 **Realiza una función que, dada una cadena de caracteres y un carácter, devuelva el número de apariciones de dicho carácter en la cadena. Realiza un programa que lea una cadena de caracteres por teclado y escriba por pantalla el número de apariciones en la cadena de cada una de las vocales haciendo uso de la función definida.**

Evaluación

1. ¿Qué es una función?
2. ¿Qué es la llamada a una función?
3. ¿Cuál es la diferencia entre argumentos formales y argumentos reales?
4. ¿La instrucción *return* sólo puede aparecer una vez dentro de una función?
5. ¿Podemos llamar a una función más de una vez desde distintas partes de un programa?
6. ¿Qué diferencia existe entre el paso de argumentos por valor y el paso de argumentos por referencia?
7. ¿De qué se vale C para implementar el paso de argumentos por referencia?
8. ¿Cuándo es recursiva una función?
9. ¿Qué son las funciones de biblioteca?
10. ¿Cómo se pasan los argumentos a la función *scanf*?

Actividades prácticas

1 Escribe un programa en C que simule una pequeña calculadora que implementa las siguientes operaciones:

- Multiplicación
- Suma
- Resta
- División
- Potencia
- Raíz

Todas las operaciones deben ser implementadas como funciones. La selección de la operación se realizará mediante un pequeño menú desplegado por pantalla. Cada operación utilizará dos operandos.

2 Se tiene una matriz de temperaturas (valores reales positivos y negativos) en distintos puntos medidas en distintos días (matriz *n* x *m*, siendo *n*, el número de puntos, y *m* el número de días). Calcula la media y la dispersión de temperatura en cada uno de esos días. La dispersión es la raíz cuadrada de la varianza, y la varianza a su vez se calcula de la siguiente forma:

$$\frac{1}{n} \sum_{i=1}^{n} (x^2 - \bar{x}^2)$$

donde $\bar{x}$ corresponde a la media y x_i a cada uno de los elementos.

Desarrolla un programa en C que contenga una función que reciba como argumento un puntero a un vector que contiene todas las temperatura tomadas en un día, o sea un vector de tamaño *n*, y que devuelva en otro vector de tamaño 2 la media y la dispersión. Para ello esta función se ayudará de otras dos funciones de tipo real que calcularán una la media y la otra la varianza; la primera recibiendo un puntero al vector de temperaturas y la segunda un puntero al vector de temperaturas y la media ya calculada. El programa principal deberá, utilizando sucesivas llamadas a la función, mostrar por pantalla la media y la dispersión para cada uno de los días (*m*). Para el problema considera cualquier valor para *n* y *m* que quieras, pero ten en cuenta que debe resultar fácil cambiarlos en el programa (uso de constantes).