Funciones

Una ***función*** es una parte de código independiente del programa principal y de otras funciones, que puede ser llamada enviándole unos datos (o sin enviarle nada), para que realice una determinada tarea y/o proporcione unos resultados.

Utilidad de las funciones

Parte esencial del correcto diseño de un programa de computadora es su modularidad, esto es su división en partes más pequeñas de finalidad muy concreta.

En C estas partes de código reciben el nombre de funciones.

Las funciones facilitan el desarrollo y mantenimiento de los programas, evitan errores, ahorran memoria y trabajo innecesario.

Una misma función puede ser utilizada por diferentes programas, y por tanto no es necesario reescribirla. Además, una función es una parte de código independiente del programa principal y de otras funciones, manteniendo también independencia entre las variables respectivas y evitando errores y otros efectos colaterales de las modificaciones que se introduzcan.

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

Definición de una función

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

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

{
declaración de variables y/o de otras funciones
codigo ejecutable
return (expresión); // no corresponde si la función retorna
                    //void
}
```

La primera línea recibe el nombre de encabezamiento (header) y el resto de la definición –encerrado entre llaves– es el cuerpo (body) de la función.

Cada función puede disponer de sus propias variables, declaradas al comienzo de su código.

Estas variables, por defecto, son de tipo auto, es decir, sólo son visibles dentro del bloque en el que han sido definidas, se crean cada vez que se ejecuta la función y 'mueren' al terminar la ejecución de la misma y permanecen ocultas para el resto del programa.

Si estas variables de definen como static, conservan su valor entre distintas llamadas a la función.

También pueden hacerse visibles a la función variables globales definidas en otro archivo (o en el mismo archivo, si la definición está por debajo de donde se utilizan), declarándolas con la palabra reservada extern.

El código ejecutable es el conjunto de instrucciones que deben ejecutarse cada vez que la función es llamada.

La lista de argumentos con tipos, también llamados argumentos formales, es una lista de declaraciones de variables, precedidas por su tipo correspondiente y separadas por comas (,).

Los argumentos formales son la forma más natural y directa para que la función reciba valores desde el programa que la llama, correspondiéndose en número y tipo con otra lista de argumentos -los argumentos actuales- en el programa que realiza la llamada a la función. Los argumentos formales son declarados en el encabezamiento de la función, pero no pueden ser inicializados en él.

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

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

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

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

```
double valor_abs(double x)

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

Declaración y llamada de una función

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

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

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

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

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

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

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

tipo_valor_de_retorno nombre_funcion(lista de tipos de argumentos);

Esta forma general coincide sustancialmente con la primera línea de la definición –el encabezamiento-, con dos pequeñas diferencias: en vez de la lista de argumentos formales o parámetros, en el prototipo basta incluir los tipos de dichos argumentos. Se pueden incluir también identificadores a continuación de los tipos, pero son ignorados por el compilador.

Además, una segunda diferencia es que el prototipo termina con un carácter (;). Cuando no hay argumentos formales, se pone entre los paréntesis la palabra void, y se pone también void precediendo al nombre de la función cuando no hay valor de retorno.

Los prototipos permiten que el compilador realice correctamente la conversión del tipo del valor de retorno, y de los argumentos actuales a los tipos de los argumentos formales. La declaración de las funciones mediante los prototipos suele hacerse al comienzo del archivo, después de los #define e #include.

En muchos casos –particularmente en programas grandes, con muchos archivos y muchas funciones–, se puede crear un archivo (con la extensión .h) con todos los prototipos de las funciones utilizadas en un programa, e incluirlo con un #include en todos los archivos en que se utilicen dichas funciones.

La llamada a una función se hace incluyendo su nombre en una expresión o sentencia del programa principal o de otra función. Este nombre debe ir seguido de una lista de argumentos separados por comas y encerrados entre paréntesis. A los argumentos incluidos en la llamada se les llama argumentos actuales, y pueden ser no sólo variables y/o constantes, sino también expresiones.

Cuando el programa que llama encuentra el nombre de la función, evalúa los argumentos actuales contenidos en la llamada, los convierte si es necesario al tipo de los argumentos formales, y pasa copias de dichos valores a la función junto con el control de la ejecución.

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

Cuando se llama a una función, después de realizar la conversión de los argumentos actuales, se ejecuta el código correspondiente a la función hasta que se llega a una sentencia return o al final del cuerpo de la función, y entonces se devuelve el control al programa que realizó la llamada, junto con el valor de retorno si es que existe (convertido previamente al tipo especificado en el prototipo, si es necesario).

Recuérdese que el valor de retorno puede ser un valor numérico, una dirección (un puntero), o una estructura, pero no una matriz o un vector.

La llamada a una función puede hacerse de muchas formas, dependiendo de qué clase de tarea realice la función. Si su papel fundamental es calcular un valor de retorno a partir de uno o más argumentos, lo más normal es que sea llamada incluyendo su nombre seguido de los argumentos actuales en una expresión aritmética o de otro tipo.

En este caso, la llamada a la función hace el papel de un operando más de la expresión. Obsérvese cómo se llama a la función seno en el ejemplo siguiente:

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

En otros casos, no existirá valor de retorno y la llamada a la función se hará incluyendo en el programa una sentencia que contenga solamente el nombre de la función, siempre seguido por los argumentos actuales entre paréntesis y terminando con un carácter (;).

Por ejemplo, la siguiente sentencia llama a una función que multiplica dos matrices (nxn) A y B, y almacena el resultado en otra matriz C. Obsérvese que en este caso no hay valor de retorno (un poco más adelante se trata con detalle la forma de pasar vectores y matrices como argumentos de una función):

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

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

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

```
// archivo prueba.c
#include <stdio.h>

double valor_abs(double); // declaración

void main (void)
{
double z, y;
y = -30.8;
z = valor_abs(y) + y*y; // llamada en una expresion
}
```

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

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

Se examina a continuación un ejemplo que encuentra el promedio de dos enteros:

```
float encontprom(int num1, int num2)
{
   float promedio;

   promedio = (num1 + num2) / 2.0;
   return(promedio);
}

main()
{
   int a=7, b=10;
   float resultado;
```

```
    resultado = encontprom(a, b);
    printf("Promedio=%f\n",resultado);
}
```

Funciones void

Las funciones void dan una forma de emular, lo que en otros lenguajes se conocen como procedimientos (por ejemplo, en PASCAL). Se usan cuando no requiere regresar un valor. Se muestra un ejemplo que imprime los cuadrados de ciertos números.

```
void cuadrados()
{
    int contador;

    for( contador=1; contador<10; contador++)
        printf("%d\n",contador*contador);
}

main()
{
    cuadrados();
}
```

En la función cuadrados no esta definido ningún parámetro y por otra parte tampoco se emplea la sentencia return para regresar de la función.

Funciones para cadenas de caracteres

En C, existen varias funciones útiles para el manejo de cadenas de caracteres. Las más utilizadas son: strlen(), strcat(), strcmp() y strcpy(). Sus prototipos o declaraciones están en el archivo string.h, y son los siguientes:

Función Strlen()

El prototipo de esta función es como sigue:

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

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

Función Strcat()

El prototipo de esta función es como sigue:

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

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

Funciones Strcmp() Y Strcomp()

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

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

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

Función Strcpy()

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

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

Explicación: Su nombre proviene de string copy y se utiliza para copiar cadenas. Utiliza como argumentos dos punteros a carácter: el primero es un puntero a la cadena copia, y el segundo es un puntero a la cadena original. El valor de retorno es un puntero a la cadena copia s1. Es muy importante tener en cuenta que en C no se pueden copiar cadenas de caracteres directamente, por medio de una sentencia de asignación. Por ejemplo, sí se puede asignar un texto a una cadena en el momento de la declaración:

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

Sin embargo, sería ilícito hacer lo siguiente:

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

Punteros como valor de retorno

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

- Es llamado por el programa principal o por otra función.
- Recibe datos a través de una lista de argumentos, o a través de variables extern.

función el nombre de otra función. Por ejemplo, si pfunc es un puntero a una función que devuelve un entero y tiene dos argumentos que son punteros, dicha función puede declararse del siguiente modo:

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

El primer paréntesis es necesario pues la declaración:

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

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

```
#include <stdio.h>
#include <math.h>
void main(void){
double (*pf)(double);
*pf = sin;
```

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

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