Funciones En C

A medida que los programas crecen en extensión y complejidad la resolución se torna más complicada y su depuración y modificaciones futuras resultan casi imposibles. Para resolver este tipo de problemas lo que se hace es dividir el programa en módulos más pequeños que cumplan una tarea simple y bien acotada (modularización). Con esto se logra:
a. El programa es más simple de comprender ya que cada módulo se dedica a realizar una tarea en particular.
b. La depuración queda acotada a cada módulo.
c. Las modificaciones al programa se reducen a modificar determinados módulos.
d. Cuando cada módulo está bien probado se lo puede usar las veces que sea necesario sin volver a revisarlo.
e. Se obtiene una independencia del código en cada módulo.

Lo que llamamos en forma genérica como “módulo” en el lenguaje C son las funciones. Una función es una rutina o conjunto de sentencias que realiza una determinada labor. Una función es un programa que realiza una tarea determinada y bien acotada a la cual le pasamos datos y nos devuelve datos.

Este programa se ejecuta cuando se lo llama (llamada a la función). Como ejemplo de funciones conocidas son: **getche ( ); scanf ( ); gets ( ); printf ( ); strcpy ( ); ...etc.** Por ejemplo la función **getche( )**. A esta función usted no le pasa nada y ella le devuelve el carácter que leyó del teclado. Cuando la llama se escribe:
respuesta = getche( );

El código que se ejecuta cuando se llama a la función **getche( )** está dentro de las librerías que vienen con el compilador, usted no conoce ese código (independencia del código) pero sabe que es lo que hace y que lo hace bien.

En C todas las funciones devuelven un valor, que por defecto es un entero. Las funciones admiten argumentos, que son datos que le pasan a la función las sentencias que la llaman estos pueden ser enviados “por valor” ó “por referencia” (punteros).

Declaración, llamado y definición de funciones:
La forma general de declarar una función es la siguiente:
Tipo_devuelto Nombre_de_funcion (tipo variable_1, tipo variable_2 , ... , tipo variable_N) ;

donde:
Tipo_devuelto /* es el tipo de dato que devuelve la función luego de terminada su ejecución */
Nombre_de_función /* es el nombre de la función */
tipo variable_1 /* es el tipo y nombre de la variable que se le pasa a la función */

Observe las 3 instancias donde se hace mención a las funciones en el código fuente:

1 **Declaración de las funciones**: En este lugar se declaran todas aquellas funciones propias que se van a utilizar durante el programa. La forma de declararlas es como se explicó al comienzo, teniendo en cuenta que cada declaración de función termina con punto y coma ‘;’ A las declaraciones que se realizan en este punto se las llama prototipo de la función. Los prototipos son necesarios para que el compilador verifique que sean coherentes los tipos de datos declarados en el prototipo contra los que realmente se usan en la llamada a la función.

2 **Llamada a la función:** Cuando se llama a la función es para usarla, es decir para que realice su trabajo. Desde cualquier parte de main ( ) o desde otra función se puede hacer la llamada. En el momento en que se produce la llamada a la función se produce un salto en el flujo del programa. En ese momento se pasa a ejecutar el código de la función que va a terminar de ejecutarse al encontrar la sentencia return o llegar a la llave que cierra la función. Cuando la función finaliza se sigue ejecutando el código de la función que produjo la llamada.

3 **Definición de las funciones:** Esta es la parte en la cual se escribe el código de la función (tal cual lo hacía con main ( )).

Un ejemplo sencillo para ver cada una de las partes puede ser el siguiente: Ingresar 2 números enteros y por medio de una función calcular la suma de los mismos

```
#include <stdio.h>
int  suma  (int,  int);    /*  Declaración  de  la  función.  Observe  que  el
prototipo de la función termina con ; */

void main (void)
{
     int x, y, z ;
     printf ("ingrese numero a sumar: ");
     scanf ("%d", &x);
     printf ("ingrese numero a sumar: ");
     scanf ("%d", &y);
     z = suma (x, y);  // llamada a la función
     printf ("La suma es %d", z);
}

int suma(int a, int b)
{
     int total;
     total = a + b;
     return total;
}
```

En el ejemplo de arriba y en negrita aparecen los lugares donde interviene la función.

Observe que el prototipo termina con ‘;’. La función se llama suma, se le pasan 2 variables enteras como parámetros y devuelve un entero. A las variables que se le pasan a la función se la llaman **parámetros formales.**

Cuando se llama a la función se le pasan las variables con las que tiene que trabajar, en este momento a las variables se las llama parámetros actuales.

En la definición de la función se vuelve a escribir el prototipo pero ahora no se coloca el ‘;’ al final de la línea. En este lugar se escribe el código de la función tal cual se escribe en el main ( ). Observe que la última línea contiene una instrucción return que lo que hace es terminar la función y devolver el valor que se encuentra a continuación, en este caso devuelve el valor que tiene la variable total.

¡ Argumento Formal ó Parámetro Formal !, ¿ Pero, qué estás diciendo ?

Sí, parámetro ó argumento formal. Relativo o referente a la forma sin importar en principio el contenido. Los técnicos en desarrollo de aplicaciones informáticas hablamos de argumentos ó parámetros y nos referimos a ellos cuando decidimos qué tipo de dato va a recibir una función cuando la invoquemos en una llamada. Decidimos la forma. Es muy importante la rigurosidad en llamar a cada cosa por su nombre. En el **lenguaje de programación C,** cuando llega el momento de definir tus propias funciones, obcecado en que tienes que dividir y dividir tu programa en módulos sucesivos hasta que la realización de la tarea sea sencilla de implementar abordándola con una programación estructurada lo más comprensible posible, uno piensa en una función siendo lo primero que se le pasa por la cabeza el prototipo de la misma; esto es, decir qué va ha recibir y qué va a devolver. En el **“qué va a recibir”** es donde toma sentido el concepto de **“argumento formal”** hasta tal punto que nos importa tanto la forma que podemos poner un prototipo de una función simplemente indicando el tipo sin necesidad de indicar el identificador del mismo. Veamos un ejemplo: supongamos que deseamos una función que reciba como parámetros dos números de tipo entero y nos devuelva el resultado de la división entre ellos en forma de un flotante de simple precisión. Inmediatamente nuestra mente ha creado el prototipo formal de la misma, diciendo “necesito dos argumentos, ambos son números enteros y devolverá un número que será un real, por ejemplo”. En código y representando el primer caso, el prototipo de nuestra función quedaría de esta forma:

float divi (int, int) ; ó float divi (int *, int) ;

Los argumentos están muy ligados a las funciones y es que la utilización de las palabras de estos conceptos en su forma correcta ayuda a comprender el paso de parámetros, la creación del prototipo y la definición de la función. De tal forma que lo correcto es llamar argumento actual cuando realizamos la llamada o invocación de una función y formal cuando nos

referimos a su prototipado o definición (definir una función es implementar su código, el **"qué hace"** la función). Una función cuyo argumento formal es un entero y opera con él para elevarlo a su cuadrado, es un entero, para lo que está preparada para recibir y su argumento actual modificará de forma, pero lo que está claro es que su forma es la de los enteros que sabemos que tiene diferente forma que los reales o las cadenas de caracteres. **int cuadrado(int);**

NOTA: Las variables definidas como parámetros formales de la función son variables locales de la misma.

Algo más sobre retorno de valores:

```
int lista ( ) {
     return 1;
}
```

/* devuelve el entero: 1 cada vez que es llamada. En C podemos devolver cualquier tipo de datos de los llamados escalares. Los tipos de datos escalares son los punteros, tipos numéricos y el tipo carácter. En C no se pueden devolver arreglos ni estructuras. */

Paso de parámetros a una función

Utilizando la lista de argumentos podemos pasar parámetros a una función. En la lista de parámetros se suele colocar un conjunto de identificadores, separados por comas, que representar cada uno de ellos a uno de los parámetros de la función. Observar que el orden de los mismos es importante. Para llamar a la función habrá que colocarlos en el orden en que la función los espera. Cada parámetro puede tener un tipo diferente. Para declarar el tipo de los parámetros añadiremos entre los paréntesis '( )' una lista de declaraciones, similar a una lista de declaraciones de variables. Es habitual colocar los parámetro en una línea, tabulados hacia la derecha, separados por comas ‘,’. Así:

```
void imprime (int numero, char letra)
{
     printf ("%d, %c\n", numero, letra);
}
/* es una función que admite dos variables, una entera y otra de tipo
carácter. */
```

Paso de parámetros por valor y por referencia

En los lenguajes de programación estructurada hay dos formas de pasar variables a una función: por referencia o por valor. Cuando la variable se pasa por referencia (a través del uso de punteros) la función puede acceder a la variable original. Sin embargo cuando los parámetros se pasan por valor, la función recibe una copia de las variables, y no puede acceder a las originales. Cualquier modificación que efectuemos sobre un parámetro no se reflejará en la variable original. Esto hace que no podamos alterar el valor de la variable por equivocación. Sin embargo, en determinadas ocasiones necesitaremos alterar el valor de la variable que le pasamos a una función. Para ello en C se emplea el mecanismo de los punteros, que se ve mas adelante.

Declaración y comprobación de tipos

Al igual que para las variables, cuando una función se va a usar en un programa antes del lugar donde se define, o cuando una función se define en otro fichero (funciones externas), la función se debe declarar. La declaración de una función consiste en especificar el tipo de datos que va a retornar la función. Esto es obligatorio cuando vamos a usar una función que no devuelve un entero. Además en la declaración se puede especificar el número de argumentos y su tipo. Una declaración típica de función es: tipo identificador (lista_de_argumentos_con_tipo);

Esto avisa al precompilador que la función ya existe, o que la vamos a definir después. La lista de argumentos con tipo difiere de la lista de argumentos antes presentada en que el tipo de cada argumento se coloca dentro de la lista, antes de su correspondiente identificador, como hacíamos en la definición de variables. Por ejemplo: **char print (int, int);** declara una función que devuelve un carácter y tiene dos parámetros, un entero y un carácter. La lista de argumentos permite al compilador hacer comprobación de tipos, ya que el tipo y número de argumentos debe coincidir en la declaración, definición y llamada a una función. Este tipo de especificación del tipo de argumentos también se puede emplear en la definición de las funciones, aunque lo contrario no es posible. Así:

```
char print (int numero, int letra)
{
     printf("%d, %c\c", numero, letra);
}
/* es otra definición válida para la función print ( ) que hemos empleado.
*/
```

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 ( ); /* declaración de la función "cuadrados ()" */
int main ( ) {
     cuadrados( ); /* llamado a la función "cuadrados ()" */
}

void cuadrados( )  /* definición de la función "cuadrados ()" */
{
     int contador;
     for (contador = 1; contador < 10; contador++)
     {
          printf(" %d \n", contador * contador);
     }
}
```

En la función cuadrados() no esta definido ningún parámetro, y por otra parte tampoco se emplea la sentencia return para ‘retornar’ valores desde la función.

Diseño de una función

El diseño de la función es una tarea que debe llevar un tiempo importante para su análisis ya que de acuerdo a la forma en la que se piensa nos puede servir para un solo programa o para muchas aplicaciones. Cada una de estas tres etapas que se comentan redundará en la eficiencia de las funciones, en la posibilidad de “re- usarlas” en otros programas y en la legibilidad del código. Para el diseño de la función debemos tener en cuenta 3 puntos:

- Determinación de la tarea a realizar: No tener en claro este punto significa no saber que hacer, hasta donde llegar y menos aún que código escribir. Se necesita tener muy en claro ‘qué es lo que va a hacer la función’. Una vez que esta clara la tarea se debe acotar hasta donde se quiere llegar con los “chiches” que se le agreguen, si no establecemos dicha cota resulta en que nunca se termina de escribir la función por que siempre se le quiere agregar algo mas lindo. Por lo tanto es importante definir en este punto cual es la tarea BÁSICA de la función mas los “chiches” que se le deseen agregar.

- Determinación de los parámetros formales: Cuando ya tenemos en claro que es lo que hay que hacer se debe determinar cómo obtengo los datos para realizar la tarea que debo hacer. Por ejemplo si mi función lee algo del teclado no es necesario pasarle datos con lo cual

el parámetro formal será VOID. En cambio si mi función se dedica a realizar alguna búsqueda podemos pensar 2 alternativas:

1. El dato a buscar se lo paso a la función
2. El dato a buscar lo lee la función

- De acuerdo al origen de los datos los parámetros formales cambian en cantidad y tipo. La respuesta a cuál de las opciones a elegir la da el sentido común y el análisis de cuál de las 2 formas es más genérica. Entonces en este momento se decide de acuerdo al origen de los datos cuales van a ser los parámetros formales de la función.

- Determinación del valor a retornar: Finalmente resta definir cuál será el tipo de dato que retorna la función. Al igual que en el punto anterior el valor a retornar va íntimamente relacionado con la tarea que realice la función.