CAPITULO 6: FUNCIONES

1. INTRODUCCIÓN

Un problema de programación en C se resuelve descomponiéndolo en varias partes. Cada una de estas partes se puede asociar a una función que resuelva su fracción correspondiente al problema. A su vez cada una de estas funciones se puede descomponer de la misma forma en varias funciones hasta que no sea necesario. De esta forma, el programa tendrá la siguiente forma.

```
main()              fun_1()
{                   {
fun_1();            fun_11();
fun_2();            fun_12();
.....               .....
fun_n();            fun_1n();
}                   }
```

Al diseñar una función, se deben tener en cuenta el número de tareas elementales a realizar y el tamaño de su código. Una función debe detener un mínimo de tareas a ser posible una sola y que su código no sea excesivamente mayor para poder leerlo y corregirlo.

Todo compilador comercial trae una o varias librerías de funciones de toda índole; matemáticas, de entrada/salida, manejo de textos, manejo de gráficos, etc, que solucionan la mayor parte de los problemas básicos de programación .

Sin embargo será inevitable que en algún momento tengamos que crear nuestras propias funciones, las reglas para ello son las que desarrollaremos en este capítulo.

Convencionalmente en C los nombres de las funciones se escriben en minúscula y seguidos por un par de paréntesis. Dentro de estos paréntesis estarán ubicados los datos que se les pasan a las funciones; "parámetros formales". El número de los argumentos puede ser uno, ninguno ó una lista de ellos separados por comas.

```
                    Llamadas a funciones.

getch();            1.   Sin argumentos.
pow10(a);           2.   1 argumento.
strcmp(s1, s2);     3.   2 argumentos separados por una coma.
```

Las funciones pueden devolver o no datos con lo cual la invocación es distinta.

```
                    Datos devueltos por una función.

                    1.   Borra la pantalla y no devuelve datos de int
clrscr();           2.   valor devuelto se asigna a la variable c
c = getch();
```

2. DECLARACIÓN Y DEFINICIÓN DE LAS FUNCIONES

Prototipos de las funciones

El prototipo de una función es una declaración previa a la definición de dicha función. El prototipo sirve exclusivamente para informarle al compilador del tipo de datos que debería esperar una función, el número de los argumentos así como el tipo de dato a devolver.

Los prototipos no son obligatorios en C, sin embargo son aconsejables ya que facilitan la comprobación de error entre las llamadas de funciones y la definición correspondiente.

La declaración debe anteceder a la propia definición de la función. Igual que los HEADER, tales como **stdio.h** y otros, contienen los prototipos de las funciones y otras definiciones.

```
sintaxis:
tipo_dato  nombre_funcion(lista_tipo_argumentos);
```

donde **tipo_dato** es el tipo de dato de valor devuelto por la función. En caso de obviarse el mismo se toma el tipo **int** por defecto. Para evitar malas interpretaciones es conveniente explicitarlo. Cuando la función no devuelve ningún valor, esto se realiza por medio de la palabra **void** (sin valor), pareciéndose en este caso a los procedimientos de Pascal. **lista_tipo_argumentos** representa los tipos de datos de los argumentos que pueden ir acompañados o no de los nombres de los argumentos. Una función que no necesita argumentos se informa mediante la palabra **void** puesta entre los paréntesis. Más adelante se profundizará sobre el tema de los argumentos y sus características.

```
                                    Prototipos de funciones.

float suma(float x, float y);       1.   La función suma tiene 2 argumentos de
                                    tipo float y devuelve un float.
                                    2.   Misma función que antes,
float suma(float , float );         sin los nombre de los argumentos.
                                    3.   Una función que no devuelve un dato.
void leer_comando(char );           4.   Una función sin argumentos y
                                    devuelve un int.
int error(void);
```

Definición de las funciones

La definición de una función puede ubicarse en cualquier lugar del programa, con sólo dos restricciones: debe hallarse después de dar su prototipo, y no puede estar dentro de la definición de otra función (incluida main()). Es decir, a diferencia de Pascal, en C las definiciones no pueden anidarse.

La definición debe comenzar con un encabezamiento, que debe coincidir totalmente con el prototipo declarado para la misma, y a continuación se escribirán las sentencias que componen el cuerpo de la función que debería ir encerrado por llaves. En la definición de una función, el encabezamiento no termina con un punto y coma.

```
#include <stdio.h>

float suma(float x, float x);
float resta(float x, float y);

main()
{
float x = 1.0, y = 2.0, z;
  z = suma(x, y);
  printf("z = %f \n", z);
  z = resta(x, y);
}
float suma(float x, float y)
{
 return(x+y);
}
float resta(float x, float y)
{
 return(x+y);
}
```

```
  Declaración, definición y llamada a
  una función.
1.   Prototipo de las funciones.
Fíjese que terminan con ";".




2.   Llamada a la función suma.

3.   Llamada a la función resta.

4.   Definición de la función suma,
la definición no termina con ";"


5. Definición de la función resta.
```

Salida de una función (Sentencia return)

Para salir de un función se utiliza la sentencia **return** que produce la salida inmediata de la función hacia el código que la invocó. La sentencia puede devolver un valor acorde con el tipo usado al declarar la función que puede ser una constante, una variable o un expresión. En caso de obviarse la sentencia **return**, la función terminará con se ejecución de la última instrucción. En este caso la función devuelve el valor cero a excepción de las funciones de tipo **void**, que no devuelve ningún valor.

```
int fun1()
{
 float z;
 z = x+y;
 return z;
}
int fun2()
{
z = x+y;
}
```

```
      Salida de una función.




1.   La función fun1() devuelve un entero
a pesar de que z sea un float.


2.   La función fun2() devuelve 0(cero)
por obviarse la sentencia return.
```

Agrupamiento de los prototipos

Es recomendable agrupar todos los prototipos de las funciones de un programa en uno o varios ficheros de cabecera (un fichero **.h**). Este fichero se guarda en el directorio de los **includes** si se usa mucho, ó en el directorio activo e incluirlo como fichero de encabezamiento en los programas que hagan referencia a esas funciones. De la misma forma, se pueden agrupar las constantes y las variables globales, sobre todo si abundan en nuestro programa.

Archivo: mi_cabecera.h

```
#define FALSO 0
#define CIERTO 1

void pausa(void);
void menu(void);
int seleccion(void);
double leer_valor(void);
```

Archivo: mi_programa.c

```
#include <stdio.h>
#include "mi_cabecera.h"

main()
{
 ......
}
```

3. PASO DE PARÁMETROS A UNA FUNCIÓN

Los parámetros formales de una función son variables locales que se crean al comenzar la función y se destruyen cuando termina. El tipo de dato de cada parámetro formal debe ser el mismo que el tipo de los argumentos se utilicen al llamar a la función. Este error no se detecta en la compilación y para remediarlo, se deben usar los prototipos de funciones.

El único caso en el que los tipos pueden no coincidir los parámetros formales y los argumentos es cuando el argumento es de tipo **char** y el parámetros formal es de tipo **int;** la conversión, en este caso, se hace a **int.**

Los parámetros de una función pueden ser valores (**llamada por valor**) o direcciones (**llamada por referencia**)

Llamada por valor

El parámetro de la función recibe el valor de la variable que se utiliza como argumento. Cualquier modificación sobre estos parámetros no afecta a las variables que se utilizan para llamar a la función, puesto que el parámetro en estos casos es una copia de la variable.

Solo se pueden pasar por valor los tipos atómicos, es decir no son arrays ni estructuras.

```
int suma(int, int);

main()
{
float x = 1.0, y = 2.0, z;
......
z = suma(x, y);
....
}
float suma(float x, float y)
{
return(x + y);
}
```

Paso de argumentos por valor.

1. Dentro de la función suma(), las variables x e y son copias de las variables x e y de main().

Llamada por referencia

Cuando un argumento es una dirección, el parámetro recibe la dirección de la variable que se ha pasado como argumento al invocar a la función. Por lo tanto, el parámetro deberá declararse como un puntero y de esta forma se puede modificar el contenido de las variables.

Si una función tiene que devolver más de un valor, lo hará utilizando sus parámetros y necesariamente los argumentos deben pasarse por referencia.

```
Void cursor(int *, int *);

main()
{
int fila, columna;
...
cursor(&fila, &columna);
....
}

void cursor(int *x, int *y)
{
....
*x =20;
.....
}
```

Paso de argumentos por referencia.

1. Declaración de 2 variables para la posición del cursor.
2. En la llamada a la función, los argumentos son las direcciones (&fila, &columna).

3. En la declaración de la función, los parámetros formales, representan punteros a tipo int.
4. Se asigna el valor 20 al sitio donde apunta el puntero x, que es fila.

```
main()
{
int x = 10, y = 20;

alternar(&x, &y);
printf("x = %d, y = %d", x,y);
}
void alternar(int *px, int *py)
{
int temp;

temp = *px;
*px = *py;
*py = temp;
}
```

```
Paso de argumentos por referencia.


1.   Los valores de x e y han sido
intercambiados.
```

La única manera de pasar un array o una estructura más compleja a una función es por referencia, pero esto lo veremos más tarde.

4. VISIBILIDAD DE LAS FUNCIONES

Una función declarada en un archivo tiene un alcance global; es decir, se puede invocarla desde cualquier punto dentro este archivo. Cuando se quiere acceder a ella desde otro archivo, hay que incluir en este último la declaración de dicha función.

A diferencia de las variables externas, donde es necesario modificar el tipo de almacenamiento mediante la palabra reservada **extern,** las funciones no necesitan de este modificador: una función puede ser invocada desde cualquier módulo. El uso de la palabra **extern** se usa para facilitar la compresión del código.

Si quiere restringir el uso de la función a un solo módulo, se usa el modificador **static** en su declaración. Esta característica permite diseñar módulos con información y herramientas privadas.

```
  Archivo1.
Main()
{
 ...
}
fun1()
{
 ...
}
```

```
 Archivo2.
fun2()
{
 ...
}
static fun3()
{
 ...
}
```

```
   Visibilidad de las funciones.

1.   la función fun3() sólo se podrá
     invocar desde fun2().

2.   fun1() y fun2() se podrán invocar
     desde cualquier punto de los 2
     archivos.
```

5. RECURSIVIDAD

El lenguaje C permite que una función pueda llamarse a sí misma. A esta característica se le denomina recursividad.

```
long int factorial(int n)
{
 if(n <= 1) return (1)
 else
  return (n*factorial(n-1))
}
```

```
     Calcular el factorial de n.
1.   La función factorial se llama a sí
     misma.
2.   Nótese que tiene una condición de
     terminación.
```

Cuando se ejecuta un programa recursivo, las llamadas repetidas a la misma función no se ejecutan inmediatamente. Lo que se hace es almacenarlas en una pila hasta que se encuentre la condición de terminación de la recursividad y luego se ejecutan en orden inverso.

```
  Orden de llamadas

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

```
  Orden de ejecución

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