P

Funciones y scope de variables

Pablo R. Ramis

1. FUNCIONES

El C es un lenguaje que se diseñó una programación modular, o sea, pequeños fragmentos de código independientes unos de otros que conforman un programa mayor. Estos fragmentos podrán ser llamados en distintas oportunidades por el resto del programa en la medida que se necesite.

Son a estas porciones de código a las que llamamos **funciones**.

Teniendo en cuenta esto, cuando nos sentemos a realizar un programa, a codificarlo, debemos antes diseñarlo y descomponerlo en funciones que interactuen entre si y tener en cuenta que una función tiene un objetivo y solo uno o sea, que cumpla con una sola tarea.

1.1. Conceptos básicos

Una función en C, en general obedecen a las siguientes reglas o características:

1. Cada función debe tener un nombre que la identifique.
2. Los nombre son elegidos y asignados por el programador de acuerdo con las mismas reglas que rigen para la asignación de nombre de variables, por lo tanto, una vez definida una función para un programa, este nombre será único y no deberá repetirse en el mismo programa.
3. Todos los nombres de funciones son seguidos por un juego de parentesis a continuación. Este espacio se reserva para enumerar las variables que recibirá de entrada, para uso de dicha función, las variables son opcionales, los paréntesis no.
4. El cuerpo de una función comienza inmediatamente despues del paréntesis de cierre incluido en el nombre, debe estar encerrado entre llaves, por lo tanto, es un bloque que contiene dentro suyo una o más sentencias.
5. Las funciones poseen un retorno, el cual podriamos considerar como el resultado de la función, podría ser un valor o una variable del tipo que se defina, también podemos indicar que la función no retornará nada específico. En el primer caso, cuando hay un retorno, la función finaliza al encontrar la palabra **return**, en el segundo caso, la función finaliza con la llave de cierre.

```
[valor de retorno] nombre_de_funcion([variable1, variable2, ..., variableN])
{ /* apertura de llave para el inicio */

        /*cuerpo de la ófuncin*/

} /* cierre de llave para el final */
```

1.2. Prototipos

Bajo el mismo concepto de modulización que se mencionaba antes, hay ciertas recomendaciones útiles y que dan prolijidad y ordenamiento al código.

Una de estas es el uso del prototipado de las funciones antes de su definición y uso. Esto implica que previo a todo, yo declararé la función, retorno, nombre y parametros de entrada, para después se define, o sea se desarrolla, y usarla. Ejemplo:

```
int f(int, int);

int f(int x, int y){
        return x + y;
}
```

Como vemos, en la linea 1 vemos el prototipo de la función f, nos indica que retorna un entero, y recibe dos enteros copmo entrada.

En la línea 3, se declara la función, se indica los nombres que corresponden a las variables de entrada y en su cuerpo se está retornando la suma de los dos enteros recibidos, o sea, se retorna un entero. Los parametros de entradas deben respetar el tipo y el orden que se establecieron en el prototipo.

Bajo esta premisa, es común y recomendable que todos los prototipos y declaraciones de tipos que se realicen globales para el uso del programa o definiciones generales se hagan en un archivo cabecera, el mismo tendrá la extensión *.h* y las definiciones en un archivo *.c*.

Un ejemplo sería:

```
1  /*funciones1.h*/
2
3  #include<stdio.h>
4
5  int cubo (int);
```

Luego en el archivo de las definiciones tendríamos:

```
1  /*funciones1.c*/
2
3  #include "funciones1.h"
4
5  int cubo (int base)
6  {
7          int potencia;
8          potencia = base * base * base;
9          return potencia;
10 }
11
12 int main ()
13 {
14         int numero;
15
16         for (numero = 1; numero <= 5; numero++)
17                 printf( "El cubo de %d es igual a %d\n", numero, cubo(numero));
18
19         return 0;
20 }
```

Como vemos, en el archivo en que se definen las funciones, encontramos en la línea 3 que se incluye la cabecera y luego el desarrollo de la función, previamente declarada en el .h, y el main que es imprescindible para que se ejecute el programa.

```
$ gcc -o funciones1 funciones1.c
$ ./funciones1
El cubo de 1 es igual a 1
El cubo de 2 es igual a 8
El cubo de 3 es igual a 27
El cubo de 4 es igual a 64
El cubo de 5 es igual a 125
```

1.3. Llamadas a funciones

En el ejemplo anterior, en el archivo .c, en la línea 17 vemos que se invoca a la función cubo pasándole la variable numero. De esta forma llamamos a las funciones.

Se debe tener en cuenta algunas cuestiones importantes, como por ejemplo, debo pasar la misma cantidad y en el mismo orden los parámetros de entradas que fueron declarados con anterioridad. Otro detalle importante es el retorno de la función.

1.3.1. Retorno de las funciones. Una función en C sólo puede devolver un valor, para esto se utiliza la palabra **return** y dicho retorno debe coincidir con el tipo de valor declarado en el prototipo.

```
1  int a, calculo;
2  a = cubo(2);
3  calculo = cubo(2)/8;
4
5  cubo(3) = a + 5; /* ESTO ESTA MAL!! */
6
7  imprime_valores(a, b, c);
```

En los ejemplos vemos variadas situaciones que nos permiten entender algunos de los usos de las funciones:

1. Cuando una función retorna un valor podemos usarla para asignarlo a una variable del mismo tipo. Pero nunca en el sentido inverso, o sea, la función no es una variable a la que se le puede asignar un valor.
2. Cuando una función retorna un valor podemos usarla como parte de una expresión mas compleja, como por ejemplo el cálculo de una fórmula, en este caso la función se terminará reemplazando por el valor que ella retorne.
3. Si la función retorna un valor y no se la asigna a una variable el retorno se pierde.
4. Las funciones que no tienen retorno de valor no se asignan, solo se las invoca.

Un ejemplo completo:

```
1  /*funciones2.h*/
2
3  #include<stdio.h>
4
5  void primera();
6  int segunda(int, int);
```

```
1  #include "funciones2.h"
2
3  int segunda (int x, int j)
4  {
5      int b;
6      b = x + j;
7      printf("se inicia la segunda funcion - retorno %d y finalizo\n", b);
8
9      return b;
10
11 }
12
13
14 void primera()
```

```
15  {
16      printf("Esta es la primera ófuncin\n");
17
18      printf("Muestro el resultado de la segunda funcion: %d\n", segunda(3, 4));
19
20      printf("Fin de la primera funcion\n");
21  }
22
23  int main()
24  {
25      printf("inicia el main\n");
26      int a;
27      primera();
28
29      a = segunda(5, 8);
30
31      printf("El valor de a es %d\n", a);
32
33      printf("Fin del main\n");
34
35      return 0;
36  }
```

```
$ gcc -o funciones2 funciones2.c
$ ./funciones2
inicia el main
Esta es la primera ófuncin
se inicia la segunda funcion - retorno 7 y finalizo
Muestro el resultado de la segunda funcion: 7
Fin de la primera funcion
se inicia la segunda funcion - retorno 13 y finalizo
El valor de a es 13
Fin del main
```

2. ÁMBITO Y TIPO O CLASES DE ALMACENAMIENTO DE VARIABLES

Hasta el momento, hemos tenido en cuenta a las variables por el tipo de dato que son: enteras (*int*, carater (*char*), etc. 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 (*auto*), externo (*extern*), estático (*static*)y registro (*register*).

2.1. Variables locales

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.

A cualquiera de estas variables declaradas en las funciones se las considera como una variable automática (*auto*) excepto que se use algún modificador para cambiar esto. Esto es así porque cuando se accede a la función se le asigna espacio en la memoria automáticamente y se libera dicho espacio una vez finalizada la función. Esto último nos lleva a dos cuestiones a tener en cuenta:

1. La variable pierde el valor en sucesivas llamadas a la función (aunque sean inmediatas).

2. Después de declaradas, las variables deben ser inicializadas para evitar problemas por un valor indeterminado con el que puedan estar.

Ejemplo de variables automáticas en diferentes ambitos:

```
/*scope1.h*/

#include<stdio.h>

void imprime_valor();
```

```
/*scope1.c*/

#include "scope1.h"

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

int main()
{
    int contador;
    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;
}
```

```
$./scope1
El valor de contador es: 1
El valor de contador es: 5
Ahora el valor de contador es: 1
$
```

2.2. Variables globales

A diferencia de la las locales, el ámbito de las variables globales se extiende desde el punto en que se definen hasta el final del programa, permitiendo así que cualquier programa haga uso de ella simplemente usando su nombre.

Es un mecanismo simple de intercambio de información entre distintos ámbitos del programa pero hay que también saber que en programas complejos su uso hace que el codigo se vuelva confuso y dificil de depurar.

Ejemplo:

```
/*scope2.h*/
#include <stdio.h>

```

```
4  void unaFuncion();
5  void otraFuncion();
6  int variable;
```

```
1  /*scope2.c*/
2  #include"scope2.h"
3
4  void unaFuncion()
5  {
6      printf("En la ófuncin unaFuncion, variable es: %d\n", variable);
7  }
8  void otraFuncion()
9  {
10     variable++;
11     printf("En la ófuncin otraFuncion, la variable es: %d\n", variable);
12 }
13 int main()
14 {
15     variable = 9;
16     printf("El valor de variable es: %d\n", variable);
17     unaFuncion();
18     otraFuncion();
19     printf("Ahora el valor de variable es: %d\n", variable);
20     return 0;
21 }
```

```
$ ./scope2
El valor de variable es: 9
En la ófuncin unaFuncion, variable es: 9
En la ófuncin otraFuncion, la variable es: 10
Ahora el valor de variable es: 10
$
```

Hay que tener en cuenta un par de cuestiones, en general, las declaraciones y definiciones de las variables globales se realizan al principio del programa, fuera de toda función, se realizan igual que cualquier variable, pero al cambiar el ámbito estando ·afuera"de las funciones ya las hace globales. Podemos en este ámbito asignarle un valor.

Como se dijo, cualquier función accederá a una variable global previamente declarada simplemente invocando su nombre. En el caso que la función se haya definido antes que la variable global se requiere incluir una definición de dicha variable en la función para poder usarla anteponiendo la palabra reservada *extern*, además, no se podrá asignar valor a la variable externa en su declaración y el tipo y nombre deberá tener que coincidir con la variable global.

Error 1 - Ejemplo en las declaraciones y definiciones.

```
1  void unaFuncion()
2  {
3      variable; /* considerara a la variable no declarada previamente */
4      printf("En la ófuncin unaFuncion, variable es: %d\n", variable);
5  }
6  void otraFuncion()
7  {
8      variable++; /* considerara a la variable no declarada previamente */
```

```
9       printf("En la ófuncin otraFuncion, la variable es: %d\n", variable);
10  }
11  int main()
12  {
13      variable = 9; /* considerara a la variable no declarada previamente */
14      printf("El valor de variable es: %d\n", variable);
15      unaFuncion();
16      otraFuncion();
17      printf("Ahora el valor de variable es: %d\n", variable);
18      return 0;
19  }
20
21  int variable;
```

En el código se indica la línea con error, el compilador nos dira lo siguiente:

```
$ gcc -o scope3 scope3.c
scope3.c: In function ‘’unaFuncion:
scope3.c:3:60: error: ‘’variable undeclared (first use in this function)
     printf("En la ófuncin unaFuncion, variable es: %d\n", variable);
                                                            ^
scope3.c:8:60: note: each undeclared identifier is reported only once for each function it appears in
scope3.c: In function ‘’otraFuncion:
scope3.c:13:64: error: ‘’variable undeclared (first use in this function)
     printf("En la ófuncin otraFuncion, la variable es: %d\n", variable);
                                                                ^
scope3.c: In function ‘’main:
scope3.c:17:45: error: ‘’variable undeclared (first use in this function)
     printf("El valor de variable es: %d\n", variable);
                                             ^
$
```

Error 2 - Ejemplo en las asignaciones.

```
1   void unaFuncion()
2   {
3       extern variable;
4       printf("En la ófuncin unaFuncion, variable es: %d\n", variable);
5   }
6   void otraFuncion()
7   {
8       extern variable++; /* el compilador dara un error por asignar valor en la
            ódeclaracin */
9       printf("En la ófuncin otraFuncion, la variable es: %d\n", variable);
10  }
11  int main()
12  {
13      extern variable = 9;/* el compilador dara un error por asignar valor en la
            ódeclaracin */
14      printf("El valor de variable es: %d\n", variable);
15      unaFuncion();
16      otraFuncion();
17      printf("Ahora el valor de variable es: %d\n", variable);
18      return 0;
19  }
20
21  int variable;
```

Están indicado en el código las líneas con error. Al compilar se tendrá el siguiente mensaje:

```
$ gcc -o scope3 scope3.c
scope3.c: In function ‘’otraFuncion:
scope3.c:8:20: error: expected ‘’=, ‘’,, ‘’;, ‘’asm or ‘’__attribute__ before ‘’++ token
     extern variable++;
                    ^
scope3.c:8:22: error: expected expression before ‘’; token
     extern variable++;
                      ^
scope3.c: In function ‘’main:
scope3.c:13:12: error: ‘’variable has both ‘’extern and initializer
     extern variable = 9;
            ^
$
```

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*.

Este es el ejemplo correcto:

```c
/*scope3.c*/
#include<stdio.h>

void unaFuncion()
{
    extern int variable;
    printf("En la ófuncin unaFuncion, variable es: %d\n", variable);
}
void otraFuncion()
{
    extern int variable;
    variable++;
    printf("En la ófuncin otraFuncion, la variable es: %d\n", variable);
}
int main()
{
    extern int variable;
    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;
}

int variable;
```

```
$gcc -o scope3 scope3.c
$./scope3
El valor de variable es: 9
```

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

2.3. Variables estáticas

Se identifican anteponiendo la palabra reservada *static*. Como cualquier variable podrían ser globales o locales, en lo que cambia es que el valor que asuma perdurará hasta el final del programa, o sea, su contenido no se borra al finalizar la función,

Ejemplo:

```c
/*scope4.c*/
#include <stdio.h>

void sumador();

void sumador()
{
    static int contador = 0;
    printf("El valor de contador es: %d\n", contador);
    contador++;
}

int main()
{
    sumador();
    sumador();
    sumador();
    sumador();
    return 0;
}
```

```
$ gcc -o scope4 scope4.c
$ ./scope4
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 vemos, la primera vez que se llama a la función, la variable se declara como static y se define con un valor 0, se muestra y se incrementa. La función finaliza y es llamada nuevamente, como la variable es estática perduró en la memoria por lo tanto no se redeclara, y se muestra con el valor incrementado, al finalizar se incrementa nuevamente. Así en las sucesivas llamadas.

Las variables que globales definidas como estáticas podrán ser usadas por cualquier función del archivo pero no podrán ser accedidas desde otros archivos. O sea, en otros archivos podrían redefinirse sin generar conflictos.

Esta última característica tambien se aplica a las funciones, la función en sí misma es externa y puede ser llamada por cualquier otra función del programa aunque esté en otro archivo mientras se incluya el primero. Para evitar esto, y restringir el uso, solo bastaría con declarar a la función como estática. Ejemplo:

```c
static int cuadrado (int numero)
{
        int calculo;
```

```
        calculo = 0;
        calculo = numero * numero;
        return calculo;
}
```

2.4. Variables de registro

Se utiliza para variables que se pretende tener acceso rápido. Al declararla como *register* generalmente se almacenan en la unidad central de procesamiento.

```
register int contador;
register char letra;

void funcion (register int numero1, register int numero2)
{
        /*codigo ...*/
}
```

El uso de variables register es bastante limitado y el compilador asumirá la responsabilidad de ignorar al calificador en caso que no haya registros libres. También hay un límite con respecto a la cantidad posible de variables a declarar, en todo caso, ocurre lo mismo que antes, el compilador ignorará el excedente.

3. PASAJE DE ARGUMENTOS Y PUNTEROS

El lenguaje C fue creado por Dennis Ritchie en 1972. Trabajaba para los Laboratorios Bell. Desde su definición se afirma que todos los argumentos que se pasan a una función se pasan por su valor. En resumen, se pasa la copia del valor del argumento y no la variable en sí misma. O sea, cualquier modificación que haga esta función la esta haciendo al valor en si, y no a la variable original que se mantendrá inalterada.

Resumiendo, la variable declarada como argumento en la función es local a esa función, Cualquier cambio al valor de ella cuando finalice la función se pierde y la variable original no tuvo modificiación alguna.

Ejemplo de pasaje de argumento:

```
/*referencia1.h*/
#include <stdio.h>

void modificar(int);
```

```
/*referencia1.c*/
#include "referencia1.h"

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

main()
{
    int i = 1;
    printf("\ni= %d antes de llamar a la ófuncin modificar", i);
    modificar(i);
```

```
16      printf("\ni=%d édespus de llamar a la ófuncin modificar\n", i);
17  }
```

```
$ gcc -o referencia1 referencia1.c
$./referencia1

i=1 antes de llamar a la ófuncin modificar
variable = 1 dentro de modificar
variable = 9 dentro de modificar
i=1 édespus de llamar a la ófuncin modificar
```

En el ejemplo corroboramos que lo dicho antes es verdad, entonces ¿Cómo logramos pasar argumentos y que se modifiquen sus valores? o mas simple ¿Cómo pasamos argumentos por referencia?

Usando la dirección de la memoria del a variable que se pasa para acceder al espacio donde está guardado el dato, el valor de la variable.

Veamos el ejemplo anterior modificado para usar las referencias:

```
1   /*referencia2.h*/
2   #include <stdio.h>
3
4   void modificar(int *);
```

```
1   /*referencia2.c*/
2   #include "referencia2.h"
3
4   void modificar(int * variable)
5   {
6       printf("\nvariable = %d dentro de modificar", *variable);
7       *variable = 9;
8       printf("\nvariable = %d dentro de modificar", *variable);
9   }
10
11  main()
12  {
13      int i = 1;
14      printf("\ni=%d antes de llamar a la ófuncin modificar", i);
15      modificar(&i);
16      printf("\ni=%d édespus de llamar a la ófuncin modificar\n", i);
17  }
```

```
$ gcc -o referencia2 referencia2.c
$ ./referencia2

i=1 antes de llamar a la ófuncin modificar
variable = 1 dentro de modificar
variable = 9 dentro de modificar
i=9 édespus de llamar a la ófuncin modificar
```

4. FUNCIONES RECURSIVAS

Una función recursiva es aquella que para resolverse se llama a sí misma. Un proceso recursivo tiene que tener si o sí una condición de finalización ya que sino, se llamaría a sí misma en forma infinita.

El ejemplo típico de una función recursiva es el cálculo del factorial de un número entero (*n!*)

$$n! = n * (n-1) * (n-2) * (n-3) * ... * 2 * 1$$

```c
/*factorial1.h*/

#include <stdio.h>

int factorial(int);
```

```c
/*factorial1.c*/
#include"factorial1.h"

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

main()
{
    int valor = 4;
    int resultado;
    resultado = factorial(valor);
    printf("El factorial de %d es %d \n", valor, resultado);
    return 0;
}
```

```
$ gcc -o factorial1 factorial1.c
$ ./factorial1
El factorial de 4 es 24
```

Esta es tal vez la forma mas clásica de representar al factorial, y el código, de forma imperativa es el que lo resuelve. Sin embargo si vemos una formulación un poco mas formal lo expresaríamos de este modo:

$$n! = \prod_{1=k}^{n} k$$

o como una relación de recurrencia:

$$n! \begin{cases} 1 & ; n = 0 \\ (n-1)! \times n & \forall n \neq 0 \end{cases}$$

Estas dos definiciones inducen a un pensamiento recursivo, se define el factorial con la multiplicación de un entero por el factorial de ese entero menos uno lo cual tiene total lógica.

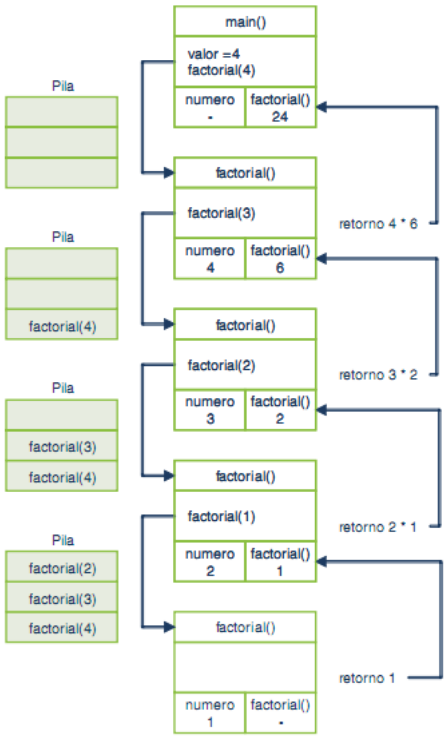
El código de la función cambiaría de la siguiente forma:

```
int factorial(int numero)
{
        if(numero == 1)
                return 1;
        else
        return (numero * factorial(numero-1));
}
```

Si analizamos la secuencia de acción veriamos que ocurre lo 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
```

Más graficamente


main()
valor =4
factorial(4)
numero
-
factorial()
24
Pila
factorial()
factorial(3)
numero
4
factorial()
6
retorno 4 * 6
Pila
factorial(4)
factorial()
factorial(2)
numero
3
factorial()
2
retorno 3 * 2
Pila
factorial(3)
factorial(4)
factorial()
factorial(1)
numero
2
factorial()
1
retorno 2 * 1
Pila
factorial(2)
factorial(3)
factorial(4)
factorial()
numero
1
factorial()
-
retorno 1