# Funciones

### Definición de Funciones

Las funciones en C tienen el siguiente formato:

```
tipo_de_retorno NOMBRE(tipo_param1 param1, tipo_param2 param2, ... )
{
    /* Cuerpo de la función */
}
```

Cuando se invoca una función se asignan valores a sus parámetros y comienza a ejecutar el cuerpo hasta que se llega al final o se encuentra la instrucción return. Si la función devuelve un resultado, esta instrucción debe ir seguida del dato a devolver. Por ejemplo:

```
1  int search(int table[], int size)
2  {
3      int i, j;
4      if (size == 0)
5      {
6          return 0;
7      }
8      j = 0;
9      for (i = 0; i < size; i++)
10     {
11         j += table[i];
12     }
13     return j;
14 }
```

La ejecución de la función comienza en la línea 4. Si el parámetro size tiene valor cero, la función termina y devuelve el valor cero. Si no, se ejecuta el bucle y se devuelve el valor de la variable j. El tipo de la expresión que acompaña a la instrucción return debe coincidir con el tipo del resultado declarado en la línea 1.

La llamada a una función se codifica con su nombre seguido de los valores de los parámetros separados por comas y rodeados por paréntesis. Si la función devuelve un resultado, la llamada se reemplaza por su resultado en la expresión en la que se incluye. Por ejemplo:

```
1  int addition(int a, int b)
2  {
3      return (a + b);
4  }
5  int main()
6  {
7      int c;
8      c = c * addition(12, 32);
9  }
```

---

## Pasaje de parámetros por Valor / Copia

Los parámetros son variables locales a los que se les asigna un valor antes de comenzar la ejecución del cuerpo de una función. Su ámbito de validez, por tanto, es el propio cuerpo de la función. El mecanismo de pasaje de parámetros a las funciones es fundamental para comprender el comportamiento de los programas en C.

Consideremos el siguiente programa:

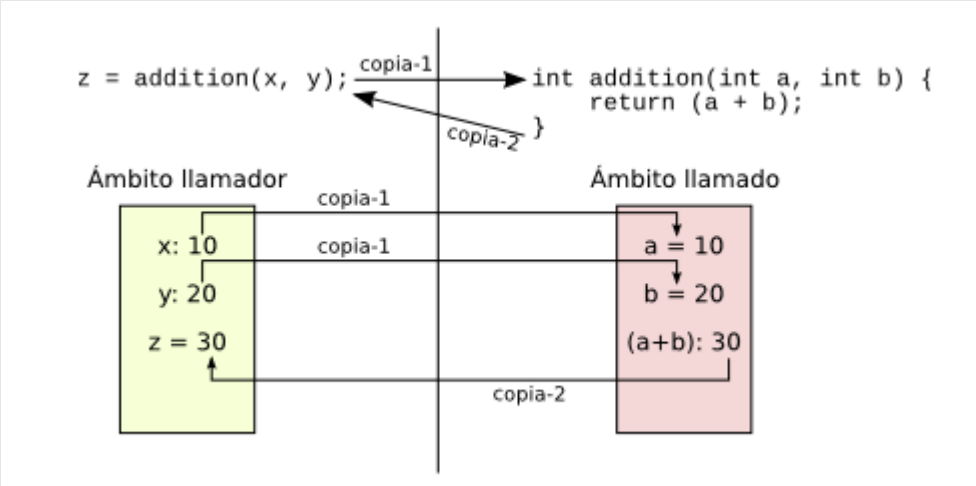
```
1  int addition(int a, int b)
2  {
3      return (a + b);
4  }
5  int main()
6  {
7      int x = 10;
8      int y = 20;
9      int z;
10
11     z = addition(x, y);
12 }
```

Los parámetros a y b declarados en la línea 1 son válidos únicamente en la expresión de la línea 3. Las variables x, y y z, por su lado, son válidas en el cuerpo de la función main (líneas 7 a 11).

El ámbito de las variables x, y y z (ámbito llamador), y el de las variables a y b (ámbito llamado) son totalmente diferentes. El ámbito llamador desaparece temporalmente cuando se invoca la función desde la liñea 11. Durante esa ejecución, el ámbito llamado es el visible. Al terminar la función, el

ámbito llamado desaparece y se recupera el ámbito llamador.

La comunicación entre estos dos ámbitos se realiza en la línea 11. Antes de comenzar la ejecución de la función, ***los valores de las variables del ámbito llamador son copiadas sobre las variables del ámbito llamado***. Cuando termina la ejecución de la función, la expresión de la llamada en la línea 11 se reemplaza por el valor devuelto. En la siguiente figura se ilustra este procedimiento para el ejemplo anterior.




> *El pasaje de parámetros y la devolución de resultado en las funciones C se realiza **por valor**, es decir, **copiando** los valores entre los dos ámbitos .*

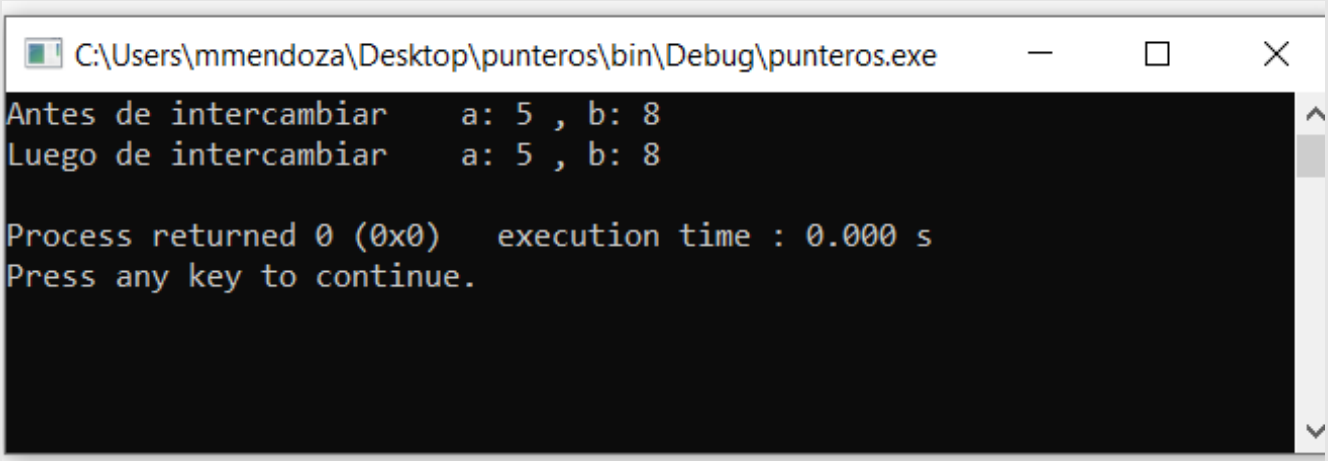
Las variables locales de la función (incluidos los parámetros) desaparecen al término de la función, por lo que cualquier valor que se quiera guardar, debe ser devuelto como resultado.

Otro ejemplo:

```
#include <stdio.h>
#include <stdlib.h>

void intercambiar (int x , int y)
{
 int xtemp;
 x=y;
 y= xtemp;
}


int main()
```

```
{
    int a = 5;
    int b= 8;
    printf("Antes de intercambiar    a: %d , b: %d \n", a,b);

    intercambiar (a,b);
    printf("Luego de intercambiar    a: %d , b: %d \n", a,b);
    return 0;
}
```

**En el caso de que dentro de la función yo llamo a las variables a y b no las conoce.**




## Pasaje de parámetros por Referencia

El lenguaje C no hace paso por referencia como normalmente creemos, solo hace copia de valores, esto implica que, si bien podemos pasar datos o direcciones siempre vamos a tener que generar una variable local dentro de la función para guardar ese dato.

Si pasamos direcciones de memoria como parámetros para la función, estamos trabajando con un único vector y no ocupa memoria innecesariamente, y estoy trabajando sobre el vector original que tenía en el main y pudiendo modificarlo desde la función sin tener que devolverlo. Esto también abre la posibilidad de que una función devuelva más información que la que puede entregar una sola variable a la hora de hacer el return.

Las estructuras deben pasarse siempre como direcciones de memoria por que las mismas ocupan demasiado espacio y al pasarla como direcciones de memoria solo ocupan 4 bytes. Pero esto no es absoluto ya que varía dependiendo de la situación, de modo que, si mi estructura está comprendida por uno o dos caracteres únicamente ocupa 2 bytes de memoria, y sería incluso conveniente pasar la estructura completa en lugar del puntero.

Otro punto para tener en cuenta es que si no quiero modificar la estructura puedo tomarla y pasarla a la función como  parámetro de modo que, al recibir la estructura, la función esté definida para que tanto la dirección ( puntero ) , el contenido de la estructura, o ambas, sean constantes.

| | |
|---|---|
| **const tipo_dato *variable** De esta forma estoy haciendo constante los valores. | **tipo_dato * const variable** De esta forma estoy haciendo constante la dirección de memoria del puntero. |

Habiendo definido todo esto, vamos a introducir el concepto de punteros.

---