PASO DE PARAMETROS A FUNCION

La sintaxis genérica de la declaración de una función en C es la siguiente:

```
tipo_funcion nombre_funcion({tipo_arg_form_i nombre_arg_form_i ...}) {
    cuerpo_funcion (que debe incluir la sentencia return(valor de tipo tipo_funcion))
}
```

donde:

- *tipo_funcion* indica el tipo que devolverá la función (puede ser void)
- *nombre_funcion* indica el nombre que va a tener la función
- *tipo_arg_form$_i$* indica el tipo del argumento (parámetro) formal i-ésimo
- *nombre_arg_form$_i$* indica el nombre del argumento formal i-ésimo
- y *cuerpo_funcion* son las sentencias de dicha función.

La sintaxis asociada a la sentencia de llamada a función es la siguiente:

```
nombre_identificador = nombre_funcion({nombre_arg_actual_i ...});
```

donde:

- *nombre_identificador* es el nombre de una variable declarada antes y de tipo *tipo_funcion*
- *nombre_funcion* es el nombre de la función declarada antes
- y *nombre_arg_actual$_i$* es el nombre de una variable, constante, parámetro, etc. declarada antes que debe ser de tipo *tipo_arg_form$_i$* y que actúa como parámetro actual.

La diferencia entre **argumentos formales** y **argumentos actuales** es que, los primeros sirven para indicar los nombres de posibles valores que utilizará la función cuando sea ejecutada, mientras los segundos aportan los valores concretos sobre los que realizar las operaciones indicadas dentro de la función. Veamos un ejemplo:

```
int maximo(int x, int y) {
/* Función que recibe dos enteros y devuelve el máximo de ambos */
int max;
  if (x <= y) max = y;
  else max = x;
  x = y = 0;
  return(max);
} /* maximo */

int a = 3, b = 5, max;
  ...
  max = maximo(a, b);
  ...
  max = maximo(9, 5);
```

Los parámetros formales son "x" e "y", mientras que "a" y "9" son parámetros actuales correspondientes al parámetro formal "x" y "b" y "5" son parámetros actuales asociados con el formal "y", para cada una de las dos llamadas que se realizan a la función "maximo".

Cuando se pasa un parámetro actual a una función la operación que tiene lugar consiste en copiar el valor del argumento (salvo el caso del paso de un array o matriz). Este tipo de operación se le conoce con el nombre de **paso de parámetros por valor** y tiene como característica específica que dentro de la función es posible cambiar el valor del parámetro formal (modificar la zona de memoria donde se encuentra ubicado), pero **NUNCA** (salvo por error) el de la zona de memoria donde se guarda el parámetro actual. En el ejemplo anterior, la función "maximo" modifica los valores de los parámetros formales "x" e "y" y les asigna el valor 0, pero no se modifica ni "a" ni "b" ni menos las constantes "9" y "5".

Las matrices o arrays se tratan de forma diferente, ya que si uno de los parámetros formales se declara de este tipo se espera que el parámetro actual asociado indique una dirección de la zona de memoria donde se almacena dicha matriz. Veamos un nuevo ejemplo:

```
void maximo1(int num[], int n) {
```

```
/* Función que recibe un vector de enteros y el número de elementos que contiene y almacena  en
el primer elemento de dicho vector el valor máximo de dicho vector */
int i;
   num[0] = -32768;
   for (i = 1; i <= n; i++)
      if (num[i] >= num[0]) num[0] = num[i];
  num = 0;
} /* maximo1 */

int NUM1[4] = {0, -3, 4, 7,};
int NUM2[6], N;
   ...
   N = 2;
   maximo(NUM1, N);
   ...
   maximo(NUM2, 3);
```

De nuevo los parámetros formales son "num" y "n", mientras que "NUM1" y "NUM2" son parámetros actuales correspondientes al parámetro formal "num" y "N" y "3" son parámetros actuales asociados con el formal "n", para cada una de las dos llamadas que se realizan a la función "maximo1".

En este caso, cuando se pasan los parámetros actuales "N" o "3" la forma de actuar es igual que la anterior, pero cuando se pasan los parámetros NUM1 o NUM2 a la salida de la función se han modificado los valores correspondientes a las variables de tipo entero NUM1[0] o NUM2[0], respectivamente. A este tipo de forma de proceder se le conoce con el nombre de **paso de parámetros por referencia (o por variable)** y debe su nombre a que realmente se está pasando la dirección de la variable y no sus valores. Al contrario que el paso de parámetros anterior, su característica principal es que permite modificar los valores de los parámetros actuales, cuando se utiliza como referencia al parámetro formal (si se observa con detenimiento la función "maximo1" se puede comprobar que existe una sentencia de asignación "num = 0" que no modifica las variables NUM1 o NUM2 , pero sí modifica, como dijimos antes, las variables NUM1[0] o NUM2[0] con la sentencia de asignación num[0] = num[i]). En la siguiente figura se representa gráficamente lo que ocurre en este paso de parámetros.

En esta primera tabla se representa el estado de la memoria cuando va a comenzar la función "maximo1"

Dirección	Nombre	Valor	Dirección	Nombre	Valor
4400	NUM1[0]	0	5860	n	2
4402	NUM1[1]	-3	5862	num	4400
4404	NUM1[2]	4			
4406	NUM1[3]	7			
4408	NUM2[0]	???			
	...				
	...				
4420	N	2			

En esta segunda tabla se presenta el estado de la memoria después de ejecutar la función "maximo1"

Dirección	Nombre	Valor	Dirección	Nombre	Valor
4400	NUM1[0]	7	5860	n	2
4402	NUM1[1]	-3	5862	num	0
4404	NUM1[2]	4			
4406	NUM1[3]	7			
4408	NUM2[0]	???			
	...				
	...				
4420	N	2			

Se puede observar que se ha modificado el valor de la variable NUM1[0] pero no NUM1 (obsérvese que la variable NUM1 no tiene un espacio físico en memoria asociado lo que provoca que no sea posible cambiar su valor, realmente poner en la llamada a la función "maximo1" como parámetro "NUM1" es una ayuda del compilador de C que hace la equivalencia NUM1 = &NUM1[0]. En este sentido, también se puede ver el paso de vectores como paso por valor, ya que cuando se introduce el parámetro actual NUM1 sé esta copiando su valor en memoria).

PASO DE PARAMETROS POR REFERENCIA Y VARIABLES PUNTEROS

La función "maximo1" anterior se puede representar también, del siguiente modo:

```
void maximo1(int *num, int n)
```

donde se ha sustituido los corchetes "[]" por el asterisco (o estrella) "*". Si bien se pueden definir de forma diferente representan lo mismo:

int num[] variable de tipo vector de enteros de nombre "num"
int *num variable de tipo puntero a elemento entero de nombre "num"

en ambos casos lo que se espera es una dirección de memoria, y se trabaja a partir de esa dirección. La desventaja (o ventaja según se mire) de la primera opción es que parece indicar que se espera un vector y no parece una buena representación cuando sólo queramos hacer referencia a un elemento, como ocurre en el ejemplo siguiente:

Supongamos que queremos construir una función que compute el máximo, mínimo y media de un vector de enteros y decidimos que la función sea de tipo void (no devuelve nada) y para mantener la mayor portabilidad posible no utilizamos ninguna variable global dentro de la función. Estos condicionantes nos obligan a utilizar los siguientes parámetros:
La dirección de comienzo del vector y el número de elementos (estos dos parámetros no es necesario modificarlos dentro de la función, por lo que se les conoce como parámetros de entrada)
Un parámetro para cada uno de los elementos a calcular: máximo, mínimo y media (estos tres parámetros deben ser actualizados dentro de la función y conocidos por la función que los llama, por lo que se les conoce con el nombre de parámetros de salida, también existen los parámetros de entrada-salida).

```
void  calcularvalores(int *v, int n, int *max, int *min, int *med) {
/* Función que recibe un vector de enteros y el número de elementos que contiene y almacena en
las variables *max, *min y *med los valores máximo, mínimo y media entera respectivamente */
int i;
  *max = -32768; *min = 32767; *med = 0;
  /* Valores mínimo y máximo de un entero almacenado en 2 bytes */
   (1)
   for (i = 0; i < n; i++) {
      if (num[i] > *max) *max = num[i];
      if (num[i] < *min) *min = num[i];
      *med  += num[i];
   } /* for ... */
   *med /= n;
   (2)
} /* calcularvalores */

int NUM1[4] = {0, -3, 4, 7,};
int N, M, m, d;
   ...
   N = 2;
   calcularvalores(NUM1, N, &M, &m, &d);
```

En la función y en la llamada destacan el acceso a diferentes variables mediante el operador contenido "*" y el acceso a la dirección de unas variables mediante el operador "&". Los dos primeros parámetros hemos dicho que son de entrada y no son modificados dentro de la función (recordar que si bien la modificación del segundo parámetro "n" no produce modificaciones en el exterior, no ocurre lo mismo con el segundo ya que se accede a el de forma indirecta "num[i]" y disponemos de la dirección donde está almacenado el vector). Los tres últimos parámetros (max, min y med) declaran elementos de tipo puntero a entero, es decir deben contener la dirección de un entero y cuando se accede a ellos aplicando el operador "*" estamos indicando que deseamos acceder al valor contenido en la dirección que almacenan. Veamos la representación en memoria de la función, y su llamada en el punto (1) y (2) de la ejecución de la función:

Dirección	Nombre	Tipo	Valor	Dirección	Nombre	Tipo	Valor

5860	Med	Puntero	4414	4400	NUM1[0]	Entero	0
5862	Min	Puntero	4412	4402	NUM1[1]	Entero	-3
5864	Max	Puntero	4410	4404	NUM1[2]	Entero	4
5866	N	Entero	2	4406	NUM1[3]	Entero	7
5868	V	Puntero	4400	4408	N	Entero	2
5870	Zona de			4410	M	Entero	-32768
5872	Contexto			4412	M	Entero	32767
5874	I	Entero	???	4414	D	Entero	0

Estado de la memoria en el punto (1)

Dirección	Nombre	Tipo	Valor	Dirección	Nombre	Tipo	Valor
5860	Med	Puntero	4414	4400	NUM1[0]	Entero	0
5862	Min	Puntero	4412	4402	NUM1[1]	Entero	-3
5864	Max	Puntero	4410	4404	NUM1[2]	Entero	4
5866	N	Entero	2	4406	NUM1[3]	Entero	7
5868	V	Puntero	4400	4408	N	Entero	2
5870	Zona de			4410	M	Entero	7
5872	Contexto			4412	M	Entero	-3
5874	I	Entero	2	4414	D	Entero	2

Estado de la memoria en el punto (2)

Se puede observar en la memoria que los parámetros "max", "min" y "med" contienen las direcciones de las variables "M", "m" y "d" respectivamente ya que en la llamada se pasa la dirección de las mismas mediante el operador "&". También se puede comprobar que cuando se utiliza el operador "*" sobre alguno de los parámetros se está accediendo a la variable correspondiente (el operador "*" se puede traducir como: "ir a la dirección de memoria que indica el nombre y tomar/modificar su valor"). En realidad en la función no existe ninguna diferencia si se sustituyen las sentencias *xxx por xxx[0] y la declaración "tipo *nombre" por "tipo nombre[]" de ahí que el código de la función anterior también se puede escribir de la siguiente manera (aunque la primera forma es más aconsejable para evitar llenar el código con accesos a elementos de matrices):

```
void  calcularvalores(int v[], int n, int max[], int min[], int med[]) {
/* Función que recibe un vector de enteros y el número de elementos que contiene y almacena en
las variables max[0], min[0] y med[0] los valores máximo, mínimo y media entera respectivamente
*/
int i;
  max[0] = -32768; min[0] = 32767; med[0] = 0;
  /* Valores mínimo y máximo de un entero almacenado en 2 bytes */

   for (i = 0; i < n; i++) {
      if (num[i] > max[0]) max[0] = num[i];
      if (num[i] < min[0]) min[0] = num[i];
      med[0]  += num[i];
   } /* for ... */
   med[0]  /= n;
} /* calcularvalores */
```

Si se observa de nuevo la parte de código referente a la llamada a la función se puede comprobar el uso del operador dirección "&". Este operador se aplica a las variables declaradas **explícitamente** "M", "m" y "d" e **implícitamente** el vector "NUM1". Ya hemos visto que todas se recogen en los parámetros formales en variables puntero y presentan en este caso una de las propiedades más importante a tener en cuenta en el uso de los punteros: *Antes de utilizar un puntero con el operador contenido "*" hay que asegurarse de que esté bien inicializado, es decir, contenga una dirección de memoria que pertenezca al programa que se está ejecutando*. Veamos un ejemplo de mala utilización: Supongamos que declaramos las variables "M", "m" y "d" de tipo puntero a entero. En este caso, la declaración de las variables y la llamada a la función se debe realizar del siguiente modo:

```
int *M, *m, *d;
  ...
  calcularvalores(NUM1, N, M, m, d);
```

Se ha eliminado la utilización del operador "&" porque las variables ya son de tipo puntero a entero, y su inclusión indicaría que estamos pasando como parámetros actuales, la dirección de unas variables que contienen la dirección de un entero (lo que parece un trabalenguas y debería ser recogido en los parámetros formales como de tipo int **). Sin embargo, con este código estamos violando la propiedad especificada anteriormente como podemos ver en la representación de la memoria siguiente:

Dirección	Nombre	Tipo	Valor	Dirección	Nombre	Tipo	Valor
5860	Med	Puntero	**???3**	4400	NUM1[0]	Entero	0
5862	Min	Puntero	**???2**	4402	NUM1[1]	Entero	-3
5864	Max	Puntero	**???1**	4404	NUM1[2]	Entero	4
5866	N	Entero	2	4406	NUM1[3]	Entero	7
5868	V	Puntero	4400	4408	N	Entero	2
5870	Zona de			4410	M	Puntero	**???1**
5872	Contexto			4412	m	Puntero	**???2**
5874	I	Entero	???	4414	d	Puntero	**???3**

Cuando accedemos a *Max ó *Min ó *Med estamos leyendo o escribiendo de una celda de memoria que no conocemos, hecho que se debe como acabamos de comentar a que no han sido inicializadas las variables puntero. Mostramos a continuación otro ejemplo del mismo tipo, pero ahora con cadenas de caracteres.

```
char *str;

  ....
  scanf("%s", str);
```

En este caso, hemos declarado una variable de tipo puntero a carácter y hacemos una llamada a la función "scanf" para la lectura de una cadena. De nuevo, se produce el problema anterior debido a que el valor de la variable "str" está sin inicializar y la cadena se almacena en esa posición y las siguientes de memoria. Este hecho se puede arreglar declarando la variable "str" como una cadena de caracteres, que implica la reserva de MAX elementos y el conocimiento de la dirección inicial.

```
char str[MAX];
```

Sin embargo, si esta declaración se realiza dentro de una función (la cadena "str" es local a dicha función), no se puede utilizar dicha cadena cuando finaliza la función (recordar que las variables locales sólo existen dentro de la función donde se han declarado y por lo tanto no puede ser utilizada como dirección de cadena a devolver por la función). Una posible solución es pasar la cadena como parámetro (con lo que se entiende que ha sido reservada en otra función anterior) y otra más elegante es asignar en **tiempo de ejecución** un espacio determinado a la cadena, que siga estando reservado cuando se salga de la función (similar al ámbito de las variables globales mientras no se indique que el espacio asignado quiere ser liberado). Esta segunda opción se puede realizar en C mediante una serie de funciones de las librerías <alloc.h> ó <stdlib.h> como las que se muestran en el ejemplo siguiente:

```
char *str;

  ....
  str = (char *) malloc(MAX * sizeof(char));
  if (str ==  NULL)  ERROR
  scanf("%s", str);
```

Se declara de nuevo una variable puntero "str" que contendrá en la llamada a la función "malloc" la dirección de comienzo de la zona que se ha solicitado, cuyo tamaño en bytes es MAX*sizeof(char) (el parámetro formal de la función espera el número de bytes que necesita el número de elementos a reservar por el tamaño de cada elemento). Este tipo de reserva de memoria en tiempo de ejecución se le conoce con el nombre de reserva de memoria dinámica, ya que se realiza de forma dinámica en la ejecución del programa y puede variar de ejecución en ejecución porque el parámetro que se pasa a la función es variable. En este sentido la declaración de arrays locales y globales se le conoce con el nombre de reserva de memoria estática, ya que en tiempo de compilación se debe expresar el número de elementos a reservar. Diferencias básicas entre la reserva dinámica y la estática son:

1) El espacio reservado por la dinámica se mantiene reservado al salir de la función, lo que no ocurre en la estática a no ser que la variable sea global (este hecho permite que una función devuelva la dirección de comienzo de esta zona que no se perderá al salir de la función).
2) El número de elementos a reservar con la dinámica puede variar en cada ejecución adaptándose a lo que se necesita en cada momento y no como en la estática en la que se debe reservar siempre un tamaño máximo.
3) La reserva dinámica se puede liberar en cualquier momento con la llamada a la función "free" que recibe como parámetro la dirección de comienzo de la zona reservada y no como la estática que tiene prefijado su ámbito antes de la ejecución.