Fundamentos de Programación – Programación Estructurada en C:
10.- ASIGNACIÓN DINÁMICA DE MEMORIA

Maider Huarte Arrayago

Fundamentos de Programación – Programación Estructurada en C:

10.- ASIGNACIÓN DINÁMICA DE MEMORIA

Copyright © 2008 Maider Huarte Arrayago

Fundamentos de Programación – Programación Estructurada en C: 10.- ASIGNACIÓN DINÁMICA DE MEMORIA by Maider Huarte Arrayago is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ or, send a letter to Creative Commons, 171 2nd Street, Suite 300, San Francisco, California, 94105, USA.

Fundamentos de Programación – Programación Estructurada en C: 10.- ASIGNACIÓN DINÁMICA DE MEMORIA por Maider Huarte Arrayago está licenciado bajo una licencia Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License. Para ver una copia de esta licencia, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ o, envíe una carta a Creative Commons, 171 2nd Street, Suite 300, Sean Francisco, California, 94105, USA.

10.- ASIGNACIÓN DINÁMICA DE MEMORIA

10.1.- INTRODUCCIÓN

En C, antes de usar una variable hay que reservarle un lugar en la memoria. Con los tipos de datos simples, es suficiente con declarar la variable especificando el tipo de dato. Con los tipos de datos que crea el programador (estructuras, uniones y enumeraciones), hay que definir el nuevo tipo de dato, antes de declarar las variables.

Es decir, **la declaración es el momento en el que se hace la reserva de memoria**, por lo que el procesador ha de saber cuánta memoria ha de reservar en cada declaración. Los **tipos de datos simples** tienen un tamaño fijo, conocido por el procesador a la hora de compilar, por lo que **no hay que hacer nada antes de la declaración**. Como **los tipos de datos que crea el programador** pueden ser de cualquier forma que se le ocurra a cada programador, habrá que indicar cómo es la forma de esas variables de tipo derivado que se van a declarar. Eso, se hace mediante la definición. Por eso **es necesario definir los tipos derivados antes de declarar variables de esos tipos**. A esta forma de reserva de memoria se le llama *estática*, ya que la cantidad de memoria reservada en la declaración no puede cambiarse nunca después.

Sin embargo, a veces es imposible saber cuánta memoria hay que reservar; por ejemplo, si vamos a trabajar con arrays cuya longitud la va a especificar el usuario, no podemos programar una capacidad fija para esos arrays. Hasta ahora lo hemos hecho así, porque era la única forma aprendida, pero no es lo óptimo. La mejor forma de hacerlo, es usando la asignación *dinámica* de memoria. El objetivo de la asignación dinámica de memoria, es reservar memoria justo en el momento que haga falta, cambiar la cantidad reservada cuando haga falta, y gestionar esa memoria como si se hubiera reservado de forma estática, es decir, en una declaración.

Así, se va reservando memoria a lo largo del programa, según se va necesitando. Para ello, hay un operador y una serie de funciones de librería, que veremos en los apartados siguientes. Esas funciones de librería, se encuentran tanto en la librería *stdlib* como en la *malloc*, por lo que habrá que incluir el fichero de cabecera alguna de esas librerías en los programas en los que se vaya a hacer asignación dinámica de memoria.

10.2.- OPERADOR *sizeof*

El operador *sizeof* tiene la siguiente sintaxis:

sizeof <expresión>

Siendo *expresión* un identificador (nombre) de alguna variable declarada anteriormente, o un tipo de dato entre paréntesis.

El operador *sizeof* devuelve el tamaño en bytes de la variable o tipo de dato especificado en la *expresión*. Así, este operador se usa para calcular cuánto espacio de memoria se quiere reservar, por ejemplo.

Nótese, que aunque su uso con tipos de datos sea parecido al de una llamada de una función, NO es una función, sino un OPERADOR, tal como lo son los operadores aritméticos (+, -, *, /, %) o los relacionales (&&, ||, !).

Ejemplos:

```
int i,arr[10];

printf("\nTamaño de un int: %d",sizeof(int));
printf("\nTamaño de i: %d",sizeof i);
printf("\nTamaño de arr: %d",sizeof arr);
printf("\nTamaño del elemento 3 de arr: %d",sizeof arr[3]);
```

10.3.- FUNCIÓN *malloc*

Devuelve un puntero a la 1ª posición de un bloque de memoria libre del tamaño que se le haya especificado en el parámetro que se le ha pasado en la llamada. Es decir, busca en la memoria un espacio libre del tamaño especificado, y devuelve un puntero al inicio de ese espacio.

La llamada a la función *malloc*, se hace de la siguiente forma:

(<*tipo_de_dato*> ***)malloc(**<tamaño>**);**

La expresión **(**<tipo_de_dato> ***)** hace un casting al valor devuelto por la función, convirtiéndolo en un puntero al tipo de dato especificado. El parámetro *tamaño* especifica el tamaño en bytes del espacio de memoria libre a buscar. Ha de ser un dato de tipo *unsigned int* (entero, positivo).

En C, a una variable puntero, sólo se le pueden asignar valores correspondientes a direcciones de memoria controladas por el procesador para la ejecución del programa. Es decir, cada vez que se ejecuta un programa, se ejecuta sobre un espacio de memoria reservado para esa ejecución, de forma que el mismo programa ejecutado en máquinas diferentes, o en la misma máquina pero en momentos diferentes, puede verse ejecutado sobre espacios de memoria del mismo tamaño, pero localizados en direcciones diferentes. Por eso, no se puede asignar valor a un puntero mediante una constante:

```
int *i;

i=1000; //ERROR!
/*Aunque no provoque error en compilación, esa asignación directa no se debe hacer, pq
puede provocar error y paro del programa en ejecución*/
```

La instrucción anterior no se debe hacer porque no se puede forzar a que la posición de memoria 1000 (o ninguna otra) esté siempre en el espacio de memoria de la ejecución del programa. El espacio del programa se reserva según la disponibilidad de la memoria en el momento de empezar la ejecución, que, a priori, no se puede saber.

Así, para asignar una dirección a una variable puntero, al procesador hay que indicarle una dirección que esté en el espacio de memoria del programa. Por ello, la asignación de una dirección de memoria a una variable puntero, se puede hacer de dos formas:

```
int *i,y;

i=&y;

/*La variable y ya tiene reservado su espacio propio dentro del espacio de memoria del
programa. Su dirección es "fiable" para C*/

i=(int *)malloc(sizeof(int));

/*
La función malloc devuelve un puntero, que por casting, se especifica que sea a entero
(int), a un espacio de memoria de tamaño sizeof(int), que ha encontrado libre del espacio
de memoria del programa, y cuya reserva ha hecho. El puntero devuelto es una dirección
"fiable" para C
*/
```

10.3.1.- Reserva dinámica de memoria para arrays n-dimensionales

En el siguiente ejemplo se ve cómo se hace la reserva dinámica de memoria para arrays unidimensionales, el caso más sencillo.
Ejemplo:

```
/*
Reserva de memoria para un array unidimensional de una cantidad de enteros
especificada por el usuario:
*/

int *i,num;

printf("\nIntroduzca el número de elementos que va a tener el array: ");
scanf("%d",&num);

i=(int *)malloc(num*sizeof(int));
```

Sin embargo, cuando el array dinámico tiene más de una dimensión, la reserva dinámica de memoria se complica. Se puede hacer de varias formas. A continuación veremos dos de ellas, y sus diferencias, con ejemplos aplicados a matrices:

- **Sin emulación**

Esta es la forma más sencilla de reservar memoria dinámicamente para arrays n-dimensionales, sin emular lo que realmente pasa cuando se hace una reserva estática.

Si tenemos por ejemplo una matriz, que el usuario ha decidido en ejecución que va a tener 2 filas y 3 columnas, se hace el cálculo 2*3=6 y se reserva memoria suficiente para 6 elementos seguidos. La situación en memoria sería:

Variable	Dirección	Memoria	Tamaño
matriz	1000	2000	Dirección
≡matriz[0][0]	2000	-	Elemento
≡matriz[0][1]	2000+Elemento	-	Elemento
≡matriz[0][2]	2000+Elemento*2	-	Elemento
≡matriz[1][0]	2000+Elemento*3	-	Elemento
≡matriz[1][1]	2000+Elemento*4	-	Elemento
≡matriz[1][2]	2000+Elemento*5	-	Elemento

Ejemplo:

```
/*
Reserva de memoria para un array bidimensional de enteros, de dimensiones especificadas por el usuario:
*/

int *i,filas,cols;

printf("\nIntroduzca el nº de filas del array: ");
scanf("%d",&filas);
printf("\nIntroduzca el nº de columnas del array: ");
scanf("%d",&cols);

i=(int *)malloc(filas*cols*sizeof(int));
```

NOTA: Lo único que hace *malloc* es reservar una zona de memoria, de un cierto tamaño, y devolver un puntero a esa zona. El cómo se gestione

después esa zona de memoria, depende del programador; es decir, se podría reservar una zona del tamaño de 10 elementos enteros, y gestionarlo como un array bidimensional de 5*2 enteros, por ejemplo.

Ejemplo:

```
void accion(int [][2]);
int main()
{
   int *i;

   i=(int *)malloc(10*sizeof(int));

   accion(i);

   return 0;
}
```

Para poder acceder después a cada uno de los elementos de la matriz, habrá que hacerlo mediante aritmética de punteros, teniendo en cuenta la fórmula:

matriz[i][j] ≡ *(puntero+(tamaño_dim_2*i)+j)

Es decir, el acceso al elemento [1][2] de la matriz, habrá que hacerlo mediante la expresión **(i+(3*1)+2)*, siendo **totalmente incorrecto** el acceso mediante *i[1][2]*. En el ejemplo anterior, dentro de la función *accion*, si se usaran los corchetes para intentar acceder a los elementos de la matriz *i*, no funcionaría bien; habría q hacerlo mediante aritmética de punteros.

- **Con emulación**

De esta segunda forma, se emula totalmente lo que ocurre en memoria cuando se hace una reserva estática.

Realmente, cuando se hace una reserva estática de un array n-dimensional, aparte de la memoria necesaria para cada elemento, se reserva también un array adicional de punteros por cada dimensión del array, excepto por la última.

Así, el caso de un array unidimensional es el más sencillo, porque no requiere la reserva de ningún array de punteros adicional. En el caso de las matrices, se crea primero un array de punteros en el que se guardan los punteros a cada fila, y después se reserva memoria para cada una de las filas.

De esta forma, en el caso de una matriz de 2*3, no es necesario tener sitio suficiente para guardar 6 elementos seguidos en memoria; se reserva primero sitio para 2 punteros seguidos, uno para cada fila, y se reservan

después cada una de las filas, con todos sus elementos seguidos, pero entre ellas no tienen porqué estarlo. La situación en memoria sería:

Variable	Dirección	Memoria	Tamaño
matriz	1000	2000	Dirección
matriz[0]	2000	3000	Dirección
matriz[1]	2000+Dirección	4000	Dirección
matriz[0][0]	3000	-	Elemento
matriz[0][1]	3000+Elemento	-	Elemento
matriz[0][2]	3000+Elemento*2	-	Elemento
matriz[1][0]	4000	-	Elemento
matriz[1][1]	4000+Elemento	-	Elemento
matriz[1][2]	4000+Elemento*2	-	Elemento

En el siguiente ejemplo se muestra cómo se hace este tipo de reserva.
Ejemplo:

```
/*
Reserva de memoria para un array bidimensional de enteros, de dimensiones especificadas por el usuario:
*/

int **matriz;
int filas,cols;
int i ;

printf("\nIntroduzca el nº de filas del array: ");
scanf("%d",&filas);
printf("\nIntroduzca el nº de columnas del array: ");
scanf("%d",&cols);


matriz=(int **)malloc(filas* sizeof(int *));
for(i=0;i<filas;i++)
  matriz[i]=(int *)malloc(cols*sizeof(int));
```

Otra ventaja de hacer la reserva de esta forma, es que el acceso a cada elemento se puede hacer con los índices entre corchetes, como si de un array estático se tratara, olvidándose de la aritmética de punteros. En el caso del ejemplo, para acceder al elemento (*1,2*), sería suficiente indicar *matriz[1][2]* (aunque en la declaración de *matriz* no se hayan usado corchetes, al haber puesto un * por cada dimensión, C admite que se acceda a los elementos con tantos corchetes como *).

10.4.- calloc(unsigned int, unsigned int)

La función *calloc* es parecida a *malloc*. La sintaxis de una llamada es:

(<tipo_de_dato> ***)calloc(**<num>**,**<tam>**);**

Lo que hace es devolver un puntero, del tipo que se le especifique en el casting *(<tipo_de_dato> *)*, a una región de memoria que tiene un tamaño *num***tam*. Así, el primer parámetro *num* nos indicaría la cantidad de elementos para los que se quiere hacer la reserva de memoria, y el segundo parámetro *tam*, el tamaño de cada uno de esos elemento.

Así, la siguiente equivalencia es cierta:

```
i=(int *)malloc(sizeof(int)*num); ⇔ i=(int *)calloc(num, sizeof(int));
```

Ejemplo:

```
float *f1,*f2;

f1=(float *)calloc(10,sizeof(float));

f2=(float *)malloc(sizeof(float)*10);
```

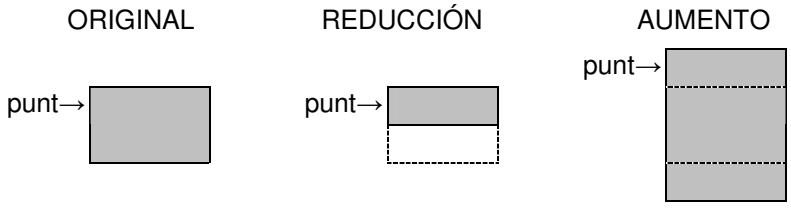
10.5.- realloc(ptr, tam)

La sintaxis de una llamada es:

(<tipo_de_dato> ***)realloc(**<punt>**,**<tam>**);**

Lo que hace es cambiar el tamaño del espacio de memoria reservado a partir de la dirección apuntada por el primer parámetro, el puntero *punt*, al nuevo tamaño indicado por el segundo parámetro, *tam*.

Si lo que se hace es disminuir el tamaño del espacio de memoria reservado, no hay problema, y la dirección al que apuntaba *punt* no cambia. Si lo que se quiere es aumentar el tamaño, la dirección al que apuntaba *punt* puede cambiar si fuese necesario. Este segundo caso puede dar problemas, ya que podría pasar que no hubiera espacio libre suficiente alrededor de la dirección *punt*, como para reservar el tamaño de memoria que se quiere.



10.6.- free(ptr)

Esta función libera memoria que previamente haya sido reservada con *malloc*, *calloc* o *realloc*. La sintaxis de la llamada es:

free(<punt>**);**

No devuelve nada, simplemente libera el espacio de memoria reservado al que apuntaba el puntero *punt* que se le pasa como parámetro.