Fundamentos de Programación – Programación Estructurada en C:
7.- ARRAYS

Maider Huarte Arrayago

Fundamentos de Programación – Programación Estructurada en C:

7.- ARRAYS

7.- ARRAYS

Un **array** es una colección de variables del mismo tipo, que se referencian por el mismo nombre.

Los arrays pueden ser de varias dimensiones: unidimensionales o vectores, bidimensionales o matrices,... n-dimensionales en general.

7.1.- Arrays unidimensionales

La declaración de un array unidimensional, se hace, en general, de la siguiente forma:

<tipo> <nombre>**[**<capacidad>**];**

Con esa declaración, se reservan en memoria el número de **posiciones de memoria consecutivas** necesarias, para guardar el número, indicado por *capacidad* (un número entero positivo), de variables del tipo *tipo*. Es decir:

```
char caracteres[5];

int enteros[5];
```

Variable	Dirección	Memoria	Tamaño
caracteres[0]	1000	-	1 byte
caracteres[1]	1001	-	1 byte
caracteres[2]	1002	-	1 byte
caracteres[3]	1003	-	1 byte
caracteres[4]	1004	-	1 byte
enteros[0]	2000 ... 2003	-	4 bytes
enteros[1]	2004 ... 2007	-	4 bytes
enteros[2]	2008 ... 2011	-	4 bytes
enteros[3]	2012 ... 2015	-	4 bytes
enteros[4]	2016 ... 2019	-	4 bytes

La capacidad indica el número de elementos del tipo especificado que se pueden almacenar en el array. Cada uno de los elementos se referencia

mediante un índice, que es un número entero perteneciente al rango que siempre empieza en 0 y acaba en (capacidad-1).

Tal y como se ha visto en el ejemplo anterior, en la declaración de un array lo único que hacemos es reservar memoria, para una cierta capacidad. Si no inicializamos los valores de los elementos en la declaración, consideraremos que el array está vacío, es decir, que su tamaño array es 0. Si no metiéramos el contenido que queremos en los elementos del array, al referirnos a alguno, obtendríamos el valor ya contenido en la posición de memoria correspondiente. Así, el tamaño indica en cada momento cuántos elementos contienen valores introducidos de forma explícita (bien por expresiones, bien por medio del usuario del programa), mientras que la capacidad indica cuántos elementos puede contener el array como máximo.

Ejemplo:

```
#include <stdio.h>

int main()
{
    int num[2];
    printf("%d,%d\n",num[0],num[1]); //En pantalla: 222,-845697

    return 0;
}
```

Para poder inicializar un array en su declaración, se indicarán los valores de cada uno de sus elementos de la siguiente forma:

<tipo> <nombre>**[**<capacidad>**]={**<valor 1>**,**<valor 2>**,…,**<valor *capacidad*>**};**

Tal y como se indica en la sintaxis anterior, la forma correcta de inicializar un array en su declaración es indicando los valores que han de tomar cada uno de sus elementos, quedando así el array lleno (*tamaño=capacidad*). Sin embargo, también se puede inicializar con menos valores e incluso con más. Cuando eso ocurre, el compilador lo indica mediante un *warning*. Así, no hay mayor problema si se tiene en cuenta que, en caso de inicializar con menos valores, el tamaño no es igual a la capacidad sino menor (tamaño < capacidad), y que en caso de inicializar con más, los valores sobrantes se ignoran y no se introducen en memoria.

Ejemplo:

```
int n[3]={34,947,67};
int n_1[3]={12,4};
int n_2[3]={85,90,8,9,2};
```

/* Es decir:

Variable	Dirección	Memoria	Tamaño
n[0]	1000 ... 1003	34	4 bytes
n[1]	1004 ... 1007	947	4 bytes
n[2]	1008 ... 1011	67	4 bytes
n_1[0]	2000 ... 2003	12	4 bytes
n_1[1]	2004 ... 2007	4	4 bytes
n_1[2]	2008 ... 2011	-	4 bytes
n_2[0]	2100 ... 2103	85	4 bytes
n_2[1]	2104 ... 2107	90	4 bytes
n_2[2]	2108 ... 2111	8	4 bytes
	2112 ... 2115	-	4 bytes

*/

Así, es **obligatorio indicar la capacidad** del array en su declaración, **si** es que **no se inicializa**. En caso de inicializarse, el procesador reserva tanta memoria como valores indicados, no siendo necesario indicar la capacidad entre los corchetes. La capacidad se mantiene constante durante todo el tiempo en el que el ámbito del array exista (activa o pasivamente) en memoria.

Una vez declarado e inicializado (o no) el array, ya se puede acceder a cualquiera de sus elementos, bien para usar su valor en alguna operación o para cambiarlo, por ejemplo. Para referirnos a un elemento concreto de un array, bastará con poner su nombre y el **índice** correspondiente entre corchetes. Si usáramos un índice no válido, es decir, **un número no entero o fuera del rango [0, *capacidad*-1]**, estaríamos cometiendo un error. Sin embargo, el compilador no indicaría ningún error, porque no se trata de un error sintáctico (los que detecta el compilador) sino de un **error conceptual**. Si el índice erróneo fuese un número no entero o negativo, el procesador reinterpretaría los bits que representen ese número como un número entero positivo (conversión implícita), y accedería al elemento indicado por él. Si el

índice erróneo fuese un número entero mayor o igual a la capacidad, también se accedería a la zona de memoria correspondiente a ese elemento, pero al no estar reservada para el array, el resultado de ese acceso es impredecible, y por lo tanto, un error a evitar.

Ejemplo:

```
#include <stdio.h>

int main()
{
    int n0[1];
    int n1[2];

    n0[0]=0;
    n0[1]=1; //El índice 1 no es válido para n0

    n1[2]=2; //El índice 2 no es válido para n1

    printf("\nn0[0]: %d",n0[0]);
    printf("\nn0[1]: %d",n0[1]);

    printf("\nn1[0]: %d",n1[0]);
    printf("\nn1[1]: %d",n1[1]);
    printf("\nn1[2]: %d",n1[2]);

    return 0;
}

/* En pantalla:
```

El valor de n0[0] no es el que se le ha dado en el programa. La situación en memoria, después de dar los valores a las variables correspondientes, es el siguiente:

Variable	Dirección	Memoria	Tamaño
n1[0]	1245044 ... 1245047	-	4 bytes
n1[1]	1245048 ... 1245051	-	4 bytes
n0[0]	1245052 ... 1245055	2	4 bytes
	1245056 ... 1245059	1	4 bytes

Lo que ha ocurrido es que se han reservado seguidos en memoria, los arrays *n1* y *n0*, en ese orden. Como 2 está fuera de rango como índice del array *n1*, *n1[2]* se considerará como los 4 siguientes bytes a *n1[1]*, que, según la reserva hecha en memoria, coincide

con *n0[0]*; por eso, el valor de *n0[0]* cambia al dar valor a *n1[2]*. Así, la consecuencia de este error conceptual ha sido, en este caso, el cambio de valor de una variable (*n0[0]*) a la que no se pretendía cambiar de valor.

*/

Cuando los valores a introducir en un array estén relacionados entre sí (y/o con el índice en el que se van a introducir), de forma que sigan alguna norma repetitiva, lo más adecuado es recorrer el array mediante alguna instrucción repetitiva, accediendo a cada uno de los elementos a cambiar. Suele hacerse con bucles ***for*** o ***while***.

Ejemplo:

```
char valor[5]={'z','\n','h','9'}; //Sit. 1
int i;

for(i=0;i<5;i++)
    valor[i]='a'+i;

//Sit. 2

/* Así, en memoria:
```

Sit.1:

Variable	Dirección	Memoria	Tamaño
valor[0]	1000	'z'	1 byte
valor[1]	1001	'\n'	1 byte
valor[2]	1002	'h'	1 byte
valor[3]	1003	'9'	1 byte
valor[4]	1004	-	1 byte

Sit.2:

Variable	Dirección	Memoria	Tamaño
valor[0]	1000	'a'	1 byte
valor[1]	1001	'b'	1 byte
valor[2]	1002	'c'	1 byte
valor[3]	1003	'd'	1 byte
valor[4]	1004	'e'	1 byte

```
*/
```

7.1.1.- Paso de arrays a funciones

Las variables en general, se pasan a las funciones por valor o por referencia. Cuando pasamos un elemento de un array a una función, también lo podemos hacer por valor o por referencia. Si, por ejemplo, queremos pasar por valor a una función el elemento 5 (que no el 5º, el 5º es el 4) de un array llamado *tabla*, lo haremos como *tabla[5]*, y para pasarlo por referencia, lo haremos como *&tabla[5]* (y el argumento formal que lo recoge, será un puntero).

Si lo que queremos es pasar un **array completo** a una función, y no solo un elemento, **siempre** lo pasaremos **por referencia**. El **identificador** (nombre) **del array** en sí, **es un puntero**, por lo que **para pasarlo por referencia, no necesita el "&"** por delante. En el encabezamiento de la función, a la hora de indicar el array en los argumentos formales, no hay porqué poner la capacidad, C no hace caso de ella. Es decir, aunque como variables locales internas a una función, los arrays no se pueden declarar sin capacidad, cuando se trate de parámetros de una función, sí, ya que, aunque se pusiera, no se tendría en cuenta.

Lo que ocurre al pasar un array completo a una función, es que se pasa la dirección del elemento 0 del array, y se trabaja, dentro de la función, con el propio array (no con un valor copiado, como ocurre cuando pasamos variables por valor).

Ejemplo:

```
#include <stdio.h>

int suma_array(int []);

int main()
{
    int tabla[10];
    int t,total;

    for(t=0;t<10;t++)
    {
        tabla[t]=t;

        printf("\ntabla[%d]: %d",t,tabla[t]);
    }


    total=suma_array(tabla);

    printf("\n\ntotal: %d",total);

    return 0;
}

int suma_array(int vector[])
{
    int ind,suma=0;

    for(ind=0;ind<10;ind++)
    {
        suma+=vector[ind];
    }

    return suma;
}
```

Al no poner la capacidad en la declaración de la lista de parámetros, se está indicando que se va a pasar la dirección de comienzo de un array, lo que sería equivalente a recoger esa dirección en un puntero. El nombre del array es un puntero en sí mismo, es la dirección del primer elemento del array. Así, las siguientes identidades son ciertas:

int *tabla ≡ int tabla[]	Como argumento formal de una función, la declaración de un array sin capacidad es lo mismo que la declaración de un puntero con el mismo nombre y al mismo tipo de dato.
*tabla ≡ tabla[0]	El contenido del puntero *tabla*, que es un array, es en realidad, el valor del 1er elemento del array *tabla*.
tabla ≡ &tabla[0]	El puntero *tabla*, que recoge un array, es en realidad la dirección del 1er elemento del array *tabla*.

El pasar el array completo por referencia, puede permitir que la misma función pueda ser usada con arrays de diferentes capacidades. Para eso, hay que pasar a la función, aparte del propio array, el **tamaño** y a veces, también la **capacidad** del array. Así, no habrá problemas con el índice dentro de la función, ya que se podrá controlar que esté en el rango [0, *tamaño-1*] o [0, *capacidad-1*], según se necesite. La capacidad será necesaria en funciones que puedan suponer el aumento de tamaño del array, ya que habrá que controlar que el tamaño no sobrepase su límite máximo, es decir, la capacidad.

La capacidad, dado que es un valor que no va a cambiarse dentro de ninguna función (un array se declara con una capacidad concreta y no se puede cambiar), podrá pasarse por valor. Sin embargo, el tamaño puede verse afectado en algunas funciones, por ejemplo, en funciones que sirvan para insertar nuevos elementos o para eliminarlos. Así, en esos casos, el cambio en el tamaño es un dato que debe verse una vez terminada la ejecución de este tipo de funciones. Por eso, puede optarse por pasarlo por referencia y recogerlo en un puntero (como cualquier variable que se pasa por referencia) o hacer que la función devuelva el nuevo valor del tamaño mediante *return*.

Ejemplo:

```
#include <stdio.h>

#define TAM 10

int suma_array(int [],int);

int main()
{
    int tabla[TAM];
    int t,total;

    for(t=0;t<TAM;t++)
    {
        tabla[t]=t;

        printf("\ntabla[%d]: %d",t,tabla[t]);
    }

    total=suma_array(tabla,TAM);

    printf("\n\ntotal: %d",total);

    return 0;
}

int suma_array(int vector[], int t) //Recogemos el valor del tamaño del array en la variable t
{
    int ind,suma=0;

    for(ind=0;ind<t;ind++)
    {
        suma+=vector[ind];
    }

    return(suma);
}
/*Esta función sólo hace un recorrido del array, no cambia para nada su tamaño, por eso,
no es necesario pasarle el tamaño por referencia, ni información sobre la capacidad total
del array. En este caso concreto, como el array está completamente lleno, el tamaño es
igual a la capacidad*/
```

Tal y como la declaración del parámetro que recoge un array completo en una función, se puede hacer con dos formatos diferentes (formato array y formato puntero), el acceso a los elementos del array dentro de la función también se puede hacer en esos dos formatos; incluso, se pueden mezclar.

Ejemplo:

```
/* En el ejemplo anterior, podríamos cambiar la definición de la función suma_array, de
esta forma:

int suma_array(int *vector,int t)
{
    int ind,suma=0;

    for(ind=0;ind<t;ind++)
    {
        suma+=vector[ind];
    }

    return(suma);
}

Incluso, tampoco sería necesario cambiar el prototipo de la función. */
```

Para acceder a los elementos del array con formato puntero, hay que hacer uso de la aritmética de punteros.

7.1.2.- Aritmética de punteros

El **acceso** a los elementos de un array se puede hacer **mediante *aritmética de punteros***, **ya que** los elementos de un array **se guardan en posiciones consecutivas de memoria**.

La aritmética de punteros permite operaciones de suma y resta de valores a punteros, teniendo en cuenta el tipo de dato contenido en la dirección apuntada por el puntero.

Ejemplo:

```
char *p; //por ejemplo, p=1000
int *i; // por ejemplo, i=2000

//Incrementamos los punteros en una unidad.
p++; //p=1001
i=i++; //i=2004
```

Por lo tanto, al gestionar el nombre del array con formato puntero, la siguiente identidad es cierta:

```
*(tabla+5) ≡ tabla[5]
```

7.1.3.- Arrays de punteros

Tal como se definen arrays de elementos de un tipo de dato concreto, **se pueden definir arrays de punteros a un tipo de dato concreto**. Así, un array de 5 punteros a entero sería:

```
int *tabla[5];
```

tabla	Es el nombre del array, un puntero en sí mismo. Apunta al primer elemento del array, que es a su vez un puntero. Es, por tanto, un puntero a puntero.
*tabla	El contenido del puntero tabla, que es un array de punteros. Es el contenido del primer elemento del array, que es el primer puntero del array de punteros
**tabla	Es el contenido de la dirección a la que apunta el primer puntero del array de punteros.

Veamos la gestión de punteros a puntero con el siguiente ejemplo:

```
#include <stdio_ext.h>

int main()
{
    char x,*y,**z;

    x='g'; //Sit.1
    y=&x; //Sit. 2
    z=&y; //Sit. 3

    printf("%c %c %c",x,*y,**z); //En pantalla: g g g

    return 0;
}

/*
Sit. 1: La variable x ocupa 1 byte porque es de tipo char; y y z ocupan 4 bytes, porque son punteros.
```

Variable	Dirección	Memoria	Tamaño
x	1000	'g'	1 byte
y	2000 ... 2003	-	4 bytes
z	3000 ... 3003	-	4 bytes

Sit. 2:

Variable	Dirección	Memoria	Tamaño
x	1000	'g'	1 byte
y	2000 ... 2003	1000	4 bytes
z	3000 ... 3003	-	4 bytes

Sit. 3:

Variable	Dirección	Memoria	Tamaño
x	1000	'g'	1 byte
y	2000 ... 2003	1000	4 bytes
z	3000 ... 3003	2000	4 bytes

Sobre los punteros, hay que tener en cuenta dos cosas:

- A un puntero no se le debe asignar como valor cualquier dirección, sino una dirección controlada por el programa en el momento de ejecución. Las únicas direcciones controladas por el programa que se conocen en el momento de programar, son las que se reserven para las variables declaradas.

- Si a un puntero se le da el valor de una dirección no controlada por el programa, no se podrá acceder al contenido de dicha dirección, parándose la ejecución del programa en cuanto se intente dicho acceso.

Se trata de un error no sintáctico (de hecho, hay compiladores que ni siquiera avisan de ello), que en ejecución pueden causar que el programa se acabe repentinamente.

Ejemplo:

```
#include <stdio_ext.h>

int main()
{
    char *a[3];
    char e0='a',e1='b',e2='c';
    int d=1000;
    //Sit. 1

    a[0]=d; //CUIDADO!
    *a[0]='r';//POSIBLE ERROR EN EJECUCIÓN!

    a[0]=&e0;
    a[1]=&e1;
    a[2]=&e2;
    //Sit. 2

    *a[0]='d';
    *a[1]='e';
    *a[2]='f';
    //Sit. 3

    return 0;
}

/*
Sit. 1: Supongamos que las variables se reservan en memoria de la siguiente manera
```

Variable	Dirección	Memoria	Tamaño
a[0]	1245044 ... 1245047	-	4 bytes
a[1]	1245048 ... 1245051	-	4 bytes
a[2]	1245052 ... 1245055	-	4 bytes
e0	1245056	'a'	1 byte
e1	1245057	'b'	1 byte
e2	1245058	'c'	1 byte
d	1245059 ... 1245062	1000	4 bytes

Sit. 2:

Variable	Dirección	Memoria	Tamaño
a[0]	1245044 ... 1245047	1245056	4 bytes
a[1]	1245048 ... 1245051	1245057	4 bytes
a[2]	1245052 ... 1245055	1245058	4 bytes
e0	1245056	'a'	1 byte
e1	1245057	'b'	1 byte
e2	1245058	'c'	1 byte
d	1245059 ... 1245062	1000	4 bytes

Sit. 3:

Variable	Dirección	Memoria	Tamaño
a[0]	1245044 ... 1245047	1245056	4 bytes
a[1]	1245048 ... 1245051	1245057	4 bytes
a[2]	1245052 ... 1245055	1245058	4 bytes
e0	1245056	'd'	1 byte
e1	1245057	'e'	1 byte
e2	1245058	'f'	1 byte
d	1245059 ... 1245062	1000	4 bytes

*/

7.2.- Arrays bidimensionales

Los arrays bidimensionales, que también se conocen como *matrices*, son arrays n-dimensionales con n=2.

La declaración se hace de la siguiente forma:

<tipo> <nombre>**[**<tamaño_dim_1>**][**<tamaño_dim_2>**];**

Para las matrices, la dimensión 1 serán las filas y la dimensión 2 las columnas, por lo que la declaración de forma más clara se ve con:

<tipo> <nombre>**[**<nº filas>**][**<nº columnas>**];**

La declaración supone la reserva de la memoria necesaria para el array. Se reservan una cantidad c de bytes **seguidos**, que en el caso de los arrays bidimensionales, se calcula como:

c=filas*columnas*(tamaño del tipo de dato)

El orden de asignación de bytes a cada uno de los elementos, se hace siguiendo el orden de las dimensiones, es decir, en el caso de los arrays bidimensionales, se almacenan primero cada uno de los elementos de las columnas de la fila 0, después los elementos de las columnas de la fila 1, etc.

Ejemplo:

int n[2][3];

/* Así, en memoria tendremos:

Variable	Dirección	Memoria	Tamaño
n[0][0]	1245032 ... 1245035	-	4 bytes
n[0][1]	1245036 ... 1245039	-	4 bytes
n[0][2]	1245040 ... 1245043	-	4 bytes
n[1][0]	1245044 ... 1245047	-	4 bytes
n[1][1]	1245048 ... 1245051	-	4 bytes
n[1[2]	1245052 ... 1245055	-	4 bytes

*/

La inicialización explícita de un array n-dimensional en su declaración, se hará como en el caso de los arrays unidimensionales; se puede, incluso, agrupar los valores por dimensiones entre llaves. Para recorrer el array en cada una de sus dimensiones y **acceder a cada uno de sus elementos de uno en uno**, se usan sentencias repetitivas (***for***, ***while***) y un índice para controlar la repetición en cada dimensión. En el caso de las matrices, sólo hay que hacer el recorrido

en dos dimensiones, *filas* y *columnas*, y el orden en el que se haga es indiferente; se suele empezar por la 1ª dimensión.

Ejemplo:

```
#include <stdio.h>

#define MAX_F 2 //Tamaño máximo de las filas
#define MAX_C 3 //Tamaño máximo de las columnas

int main()
{
    int n[MAX_F][MAX_C]={{23,908,75},{187,375,3}};
    int i_f,i_c;

    for(i_f=0;i_f<MAX_F;i_f++)
        for(i_c=0;i_c<MAX_C;i_c++)
            n[i_f][i_c]=(MAX_C*i_f)+i_c;

    return 0;
}

/* Así, en memoria tendremos:
```

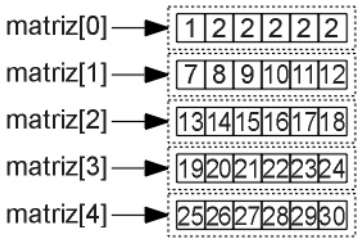
Variable	Dirección	Memoria	Tamaño
n[0][0]	1245032 ... 1245035	0	4 bytes
n[0][1]	1245036 ... 1245039	1	4 bytes
n[0][2]	1245040 ... 1245043	2	4 bytes
n[1][0]	1245044 ... 1245047	3	4 bytes
n[1][1]	1245048 ... 1245051	4	4 bytes
n[1[2]	1245052 ... 1245055	5	4 bytes

```
*/
```

Un array bidimensional es equivalente a un array unidimensional de arrays unidimensionales (vector de vectores). Así, si tenemos:

```
int matriz[5][6];
int *punt;

punt=matriz;
```

1	2	2	2	2	2
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30



Como se ve en la figura, *matriz[i]* es la dirección del array unidimensional de la fila *i*. Así:

- *matriz[i]* es la dirección de memoria donde empiezan a almacenarse de forma secuencial los elementos del array *i* de la matriz. Entonces, la dirección *matriz[i]* es también la dirección del primer elemento del array de la fila *i*, es decir, el contenido de la dirección *matriz[i]*, es el primer elemento del array de la fila *i*:

$$*(matriz[i]) \equiv matriz[i][0]$$

- Al haber hecho *punt=matriz*, podemos acceder a los elementos de la matriz a través de la variable *punt*. Para ello, sólo hay que tener en cuenta que, mediante la aritmética de punteros:

$$matriz[i][j] \equiv *(punt+(columnas*i)+j)$$

- Aunque *matriz[i]* es el puntero al primer elemento del array de la fila *i*, *matriz[i]*+1 no es el puntero al array de la siguiente fila, sino el puntero al siguiente elemento del array de la fila *i*:

$$matriz[i]+1 \neq matriz[i+1]$$

$$matriz[i]+1 \equiv \&matriz[i][1]$$

7.2.1.- Paso de arrays n-dimensionales a funciones

El paso de un **array completo por valor** a una función, sólo se podría hacer pasando todos los elementos del mismo de uno en uno. El encabezamiento de la función tendría que tener tantos argumentos declarados como elementos tuviera el array.

Eso, podría resultar complicado con arrays muy grandes, e incluso en situaciones en las que el programador no tiene porqué saber las dimensiones del array (arrays dinámicos, p. e., arrays cuyas dimensiones se piden al usuario

del programa en tiempo de ejecución). Por eso, lo mejor es pasar el array completo por referencia, y tener en cuenta al escribir la definición de la función, que los cambios que se hagan en los elementos del array se habrán realizado sobre los elementos verdaderos del array y no sobre copias.

El paso de un **array completo por referencia** a una función, se hace poniendo en la llamada a la función el nombre del array (no hace falta poner "&"), que es un puntero en sí. En el encabezamiento de la función, habrá que indicar que se va a recibir como argumento un array; para ello, se declara un array del mismo tipo de elementos y de las mismas dimensiones que el array que se va a pasar, pero la diferencia está en que en esta declaración **se indican** las capacidades de **todas las dimensiones menos la primera**.

Por eso, cuando lo que se va a pasar es un array unidimensional, no se indica su capacidad en la declaración del argumento formal que lo recoge en el encabezamiento de la función. Cuando lo que se pasa es una matriz, sólo se indican el número de columnas.

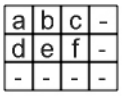
Así, la misma función puede valer para arrays de diferente capacidad en la 1ª dimensión. Para ello, dentro de la función, habrá que saber cuales son los tamaños reales del array, es decir, una cosa es la memoria máxima reservada para el array, y otra que se esté usando toda esa memoria. En cualquier caso, cuando se pasa un array a una función, habrá que pasarle también los valores de los tamaños de las dimensiones necesarias, para que el acceso a los elementos dentro de la función se haga de la forma adecuada (con los índices correctos).

Ejemplo:

/* Programa que crea una matriz de dimensiones máximas 3x4, según los valores introducidos por el usuario, y la rellena con caracteres consecutivos desde la 'a'. */

```
#include <stdio.h>

#define MAX_F 3 //Tamaño máximo de las filas
#define MAX_C 4 //Tamaño máximo de las columnas

void rellenar_matriz(char [][MAX_C],int,int);

int main()
{
    char m[MAX_F][MAX_C];
    int n_f,n_c;

    do
    {
        printf("\nIntroduzca el nº total de filas (máximo %d): ",MAX_F);
        scanf("%d",&n_f);
    }
    while(n_f>MAX_F);

    do
    {
        printf("\nIntroduzca el nº total de columnas (máximo %d): ",MAX_C);
        scanf("%d",&n_c);
    }
    while(n_c>MAX_C);

    rellenar_matriz(m,n_f,n_c);

    return 0;
}

void rellenar_matriz(char m[][MAX_C],int n_f,int n_c)
{
    int i_f,i_c,l;


    for(i_f=0,l=0;i_f<n_f;i_f++)
        for(i_c=0;i_c<n_c;i_c++,l++)
            m[i_f][i_c]='a'+l;
}
```

/* Así, si el usuario elige los tamaños 2 y 3 para las filas y columnas, lo que quiere es lo siguiente:



En memoria tendremos:

Variable	Dirección	Memoria	Tamaño
m[0][0]	1245032	'a'	1 byte
m[0][1]	1245033	'b'	1 byte
m[0][2]	1245034	'c'	1 byte
m[0][3]	1245035	-	1 byte
m[1][0]	1245036	'd'	1 byte
m[1][1]	1245037	'e'	1 byte
m[1][2]	1245038	'f'	1 byte
m[1][3]	1245039	-	1 byte
m[2][0]	1245040	-	1 byte
m[2][1]	1245041	-	1 byte
m[2][2]	1245042	-	1 byte
m[2][3]	1245043	-	1 byte

*/