Arrays Bidimensionales

[en Lenguaje C]

Ejemplo:

Para un grupo de 4 empleados se desea obtener el promedio de ventas de 3 meses.

	mes1	mes2	mes3	Promedio
Pedro	9,00	4,00	2,00	5,00
Juan	5,30	10,00	6,00	7,10
María	4,20	7,00	9,00	6,73
José	6,30	6,80	2,00	5,03

•Un array es una estructura de datos que contiene datos de un mismo tipo, organizados de un modo particular. Si el array es de una dimension, decimos que los datos, estan 'organizados secuencialmente'. Esta idea se generaliza a otras configuraciones posibles, y por eso hablamos de arrays **bidimensionales**, **tridimensionales**, y más.

• De este modo, es posible tratar un conjunto de datos del mismo tipo como si estuviesen dispuestos en una grilla (dos dimensiones) o apilados por capas en tres dimensiones, o en más, **aunque en la memoria se encuentren en forma consecutiva.**

Declaración (*primera forma*) :

<tipo de dato> <nombre_array> [1°dimensión-filas][2°dimensión-columnas];

Ejemplos:

int arrayB[15][7];

float arrayB1[5][10];

double arrayB2[100][100];

La matriz del ejemplo puede declararse como:

float ventas[NUMEMPLE][NUMMES]

lo que equivale a decir:

float ventas[4][3];

Es decir que tenemos dos índices para operar sobre este tipo de arrays

Inicialización:

Puede realizarse en el momento de la declaración:

```
int ventas[4][3]= {7,9,15,13,11,8,5,9,10,2,8,4};  o bien
int ventas[4][3]= {{7,9,15},{13,11,8},{5,9,10},{2,8,4}};
```

Si se omitieran valores en la inicialización, el compilador setea con valor '0' el resto :

```
int ventas[4][3] = {7,9,15};
```

No se puede inicializar todos los elementos de un array en una línea diferente a la de la declaración:

```
int ventas[4][3];
ventas = {7,9,15,13,11,8,5,9,10,2,8,4}; //error
```

No se puede inicializar un array con más elementos de los declarados en la dimensión :

```
int ventas[4][3] = {7,9,15,13,11,8,5,25,2,9,10,4,3,6,7}; //error
```

Inicialización

Otra forma de asignar valores es:

```
ventas[0][0] = 7;
ventas[0][1] =9;
ventas[0][2] = 15;
ventas[1][0] = 13;
ventas[1][1] = 11;
ventas[1][2] =8;
…..
```

Pero es un método poco práctico.

Declaración (*segunda forma*):

<tipo de dato> <nombre_array> [] [2°dimensión-columnas];

Ejemplo: int ventas[][3];

Inicialización

Se puede omitir la primera dimensión si se inicializa en la declaración, es decir que en este caso la inicialización es forzosa:

<tipo de dato>< nombre_array> [][3] = {valor1, valor2,…};

Ejemplo: int ventas[][3] = {7,9,15,13,11,8,5,6,7,1,3,8};

El array toma la dimensión de la cantidad de elementos.

Ingreso de datos:

```
scanf("%d", &dias[0][0]);
```

Posición del elemento en el array

9

Ingreso del valor por teclado en la **posición: fila 0, columna 0**

```
…
scanf("%d", &ventas[0][1]);
scanf("%d", &ventas[0][2]);
…?
```

Ingreso de datos:

```
…
for (i=0; i<filas; i++){
 for(j=0; j<columnas;j++)
   {  printf("Ingrese la vent
      scanf("%d", &ventas
} …
```

```
scanf("%d", &ventas[i][j]);
```

Segunda dimensión (columnas)

Primera dimensión (filas)

Columnas →		ventas[][0]	ventas[][1]	ventas[][2]
Filas		0	1	2
ventas[0][]	0	9,00 ventas[0,0]	4,00 ventas[0,1]	2,00 ventas[0,2]
ventas[1][]	1	5,30 ventas[1,0]	10,00 ventas[1,1]	6,00 ventas[1,2]
ventas[2][]	2	4,20 ventas[2,0]	7,00 ventas[2,1]	9,00 ventas[2,2]
ventas[3][]	3	6,30 ventas[3,0]	6,80 ventas[3,1]	2,00 ventas[3,2]

Acceso:

```
…
for (i=0; i<filas; i++){
 for(j=0; j<columnas;j++)
    printf(«La venta es: %d\n»
} …
```

ventas[i][j]

…unda dimensión (columnas)

Primera dimensión (filas)

Columnas →		ventas[][0]	ventas[][1]	ventas[][2]
Filas		**0**	**1**	**2**
ventas[0]	**0**	9,00 ventas[0,0]	4,00 ventas[0,1]	2,00 ventas[0,2]
ventas[1]	**1**	5,30 ventas[1,0]	10,00 ventas[1,1]	6,00 ventas[1,2]
ventas[2]	**2**	4,20 ventas[2,0]	7,00 ventas[2,1]	9,00 ventas[2,2]
ventas[3]	**3**	6,30 ventas[3,0]	6,80 ventas[3,1]	2,00 ventas[3,2]

Arrays y direcciones de memoria:

Tal como sucede con los arrays unidimensionales, la palabra o declaración '**ventas**' **corresponde a la dirección de comienzo de la matriz** ventas[4][3].

En C, una matriz se maneja como un array de arrays.

Eso hace, que sea válido usar el **nombre de la matriz con un solo subíndice para indicar la dirección de comienzo de cada fila**. Así,

ventas identifica el array de direcciones de filas.

ventas = ventas[0] = &ventas[0][0]

ventas + 1 = ventas[1] = &ventas[1][0]

.....

La dirección de cada fila i es ventas[i] , con un solo subíndice

Arrays y direcciones de memoria:

De acuerdo a eso puede decirse que si ...

La matriz ventas comienza en la dirección 1000

1000
1004
1008
1012
1016
1020
1024
1028
1032
1036
1040
1044

fila 0 comienza en 1000 = ventas[0]

fila 1 comienza en 1012 = ventas[1]

fila 2 comienza en 1024 = ventas[2]

fila 3 comienza en 1036 = ventas[3]

Arrays y direcciones de memoria:

La direcciones –considerando el formato de tabla- podrían verse de esta manera:

1000	1004	1008
1012	1016	1020
1024	1028	1032
1036	1040	1044

Arrays y direcciones de memoria:

Es decir:

	mes1	mes2	mes3
Pedro	9,00	4,00	2,00
direcciones	**1000**	**1004**	**1008**
María	4,20	7,00	9,00
direcciones	**1012**	**1016**	**1020**
Miguel	8,50	5,00	5,00
direcciones	**1024**	**1028**	**1032**
Walter	10,00	5,30	7,00
direcciones	**1036**	**1040**	**1044**

Arrays y direcciones de memoria:

Cada dirección puedo obtenerla escribiendo: **&<nombre_array>[1°d][2°d]:**

9	4	2
1000	1004	1008
&ventas[0,0]	&ventas[0,1]	&ventas[0,2]
5,3	10	6
1012	1016	1020
&ventas[1,0]	&ventas[1,1]	&ventas[1,2]
4,2	7	9
1024	1028	1032
&ventas[2,0]	&ventas[2,1]	&ventas[2,2]
6,3	6,8	2
1036	1040	1044
&ventas[3,0]	&ventas[3,1]	&ventas[3,2]

Sabiendo que el nombre de un array es su dirección de comienzo puede decirse que: ventas = ventas[0]=&ventas[0][0]

Arrays y direcciones de memoria:

Se deduce que invocando solo la primera dimensión obtenemos la dirección de comienzo de cada fila:

9	4	2
1000	1004	1008
&ventas[0,0]	&ventas[0,1]	&ventas[0,2]
ventas[0]	ventas[0]+1	ventas[0]+2
5,3	10	6
1012	1016	1020
&ventas[1,0]	&ventas[1,1]	&ventas[1,2]
ventas[1]	ventas[1]+1	ventas[1]+2
4,2	7	9
1024	1028	1032
&ventas[2,0]	&ventas[2,1]	&ventas[2,2]
ventas[2]	ventas[2]+1	ventas[2]+2
6,3	6,8	2
1036	1040	1044
&ventas[3,0]	&ventas[3,1]	&ventas[3,2]
ventas[3]	ventas[3]+1	ventas[3]+2

Arrays y direcciones de memoria:

Entonces si, por ejemplo, ventas[3] es la dirección de la fila 3 (1036), considerando el operador de desreferencia *, se observa que:

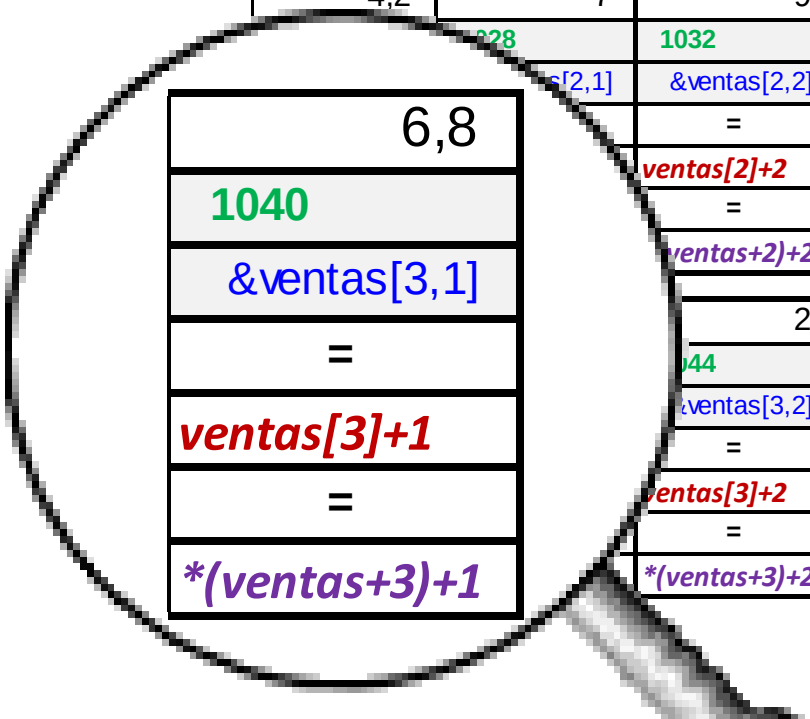
ventas [3] = *(ventas+3) = &ventas[3][0]
ventas [3]+1= *(ventas+3)+1 = &ventas[3][1]
ventas [3]+2= * (ventas+3)+2) = &ventas[3][1]

9	4	2
1000	1004	1008
&ventas[0,0]	&ventas[0,1]	&ventas[0,2]
=	=	=
ventas[0]	*ventas[0]+1*	*ventas[0]+2*
=	=	=
**(ventas+0)+0*	**(ventas+0)+1*	**(ventas+0)+2*

5,3	10	6
1012	1016	1020
&ventas[1,0]	&ventas[1,1]	&ventas[1,2]
=	=	=
ventas[1]	*ventas[1]+1*	*ventas[1]+2*
=	=	=
**(ventas+1)+0*	**(ventas+1)+1*	**(ventas+1)+2*

4,2	7	9
	028	1032
	s[2,1]	&ventas[2,2]
		=
		ventas[2]+2
		=
		ventas+2)+2

2
044
&ventas[3,2]
=
entas[3]+2
=
**(ventas+3)+2*



Arrays y direcciones de memoria:

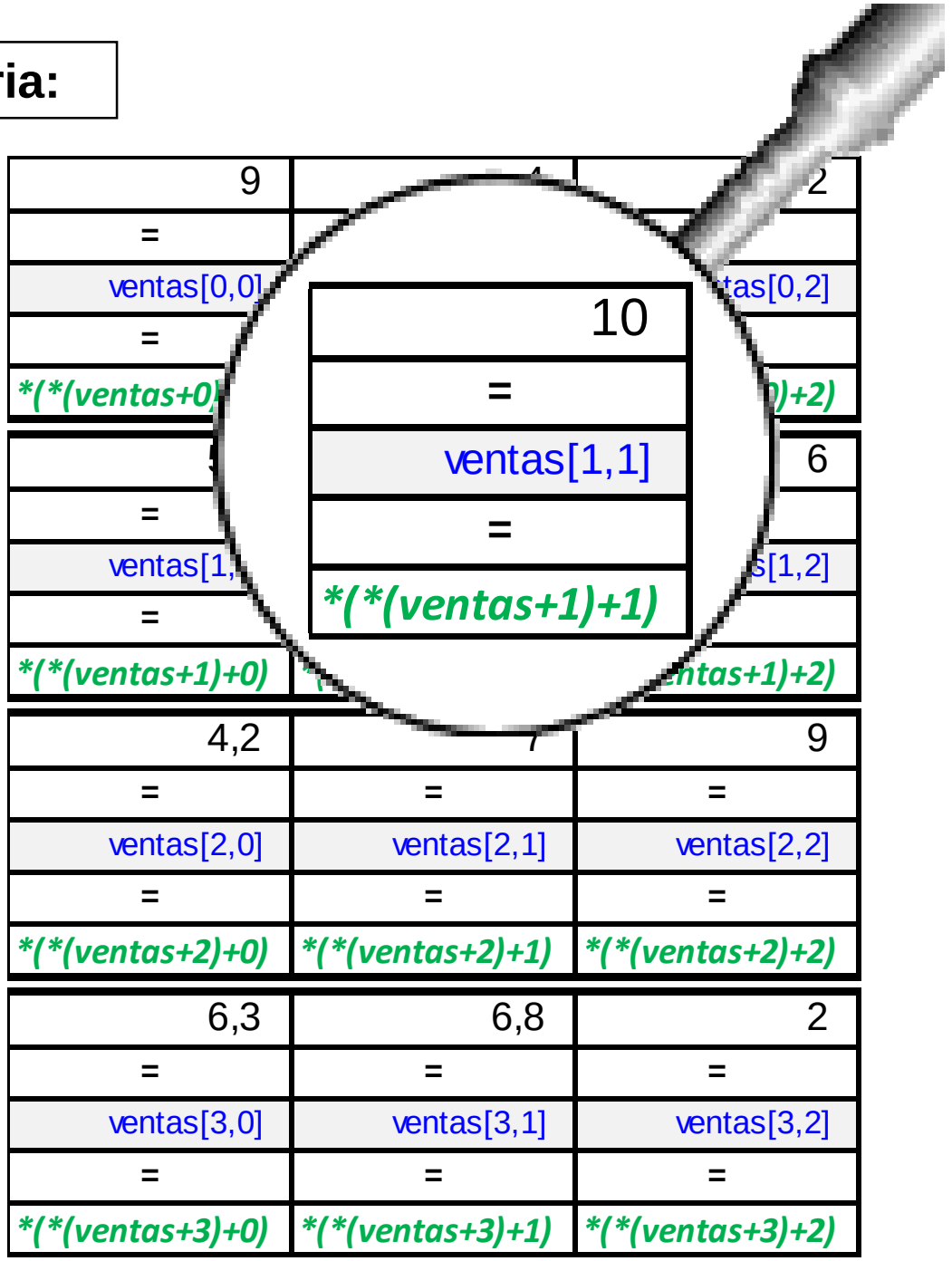
Siguiendo el mismo razonamiento, si 'ventas' es la dirección de comienzo del array de direcciones de filas, considerando una fila en particular, por ejemplo:

***(*(ventas+2)+0)**
***(*(ventas+2)+1)**
***(*(ventas+2)+2)**

Obtenemos el elemento alojado

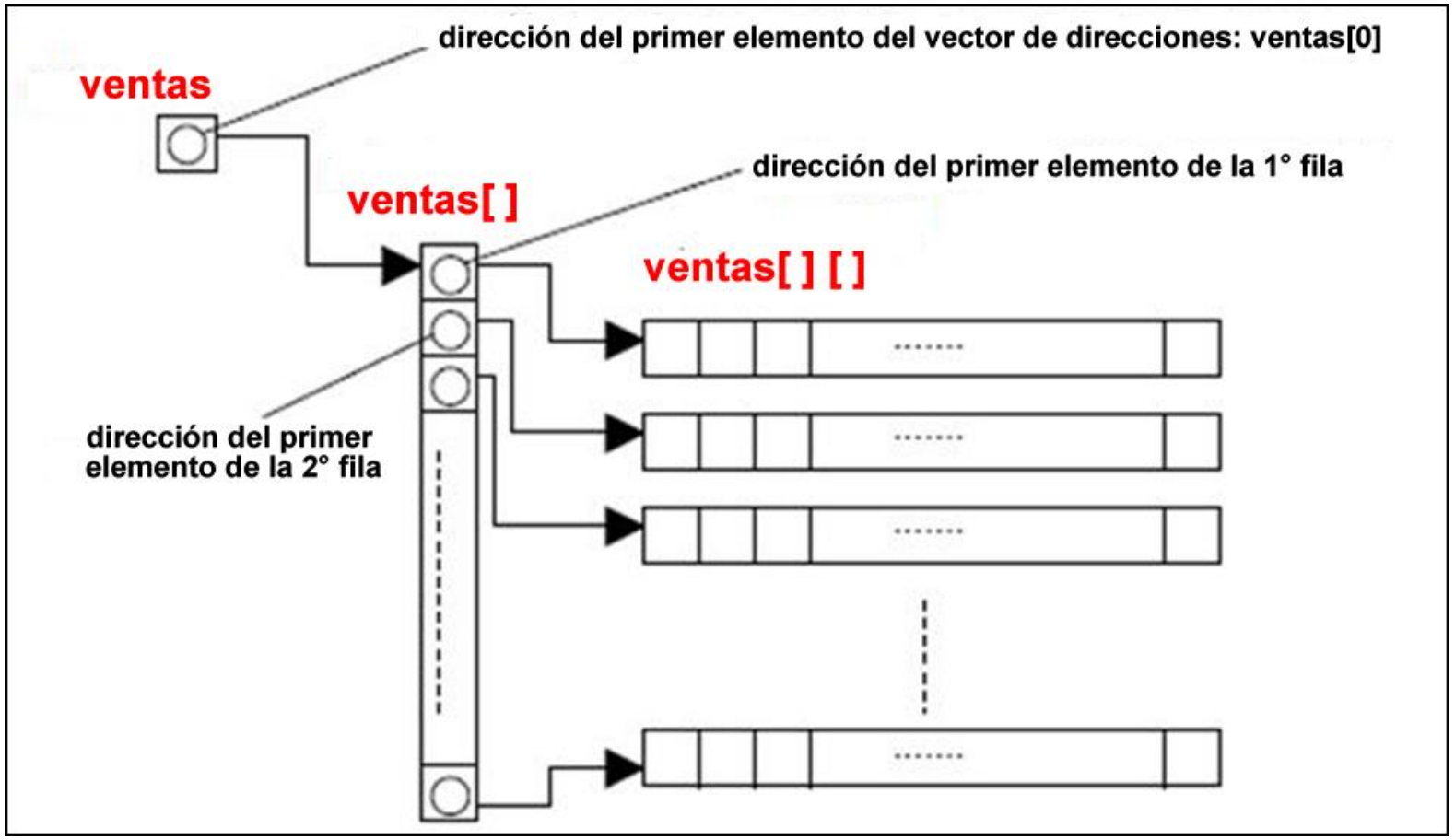
Ciclo de ejemplo:

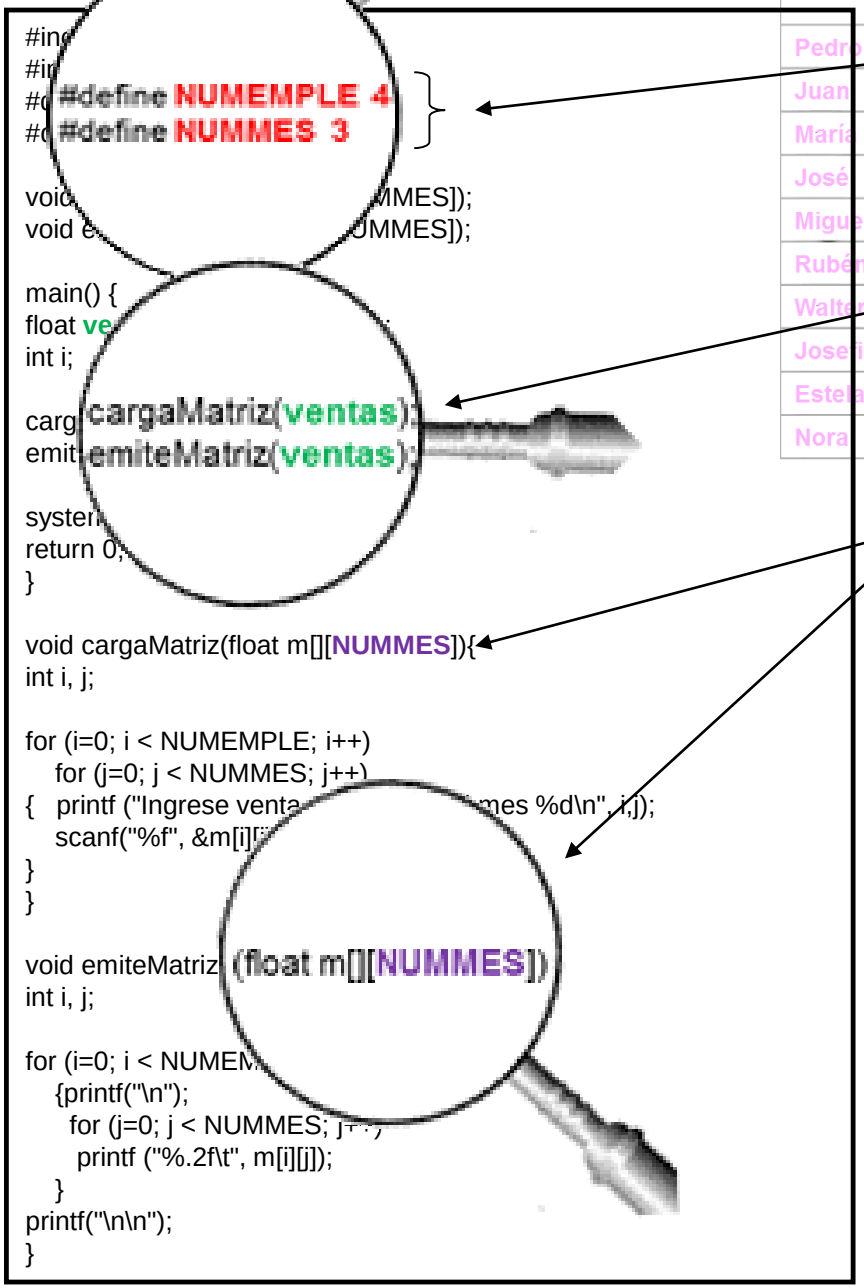
```
printf("\n Contenido de la matriz :\n");
for(i=0;i<filas;i++)
 {printf("\n");
  for (j=0;j<columnas;j++)
    printf("\n %.2f\n", *(*(ventas+i)+j));}
```



Arrays y direcciones de memoria:

Resumiendo, la figura siguiente muestra la relación entre matrices y vectores de punteros:





	mes1	mes	mes3	Promedio
Pedro				5,00
Juan				7,10
María				6,73
José				5,03
Miguel	8,50	5,00	5,00	6,17
Rubén	4,00	9,00	8,00	7,00
Walter			7,00	7,43
Josefina			2,00	4,40
Estela			3,00	4,60
Nora			8,00	7,00

Se puede dimensionar la matriz con constantes simbólicas.

Se pasa como parámetro el nombre de la matriz.

En la definición se escribe la segunda dimensión

```
#include <stdio.h>
#include <stdlib.h>
#define NUMEMPLE 10
#define NUMMES 3

/*se agregan 1) función que calcula la nota media; recibe
por parámetro la fila de la matriz que corresponde al
mes cuya venta media se va a calcular y 2) emisión de
la matriz por fila*/

void cargaMatriz(float m[][NUMMES]);
void emiteMatriz(float m[][NUMMES]);
void emiteMatrizXdirF(float * ventas);
float calculoMedia(float * ventas);

main() {
float ventas[][NUMMES]={0.0};
int i;

cargaMatriz(ventas);
emiteMatriz(ventas);

for (i=0; i < NUMEMPLE; i++)
emiteMatrizXdirF(ventas[i]);

for (i=0; i < NUMEMPLE; i++)
printf("\nLa venta del empleado %d es %.2f\n", i+1,
calculoMedia(ventas[i]));

system("pause");
return 0;
}
```

```
void cargaMatriz(float m[][NUMMES]){
int i, j;
for (i=0; i < NUMEMPLE; i++)
    for (j=0; j < NUMMES; j++)
{   printf ("Ingrese venta empleado %d mes %d\n", i,j);
    scanf("%f", &m[i][j]);
}}

void emiteMatriz(float m[][NUMMES]){
int i, j;
for (i=0; i < NUMEMPLE; i++)
    {printf("\n");
      for (j=0; j < NUMMES; j++)
       printf ("%.2f\t", m[i][j]);}
     printf("\n\n");  }

void emiteMatrizXdirF(float * ventas){
int i, j;
for (i=0; i < NUMMES; i++)
        printf ("%.2f\t", ventas[i]);
        printf("\n\n");}

float calculoMedia(float * ventas)
{
float media=0.0;
int i;
    for (i=0; i < NUMMES; i++)
    media+=ventas[i];
    media/=NUMMES;
return media;
}
```

ARRAYS BIDIMENSIONALES:

De acuerdo a la imagen anterior, un arreglo multidimensional puede ser visto en varias formas en C, por ejemplo:

***Un arreglo de dos dimensiones es un arreglo de una dimensión, donde cada uno de los elementos es en sí mismo un arreglo*.**

Por lo tanto, la notación: **a[n][m]** nos indica que los elementos del arreglo están guardados fila por fila.

Cuando se pasa una arreglo bidimensional a una función se debe especificar el número de columnas - el número de filas es irrelevante. La razón de lo anterior, son los apuntadores. C requiere conocer cuántas son las columnas para que pueda saltar de fila en fila en la memoria.

Considerando que una función deba recibir, por ejemplo:
int a[5][35],
se puede declarar el argumento de la función como:

f(int a[][35]) { ..... }

o aún

f(int (*a)[35]) { ..... }

En el último ejemplo se requieren los paréntesis ***(*a)*** ya que ***[ ]*** tiene una **precedencia** más alta que ***.

ARRAYS BIDIMENSIONALES:

Por lo tanto:

int (*a)[35]; declara un apuntador a un arreglo de 35 enteros, y por ejemplo si hacemos la siguiente referencia ***a+2***, nos estaremos refiriendo a la dirección del primer elemento que se encuentran en el tercer renglón de la matriz supuesta, mientras que

int *a[35]; declara un arreglo de 35 apuntadores a enteros.

Ahora veamos la diferencia (sutil) entre apuntadores y arreglos. El manejo de cadenas es una aplicación común de esto. Considera:

char *nomb[10];
char anomb[10][20];

En donde es válido hacer *nomb[3][4]* y *anomb[3][4]* en C. Sin embargo:

anomb es un arreglo *verdadero* de 200 elementos de dos dimensiones tipo ***char***. En cambio ***nomb*** tiene 10 apuntadores a elementos.

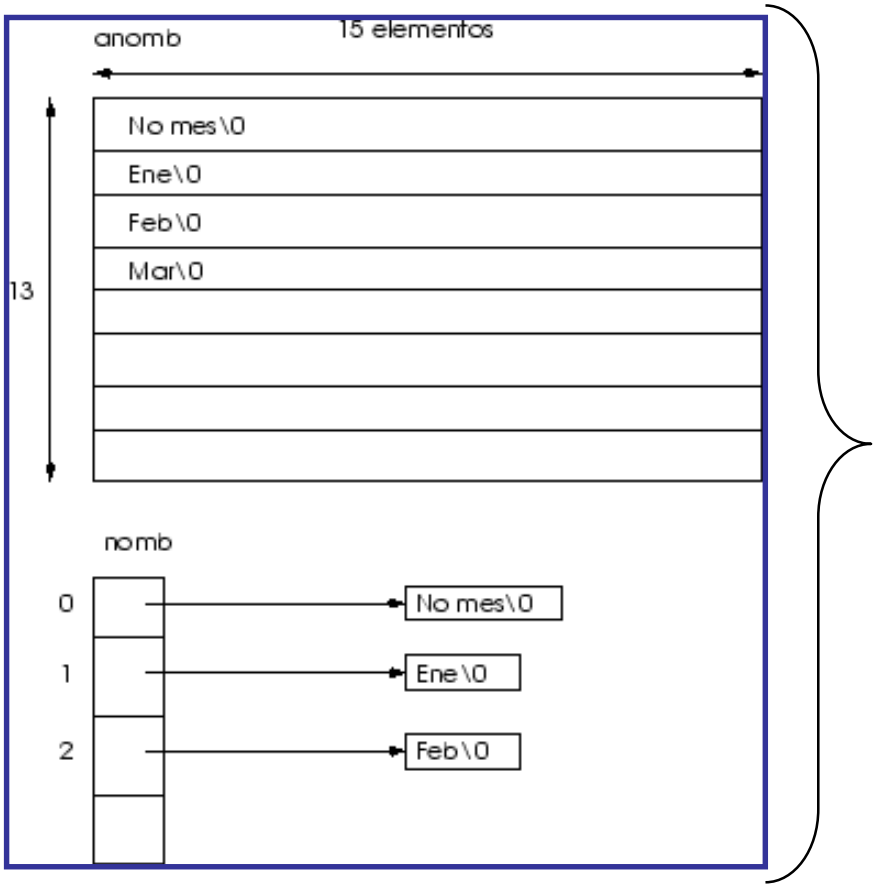
El acceso de los elementos ***anomb*** en memoria se hace bajo la siguiente fórmula *20*renglon + columna + dirección_base*

Si cada apuntador en ***anomb*** indica un arreglo de 20 elementos entonces y solamente entonces 200 chars estarán disponibles (10 elementos). **Con el primer tipo de declaración se tiene la ventaja de que cada apuntador puede apuntar a arreglos de diferente longitud.**

ARRAYS BIDIMENSIONALES:

Considerar:

char *nomb[] = { "No mes", "Ene", "Feb", "Mar", ... };
char anomb[][15] = { "No mes", "Ene", "Feb", "Mar", ... };



*Según la figura, puede indicarse que **se hace un manejo más eficiente del espacio haciendo uso de un arreglo de apuntadores** que usando un arreglo bidimensional.*

La inicialización de arreglos de apuntadores es una aplicación ideal para un arreglo estático interno, por ejemplo:

func_cualquiera() { static char *nomb[] = { "No mes", "Ene", "Feb", "Mar", ... }; }

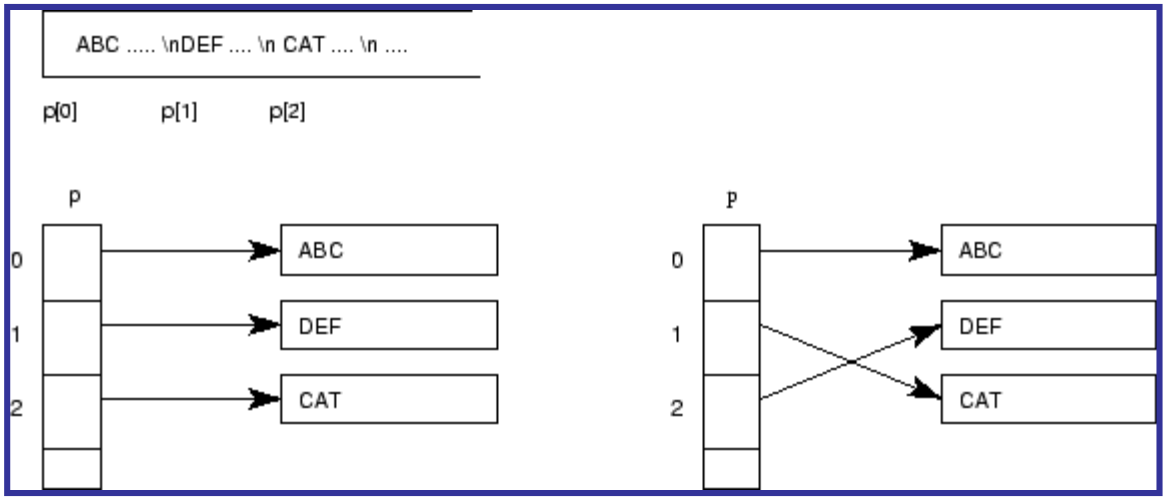
Recordando que con el especificador de almacenamiento de clase static se reserva en forma permanente memoria el arreglo, mientras el código se esta ejecutando.

Arreglo de 2 dimensiones VS. Arreglo de apuntadores

ARRAYS BIDIMENSIONALES:

Como mencionamos, en C se pueden tener arreglos de apuntadores ya que los apuntadores son variables. A continuación se muestra un ejemplo de su uso: **ordenar las líneas de un texto de diferente longitud**. *(Los arreglos de apuntadores son una representación de datos que manejan de una forma eficiente y conveniente líneas de texto de longitud variable. ¿Cómo se puede hacer lo anterior?)*

- Guardar todas las líneas en un arreglo de tipo ***char*** grande. Observando que ***\n*** marca el fin de cada línea.
- Guardar los apuntadores en un arreglo diferente donde cada apuntador apunta al primer caracter de cada línea.
- Comparar dos líneas usando la función de la biblioteca estándar ***strcmp()***.
- Si dos líneas no están en orden -- intercambiar (swap) los apuntadores (no el texto).



Arreglos de apuntadores (Ejemplo de ordenamiento de cadenas).

De esta manera se elimina el manejo complicado del almacenamiento y la alta sobrecarga por el movimiento de líneas.

Fin