CAPÍTULO 8

PUNTEROS

La memoria puede considerarse como una enorme cantidad de posiciones de almacenamiento de información perfectamente ordenadas. Cada posición es un octeto o byte de información. Cada posición viene identificada de forma inequívoca por un número que suele llamarse dirección de memoria. Cada posición de memoria tiene una dirección única. Los datos de nuestros programas se guardan en esa memoria.

La forma en que se guardan los datos en la memoria es mediante el uso de variables. Una variable es un espacio de memoria reservado para almacenar un valor: valor que pertenece a un rango de valores posibles. Esos valores posibles los determina el tipo de dato de esa variable. Dependiendo del tipo de dato, una variable ocupará más o menos bytes de la memoria, y codificará la información de una u otra manera.

Si, por ejemplo, creamos una variable de tipo ***float***, estaremos reservando cuatro bytes de memoria para almacenar sus posibles valores. Si, por ejemplo, el primero de esos bytes es el de posición

ABD0:FF31 (es un modo de escribir: en definitiva estamos dando 32 bits para codificar las direcciones de la memoria), el segundo byte será el ABD0:FF32, y luego el ABD0:FF33 y finalmente el ABD0:FF34. La dirección de memoria de esta variable es la del primero de sus bytes; en este caso, diremos que toda la variable ***float*** está almacenada en ABD0:FF31. Ya se entiende que al hablar de variables ***float***, se emplean un total de 4 bytes.

Ese es el concepto habitual cuando se habla de la posición de memoria o de la dirección de una variable.

Además de los tipos de dato primitivos ya vistos en un tema anterior, existe un C un tipo de dato especial, que ofrece muchas posibilidades y confiere al lenguaje C de una filosofía propia. Es el tipo de dato puntero. Mucho tiene que ver ese tipo de dato con la memoria de las variables. Este capítulo está dedicado a su presentación.

Definición y declaración.

Una variable tipo puntero es una variable que contiene la dirección de otra variable.

Para cada tipo de dato, primitivo o creado por el programador, permite la creación de variables puntero hacia variables de ese tipo de dato. Existen punteros a ***char***, a ***long***, a ***double***, etc. Son nuevos tipos de dato: *puntero a* ***char***, *puntero a* ***long***, *puntero a* ***double***,…

Y como tipos de dato que son, habrá que definir para ellos un dominio y unos operadores.

Para declarar una variable de tipo puntero, la sintaxis es similar a la empleada para la creación de las otras variables, pero precediendo al nombre de la variable del carácter asterisco (*).

tipo *nombre_puntero;

Por ejemplo:

short int **p;*

Esa variable *p* así declarada será una variable *puntero a* ***short***, que no es lo mismo que *puntero a* ***float***, etc.

En una misma instrucción, separados por comas, pueden declararse variables puntero con otras que no lo sean:

long *a, b, *c;*

Se han declarado dos variables de tipo ***long*** y una tercera que es *puntero a* ***long***.

Dominio y operadores para los punteros

El dominio de una variable puntero es el de las direcciones de memoria. En un PC las direcciones de memoria se codifican con 32 bits (4 bytes), y toman valores desde 0 hasta FFFFFFFF, en base hexadecimal.

El operador ***sizeof***, aplicado a cualquier variable de tipo puntero, devuelve el valor 4.

Una observación importante: si un PC tiene direcciones de 32 bytes… ¿cuánta memoria puede llegar a direccionar?: Pues con 32 bits es posible codificar hasta 2^{32} bytes, es decir, hasta $4 \cdot 2^{30}$ bytes, es decir 4 Giga bytes de memoria.

Pero sigamos con los punteros. Ya tenemos el dominio. Codificará, en un formato similar al de los enteros largos sin signo, las direcciones de toda nuestra memoria. Ese será su dominio de valores.

Los operadores son los siguientes:

Operador dirección (&): Este operador se aplica a cualquier variable, y devuelve la dirección de memoria de esa variable. Por ejemplo, se puede escribir:

```
long x, *px;
px = &x;
```

Y así se ha creado una variable *puntero a* ***long*** llamada *px*, que servirá para almacenar direcciones de variables de tipo ***long***. Mediante la segunda instrucción asignamos a ese puntero la dirección de la variable *x*. Habitualmente se dice que ***px* apunta a *x***.

El operador dirección no es propio de los punteros, sino de todas las variables. Pero no hemos querido presentarlo hasta el momento en que por ser necesario creemos que también ha de ser fácilmente comprendido. De hecho este operador ya lo usábamos, por ejemplo, en la función *scanf*, cuando se le indica a esa función "dónde" queremos que almacene el dato que introducirá el usuario por teclado: por eso, en esa función precedíamos el nombre de la variable cuyo valor se iba a recibir por teclado con el operador &.

Hay una excepción en el uso de este operador: no puede aplicarse sobre una variable que haya sido declarada ***register***. El motivo es claro: al ser una variable ***register***, le hemos indicado al compilador que no almacene su información en la memoria sino en un registro de la ALU. Y si la variable no está en memoria, no tiene sentido que le solicitemos la dirección de donde no está.

Operador indirección (*): Este operador sólo se aplica a los punteros. Al aplicar a un puntero el operador indirección, se obtiene el contenido de la posición de memoria "apuntada" por el puntero. Supongamos:

```
float pi = 3.14, *pt;
pt = &pi;
```

Con la primera instrucción, se han creado dos variables:

<pi, ***float****,* R_1*, 3.14>* y *<pt,* ***float*****,* R_2*, ¿? >*

Con la segunda instrucción damos valor a la variable puntero:

<pt, ***float*****,* R_2*,* R_1*>*

Ahora la variable puntero *pt* vale R_1, que es la dirección de memoria de la variable *pi*.

Hablando ahora de la variable *pt*, podemos hacernos tres preguntas, todas ellas referidas a direcciones de memoria.

1. ¿Dónde está *pt*? Porque *pt* es una variable, y por tanto está ubicada en la memoria y tendrá una dirección. Para ver esa dirección, basta aplicar a *pt* el operador dirección &. *pt* está en *&pt*.
2. ¿Qué vale *pt*? Y como *pt* es un puntero, *pt* vale o codifica una determinada posición de memoria. Su valor pertenece al dominio de direcciones y está codificado mediante 4 bytes. En concreto, *pt* vale la dirección de la variable *pi*. *pt* vale *&pi*.
3. ¿Qué valor está almacenado en esa dirección de memoria a donde apunta *pt*? Esta es una pregunta muy interesante, donde se muestra la gran utilidad que tienen los punteros. Podemos llegar al valor de cualquier variable tanto si disponemos de su nombre como si disponemos de su dirección. Podemos llegar al valor de la posición de memoria apuntada por *pt*, que como es un puntero a ***float***, desde el puntero tomará ese byte y los tres siguientes como el lugar donde se aloja una variable ***float***. Y para llegar a ese valor, disponemos del operador indirección. El valor codificado en la posición almacenada en *pt* es el contenido de *pi*: **pt* es 3.14.

Al emplear punteros hay un peligro de confusión, que desconcierta al principiante: al hablar de la dirección del puntero es fácil no entender si nos referimos a la dirección que trae codificada en sus cuatro bytes, o la posición de memoria dónde están esos cuatro bytes del puntero que codifican direcciones.

Es muy importante que las variables puntero estén correctamente direccionadas. Trabajar con punteros a los que no se les ha asignado una dirección concreta conocida (la dirección de una variable) es muy peligroso. En el caso anterior de la variable *pi*, se puede escribir:

**pt = 3.141596;*

y así se ha cambiado el valor de la variable *pi*, que ahora tiene algunos decimales más de precisión. Pero si la variable *pt* no estuviera correctamente direccionada mediante una asignación previa… ¿en qué zona de la memoria se hubiera escrito ese número 3.141596? Pues en la posición que, por defecto, hubiera tenido esos cuatro bytes que codifican el valor de la variable *pt*: quizá una dirección de otra variable, o a mitad entre una variable y otra; o en un espacio de memoria no destinado a almacenar datos, sino instrucciones, o el código del sistema operativo,… En general, las consecuencias de usar punteros no inicializados, son catastróficas para la buena marcha de un ordenador. Detrás de un programa que "cuelga" al ordenador, muchas veces hay un puntero no direccionado.

Pero no sólo hay que inicializar las variables puntero: hay que inicializarlas bien, con coherencia. No se puede asignar a un puntero a un tipo de dato concreto la dirección de una variable de un tipo de dato diferente. Por ejemplo:

```
float x, px;
long y;
px = &y;
```

Si ahora hacemos referencia a **px*… ¿trabajaremos la información de la variable *y* como ***long***, o como ***float***? Y peor todavía:

```
float x, px;
char y;
px = &y;
```

Al hacer referencia a **px*… ¿leemos la información del byte cuya dirección es la de la variable *y*, o también se va a tomar en consideración los otros tres bytes consecutivos? Porque la variable *px* considera que apunta a variables de 4 bytes, que pasa eso es un puntero a ***float***. Pero la posición que le hemos asignado es la de una variable tipo ***char***, que únicamente ocupa un byte.

El error de asignar a un puntero la dirección de una variable de tipo de dato distinto al puntero está, de hecho, impedido por el compilador, y si

encuentra una asignación de esas características, aborta el proceso de compilación.

Punteros y vectores

Los punteros sobre variables simples tienen una utilidad clara en las funciones. Allí los veremos con detenimiento. Lo que queremos ver ahora es el uso de punteros sobre arrays.

Un array, o vector, es una colección de variables, todas del mismo tipo, y colocadas en memoria de forma consecutiva. Si creo una array de cinco variables ***float***, y el primero de los elementos queda reservado en la posición de memoria FF54:AB10, entonces no cabe duda que el segundo estará en FF54:AB14, y el tercero en FF54:AB18, y el cuarto en FF54:AB1C y el último en FF54:AB20.

Supongamos la siguiente situación:

```
long a[10];
long *pa;
pa = &a[0];
```

Y con esto a la vista, pasamos ahora a presentar otros dos operadores, muy usados con los punteros.

Operador incremento (++) y decremento (––): Estos operadores no son nuevos, y ya los conocemos. Todas sus propiedades que se vieron con los enteros siguen vigentes ahora con las direcciones de memoria.

Desde luego, si el incremento es sobre el contenido de la variable apuntada sobre el puntero (**pa)++;*, no hay nada que añadir: se incrementará el contenido de esa variable apuntada, de la misma forma que si el operador estuviera aplicado sobre la variable apuntada misma: es lo mismo que escribir *a[0]++;*.

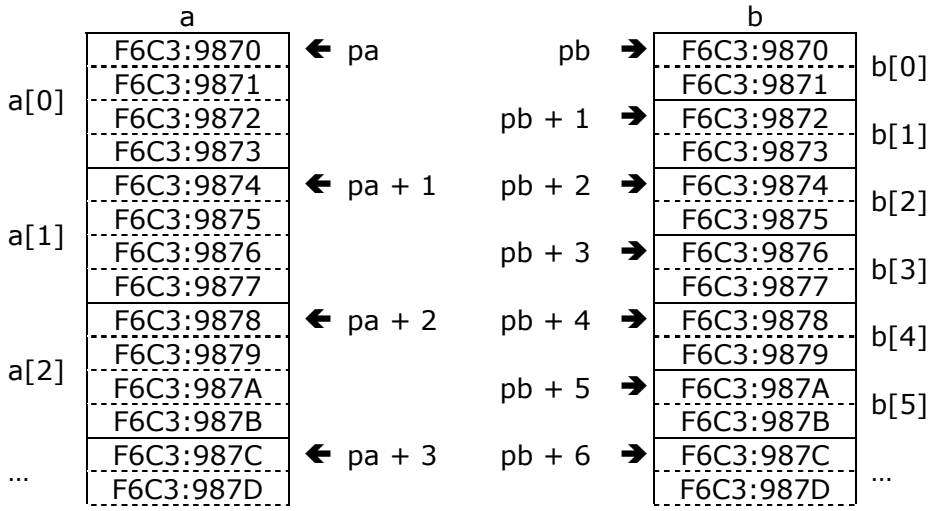
Nos referimos a incrementar el valor del puntero. ¿Qué sentido tiene incrementar en 1 una dirección de memoria? El sentido será el de acudir

al siguiente valor situado en la memoria. Y si el puntero es de tipo ***long***, y apunta a variables ***long***, entonces lo que se espera cuando se incremente un 1 ese puntero es que su valor codificado se incremente en 4: porque 4 es el número de bytes que deberemos saltar para pasar de apuntar a una variable ***long*** a pasar a apuntar a otra variable del mismo tipo almacenada de forma consecutiva.

Es decir, que si *pa* contiene la dirección de *a[0]*, entonces *pa + 1* es la dirección del elemento *a[1]*, y *pa + 9* es la dirección del elemento *a[9]*.

Y en el siguiente ejemplo, tenemos:

```
long a[10], *pa;
short b[10], *pb;
pa = &a[0];
pb = &b[0];
```



Al ir aumentando el valor del puntero, nos vamos desplazando por los distintos elementos del vector, de tal manera que hablar de *a[0]* es lo mismo que hablar de **pa*; y hablar de *a[1]* es lo mismo que hablar de **(pa + 1)*; y, en general, hablar de *a[i]* es lo mismo que hablar de **(pa + i)*. Y lo mismo si comentamos el ejemplo de las variables de tipo ***short***.

La operatoria o aritmética de punteros tiene en cuenta el tamaño de las variables que se recorren. En el siguiente programa, y en la salida que ofrece por pantalla, se puede ver este comportamiento de los punteros. Sea cual sea el tipo de dato del puntero y de la variable a la que apunta, si calculo la resta entre dos punteros situados uno al primer elemento de un array y el otro al último, esa diferencia será la misma, porque la resta de direcciones indica cuántos elementos de este tipo hay (caben) entre esas dos direcciones. En nuestro ejemplo, todas esas diferencias valen 9. Pero si lo que se calcula es el número de bytes entre la última posición (apuntada por el segundo puntero) y la primera (apuntada por el primer puntero), entonces esa diferencia sí dependerá del tamaño de la variable del array.

```
#include <stdio.h>
void main(void)
{
char c[10], *pc1, *pc2;
short h[10], *ph1, *ph2;
float f[10], *pf1, *pf2;
double d[10], *pd1, *pd2;
long double ld[10], *pld1, *pld2;

pc1 =   &c[0];    pc2 =   &c[9];
ph1 =   &h[0];    ph2 =   &h[9];
pf1 =   &f[0];    pf2 =   &f[9];
pd1 =   &d[0];    pd2 =   &d[9];
pld1 = &ld[0];    pld2 = &ld[9];

printf(" pc2(%p) -  pc1(%p) = %hd\n",pc2,pc1,pc2 - pc1);
printf(" ph2(%p) -  ph1(%p) = %hd\n",ph2,ph1,ph2 - ph1);
printf(" pf2(%p) -  pf1(%p) = %hd\n",pf2,pf1,pf2 - pf1);
printf(" pd2(%p) -  pd1(%p) = %hd\n",pd2,pd1,pd2 - pd1);
printf("pld2(%p) - pld1(%p) = %hd\n",pld2,pld1,pld2 - pld1);
printf("\n\n");
printf("(long)pc2-(long)pc1=%3ld\n",(long)pc2-(long)pc1);
printf("(long)ph2-(long)ph1=%3ld\n",(long)ph2-(long)ph1);
printf("(long)pf2-(long)pf1=%3ld\n",(long)pf2-(long)pf1);
printf("(long)pd2-(long)pd1=%3ld\n",(long)pd2-(long)pd1);
printf("(long)pld2-(long)pld1=%3ld\n",(long)pld2-(long)pld1);
}
```

Que ofrece, por pantalla, el siguiente resultado:

```
 pc2(0012FF89) -  pc1(0012FF80) = 9
 ph2(0012FF62) -  ph1(0012FF50) = 9
```

```
 pf2(0012FF4C) -  pf1(0012FF28) = 9
 pd2(0012FF20) -  pd1(0012FED8) = 9
pld2(0012FECE) - pld1(0012FE74) = 9

 (long)pc2 -  (long)pc1 =   9
 (long)ph2 -  (long)ph1 =  18
 (long)pf2 -  (long)pf1 =  36
 (long)pd2 -  (long)pd1 =  72
(long)pld2 - (long)pld1 =  90
```

Que hemos de interpretar bien y entender.

Repetimos: al calcular la diferencia entre el puntero que apunta al noveno elemento de la matriz y el que apunta al elemento cero, en todos los casos el resultado ha de ser 9: porque en la operatoria de punteros, independientemente del tipo del puntero, lo que se obtiene es el número de elementos que hay entre las dos posiciones de memoria señaladas.

Al convertir las direcciones en valores tipo ***long***, ya no estamos calculando cuántas variables hay entre ambas direcciones, sino la diferencia entre el valor que codifica la última posición del vector y el valor que codifica la primera dirección. Y en ese caso, el valor será mayor según sea mayor el número de bytes que emplee el tipo de dato referenciado por el puntero. Si es un ***char***, entre la posición última y la primera hay, efectivamente, 9 elementos; y el número de bytes entre esas dos direcciones también es 9. Si es un ***float***, entre la posición última y la primera hay, efectivamente y de nuevo, 9 elementos; pero ahora el número de bytes entre esas dos direcciones es 36, porque cada uno de los nueve elementos ocupa cuatro bytes de memoria.

Índices y operatoria de punteros

Se puede recorrer un vector, o una cadena de caracteres mediante índices. Y también, de forma equivalente, mediante operatoria de punteros.

Pero además, los arrays y cadenas tienen la siguiente propiedad: Si declaramos ese array o cadena de la siguiente forma:

tipo nombre_array[dimensión];

El nombre del vector o cadena es *nombre_array*. Para hacer uso de cada una de las variables, se utiliza el nombre del vector o cadena seguido, entre corchetes, del índice del elemento al que se quiere hacer referencia: *nombre_array[índice]*.

Y ahora introducimos otra novedad: el nombre del vector o cadena recoge la dirección de la cadena, es decir, la dirección del primer elemento de la cadena: decir *nombre_array* es lo mismo que decir *&nombre_array[0]*.

Y por tanto, y volviendo al código anteriormente visto:

```
long a[10], *pa;
short b[10], *pb;
pa = &a[0];
pb = &b[0];
```

Tenemos que **(pa + i)* es lo mismo que *a[i]*. Y como decir *a* es equivalente a decir *&a[0]* entonces, decir *pa = &a[0]* es lo mismo que decir *pa = a*, y trabajar con el valor **(pa + i)* es lo mismo que trabajar con el valor **(a + i)*.

Y si podemos considerar que dar el nombre de un vector es equivalente a dar la dirección del primer elemento, entonces podemos considerar que ese nombre funciona como un puntero constante, con quien se pueden hacer operaciones y formar parte de expresiones, mientras no se le coloque en la parte *Lvalue* de un operador asignación.

Y muchos programadores, en lugar de trabajar con índices, recorren todos sus vectores y cadenas mediante la operatoria o aritmética de punteros.

Veamos un programa sencillo, resuelto mediante índices de vectores y mediante la operatoria de punteros. Por ejemplo, un programa que solicite al usuario una cadena de caracteres y luego la copie en otra

cadena en orden inverso: primero el último carácter, luego el penúltimo, etc.

Con índices:

```
#include <stdio.h>
#include <string.h>
void main(void)
{
      char orig[100], copia[100];
      short i, l;
      printf("Introduzca la cadena ... \n");
      gets(orig);
      l = strlen(orig);
      for(i = 0 ; i < l ; i++)
            copia[l - i - 1] = orig[i];
      copia[i] = NULL;
      printf("Cadena original: %s\n",orig);
      printf("Cadena copia: %s\n",copia);
}
```

Con operatoria de punteros:

```
#include <stdio.h>
#include <string.h>
void main(void)
{
      char orig[100], copia[100];
      short i, l;

      printf("Introduzca la cadena ... \n");
      gets(orig);
      l = strlen(orig);
      for(i = 0 ; i < l ; i++)
            *(copia + l - i - 1) = *(orig + i);
      *(copia + i) = NULL;
      printf("Cadena original: %s\n",orig);
      printf("Cadena copia: %s\n",copia);
}
```

Desde luego, ambas formas de referirse a los distintos elementos del vector son válidas.

En el capítulo en que hemos presentado los arrays hemos indicado que es competencia del programador no recorrer el vector más allá de las posiciones reservadas. Si se llega, mediante operatoria de índices o mediante operatoria de punteros a una posición de memoria que no pertenece realmente al vector, el compilador no detectará error alguno,

e incluso puede que tampoco se produzca un error en tiempo de ejecución, pero estaremos accediendo a zona de memoria que quizá se emplea para almacenar otra información. Y entonces alteraremos esos datos de forma inconsiderada, con las consecuencias desastrosas que eso pueda llegar a tener para el buen fin del proceso. Cuando en un programa se llega equivocadamente, mediante operatoria de punteros o de índices, más allá de la zona de memoria reservada, se dice que se ha producido o se ha incurrido en una **violación de memoria**.

Puntero a puntero

Un puntero es una variable que contiene la dirección de otra variable. Según sea el tipo de variable que va a ser apuntada, así, de ese tipo, debe ser declarado el puntero. Ya lo hemos dicho.

Pero un puntero es también una variable. Y como variable que es, ocupa una porción de memoria: tiene una dirección.

Se puede, por tanto, crear una variable que almacene la dirección de esa variable puntero. Sería un puntero que almacenaría direcciones de tipo de dato puntero. Un puntero a puntero.

Por ejemplo:

float *F, *pF, **ppF;*

Acabamos de crear tres variables: una, de tipo ***float***, llamada *F*. Una segunda variable, de tipo *puntero a* ***float***, llamada *pF*. Y una tercera variable, de tipo *puntero a puntero* ***float***, llamada *ppF*.

Y eso no es un rizo absurdo. Tiene mucha aplicación en C. Igual que se puede hablar de un *puntero a puntero a puntero… a puntero a* ***float***.

Y así como antes hemos visto que hay una relación directa entre punteros a un tipo de dato y vectores de este tipo de dato, también veremos ahora que hay una relación directa entre punteros a punteros y matrices de dimensión 2. Y entre punteros a punteros a punteros y

matrices de dimensión 3. Y si es conveniente trabajar con matrices de dimensión *n* , entonces también lo es trabajar con punteros a punteros a punteros…

Veámoslo con un ejemplo. Supongamos que creamos una matriz de dimensión 2:

double *m[4][6];*

Antes hemos dicho que al crea un array, al hacer referencia a su nombre estamos indicando la dirección del primero de sus elementos. Ahora, al crear esta matriz, la dirección del elemento *m[0][0]* la obtenemos con el nombre de la matriz: Es equivalente decir *m* que decir *&m[0][0]*.

Pero la estructura que se crea al declarar una matriz es algo más compleja que una lista de posiciones de memoria. En el ejemplo expuesto de la matriz ***double***, se puede considerar que se han creado cuatro vectores de seis elementos cada uno y colocados en la memoria uno detrás del otro de forma consecutiva. Y cada uno de esos vectores tiene, como todo vector, la posibilidad de ofrecer la dirección de su primer elemento. El cuadro 8.1. presenta un esquema de esta construcción. Desde luego, no existen los punteros m, ni ninguno de los **(m + i)*. Pero si empleamos el nombre de la matriz de esta forma, entonces trabajamos con sintaxis de punteros.

De hecho, si ejecutamos el siguiente programa:

```
#include <stdio.h>
void main(void)
{
      double m[4][6];
      short i;
      printf("m = %p\n",m);
      for(i = 0 ; i < 4 ; i++)
      {
            printf("*(m + %hd) = %p\t",i, *(m + i));
            printf("&m[%hd][0] = %p\n",i, &m[i][0]);
      }
}
```

Obtenemos la siguiente salida:

```
m = 0012FECC
*(m + 0) = 0012FECC      &m[0][0] = 0012FECC
*(m + 1) = 0012FEFC      &m[1][0] = 0012FEFC
*(m + 2) = 0012FF2C      &m[2][0] = 0012FF2C
*(m + 3) = 0012FF5C      &m[3][0] = 0012FF5C
```

m	➔							
		*(m+0)	➔	m[0][0]	m[0][1]	m[0][2]	m[0][3]	m[0][4]	m[0][5]
		*(m+1)	➔	m[1][0]	m[1][1]	m[1][2]	m[1][3]	m[1][4]	m[1][5]
		*(m+2)	➔	m[2][0]	m[2][1]	m[2][2]	m[2][3]	m[2][4]	m[2][5]
		*(m+3)	➔	m[3][0]	m[3][1]	m[3][2]	m[3][3]	m[3][4]	m[3][5]

Cuadro 8.1.: Distribución de la memoria en la matriz ***double*** *m[4][6];*

Tenemos que *m* vale lo mismo que **(m + 0)*; su valor es la dirección del primer elemento de la matriz: *m[0][0]*. Después de él, vienen todos los demás, uno detrás de otro: después de *m[0][5]* vendrá el *m[1][0]*, y esa dirección la podemos obtener con **(m + 1)*; después de *m[1][5]* vendrá el *m[2][0]*, y esa dirección la podemos obtener con **(m + 2)*; después de *m[2][5]* vendrá el *m[3][0]*, y esa dirección la podemos obtener con **(m + 3)*; y después de *m[3][5]* se termina la cadena de elementos reservados.

Es decir, en la memoria del ordenador, no se distingue entre un vector de 24 variables tipo ***double*** y una matriz 4 * 6 de variables tipo ***double***. Es el lenguaje el que sabe interpretar, mediante una operatoria de punteros, una estructura matricial donde sólo se dispone de una secuencia lineal de elementos. Esos punteros no existen en realidad, y si se imprime la posición que ocupa *m*, ó **m* la pantalla nos mostrará el mismo valor que al solicitarle que nos muestre la dirección de *m[0][0]*. No existen, pero el lenguaje C admite esa sintaxis, y podemos trabajar como si de punteros constantes se trataran.

Y así como antes hemos podido trabajar un programa con un array mediante operatoria de punteros, ahora vamos a hacer lo mismo con un programa que emplee matrices.

Veamos un programa que calcula el determinante de una matriz de tres por tres.

Con índices:

```
#include <stdio.h>
void main(void)
{
      double m[3][3];
      double det;
      short i,j;
      for(i = 0 ; i < 3 ; i++)
            for(j = 0 ; j < 3 ; j++)
            {
                  printf("m[%hd][%hd] = ", i, j);
                  scanf("%lf",&m[i][j]);
            }
      det = 0;
      det += (m[0][0] * m[1][1] * m[2][2]);
      det += (m[0][1] * m[1][2] * m[2][0]);
      det += (m[0][2] * m[1][0] * m[2][1]);
      det -= (m[0][2] * m[1][1] * m[2][0]);
      det -= (m[0][1] * m[1][0] * m[2][2]);
      det -= (m[0][0] * m[1][2] * m[2][1]);
      printf("El determinante ... \n");
      for(i = 0 ; i < 3 ; i++)
      {
            printf("\n | ");
            for(j = 0 ; j < 3 ; j++)
                  printf("%8.2lf",m[i][j]);
            printf(" | ");
      }
      printf("\n\n es ... %lf",det);
}
```

Con operatoria de punteros:

```
#include <stdio.h>
void main(void)
{
double m[3][3];
double det;
short i,j;
for(i = 0 ; i < 3 ; i++)
      for(j = 0 ; j < 3 ; j++)
      {
```

```
            printf("m[%hd][%hd] = ", i, j);
            scanf("%lf",*(m + i) + j);
      }
det = 0;
det += *(*(m + 0) + 0) * *(*(m + 1) + 1) * *(*(m + 2) + 2);
det += *(*(m + 0) + 1) * *(*(m + 1) + 2) * *(*(m + 2) + 0);
det += *(*(m + 0) + 2) * *(*(m + 1) + 0) * *(*(m + 2) + 1);
det -= *(*(m + 0) + 2) * *(*(m + 1) + 1) * *(*(m + 2) + 0);
det -= *(*(m + 0) + 1) * *(*(m + 1) + 0) * *(*(m + 2) + 2);
det -= *(*(m + 0) + 0) * *(*(m + 1) + 2) * *(*(m + 2) + 1);
printf("El determinante ... \n");
for(i = 0 ; i < 3 ; i++)
{
      printf("\n | ");
      for(j = 0 ; j < 3 ; j++)
            printf("%8.2lf",*(*(m + i) + j));
      printf(" | ");
}
printf("\n\n es ... %lf",det);
```

Desde luego, la operatoria de punteros con matrices resulta algo farragosa en un primer momento. Pero no encierra dificultad de concepto.

Advertencia final

El uso de puntero condiciona el modo de programar. El puntero es una herramienta muy poderosa y muy arriesgada también. Es necesario saber bien qué se hace, porque jugando con punteros se pueden "burlar" muchas seguridades de C.

Veamos un ejemplo. Primero presentamos el siguiente código, donde se emplea una variable ***static***: una variable que se extiende a la duración de todo el programa, pero cuyo ámbito queda reducido al del bloque en el que se ha definido. Esa variable, que en el ejemplo llamamos *local*, se inicializa a cero la primera vez que se entra en el bloque donde está definida, y posteriormente se va aumentando de uno en uno cada vez que se ejecuta su bloque.

```
#include <stdio.h>
void main(void)
{
      short a, b = 0;
```

```
      do
      {
            a = 0;
            do
            {
                  static short local = 0;
                  local++;
                  printf("local = %2hd\n",local);
                  a++;
            }while (a < 5);
            printf("\n");
            b++;
      }while(b < 5);
}
```

La salida de este programa es la que sigue:

```
local =  1  local =  2  local =  3  local =  4  local =  5
local =  6  local =  7  local =  8  local =  9  local = 10
local = 11  local = 12  local = 13  local = 14  local = 15
local = 16  local = 17  local = 18  local = 19  local = 20
local = 21  local = 22  local = 23  local = 24  local = 25
```

Cada vez que se entra en el ámbito de la variable *local*, se vuelve a poder trabajar sobre ella y se sigue incrementando desde el valor en que quedo después de la última modificación.

Pero si ahora introducimos en el programa los siguientes cambios:

```
#include <stdio.h>
void main(void)
{
      short a, b = 0;
      short *c;
      do
      {
            a = 0;
            do
            {
                  static short local = 0;
                  c = &local;
                  local++;
                  printf("local = %hd\t",local);
                  a++;
            }while (a < 5);
            printf("\n");
            b++;
            *c = 0;
      }while(b < 5);
```

Entonces la salida es:

```
local = 1   local = 2   local = 3   local = 4   local = 5
local = 1   local = 2   local = 3   local = 4   local = 5
local = 1   local = 2   local = 3   local = 4   local = 5
local = 1   local = 2   local = 3   local = 4   local = 5
local = 1   local = 2   local = 3   local = 4   local = 5
```

Y es que con el puntero, y desde fuera del ámbito de la variable *local*, le hemos cambiado su valor y la hemos inicializado a cero una y otra vez. Y eso es peligroso, porque podemos violar las reglas de validez de las variables. Es mejor evitar operaciones de este estilo. No se debe jugar en contra de las reglas de la sintaxis del lenguaje C. Es mejor programar de acuerdo con esas reglas.

Ejercicios

52. ***Leer el siguiente código y completar la salida que ofrece por pantalla:*** *(No está resuelto)***.**

```
#include <stdio.h>
void main(void)
{
      char c[3];
      short i[3], cont;
      float f[3];
      printf("Las direcciones de memoria son:\n");
      for(cont = 0 ; cont < 3 ; cont++)
      {
            printf("&c[%2d] = %10p\t",cont,c + cont);
            printf("&i[%2d] = %10p\t",cont,i + cont);
            printf("&f[%2d] = %10p\n\n",cont,f + cont);
      }
}
```

```
Las direcciones de memoria son:

&c[ 0] = 0064FE01     &i[ 0] = 0064FDF8     &f[ 0] = 0064FDEC

&c[ 1] =              &i[ 1] =              &f[ 1] =

&c[ 2] =              &i[ 2] =              &f[ 2] =
```

53. ***Leer el siguiente código y completar la salida que ofrece por pantalla*.**

```
#include <stdio.h>
main()
{
      char c[20],*pc1,*pc2;
      short int i[20],*pi1,*pi2;
      float f[20],*pf1,*pf2;

      pc1 = &c[0];
      pc2 = &c[19];
      pi1 = &i[0];
      pi2 = &i[19];
      pf1 = &f[0];
      pf2 = &f[19];

      printf("(int)pc2-(int)pc1 es %d\n",(int)pc2-(int)pc1);
      printf("pc2 - pc1 es %d\n\n",pc2 - pc1);

      printf("(int)pi2-(int)pi1 es %d\n",(int)pi2-(int)pi1);
      printf("pi2 - pi1 es %d\n\n",pi2 - pi1);

      printf("(int)pf2-(int)pf1 es %d\n",(int)pf2-(int)pf1);
      printf("pf2 - pf1 es %d\n\n",pf2 - pf1);
}
```

```
(int)pc2 - (int)pc1 es 19
pc2 - pc1 es 19
(int)pi2 - (int)pi1 es 38
pi2 - pi1 es 19
(int)pf2 - (int)pf1 es 76
pf2 - pf1 es 19
```

En general una buena forma de aprendere a manejar punteros es intentar rehacer todos los ejercicios ya resueltos en los dos capítulos anteriores empleando ahora operatoria de punteros y recorriendo los vectores y matrices mediante la indirección.