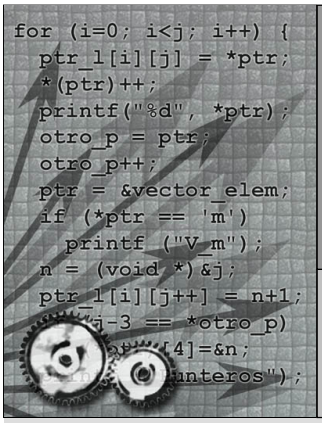

Punteros en C

Escuela Superior de Informática - Ciudad Real
Universidad de Castilla-La Mancha

Carlos González Morcillo ::: Carlos.Gonzalez@uclm.es
Miguel Ángel Redondo Duque ::: Miguel.Redondo@uclm.es
http://www.inf-cr.uclm.es/www/cglez
http://www.inf-cr.uclm.es/www/mredondo

© 2003 Carlos González Morcillo - Miguel Ángel Redondo Duque. Se permite la copia, distribución y/o modificación de este documento bajo los términos de la licencia de documentación libre GNU, versión 1.1 o cualquier versión posterior publicada por la *Free Software Foundation*; sin secciones invariantes. Puede consultar esta licencia en http://www.gnu.org

Tabla de Contenidos

Introducción

«Díjole el perro al hueso: Si tú eres duro, yo tengo tiempo...» Refranero Popular

Suele resultar difícil dominar el concepto de puntero. Especialmente a los programadores que aprendieron con lenguajes de alto nivel, donde no se permite trabajar directamente con posiciones de memoria. El mayor problema en C surge al tratar de detectar esos «*segmentation fault*» maquiavélicos que a veces aparecen, a veces no.

Este documento pretende llenar esta laguna con un texto breve, que introduce los fundamentos de los punteros con numerosos ejemplos. No es posible aprender a programar únicamente leyendo un documento, así que aplicando el «*Caminante no hay camino, se hace camino al andar*», utiliza un compilador ANSI C y compila, prueba y cambia los ejemplos de esta guía. Si únicamente lees el texto, no cogerás manejo y tu lectura valdrá para poco.

Miles de agradecimientos a nuestros compañeros Miguel Ángel Laguna, Ramón Manjavacas y David Villa por aportar ideas y depurar este documento.

Será bienvenida cualquier indicación acerca de este texto, errores encontrados, dudas y aclaraciones en nuestra cuenta de e-mail.

¿Qué es un puntero?

En el famoso libro de Kernighan y Ritchie "El lenguaje de programación C", se define un puntero como «*una variable que contiene la dirección de una variable*». Tras esta definición clara y precisa, podemos remarcar que un puntero es un tipo especial de variable que almacena el valor de una dirección de memoria.

La memoria de los ordenadores está formada por un conjunto de bytes. Simplificando, cada byte tiene un número asociado; una dirección en esa memoria.

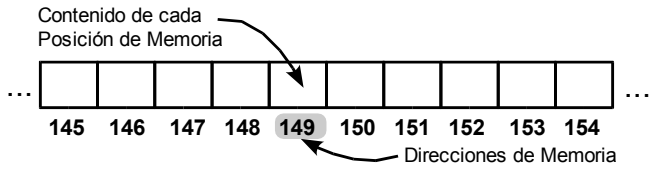

Figura 1. Representación de la Memoria

La representación anterior se utilizará a lo largo del texto para ilustrar los distintos ejemplos. De esta forma, si ejecutamos el ejemplo del Listado 1:

```
01  #include <stdio.h>
02  int main(void)
03  {
04    float pi=3.14;
05    printf ("%.2f\n", pi);
06    return 0;
07  }
```

Listado 1

Obtenemos en pantalla el esperado *3.14*. Sin embargo, ¿qué operaciones ha hecho el ordenador hasta mostrar el resultado?. En la línea 4, la instrucción reserva memoria para la variable *pi*. En este ejemplo, asumimos que un decimal de tipo *float* ocupa 4 bytes. Dependiendo de la arquitectura del computador, la cantidad de bytes requerida para cada tipo de datos puede variar.

La Figura 2 muestra el estado de la memoria después de la declaración de la variable *pi*. La representación elegida facilita el seguimiento de los ejemplos, sin embargo, la representación real en la memoria de ordenador no tendría esa disposición.

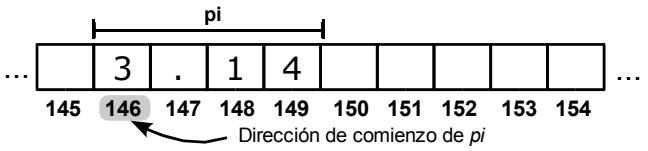

Figura 2. Memoria reservada para una variable

Cuando se utiliza *pi* en la línea 5, ocurren dos pasos diferenciables. En primer lugar, el programa busca la dirección de memoria reservada para *pi* (en nuestro ejemplo, sería la dirección 146). Hecho esto, en segundo lugar, el programa recupera el contenido de esa dirección de memoria.

Así, distinguimos por un lado la dirección de memoria asignada a una variable, y por otro lado el contenido de la posición de memoria reservada. Podemos acceder a la dirección de una variable utilizando el operador &. Mediante este operador, obtenemos la dirección donde se almacena la variable (¡un número!). En el ejemplo del Listado 2, se ha utilizado `%u` y una conversión forzada a *entero sin signo* para que la salida por pantalla sea más entendible. Se podía haber utilizado el conversor estándar para punteros `%p` y haber quitado la conversión forzada a *entero sin signo*.

```
01  #include <stdio.h>
02  int main(void)
03  {
04    float pi=3.14;
05    printf ("Dirección de pi: %u\n",
              (unsigned int)&pi);
06    return 0;
07  }
```

Listado 2

Si ejecutáramos el programa del listado 2 en la memoria del ejemplo anterior, debería mostrarnos la dirección *146*. Recordemos que una dirección de memoria es únicamente un número. De hecho, podríamos almacenarla en una variable entera sin signo (ver Listado 3).

```
01  #include <stdio.h>
02  int main(void)
03  {
04    float pi=3.14;
05    unsigned int dir=(unsigned int)&pi;
06    printf ("Direcc. de pi: %u\n", dir);
07    return 0;
08  }
```

Listado 3

El código anterior demuestra que no hay nada de magia ni efectos especiales en el tema de las direcciones. Simplemente son números que pueden ser almacenados en variables enteras[1]. Esto se puede ver gráficamente en la siguiente figura (suponiendo que las variables de tipo entero sin signo requieran 3 bytes en memoria).

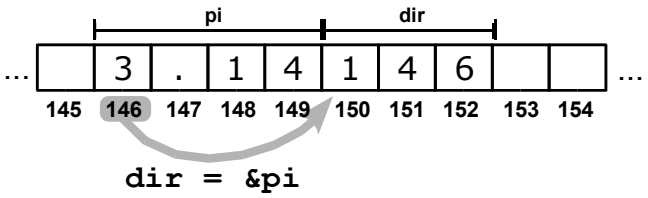

Figura 3. Uso del operador '&'

Lo que se quiere representar simplemente es que mediante el operador & accedemos a la dirección de memoria de una variable. Esta dirección es un simple número entero con el que podemos trabajar directamente.

Para diferenciar una variable entera que almacene valores de nuestro programa (por ejemplo, índices, contadores, valores...) de las variables que almacenan direcciones, en C se creó un nuevo tipo de variable para almacenar direcciones. Este tipo es, precisamente, el puntero. Así, podemos tener algunas declaraciones de punteros como las siguientes:

```
char * nombre;
float * altura;
```

¿Cómo interpretar las líneas anteriores? Podemos leer que `nombre` es un entero, pero podemos añadir que es un entero especialmente diseñado para guardar la dirección de un carácter. Por su parte, `altura` es un entero, además es un entero especialmente diseñado para guardar la dirección de un float. Es lo que decimos resumidamente con "*nombre es un puntero a char*" o "*altura es un puntero a float*".

El operador * también se utiliza para recuperar el contenido de una posición de memoria identificada por una dirección. No debemos confundir su uso en la declaración de un puntero, como hemos visto antes, con los casos en que se emplea para recuperar el contenido de una posición de memoria. Este uso lo veremos a continuación.

¿Para qué usar punteros?

Hagamos un programa que cambie el valor de la variable `pi` del ejemplo anterior a cero. Para ello, realizaremos el código que se muestra en el Listado 4. En este caso intentamos pasar el parámetro `vpi` por referencia, para que cambie el contenido de `pi` dentro de la función. Si ejecutamos el programa, el resultado obtenido no es el esperado. ¿Por qué?

```
01  void cambiavalor(float vpi) {
02    vpi = 0;
03    printf ("Dirección de vpi: %u\n",
              (unsigned int) &vpi);
04    printf ("Valor de vpi: %f\n", vpi);
05  }
06  int main(void){
07    float pi=3.14;
08    cambiavalor(pi);
09    printf ("Dirección de pi: %u\n",
              (unsigned int) &pi);
10    printf ("Valor de pi: %f\n", pi);
11    return 0;
14  }
```

Listado 4

[1] Recordemos que el modificador unsigned simplemente indica que no se permiten números negativos. Las direcciones no pueden ser negativas.

Carlos González Morcillo (*Carlos.Gonzalez@uclm.es*) ::: Miguel Ángel Redondo Duque (*Miguel.Redondo@uclm.es*)

La ejecución del programa muestra que el parámetro `pi` ha sido pasado por valor (en lugar de por referencia). Si hacemos un seguimiento más exhaustivo del programa, vemos que cuando llamamos a la función `cambiavalor` en la línea 8, el programa salta a la línea 1 para ejecutar la función. Allí, se crea una nueva variable llamada `vpi`, y se copia el valor de `pi` en ella. Esto se representa en el esquema de la Figura 4.

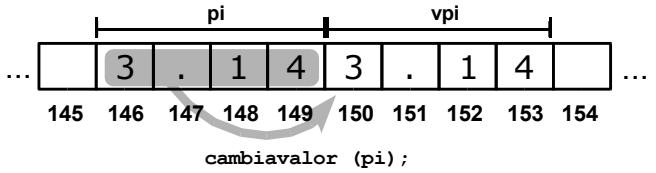

Figura 4. Copia del valor de pi en vpi

Queda clara la razón por la que `vpi` y `pi` tienen diferente dirección en memoria; se ha realizado una copia del contenido de la memoria que tiene `pi`, se ha llevado a otra zona de memoria y se le ha puesto de nombre `vpi`. De esta forma, cuando se asigna 0 a la variable `vpi` en la línea 2, se está asignando al contenido de memoria de `vpi`, no al de `pi` (ver Figura 5).

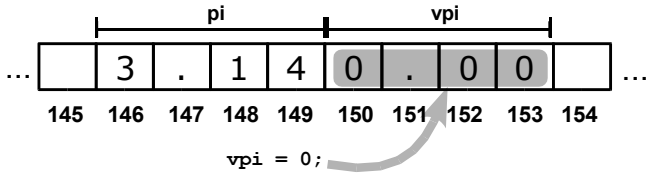

Figura 5. Cambio de valor de vpi

Los argumentos de las funciones en C se pasan siempre por valor. No existe una forma directa para que la función que se invoca altere una variable de la función que la llama. Es imprescindible el uso de punteros en el caso de querer modificar el valor de una variable pasada como parámetro a una función.

Así, podemos modificar el programa del Listado 4, para que el paso del argumento se realice por referencia. Tendremos que utilizar los operadores * y & que hemos visto anteriormente. El nuevo programa se muestra en el Listado 5.

```
00  #include <stdio.h>
01  void cambiavalor(float *vpi) {
02    *vpi = 0;
03    printf ("Dir. vpi: %u\n",
                (unsigned int) vpi);
04    printf ("Val. vpi: %f\n", *vpi);
05  }
06  int main(void) {
07    float pi=3.14;
08    cambiavalor(&pi);
09    printf ("Dir. pi: %u\n",
                (unsigned int)&pi);
10    printf ("Val. pi: %f\n", pi);
11    return 0;
12  }
```

Listado 5

En esta nueva versión del programa, la función `cambiavalor`, recibe el parámetro como un puntero (línea 1). Le estamos diciendo al compilador que pase como argumento a la función una dirección de memoria. Es decir, queremos que le pase un entero; ese entero será una dirección de memoria de un float.

En un seguimiento en detalle del programa vemos que, en la línea 8, cuando llamamos a la función `cambiavalor`, le tenemos que pasar una dirección de memoria. Para ello, tenemos que hacer uso del operador &, para pasarle la dirección de la variable `pi`. Con esto, cuando llamamos a la función `cambiavalor` en 1, lo que se hace es una copia de la dirección de memoria de `pi` en la variable `vpi` (ver Figura 6). Así, mediante el operador * podremos acceder al contenido de la posición de memoria de `pi` y cambiar su valor.

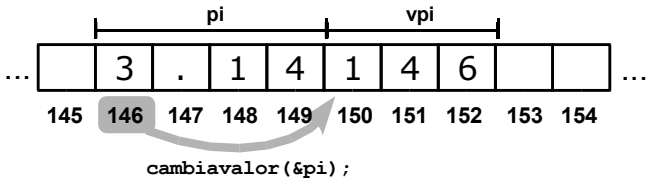

Figura 6. Paso por referencia de "pi"

En la línea 2 del programa se cambia el valor de la zona de memoria apuntada por `vpi`. Podemos leer esa instrucción como "*asignamos cero al contenido de la dirección de memoria apuntada por vpi*". Recordemos que esa zona de memoria tiene que ser de tipo float. En el ejemplo de la Figura 6, esta dirección de memoria es la 146. Podemos ver un esquema del resultado de la operación en la Figura 7.

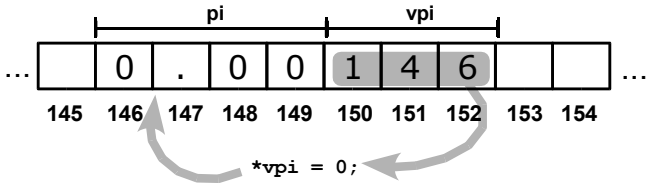

Figura 7. Asignación de 0 al contenido de memoria apuntado por vpi

Ahora, cuando acabe la función `cambiavalor` y retornemos a la función principal `main`, el valor de `pi` será 0. Hemos realizado un paso de parámetro por referencia.

A continuación examinaremos en profundidad algunos tipos de datos complejos (como vectores y estructuras) y su utilización con punteros, así como algunos operadores nuevos específicos de punteros.

Aritmética de Punteros

Como hemos mencionado anteriormente, podemos considerar el tipo de un puntero como el tipo de datos que contiene la zona de memoria a la que éste apunta. Es decir, si tenemos:

```
int *ptr;
*ptr = 2;
```

la primera línea indica que tenemos una variable llamada `ptr`, cuyo contenido es una dirección de memoria a una zona donde hay un entero. En la segunda línea asignamos al contenido de esa dirección de memoria el valor 2.

Una operación interesante que podemos realizar con punteros es incrementarlos. Obviamente, la operación de incremento en un puntero no es igual que en una variable normal. Un incremento en un puntero hará

que apunte al siguiente elemento del mismo tipo de datos que se encuentre en memoria.

En el ejemplo de la Figura 8, inicialmente el puntero `ptr` apunta al tercer entero almacenado en el bloque de memoria representado (suponiendo que cada entero ocupa 3 posiciones de memoria). Así, la asignación `*ptr=168` hará que el contenido de esa posición de memoria sea 168. Si incrementamos el puntero (¡cuidado!, no hablamos del contenido del puntero, sino del propio puntero), haremos que apunte al siguiente entero que hay en memoria. Este incremento del puntero no implica sumar 1 a la dirección que contiene el puntero. En este ejemplo, sería necesario sumar 3 (tres posiciones de memoria que ocupa el entero para acceder al siguiente).

Naturalmente esta "*suma de posiciones de memoria*" no la tendrá que hacer el programador. Será el compilador el que se encargue de realizar las cuentas para ver cuántas posiciones de memoria debe avanzar. En general, el compilador tendrá que sumar tantas posiciones de memoria como requiera el tipo de datos sobre el que se haya definido el puntero.

Figura 8. Incremento de punteros

En el caso de que incrementemos alegremente el puntero y nos salgamos de la zona de memoria que tengamos reservada, recibiremos la visita de nuestro amigo «*segmentation fault*».

Vamos a ilustrar el uso de punteros con vectores mediante un ejemplo. Supongamos que definimos un array de enteros (que, a fin de cuentas, es un conjunto de enteros a los que se reservan posiciones de memoria consecutivas) como se muestra en la línea 2 del Listado 6.

```
01  #include <stdio.h>

02  int vector[] = {1, 56, 47, -89, 10};
03  int *ptr;

04  int main(void)
05  {
06    int i;
07    ptr = &vector[0];
08    for (i=0; i<5; i++) {
09      printf ("Vect[%d] = %d\n", i, vector[i]);
10      printf ("Ptr + %d = %d\n", i, *(ptr+i));
11    }
12    return 0;
13  }
```

Listado 6

Tenemos un vector de 5 enteros. Podemos acceder a ellos mediante el índice que ocupan en el array (como por ejemplo, `vector[3]` nos daría el cuarto entero del array). De igual forma podemos acceder a él mediante un puntero. En la línea 7, asignamos al puntero la dirección del primer elemento del vector. En C, el nombre del array apunta al primer elemento del array; por lo que podríamos cambiar la línea 7 por la instrucción `ptr=vector;` con idénticos resultados.

De cualquier forma, debemos tener muy claro que, una vez definido un array, el nombre del array identifica a la posición del primer elemento, pero es una constante. Es decir, podemos utilizarlo como hemos indicado antes para asignárselo a un puntero (copiamos la dirección de memoria del primer elemento), pero no podemos asignarle a un array ya creado una dirección distinta; `vector = otro_puntero` provocaría un error. Algunos programadores llaman al nombre de un array como "*puntero constante*", indicando que no puede cambiarse su valor. Es simplemente un puntero al que no podemos cambiar su dirección.

Cadenas de caracteres

Recordemos que en C, las cadenas de caracteres son simplemente arrays de caracteres. No hay un tipo de datos especial, como en Basic o Pascal. Una cadena en C terminará siempre con un carácter especial reservado para indicar "*fin de cadena*", que es un cero binario, representado por `'\0'`.

Vamos a implementar nuestra función de copia de cadenas, similar a la de ANSI C: `strcpy()`.

```
01  #include <stdio.h>

02  char origen[40] = "Apuntes de Punteros";
03  char destino[40];

04  char *mi_strcpy(char *dest, const char *orig)
05  {
06    char *p = dest;
07    while (*orig != '\0')
08      *p++ = *orig++;
09    *p = '\0';
10    return dest;
11  }

12  int main(void)
13  {
14    int i;

15    mi_strcpy(destino, origen);
16    printf ("%s", destino);
17    return 0;
18  }
```

Listado 7

En el listado anterior, el espacio necesario para las dos cadenas se reserva en `origen` y `destino`. Cuando llamamos a la función `mi_strcpy`, en la línea 15 lo único que se pasa es la dirección de memoria de ambas cadenas. Este planteamiento nos permite trabajar con arrays de gran tamaño sin tener que mover gran cantidad de información en memoria. Como las cadenas de caracteres son exactamente iguales que los arrays de cualquier otro tipo de datos, el enfoque utilizado es el mismo en todos los casos.

Recordemos que (ver Figura 6), lo que pasamos como argumento a la función es una copia de la dirección de memoria. Así, si cambiamos el puntero dentro de la función `mi_strcpy`, a la vuelta de la función los punteros originales seguirán apuntando a donde estaban. Esto se puede ver en el ejemplo anterior; cuando llamamos a la función en la línea 15, los punteros `origen` y `destino` apuntan al comienzo de las cadenas. Dentro de la función `mi_strcpy`, los

punteros van avanzando en la cadena (línea 8), y al final ambos están situados al final de las cadenas. Al ser copias, cuando volvemos de la función, los punteros *origen* y *destino* siguen apuntando al inicio de la cadena.

El modificador `const` usado en la línea 4 hace que, dentro de la función, no podamos modificar el contenido de la memoria apuntada por `orig`, tomando el contenido como valor constante.

En la línea 6, hacemos que un puntero auxiliar apunte al inicio de la cadena destino (`dest`). Después, mientras el carácter apuntado por el puntero origen (`orig`) sea distinto del fin de cadena, copiamos el carácter en la zona de memoria apuntada por `p` y avanzamos. Esto lo realizamos con una única instrucción (línea 8). ¿Es equivalente `*p++` a `(*p)++`?. No, según la precedencia de los operadores, `*p++` es equivalente a `*(p++)`. Y, ¿qué decimos en cada caso?. Pues con la primera instrucción, `*p++`, decimos que primero veamos el contenido del puntero y después incrementemos la posición del puntero (utilizado en la función de copia de arrays), mientras que con `(*p)++` decimos que incremente el contenido de la posición de memoria donde apunta `p`. Si `p` apuntara a un carácter "*C*", haría que ese carácter pasara a ser "*D*". Si `p` apunta a un entero cuyo valor es *24*, después de esa instrucción sería *25*.

Queda claro, por tanto, la necesidad de establecer correctamente el orden de aplicación entre operadores. Personalmente creo una pérdida de tiempo tratar de recordar la precedencia de los operadores. Una buena práctica que mantiene el código más legible es utilizar paréntesis siempre que haya dudas. Naturalmente, siempre dentro de unos límites razonables.

¿Es necesario el puntero auxiliar `p`? Claramente no, se ha utilizado únicamente para dejar bien claro que trabajamos con direcciones de memoria, y que `*p` y `*dest` es exactamente lo mismo. Por eso, al retornar de la función, devolvemos `dest`. El código de la función `mi_strcpy` del Listado 7 podía simplificarse como muestra el Listado 8.

```
01  while (*orig != '\0')
02    *(dest++) = *(orig++);
03  *dest = '\0';
04  return dest;
```

Listado 8

En esta última versión se ha hecho uso de paréntesis para indicar de forma clara, que queremos copiar el contenido de la memoria apuntada por `orig` en `dest` y, después, incrementar los punteros. Finalmente hay que añadir la constante '*\0*' en la última posición de `dest` (para que cumpla el convenio utilizado con cadenas de caracteres en ANSI C). Incluso podríamos optimizar algo más el código poniendo en la definición del bucle `while(*orig)`, ya que será cierto mientras el valor apuntado por `orig` sea distinto de cero.

Finalmente recordemos que podemos utilizar también la notación de array para trabajar con cadenas de caracteres. Así, tendríamos las expresiones `dest[i]` y `*(dest + i)` como equivalentes.

Estructuras

Una estructura es un bloque que contiene una secuencia de variables de diferentes tipos de datos. Vamos a realizar un programa que muestre por pantalla un elemento de tipo estructura.

```
01  #include <stdio.h>
02  #include <string.h>

03  struct alumno{
04    char nombre[20];
05    char apell[40];
06    int edad;
07    float nota;
08 };

09  void ver_alumno (struct alumno *a) {
10    printf ("%s %s ", a->nombre, a->apell);
11    printf ("Calificacion: %f\n", a->nota);
12  }

13  int main(void)
14  {
15    struct alumno alum;
16    strcpy (alum.nombre, "Perico");
17    strcpy (alum.apell, "Palotes");
18    alum.edad = 22;
19    alum.nota = 8.75;
20    ver_alumno (&alum);

21    return 0;
22  }
```

Listado 9

Un error típico es olvidar el punto y coma al terminar la definición de una estructura (línea 8). Después de declarar una variable del tipo estructura `alumno`, dentro de `main`, podemos acceder a sus campos con el operador punto siempre que estemos trabajando con la estructura directamente. Cuando estemos utilizando un puntero a estructura, utilizaremos el operador flecha (`->`). Este operador es equivalente a utilizar `(*puntero_estructura).campo`

Así, en la línea 11 podíamos haber accedido al campo `nota` del puntero a estructura "`a`", mediante la instrucción `(*a).nota`. Sin embargo, es más cómodo utilizar la notación flecha para estos casos.

ANSI C permite pasar por valor las estructuras completas. Sin embargo, esto implica que se copia el contenido de toda la estructura (completa) desde la función llamante a la función llamada. Esto, para estructuras con un gran número de elementos, puede ser un problema. Por cuestiones de eficiencia y comodidad, pasaremos las estructuras a las funciones mediante un puntero.

Punteros a Matrices

Anteriormente hemos visto la relación entre punteros y arrays; vamos a ampliar nuestro campo de visión utilizando arrays de varias dimensiones (matrices).

```
01  #include <stdio.h>
02  #define NFILAS 5
03  #define NCOLUMNAS 8

04  int main(void)
05  {
06    int fil, col;
07    int matriz[NFILAS][NCOLUMNAS];

08    /* Asignamos valores a los elementos */
```

```
09    for (fil=0; fil<NFILAS; fil++)
10      for (col=0; col<NCOLUMNAS; col++)
11        matriz[fil][col] = fil*col;

12    /* Mostramos los valores de 2 formas */
13    for (fil=0; fil<NFILAS; fil++)
14      for (col=0; col<NCOLUMNAS; col++){
15        printf ("%d - ", matriz[fil][col]);
16        printf ("%d\n", *(*(matriz+fil)+col));
17      }
18    return 0;
19  }
```

Listado 10

En el ejemplo del Listado 10, tenemos una declaración de matriz en la línea 7. ¿Qué significa esa instrucción? Por un lado tenemos un array de 5 elementos (filas), pero a su vez cada uno de esos elementos son arrays de 8 enteros (columnas).

Sabemos que los arrays se almacenan contiguos en memoria. Así, el estado de nuestra memoria podría representarse como:

0000000001**2**345670246810121403691215182 1...

El primer cero indica la dirección de `&matriz[0][0]` (que podemos nombrar también únicamente con el nombre `matriz`). De esta forma, si quisiéramos mostrar el tercer elemento de la segunda fila (el número 2 marcado en negrita), podríamos hacerlo como `matriz[1][2]` o bien como `*(*(matriz+1)+2)`. La primera forma es sencilla; utilizamos la notación de array a la que estamos acostumbrados. La notación de punteros podemos leerla como: *«Dame el contenido de una dirección. Esa dirección está formada por el contenido de donde apunta matriz más uno. A ese resultado súmale dos»*. Podemos ver cada fila de la matriz como un puntero cuyos elementos son punteros a arrays de enteros (ver Figura 9).

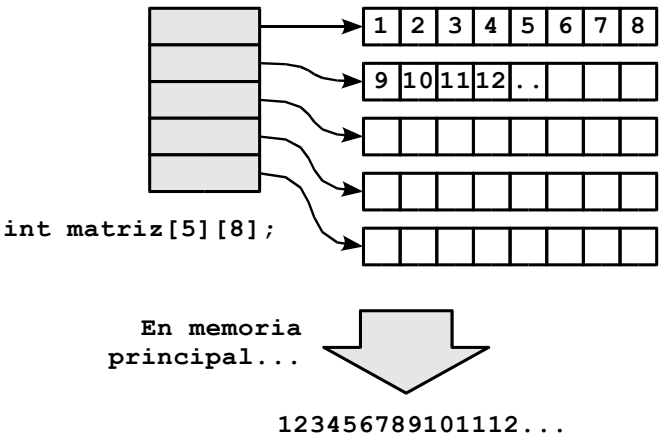

Figura 9. Representación de matriz bidimensional

Así, podemos acceder al elemento de la segunda fila, segunda columna pidiendo primero el contenido de `matriz+1`. Esto nos dará el segundo puntero del array (recordemos que los arrays en C empiezan a numerarse en 0). Con esto, tenemos una dirección de memoria. Si a esa dirección de memoria le sumamos dos, accedemos al tercer elemento del segundo array (en el ejemplo de la Figura 9, tendríamos el 11). En caso de querer acceder al primer elemento del segundo array, bastaría con escribir `*(*(matriz+1))` o, de forma equivalente `**(matriz+1)`.

Queda claro, por tanto, que el primer componente de la matriz bidimensional es equivalente a un array del siguiente tipo: `int *filas[NFILAS];`. En esta declaración estamos diciendo que tenemos un vector cuyos elementos son punteros a enteros, lo mismo que teníamos en la parte de filas de la declaración anterior.

¿Qué tiene que hacer el compilador para acceder a cada uno de los elementos de la matriz? Debe calcular la posición del elemento en memoria. Para ello, deberá multiplicar la fila en la que se encuentra por el número de elementos por columna y sumarle la columna en la que está actualmente. Es decir:

Posición en Memoria = (fila * NCOLS) + columna

Como puede verse en la expresión anterior, es necesario que el compilador sepa a priori, el número de elementos que tiene cada columna de nuestra matriz. En general, es necesario especificar, al menos, la segunda dimensión de la matriz de forma fija. Puede interesarnos implementar una función que nos permita trabajar con nuestro array multidimensional. Un ejemplo muy sencillo (aunque inútil) de esto puede verse en el Listado 11.

```
01  void reset_matriz(int mat[][NCOLUMNAS])
02  {
03    int fil, col;
04    for (fil=0; fil<NFILAS; fil++)
05      for (col=0; col<NCOLUMNAS; col++)
06          mat[fil][col] = 1;
07  }
```

Listado 11

En general, en arrays de más de una dimensión es necesario indicar, en la definición de los parámetros a la función, las dimensiones segunda, tercera, etc...

Asignación dinámica de memoria

En muchas ocasiones no podemos saber a priori el espacio que vamos a necesitar para nuestras estructuras de datos. Esto deberá decidirse en tiempo de ejecución y llevarse a cabo mediante funciones de gestión de memoria como *malloc*, *calloc*, u otras similares.

Cuando pedimos memoria con alguna función de gestión (como *malloc()*), ésta nos devuelve un puntero. En compiladores ANSI, este puntero es de tipo `void`. Como `void` es un tipo genérico que apunta a "cualquier cosa", tendremos que indicar al compilador que haga una conversión de tipo (*cast*) al tipo de datos que nos interese; es decir, a un puntero al tipo de datos que queramos.

Además, será necesario indicarle a la función que llamemos para pedir memoria el número de bytes que necesitamos. En C, el número de bytes que se necesitan para almacenar cada tipo de datos es dependiente de la máquina sobre la que ejecutemos el programa. Por ejemplo, en dos máquinas distintas una puede requerir 2 bytes para un entero y otra utilizar 4 bytes para el mismo entero. ¿Cómo podemos decirle el número de bytes que vamos a necesitar? Utilizando la función `sizeof()`. Así, el tamaño que queremos reservar estará en función del tamaño del tipo de datos que utilicemos. Esto puede verse en el ejemplo del Listado 12.

Carlos González Morcillo (*Carlos.Gonzalez@uclm.es*) ::: Miguel Ángel Redondo Duque (*Miguel.Redondo@uclm.es*)

```
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void)
04  {
05    int *ptr, i;

06    ptr = (int *)malloc(10*sizeof(int));
07    if (ptr==NULL) printf ("Error de Mem.");
08    for (i=0; i<10; i++)
09      ptr[i]=1;
10    return 0;
11  }
```

Listado 12

Utilizando la equivalencia entre la notación de punteros y de arrays, en la línea 9 asignamos a todos los elementos del array creado anteriormente un valor constante 1.

El ejemplo anterior permite crear un array de una dimensión dinámico en tiempo de ejecución, pero ¿cómo crearíamos un array multidimensional?. Hemos visto en el punto anterior que los arrays bidimensionales se gestionan como arrays de punteros a arrays del tipo de datos que queremos utilizar. ¿Cómo reservamos memoria para estas matrices?. En un planteamiento general, necesitaremos generar la matriz dinámicamente, tanto el número de filas como de columnas. En los siguientes ejemplos utilizaremos como tipo de datos base el entero, pero podría utilizarse cualquier otro.

```
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void)
04  {
05    int nfilas=5, ncols=10;
06    int fila;
07    int **filaptr;

08    filaptr = malloc(nfilas * sizeof(int *));
09    if (filaptr == NULL) printf ("Error");

10    for (fila=0; fila<nfilas; fila++) {
11      filaptr[fila] =
            (int *) malloc(ncols * sizeof(int));
12      if (filaptr == NULL) printf ("Error");
13    }
14    return 0;
15  }
```

Listado 13

En el Listado 13, `filaptr` es un puntero a puntero a entero. En este caso, el programa crea una matriz de 5 filas por 10 columnas. Esos datos, naturalmente, podrían gestionarse en tiempo de ejecución.

La primera llamada a `malloc` (línea 8) reserva espacio para el array de punteros a los arrays de enteros. Por esta razón, a la función `sizeof` le pasamos el tipo "*puntero a entero*". Con el primer array de punteros creado, pasamos a reservar memoria para los arrays que son apuntados por cada elemento "*fila*" (en la línea 11).

En el ejemplo anterior se necesitan en total 6 llamadas a `malloc`; una para reservar el array de filas y una para cada array de enteros asociado a cada fila. Además, no nos aseguramos de tener toda la matriz en un bloque contiguo de memoria. Se puede comprobar, utilizando un puntero auxiliar al primer elemento de la primera fila (es decir, usando `int *ptr_aux` que apunte a `filaptr[0]` e incrementando `*(testptr++)` no accedemos a los valores introducidos en la matriz). ¿Tal vez obtenemos un bonito «*segmentantion fault*»?.

Con este planteamiento, deberemos acceder a los elementos de la matriz únicamente utilizando la notación de corchetes. No podemos utilizar un puntero para recorrer la memoria.

```
01  #include <stdio.h>
02  #include <stdlib.h>

03  int main(void)
04  {
05    int nfilas=5, ncols=10;
06    int *mem_matriz;
07    int **filaptr;
08    int *testptr;
09    int fila, col;
10    mem_matriz =
          malloc(nfilas * ncols * sizeof(int));
11    if (mem_matriz == NULL) printf ("Error");

12    filaptr = malloc(nfilas * sizeof(int *));
13    if (filaptr == NULL) printf ("Error");

14    for (fila=0; fila<nfilas; fila++)
15      filaptr[fila] = mem_matriz+(fila*ncols);

16    for (fila=0; fila<nfilas; fila++) {
17      for (col=0; col<ncols; col++)
18        filaptr[fila][col] = 1;
19    }

20    testptr = filaptr[0];
21    for (fila=0; fila<nfilas; fila++)
22     for (col=0; col<ncols; col++)
23        printf ("[%d][%d]%d \n",
                    fila, col, *(testptr++));
24    return 0;
25  }
```

Listado 14

En el ejemplo del Listado 14 se reserva la memoria para la matriz en un bloque contiguo. En la línea 10 se reserva la memoria para todas las celdas de la matriz. En la línea 12 hacemos el array de punteros correspondiente a las filas de la matriz. En las líneas 14-15 hacemos que cada puntero de las filas apunte a la celda donde empieza cada fila. Esto puede calcularse con un simple producto (`fila*ncols`).

Las líneas siguientes (16..23) demuestran que el bloque creado es contiguo, y que podemos utilizar un puntero auxiliar para recorrer las posiciones de memoria.

Con este método se han tenido que utilizar únicamente dos llamadas a la función `malloc`; una para crear el array de filas y otro para reservar el espacio de todas las celdas. Además, sería posible utilizar funciones que trabajan con zonas de memoria para su inicialización de una vez (como `memset()`).

Punteros a funciones

El lenguaje C permite la declaración de punteros a funciones. ¿Para qué puede servir un puntero a una función?. Supongamos que tenemos una función de ordenación implementada (un *quicksort*, por ejemplo). Esta función debería poder utilizarse para ordenar arrays de enteros, de caracteres o de cualquier

tipo de datos. Podíamos implementar diferentes funciones que comparen elementos de cada tipo de datos y pasarle la referencia (puntero) de la función de comparación que corresponda a la función *quicksort*. Así, sólo tendremos una versión del algoritmo de ordenación, exactamente igual para todos los tipos de datos.

```
01  #include <stdio.h>

02  int escribeChar (void *c, void *n)
03  {
04    int i;
05    int num = *((int *)n);
06    char *cadena = (char *)c;

07    for (i=0; i<num; i++)
08      printf ("%c", cadena[i]);

09    return 1;
10  }

11  int escribeInt (void *v, void *n)
12  {
13    int i;
14    int num = *((int *)n);
15    int *entero = (int *)v;

16    for (i=0; i<num; i++)
17      printf ("%d", entero[i]);

18    return 0;
19  }

20  void escribe (void *p, int n,
      int (*funcptr)(void *, void *))
21  {
22    int tipo;
23    tipo = funcptr(p, &n);
24    tipo ? printf (" Cadena\n")
           : printf (" Entero\n");
25  }

26  int main(void)
27  {
28    char cad[3] = {'a','b','c'};
29    int vect[5] = {2,3,4,5,6};

30    escribe (cad, 3, escribeChar);
31    escribe (vect, 5, escribeInt);
32    return 0;
33  }
```

Listado 15

En el ejemplo del Listado 15 se ha realizado un programa muy sencillo que tiene una función llamada `escribe` que permite procesar cualquier tipo de datos. Esta función se encargará de imprimir en pantalla el tipo de datos que se le pase por argumento. Para ello, `escribe` (línea 20) espera como argumentos un puntero (que apuntará a los datos que queremos escribir en pantalla), un entero con el número de elementos a escribir y una función con dos argumentos de tipo puntero.

Este último parámetro debe definirse con cuidado. Estamos indicando que `funcptr` es un puntero a una función que tiene dos argumentos `void *` y que devuelve un entero. Los paréntesis son totalmente necesarios, sin ellos tendríamos: `int * funcptr (const void *, const void *)` con lo que decimos que `funcptr` es una función que devuelve un puntero a entero, lo cual es muy diferente.

En los argumentos de la función se han utilizado punteros genéricos `void *`. Cualquier puntero puede ser forzado (mediante un *cast*) a `void *` y devuelto de nuevo a su tipo original sin pérdida de información. De este modo, podemos pasar datos a la función `escribe` sin tener que indicar previamente su tipo.

Analizando el programa en detalle, vemos que en las líneas 28 y 29 se declaran dos arrays de datos, uno de enteros y otro de caracteres. Estos serán los datos que vamos a imprimir en pantalla mediante la llamada a `escribe`. Seguidamente, en las líneas 30 y 31 se llama a la función `escribe`. Si recordamos el prototipo de llamada a la función (línea 20), necesitamos pasarle un puntero con los datos, un entero y un puntero a función. El *cast* a `void *` se realiza automáticamente, por lo que basta con pasarle `cad` y `vect` en cada llamada. Los punteros a función se pasan con el nombre de la función que ha sido definida previamente (`escribeInt` en la línea 11 y `escribeChar` en la línea 2).

Cuando entramos en `escribeInt`, lo primero que hacemos (en la línea 14) es recuperar el número de elementos a mostrar, `num`. Para ello, accedemos al contenido de la memoria apuntada por `n`, que forzamos mediante un *cast* a ser puntero a entero. Después, forzamos el tipo de datos del `v` a array de enteros. Hecho esto, recorremos el array y escribimos en pantalla cada elemento. De forma similar se trabaja en la función `escribeChar`.

Finalmente, devolvemos un entero a la función `escribe` con información sobre el tipo de datos que se ha tratado. Esta función es totalmente genérica, en la línea 23 llama con los argumentos que ha recibido como parámetro. Podríamos implementar una nueva función que mostrara en pantalla matrices bidimensionales y no habría que añadir nada a la función `escribe`. Bastaría con implementar esta nueva funcionalidad y llamar a la función `escribe` desde `main` pasándole como tercer argumento el nombre de la nueva función.

El mecanismo de punteros a funciones nos permite construir programas fácilmente ampliables, con capacidad de trabajar con tipos de datos genéricos en las funciones principales del programa. Así, el núcleo principal de la aplicación se realizará una vez y podrá trabajar con diferentes tipos de datos sin tener que duplicar código.

¿Y ahora qué?

Simplemente te queda practicar, practicar y practicar. La única forma de aprender los entresijos de un lenguaje tan potente como C es programando con él. Puedes profundizar en el estudio del lenguaje C y, en concreto, en el estudio de punteros, leyendo algunos libros y tutoriales como los que os comentamos a continuación. Mucha suerte y a disfrutar con uno de los lenguajes de programación más usado del mundo.

- Kernighan B. W., Ritchie D. M. "*El lenguaje de programación C*". Ed. Prentice Hall, 1991.
- Gibson T. A. "*Tutorial: Pointers in C and C++*".
- Hefty J. W. "*Punteros en C/C++*".
- Jensen T. "*A tutorial on Pointers and Arrays in C*".