TUTORIAL DE PUNTEROS Y ARRAYS EN C

por Ted Jensen

Versión 1.2 (PDF)

Febrero 2000

Disponible en varios formatos desde
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

Traducido al español por Ramiro Encinas Alarza en diciembre de 2012

http://ramix.net76.net/tutorial_punteros/punteros.pdf

TABLA DE CONTENIDOS

PRÓLOGO

Este documento tiene como objetivo dar una introducción de los punteros para los programadores principiantes del lenguaje de programación C. Tras varios años de lectura y contribución en varias conferencias de C incluidas las relativas a FidoNet y UseNet, he notado que un gran número de recién llegados a C parecen tener dificultades para comprender los fundamentos de los punteros. Por ello emprendí la tarea de tratar de explicarlo mediante un lenguaje sencillo y con un montón de ejemplos.

Esta versión de este documento al igual que la primera versión es de dominio público. Este documento fue recogido por Bob Stout quien lo incluyó en un archivo llamado PTR-HELP.TXT en su ámpliamente distribuida colección de fragmentos de código C. Desde ese lanzamiento original en 1995 he agregado bastante material y he realizado algunas correcciones de menor importancia sobre el trabajo original.

En la versión HTML 1.1, como resultado de los comentarios que me enviaron por correo electrónico desde todo el mundo, realicé unos pocos cambios en la redacción. En la versión 1.2 actualicé los dos primeros capítulos para adaptar en los PCs el cambio de los compiladores de 16 bits a los de 32 bits.

Agradecimientos:

Hay tantas personas que sin saberlo han contribuido a este trabajo mediante las preguntas que se han planteado en C Echo de FidoNet o en el Grupo de Noticias comp.lang.c de UseNet, o en otras conferencias y redes, que sería imposible mencionarlos a todos. Quiero dar un especial agradecimiento a Bob Stout quien tuvo la amabilidad de incluir la primera versión de este material en su archivo de fragmentos de código C.

Sobre el Autor:

Ted Jensen es un Ingeniero Electrónico jubilado que trabajó como diseñador de hardware y responsable de diseñadores de hardware en el campo de la grabación magnética. La programación ha sido su hobby desde 1968, cuando aprendió cómo perforar tarjetas para ejecutarlas en un mainframe (¡el mainframe tenía 64k de memoria magnética principal!).

Uso de este Material:

Por cierto, no necesitas ser un instructor o profesor para ponerte en contacto conmigo. Agradeceré una nota de cualquier persona que le sea útil este material, o que tenga una crítica constructiva para ofrecer. También estaré dispuesto a responder a las preguntas enviadas por correo electrónico a las direcciones indicadas más abajo.

Otras versiones de este documento:

Además de esta versión de hypertexto de este documento, he dejado disponibles otras versiones más adecuadas para imprimir o descargar todo el documento. Si estás interesado en conocer las novedades o nuevas versiones de este documento, visite mi Sitio Web
http://www.netcom.com/~tjensen/ptr/cpoint.htm

Ted Jensen
Redwood City, California
tjensen@ix.netcom.com
Feb. 2000

INTRODUCCIÓN

Si quieres ser un experto en el lenguaje de programación C, debes tener un profundo conocimiento práctico de cómo usar punteros. Por desgracia, los punteros de C parecen representar un obstáculo para los recién llegados, en particular los procedentes de otros lenguajes de programación como Fortran, Pascal o Basic.

He escrito este material para ayudar a los recién llegados con la comprensión de los punteros. Para obtener el máximo beneficio de este material, creo que es importante que el usuario ejecute el código de los listados incluidos. He intentado que el código sea compatible con ANSI para que funcione en cualquier compilador ANSI. También he tratado de escribir el código en bloques de texto, así, con ayuda de un editor de texto ASCII, puedes copiar un bloque de código en un archivo nuevo y compilarlo en tu sistema. Recomiendo a los lectores hacer esto pues será de ayuda para entender el material.

CAPÍTULO 1: ¿Qué es un puntero?

Una de las dificultades que suelen encontrarse los principiantes de C es el concepto de punteros. El objetivo de este tutorial es dar a los principiantes de C una introdución de los punteros y cómo utilizarlos.

Muchas veces he visto que la razón principal de los problemas que tienen los principiantes con los punteros es la carencia o el mínimo conocimiento de las variables (como las utilizadas en C). Por eso, empezaremos con las variables de C en general.

Una variable en un programa es algo con un nombre cuyo valor puede cambiar. El compilador y el enlazador asignan un bloque concreto de memoria del ordenador para alojar el valor de esta variable. El tamaño de ese bloque depende del rango que tiene esa variable para cambiar su valor. Por ejemplo, en un PC de 32 bit, el tamaño de una variable de tipo entero (integer) es de 4 bytes. En PCs viejos de 16 bit los enteros tienen 2 bytes. En C, el tamaño de una variable entera no es necesariamente el mismo y puede variar entre máquinas. Además, hay más de un tipo de variable entera en C. Tenemos enteros (int), enteros largos (long) y enteros cortos (short) que puedes leer en cualquier programa básico en C. Este documento asume que utilizamos un sistema de 32 bit con enteros de 4 bytes.

Si quieres conocer el tamaño de los distintos tipos de enteros que soporta tu sistema, ejecuta el siguiente código:

```
#include <stdio.h>

int main()
{
  printf("capacidad de un short: %d\n", sizeof(short));
  printf("capacidad de un int: %d\n", sizeof(int));
  printf("capacidad de un long: %d\n", sizeof(long));
}
```

Cuando declaramos una variable, le estamos diciendo al compilador dos cosas: el nombre de la variable y su tipo. Por ejemplo, declaramos una variable de tipo entero con el nombre **k** así:

```
    int k;
```

La parte "int" de esta declaración indica al compilador que reserve 4 bytes de memoria (en un PC) para alojar el valor del entero. Además, prepara una tabla de símbolos donde agrega el símbolo **k** y la dirección relativa en memoria donde están esos 4 bytes.
Además, si después escribimos:

```
    k = 2;
```

esperamos que cuando el programa ejecute esta sentencia aloje el valor 2 en el lugar donde **k** tiene reservado su espacio en memoria. En C llamamos "objeto" a una variable como el entero **k**.
Hay dos "valores" asociados con el objeto **k**. Uno es el valor del entero alojado en él (2 del ejemplo anterior) y el otro es el "valor" de la ubicación de la memoria, esto es, la dirección de **k**. Algunos textos se refieren a estos dos valores con la nomenclatura ***rvalue*** (right value, pronunciado "are value") y ***lvalue*** (left value, pronunciado "el value") respectivamente.

En algunos lenguajes, lvalue es el valor permitido en el lado izquierdo del operador de asignación '=' (la dirección donde queda el resultado de la evaluación del lado derecho del operador de asignación). rvalue es lo que está en el lado derecho de la instrucción de asignación (el **2** anterior). El rvalue no puede utilizarse en el lado izquierdo de la instrucción de asignación. Por eso **2 = k** es ilegal.

En realidad, la definición anterior de "lvalue" está adaptada en C según K&R II (página 197):

> "Un ***objeto*** es una región de almacenamiento con un nombre; un ***lvalue*** es una expresión que se refiere a un objeto."

En cualquier caso hasta ahora, la definición antes citada originalmente es suficiente. A medida que nos familiaricemos con los punteros entraremos en más detalles.

Bien, ahora consideremos lo siguiente:

```
int j, k;

k = 2;
j = 7;    <-- línea 1
k = j;    <-- línea 2
```

En el código de arriba, el compilador interpreta la **j** de la línea 1 como la dirección de la variable **j** (su lvalue) y crea el código para copiar el valor 7 a esa dirección. Sin embargo, en la línea 2, **j** lo interpreta como su rvalue (porque está en el lado derecho del operador de asignación '='). Esto es, **j** se refiere al valor ***guardado*** en la ubicación de memoria de **j**, que en este caso es 7. Por tanto, el 7 se copia a la dirección asignada por el lvalue de **k**.

En estos ejemplos utilizamos enteros de 4 bytes, y por tanto cuando se copia un rvalue de un sitio a otro se están copiando 4 bytes. De la misma forma, si los enteros son de 2 bytes, estaremos copiando 2 bytes.

Digamos ahora que tenemos una razón para querer una variable que aloje un lvalue (una dirección). El tamaño para este valor dependerá del sistema. En ordenadores viejos de 64k de memoria total, la dirección de cualquier punto de memoria ocupa 2 bytes. Los ordenadores con más memoria necesitarán más bytes para alojar una dirección. El tamaño actual necesario no es demasiado importante pues siempre hay una manera de decirle al compilador que lo que queremos guardar es una dirección.

Esta variable se denomina una ***variable puntero*** (por razones que espero se aclaren después). Cuando en C definimos una variable puntero, lo hacemos con un nombre precedido por un asterisco. En C también le asignamos un tipo al puntero que, en este caso, se refiere al tipo del dato guardado en la dirección donde apuntará nuestro puntero. Por ejemplo, consideremos la siguiente declaración de variable:

```
int *ptr;
```

ptr es el nombre de nuestra variable (como **k** fue el nombre de nuestra variable entera). El '*' indica al compilador que queremos una variable puntero, es decir, reserva los bytes necesarios para almacenar una dirección de memoria. **int** indica que queremos utilizar nuestra variable puntero para alojar la dirección de un entero. Se dice que este puntero "apunta a" un entero. Sin embargo, hay que tener en cuenta que cuando escribimos **int k;** no le estamos dando un valor a **k**.

Si esta definición se realiza fuera de cualquier función compatible con ANSI, el compilador la inicializará a cero. Igualmente, **ptr** no tiene valor porque en la declaración no hemos guardado una dirección en él.

En este caso, de nuevo, si la declaración se realiza fuera de cualquier función, se inicializará a un valor que no apuntará a ningún objeto o función de C. Un puntero inicializado de esta manera se llama puntero "null (nulo)".

El patrón actual de bits utilizado para un puntero nulo puede o no evaluarse como cero porque depende específicamente del sistema donde se desarrolle el código. Para que el código fuente sea compatible entre compiladores de varios sistemas se utiliza una macro para representar un puntero nulo. Esta macro se llama NULL. De esta forma, estableciendo el valor del puntero con la macro NULL (ptr = NULL), se garantiza que el puntero es un puntero nulo. De la misma forma que podemos evaluar si el valor de un entero es cero, como en **if(k == 0)**, también podemos evaluar si un puntero es nulo con **if (ptr == NULL)**.

Volvamos a utilizar nuestra nueva variable **ptr**. Supongamos que queremos guardar en **ptr** la dirección de nuestra variable entera **k**. Para hacerlo utilizamos el operador unario **&** y escribimos:

```
    ptr = &k;
```

Lo que hace el operador **&** es conseguir el lvalue (la dirección) de **k**, incluso aunque **k** esté en el lado derecho del operador de asignación '=', y lo copia al contenido de nuestro puntero ptr. Ahora decimos que ptr "apunta a" **k**. Ahora sólo queda por ver un operador más.
El "operador de desreferencia" es el asterisco y se utiliza de la siguiente forma:

```
    *ptr = 7;
```

copiará 7 a la dirección donde apunta **ptr**. De esta forma, si **ptr** "apunta a" (contiene la dirección de) **k**, la declaración anterior establecerá el valor de **k** a 7. Esto es, cuando utilizamos '*' de esta manera nos referimos al valor a donde ptr está apuntando, no al valor en sí del puntero.
Igualmente, podemos escribir:

```
    printf("%d\n",*ptr);
```

para mostrar por pantalla el valor del entero alojado en la dirección donde está apuntando **ptr**;.
Una forma de ver cómo todo esto encaja sería la de ejecutar el siguiente programa y luego revisar con antención el código y la salida.

```
------------ Programa 1.1 --------------------------------

/* Programa 1.1 de PTRTUT10.TXT   6/10/97 */

#include <stdio.h>

int j, k;
int *ptr;

int main(void)
{
    j = 1;
    k = 2;
    ptr = &k;
    printf("\n");
    printf("j tiene el valor %d y se aloja en %p\n", j, (void *)&j);
```

```
    printf("k tiene el valor %d y se aloja en %p\n", k, (void *)&k);
    printf("ptr tiene el valor %p y se aloja en %p\n", ptr, (void *)&ptr);
    printf("El valor del entero apuntado por ptr es %d\n", *ptr);

    return 0;
}
```

Nota: Aún tenemos que ver los aspectos de C que requieren el uso de la expresión **(void *)** utilizada aquí. Por el momento, lo utilizaremos en nuestro código de prueba y explicaremos más adelante la razón que hay detrás de esta expresión.

Para repasar:

- Una variable se declara dándola un tipo y un nombre (**int k;**)
- Una variable puntero se declara dándola un tipo y un nombre (**int *ptr**) donde el asterisco indica al compilador que la variable llamada **ptr** es una variable puntero y el tipo indica al compilador a qué tipo está apuntando (tipo entero, en este caso).
- Una vez declarada la variable **k**, podemos conseguir su dirección precediendo su nombre con el operador unario **&**, como en **&k**.
- Podemos "desreferenciar" un puntero, es decir, se refiere al valor de donde está apuntando, utilizando el operador unario '*' como en ***ptr**.
- Un "lvalue" de una variable es el valor de su dirección, es decir, donde está alojada en memoria. El "rvalue" de una variable es el valor alojado en esa variable (donde indica la dirección).

Referencias del Capítulo 1:

1."The C Programming Language" 2nd Edition
B. Kernighan and D. Ritchie
Prentice Hall
ISBN 0-13-110362-8

CAPÍTULO 2: Tipos de punteros y Arrays

Bueno, seguimos adelante. Veamos por qué tenemos que identificar el ***tipo*** de variable a la que apunta un puntero, como en:

```
int *ptr;
```

Una vez que ptr "apunta a" algo, si escribimos:

```
*ptr = 2;
```

el compilador sabrá cuantos bytes tiene que copiar en la ubicación de memoria donde apunta **ptr**. Si **ptr** se declara como un puntero a un entero, se copiarán 4 bytes. De igual forma ocurre para floats y doubles donde se copiará el número de bytes apropiado. De todas formas, la definición del tipo a donde apunta el puntero le da al compilador otras opciones interesantes. Por ejemplo, consideremos un bloque en memoria de diez enteros contiguos. Esto es, reservamos 40 bytes de memoria para alojar 10 enteros.

Ahora, digamos que apuntamos con nuestro puntero entero **ptr** al primero de esos enteros de ese bloque. Por otro lado, digamos que ese entero está ubicado en la posición 100 (decimal) de la memoria. ¿Qué ocurre si escribimos lo siguiente?:

```
ptr + 1;
```

Dado que el compilador "sabe" que esto es un puntero (es decir, su valor es una dirección) y que apunta a un entero (su valor actual, 100, es la dirección de un entero), agrega 4 a **ptr** en lugar de 1, por lo que el puntero "apunta a" el **siguiente entero**, en la posición 104 de memoria. De igual forma, si **ptr** se declara como un puntero a un short, agregaría 2 en lugar de 1. Lo mismo ocurre con otros tipos de datos como floats, doubles, o incluso con tipos de datos definidos por el usuario como las estructuras. Obviamente, esto no es el mismo tipo de "adición" que solemos pensar. En C se lo conoce como adición mediante "aritmética de punteros", un término que veremos después.

Asimismo, dado que **++ptr** y **ptr++** son ambos equivalentes a **ptr + 1** (aunque puede diferir el lugar donde **ptr** se incremente), el incremento de un puntero mediante el operador unario ++, ya sea a la izquierda o a la derecha del nombre del puntero, incrementa la dirección que está alojando por la cantidad de sizeof(type) donde "type" es el tipo del objeto donde apunta (es decir, 4 para un entero).

Puesto que un bloque de 10 enteros contiguos en memoria es, por definición, un array de enteros, esto nos lleva a una interesante relación entre arrays y punteros.

Consideremos lo siguiente:

```
int mi_array[] = {1,23,17,4,-5,100};
```

Aquí tenemos un array con 10 enteros. Nos referimos a cada uno de estos enteros mediante una posición de **mi_array**, es decir, utilizando **mi_array[0]** hasta **mi_array[5]**. Pero también podemos acceder a ellos mediante un puntero de la siguiente forma:

```
int *ptr;
ptr = &mi_array[0];       /* ahora el puntero ptr apunta al
                             primer entero de nuestro array */
```

Y entonces podemos mostrar nuestro array mediante la notación de array o desreferenciando nuestro puntero. El siguiente código lo muestra:

```
-----------  Programa 2.1  ----------------------------------

/* Programa 2.1 de PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

int mi_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
    int i;
    ptr = &mi_array[0];       /* ahora el puntero ptr apunta al
                                 primer elemento del array */
    printf("\n\n");

    for (i = 0; i < 6; i++)
    {
      printf("mi_array[%d] = %d   ",i,mi_array[i]);   /*<-- A */
      printf("ptr + %d = %d\n",i, *(ptr + i));  /*<-- B */
    }
    return 0;
}
```

Compila y ejecuta el código anterior y pon atención a las líneas A y B, pues el programa muestra los mismos valores en los dos casos. Fíjate también en cómo se desreferencia nuestro puntero en la línea B, ese decir, primero agregamos i al puntero y después desreferenciamos el nuevo puntero. Ahora cambia la línea B de esta forma:

```
    printf("ptr + %d = %d\n",i, *ptr++);
```

ejecútalo de nuevo... ahora cámbialo por:

```
    printf("ptr + %d = %d\n",i, *(++ptr));
```

y prueba una vez más. En cada prueba, intenta predecir el resultado y pon atención al resultado real.

En C, la norma indica que donde utilicemos **&nombre_var[0]**, se puede reemplazar con **nombre_var**. En nuestro código, donde escribimos:

```
    ptr = &mi_array[0];
```

podemos sustituirlo por:

```
    ptr = mi_array;
```

para conseguir el mismo resultado.

Esto lleva a que muchos textos indican que el nombre de un array es un puntero. Yo prefiero pensar **"el nombre de un array es la dirección de su primer elemento"**. Muchos principiantes (yo incluido cuando estaba aprendiendo) tienen la tendencia a confundirse cuando piensan en ello como un puntero. Por ejemplo, sí podemos escribir

```
ptr = mi_array;
```

pero no podemos escribir

```
mi_array = ptr;
```

La razón es que **ptr** es una variable, pero **mi_array** es una constante. Es decir, la posición del primer elemento de **mi_array** no puede cambiarse una vez que **mi_array[]** se ha declarado.

Cuando antes hablábamos del término "lvalue" yo cité K&R-2 donde indica:

> "Un ***objeto*** es una región de almacenamiento con un nombre; un ***lvalue*** es una expresión que se refiere a un objecto."

Esto plantea un problema interesante. Debido a que **mi_array** es una región de almacenamiento con un nombre, ¿por qué **mi_array** en la instrucción de asignación anterior no es un lvalue? Para resolver este problema, algunos dicen que **mi_array** es un "lvalue no modificable".

Modifica el programa anterior cambiando

```
ptr = &mi_array[0];
```

por

```
ptr = mi_array;
```

y ejecútalo de nuevo para comprobar que el resultado es el mismo.

Ahora, vamos a profundizar un poco más en la diferencia entre los nombres **ptr** y **mi_array** como hemos visto. Algunos autores se refieren al nombre de un array como un puntero ***constante***. ¿Qué quiere decir eso? Bien, para entender el término "constante" en este caso, volvamos a nuestra definición del término "variable". Cuando declaramos una variable estamos eligiendo un punto en la memoria para alojar en él un valor con un tipo apropiado. Realizado esto, el nombre de la variable puede interpretarse de dos formas. Cuando se utiliza a la izquierda del operador de asignación, el compilador lo interpreta como la dirección de memoria donde se alojará el valor resultante de la evaluación del lado derecho del operador de asignación. Pero cuando se utiliza a la derecha del operador de asignación, el nombre de una variable se interpreta como el contenido de esa dirección de memoria (el valor de esa variable).

Con esto en mente, consideremos ahora unas constantes simples como:

```
int i, k;
i = 2;
```

Aquí, mientras **i** es una variable que ocupa un espacio en la porción de datos de la memoria, **2** es una constante que, en lugar de alojarse en el segmento de datos de la memoria, se encuentra incrustada directamente en el segmento de código de la memoria.

Esto es, cuando escribimos algo como **k = i;** el compilador crea código que al ejecutarse determinará el valor alojado en la dirección de memoria **&i** para moverlo a **k**, y cuando escribimos algo como **i = 2;** el compilador simplemente coloca el **2** en el código, donde no hay ninguna referencia al segmento de datos. Tanto **k** como **i** son objetos, pero **2** no es un objeto.

Del mismo modo, como **mi_array** es una constante, una vez que el compilador establece el lugar donde guardará el array, ya "sabe" la dirección de **mi_array[0]** y viendo:

```
ptr = my_array;
```

sabemos que utiliza esta dirección como una constante en el segmento de código y no existe referencia al segmento de datos.

Este podría ser un buen lugar para explicar con más detalle el uso de la expresión **(void*)** del programa 1.1 del Capítulo 1. Como hemos visto, podemos tener punteros de varios tipos. Hasta ahora hemos hablado de punteros a enteros (int) y punteros a caracter (char). En próximos capítulos vamos a ver punteros a estructuras e incluso punteros a punteros.

También hemos aprendido que entre sistemas, el tamaño de un puntero puede variar. Resulta que también es posible que el tamaño de un puntero pueda variar en función del tipo de datos del objeto al que apunta. Por lo tanto, podemos tener problemas si asignamos una variable de tipo entero largo (long) a una variable de tipo entero corto (short) y del mismo modo, también podemos tener problemas si intentamos asignar punteros de varios tipos a punteros de otros tipos.

Para minimizar este problema, C proporciona el tipo void para punteros. Podemos declarar este tipo de punteros así:

```
void *vptr;
```

Un puntero void es una especie de puntero genérico. Por ejemplo: C no permite comparar un puntero de tipo int con un puntero de tipo char, pero sí permite comparar cualquiera de ellos con un puntero de tipo void. Por supuesto, como con otras variables, se pueden utilizar conversiones (cast) para convertir un tipo de puntero a otro tipo bajo las condiciones adecuadas. En el programa 1.1. del Capítulo 1 se realiza una conversión de punteros enteros a punteros void para hacerlos compatibles con la especificación de conversión %p. En capítulos posteriores realizaremos otras conversiones por razones que ya veremos.

Bien, con esto tenemos un montón de información técnica para digerir y no espero que un principiante entienda todo esto a la primera. Con tiempo y práctica querrás repasar los dos primeros capítulos. Por ahora, vamos a pasar a la relación entre punteros, arrays de caracteres (char) y cadenas de texto (strings).

CAPÍTULO 3: Punteros y Cadenas de Texto

El estudio de las cadenas de texto es útil y hace más estrecha la relación entre punteros y arrays. También ilustra de forma fácil cómo puede ser la implementación de algunas funciones de cadenas de texto en C estándar. Finalmente se muestra cómo y cuándo los punteros pueden y deben pasarse como argumentos a una función.

En C, las cadenas son arrays de carácteres (char), pero esto a veces no se cumple en otros lenguajes. En BASIC, Pascal, Fortran y otros lenguajes, una cadena tiene su propio tipo, pero en C no. En C una cadena es un array de caracteres que finaliza con un caracter binario cero (escrito como **'\0'**). Para comenzar vamos a escribir algo de código que, sólo con fines ilustrativos, probablemente nunca escribirías en un verdadero programa. Consideremos, por ejemplo:

```
char mi_cadena[40];

mi_cadena[0] = 'T';
mi_cadena[1] = 'e';
mi_cadena[2] = 'd':
mi_cadena[3] = '\0';
```

Uno nunca crearía una cadena así, pero el resultado final es una cadena compuesta por un array de caracteres **finalizada por un caracter nul** (así lo define C). Hay que ser conscientes de que "nul" **no es** lo mismo que "NULL". nul se refiere a un cero definido por la secuencia de escape **'\0'** (ocupa un byte en memoria). NULL, por otra parte, es el nombre de una macro que sirve para inicializar punteros nulos y se define en el archivo de cabecera del compilador de C (nul no puede definirse).

Tardaríamos demasiado tiempo en escribir el código anterior, así que C permite dos formas alternativas para conseguir lo mismo. En primer lugar, podríamos escribir:

```
char mi_cadena[40] = {'T', 'e', 'd', '\0',};
```

Pero también tardaríamos más de lo conveniente, por eso C permite:

```
char mi_cadena[40] = "Ted";
```

Al utilizar comillas dobles en lugar de comillas simples, el carácter nul (**'\0**') se agrega automáticamente al final de la cadena.

En el resto de casos ocurre lo mismo: el compilador reserva un bloque en memoria de 40 bytes para alojar los caracteres e inicializa sus primeros 4 caracteres con **Ted\0**.

Ahora consideremos el siguiente programa:

```
------------------programa 3.1-------------------------------------

/* Programa 3.1 de PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

char strA[80] = "Una cadena de prueba";
char strB[80];

int main(void)
{
    char *pA;     /* un puntero de tipo caracter */
    char *pB;     /* otro puntero de tipo caracter */
    puts(strA);   /* muestra la cadena strA */
    pA = strA;    /* apunta pA a strA */
    puts(pA);     /* muestra a donde apunta pA */
    pB = strB;    /* apunta pB a strB */
    putchar('\n');       /* salto de línea */
    while(*pA != '\0')   /* línea A (ver texto) */
    {
        *pB++ = *pA++;   /* línea B (ver texto) */
    }
    *pB = '\0';          /* línea C (ver texto) */
    puts(strB);          /* muestra strB */
    return 0;
}

--------- fin programa 3.1 ---------------------------------------
```

En el código anterior comenzamos definiendo dos arrays de 80 caracteres cada uno. Como están definidos de forma global, ambos se inicializan primero con **'\0** y después **strA** se inicializa con los caracteres de la cadena de texto entre comillas.

Ahora, volviendo al código, declaramos dos punteros y mostramos por pantalla la cadena de texto. Entonces "apuntamos" el puntero **pA** a **strA**, es decir, por medio de la instrucción de asignación se copia la dirección de **strA[0]** en nuestra variable **pA** y utilizamos **puts()** para mostrar por pantalla donde apunta **pA**. Consideremos ahora que el prototipo de la función **puts()** es:

```
    int puts(const char *s);
```

Por ahora ignoremos **const**. El parámetro pasado a **puts()** es un puntero, es decir, el **valor** de un puntero (ya que todos los parámetros en C se pasan por valor), y el valor de un puntero es la dirección a donde apunta, o, simplemente una dirección. Así, cuando escribimos **puts(strA);** como hemos visto, estamos pasando la dirección de **strA[0]**.

Del mismo modo, cuando escribimos **puts(pA);** estamos pasando la misma dirección, ya que hemos realizado la asignación **pA = strA;**

Teniendo en cuenta todo esto, bajamos hasta la sentencia **while()** en la línea A. La línea A dice:

Mientras el caracter al que apunta **pA** (es decir ***pA**) no sea un caracter nul (es decir **'\0'**), haz lo siguiente:

La línea B dice: copia el carácter apuntado por **pA** al espacio donde apunta **pB**, y luego incrementa **pA** para que apunte al siguiente caracter y **pB** para que apunte al siguiente espacio.

Cuando se copia el último caracter, **pA** apunta al caracter de terminación nul y finaliza el bucle. Sin embargo no se ha copiado el caracter nul y como en C, por definición, una cadena de texto **debe** finalizar con nul, lo agregamos con la línea C.

Es muy educativo ejecutar este programa con el depurador para ver **strA**, **strB**, **pA** y **pB** paso a paso. Incluso es más educativo definir **strB[]** con algo como:

```
strB[80] = "12345678901234567890123456789012345678901234567890"
```

donde el número de dígitos utilizados es mayor que la longitud de **strA** y repetir la depuración paso a paso viendo las variables anteriores. ¡Pruébalo!

Volviendo al prototipo de **puts()** por un momento, al utilizar "const" como un modificador del parámetro indica al usuario que la función no modificará la cadena apuntada por **s**, es decir, se tratará a esa cadena como una constante.

Por supuesto, lo que muestra el programa anterior es una forma simple de copiar una cadena de texto. Después de que comprendas bien lo que sucede jugando con lo anterior, podemos crear nuestra propia versión de la función **strcpy()** estándar de C, que podría ser:

```
char *mi_copiacadena(char *destino, char *origen)
{
    char *p = destino;
    while (*origen != '\0')
    {
        *p++ = *origen++;
    }
    *p = '\0';
    return destino;
}
```

En este caso, he seguido la práctica utilizada en la rutina estándar de devolver un puntero a la cadena de destino.

De nuevo, la función está diseñada para aceptar dos valores de dos punteros de tipo caracter (char), es decir, direcciones, por lo que en el programa anterior podríamos escribir:

```
int main(void)
{
    mi_copiacadena(strB, strA);
    puts(strB);
}
```

Me he desviado ligeramente de la forma utilizada en C estándar que tendría el prototipo:

```
char *mi_copiacadena(char *destino, const char *origen);
```

Aquí, el modificador "const" se utiliza para asegurar que la función no modificará el contenido donde apunta el puntero origen. Puedes probarlo modificando la función anterior y su prototipo para incluir el modificador "const" como indica. Después, dentro de la función puedes agregar una declaración que cambie el contenido donde apunta el puntero origen, como por ejemplo:

```
*origen = 'X';
```

que cambiaría el primer caracter de la cadena con una X. El modificador const debería provocar un error al compilar el programa. Pruébalo y verás.

Ahora, consideremos algunas de las cosas que hemos visto en ejemplos anteriores. En primer lugar, hay que tener en cuenta el hecho de que ***ptr++** debe ser interpretado como el valor donde apunta **ptr** y después el incremento del valor del puntero. Esto tiene que ver con la precedencia de los operadores. Si escribiéramos **(*ptr)++** no estaríamos incrementado el puntero, sino el valor donde apunta, es decir, si se utiliza en el primer caracter del ejemplo anterior ("Ted"), 'T' se incrementaría en uno, es decir: 'U'. Puedes escribir algún ejemplo simple para verlo.

Recordemos una vez más que una cadena de texto no es más que una serie de carácteres donde el último caracter es un **'\0'**. Lo que hemos hecho antes es copiar un array de caracteres, pero la técnica se puede aplicar a un array de enteros, floats, etc. En esos casos, sin embargo, al no tratarse de cadenas de texto, el final del array no acabaría con ningún caracter especial como nul. Podríamos implementar una versión basada en un valor especial para identificar el final del array, por ejemplo: copiar un array de enteros positivos marcando el final con un entero negativo. Lo normal cuando escribimos una función para copiar un array de elementos que no sean una cadena de texto es pasar un argumento más con el número de elementos a copiar, así como otro argumento con la dirección del array, es decir, algo parecido al siguiente prototipo:

```
void int_copiar(int *ptrA, int *ptrB, int nbr);
```

donde **nbr** es el número de enteros a copiar. Puedes jugar con esta idea, crear un array de enteros, ver si puedes escribir la función **int_copiar** y hacerla funcionar.

Esto permite utilizar funciones para manipular grandes arrays. Por ejemplo, si tenemos un array de 5000 enteros y queremos manipularlo con una función, sólo necesitamos pasar a esta función la dirección del array (y alguna información auxiliar como el entero nbr anterior, dependiendo de lo que necesites). El array en sí **no** se pasa a la función, es decir, no se copia todo el array y se pone en la pila antes de llamar a la función, sólo se pasa su dirección.

Otra cosa distinta es pasar, digamos un entero a una función. Cuando pasamos un entero hacemos una copia del entero, es decir, ponemos su valor en la pila. Cualquier manipulación del valor pasado dentro de la función no afecta al entero original, pero con arrays y punteros podemos pasar la dirección de la variable y manipular los valores de las variables originales.

CAPÍTULO 4: Más sobre Cadenas de Texto

Bien, ¡hemos avanzado mucho en poco tiempo! Vamos a retroceder un poco para ver lo que hicimos en el Capítulo 3 cuando copiábamos cadenas de texto, pero ahora lo haremos desde una perspectiva diferente. Consideremos la siguiente función:

```
char *mi_copiacadena(char destino[], char origen[])
{
    int i = 0;
    while (origen[i] != '\0')
    {
        destino[i] = origen[i];
        i++;
    }
    destino[i] = '\0';
    return destino;
}
```

Recordemos que las cadenas de texto son arrays de caracteres. Para esta copia hemos utilizado la notación de array en lugar de la notación de punteros. El resultado es el mismo, es decir, con esta notación la cadena se copia con la misma precisión que en el ejemplo anterior con punteros. Esto plantea algunas cosas interesantes que vamos a ver.

Como los parámetros se pasan por valor, tanto si pasamos un puntero a un carácter como si pasamos el nombre del array, como hemos visto antes, lo que realmente se pasa es la dirección del primer elemento de cada array. Por lo tanto, el valor numérico del parámetro que se pasa a la función es el mismo, tanto si se utiliza un puntero a un caracter, como si utilizamos un nombre de array. Esto implica de alguna manera que **origen[i]** es lo mismo que ***(p+i)**.

De hecho esto es cierto, es decir, donde escribimos **a[i]**, puede ser reemplazado con ***(a + i)** sin ningún problema, pues el compilador creará el mismo código en ambos casos. Así vemos que la aritmética de punteros es lo mismo que la indexación de un array. Ambas sintáxis producen el mismo resultado.

Esto NO quiere decir que los punteros y los arrays sean la misma cosa, porque no lo son. Sólo decimos que para identificar un elemento de un array podemos elegir una de entre dos sintáxis: una utiliza la indexación de un array y la otra utiliza aritmética de punteros, ambas con el mismo resultado.

Ahora miremos una parte concreta de esta última expresión: **(a + i)**, que es una adición simple utilizando el operador + . Las reglas de C indican que esta expresión es conmutativa, esto es: **(a + i)** es idéntico a **(i + a)**, por eso podemos escribir ***(i + a)** como ***(a + i)** y obtendremos el mismo resultado.

Pero ***(i + a)** podría haber venido de **i[a]** curiosamente:

```
char a[20];
int i;
```

si escribirmos

```
a[3] = 'x';
```

es lo mismo que si escribimos

```
3[a] = 'x';
```

¡Pruébalo! Crea un array de elementos de tipo char, int o long, etc, después al elemento 3 o 4 asígnale un valor de forma convencional y después muestra el resultado. Después invierte la notación del array como acabamos de ver. Un buen compilador no apreciará la diferencia y los resultados serán idénticos. Esto es una curiosidad... ¡nada más!

Ahora, volviendo a la función de antes, cuando escribimos:

```
destino[i] = origen[i];
```

como la indexación de arrays y la aritmética de punteros dan los mismos resultados, podemos escribirlo como:

```
*(destino + i) = *(origen + i);
```

Aquí vemos dos adiciones por cada valor de i, y normalmente las adiciones tardan más que las incrementaciones (como las realizadas con el operador ++ como en **i**++). En compiladores modernos y optimizados es posible que no sea así, pero uno nunca sabe. Por lo tanto, la versión con punteros puede ser un poco más rápida que la versión con arrays.

Otra forma de acelerar la versión con punteros sería cambiar:

```
while (*origen != '\0')
```

a simplemente :

```
while (*origen)
```

puesto que en ambos casos el valor dentro del paréntesis será cero (FALSE).

En este punto es posible que quieras experimentar un poco escribiendo tus propios programas utilizando punteros para manipular cadenas de texto. También es posible que quieras escribir tus propias versiones de funciones estándar como:

```
strlen();
strcat();
strchr();
```

y de cualquier otra que tengas en tu sistema.

Volveremos con las cadenas de texto y su manipulación mediante punteros en un capítulo posterior. En el siguiente capítulo vamos a ver un poco de estructuras.

CAPÍTULO 5: Punteros y Estructuras

Como sabemos, podemos declarar un bloque de datos que alojen datos de diferentes tipos mediante la declaración de una estructura. Por ejemplo, un archivo personal puede contener una estructura parecida a la siguiente:

```
    struct registro {
        char nombre[20];        /* nombre */
        char apellidos[50];     /* apellidos */
        int edad;               /* edad */
        float salario;          /* por ej. 12.75 por hora */
    };
```

Digamos que tenemos un montón de estructuras como éstas en un archivo y queremos leerlas para mostrar los nombres y apellidos en un listado de personas. El resto de información no la necesitamos. Queremos hacer esto llamando a una función y pasarle un puntero a la estructura en cuestión. Para demostrarlo utilizaremos por ahora sólo una estructura y el objetivo será la escritura de esta función, no la lectura del archivo, que, seguramente ya sabemos hacer.

Como repaso, recuerda que podemos acceder a los miembros de una estructura mediante el operador punto, como en:

```
--------------- programa 5.1 -----------------

/* Programa 5.1 de PTRTUT10.HTM     6/13/97 */


#include <stdio.h>
#include <string.h>

struct registro {
    char nombre[20];      /* nombre */
    char apellidos[50];   /* apellidos */
    int edad;             /* edad */
    float salario;        /* por ej. 12.75 por hora */
};

struct registro mi_estructura;       /* declaración de la estructura
mi_estructura */

int main(void)
{
    strcpy(mi_estructura.nombre,"Ted");
    strcpy(mi_estructura.apellidos,"Jensen");
    printf("\n%s ",mi_estructura.nombre);
    printf("%s\n",mi_estructura.apellidos);
    return 0;
}

--------------- fin del programa 5.1 ---------
```

Ahora bien, esta estructura en particular es bastante pequeña en comparación con otras muchas utilizadas en programas escritos en C. A lo anterior es posible que quieras agregar:

```
fecha_contratacion;      (no se muestran los tipos de datos)
fecha_ultimo_ascenso;
ultimo_porcentaje_incremento;
telefono_emergencia;
seguro_medico_privado;
numero_seguridad_social;
etc.....
```

Si tenemos muchos empleados, lo suyo es manipular los datos de esas estructuras mediante funciones. Por ejemplo, podríamos querer pasar una estructura a una función para visualizar el nombre del empleado de esta estructura. Sin embargo en el original C (Kernighan & Ritchie, 1st Edition) no era posible pasar una estructura a una función, sólo se podía pasar un puntero que apunte a la estructura en cuestión. En ANSI C sí es posible pasar una estructura completa. Pero, ya que ahora nuestro objetivo es aprender más acerca de los punteros, lo veremos con punteros.

De todas formas, si pasamos toda la estructura a la función, el contenido de la estructura se copiará desde la función que realiza la llamada a la función que recibe la llamada. En sistemas que utilizan pilas, esto se realiza metiendo los contenidos de la estructura en la pila, pero si tenemos estructuras muy grandes, esto puede ser un problema. Sin embargo, si a la función le pasamos un puntero que apunta a una estructura, estaremos metiendo un puntero en la pila, que es una cantidad mínima de espacio.

En cualquier caso, como estamos hablando de punteros, veremos cómo pasar un puntero a una estructura y utilizarlo dentro de una función.

Consideremos el caso que hemos descrito, es decir, queremos una función que acepte un puntero (que apunta a una estructura) como parámetro, y dentro de esta función queremos acceder a los miembros de esta estructura para mostrar el nombre del empleado.

Bien, sabemos que nuestro puntero tiene que apuntar a una estructura declarada mediante struct registro. Este puntero lo declaramos así:

```
struct registro *st_ptr;
```

y después hacemos que el puntero apunte a nuestra estructura con:

```
st_ptr = &mi_estructura;
```

Ahora podemos acceder a un miembro concreto mediante la desreferenciación del puntero, pero ¿cómo lo hacemos con un puntero que apunta a una estructura? Bien, consideremos que queremos utilizar el puntero para guardar la edad de un empleado. Lo podemos hacer así:

```
(*st_ptr).edad = 63;
```

Mira esto con atención porque esto quiere decir: reemplaza lo que está entre paréntesis con el valor donde apunta **st_ptr**, que es la estructura **mi_estructura**. Si utilizamos **mi_estructura.edad**, el resultado sería el mismo.

Como esta es una expresión bastante frecuente, los diseñadores de C han creado una sintáxis alternativa para hacer lo mismo:

```
st_ptr->edad = 63;
```

Con esto en mente, mira el siguiente programa:

```
------------ programa 5.2 --------------------

/* Programa 5.2 de PTRTUT10.HTM   6/13/97 */

#include <stdio.h>
#include <string.h>

struct registro{                /* el tipo de estructura */
    char nombre[20];            /* nombre */
    char apellidos[50];         /* apellidos */
    int edad;                   /* edad */
    float salario;              /* por ej. 12.75 por hora */
};

struct registro mi_estructura;            /* define la estructura */
void mostrar_nombre(struct registro *p);  /* prototipo de función */

int main(void)
{
    struct registro *st_ptr;     /* un puntero de tipo registro (estructura) */
    st_ptr = &mi_estructura;     /* el puntero apuntando a mi_estructura */
    strcpy(mi_estructura.nombre,"Ted");
    strcpy(mi_estructura.apellidos,"Jensen");
    printf("\n%s ",mi_estructura.nombre);
    printf("%s\n",mi_estructura.apellidos);
    mi_estructura.edad = 63;
    mostrar_nombre(st_ptr);      /* pasa el puntero */
    return 0;
}

void mostrar_nombre(struct registro *p)
{
    printf("\n%s ", p->nombre);  /* p apunta a una estructura */
    printf("%s ", p->apellidos);
    printf("%d\n", p->edad);
}

-------------------- fin del programa 5.2 ----------------
```

Una vez más, tenemos un montón de información para digerir. El lector debe compilar y ejecutar los diversos fragmentos de código y utilizar un depurador para ver cosas como **mi_estructura** y **p** paso a paso para ver lo que sucede.

CAPÍTULO 6: Más sobre Cadenas de Texto y Arrays de Cadenas

Bien, vamos a volver un poco con las cadenas. En el código que vamos a ver, todas las asignaciones serán globales, es decir, fuera de cualquier función y también fuera de main().

Antes hemos visto que:

```
char mi_cadena[40] = "Ted";
```

asigna espacio para un array de 40 bytes y coloca la cadena en los primeros 4 bytes (tres para los caracteres entre comillas y un cuarto para la terminación **'\0'**).

En realidad, si sólo queremos guardar el nombre "Ted" podríamos escribir:

```
char mi_nombre[] = "Ted";
```

entonces el compilador contaría los caracteres, dejaría espacio para el caracter nul y guardaría los cuatro caracteres en la memoria devolviendo su ubicación en el nombre del array, que en este caso sería **mi_nombre**.

En algún código, en lugar de lo anterior, podrías ver:

```
char *mi_nombre = "Ted";
```

que es una alternativa. ¿Hay alguna diferencia entre ambos? La respuesta es.. si. Utilizando la notación de array se toman 4 bytes de almacenamiento en un bloque estático de memoria, uno para cada carácter y uno para el carácter de terminación nul. En la notación de punteros se requieren esos mismos 4 bytes **más** N bytes para guardar la variable puntero **mi_nombre** (donde N depende del sistema, y por lo general tendrá un mínimo de 2 bytes y puede ser de 4 bytes o más).

En la notación de array, **mi_nombre** es la abreviación de **&mi_nombre[0]**, que es la dirección del primer elemento del array. Como el array tiene una ubicación fija en tiempo de ejecución, el array es una constante (no una variable). En la notación de punteros **mi_nombre** es una variable. Dependiendo de lo que quieras hacer con el resto del programa, un método será **mejor** que el otro.

Demos un paso más y veamos qué ocurre si cada una de estas declaraciones se realizan dentro de una función, en lugar de realizarlas globalmente.

```
void mi_funcion_A(char *ptr)
{
    char a[] = "ABCDE"
    .
    .
}


void mi_funcion_B(char *ptr)
{
    char *cp = "FGHIJ"
    .
    .
}
```

En el caso de **mi_funcion_A**, el dato es el contenido (los valores del array **a[]**) y este array se inicializa con los valores ABCDE. En el caso de **mi_funcion_B**, el dato es el valor del puntero **cp**. El puntero se inicializa para que apunte a la cadena **FGHIJ**. Tanto en **mi_funcion_A** como en **mi_funcion_B** se definen variables locales y por tanto la cadena **ABCDE** y el valor del puntero **cp** se guardan en la pila. La cadena **FGHIJ** puede guardarse en cualquier lugar (en mi sistema se guarda en el segmento de datos).

Por cierto, la inicialización de arrays con variables automáticas como vemos en **mi_funcion_A** no se permite en el antiguo C de K&R y sólo se permite en la "mayoría de edad" del nuevo ANSI C. Esto puede ser importante en casos donde sea necesaria la portabilidad y la compatibilidad hacia atrás.

Mientras hablamos de las similitudes y las diferencias entre punteros y arrays, vamos a ver los arrays de varias dimensiones. Consideremos por ejemplo el array:

```
char multi[5][10];
```

¿Qué significa esto? Bien, veámoslo desde la siguiente perspectiva:

```
char multi[5][10];
```

Ignoremos la parte subrayada que es el "nombre" del array y nos queda **char** y **[10]** que sería un array de 10 caracteres. El nombre **multi[5]** en sí es un array indicando que hay 5 elementos donde cada uno es un array de 10 caracteres. Por tanto, tenemos un array de 5 arrays de 10 caracteres cada uno..

Asumamos que rellenamos este array de dos dimensiones con datos de algún tipo. En memoria parecería como si los arrays hubieran sido inicializados por separado mediante algo así:

```
multi[0] = {'0','1','2','3','4','5','6','7','8','9'}
multi[1] = {'a','b','c','d','e','f','g','h','i','j'}
multi[2] = {'A','B','C','D','E','F','G','H','I','J'}
multi[3] = {'9','8','7','6','5','4','3','2','1','0'}
multi[4] = {'J','I','H','G','F','E','D','C','B','A'}
```

Podríamos tener acceso individual a cada elemento con la siguiente sintáxis:

```
multi[0][3] = '3'
multi[1][7] = 'h'
multi[4][0] = 'J'
```

Como los arrays se escriben en memoria de forma contigua, nuestro bloque de memoria para lo anterior debería ser:

```
0123456789abcdefghijABCDEFGHIJ9876543210JIHGFEDCBA
^
|_____ comienzo de la dirección de &multi[0][0]
```

Fíjate que **no** he escrito **multi[0] = "0123456789"**, porque si lo hubiera hecho, se habría agregado la terminación **'\0'** a los carácteres que van dentro de las comillas dobles y tendría 11 caracteres por línea en lugar de 10.

Mi objetivo aquí es mostrar cuanta memoria se utiliza para arrays de dos dimensiones, esto es, un array de dos dimensiones de caracteres, NO un array de "cadenas".

Ahora el compilador sabe cuantas columnas tiene el array y puede interpretar a **multi + 1** como la dirección de la 'a' ubicada en la segunda línea, esto es, agrega 10 que es el número de columnas para llegar a la dirección de 'a'. Si hubiéramos utilizado enteros en este array en lugar de caracteres, el compilador agregaría **10*sizeof(int)** que, en mi máquina sería 20. Por tanto, la dirección del valor **9** ubicado en la cuarta línea sería **&multi[3][0]** o ***(multi + 3)** en notación de punteros, y para llegar al contenido del segundo elemento de la cuarta línea agregaríamos 1 a esta dirección y desreferenciaríamos el resultado así:

```
*(*(multi + 3) + 1)
```

Pensando un poco vemos que:

```
*(*(multi + row) + col)     y
multi[row][col]             llegan al mismo resultado.
```

El siguiente programa muestra esto utilizando arrays de enteros en lugar de arrays de caracteres.

```
------------------- programa 6.1 ----------------------
/* Programa 6.1 de PTRTUT10.HTM   6/13/97*/

#include <stdio.h>
#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];

int main(void)
{
    int row, col;
    for (row = 0; row < ROWS; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            multi[row][col] = row*col;
        }
    }

    for (row = 0; row < ROWS; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            printf("\n%d  ",multi[row][col]);
            printf("%d ",*(*(multi + row) + col));
        }
    }

    return 0;
}
----------------- fin del programa 6.1 ---------------------
```

Debido a que en la versión con punteros es necesaria una doble desreferenciación, con frecuencia se dice que el nombre de un array de dos dimensiones es lo mismo que un puntero a un puntero. En un array de tres dimensiones tendríamos un array de arrays de arrays y algunos dicen que su nombre es lo mismo que un puntero a un puntero a un puntero. Sin embargo, aquí hemos inicializado el bloque de memoria utilizando la notación de arrays, por tanto el array es una constante, no una variable. Es decir, estamos hablando de una dirección fija en memoria, no de un puntero que puede apuntar de forma variable. La desreferencia permite acceder a cualquier elemento del array de arrays sin cambiar el valor de esa dirección (tanto la dirección de **multi[0][0]** como la dirección del símbolo **multi**).

CAPÍTULO 7: Más sobre Arrays Multidimensionales

En el capítulo anterior vimos que lo siguiente

```
#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];
```

permite acceder a los elementos del array **multi** mediante:

```
multi[row][col]
```

o mediante:

```
*(*(multi + row) + col)
```

Para entenderlo mejor, cambiemos

```
*(multi + row)
```

por **X** como en:

```
*(X + col)
```

Ahora vemos que **X** es como un puntero porque la expresión es desreferenciada y sabemos que **col** es un entero. Aquí estamos utilizando "aritmética de punteros", y al tratarse de un array de enteros, la dirección donde apunta (por ej. el valor de) **X + col + 1** debe ser mayor que la dirección **X + col** por una diferencia de **sizeof(int)**.

Ya sabemos como se guarda en memoria un array de dos dimensiones, y por tanto podemos determinar que en la expresión **multi + row** anterior, **multi + row + 1** debe incrementarse por el valor necesario para "apuntar" a la siguiente línea, que en este caso es **COLS * sizeof(int)**.

Esto quiere decir que para la evaluación correcta en tiempo de ejecución de la expresión ***(*(multi + row) + col)** el compilador debe generar el código teniendo en cuenta el valor de **COLS** (la segunda dimensión). Tanto la versión con punteros como la versión con arrays **multi[row][col]** son equivalentes.

Por tanto, para evaluar la expresión dada, se necesitan 5 valores:

1. La dirección del primer elemento del array, que es el valor devuelto por la expresión **multi**, es decir, el nombre del array.
2. El tamaño del tipo de los elementos del array, en este caso **sizeof(int)**.
3. La segunda dimensión del array .
4. El valor específico del índice de la primera dimensión, que en este caso es **row**.
5. El valor específico del índice de la segunda dimensión, que en este caso es **col**.

Con todo esto, vamos a diseñar una función para manipular los valores de los elementos del array declarado anteriormente. Por ejemplo, podemos iniciar todos los elementos del array **multi** con el valor 1.

```
void set_value(int m_array[][COLS])
{
    int row, col;
    for (row = 0; row < ROWS; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            m_array[row][col] = 1;
        }
    }
}
```

Y para llamar a esta función, podemos utilizar:

```
set_value(multi);
```

Ahora, dentro de la función hemos utilizado los valores de ROWS y COLS definidos al principio con #define ROWS 5 y #define COLS 10 para establecer los límites de los bucles for. Para el compilador estas definiciones son constantes. Tanto **row** como **col** son variables locales. La definición formal de los parámetros permiten al compilador determinar las características del valor del puntero que será pasado en tiempo de ejecución. Realmente no necesitamos la primera dimensión, y como veremos después, hay momentos donde es preferible no definirlo dentro de la definición de parámetros (por hábito o por consistencia, yo no lo he utilizado aquí). Ahora, la segunda dimensión debe utilizarse como vemos en la expresión del parámetro. La razón es que lo necesitamos en la evaluación de **m_array[row][col]** como hemos visto. Como el parámetro define el tipo de dato (en este caso **int**), y las variables automáticas (row y col) se definen dentro de la función para los bucles for, con un único parámetro sólo puede pasarse un valor. En este caso, éste parámetro es el valor de **multi** como vemos en la llamada a la función, que es la dirección del primer elemento, a menudo también llamado como un puntero a un array. Por tanto, la única forma que tenemos de indicar la segunda dimensión al compilador es incluirlo explícitamente en la definición del parámetro.

De hecho, generalmente todas las dimensiones superiores a una son necesarias en arrays multidimensionales. Esto es: si hablamos de 3 dimensiones, la dimensión 2 y 3 deben especificarse en la definición de parámetros.

CAPÍTULO 8: Punteros a Arrays

Los punteros pueden "apuntar a" cualquier tipo de dato, arrays incluidos. En el programa 3.1 vimos que esto es evidente, pero es importante entenderlo mejor cuando tratamos con arrays multidimensionales.

Como repaso, en el Capítulo 2 vimos que un puntero de tipo entero puede apuntar a un array de enteros, así:

```
int *ptr;
ptr = &my_array[0];    /* el puntero apunta al
                          primer elemento del array */
```

Como vemos, el tipo de la variable puntero debe ser igual al tipo del primer elemento del array.

Además, podemos utilizar un puntero como parámetro formal de una función diseñada para manipular un array, es decir:

```
int array[3] = {1, 5, 7};
void a_func(int *p);
```

Algunos programadores prefieren escribir el prototipo de esta función como:

```
void a_func(int p[]);
```

para informar a cualquiera que utilice esta función de que está diseñada para manipular elementos de un array. En ambos casos lo que se pasa es el valor de un puntero apuntando al primer elemento del array, independientemente de la notación utilizada en el prototipo o definición de la función. Si se utiliza la notación de array no es necesario pasar la dimensión actual del array porque no estamos pasando el array completo, sólo estamos pasando una dirección a su primer elemento.

Volvamos al problema del array de dos dimensiones. Como vimos en el capítulo anterior, C interpreta un array de dos dimensiones como dos arrays de una dimensión. El primer elemento de un array de enteros de dos dimensiones es un array de enteros de una dimensión. Un puntero que apunte a un array de enteros de dos dimensiones debe ser un puntero de ese tipo de dato. Una forma de lograr esto es mediante el uso de la palabra clave "typedef", que asigna un nombre nuevo a un tipo de dato especificado. Por ejemplo:

```
typedef unsigned char byte;
```

hace que el nombre **byte** sea el tipo **unsigned char** (caracter sin signo). Esto es:

```
byte b[10];     es un array de caracteres sin signo.
```

Fíjate en la declaración de typedef: la palabra **byte** es sustituida por lo que sería el nombre de nuestro tipo **caracter sin signo**. Esto es, la regla para utilizar **typedef** es que el nuevo nombre para el tipo de dato es el nombre utilizado en la definición del tipo de dato. De esta forma:

```
typedef int Array[10];
```

Array será un tipo de dato para un array de 10 enteros, es decir, **Array mi_arr;** declara **mi_arr** como un array de 10 enteros y **Array arr2d[5];** declara **arr2d** como un array de 5 arrays de 10 enteros cada uno.

Fíjate también que **Array *p1d;** hace que **p1d** sea un puntero a un array de 10 enteros. Debido a que ***p1d** apunta al mismo tipo que **arr2d**, **p1d** puede apuntar a la dirección de un array de dos dimensiones **arr2d** como a un array de una dimensión de 10 enteros, es decir, es correcto tanto

```
p1d = &arr2d[0];
```

como

```
p1d = arr2d;
```

Como el tipo de dato de nuestro puntero es un array de 10 enteros, podemos esperar que al incrementar **p1d** por 1 éste cambie su valor por **10*sizeof(int)**, y así es porque **sizeof(*p1d)** sería 20. Puedes probar esto escribiendo y ejecutando un pequeño y simple programa.

Ahora, utilizar typedef no hace las cosas más fáciles al programador ni más claras para el lector. Lo que necesitamos es declarar un puntero como **p1d** sin la palabra clave **typedef**. Resulta que esto se puede hacer y que

```
int (*p1d)[10];
```

es la declaración apropiada, es decir, aquí **p1d** es un puntero a un array de 10 enteros como lo es con la declaración mediante el tipo Array. Ten en cuenta que esto es distinto de

```
int *p1d[10];
```

pues esto hace que **p1d** sea el nombre de un array de 10 punteros de tipo **int**.

CAPÍTULO 9: Punteros y Asignación Dinámica de Memoria

Hay momentos donde es conveniente asignar memoria en tiempo de ejecución mediante **malloc()**, **calloc()**, u otras funciones de asignación. Este enfoque permite aplazar la decisión del tamaño del bloque de memoria necesario para guardar un array en tiempo de ejecución, o también permite disponer de una sección de memoria para guardar un array de enteros en un momento dado, y después, cuando esa memoria no sea necesaria, podría ser liberada para otros usos, como el almacenamiento de un array de estructuras.

Cuando se asigna memoria, la función de asignación (como **malloc()**, **calloc()**, etc.) devuelve un puntero. El tipo de este puntero depende de si utilizas un compilador antiguo como K&R o un más nuevo de tipo ANSI. Con el compilador antiguo, el puntero devuelto es de tipo **char**, y con el compilador ANSI es **void**.

Si utilizas un compilador antiguo y quieres asignar memoria para un array de enteros, necesitarás realizar una conversión (cast) del puntero devuelto a un puntero entero. Por ejemplo, para asignar espacio para 10 enteros podemos escribir:

```
int *iptr;
iptr = (int *)malloc(10 * sizeof(int));
if (iptr == NULL)

{ .. AQUÍ VA LA RUTINA DE ERROR .. }
```

Si utilizas un compilador ANSI, **malloc()** devuelve un puntero **void** y como un puntero void puede ser asignado a una variable puntero de cualquier tipo, no es necesaria la conversión a **(int *)** que hemos visto. La dimensión del array puede determinarse en tiempo de ejecución y no es necesaria en tiempo de compilación, esto es, el **10** que hemos visto podría ser una variable leída de un archivo o teclado, o calculada en tiempo de ejecución.

Como la notación de arrays y punteros es equivalente, una vez que **iptr** ha sido asignado como hemos visto, puede utilizarse la notación de arrays. Por ejemplo, podemos escribir:

```
int k;
for (k = 0; k < 10; k++)
   iptr[k] = 2;
```

para iniciar todos los elementos con el valor 2.

Incluso con una buena comprensión de punteros y arrays, es posible que un principiante en C tropiece primero con la asignación dinámica de arrays multidimensionales. En general y siempre que nos sea posible, nos gustaría acceder a los elementos de estos arrays mediante la notación de array en lugar de la de punteros. Dependiendo de la aplicación, podemos conocer o no ambas dimensiones en tiempo de compilación. A partir de aquí se abren varios caminos para seguir con la tarea.

Como hemos visto, cuando se asigna de forma dinámica un array de una dimensión, se puede determinar su dimensión en tiempo de ejecución. Ahora, cuando utilizamos asignación dinámica de memoria de arrays con más de una dimensión, no es necesario conocer la primera dimensión en tiempo de compilación. Dependiendo de cómo escribamos el código, necesitaremos conocer o no el resto de dimensiones. Aquí voy a mostrar varios métodos de asignación dinámica de espacio para arrays de dos dimensiones de enteros.

En primer lugar vamos a considerar los casos donde se conoce la segunda dimensión en tiempo de compilación.

MÉTODO 1:

Una forma de abordar el problema es utilizando **typedef**. Recordemos que para asignar un array de dos dimensiones de enteros podemos utilizar cualquiera de las dos notaciones siguientes, pues ambas generan el mismo código objeto:

```
multi[row][col] = 1;      *(*(multi + row) + col) = 1;
```

También es cierto que las siguientes dos anotaciones generan el mismo código:

```
multi[row]            *(multi + row)
```

La notación de la derecha se evalúa como un puntero, y la notación de la izquierda también se evalúa como un puntero. De hecho, **multi[0]** devuelve un puntero al primer entero de la primera fila (row), **multi[1]** devuelve un puntero al primer entero de la segunda fila, etc. En realidad **multi[n]** se evalúa como un puntero a un array de enteros que componen la fila n-ésima de nuestro array de dos dimensiones. Podemos pensar que **multi** es un array de arrays y **multi[n]** es un puntero que apunta al n-ésimo elemento de este array de arrays. Aquí la palabra **puntero** se utiliza para representar un valor de dirección. Aunque esto es bastante común, hay que poner especial cuidado al leer estas declaraciones para distinguir entre la dirección constante de un array y un puntero variable que es un objeto de datos en sí.

Veamos ahora el siguiente programa:

```
--------------- Programa 9.1 --------------------------------

/* Programa 9.1 de PTRTUT10.HTM  6/13/97 */

#include <stdio.h>
#include <stdlib.h>

#define COLS 5

typedef int RowArray[COLS];
RowArray *rptr;

int main(void)
{
    int nrows = 10;
    int row, col;
    rptr = malloc(nrows * COLS * sizeof(int));
    for (row = 0; row < nrows; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            rptr[row][col] = 17;
        }
    }

    return 0;
}
------------- Fin del Prog. 9.1 --------------------------------
```

Aquí asumimos que utilizamos un compilador ANSI y por tanto no es necesaria la conversión del puntero void devuelto por **malloc()**. Si utilizas un compilador antiguo como K&R necesitarás realizar la conversión así:

```
rptr = (RowArray *)malloc(.... etc.
```

Con este enfoque, **rptr** cumple con todas las características de un nombre de array (excepto que rptr se puede modificar), y se puede utilizar la notación de array en el resto del programa. Esto también significa que si quieres escribir una función para modificar los elementos de un array, debes utilizar COLS como parte del parámetro formal de esa función, tal y como lo hicimos cuando vimos cómo pasar un array de dos dimensiones a una función.

MÉTODO 2:

En el MÉTODO 1 anterior, rptr es un puntero de tipo "array de una dimensión de COLS enteros". También podemos utilizar otra sintáxis para este tipo:

```
int (*xptr)[COLS];
```

la variable **xptr** tendrá las mismas características que la variable **rptr** del MÉTODO 1 anterior, sin necesidad de utilizar **typedef**. Aquí, **xptr** es un puntero a un array de enteros y el tamaño de este array viene dado por los COLS definidos. La ubicación de los paréntesis hace que prevalezca la notación de puntero aunque pensemos que la notación de array tiene mayor precedencia, es decir, si escribimos

```
int *xptr[COLS];
```

estaríamos definiendo a **xptr** como un array de punteros con un número COLS de punteros. Como vemos, esto no es lo mismo que lo anterior. De cualquier modo, los arrays de punteros se utilizan en la asignación de memoria dinámica de arrays de dos dimensiones, como veremos en los dos siguientes métodos.

MÉTODO 3:

Supongamos que no conocemos el número de elementos en cada fila en tiempo de compilación, es decir, en tiempo de ejecución debemos determinar el número de filas y el número de columnas. Una forma de hacer esto es creando un array de punteros de tipo **int**, después asignar espacio para cada fila y hacer que esos punteros apunten a cada fila. Veámoslo en este programa:

```
-------------- Programa 9.2 -----------------------------------

/* Programa 9.2 de PTRTUT10.HTM   6/13/97 */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int nrows = 5;     /* Tanto nrows como ncols deben asignarse */
    int ncols = 10;    /* o ser leídas en tiempo de ejecución */
    int row;
    int **rowptr;
    rowptr = malloc(nrows * sizeof(int *));
    if (rowptr == NULL)
    {
        puts("\nError asignando espacio para los punteros de filas.\n");
        exit(0);
    }

    printf("\n\n\nIndice   Puntero(hex)   Puntero(dec)   Dif.(dec)");

    for (row = 0; row < nrows; row++)
    {
        rowptr[row] = malloc(ncols * sizeof(int));
        if (rowptr[row] == NULL)
        {
            printf("\nError asignando memoria para row[%d]\n",row);
            exit(0);
        }
        printf("\n%d         %p         %d", row, rowptr[row], rowptr[row]);
        if (row > 0)
        printf("              %d",(int)(rowptr[row] - rowptr[row-1]));
    }

    return 0;
}

-------------- Fin 9.2 -----------------------------------
```

En el código anterior **rowptr** es un puntero que apunta al tipo **int**. En este caso apunta al primer elemento de un array de punteros de tipo **int**. Consideremos el número de llamadas a **malloc()**:

```
    Para conseguir el array de punteros      1     llamada
    Para conseguir espacio para las filas    5     llamadas
                                            -----
                    Total                    6     llamadas
```

En este caso ten en cuenta que puedes utilizar la notación de arrays para acceder a los elementos de un array individualmente, por ej. **rowptr[row][col] = 17;**, pero esto no quiere decir que los datos del "array de dos dimensiones" estén ubicados de forma contigua en memoria.

De todas formas puedes utilizar la notación de arrays como si se tratara de un bloque contiguo de memoria. Por ejemplo, puedes escribir:

```
    rowptr[row][col] = 176;
```

como si rowptr fuera el nombre de un array de dos dimensiones creado en tiempo de compilación. Obviamente, tanto **row** como **col** deben figurar dentro de los límites del array creado, al igual que un array creado en tiempo de compilación.

Si lo que quieres es un bloque contiguo de memoria dedicado a almacenar los elementos de un array, puedes hacer lo siguiente:

MÉTODO 4:

En este método asignamos un bloque de memoria para almacenar primero a todo el array. Después creamos un array de punteros para que cada puntero apunte a cada fila del primer array. De esta forma, aunque se utilice el array de punteros, el array realmente está en un bloque contiguo de memoria. El código sería como el siguiente:

```
----------------- Programa 9.3 ---------------------------------

/* Programa 9.3 de PTRTUT10.HTM   6/13/97 */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int **rptr;
    int *aptr;
    int *testptr;
    int k;
    int nrows = 5;     /* Tanto nrows como ncols deben asignarse */
    int ncols = 8;     /* o ser leídas en tiempo de ejecución */
    int row, col;

    /* ahora asignamos memoria para el array */

    aptr = malloc(nrows * ncols * sizeof(int));
    if (aptr == NULL)
    {
        puts("\nError al asignar memoria para el array");
        exit(0);
    }

    /* asignamos espacio para los punteros que apuntan a las filas */

    rptr = malloc(nrows * sizeof(int *));
    if (rptr == NULL)
    {
        puts("\nError asignado memoria para los punteros");
        exit(0);
    }
```

```
    /* y ahora hacemos que los punteros 'apunten' */

    for (k = 0; k < nrows; k++)
    {
        rptr[k] = aptr + (k * ncols);
    }

    /* Ahora vemos cómo se incrementan los punteros de las filas */
    printf("\n\nIncremento de los punteros de las filas");
    printf("\n\nIndice   Puntero(hex)  Dif.(dec)");

    for (row = 0; row < nrows; row++)
    {
        printf("\n%d         %p", row, rptr[row]);
        if (row > 0)
        printf("              %d",(rptr[row] - rptr[row-1]));
    }
    printf("\n\nY ahora mostramos el array\n");
    for (row = 0; row < nrows; row++)
    {
        for (col = 0; col < ncols; col++)
        {
            rptr[row][col] = row + col;
            printf("%d ", rptr[row][col]);
        }
        putchar('\n');
    }

    puts("\n");

    /* y aquí mostramos lo que realmente es: un array de
       2 dimensiones en un bloque contiguo de memoria. */
    printf("Demostracion de que el array esta en un bloque contiguo de
memoria:\n");

    testptr = aptr;
    for (row = 0; row < nrows; row++)
    {
        for (col = 0; col < ncols; col++)
        {
            printf("%d ", *(testptr++));
        }
        putchar('\n');
    }

    return 0;
}

------------- Fin Programa 9.3 ----------------
```

Consideremos de nuevo el número de llamadas a malloc()

```
    Para conseguir el array en sí         1      llamada
    Para el espacio del array de ptrs     1      llamada
                                         ----
                         Total            2      llamadas
```

Ahora, cada llamada a **malloc()** crea espacio adicional puesto que **malloc()** normalmente se implementa mediante el sistema operativo en forma de lista enlazada conteniendo el tamaño del bloque. Pero es más importante todavía, con arrays grandes (varios cientos de filas), utilizar un registro para liberar espacio antes de que se haga más complicado hacerlo. Esto, junto a la contigüidad del bloque de datos que permite su inicialización a ceros utilizando **memset()** parece ser la segunda alternativa preferida.

Como ejemplo final de arrays multidimensionales veremos la asignación dinámica de un array de tres dimensiones. Este ejemplo nos enseña una cosa más cuando hacemos este tipo de asignación. Por las razones antes citadas utilizaremos el enfoque de la segunda alternativa. Consideremos el siguiente código:

```
------------------- Programa 9.4 -----------------------------------

/* Programa 9.4 de PTRTUT10.HTM   6/13/97 */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

int X_DIM=16;
int Y_DIM=5;
int Z_DIM=3;

int main(void)
{
    char *space;
    char ***Arr3D;
    int y, z;
    ptrdiff_t diff;

    /* primero asignamos espacio para el array en sí */

    space = malloc(X_DIM * Y_DIM * Z_DIM * sizeof(char));

    /* asignamos espacio para un array de punteros, cada
       puntero apuntará eventualmente al primer elemento de un
       array de 2 dimensiones de punteros a punteros */

    Arr3D = malloc(Z_DIM * sizeof(char **));

    /* y por cada uno de ellos asignamos un puntero a un nuevo
       array asignado de punteros que apunta a filas */

    for (z = 0; z < Z_DIM; z++)
    {
        Arr3D[z] = malloc(Y_DIM * sizeof(char *));

        /* y por cada espacio en este array ponemos un puntero
           apuntando al primer elemento de cada fila en el espacio
           del array originalmente asignado */

        for (y = 0; y < Y_DIM; y++)
        {
            Arr3D[z][y] = space + (z*(X_DIM * Y_DIM) + y*X_DIM);
        }
    }

    /* Y ahora, comprobamos cada dirección en nuestro array 3D para
       ver si la indexación del puntero Arr3d se realiza de forma
       contigua */
```

```
    for (z = 0; z < Z_DIM; z++)
    {
        printf("La ubicacion del array %d es %p\n", z, *Arr3D[z]);
        for ( y = 0; y < Y_DIM; y++)
        {
            printf("  Array %d y Fila %d comienzan en %p", z, y, Arr3D[z][y]);
            diff = Arr3D[z][y] - space;
            printf("    diff = %d  ",diff);
            printf(" z = %d  y = %d\n", z, y);
        }
    }
    return 0;
}

------------------- Fin del Programa 9.4 -------------------------
```

Si has seguido este tutorial hasta aquí, no tendrás problemas en descifrar el programa anterior leyendo sus comentarios. De todas formas hay que ver un par de puntos. Comencemos con esta línea:

```
    Arr3D[z][y] = space + (z*(X_DIM * Y_DIM) + y*X_DIM);
```

Fíjate aquí que **space** es un puntero de tipo caracter, y tiene el mismo tipo que **Arr3D[z][y]**. Es importante saber que cuando agregamos un entero, como el obtenido evaluando la expresión **(z*(X_DIM * Y_DIM) + y*X_DIM)**, a un puntero, el resultado es un nuevo valor puntero. Y cuando asignamos valores puntero a variables puntero, el tipo de dato del valor y el tipo de dato de la variable tiene que ser el mismo.

CAPÍTULO 10: Punteros a Funciones

Hasta ahora hemos visto punteros que apuntan a objetos de datos, pero C también permite declarar punteros para apuntar a funciones. Este tipo de punteros tienen gran variedad de usos y vamos a ver algunos de ellos.

Considera el siguiente problema: quieres escribir una función para ordenar virtualmente cualquier colección de datos que pueda guardarse en un array. Esto podría ser un array de cadenas, o de enteros, o de floats, o incluso de estructuras. El algoritmo de ordenación puede ser para todos el mismo. Por ejemplo, podría ser un algoritmo de ordenación de burbuja simple, o de tipo shell (más complicado) o el algoritmo de ordenación quick sort. Para nuestro propósito, aquí veremos una ordenación de burbuja simple.

Sedgewick [1] escribió en C la ordenación de burbuja utilizando una función que, pasándola un puntero a un array, lo ordena. Vamos a ver la función **bubble()** en el siguiente programa bubble_1.c:

```
/*-------------------- bubble_1.c --------------------*/
/* Programa bubble_1.c de PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int a[], int N);

int main(void)
{
    int i;
    putchar('\n');
    for (i = 0; i < 10; i++){
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++){
        printf("%d ", arr[i]);
    }
    return 0;
}

void bubble(int a[], int N)
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (a[j-1] > a[j])
            {
                t = a[j-1];
                a[j-1] = a[j];
                a[j] = t;
            }
        }
    }
}

/*---------------------- fin bubble_1.c ----------------------*/
```

La ordenación de burbuja es una de las más simples. El algoritmo escanea el array desde el segundo hasta el último elemento, comparando cada uno con el que le precede. Si el predecesor es mayor al actual elemento, se intercambian quedando el mayor más cerca del fin del array. El resultado de la primera pasada es que el elemento mayor queda el último en el array. El array ahora queda limitado a todos los elementos menos el último y se repite el proceso. En la siguiente pasada, el siguiente elemento mayor quedará como predecesor del elemento mayor anterior. El proceso se repite un número de veces igual al número de elementos menos uno. El resultado es un array ordenado.

Nuestra función está diseñada para ordenar un array de enteros, primero comparando los enteros y después utiliza un entero temporal para realizar el intercambio. Lo que vamos a ver ahora es si podemos convertir este código para utilizar cualquier tipo de dato, no sólo enteros.

A la vez, no queremos analizar nuestro algoritmo y el código correspondiente cada vez que queramos utilizarlo. Empezaremos creando una nueva función de comparación sustituyendo la parte de la comparación dentro de la función **bubble()** con la llamada a la nueva función, así no hará falta reescribir las partes relacionadas del algoritmo actual. El resultado lo vemos en bubble_2.c:

```
/*---------------------- bubble_2.c -------------------------*/

/* Programa bubble_2.c de PTRTUT10.HTM   6/13/97 */

/* Separación de la función de comparación */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int a[], int N);
int compare(int m, int n);

int main(void)
{
    int i;
    putchar('\n');
    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```



```
void bubble(int a[], int N)
```

```
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare(a[j-1], a[j]))
            {
                t = a[j-1];
                a[j-1] = a[j];
                a[j] = t;
            }
        }
    }
}

int compare(int m, int n)
{
    return (m > n);
}

/*---------------------- fin de bubble_2.c -----------------------*/
```

Si el objetivo es hacer que nuestra rutina de ordenación sea independiente del tipo de datos, una forma de hacerlo es utilizar punteros de tipo void para apuntar a los datos en lugar de utilizar el tipo de dato entero. Para ello, modifiquemos algunas cosas del código anterior para incorporar punteros, empezando con los de tipo entero.

```
/*----------------------- bubble_3.c -------------------------*/

/* Programa bubble_3.c de PTRTUT10.HTM    6/13/97 */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int *p, int N);
int compare(int *m, int *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```



```
void bubble(int *p, int N)
```

```
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare(&p[j-1], &p[j]))
            {
                t = p[j-1];
                p[j-1] = p[j];
                p[j] = t;
            }
        }
    }
}

int compare(int *m, int *n)
{
    return (*m > *n);
}

/*------------------ fin de bubble3.c -------------------------*/
```

Fíjate en los cambios. Ahora estamos pasando un puntero de tipo entero (o un array de enteros) a **bubble()**. Dentro de la función bubble estamos pasando punteros a los elementos del array que queremos comparar con la función de comparación. Y, por supuesto, desreferenciamos estos punteros en nuestra función **compare()** para realizar la comparación actual. Lo siguiente será convertir los punteros en **bubble()** a punteros de tipo void para que la función no tenga en cuenta el tipo. Esto lo vemos en bubble_4.

```
/*------------------ bubble_4.c ----------------------------*/

/* Programa bubble_4.c de PTRTUT10,HTM   6/13/97 */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int *p, int N);
int compare(void *m, void *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}

void bubble(int *p, int N)
```

```
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare((void *)&p[j-1], (void *)&p[j]))
            {
                t = p[j-1];
                p[j-1] = p[j];
                p[j] = t;
            }
        }
    }
}

int compare(void *m, void *n)
{
    int *m1, *n1;
    m1 = (int *)m;
    n1 = (int *)n;
    return (*m1 > *n1);
}

/*------------------ fin de bubble_4.c ----------------------*/
```

En **compare()** hemos realizado una conversión del puntero de tipo void al actual tipo que estamos ordenando (int *), y como veremos después, esto es correcto. Como estamos pasando un puntero que apunta a un array de enteros (int *p) como parámetro a **bubble()**, tenemos que convertir esos punteros a void cuando los pasamos a **compare()**.

Como queremos que el primer parámetro de **bubble()** sea también un puntero de tipo void, esto implica que dentro de **bubble()** hay que hacer algo con la variable **t**, que actualmente es un entero. Además, cuando usamos **t = p[j-1];** es necesario conocer el tipo de **p[j-1]** para saber cuantos bytes hay que copiar a la variable **t** (o la que utilicemos para realizar el reemplazo).

Actualmente en bubble_4.c, dentro de **bubble()** sabemos que el tipo de dato para ordenar (y por tanto el tamaño de cada elemento individual) se obtiene a partir del hecho de que el primer parámetro es un puntero de tipo entero. Si queremos utilizar **bubble()** con cualquier tipo de dato, necesitamos que ese puntero sea de tipo **void**, pero al hacerlo, vamos a perder información relativa al tamaño de los elementos individuales del array. Por ello, en bubble_5.c vamos a agregar un parámetro independiente para manejar esta información.

Los cambios que tiene bubble_5.c respecto a bubble_4.c son, quizás, más numerosos que los realizados antes. Por ello es conveniente comparar con cuidado los dos módulos para dar con las diferencias.

```
/*---------------------- bubble_5.c --------------------------*/

/* Programa bubble_5.c de PTRTUT10.HTM    6/13/97 */

#include <stdio.h>
#include <string.h>

long arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(void *p, size_t width, int N);
int compare(void *m, void *n);
```



```
int main(void)
```

```
{
    int i;
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr, sizeof(long), 10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%ld ", arr[i]);
    }

    return 0;
}

void bubble(void *p, size_t width, int N)
{
    int i, j;
    unsigned char buf[4];
    unsigned char *bp = p;

    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare((void *)(bp + width*(j-1)),
                        (void *)(bp + j*width)))  /* 1 */
            {
/*              t = p[j-1];   */
                memcpy(buf, bp + width*(j-1), width);
/*              p[j-1] = p[j];   */
                memcpy(bp + width*(j-1), bp + j*width , width);
/*              p[j] = t;   */
                memcpy(bp + j*width, buf, width);
            }
        }
    }
}

int compare(void *m, void *n)
{
    long *m1, *n1;
    m1 = (long *)m;
    n1 = (long *)n;
    return (*m1 > *n1);
}

/*---------------------- fin de bubble_5.c ----------------------*/
```

He cambiado el tipo de datos del array de **int** a **long** para ver los cambios necesarios en la función **compare()**. Dentro de **bubble()** he eliminado la variable **t** (lo que hubiera supuesto cambiarla de tipo **int** al tipo **long**). He añadido un buffer con un tamaño de 4 caracteres sin signo para alojar un long (esto cambiará en las futuras modificaciones de este código). El puntero caracter sin signo ***bp** se utiliza para apuntar a la base del array a ordenar, es decir, al primer elemento de ese array.

También he modificado lo que se pasa a **compare()**, y cómo se realiza el cambio de elementos que indica la comparación. El uso de **memcpy()** y la notación de punteros en lugar de la notación de arrays permite reducir la sensibilidad al tipo.

De nuevo, para tener una mejor compresión de lo que sucede y porqué, es necesaria una atenta comparación entre bubble_5.c y bubble_4.c

Ahora vamos a ver a bubble_6.c donde utilizaremos la misma función bubble() de bubble_5.c, pero para ordenar cadenas en lugar de enteros largos (long). Obviamente tenemos que cambiar la función de comparación, pues la comparación entre cadenas y la comparación entre enteros largos es distinta. Además, en bubble_6.c hemos eliminado las líneas dentro de **bubble()** que estaban comentadas en bubble_5.c.

```
/*---------------------- bubble_6.c ----------------------*/

/* Programa bubble_6.c de PTRTUT10.HTM   6/13/97 */

#include <stdio.h>
#include <string.h>

#define MAX_BUF 256

char arr2[5][20] = {  "Mickey Mouse",

                      "Donald Duck",

                      "Minnie Mouse",

                      "Goofy",

                      "Ted Jensen" };

void bubble(void *p, int width, int N);
int compare(void *m, void *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 5; i++)
    {
        printf("%s\n", arr2[i]);
    }
    bubble(arr2, 20, 5);
    putchar('\n\n');

    for (i = 0; i < 5; i++)
    {
        printf("%s\n", arr2[i]);
    }
    return 0;
}
```

```
void bubble(void *p, int width, int N)
{
    int i, j, k;
    unsigned char buf[MAX_BUF];
    unsigned char *bp = p;

    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
          k = compare((void *)(bp + width*(j-1)), (void *)(bp + j*width));
          if (k > 0)
            {
             memcpy(buf, bp + width*(j-1), width);
             memcpy(bp + width*(j-1), bp + j*width , width);
             memcpy(bp + j*width, buf, width);
            }
        }
    }
}

int compare(void *m, void *n)
{
    char *m1 = m;
    char *n1 = n;
    return (strcmp(m1,n1));
}

/*------------------- fin de bubble_6.c ----------------------*/
```

Como **bubble()** no tiene modificaciones desde bubble_5.c, esta función puede ordenar una gran variedad de tipos de datos. Lo que queda por hacer es pasar a **bubble()** el nombre de la función de comparación que queremos usar para que pueda ser verdaderamente universal. Como el nombre de un array es la dirección de su primer elemento en el segmento de datos, el nombre de una función está dentro de la dirección de esa función en el segmento de código. Por lo tanto necesitamos utilizar un puntero a una función, que en este caso es la función de comparación.

Los punteros a funciones deben coincidir con el número y tipos de parámetros y el tipo del valor de retorno de la función a donde apuntan. En nuestro caso declaramos nuestro puntero a función así:

```
int (*fptr)(const void *p1, const void *p2);
```

Fíjate que si escribimos:

```
int *fptr(const void *p1, const void *p2);
```

tendríamos un prototipo de función para una función que devuelve un puntero de tipo **int**. Esto se produce porque en C, el operador paréntesis () tiene una precedencia mayor que el operador de puntero *. Indicamos que es una declaración de un puntero a función colocando los paréntesis alrededor de la cadena (*fptr).

Ahora modificaremos nuestra declaración de **bubble()** agregando como cuarto parámetro un puntero a una función del tipo apropiado. El prototipo de la función sería:

```
    void bubble(void *p, int width, int N,
                int(*fptr)(const void *, const void *));
```

Cuando llamamos a **bubble()**, insertamos el nombre de la función de comparación que queramos usar. El código de bubble_7.c muestra cómo esta forma permite utilizar la misma función **bubble()** para ordenar diferentes tipos de datos.

```
/*------------------- bubble_7.c ------------------*/
/* Programa bubble_7.c de PTRTUT10.HTM  6/10/97 */

#include <stdio.h>
#include <string.h>

#define MAX_BUF 256

long arr[10] = { 3,6,1,2,3,8,4,1,7,2};
char arr2[5][20] = {  "Mickey Mouse",
                      "Donald Duck",
                      "Minnie Mouse",
                      "Goofy",
                      "Ted Jensen" };

void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *));
int compare_string(const void *m, const void *n);
int compare_long(const void *m, const void *n);

int main(void)
{
    int i;
    puts("\nBefore Sorting:\n");

    for (i = 0; i < 10; i++)               /* muestra enteros largos */
    {
        printf("%ld ",arr[i]);
    }
    puts("\n");

    for (i = 0; i < 5; i++)                /* muestra cadenas */
    {
        printf("%s\n", arr2[i]);
    }
    bubble(arr, 4, 10, compare_long);         /* ordena enteros largos */
    bubble(arr2, 20, 5, compare_string);      /* ordena cadenas */
    puts("\n\nAfter Sorting:\n");

    for (i = 0; i < 10; i++)             /* muestra enteros largos ordenados */
    {
        printf("%d ",arr[i]);
    }
    puts("\n");

    for (i = 0; i < 5; i++)                /* muestra cadenas ordenadas */
    {
        printf("%s\n", arr2[i]);
    }
    return 0;
}
```

```
void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *))
{
    int i, j, k;
    unsigned char buf[MAX_BUF];
    unsigned char *bp = p;

    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            k = fptr((void *)(bp + width*(j-1)), (void *)(bp + j*width));
            if (k > 0)
            {
                memcpy(buf, bp + width*(j-1), width);
                memcpy(bp + width*(j-1), bp + j*width , width);
                memcpy(bp + j*width, buf, width);
            }
        }
    }
}

int compare_string(const void *m, const void *n)
{
    char *m1 = (char *)m;
    char *n1 = (char *)n;
    return (strcmp(m1,n1));
}

int compare_long(const void *m, const void *n)
{
    long *m1, *n1;
    m1 = (long *)m;
    n1 = (long *)n;
    return (*m1 > *n1);
}

/*----------------- fin de bubble_7.c -----------------*/
```

Referencias del Capítulo 10:

1. "Algorithms in C"
   Robert Sedgewick
   Addison-Wesley
   ISBN 0-201-51425-7

EPÍLOGO

He escrito este material para ofrecer una introducción a los principiantes en C sobre los punteros. En C tendrás mucha más flexibilidad escribiendo código cuantos más conocimientos tengas sobre punteros. Lo que has visto es mi primer esfuerzo sobre ello y que se titula ptr_help.txt, y lo podrás encontrar en una primera versión de Bob Stout's collection of C code SNIPPETS. El contenido de esta versión ha sido actualizada en PTRTUTOT.ZIP incluido en SNIP9510.ZIP.

Siempre estaré dispuesto a aceptar críticas constructivas de este material o solicitudes de revisión para ampliarlo con más cosas relevantes. Por ello, si tienes preguntas, comentarios, críticas, etc. relacionadas con este material, agradecería mucho que te pusieras en contacto conmigo a través de mi correo electrónico tjensen@ix.netcom.com.