TUTORIAL SOBRE APUNTADORES Y ARREGLOS EN C

por Ted Jensen

Versión 1.2

Febrero de 2000

Disponible en diferentes formatos en:
http://www.netcom.com/~tjensen/ptr/cpoint.htm

– CONTENIDO –

Traducido al español por Marte Baquerizo
martemorfosis@yahoo.com.mx

Universidad de los Altos de Chiapas.
México.
Junio de 2003

PREFACIO

Este documento pretende dar una introducción sobre apuntadores a los programadores novatos del lenguaje C. Después de varios años de leer y de contribuir en varias conferencias de C, incluyendo aquellas en FidoNet y UseNet, he notado que un buen número de principiantes en C presentan dificultades en comprender los fundamentos sobre apuntadores. Es por esto que me he dado a la tarea de tratar de explicarlos en un lenguaje simple y con un montón de ejemplos.

La primera versión de este documento se otorgó al dominio público, al igual que esta. Fue recogida por Bob Stout quien la incluyó como un archivo de nombre PTR-HELP.TXT en su ampliamente distribuida colección de SNIPPETS. Desde esa edición original de 1995, he añadido una cantidad significativa de material y corregido algunos pequeños errores.

En la versión 1.1 de HTML hice algunas correcciones en el manejo de terminología como resultado de los comentarios que he recibido de todas partes del mundo. En la versión 1.2 he actualizado los primeros 2 capítulos para hacer notar el cambio de 16 a 32 bits en los compiladores para PC's.

Reconocimientos:

Son tantos los que sin saberlo han contribuido a este trabajo debido a las preguntas que han publicado en FidoNet C Echo, o en el grupo de noticias de UseNet comp.lang.c, o en muchas otras conferencias en otras redes, que sería imposible hacer una lista de todos ellos. Agradecimientos especiales a Bob Stout quien fue tan amable en incluir la primera versión de este archivo en sus SNIPPETS.

Sobre el Autor:

Ted Jensen es un Ingeniero en Electrónica retirado que ha trabajado tanto como diseñador de hardware o gerente de diseñadores de hardware en el campo de almacenamiento magnético. La programación ha sido uno de sus pasatiempos desde 1968 cuando aprendió a perforar tarjetas para ser ejecutadas en un mainframe. (¡La mainframe tenía 64Kb de memoria magnética!).

Uso de este Material:

Todo lo que se encuentra contenido en este documento es liberado al dominio público. Cualquier persona es libre de copiar o distribuir este material en la manera que prefiera. Lo único que pido, en caso de que este material sea usado como material de apoyo en una clase, es que fuera distribuido en su totalidad, es decir, incluyendo todos los capítulos, el prefacio y la introducción. También apreciaría que en ese caso, el instructor de la clase me mandara una nota a alguna de las direcciones de abajo informándome al respecto. Escribí esto con la esperanza de que fuese útil a otros y es por eso que no solicito remuneración económica alguna, el único modo de enterarme en que he alcanzado este objetivo es a través de los comentarios de quienes han encontrado útil este material.

No tienes que ser un instructor o maestro para contactarte conmigo. Apreciaría mucho un mensaje de cualquier persona que encuentre útil este material, o de quien tenga alguna crítica constructiva que ofrecer. También espero poder contestar las preguntas enviadas por e-mail.

Ted Jensen
Redwood City, CA 94064

tjensen@ix.netcom.com

Febrero de 2000.

INTRODUCCION

Si uno quiere ser eficiente escribiendo código en el lenguaje de programación C, se debe tener un profundo y activo conocimiento del uso de los apuntadores. Desafortunadamente, los apuntadores en C parecen representar una piedra en el camino de los principiantes, particularmente de aquellos que vienen de otros lenguajes de programación como Fortran, Pascal o Basic.

Es para ayudar a estos principiantes en el uso de apuntadores que he escrito el siguiente material. Para obtener el máximo beneficio del mismo, siento que es necesario que el usuario sea capaz de ejecutar el código fuente que se incluye en los artículos. Debido a esto he intentado mantener todo el código dentro de las especificaciones del ANSI[1] para que este pueda trabajar en cualquier compilador compatible con ANSI. He tratado de dar formato al código dentro del texto de tal manera que con la ayuda de un editor de texto ASCII uno pueda copiar el bloque de código que interesa a un archivo nuevo y compilarlo en su sistema. Recomiendo a los lectores el hacerlo porque esto es de gran ayuda para la comprensión del material.

[1]ANSI: American National Standards Institute (Instituto Nacional Americano de Estándares), definió a través del comité X3J11 formado en 1982, el estándar del lenguaje C en 1989 y de sus funciones de librería. Esto debido a que surgieron varias versiones del lenguaje que diferían en cuanto a características y extensiones hechas al mismo. (Nota del traductor).

CAPITULO 1: ¿QUE ES UN APUNTADOR?

Una de las cosas más difíciles que encuentran los principiantes en C es entender el concepto de apuntadores. El propósito de este documento es dar una introducción sobre apuntadores y de su uso a estos principiantes.

Me he encontrado a menudo que la principal razón por la que los principiantes tienen problemas con los apuntadores es que tienen una muy pobre o mínima concepción de las variables, (del modo en que C hace uso de ellas). Así que comencemos con una discusión sobre las variables de C en general.

Una variable en un programa es algo con un nombre, que contiene un valor que puede variar. El modo en que el compilador y el enlazador (linker) manejan esto es que asignan un bloque específico de la memoria dentro de la computadora para guardar el valor de una variable. El tamaño de este bloque depende del rango en que a esta variable le es permitido variar. Por ejemplo, en PC's de 32 bits, el tamaño de una variable de tipo entero (int) es de 4 bytes, en una máquina antigua de 16 bits los enteros tienen un tamaño de 2 bytes. En C el tamaño de un tipo de variable como una de tipo entero no tiene porqué ser el mismo en todos los tipos de máquinas. Es más en C disponemos de diferentes tipos de variables enteras, están los enteros largos (long int) y los enteros cortos (short int) sobre los que puedes averiguar en cualquier texto básico sobre C. El presente documento asume que se está usando un sistema de 32 bits con enteros de 4 bytes.

Si quieres conocer el tamaño de los diferentes tipos de enteros de tu sistema, ejecutar el siguiente código te dará la información.

```
#include <stdio.h>

int main()
{
 printf("El tamaño de short (entero corto), es: %d\n", sizeof(short));
 printf("El tamaño de int (entero), es: %d\n", sizeof(int));
 printf("El tamaño de long (entero largo), es: %d\n", sizeof(long));
}
```

Cuando declaramos una variable le informamos al compilador 2 cosas, el nombre de la variable y el tipo de la variable. Por ejemplo, declaramos una variable de tipo entero llamada **k** al escribir:

```
int k;
```

Cuando el compilador encuentra la palabra "int" de esta instrucción, reserva 4 bytes (en un PC) de memoria para almacenar el valor del entero. También construye una tabla de símbolos. Y en esa tabla agrega el símbolo **k** y la correspondiente dirección de la memoria en donde esos 4 bytes han sido reservados.

Así que si luego escribimos:

```
k = 2;
```

esperamos encontrar al momento de la ejecución, un 2 colocado en el área de memoria reservada para guardar el valor de **k**. En C nos referimos a una variable como la de tipo entero **k** como un "objeto"[2].

Tiene sentido decir que hay dos valores asociados con el objeto **k**, uno es el valor del entero alojado ahí (un 2 en el ejemplo de arriba) y el otro el "valor" de la localidad de la memoria donde se ha guardado, es decir, la dirección de **k**. Algunos textos se refieren a estos dos valores con la nomenclatura ***rvalue*** ("are value", –right value– valor a la derecha) y ***lvalue*** ("el value" –left value– valor a la izquierda).

En algunos lenguajes, el lvalue, es el valor que se permite a la izquierda del operador de asignación '=' (la dirección donde se alojará el resultado de la evaluación de la expresión). El rvalue es el que se encuentra a la derecha de la operación de asignación, el **2** de arriba. Los rvalues no pueden ser usados en la parte izquierda de una instrucción de asignación. Así que hacer algo como: **2 = k;** No es permitido**.**

En realidad, la definición de arriba para "lvalue" es modificada de algún modo para C, de acuerdo con K&R II (página 197) [1]:

"Un ***objeto*** es una región de almacenamiento; Un ***lvalue*** es una expresión que hace referencia a un *objeto*."

En este momento, nos basta la definición de arriba. A medida que nos vayan resultado familiares los apuntadores entraremos más a detalle con esto.

Bien, ahora consideremos:

```
int j, k;

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

En lo de arriba, el compilador interpreta la **j** en la línea 1 como la dirección de la variable **j** (su lvalue) y crea código para copiar el valor 7 a esa dirección. En la línea 2, sin embargo, la **j** es interpretada como su rvalue (ya que está del lado derecho del operador de asignación '='). Esto significa que aquí **j** hace referencia **al valor alojado** en la dirección de memoria asignado a **j**, 7 en este caso. Así que el 7 es copiado a la dirección designada por el *lvalue* de **k**.

En estos ejemplos hemos estado usando enteros de 4 bytes para almacenar números, así que al copiar rvalues de una dirección de alojamiento a otra es hecha copiando 4 bytes. Si estuviéramos usando enteros de dos bytes, estaríamos copiando 2 bytes en cada ocasión.

Ahora, digamos que por alguna razón queremos que una variable almacene un *lvalue* (una dirección). El tamaño requerido para almacenar un valor así depende del sistema. En ordenadores antiguos con 64Kb de memoria total, la dirección de cualquier lugar en la memoria puede ser contenida en 2 bytes. Computadores con más memoria pueden requerir de más bytes para almacenar una dirección. El tamaño real requerido no es muy importante mientras podamos decirle al compilador que lo que queremos almacenar es una dirección.

[2]Se usa la palabra objeto para referirse mas bien a un conjunto que consta de símbolo (nombre), valor y dirección. No se trata del paradigma de programación "objeto", usado en la programación orientada a objetos. (Nota del traductor).

Este tipo de variable es conocido como "variable apuntador" (por razones que esperamos resulten claras un poco más tarde). En C cuando definimos una variable de apuntador lo hacemos cuando precedemos su nombre con un asterisco. En C además le damos a nuestro apuntador un tipo, el cual, en este caso, hace referencia al tipo de dato que se encuentra guardado en la dirección que alojaremos en nuestro apuntador. Por ejemplo, consideremos la siguiente declaración de una variable:

```
int *ptr;
```

ptr es el *nombre* de nuestra variable (tal y como **k** era el nombre de nuestra variable de tipo entero).

El ***** informa al compilador que lo que queremos es una variable apuntador, es decir, que se reserven los bytes necesarios para alojar una dirección en la memoria. Lo de "int" significa que queremos usar nuestra variable apuntador para almacenar la dirección de un entero. Se dice entonces que dicho tipo de apuntador "apunta" a un entero. Sin embargo, nótese que cuando escribíamos **int k**; no le asignamos un valor a **k**.

Si la declaración se hace fuera de cualquier función, los compiladores ANSI la inicializarán automáticamente a cero. De modo similar, **ptr** no tiene un valor asignado, esto es, no hemos almacenado una dirección en la declaración hecha arriba. En este caso, otra vez si la declaración fuese hecha fuera de cualquier función, es inicializado a un valor garantizado de tal manera que no apunte a un objeto de C o a una función, un apuntador inicializado de este modo es definido como un apuntador "null". Entonces se llama un apuntador nulo (null pointer).

El patrón real de bits para un apuntador nulo, puede o no evaluarse a cero, ya que depende específicamente del sistema en que está siendo desarrollado el código. Para hacer el código fuente compatible entre distintos compiladores en varios sistemas, se usa una macro para representar un apuntador nulo. Este macro recibe el nombre de NULL. Así que, estableciendo el valor de un apuntador utilizando este macro, con una instrucción como ptr = NULL, garantiza que el apuntador sea un apuntador nulo. De modo similar cuando comprobamos el valor de cero en una variable entera, como en **if (k==0)** podemos comprobar un que un apuntador sea nulo usando **if (ptr == NULL).**

Pero, volviendo de nuevo con el uso de nuestra nueva variable **ptr**. Supongamos ahora que queremos almacenar en **ptr** la dirección de nuestra variable entera **k**. Para hacerlo hacemos uso del operador unitario **&** y escribimos:

```
ptr = &k;
```

Lo que el operador **&** hace es obtener la dirección de **k**, aún cuando **k** está en el lado derecho del operador de asignación '=' y copia esa dirección en el contenido de nuestro apuntador ptr. Ahora, ptr es un "puntero a **k**".

Hay un operador más que discutir: El operador de "indirección" (o de desreferencia) es el asterisco y se usa como sigue:

```
*ptr = 7;
```

esto copiará el 7 a la dirección a la que apunta ptr. Así que como ptr "apunta a" (contiene la dirección de) **k**, la instrucción de arriba asignará a **k** el valor de 7. Esto es, que cuando usemos el '*****' hacemos referencia al valor al que ptr está apuntando, no el valor de el apuntador en si.

De modo similar podríamos escribir:

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

para imprimir en la pantalla el valor entero que se encuentra alojado en la dirección a la que apunta "ptr".

Una manera de observar como todo esto encaja entre sí sería ejecutar el siguiente programa, revisar el código y la salida concienzudamente.

PROGRAMA 1.1

```
/* Program 1.1 from 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 esta alojado en: %p\n", j, (void *)&j);
    printf("k tiene el valor: %d y esta alojado en: %p\n", k, (void *)&k);
    printf("ptr tiene el valor: %p y esta alojado en: %p\n", ptr, (void *)&ptr);
    printf("El valor del entero al que apunta ptr es: %d\n", *ptr);

    return 0;
}
```

Nota: Aún tenemos que discutir los aspectos de C que requieren el uso de la expresión **`(void *)`** usada aquí. Por el momento inclúyela en el código de prueba. Ya explicaremos las razones de esta expresión más adelante.

Recordando:

- Una variable es declarada dándole un tipo y un nombre (por ejemplo: **int k;**)
- Una variable apuntador es declarada dándole un tipo y un nombre (por ejemplo: **int *ptr**) en donde el asterisco le dice al compilador que la variable de nombre ptr es una variable apuntador. Y el tipo le dice al compilador a que tipo de variable va a apuntar nuestro apuntador (int en este caso).

- Una vez que una variable ha sido declarada, podemos obtener su dirección anteponiendo a su nombre el operador unitario **&**, como en **&k**.
- Podemos “desreferenciar” un apuntador, es decir, hacer referencia al valor contenido en la dirección a la que apunta, usando el operador unitario ‘*****’, como en ***ptr**.
- Un "*lvalue*" de una variable es el valor de su dirección, es decir la posición (dirección) de memoria en donde se encuentra alojado el valor que contiene. El "*rvalue*" de una variable es el valor alojado en esa en esa dirección.

Referencias en el capítulo 1:

[1] B. Kernighan and D. Ritchie,

"The C Programming Language"

2nd Edition,

Prentice Hall.

ISBN 0-13-110362-8

CAPITULO 2: TIPOS DE APUNTADORES Y ARREGLOS

Consideremos el porqué tenemos que identificar el "tipo" de variable a la que apunta un puntero como en:

```
int *ptr;
```

Una de las razones para hacer esto es que una vez que ptr apunta a algo y si escribimos:

```
*ptr = 2;
```

El compilador sabrá cuantos bytes va a copiar en la posición de memoria a la que apunta **ptr**. Si **ptr** fuera declarado como un puntero a entero, se copiarían 4 bytes. De modo similar para números de punto flotante (float) y enteros dobles (doubles), se copiaría el número apropiado de bytes. Pero definir el tipo al que el apuntador apunta permite un cierto número de maneras interesantes en que el compilador puede interpretar el código. Por ejemplo, consideremos un bloque de memoria consistente en 10 números enteros en una fila. Eso es 40 bytes de memoria son reservados para colocar 10 enteros.

Digamos que ahora apuntamos nuestro apuntador entero **ptr** al primero de estos números enteros. Es más, supongamos que este primer entero está almacenado en la posición de memoria 100 (decimal). Entonces que pasa cuando escribimos:

```
ptr + 1;
```

Ya que el compilador “sabe” que este es un apuntador (que su valor es una dirección de memoria) y que apunta a un entero (su dirección actual: 100, es la dirección donde se aloja un entero), añade 4 a ptr en lugar de 1, así que ptr apunta **al siguiente entero**, en la posición de memoria 104. Similarmente, si ptr fuera declarado como apuntador a entero corto, añadiría 2 en lugar de 1. Lo mismo va para los otros tipos de datos como flotantes, dobles o aún tipos definidos por el usuario como estructuras. No se trata obviamente del tipo “normal” de “adición” a la que estamos acostumbrados. En C, se le conoce como adición usando “aritmética de punteros”, algo que veremos un poco más adelante.

Igualmente, como ++ptr y ptr++ son equivalentes a ptr + 1 (aunque el momento en el programa cuando ptr es incrementado sea diferente), incrementar un apuntador usando el operador unitario de incremento ++, ya sea pre- o post-, incrementa la dirección por la cantidad sizeof (tipo)[3] donde “tipo” es el tipo de objeto al que se apunta (por ejemplo 4 si se trata de un entero).

Y ya que un bloque de 10 enteros acomodados contiguamente en la memoria es, por definición, un arreglo de enteros, esto nos revela una interesante relación entre arreglos y apuntadores.

Consideremos lo siguiente:

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

[3] sizeof: Tamaño de. Palabra clave de C que en una instrucción calcula el tamaño en bytes de la expresión o tipo dados. (Nota del Traductor).

Tenemos entonces un arreglo conteniendo seis enteros. Nos referimos a cada uno de estos enteros por medio de un subíndice a **mi_arreglo**, es decir usando **`mi_arreglo[0]`** hasta **`mi_arreglo[5]`**. Pero podemos acceder a ellos de un modo alternativo usando un puntero de esta manera:

```
int *ptr;
ptr = &mi_arreglo[0];       /* apuntamos nuestro apuntador al
                              primer entero de nuestro arreglo */
```

Y entonces podemos imprimir los valores de nuestro arreglo, ya sea usando la notación de arreglos o "desreferenciando" nuestro apuntador.

El siguiente código ilustra este concepto.

PROGRAMA 2.1

```
#include <stdio.h>

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

int main(void)
{
    int i;
    ptr = &mi_arreglo[0];     /* apuntamos nuestro puntero
                                    al primer elemento del arreglo*/
    printf("\n\n");
    for (i = 0; i < 6; i++)
    {
      printf("mi_arreglo[%d] = %d   ", i, mi_arreglo[i]);   /*<-- A */
      printf("ptr + %d = %d\n",i, *(ptr + i));              /*<-- B */
    }
    return 0;
}
```

Compila y ejecuta el programa de arriba y nota como las líneas marcadas con A y B imprimen los mismos valores. Observa además como desreferenciamos nuestro puntero ptr en la línea B, primero le añadimos i y luego desreferenciamos el nuevo apuntador. Cambia la línea B de tal modo que quede:

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

Compila y ejecuta de nuevo, y luego vuelve a modificarla por:

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

Compila y ejecuta una vez más. Trata de predecir lo que saldrá a pantalla cada vez y revisa con cuidado la salida real.

En C, el estándar establece que donde usemos **&nombre_de_la_variable[0]** podemos reemplazarle con **nombre_de_la_variable**, esto en el código de ejemplo lo que escribimos como:

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

podemos escribirlo como:

```
ptr = mi_arreglo;
```

y obtenemos el mismo resultado.

Esto conduce a muchos textos a decir que el nombre de un arreglo es un apuntador. Si bien esto es cierto, prefiero pensar que "el nombre de un arreglo es la dirección del primer elemento que contiene el arreglo". Muchos principiantes (incluyéndome a mi cuando estuve aprendiendo) muestran tendencia a confundirse pensando en el como un puntero.

Por ejemplo, si bien podemos hacer `ptr = mi_arreglo;` no podemos hacer:

```
mi_arreglo = ptr;
```

La razón es que mientras **ptr** es una variable, **mi_arreglo** es una constante. Esto es, la dirección en la que el primer elemento de mi_arreglo será almacenado no puede ser cambiado una vez que **mi_arreglo[]** ha sido declarado.

Anteriormente , al discutir el término "lvalue", se estableció que:

> "Un ***objeto*** es una región de almacenamiento; Un ***lvalue*** es una expresión que hace referencia a un *objeto*."

Esto plantea un problema interesante. Ya que **mi_arreglo** es una región nominada de almacenamiento, ¿Por qué no es **mi_arreglo** en la asignación que se hace arriba el ***lvalue***?. Para resolver este problema, algunos se refieren a cosas como **mi_arreglo** como un "lvalue no modificable".

Modifiquemos el programa de ejemplo cambiando:

```
ptr = &mi_arreglo [0];      por      ptr = mi_arreglo;
```

ejecuta de nuevo para verificar que los resultados son idénticos.

Profundicemos un poco más con la diferencia entre los nombres **ptr** y **mi_arreglo** como hicimos arriba. Algunos escritores se refieren al nombre de un arreglo como un puntero ***constante***. ¿Qué queremos decir con esto? Bueno, para entender el término "constante" en este contexto, volvamos a nuestra definición del término "variable". Cuando declaramos una variable reservamos un lugar de la memoria para almacenar el valor del tipo apropiado. Una vez hecho esto, el nombre de la variable puede ser interpretado en una de dos maneras. Cuando es usada en el lado izquierdo del operador de asignación, el compilador la interpreta como la dirección de memoria en la cual colocar el resultado de la evaluación de lo que se encuentra al lado derecho del operador de asignación. Pero cuando se usa del lado derecho del operador de asignación, el nombre de una variable es

interpretada de modo que representa el contenido de la dirección de memoria reservada para contener el valor de dicha variable.
Con esto en mente analicemos la más simple de las constantes, como en:

```
int i, k;

i = 2;
```

Aquí, mientras **i** es una variable y ocupa un espacio en la sección de datos de la memoria, **2** es una constante y como tal, en lugar de ocupar memoria en el segmento de datos, es embebida directamente en la sección de código de la memoria. Esto quiere decir que cuando escribimos algo como **k = i;** le decimos al compilador que cree código que en el momento de ejecución observará en la dirección **&i** para determinar el valor a ser movido a **k**, mientras que el código creado al hacer algo como **i = 2**, simplemente pone el **2** en el código (no se hace referencia al segmento de datos). Esto es porque ambos, **k** e **i** son *objetos*, pero **2** no es un objeto.

De modo similar a lo que ocurre arriba, ya que **mi_arreglo** es una constante, una vez que el compilador establece donde será almacenado el arreglo en la memoria, ya "sabe" la dirección de **mi_arreglo[0**] y cuando encuentra:

```
ptr = mi_arreglo;
```

Simplemente usa esta dirección como una constante en el segmento de código y no hace más que eso (no se hace una referencia al segmento de código).

Este sería buen lugar para explicar el uso de la expresión **(void *)** usada en el programa 1.1 del capítulo 1. Como ya hemos visto, podemos tener apuntadores de distintos tipos. Es más, ya hemos discutido sobre apuntadores a enteros y a caracteres. En las siguientes lecciones veremos apuntadores a estructuras y aún apuntadores a apuntadores.

Ya hemos aprendido que en diferentes sistemas, el tamaño de un apuntador puede variar. También es posible que el tamaño de un apuntador varíe dependiendo del tipo de datos del objeto al que apunta. Así que al igual que con los enteros donde podemos tener problemas al asignar, por ejemplo un valor entero largo a una variable del tipo entero corto, podemos igualmente encontrarnos con problemas al intentar asignar valores desde un cierto tipo de apuntador a un apuntador de otro tipo.

Para reducir este problema C nos ofrece un apuntador de tipo void (carente de tipo). Podemos declarar un apuntador de este tipo al escribir algo como:

```
void *vptr;
```

Un apuntador void es una especie de apuntador genérico. Por ejemplo, mientras C no permite la comparación entre un apuntador del tipo entero con uno del tipo caracter, cada uno de estos puede ser comparado con un apuntador del tipo void.

Por supuesto, como con los otros tipos de variables, las conversiones (casts) pueden ser utilizadas para convertir un tipo de apuntador en otro bajo las circunstancias apropiadas. En el Programa 1.1 del capítulo 1, convertí los punteros a enteros a punteros void, para hacerlos compatibles con la especificación de conversión de %p. En capítulos posteriores, otras conversiones serán realizadas por las razones que se han expuesto aquí.

Bueno, han sido bastantes cosas técnicas que digerir y no espero que un principiante entienda todo esto en la primera lectura. Con tiempo y experimentación seguramente volverás a leer los primeros dos capítulos. Pero por ahora, continuemos con la relación existente entre apuntadores, arreglos de caracteres, y cadenas.

CAPITULO 3: APUNTADORES Y CADENAS

El estudio de las cadenas es útil para profundizar en la relación entre apuntadores y arreglos. Facilita, además la demostración de cómo algunas de las funciones estándar de cadenas de C pueden ser implementadas. Finalmente ilustraremos cómo y cuando los apuntadores pueden y deben ser pasados a una función.

En C, las cadenas son arreglos de caracteres. Esto no es necesariamente cierto para otros lenguajes. En Basic, Pascal, Fortran y en otros lenguajes, una cadena tiene definido su propio tipo de datos. Pero en C, esto no es así. En C una cadena es un arreglo de caracteres terminado con un carácter binario de cero (escrito como **\0**).

Para comenzar nuestra discusión escribiremos algo de código, el cual si bien es preferido para propósitos meramente ilustrativos, probablemente no lo escribirás en un programa real. Consideremos por ejemplo:

```
char mi_cadena[40];
mi_cadena [0] = 'T';
mi_cadena [1] = 'e';
mi_cadena [2] = 'd':
mi_cadena [3] = '\0';
```

Si bien uno nunca construiría cadenas de este modo, el resultado final es una cadena que es en realidad un arreglo de caracteres terminado con un caracter nul. Por definición, en C, una cadena es un arreglo de caracteres terminado con el carácter *nul*. Hay que tener cuidado con que **nul** no es lo mismo que **NULL**. El "nul" se refiere a un cero definido por la secuencia de escape \0'. Esto es, que ocupa un byte de memoria. El "NULL", por otra parte, es el nombre de la macro usada para inicializar apuntadores nulos. NULL está definido en un archivo de cabecera del compilador de C, mientras que nul puede no estar definido del todo.

Ya que al estar escribiendo código como el de arriba gastaríamos mucho tiempo, C permite dos modos alternativos de llegar al mismo resultado. El primero sería escribir:

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

Pero se lleva más tecleado del que es conveniente. Así que C permite:

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

Cuando usamos las comillas dobles, en lugar de las simples usadas en los ejemplos anteriores, el carácter nul ('\0') se añade automáticamente al final de la cadena.

En cualquiera de los casos descritos arriba sucede la misma cosa,. El compilador asigna un bloque continuo de memoria de 40 bytes de longitud para alojar los caracteres y los inicializa de tal manera que los primeros 4 caracteres son Ted\0.

Veamos ahora el siguiente programa:

PROGRAMA 3.1

```
/* Program 3.1 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

char strA[80] = "Cadena a usar para el programa de ejemplo";
char strB[80];

int main(void)
{

    char *pA;               /* un apuntador al tipo caracter */
    char *pB;               /* otro apuntador al tipo caracter */
    puts(strA);             /* muestra la cadena strA */
    pA = strA;              /* apunta pA a la cadena strA */
    puts(pA);               /* muestra a donde apunta pA  */
    pB = strB;              /* apunta pB a la cadena strB */
    putchar('\n');          /* dejamos una línea en blanco */
    while(*pA != '\0')      /* linea A (ver texto) */
    {
        *pB++ = *pA++;      /* linea B (ver texto) */
    }
    *pB = '\0';             /* linea C (ver texto) */
    puts(strB);             /* muestra strB en la pantalla */
    return 0;
}
```

Lo que hicimos arriba fue comenzar por definir dos arreglos de 80 caracteres cada uno. Ya que estos son definidos globalmente, son inicializados a **\0** primeramente. Luego **strA** tiene sus primeros 42 caracteres inicializados a la cadena que está entre comillas.

Ahora, yendo al código, declaramos dos apuntadores a caracter y mostramos la cadena en pantalla. Después apuntamos con el puntero **pA** a **strA**. Esto quiere decir que, por el significado de la operación de asignación, copiamos la dirección de memoria de **strA[0]** en nuestra variable apuntador **pA**. Usamos entonces **puts()** para mostrar lo que estamos apuntando con **pA** en la pantalla. Consideremos aquí que el prototipo de la función **puts()** es:

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

Por el momento ignoremos eso de "**const**". El parámetro pasado a **puts()** es un apuntador, esto es, el *valor* del un apuntador (ya que en C todos los parámetros son pasados por valor), y ya que el valor de un apuntador es la dirección de memoria a la que apunta, o , para decirlo simple: una dirección. Así que cuando escribimos: **`puts(strA);`** como hemos visto, estamos pasando la dirección de **strA[0]**.

De modo similar cuando hacemos: **`puts(pA);`** estamos pasando la misma dirección, ya que habíamos establecido que **pA = strA;**

Sigamos examinando el código hasta el **while()** en la línea A:

- La línea A indica: “Mientras el caracter apuntado por **pA** (es decir: ***pA**) no sea un caracter nul (el que es ‘\0’), haz lo siguiente”

- La línea B indica: “copia el caracter apuntado por **pA** (es decir ***pA**) al espacio al que apunta **pB**, luego incrementa **pA** de tal manera que apunte al siguiente caracter, de igual modo incrementa **pB** de manera que apunte al siguiente espacio”

Una vez que hemos copiado el último caracter, **pA** apunta ahora a un caracter nul de terminación de cadena y el ciclo termina. Sin embargo, no hemos copiado el caracter de terminación de cadena. Y, por definición: una cadena en C debe terminar en un caracter *nul.* Así que agregamos nul con la línea C.

Resulta realmente didáctico ejecutar este programa en un depurador (debugger), mientras se observa **strA, strB, pA** y **pB** e ir recorriendo cada paso del programa, también es bueno probar inicializando **strB[]** a una cadena en lugar de hacerlo simplemente declarándole; puede ser algo como:

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

Donde el número de dígitos sea mayor que la longitud de **strA** y luego repite la observación paso a paso mientras observas el contenido de las variables. ¡Inténtalo!

Volviendo al prototipo para **puts()** por un momento, la “const” usada como parámetro informa al usuario que la función no modificará a la cadena apuntada por **s**, es decir que se tratará a esa cadena como una constante.

Desde luego, lo que hace el programa de arriba es una manera simple de copiar una cadena. Después de jugar un poco con esto y una vez que tengas bien entendido lo que pasa, procederemos entonces a crear nuestro reemplazo para la función estándar **strcpy()** que viene con C.

Sería algo como:

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

En este caso he seguido el procedimiento usado en la rutina estándar de regresar un puntero al destino.

De nuevo, la función esta diseñada para aceptar valores de dos punteros a caracter, es decir las direcciones, y por esto en el programa anterior pudimos haber escrito:

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

Me he desviado ligeramente de la forma usada en la función estándar de C, la cual tiene por prototipo:

```
char *mi_strcpy(char *destino, const char *fuente);
```

Aquí el modificador "const" es usado para asegurar que la función no modificará el contenido al que apunta el puntero de la fuente. Puedes probar esto modificando la función de arriba y su prototipo incluyendo el modificador "const" como hemos mostrado. Entonces dentro de la función puedes agregar código que intente cambiar el contenido de lo que está apuntado por la fuente. Algo como: `*fuente = 'X';` Lo que normalmente cambiaría el primer caracter de la cadena por una 'X'. El modificador constante hace que el compilador detecte esto como un error. Prueba y verás.

Consideremos ahora algunas de las cosas que el ejemplo de arriba nos ha demostrado. En primera, consideremos el hecho de que ***ptr++** se interpreta como que devuelve el valor apuntado por **ptr** y luego incrementa el valor del apuntador. Esto tiene que ver con la precedencia de operadores. Si hubiéramos escrito **(*ptr)++** no estaríamos incrementando el apuntador, ¡sino lo que contiene!. Es decir que si lo usáramos así con el primer caracter de la cadena del ejemplo, la 'C' sería incrementada a una 'D'. Puedes escribir código sencillo para demostrar esto.

Recordemos de nuevo que una cadena no es otra cosa más que un arreglo de caracteres, siendo su último caracter un '\0'. Lo que hemos hechos es ideal para copiar un arreglo. Sucede que lo hemos hecho con un arreglo de caracteres, pero la misma técnica puede ser aplicada a un arreglo de enteros, dobles, etc. En estos casos, no estaríamos tratando con cadenas y entonces el arreglo no tendría porque estar marcado por un valor especial como el caracter nul. Podemos implementar una versión que se basara en una terminación especial para identificar el final. Por ejemplo, podríamos copiar un arreglo de enteros positivos y señalar el final con un entero negativo. Por otra parte, es más usual que cuando escribamos una función que copie un arreglo que no sea una cadena, indiquemos el número de elementos que serán copiados así como la dirección del arreglo. Por ejemplo algo como lo que indicaría el siguiente prototipo:

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

Donde **nbr** es el número de elementos enteros que serán copiados. Puedes juguetear un poco con esta idea y crear un arreglo enteros y probar si puedes implementar una función copiadora de enteros **int_copy()** y hacer que trabaje bien.

Esto permite el uso de funciones para manipular arreglos grandes. Por ejemplo, si tuviéramos un arreglo de 5000 enteros que quisiéramos manipular con una función, sólo necesitaríamos pasarle a esa función la dirección donde se encuentra alojado el arreglo (y alguna información auxiliar como el nbr de arriba, dependiendo de lo que vayamos a hacer). El arreglo en sí **no** es pasado a la función, es decir, este no se copia y se pone en la pila (stack) antes de llamar a la función. Sólo se envía la dirección donde se encuentra su primer elemento (la dirección de arreglo es pasada a la función) .

Esto es diferente de pasar, digamos, un entero a una función (una variable int). Cuando nosotros pasamos una variable entera a una función, lo que sucede es que hacemos una copia de este entero, es decir, obtenemos su valor y lo ponemos en la pila. Dentro de la función cualquier manipulación que se haga a esta variable no afectará al contenido de la variable original. Pero con arreglos y apuntadores podemos pasar a una función las direcciones de las variables y por tanto manipular los valores contenidos en las variables originales.

CAPITULO 4: MÁS SOBRE CADENAS

Bien, hemos progresado bastante en corto tiempo. Retrocedamos un poco y veamos lo que hicimos en el capítulo 3 al copiar cadenas, pero viéndolo desde otra perspectiva. Consideremos la siguiente función:

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

Recordemos que las cadenas son arreglos de caracteres. Aquí hemos elegido usar la notación de arreglos en lugar de la notación de punteros para hacer la copia. El resultado es el mismo, esto es, la cadena es copiada de la misma manera usando esta notación que como lo hizo antes. Esto expone algunos puntos interesantes que discutir.

Ya que los parámetros son pasados por valor, ya sea pasando el apuntador de tipo caracter que apunta a la dirección del arreglo o el nombre del arreglo como arriba, lo que en realidad se está pasando es la dirección del primer elemento del arreglo para cada caso. Esto significa que el valor numérico del parámetro que se pasa es el mismo ya sea que pasemos un puntero tipo caracter o el nombre del arreglo como parámetro. Esto implicaría de alguna manera que: **`fuente[i]`** es lo mismo que **`*(p+i);`**

Lo cual es cierto, dondequiera que uno escriba **`a[i]`** se puede reemplazar por **`*(a + i)`** sin ningún problema. De hecho el compilador creará el mismo código en cualquier caso. Por esto nos damos cuenta que la aritmética de punteros es lo mismo que usar subíndices con los arreglos. Cualquiera de las dos sintaxis produce el mismo resultado.

Esto NO significa que apuntadores y arreglos sean lo mismo, porque no es así. Sólo estamos exponiendo que para identificar un elemento dado de un arreglo tenemos la opción de usar dos sintaxis diferentes, una usando subíndices para arreglos y la otra es usando aritmética de punteros, lo cual conduce a un mismo resultado.

Ahora, analizando la última expresión, la parte de ella que dice : **`(a + i)`**, es una simple adición usando el operador + y las reglas de C dicen que una expresión así es conmutativa. Por lo que **`(a + i)`** es lo mismo que: **`(i + a)`**, así que podemos escribir: **`*(i + a)`** al igual que **`*(a + i)`**.

¡ Pero **`*(i + a)`** pudo venir de **`i[a]`** ! De todo esto obtenemos una verdad que resulta curiosa tal que si:

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

Escribir: **a[3] = 'x';**

Es lo mismo que: **3[a] = 'x';**

¡Pruébalo! Inicializa un arreglo de caracteres, enteros, largos, etc. Y usando el método convencional asigna un valor al 3er o 4to elemento e imprime ese valor para confirmar que esta almacenado. Luego invierte la notación como se ha hecho arriba. Un buen compilador no se quejará con un mensaje de error o de advertencia y se obtendrán los mismos resultados. Una curiosidad… ¡y nada más!

Ahora volviendo a nuestra función de más arriba, escribimos:

```
destino[i] = fuente[i];
```

Debido al hecho de que usar subíndices en arreglos o aritmética de punteros lleva a un mismo resultado, pudimos escribir lo anterior como:

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

Pero, esto implica 2 adiciones por cada valor que toma i. Las adiciones, generalmente hablando, se llevan más tiempo que los incrementos (como los hechos usando ++ en i++). Esto no es necesariamente cierto con los modernos compiladores optimizados, pero uno no siempre puede estar seguro. Así que es posible que la versión con punteros sea un poco más rápida que la versión con arreglos.

Otro método para acelerar la versión de punteros sería cambiar:

`while (*fuente != '\0')` a simplemente `while (*fuente)`

Ya que el valor dentro del paréntesis se hará cero (FALSO) al mismo tiempo en cualquiera de los dos casos.

Llegado este momento tal vez quieras experimentar un poco escribiendo tus propios programas usando apuntadores. Manipular cadenas de texto es una buena idea para experimentar. Tal vez puedas implementar tu propia versión de las funciones de la librería estándar <string.h> como:

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

y de cualquiera otras que tuvieras en tu sistema.

Ya volveremos a ver cadenas y su manipulación a través de punteros en un capítulo futuro. Por ahora sigamos avanzando y discutamos un poco sobre las estructuras.

CAPITULO 5: APUNTADORES Y ESTRUCTURAS

Como sabrás, es posible declarar la forma de un bloque de datos conteniendo distintos tipos de datos por medio de la declaración de una estructura. Por ejemplo, un archivo de personal contendría estructuras que serían algo como:

```
struct ficha{
     char nombre[20];         /* nombre */
     char apellido[20];       /* apellido */
     int  edad;               /* edad */
     float salario;           /* por ejemplo 12.75 por hora */
     };
```

Supongamos que tenemos muchas de estas estructuras en un archivo de disco y queremos leer cada una e imprimir el nombre y apellido de cada una, de modo que tengamos una lista con el nombre y apellido de cada persona que se encuentra en el archivo. La información restante no se imprimirá. Queremos hacer esto por medio de una función a la que pasemos como parámetro un apuntador a la estructura. Para propósitos didácticos sólo usaré una estructura por ahora. Pero concentrémonos en que el objetivo es implementar la función, no leer desde un archivo de disco, lo que presumiblemente, sabemos cómo hacer.

Recordemos que podemos acceder a los miembros de una estructura por medio del operador '.' (punto).

PROGRAMA 5.1

```
/* Program 5.1 from PTRTUT10.HTM     6/13/97 */

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

   struct ficha{
        char nombre[20];         /* nombre */
        char apellido[20];       /* apellido */
        int  edad;               /* edad */
        float salario;           /* salario */
        };

struct ficha  mi_ficha;     /* declaramos mi_ficha como una
                               estructura del tipo ficha */
int main(void)
{
   strcpy(mi_ficha.nombre,"Jensen");
   strcpy(mi_ficha.apellido,"Ted");

   printf("\n%s ",mi_ficha.nombre);
   printf("%s\n",mi_ficha.apellido);

   return 0;
}
```

Ahora que esta estructura en particular es muy pequeña comparada con aquellas usadas en muchos programas de C. A la de arriba tal vez quisiéramos añadir (sin mostrar el tipo de datos en particular):

-Dirección	-Teléfono	-Código Postal
-Ciudad	-Estado Civil	-Nº del seguro social,… etc.

Si tenemos una cantidad considerable de empleados, lo ideal sería manejar los datos dentro de estas estructuras por medio de funciones. Por ejemplo queremos implementar una función que imprimiera el nombre de los empleados, contenidos en cualquier estructura que le pasara. Sin embargo, en el lenguaje C original (Kernighan & Ritchie, 1ª Edición) no era posible pasar una estructura como parámetro a una función, sólo un puntero que apuntara a una estructura. En el ANSI C, ahora es posible pasar una estructura completa. Pero, ya que nuestro objetivo es aprender sobre punteros, no iremos tras esto ahora.

De cualquier modo, si pasáramos la estructura completa significa que debemos copiar el contenido de la estructura desde la función que llama a la función llamada. En sistemas que usan pilas (stacks), esto se hace metiendo los contenidos de la estructura dentro de la pila. Con estructuras grandes esto puede representar un problema. Sin embargo pasar apuntadores usa un mínimo de espacio en la pila.

Como sea, ya que estamos discutiendo apuntadores, discutiremos entonces como pasarle a una función un puntero que apunta a una estructura y cómo usarlo dentro de la función.

Considera el caso descrito: queremos una función que acepte como parámetro un puntero a una estructura y dentro de esa función queremos acceder a los miembros de la estructura. Por ejemplo, queremos imprimir el nombre del empleado de nuestra estructura de ejemplo.

Bien, como sabemos que nuestro apuntador va a apuntar a una estructura declarada usando struct ficha. Declaramos dicho apuntador con la declaración:

```
struct ficha  *st_ptr;
```

Y hacemos que apunte a nuestra estructura de ejemplo con:

```
st_ptr = &mi_ficha;
```

Ahora podremos acceder a un miembro de la estructura desreferenciando el puntero. Pero, ¿Cómo desreferenciamos un puntero a estructura? Bueno, consideremos el hecho de que queramos usar el puntero para cambiar la edad del empleado. Para esto escribiríamos:

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

Observa cuidadosamente. Dice, reemplaza lo que se encuentra entre paréntesis por aquello a lo que **st_ptr** está apuntando, lo cual es la estructura **mi_ficha**. Así que esto se reduce a lo mismo que **mi_ficha.edad**.

Sin embargo, esta notación no es muy usada y los creadores de C nos han brindado la posibilidad de utilizar una sintaxis alternativa y con el mismo significado, la cual sería:

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

Con esto en mente, veamos el siguiente programa.

PROGRAMA 5.2

```
/* Program 5.2 from PTRTUT10.HTM   6/13/97 */

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


   struct ficha{
        char nombre[20];         /* nombre */
        char apellido[20];       /* apellido */
        int  edad;               /* edad */
        float salario;           /* salario */
        };


struct ficha mi_ficha;         /* definimos mi_ficha del tipo
                                        estructura ficha */

void show_name (struct tag *p);    /* prototipo de la función */


int main(void)
{
  struct ficha *st_ptr;            /* un apuntador a una estructura
                                        del tipo ficha */

  st_ptr = &mi_ficha;              /* apuntamos el apuntador a mi_ficha */


   strcpy(mi_ficha.apellido,"Jensen");
   strcpy(mi_ficha.nombre,"Ted");

   printf("\n%s ",mi_ficha.nombre);
   printf("%s\n",mi_ficha.apellido);

   mi_ficha.edad = 63;


   show_name (st_ptr);          /* Llamamos a la función pasándole el puntero */


   return 0;
}
```

```
void show_name(struct tag *p)
{
   printf("\n%s ", p -> nombre);     /* p apunta a una estructura */
   printf("%s ", p -> apellido);
   printf("%d\n", p -> edad);
}
```

De nuevo, esta es mucha información para absorber de una sola vez. Sugiero al lector compilar y ejecutar los programas y de ser posible usar un depurador (debugger) para ver el estado de las variables ejecutando el programa paso por paso a través de la función principal (main()) y siguiendo el código hasta la función show_name() para ver que es lo que sucede.

CAPITULO 6: MÁS SOBRE CADENAS Y ARREGLOS DE CADENAS

Bien, regresemos con las cadenas. En lo consiguiente todas las declaraciones se entenderán como hechas globalmente. Es decir, son hechas fuera de cualquier función, incluyendo main().

Habíamos dicho en un capítulo anterior que podíamos hacer:

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

Con lo cual reservaríamos espacio para alojar un arreglo de 40 bytes y colocar la cadena dentro de los primeros 4 bytes del arreglo (3 para los caracteres entre comillas y uno más para '**\0**')

Si realmente sólo quisiéramos un arreglo donde alojar el nombre "Ted", podemos hacer:

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

Y el compilador contaría los caracteres, dejando espacio para el caracter de terminación nul y entonces guardará un total de 4 caracteres en la memoria, la dirección de la cual sería regresada por el nombre del arreglo, en este caso **mi_nombre**.

En ocasiones, en lugar de código como el de arriba encontraremos:

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

Lo cual es una declaración alterna. ¿Existe alguna diferencia entre ellas? La respuesta es… si. Usando la notación de arreglo 4 bytes de almacenamiento en el bloque de memoria estática son tomados, uno para cada caracter y uno para el caracter de terminación nul.

Pero en la notación de apuntador los mismos 4 bytes son requeridos *más* N bytes para alojar la variable apuntadora mi_nombre (donde N depende del sistema pero es usualmente un mínimo de 2 bytes y pueden ser 4 o más).

En la notación de arreglo, "**mi_nombre**" es la forma corta de **&mi_nombre[0]** lo cual es la dirección del primer elemento del arreglo. Ya que la dirección en que se alojará el arreglo será fijada durante el tiempo de ejecución del programa, esto es una constante (no una variable).

En la notación de punteros, **mi_nombre** es una variable. Decidir el método a utilizar y cual es el mejor depende de lo que se vaya a hacer en el resto del programa.

Vayamos ahora un paso más adelante y consideremos que pasaría si cada una de estas declaraciones fueran hechas dentro de una función, de manera opuesta a lo global que es fuera de los dominios de cualquier función:

```
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 contenido, o valor(es), del arreglo **a[]** son considerados como los datos. Se dice que el arreglo ha sido inicializado a los valores ABCDE. En el caso de **mi_funcion_B**, el valor del apuntador **cp** se considera como dato. El puntero ha sido inicializado para apuntar a la cadena **FGHIJ**. En ambas funciones las definiciones son variables locales y por tanto la cadena **ABCDE** se guarda en la pila (stack), así como el valor del apuntador **cp**. La cadena **FGHIJ** se guarda en cualquier lugar que resulte adecuado. En mi sistema se guarda en el segmento de datos.

Dicho sea de paso, la inicialización por arreglo de variables automáticas como se hizo en mi_funcion_A era ilegal en el viejo C de K&R y solo "vino a darse" en el nuevo ANSI C. Un factor que puede ser importante cuando se esta considerando la portabilidad y la compatibilidad "hacia atrás".

A la vez que vamos discutiendo las relaciones/diferencias entre apuntadores y arreglos, veamos algo sobre arreglos multidimensionales. Consideremos por ejemplo el arreglo:

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

¿Qué hay con eso? Bien, veámoslo con otra perspectiva:

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

Tomemos como el "nombre" del arreglo la parte subrayada. Luego entonces anteponiéndole **char** y posponiendo **[10]** tenemos entonces un arreglo de 10 caracteres. Pero el nombre **multi[5]** es en si un arreglo indicando que consta de 5 elementos los cuales a su vez constan de 10 caracteres cada uno. Por tanto tenemos un arreglo de 5 arreglos de 10 caracteres cada uno.

Asumamos que hemos rellenado este arreglo bidimensional. En memoria tendríamos algo similar a que si el arreglo bidimensional estuviera formado por 5 arreglos separados que hubiéramos inicializado con algo como:

```
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'}
```

Al tiempo que podemos acceder a elementos individuales del arreglo usando la siguiente sintaxis:

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

Ya que los arreglos son continuos en memoria, nuestro bloque de memoria debe ser algo así en realidad:

```
0123456789abcdefghijABCDEFGHIJ9876543210JIHGFEDCBA
└─ Iniciando en la dirección &multi[0][0]
```

Nótese que no escribí **multi[0]** = **"0123456789"**. Si lo hubiera hecho de este modo un terminador de cadena '\0' habría sido añadido al final, ya que dondequiera que haya comillas dobles un caracter **nul** se añade al final de los caracteres contenidos entre esas comillas.

Si este fuera el caso habría tenido que declarar el arreglo de tal modo que hubiera espacio para 11 caracteres por renglón en lugar de 10. Esto porque este es un arreglo bidimensional de caracteres, NO un arreglo de cadenas.

Ahora, el compilador sabe de cuantas columnas consta el arreglo de modo que puede interpretar **`multi+1`** como la dirección de la 'a' en el segundo renglón del arreglo de arriba. Esto es, añade 10, el número de columnas para obtener esta dirección. Si estuviéramos trabajando con números enteros y el arreglo fuera de las mismas dimensiones, el compilador añadiría **10 * sizeof(int)**, lo que en mi máquina es un total de 20 (usando enteros de 2 bytes). Así que, la dirección de el '**9**' en el cuarto renglón sería **`&multi[3][0]`** o **`*(multi + 3)`** en notación de apuntadores. Para obtener el contenido del 2º elemento en el 4º renglón añadimos 1 a esta dirección y la desreferenciamos: `*(*(multi + 3) + 1)`

Pensando un poco podemos observar que:

`*(*(multi + renglon) + columna)`	este modo de acceder a un elemento de la matriz
`multi[renglon][columna]`	y este otro son lo mismo.

El siguiente programa demuestra el uso de una matriz de enteros en lugar de una de caracteres.

PROGRAMA 6.1

```
/* Program 6.1 from PTRTUT10.HTM   6/13/97*/


#include <stdio.h>

#define RENGLONES 5
#define COLUMNAS  10


int multi[RENGLONES][COLUMNAS];
```

```
int main(void)
{
    int renglon, columna;

    for (renglon = 0; renglon < RENGLONES; renglon++)
    {
        for (columna = 0; columna < COLUMNAS; columna++)
        {
            multi[renglon][columna] = renglon*columna;
        }
    }

    for (renglon = 0; renglon < RENGLONES; renglon++)
    {
        for (columna = 0; columna < COLUMNAS; columna++)
        {
            printf("\n%d  ",multi[renglon][columna]);
            printf("%d ",*(*(multi + renglon) + columna));
        }
    }

    return 0;
}
```

Debido a la doble desreferencia requerida en la versión de apuntador, se dice que el nombre de una matriz bidimensional es equivalente a un apuntador a apuntador. Con arreglos de 3 dimensiones estaríamos hablando de arreglos de arreglos de arreglos y entonces el nombre de tal seria el equivalente de un apuntador a apuntador a apuntador.

Sin embargo, aquí hemos reservado inicialmente el bloque de memoria para el arreglo usando notación de arreglos. Por lo que estamos manejando una constante, no una variable, esto significa que estamos hablando de una dirección fija.

La desreferenciación usada arriba nos permite acceder a cualquier elemento en el arreglo de arreglos sin necesidad de cambiar el valor de la dirección (la dirección de **multi[0][0]** es proporcionada por el símbolo **multi**).

CAPÍTULO 7: MÁS SOBRE ARREGLOS MULTIDIMENSIONALES

En el capítulo anterior notamos que una vez dados:

```
#define RENGLONES 5
#define COLUMNAS 10

int multi[RENGLONES][COLUMNAS];
```

podemos acceder a elementos individuales del arreglo **multi** utilizando ya sea:

```
multi[renglon][columna]
```

ó

```
*(*(multi + renglon) + columna)
```

Para entender mejor lo que sucede, reemplacemos `*(multi + renglon)` con una **X**, tal que la expresión nos quede como `*(X + columna)`

Ahora vemos que esta **X** es como un apuntador ya que la expresión se encuentra desreferenciada y sabemos que col es un entero. Aquí la aritmética a utilizar es de un tipo especial llamada "aritmética de punteros". Eso significa que, ya que hablamos de un arreglo de enteros, la dirección a ser apuntada por (el valor de) **X + columna + 1** debe ser mayor que la dirección de **X + columna** por una cantidad que es igual a **sizeof(int)** (el tamaño del tipo de dato entero).

Ya que sabemos la estructura de memoria para arreglos bidimensionales, podemos determinar que en la expresión **multi + renglon** como se hizo arriba, **multi + renglon + 1** hace que esto se incremente por un valor igual al necesario para "apuntar a" el siguiente renglón, lo cual sería entonces **COLUMNAS * sizeof (int)**.

Esto nos dice que si la expresión ***(*(multi + renglones) + columnas)** va a ser evaluada correctamente en tiempo de ejecución, el compilador debe generar código que tome en consideración el valor de **COLUMNAS**, es decir, la segunda dimensión. Debido a la equivalencia entre las dos formas de expresión, esto se hace cierto ya sea que usemos la sintaxis de punteros o la de arreglos **multi[renglon][columna]**.

Así que para evaluar cualquiera de estas dos expresiones, se deben conocer 5 valores:

1. La dirección del primer elemento del arreglo, la cual es conocida por la expresión **multi**, es decir, el nombre del arreglo.
2. El tamaño y el tipo de los elementos que conforman el arreglo, en este caso **sizeof(int)**.
3. La segunda dimensión del arreglo.
4. El valor específico del índice para la primera dimensión, **renglon** en este caso.
5. El valor específico del índice para la segunda dimensión, **columna** en este caso.

Una vez que conocemos esto, consideremos el problema de diseñar una función que manipula los elementos de un arreglo previamente declarado. Por ejemplo, uno que establecería un valor de 1 todos los elementos del arreglo **multi**.

```
void set_value(int m_arreglo[][COLUMNAS])
{
    int renglon, columna;
    for (renglon = 0; renglon < RENGLONES; renglon++)
    {
        for (columna = 0; columna < COLUMNAS; columna++)
        {
            m_arreglo[renglon][columna] = 1;
        }
    }
}
```

Y para llamar a esta función usaríamos entonces:

```
  set_value(multi);
```

Dentro de esta función, hemos usado los valores establecidos por #define en RENGLONES y COLUMNAS, los cuales establecen los límites para los ciclos. Pero dentro de lo que le concierne al compilador, estas son simples constantes, es decir, no hay nada que les relacione directamente con el tamaño del arreglo dentro de la función. **renglon** y **columna** son variables locales, por supuesto. La definición formal del parámetro le permite 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 más adelante, habrá ocasiones en las que preferiremos no declararla en la definición de los parámetros de una función, no quiere decir que eso se vuelva un hábito o algo con lo que haya que ser consistente. Pero la segunda dimensión debe ser usada como se ha mostrado en la expresión del parámetro. La razón por la que la necesitamos es por la forma de la evaluación de **m_arreglo[renglon][columna]**.

Mientras que el parámetro define el tipo de datos (int en este caso) y las variables automáticas para renglón y columna son definidas en los ciclos for, sólo un valor puede ser pasado usando un único parámetro. En este caso, es el valor de **multi**, como lo pasamos en la llamada a la función, es decir, la dirección de su primer elemento, más bien referido como un apuntador al arreglo. Así que la única manera que tenemos de informar al compilador de la segunda dimensión es incluirlo explícitamente en la definición del parámetro de la función.

De hecho y por lo general todas las dimensiones de orden mayor que uno, necesitan de arreglos multidimensionales. Esto significa que si hablamos de arreglos de 3 dimensiones, la segunda y tercera dimensiones deben estar especificadas en la definición del parámetro.

CAPITULO 8: APUNTADORES A ARREGLOS

Por supuesto que los apuntadores pueden apuntar a cualquier tipo de dato, incluyendo arreglos. Mientras que eso ya era evidente cuando discutíamos el programa 3.1, es importante expandir como es que hacemos esto cuando se trata de arreglos multidimensionales.

Para revisar, en el capítulo 2, establecimos que, dado un arreglo de enteros podemos apuntar un puntero a entero a ese arreglo usando:

```
int *ptr;
ptr = &mi_arreglo[0];     /* apuntamos nuestro apuntador al primer elemento del arreglo */
```

Como habíamos establecido, el tipo de la variable apuntadora debe coincidir con el tipo de el primer elemento en el arreglo. En adición, podemos usar un puntero como un parámetro de una función que esté diseñada para manipular un arreglo. Ejemplo:

Dados:

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

Algunos programadores prefieren escribir el prototipo de una función así como: `void a_func(int p[]);`
Lo que informaría a otros que usaran esta función que la función sirve para manipular los elementos de un arreglo. Por supuesto, en cualquier caso, lo que realmente le estamos pasando a la función es un puntero al primer elemento del arreglo, independientemente de la notación usada en el prototipo o definición de la función . Nótese que si usamos la notación de arreglos, no hay necesidad de pasar la dimensión real del arreglo, ya que no estamos pasando el arreglo completo, sino únicamente la dirección de su primer elemento.

Pasemos al problema de tratar con un arreglo bidimensional. Como se estableció en el último capítulo, C interpreta un arreglo de 2 dimensiones como si se tratara de un arreglo que consta de un arreglo de una dimensión. En ese caso, el primer elemento de un arreglo de 2 dimensiones de enteros, es un arreglo unidimensional de enteros. Y un puntero a un arreglo de enteros de dos dimensiones debe apuntar a ese tipo de datos. Una manera de cumplir con esto es por medio del uso de la palabra clave “typedef”. typedef asigna un nuevo nombre para el tipo de datos especificado. Por ejemplo:

```
typedef unsigned char byte;
```

hace que el nombre **byte** signifique el tipo de datos **unsigned char**. Por tanto:

```
byte b[10];
```

sería entonces un arreglo de 10 elementos del tipo unsigned char.

Observemos cómo en la declaración del typedef, la palabra **byte** ha reemplazado aquello que normalmente sería el nombre de nuestra variable **unsigned char**. Por tanto, la regla para usar **typedef** es que el nombre del nuevo tipo de datos sea el nombre usado en la definición del tipo de datos.

Así que al declarar:

```
typedef int Arreglo[10];
```

Arreglo se vuelve un nuevo tipo de datos para un arreglo de 10 enteros. Es decir **`Arreglo mi_arreglo`**, declara **`mi_arreglo`** como un arreglo de 10 enteros y **`Arreglo arr2d[5];`** hace que **`arr2d`** sea un arreglo de 5 arreglos de 10 enteros cada uno.

Nótese que al hacer **`Arreglo *p1d;`** hace de **p1d** un apuntador a un arreglo de 10 elementos. Debido a que ***p1d** apunta al mismo tipo de datos que **arr2d**, asignar la dirección de el arreglo bidimensional **arr2d** a **p1d**, el puntero a arreglo unidimensional de 10 enteros, es aceptable. Es decir, si hacemos tanto: **`p1d = &arr2d[0];`** como **`p1d = arr2d;`** ambos son correctos.

Ya que el tipo de datos que usamos para nuestro apuntador es un arreglo de 10 enteros, esperaríamos que al incrementar **p1d** por 1 cambiaría su valor por **10*sizeof(int)**, y lo hace. Esto es que sizeof(*p1d) es 40. puedes comprobar esto tú mismo escribiendo un pequeño programa.

El usar typedef hace las cosas más claras para el lector y fáciles para el programador, pero no es realmente necesario. Lo que se necesita es una manera de declarar un puntero como p1d sin usar la palabra clave typedef. Es posible hacerlo y que:

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

es la declaración apropiada, es decir p1d es aquí un apuntador a un arreglo de 10 enteros tal y como era como cuando fue declarado usando el tipo de datos Arreglo. Observa que es diferente de hacer:

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

lo que haría de p1d el nombre de un arreglo de 10 apuntadores al tipo entero.

CAPITULO 9: APUNTADORES Y GESTIÓN DINÁMICA DE MEMORIA

Hay veces en que resulta conveniente reservar memoria en tiempo de ejecución usando **malloc()**, **calloc()**, o cualquier otra función de reservación de memoria.

Usar este método permite posponer la decisión del tamaño del bloque de memoria necesario para guardar, por ejemplo un arreglo, hasta el tiempo de ejecución. O permitirnos usar una sección de la memoria para guardar un arreglo de enteros en un tiempo determinado, y posteriormente, cuando esa memoria no sea necesaria, liberarla para otros usos, como para guardar un arreglo de estructuras.

Cuando la memoria es reservada, las funciones de reservación de memoria (como **malloc()** o **calloc()**, etc.) regresan un puntero. El tipo de este puntero depende del estamos usando un viejo compilador de K&R o uno de los nuevos compiladores con especificaciones ANSI. Con el tipo de compilador viejo, el puntero retornado es del tipo **char**, mientras que con los ANSI es del tipo **void**.

Si usas uno de estos compiladores antiguos, y quieres reservar memoria para un arreglo de enteros, tienes que hacer la conversión (cast) del puntero tipo char a un puntero de tipo entero (int). Por ejemplo, para reservar espacio para 10 enteros, escribiríamos:

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

{ .. Rutina del manejo de error va aquí .. }
```

Si estamos utilizando un compilador compatible con ANSI, **malloc()** regresa un apuntador del tipo **void** y ya que un puntero de este tipo puede ser asignado a apuntar a una variable de cualquier tipo de objeto, el cast convertidor **(int *)** mostrado en el código expuesto arriba no es necesario. La dimensión del arreglo puede ser determinada en tiempo de ejecución por lo que no es necesario conocer este dato en tiempo de compilación. Esto significa que el **10** de arriba puede ser una variable leída desde un archivo de datos, desde el teclado, o calculada en base a una necesidad, en tiempo de ejecución.

Debido a la equivalencia entre la notación de arreglos y la notación de punteros, una vez que el apuntador **iptr** ha sido asignado como arriba, podemos usar la notación de arreglos. Por ejemplo, uno puede escribir:

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

para establecer el valor de todos los elementos a 2.

Aún con un buen entendimiento de los apuntadores y de los arreglos, es usual que algo que hace tropezar a los novatos en C sea la asignación dinámica de memoria para arreglos multidimensionales. En general, nos gustaría ser capaces de acceder a los elementos de dichos arreglos usando notación de arreglos, no notación de punteros, siempre que sea posible. Dependiendo de la aplicación podemos o no conocer las dimensiones de un arreglo en tiempo de compilación. Esto nos conduce a una variedad de caminos a seguir para resolver nuestra tarea.

Como hemos visto, cuando alojamos dinámicamente un arreglo unidimensional, su dimensión puede ser determinada en tiempo de ejecución. Ahora que para el alojamiento dinámico de arreglos de orden superior, nunca necesitaremos conocer la primera dimensión en tiempo de compilación. Si es que vamos a necesitar

conocer las otras dimensiones depende de la forma en que escribamos el código. Vamos a discutir sobre varios métodos de asignarle espacio dinámicamente a arreglos bidimensionales de enteros. Para comenzar consideremos el caso en que la segunda dimensión es conocida en tiempo de compilación:

METODO 1:

Una manera de enfrentar este problema es usando la palabra clave **typedef**. Para alojar arreglos de 2 dimensiones, recordemos que las siguientes dos notaciones dan como resultado la generación del mismo código objeto:

```
multi[renglon][columna] = 1;     *(*(multi + renglon) + columna) = 1;
```

También es cierto que las siguientes dos notaciones dan el mismo resultado:

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

Ya que la que está a la derecha debe evaluarse a un apuntador, la notación de arreglos a la izquierda debe hacerlo también. De hecho **multi[0]** retornará un puntero al primer entero en el primer renglón, **multi[1]** un puntero al primer entero del segundo renglón, etc. En realidad **multi[n]** se evalúa como un puntero a ese arreglo de enteros que conforma el n-ésimo renglón de nuestro arreglo bidimensional. Esto significa que podemos pensar en **multi** como un arreglo de arreglos y **multi[n]** como un puntero al n-ésimo arreglo de este arreglo de arreglos. Aquí la palabra **puntero (apuntador)** es usada para representar el valor de una dirección. Mientras que este uso es común en los libros, al leer instrucciones de este tipo, debemos ser cuidadosos al distinguir entre la dirección constante de un arreglo y una variable apuntadora que es un objeto que contiene datos en si misma.

Veamos ahora el:

PROGRAMA 9.1

```
/* Program 9.1 from PTRTUT10.HTM  6/13/97 */

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

#define COLUMNAS 5

typedef int Arreglo_de_renglones[COLUMNAS];
Arreglo_de_renglones *rptr;

int main(void)
{
    int nrenglones = 10;
    int renglon, columna;
    rptr = malloc(nrenglones * COLUMNAS * sizeof(int));
    for (renglon = 0; renglon < nrenglones; renglon++)
    {
        for (columna = 0; columna < COLUMNAS; columna++)
        {
            rptr[renglon][columna] = 17;
        }
    }

    return 0;
}
```

He asumido que se ha usado un compilador ANSI, así que la conversión (cast) al puntero del sin tipo (void) regresado por la función **malloc()** no es necesaria. Si estas usando un compilador no compatible con ANSI, hay que hacer la conversión usando:

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

Usando este método, **rptr** tiene todas las características del nombre de un arreglo (excepto que rptr es modificable), y la notación de arreglo puede ser usada en el resto del programa. Esto significa también que si se pretende escribir una función para modificar los contenidos del arreglo, se debe usar COLUMNAS como parte del parámetro de esa función, tal y como se hizo al estar discutiendo el paso de arreglos bidimensionales a una función.

METODO 2:

En el método 1 de arriba, rptr se volvió un apuntador del tipo "arreglo unidimensional de COLUMNAS de enteros". Es evidente entonces que existe una sintaxis para usar este tipo sin la necesidad de usar la palabra clave **typedef**. Si escribimos:

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

las variable **xptr** tendría las mismas características que la variable **rptr** del método 1, y no habremos usado la palabra clave **typedef**. Aquí **xptr** es un puntero aun arreglo de enteros y el tamaño de ese arreglo está dado por la constante **COLUMNAS**. Los paréntesis hacen que la notación de punteros predomine, a pesar de que la notación de arreglo tiene una mayor precedencia de evaluación. Es decir que si hubiéramos escrito:

```
int *xptr[COLUMNAS];
```

habríamos definido a **xptr** como un arreglo de apuntadores consistente en un número de apuntadores igual a la cantidad definida por COLUMNAS. Es obvio que no se trata de lo mismo. Como sea, los arreglos de apuntadores tienen utilidad al alojar dinámicamente arreglos bidimensionales, como veremos en los siguientes dos métodos.

METODO 3:

Consideremos el caso en el que no conozcamos el número de elementos por cada renglón en tiempo de compilación, es decir que el número de renglones y el número de columnas será determinado en tiempo de ejecución. Un modo de hacerlo sería crear un arreglo de apuntadores de tipo entero (**int**) y luego reservar memoria para cada renglón y apuntar estos apuntadores a cada renglón. Consideremos:

PROGRAMA 9.2

```
/* Program 9.2 from PTRTUT10.HTM   6/13/97 */

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


int main(void)
{
    int nrenglones = 5;   /* Ambos, nrenglones y ncolumnas pueden ser */
    int ncolumnas = 10;   /* evaluados o leídos en tiempo de ejecución */
    int renglon;
    int **renglonptr;
    renglonptr = malloc(nrenglones * sizeof(int *));
    if (renglonptr == NULL)
```

```
    {
        puts("\nError al reservar espacio para apuntadores de renglon.\n");
        exit(0);
    }

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

    for (renglon = 0; renglon < nrenglones; renglon++)
    {
        renglonptr[renglon] = malloc(ncolumnas * sizeof(int));
        if (renglonptr[renglon] == NULL)
        {
            printf("\nError al reservar memoria para el renglon[%d]\n",renglon);
            exit(0);
        }
        printf("\n%d         %p         %d", renglon, renglonptr[renglon],
renglonptr[renglon]);

        if (renglon > 0)
        printf("              %d",((int)renglonptr[renglon] –
(int)renglonptr[renglon-1]));
    }

    return 0;
}
```

En el código de arriba, **renglonptr** es un apuntador a apuntador de tipo **entero**. En este caso, apunta al primer elemento de un arreglo de apuntadores del tipo **int**. Consideremos el número de llamadas a **malloc()**:

Para obtener nuestro arreglo de apuntadores:	1	llamada
Para obtener espacio para los renglones:	5	llamadas

Total:	6	llamadas.

Si optas por este método, observa que mientras puedes usar la notación de arreglos para acceder a elementos individuales del arreglo, por ejemplo: **`renglonptr[renglon][columna] = 17;`**, esto no significa que los datos en el arreglo bidimensional sean continuos en memoria.

Puedes, sin embargo, usar la notación de arreglos tal y como si los datos se encontraran en un bloque continuo de memoria. Por ejemplo, puedes escribir:

```
renglonptr[renglon][columna] = 176;
```

tal y como se haría si renglonptr fuera el nombre de un arreglo bidimensional creado en tiempo de compilación. Por supuesto que los valores de **`[renglon]`** y **`[columna]`** deben estar dentro de los límites establecidos del arreglo que se ha creado, igual que con un arreglo creado en tiempo de compilación.

Si lo que queremos es tener un bloque continuo de memoria dedicado al almacenamiento de los elementos del arreglo, puede hacerse del siguiente modo:

METODO 4:

Con este método reservamos un bloque de memoria para contener primero el arreglo completo. Después creamos un arreglo de apuntadores para apuntar a cada renglón. Así, aunque estamos usando un arreglo de punteros, el arreglo real en memoria es continuo. El código es este:

PROGRAMA 9.3

```
/* Program 9.3 from PTRTUT10.HTM   6/13/97 */

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

int main(void)
{
    int **rptr;
    int *aptr;
    int *pruebaptr;
    int k;
    int nrenglones = 5;     /* Ambos, nrenglones y ncolumnas pueden ser */
    int ncolumnas = 8;      /* evaluados o leidos en tiempo de ejecucion */
    int renglon, columna;

    /* ahora reservamos memoria para el arreglo completo */
    aptr = malloc(nrenglones * ncolumnas * sizeof(int));
    if (aptr == NULL)
    {
        puts("\nError al reservar memoria para el arreglo completo.");
        exit(0);
    }

    /* ahora reservamos espacio para los apuntadores a renglones */
    rptr = malloc(nrenglones * sizeof(int *));
    if (rptr == NULL)
    {
        puts("\nError al reservar memoria para los punteros");
        exit(0);
    }

    /* y ahora hacemos que los apuntadores “apunten” */
    for (k = 0; k < nrenglones; k++)
    {
        rptr[k] = aptr + (k * ncolumnas);
    }

    /* Ahora demostramos que los punteros a renglones se han incrementado */
    printf("\nDemostramos que los punteros a renglones se han incrementado:");
    printf("\n\nIndice  Apuntador(dec)    Diferencia(dec)");

    for (renglon = 0; renglon < nrenglones; renglon++)
    {
        printf("\n%d         %d", renglon, rptr[renglon]);
        if (renglon > 0)
        printf("            %d",((int)rptr[renglon] – (int)rptr[renglon-1]));
    }

    printf("\n\nY ahora mostramos el arreglo:\n");
    for (renglon = 0; renglon < nrenglones; renglon++)
    {
        for (columna = 0; columna < ncolumnas; columna++)
        {
            rptr[renglon][columna] = renglon + columna;
            printf("%d ", rptr[renglon][columna]);
        }
        putchar('\n');
    }
```

```
    puts("\n");

    /* Y aquí es donde demostramos que efectivamente estamos manejando un
       arreglo bidimensional contenido en un bloque continuo de memoria */

    printf("Demostrando que los elementos son continuos en memoria:\n");

    pruebaptr = aptr;
    for (renglon = 0; renglon < nrenglones; renglon++)
    {
        for (columna = 0; columna < ncolumnas; columna++)
        {
            printf("%d ", *(pruebaptr++));
        }
        putchar('\n');
    }

    return 0;
}
```

Consideremos de nuevo el número de llamadas a malloc():

Para reservar la memoria que contendrá todo el arreglo:	1	llamada
Para reservar la memoria para el arreglo de punteros:	1	llamada
	---	
Total:	2	llamadas.

Bien, pues cada llamada a **malloc()** crea un gasto adicional de espacio ya que **malloc()** es por lo general implementada por el sistema operativo formando una lista enlazada que contiene los datos correspondientes al tamaño del bloque. Pero lo más importante es que con arreglos grandes (varios cientos de renglones), seguirle el rastro a la memoria que debe ser liberada en algún momento, puede llegar a ser engorroso. Este último método, combinado con la conveniencia de la continuidad del bloque de memoria, lo que nos permite la inicialización de todo el bloque a ceros usando **memset()**, haría de esta alternativa la más conveniente.

Como ejemplo final sobre el tema de arreglos multidimensionales, demostraremos el alojamiento dinámico de un arreglo de 3D. Este ejemplo mostrará una cosa más a tener en cuenta con este tipo de alojamiento. Por las razones expuestas arriba, usaremos el último método. Veamos pues el siguiente código:

PROGRAMA 9.4

```
/* Program 9.4 from PTRTUT10.HTM   6/13/97 */

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

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

int main(void)
{
    char *espacio;
    char ***Arr3D;
    int y, z, diff;
```

```
/* Primeramente reservamos el espacio necesario para el arreglo completo */

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

/* enseguida reservamos el espacio para un arreglo de apuntadores, los cuales
   eventualmente apuntaran cada uno al primer elemento de un arreglo bidimensional
   de apuntadores a apuntadores */

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

/* para cada uno de estos asignamos un apuntador a un recien
   asignado arreglo de apuntadores a renglon */


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

 /* y para cada espacio de este arreglo, colocados un apuntador el primer elemento
    de cada renglon es el espacio del arreglo originalmente alojado */

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

/* Y, ahora, revisamos cada direccion en nuestro arreglo 3D para comprobar que los
 indices de nuestro apuntador Arr3d esten alojados de manera continua en memoria */

    for (z = 0; z < Z_DIM; z++)
    {
        printf("Direccion del arreglo %d es %ph\n", z, *Arr3D[z]);
        for ( y = 0; y < Y_DIM; y++)
        {
            printf("El arreglo %d y el renglon %d comienzan en %ph", z, y,
Arr3D[z][y]);
            diff = Arr3D[z][y] - espacio;
            printf("    dif = %ph  ",diff);
            printf(" z = %d  y = %d\n", z, y);
        }
    }
    return 0;
}
```

Si has seguido este tutorial hasta este punto no debes tener problemas descifrando el código del programa de arriba basándote en los comentarios. Hay un par de cuestiones que explicar de todos modos, comencemos por la línea que dice:

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

Observa que aquí **espacio** es un apuntador de tipo caracter, que es el mismo tipo que **Arr3D[z][y]**. Es importante que, al agregar un entero, como el obtenido por la evaluación de la expresión **(z*(X_DIM * Y_DIM) + y * X_DIM)**, a un apuntador, el resultado es un nuevo valor de apuntador. Y cuando asignamos valores de apuntador a variables apuntadoras, los tipos de datos del valor y de la variable deben coincidir.

CAPITULO 10: APUNTADORES A FUNCIONES

Hasta este punto hemos discutido el uso de apuntadores con objetos de datos. C permite también la declaración de apuntadores a funciones. Los apuntadores a funciones tienen variedad de usos y algunos de estos serán expuestos aquí.

Consideremos el siguiente problema real: Tenemos que escribir una función que sea capaz de ordenar virtualmente cualquier colección de datos que pueda ser contenida en un arreglo. Sea este un arreglo de cadenas, de enteros, flotantes, e incluso estructuras.

El algoritmo de ordenación puede ser el mismo para todos. Por ejemplo, puede ser un simple algoritmo de ordenación por el método de la burbuja, o el mucho más complejo algoritmo de ordenación quick sort o por shell sort. Usaremos un simple algoritmo de burbuja para nuestros fines didácticos.

Sedgewick [1] ha descrito el algoritmo de ordenación por burbuja usando código C al establecer una función a la que, una vez que se le ha pasado el apuntador al arreglo, ordena el arreglo. Si le llamamos a esta función **bubble()**, un programa de ordenación es descrito por bubble_1.c en el código que sigue:

bubble_1.c

```
/* Program bubble_1.c from 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;
            }
        }
    }
}
```

El método de la burbuja es un algoritmo de ordenación de los más sencillos. El algoritmo busca en el arreglo desde el segundo hasta el último elemento comparando cada uno con el que le precede. Si el elemento que le precede es mayor que el elemento actual, los elementos son intercambiados en su posición de tal modo que el más grande quede más cerca del final del arreglo. En la primera pasada, esto resulta en que el elemento más grande termina al final del arreglo.

El arreglo está ahora limitado a todos los elementos, a excepción del último y el proceso se repite. Eso pone el siguiente elemento más grande en el lugar que precede al elemento más grande. El proceso se repite un número de veces igual al número total de elementos menos 1. El resultado final es un arreglo ordenado.

Aquí nuestra función está diseñada para ordenar un arreglo de enteros. Así que en la línea comparamos enteros y de la línea 2 a la 4 usamos un almacén temporal de enteros para guardar enteros. Lo que queremos averiguar es que si podemos modificar este código para que pueda usarse con cualquier tipo de datos, es decir que no esté restringido a la ordenación de enteros.

Al mismo tiempo no deseamos modificar nuestro algoritmo ni el código asociado a él cada vez que lo usamos. Comencemos por remover la comparación dentro de la función **bubble()** para que nos sea relativamente fácil modificar la función de comparación sin tener que re-escribir pedazos relacionados con el algoritmo de ordenación en sí.

Esto nos trae como resultado el programa bubble_2.c:

bubble_2.c

```
/* Program bubble_2.c from PTRTUT10.HTM   6/13/97 */

/* Separamos 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);
}
```

Si nuestro objetivo es hacer de nuestra rutina de ordenación independiente del tipo de datos, una manera de lograrlo es usar apuntadores sin tipo (void) para que apunten a los datos en lugar de usar el tipo de datos enteros. Para dirigirnos a este objetivo, comenzaremos por modificar algunas cosas en nuestro programa de arriba, de modo que podamos usar apuntadores. Empecemos por meter apuntadores del tipo entero:

bubble_3.c

```
/* Program bubble_3.c from 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);
}
```

Observa los cambios. Estamos ahora pasando un apuntador a un entero (o a un arreglo de enteros) a **bubble()**. Y desde dentro de bubble estamos pasando apuntadores a los elementos que queremos comparar del arreglo a nuestra función de comparación. Y por supuesto estamos desreferenciando estos apuntadores en nuestra función **compare()** de modo que se haga la comparación real entre elementos. Nuestro siguiente paso será convertir los apuntadores en **bubble()** a apuntadores sin tipo de tal modo que la función se vuelva más insensible al tipo de datos a ordenar. Esto se muestra en el programa bubble_4.c

bubble_4.c

```
/* Program bubble_4.c from 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);
}
```

Observa que, al hacer esto, en **compare()** tuvimos que introducir la conversión del puntero void pasado, al tipo de datos que realmente se está ordenando. Pero, como veremos más tarde, eso está bien. Y ya que lo que se está pasando a **bubble()** es aún un puntero a un arreglo de enteros, tuvimos que convertir esos punteros a punteros sin tipo cuando los pasamos como parámetros en nuestra llamada a la función **compare()**.

Tratemos ahora el problema de qué le vamos a pasar a **bubble()**. Queremos hacer que el primer parámetro de esta función sea un puntero sin tipo. Pero, eso significa que dentro de **bubble()** debemos hacer algo al respecto de la variable **t**, la que actualmente es de tipo entero. Además, donde usamos **t=p[j-1];** el tipo de **p[j-1]** necesita ser conocido en razón de saber cuantos bytes serán copiados a la variable t (o a cualquier otras cosa con la que reemplacemos a **t**).

Hasta ahora, en bubble_4.c, el conocimiento dentro de **bubble()** del tipo de datos a ser ordenado (y por tanto el tamaño individual de cada elemento) es obtenido de el hecho de que el primer parámetro es un puntero de tipo entero. Si queremos ser capaces de utilizar a **bubble()** para ordenar cualquier tipo de datos, necesitamos hacer de este parámetro un puntero del tipo **void**. Pero al hacer eso, perdemos la información respecto del tamaño individual de los elementos dentro del arreglo. Así que en bubble_5.c añadiremos un parámetro extra para manejar la información del tamaño.

Estos cambios desde bubble_4.c a bubble_5.c son, al parecer, un poco más extensos que aquellos que hicimos antes.

Así que revisa cuidadosamente las diferencias:

bubble_5.c

```
/* Program bubble_5.c from 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);
}
```

Revisa que he cambiado el tipo de datos del arreglo de **int** a **long** para mostrar los cambios necesarios en la función **compare()**. Dentro de **bubble()** he construido un arreglo con la variable **t** (la cual hemos tenido que cambiar del tipo de datos **int** al tipo **long**). He añadido un buffer de tamaño 4 de tipo unsigned char, el cual es el tamaño requerido para almacenar un entero largo (esto cambiará en futuras modificaciones al código). El apuntador ***bp** de tipo unsigned char es usado para apuntar al inicio del arreglo a ordenar, es decir al primer elemento del arreglo.

Tuvimos también que modificar lo que se le pasa a **compare()**, y el modo en que hacíamos el intercambio de elementos cuando la comparación indicaba que se hacía el intercambio. El uso de **memcpy()** y de la notación de punteros va en función de reducir la sensibilidad al tipo de datos.

De nuevo, el hacer una revisión cuidadosa de bubble_5.c con bubble_4.c puede resultar muy provechoso para entender lo que está sucediendo y porqué.

Vamos ahora con bubble_6.c donde usamos la misma función **bubble()** que usamos en bubble_5.c ahora para ordenar cadenas en lugar de números enteros. Por supuesto que hemos tenido que modificar la función de comparación ya que el modo de comparar cadenas es diferente del modo en que se comparan números enteros. También en bubble_6.c hemos removido las líneas que dentro de **bubble()** estaban como comentarios en bubble_5.c

bubble_6.c

```
/* Program bubble_6.c from 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));
}
```

Pero el hecho de que **bubble()** no fuera modificada de aquella que usamos en bubble_5.c indica que esta función es capaz de ordenar una amplia variedad de tipos de datos. Lo que entonces hay que hacer es pasarle a **bubble()** el nombre de la función de comparación a usar para que esta sea realmente universal. Al igual que el nombre de un arreglo es la dirección de su primer elemento en el segmento de datos, el nombre de una función se interpreta como la dirección de esa función en el segmento de código. Así que necesitamos un apuntador a una función. En este caso, la función de comparación.

Los punteros a funciones deben coincidir a las funciones apuntadas en el número y tipos de parámetros y en el tipo del valor regresado por la función. En nuestro caso, declaramos nuestro apuntador de función como:

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

Considera que si lo hubiéramos escrito como:

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

obtendríamos el prototipo de un función que regresa un puntero de tipo **entero**. Debido a que en C los paréntesis tienen un orden de precedencia mayor que el operador de punteros **'*'**, al poner entre paréntesis la cadena (*fptr) estamos indicando que estamos declarando un apuntador a una función.

Ahora modifiquemos nuestra declaración de **bubble()** al añadirle como 4º parámetro, un apuntador a una función del tipo apropiado.

Entonces su prototipo queda como:

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

Cuando llamemos a **bubble()**, insertaremos el nombre de la función de comparación a utilizar. bubble_7.c demuestra cómo este método permite el uso de la misma función **bubble()** para ordenar datos de diferentes tipos.

bubble_7.c

```
/* Program bubble_7.c from 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("\nAntes de ordenar:\n");

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

    for (i = 0; i < 5; i++)                 /* mostramos las cadenas */
    {
        printf("%s\n", arr2[i]);
    }
    bubble(arr, 4, 10, compare_long);        /* ordenamos los enteros largos */
    bubble(arr2, 20, 5, compare_string);     /* ordenamos las cadenas */
    puts("\n\nDespués de ordenar:\n");

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

    for (i = 0; i < 5; i++)           /* mostramos las cadenas ya 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);
}
```

Referencias en el capítulo 10:

[1] Robert Sedgewick

"Algorithms in C"

Addison-Wesley

ISBN 0-201-51425-7

EPILOGO

Escribí el presente material para dar una introducción sobre los apuntadores a los que comienzan en C. En C, entre más entienda uno sobre apuntadores, mayor será la flexibilidad que se adquiera para escribir código.

El contenido de este trabajo expande lo que fue mi primer esfuerzo que tenía por nombre ptr_help.txt y que se encontraba en una de las primeras versiones de la colección de SNIPPETS de C de Bob Stout. El contenido de esta versión se ha actualizado de aquella encontrada como PTRTUTOT.ZIP incluida en el archivo SNIP9510.ZIP.

Estoy siempre dispuesto a recibir crítica constructiva sobre el presente material, o revisiones o peticiones para la adición de otro material importante. Por tanto, si tienes preguntas, comentarios, críticas, etc. respecto de lo que en este documento se ha expuesto, me agradaría mucho que me contactaras por medio de email a:

tjensen@ix.netcom.com