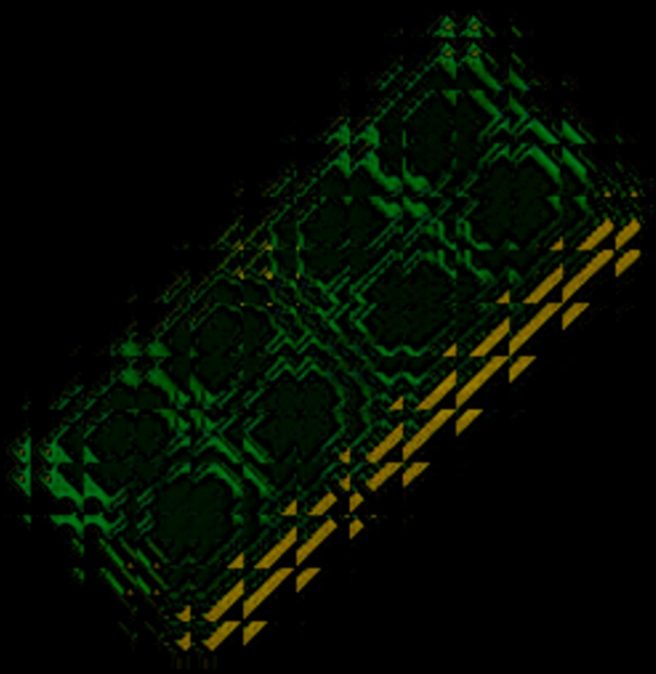
HACK X CRACK

Introducción a los punteros en C

By Piou



WWW.HACKXCRACK.ES

Introducción a los punteros en C

En este artículo voy a introducirlos en el manejo y usos de punteros.

Índice

1. Presentación
2. Introducción
3. Introducción a la memoria del ordenador
4. ¿Qué es un puntero?
5. Funcionamiento de los punteros
6. Cuidados a tener en cuenta
7. Usos
   - Argumentos por referencia
   - Recorrer arrays
   - Pasar arrays a funciones
   - Memoria dinámica

Presentación

Dicen por ahí que empezar a escribir un texto es lo más difícil, da igual que cuentes una historia de ciencia ficción o que intentes enseñar a alguien a programar, así que voy a empezar pidiendo, y luego daré sobre el tema lo que en mi mano (y en mi cabeza) esté.

Este "manual" ha sido escrito para HackXCrack, una comunidad on-line que podéis visitar aquí: http://www.hackxcrack.es/forum/index.php destinada a servir de base para cualquier usuario que quiera empezar en la informática, para que el que sepa pueda profundizar y para el que quiera pueda contribuir y aportar su granito de arena. Nadie gana nada con esto, no tenemos ningún ánimo de lucro y desarrollamos la comunidad en nuestro tiempo libre.
Llevar un página web es algo que cuesta dinero, así que pido desde aquí a todo el que pueda, y sobre todo, quiera, que done la cantidad que desee, por pequeña que sea, para ayudar al mantenimiento de la comunidad.
Puedes donar desde esta dirección: http://www.hackxcrack.es/donaciones/ via PayPal o por transferencia bancaria.
Y ya que estoy, aprovecho para hacer un poco de publicidad de mi blog personal, no tengo mucho tiempo pero publico algo cuando puedo: http://whitehathacking.wordpress.com/

1. Introducción

Y ahora a lo que toca.

Los punteros en programación en C/C++ es algo que a la gente le cuesta entender. Lo se de primera mano. Yo aprendí a programar por mi cuenta siendo muy pequeño y encima con C, a lo bestia. Cuando llegué a punteros lo leí por encima entendiendo lo que eran pero al terminar me quedé igual que al principio, ¿y esto para qué sirve? Como es algo que he aprendido por mi cuenta y hace relativamente poco tiempo creo que podré explicarlo bien porque se qué es lo que cuesta y en lo que la gente tiene dificultad.
Así que desde mi experiencia, puedo decir que lo que cuesta son dos cosas: la primera, la comprensión, qué es un puntero y qué hace; la segunda, la aplicación, para qué se puede usar.

En realidad no creo que haya mucho más de lo que hablar. A la hora de programar y optimizar algoritmos usando punteros lo que cuenta es lo

que lo hayas interiorizado y los conocimientos que tengas sobre programación general, así que queda decir que después de leer esto y algún que otro manual que lo oriente de otro modo (contrasta todo siempre), empieces a practicar aunque sea con programas chorras que no sirvan para nada y veas cómo funcionan.

Por último, decir que cualquier duda que surja me la podéis preguntar en el foro de "Programación en C/C++", de HackXCrack, intentaré ayudar en lo que pueda: http://www.hackxcrack.es/forum/index.php?board=17.0

2. Introducción a la memoria del ordenador.

Esto es una pequeña introducción que considero de ayuda y quizás falte.
La memoria se organiza en bits. Los bits son la unidad de memoria más pequeñas y puede valer 1 ó 0. Para trabajar con comodidad se usan bytes, que son agrupaciones de 8 bits. Se usan 8 porque es un número suficientemente grande para considerarlo unidad (por ejemplo en el código ASCII cada letra o símbolo es un byte) y porque permite una conversión directa muy sencilla entre binario y hexadecimal.
La memoria del ordenador se estructura en varios niveles, según el tiempo de acceso:

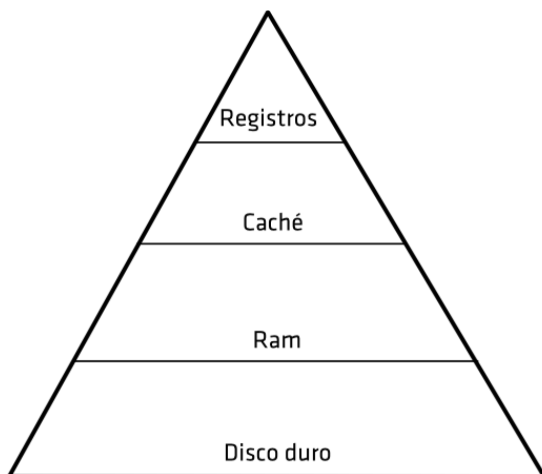


Figura 1. Memoria de un ordenador.

Los registros son una zona de memoria del procesador de muy rápido acceso, pero muy pequeños, suelen ser de 32 o 64 bits. La caché es un poco más grande. Pero bueno, lo que ahora nos interesa es la RAM y el disco duro, cuyos papeles en esto veremos ahora.
La RAM es una zona con un tiempo de acceso que permite una ejecución de programas sin problemas de rendimiento (evidentemente depende del programa y velocidad del procesador), y el disco duro es una zona muy lenta, pero mucho más grande que la RAM.
En la RAM (y en memorias grandes), las direcciones de memoria se numeran en hexadecimal. Como he dicho antes la conversión entre binario y hexadecimal es muy sencilla y directa. Se pueden transformar grupos de cuatro bits en una letra hexadecimal, por ejemplo, el número binario 10010101111, podemos separar bloques de cuatro bits, empezando por el bit menos significativo, osea el que está a la derecha, y nos quedan dos grupos de cuatro y uno de tres: 100 | 1010 | 1111, ahora convertimos. En nuestra cabeza podemos pasar a decimal si nos es más cómodo.
$100_2 = 4_{10} = 4_{16}$
$1010_2 = 10_{10} = A_{16}$
$1111_2 = 16_{10} = F_{16}$
El subíndice indica la base (2 -> binario, 10 -> decimal, 16 -> hexadecimal)
El número quedaría, juntándolo otra vez: $10010101111_2 = 4AF_{16}$.
Como por esta conversión los decimales solo pueden ir desde 0_{10} (0_2) hasta 16_{10} (1111_2), que son justo las cifras del sistema hexadecimal (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F), esta conversión es muy sencilla.

Esta es la razón por la que se usan números hexadecimales. Y por esto encontraremos números hexadecimales en las direcciones de memoria.

Por último, algo que deberías saber si sabes C, es que los números hexadecimales se nombran poniendo un 0x delante, y así me referiré a ellos en este documento.

Me he dejado un montón de cosas en el tintero, pero la verdad es que esto no es muy importante saberlo, aunque creo que ir un poco más allá del tema, y más cuando no se sabe lo del más acá, ayudará a comprender y afianzar mucho mejor lo que nos interesa.

Después de esta introducción chapucera vamos a lo que toca:

3. ¿Qué es un puntero?

La manera más fácil de definir un puntero y que no echará a nadie para atrás es: un puntero es una variable, nada más que eso.
Tenemos muchos tipos de variables que cambian en lo que podemos meter dentro. Tenemos variables en las que podemos meter números enteros (tipo int), números en coma flotante (tipo float, double, etc), letras (tipo char). ¿Y qué podemos meter en un puntero? Pues algo tan simple como una dirección de memoria. ¿Y eso qué es?

Vamos a ver un poco cómo funciona la memoria del ordenador. Quizás no te hayas planteado qué pasa cuando defines una variable. Cuando escribimos

```
int numero=5;
```

¿Qué hace el ordenador? Vayamos un poco más para atrás.
Cuando compilamos un programa, el compilador lo que hace es traducirlo a lenguaje máquina. El lenguaje máquina son montones de sentencias que el procesador puede entender directamente y ejecutar. Un ejecutable es un archivo que guarda la sucesión lineal de estas sentencias y nada más.

Cuando hacemos click en un ejecutable, el ordenador vuelca todo el contenido de este en la memoria RAM, es decir copia las sentencias del disco duro a la RAM. Pero no lo hace tal cual, sino que separa varias zonas. En una zona tenemos el código en sí, lo que hace el programa, y en la otra zona tenemos memoria reservada para variables. Y esto es lo que pasa cuando definimos una variable: le estamos diciendo al ordenador que nos reserve una zona de memoria en la que poder guardar datos.

Vamos a verlo con un ejemplo. En nuestro programa definimos una variable de tipo int:

```
int numero;
```

El ordenador va a reservar una cantidad de memoria que dependerá del tipo de variable.
En mi caso el tipo int ocupa 4 bytes. Podemos ver cuánto ocupa cada tipo de variable con el operador sizeof:

```
printf("Tamaño de int: %i\n", sizeof(int));
```

Sigamos, el ordenador reservará entonces 4 bytes para esta variable.
Podemos ver la memoria RAM como una tabla, o una lista de direcciones, cada una con un espacio en el que podemos meter cosas. Estas direcciones, como dije en la introducción, se numeran de manera hexadecimal, es decir con números del 0-F.

Supongamos que tenemos la memoria de la siguiente manera en bytes:

Dirección	Contenido
0x00000001	-
0x00000002	-
0x00000003	-
0x00000004	-
0x00000005	-
0x00000006	-

Ahora en nuestro programa ponemos la definición de una variable tipo char, que se llame por ejemplo carácter, y que como ya sabemos ocupa un byte.

```
char caracter;
```

Al definir la variable, el ordenador nos reserva en la memoria el espacio necesario para esa variable, que depende de tantos factores que podemos decir que es aleatorio. Supongamos que le reserva la posición 0x00000002. La variable char la podemos ver como un número entero que va de 0 a 255, o de -127 a +127, dependiendo de si la definimos como signed (por defecto), o como unsigned.

Como hemos visto: 1 byte = 8 bits. En binario, ocho bits completos 11111111 = 256, que es el máximo número que podemos guardar ahí. Cuando el ordenador reserve la memoria, la tabla quedará así (la letra en negrita significa espacio reservado):

Dirección	Contenido
0x00000001	-
0x00000002	-
0x00000003	-
0x00000004	-
0x00000005	-
0x00000006	-

Vamor a darle un valor de la variable, por ejemplo la letra m.

```
caracter = 'm';
```

El código ASCII de la letra 'm' minúscula es el 109, o en hexadecimal, el 6D, así que cuando el ordenador reserve la memoria y le ponga a la variable el valor de la letra m, la tabla de memoria nos quedará así:

Dirección	Contenido
0x00000001	-
0x00000002	**6D**
0x00000003	-
0x00000004	-
0x00000005	-
0x00000006	-

4. Funcionamiento de los punteros

Ahora, y siguiendo con lo que tenemos, vamos a usar un puntero para apuntar a esta variable.
El puntero como ya hemos dicho es una variable que puede almacenar posiciones de memoria, y que tiene un uso un poco diferente de otras variables. Para definir un puntero lo hacemos igual que con una variable, pero usando un asterisco entre el tipo de puntero y el nombre que le demos, pegado al nombre del puntero.
Vamos a definir un puntero que apunte a la variable caracter, y que se llame puntero, de momento sólo lo definimos.

```
char *puntero;
```

Supongamos que el ordenador le reserva espacio a la variable puntero justo depués de la variable caracter, la tabla quedará así:

Dirección	Contenido
0x00000001	-
0x00000002	**6D**
0x00000003	**-**
0x00000004	**-**
0x00000005	**-**
0x00000006	**-**

Ahora vamos a darle un valor, vamos a hacer que apunte a la variable carácter. ¿Cómo le damos la posición de memoria de la variable? Esto se hace con el operador &. Este operador

puesto delante de una variable nos devuelve su dirección de memoria y no su valor. Para hacer que apunte a la variable hacemos esto:

```
puntero = &caracter;
```

Fácil, ¿no? Más tarde explicaré cuándo usar el * para un puntero y cuándo no. Pero antes, ¿cómo queda ahora la tabla de memoria? Es fácil imaginarlo:

Dirección	Contenido
0x00000001	-
0x00000002	**6D**
0x00000003	**0x00000002**
0x00000004	**-**
0x00000005	**-**
0x00000006	**-**

¿Quedaŕa así? ¿Seguro? Pues no. ¿Cómo se organizaba la memoria? En bytes ¿Y cuántas cifras hexadecimales puede tener un byte? 1 byte = 8 bits. 1Cifra hexadecimal por cada 4 bits = 2 cifras hexadecimales. Cada posición puede contener sólo dos cifras hexadecimales, o un byte, que es lo mismo. Las posiciones de memoria son 8 cifras hexadecimales, osea 4 bytes. Si nos fijamos en la negrita, al definir el puntero se reservaron cuatro posiciones. La tabla quedará así:

Dirección	Contenido
0x00000001	-
0x00000002	**0x6D**
0x00000003	**0x00**
0x00000004	**0x00**
0x00000005	**0x00**
0x00000006	**0x02**

Un puntero siempre ocupa la misma memoria para un sistema concreto. Si la memoria se numera como aquí, es decir, si se numera en cuatro bytes, los punteros ocuparan lo suficiente para guardar 4 bytes, es decir, ocuparan 4 bytes.

Es importante tener en cuenta que un puntero ocupa siempre lo mismo, independientemente del tipo de variable al que apunte.

Pues en este punto ya tenemos la memoria como en la anterior tabla y dos variables, una tipo char que contiene la letra 'm', y un puntero que apunta a esta última. Veamos ahora como usar esto.

Al definir el puntero vimos que se hacía con un asterisco *. Esto es siempre necesario al definir un puntero. Si ponemos un asterisco al definir una variable, esta se entenderá como un puntero que apuntará a una variable del tipo que le hayamos puesto. Pero vemos también que al decirle que apuntara a la variable carácter (puntero = &caracter;), no se usó el asterisco. ¿Cuándo se usa y cuando no?
Hay multitud de usos de un puntero, en algunos se ha de usar el asterisco y en otros no, así que para saber si debemos usarlo, mejor que aprender cada caso concreto, que es imposible, aprenderemos la regla general:

Siempre que definamos un puntero se usa un asterisco *.
Cuando lo usamos (después de definirlo), NO poner un asterisco significa usar la dirección de memoria a la que apunta, es decir, su contenido; y usar el asterisco significa acceder al valor de la dirección de memoria a la que apunta.

Entenderemos esto mejor con un ejemplo. Retomamos el caso anterior. Tenemos una variable (carácter), con el valor 'm' en la posición 0x00000002, y una variable (puntero), con el valor 0x00000002, es decir, apuntando a carácter, tal y como está en la tabla anterior.

Si nos referimos a la variable puntero sin el asterisco, nos estamos refiriendo, según la regla, a la dirección que ocupa. En realidad esto es

como con cualquier otra variable, cuando pones una variable, por ejemplo tipo int, lo que estás cambiando es su valor. Por ejemplo:

```
int numero = 5;
numero = 10;
```

Al poner numero nos referimos a su valor (5), y lo cambiamos a 10. En un puntero es igual, si lo pones tal cual, lo que estás cambiando es su valor, y el valor de un puntero es la dirección a la que apunta. Si hacemos:

```
printf("Dirección: %p\n", puntero);
//El %p significa que le pasamos un puntero
```

Nos mostrará el contenido de puntero, que es la dirección a la que apunta: 0x00000002.

El asterisco, según la regla, es el contenido de la dirección a la que apunta el puntero. Puesto que apunta un char, ponerlo con asterisco significa acceder al valor del char. Por ejemplo si queremos cambiar el contenido de "caracter", que tiene una 'm', por una 'c', y lo queremos hacer mediante el puntero, haremos:

```
*puntero = 'c';
```

El asterisco significa contenido la dirección (0x00000002), que es un 'm', y le pone un 'c', cuyo valor hexadecimal es 0x63. La tabla quedará así:

Dirección	Contenido
0x00000001	-
0x00000002	**0x63**
0x00000003	**0x00**
0x00000004	**0x00**
0x00000005	**0x00**
0x00000006	**0x02**

Creo que con esto ha quedado bastante claro el uso del asterisco, en cada caso particular tenemos que ver si nos interesa usar la dirección de memoria o nos interesa usar el valor de la variable a la que apunta.

Lo siguiente es un pequeño texto para explicar qué sentido tiene que un puntero deba tener el mismo tipo que la variable a la que apunta.
¿Por qué hay que darle entonces el mismo tipo que la variable? Esto tiene una respuesta simple. Las variables, independientemente del tipo, siempre contienen lo mismo vistas en memoria: un número hexadecimal. Lo que cambia es (a parte de la manera en la que el lenguaje las interpreta) el espacio que ocupa. Vamos a ver el problema con un ejemplo. Tenemos una variable int que ocupa 4 bytes (32 bits). Como 2 cifras hexadecimales representan un byte completo, tenemos 8 cifras hexadecimales. La variable tiene guardado el número 0x98FC3F4A. Los bytes, por razones de la estructura de la pila del procesador, algo que ahora no interesa, se guardan en sentido inverso, así que el número quedará en memoria así:

4A	3F	FC	98

Ahora creamos un puntero que apunte a la variable. El puntero en vez de hacerlo tipo int lo hacemos tipo char. El programa pensará que, al ser el puntero tipo char, está apuntando a una variable tipo char, es decir, de un byte. Osea que se quedará en tan solo el primer byte (4A).

Si cambiamos el valor de la variable a través del puntero, sólo afectará al último (por ser el primero en memoria), ya que cree que es un char y que sólo ocupa 1 byte en vez de cuatro. Si le damos por ejemplo el valor 0x43, la variable, en vez de pasar a ser 0x00000043, será 0x98FC3F43.

Puedes comprobar el efecto de esto compilando y ejecutando lo siguiente:

```
#include <stdio.h>

int main()
{
        int numero = 0x98fc3f4a;
        char *puntero_malo = &numero;

        printf("Número original: %#X\n",
                          numero);
        printf("Número desde puntero
                     original: %#X\n",
                                 *punter
o_malo);

        *puntero_malo = 0x43;

        printf("Número: %#X\n", numero);
        printf("Número desde puntero:
                              %#X\n",
*puntero_malo);

        return 0;
}
```

En mi ordenador la salida del programa es:

```
Numero original: 0x98FC3F4A
numero desde puntero original: 0x4A
numero: 0x98FC3F43
numero desde puntero: 0x43
```

Esa es la razón por la que los punteros se les tiene que poner el mismo tipo que la variable a la que apuntan. No dejes que todo esto de los bytes que se guardan al revés te líe, quédate con la teoría más o menos de cómo funciona un puntero y es suficiente.

5. Cuidados a tener en cuenta

Como todo en esta vida, los punteros tienen también sus inconvenientes, y es que manejar la memoria directamente puede ser "peligroso", y los fallos de programación originados por un mal uso de punteros suelen ser los más difíciles de encontrar y corregir. En realidad sólo de encontrar. En esta parte veremos algunos cuidados que tenemos que llevar a cabo a la hora de usar punteros.

Como dijimos al principio del documento, el ordenador vuelca en la RAM todos los programas que se tengan que ejecutar. Cuando un programa se libera de la memoria, es decir cuando cerramos un programa, la memoria que ocupaba se deja de tener en cuenta y pasa a ser memoria libre, que puede ser usada por otro programa, pero la información que tenía no tiene porqué borrarse.

Esto quiere decir que si definimos una variable y no le damos un valor, esta variable tendrá lo que tuviera la zona de memoria que se le reservó cuando fue usada por otro programa, es decir, es imposible saber lo que tiene una variable si no le hemos asignado un valor inicial. Prueba a ejecutar el siguiente programa.

```
#include <stdio.h>

int main()
{
        int number;
        printf("Número: %i\n", number);
}
```

Si lo ejecutas varias veces te dará una serie de números que desde luego no son cero y son diferentes entre sí. Esto ocurre porque hemos definido la variable, el ordenador le ha reservado un hueco en memoria pero no hemos guardado nada en ese hueco, no sabemos lo que hay.

Este problema con las variables no tiene mucha relevancia porque con las variables sólo modificamos el contenido de la variable, y esa zona de la memoria ya pertenece a nuestro programa, con lo que cualquier cambio es legítimo y no surgirán problemas. ¿Pasa lo mismo con los punteros?

Si definimos un puntero pero no hacemos que apunte a ninguna variable:

```
int *puntero;
```

El contenido de la variable puntero puede contener cualquier valor, y ese valor, puesto que es un puntero, será interpretado como una dirección de memoria. Tenemos una variable con una dirección de memoria aleatoria. Esto supone un problema muy grande. Si hacemos:

```
*puntero = 5;
```

Estamos diciendo que la variable a la que apunta valga 5, pero el puntero está apuntando a una dirección que no conocemos. Podría estar sobreescribiendo cualquier cosa, incluso partes del sistema operativo, lo que supondría un cuelgue.

Este problema se encuentra presente sobre todo en Windows, puedes probar a hacer eso, y verás como quizás tu ordenador se cuelga y lo tienes que reiniciar, o quizás la dirección a la que apunta no tenga nada importante y no pase nada. En sistemas GNU/Linux, aunque también hay que tener en cuenta estas cosas, no supone un peligro tan grande, porque el sistema avisará de una "Violación de segmento" y cerrará el programa. Violación de segmento significa que el programa ha intentado escribir en una zona de memoria más allá de la que el sistema le ha asignado, literalmente, se ha salido del segmento de memoria que el sistema le ha reservado.

Esta sección se resume en esto: cuidado cuando uses un puntero, ten en cuenta a dónde apunta y que siempre tenga un valor conocido.

En la sección siguiente explicaré cuidados concretos que hay que tener para los usos que doy, aunque sabiendo esto deberías ser capaz de suponerlos.

6. Algunos usos

Sería inútil intentar enumerar todos los usos que tienen los punteros. Tienen simplemente un montón y más. Aquí pondré algunos comunes con ejemplos para que puedas practicar y acostumbrarte a usarlos. Los punteros son sin duda el punto fuerte de C. Te permiten un control mucho más a fondo del ordenador, un control directo de la memoria que se puede aprovechar de mil y una maneras, y depende de ti el saber aprovecharlos en cada situación.

Pasar argumentos por referencia. (Sólo en C, no en C++)

Cuando creamos una función podemos ponerle argumentos, una serie de datos con los que la función trabaja, pero C siempre pasa argumentos por valor. Tenemos la siguiente función.

```
int suma(int a, int b)
{
        a += b;
        return a;
}
```

Cuando le pasemos una variable a los argumentos se pasará sólo su valor, quedando la variable intacta. Si hacemos:

```
int numero1 = 5;
int numero2 = 10;
int numero3 = suma(numero1,numero2);
```

A pesar de que dentro de la función a la variable a se le suma la variable b, la variable numero1 no sufre ningún cambio, esto es evidente.

¿Qué pasa si queremos que la variable numero1 si que sufra los cambios que se le hacen dentro de la función? En principio no podemos, porque sólo le estamos pasando el valor de la variable, y sólo podemos pasar valores, nunca la variable en si.
Pero tenemos nuestros maravillosos punteros. Una solución al problema es pasarle la dirección de memoria de la variable numero1, de modo

que dentro de la función tenemos la dirección de la memoria donde está guardado el valor y podemos modificarlo directamente. Para ello habrá que especificar el primer argumento como un puntero, ya que no le vamos a pasar un número sino una dirección de memoria. La función queda así:

```
int suma(int *a, int b)
{
       *a += b;
       return *a;
}
```

El argumento lo definimos como puntero poniéndole un asterisco. Dentro de la función, puesto que nos referimos al valor de la dirección a la que apunta (queremos cambiar el número), también usamos el asterisco. La función se usaría así.

```
int numero1 = 5;
int numero2 = 10;
int numero3 = suma(&numero1, numero2);
//Aquí la variable numero1 valdrá 15 ya que
//la hemos cambiado en la función
```

Como el argumento es un puntero a int (puntero que apunta a variable int), le tenemos que pasar la dirección de memoria de la variable, que se hace con el símbolo &.

En el título del caso he puesto que esto no vale en C++. En realidad sí que vale, pero es absurdo porque C++ ya tiene un sistema de paso de argumentos por referencia, que se hace definiendo los argumentos con el símbolo & y pasándole la variable de manera normal, pero esto no compete a este documento.

Recorrer arrays

Este es otro uso bastante importante y muy común.

Supongamos que tenemos un array de variables int formado por 100 elementos.

```
int numeros[100];
```

Antes de seguir hay que hacer una aclaración. Los arrays y los punteros son similares en uso. Si nos referimos a un array tan solo con su nombre, si ponemos numeros, sin especificar ningún índice, lo que nos va a devolver va a ser la posición de memoria del primer elemento, es decir, poner numeros es lo mismo que poner &numeros[0]. Y es importante saber para entender esto que los elementos de un array están colocados de manera lineal en la memoria, todos seguidos y en orden.

Podemos recorrer un array si creamos un puntero de tipo int y hacemos que apunte al primer elemento.

```
int *puntero = numeros;
// Sería lo mismo que hacer int *puntero =
// &numeros[0];
```

Ahora podemos sumarle y restarle posiciones al puntero (a la dirección que ocupa) como a una variable normal. Por ejemplo queremos que numero[5] valga 20 y numero[7] valga 30. Podemos hacer:

```
puntero += 5;
// Ahora puntero apunta a &numeros[5]
*puntero = 20;
puntero +=2;
// Ahora apunta a &numeros[5+2]
*puntero = 30;
```

Si queremos moverlo de vuelta a la primera posición, hacemos puntero = numeros; y ya está.

Cuando usamos un puntero para recorrer arrays conviene tener cuidado con no salirse del array. Si tenemos, siguiendo el caso anterior, un array "numeros[100]", y tenemos que el puntero apunta al último elemento, es decir: puntero = &numeros[99]; Si ahora hacemos puntero++; no pasará nada, simplemente le haremos que apunta a la dirección siguiente en la memoria,

pero esta dirección ya no pertenece al array. No tiene porqué pertenecer a la memoria que el sistema ha reservado a nuestro programa, o podría estar ya en uso por el mismo o por otro programa. Si intentamos escribir en ella podríamos estar sobreescribiendo algo importante y podríamos obtener una "Violación de segmento".

Este uso de los punteros puede no tener mucho sentido ya que lo podríamos haber hecho refiriénonos al array directamente.

```
numeros[5] = 20;
numeros[7] = 30;
```

Pero es muy importante para el siguiente caso.

Pasar arrays a funciones.

Quizás alguna vez se te haya presentado este problema. No podemos pasar un array completo a una función y poder acceder a él desde dentro de la función. De nuevo los punteros son la solución.
Siguiendo el caso anterior, podemos pasarle a la función por medio de un puntero la dirección de memoria del primer elemento y recorrerlo con el puntero. Por ejemplo un función que le sume a todos los elementos del array el número que le digamos:

```
void suma_array(int *puntero, int elementos,
int numero)
{
        int i;
        for (i=0;i<elementos;i++)
        {
                *puntero += numero;
                puntero++;
        }
}
```

Y problema resuelto, le pasamos el array y punto. Recuerda que es peligroso hacer que un puntero señale a una dirección en la que no sabemos qué hay. Así que, al igual que en el caso anterior, procura no salirte del array.

Memoria dinámica

Este uso de los punteros es uno de los más útiles en C. Nos permite pedirle al sistema que nos reserve más memoria desde el propio programa, en casos en los que no sabemos a priori cuántas variables necesitamos en nuestro programa.

Vamos a ver esto con un ejemplo sencillo, que es lo primero que yo vi cuando oí hablar de memoria dinámica, y que es bastante fácil de entender.

Supongamos que hacemos un programa que actúa como un libro de contactos, podemos crear contactos, borrarlos o leerlos. En este programa, al programador, o sea, nosotros, le es imposible saber de antemano cuántos contactos va a crear el usuario. Creamos la siguiente estructura en donde guardaremos cada contacto:

```
typedef struct contacto {
        char nombre[20];
        int telefono;
        char edad;
}
```

Si alguno de nuestros conocidos tiene más de 127 años, podemos cambiar el "char edad;" por un "unsigned char edad", y si tiene más de 255 ya lo podemos cambiar por un "int edad;"; resumiendo, que con un char para la edad nos sobra aunque sea un número.

Evidentemente, no vamos a crear un array enorme de estas estructuras, porque no queremos abusar de memoria, y porque nunca sabemos si el usuario llegará a sobrepasarlo. ¿Qué hacemos entonces?
Haremos que el sistema nos reserve memoria para cada nuevo contacto que creemos, y la

vaya borrando para cada uno que borremos.

Las funciones que utilizaremos para manejar memoria dinámica están definidas en la librería stdlib.h, que es estándar, aunque dada su importancia algunos compiladores las llevan ya definidas y no hace incluirlas en los ficheros. De todos modos, inclúyela siempre para evitarte un aviso del compilador (que lo dará si es un compilador decente), y para maximizar la compatibilidad.

De estas funciones, podemos distinguir en tres tipos de funciones, yo sólo explicaré dos tipos:
- Las funciones para asignar memoria, que son malloc y calloc.
- Las funcioner para reasignar memoria, realloc
- Y para liberar memoria, free.

Empecemos con las de asignación. Malloc, que viene de memory allocation es una función que reservará la memoria que le pidamos y nos devolverá un puntero apuntado a la primera dirección de las que haya reservado. La función malloc tiene el siguiente prototipo:

```
void *malloc(int n_bytes);
```

Como vemos, sólo pide un argumento, que es el número de bytes a reservar. La función devolverá un puntero al primer elemento de ellos, con lo que la mayoría de las veces haremos casting a estos punteros para adecuarlos a lo que necesitamos. Para poner un ejemplo, si queremos reservar espacio para cuatro variables tipo int, podemos hacer:

```
int *puntero;
puntero = (int *)malloc(sizeof(int)*4);
```

Utilizamos el operador sizeof para decirle cuánto espacio vamos a necesitar, y convertimos el puntero devuelto a un puntero a int poniendo (int *) antes de la función.
La función calloc es similar, su prototipo es el siguiente:

```
void *calloc(int n_datos, int tam_dato);
```

En este caso necesita dos argumentos, el n_datos, es como bien dice el nombre, el número de datos o elementos que queremos reservar, y tam_dato es el tamaño en bytes de cada uno de los elementos. Con el mismo ejemplo que antes, para reservar memoria para cuatro variables tipo int, haríamos:

```
int *puntero;
puntero = (int *)calloc(4, sizeof(int));
```

La diferencia entre calloc y malloc, además de su uso, es que calloc inicializa toda la memoria reservada a 0, mientras que malloc la reserva y la deja como está.

Estas dos funciones devolverá NULL si no consiguen reservar espacio suficiente, así que conviene que, después de usar cualquiera de las dos, hagamos algo como:

```
if (puntero == NULL)...
```

Y si se cumple, mostremos un mensaje diciendo que no se ha podido reservar la memoria y cerrar el programa o lo que sea.

La memoria reservada con estas dos funciones no volverá a quedar disponible hasta que no la liberemos nosotros, y es importante liberarla porque podríamos tener problemas de rendimiento. Para liberar la memoria usamos la función free, cuyo prototipo es así:

```
void free(void *);
```

Bien, volvamos a nuestro programa. Resolveremos el problema haciendo que el sistema nos reserve memoria para cada nuevo contacto y guardaremos la dirección de cada contacto en el anterior. La nueva estructura será así:

```
typedef struct contacto {
        char nombre[20];
        int telefono;
        char edad;
        struct contacto *siguiente;
}
```

Es importante inicializar el puntero a NULL, porque así es como sabremos que hemos llegado al final de la lista. Con este gráfico lo entenderás mejor:

Dirección: 0x001	Dirección: 0x01E	Dirección: 0x03C
Nombre = "Ana"	Nombre = "Carlos"	Nombre = "María"
Telefono = 3245234	Telefono = 1728320	Telefono = 2345234
Edad = 25;	Edad = 26;	Edad = 22;
Siguiente = 0x01E	Siguiente = 0x03C	Siguiente = NULL

En el puntero <siguiente> de cada contacto guardamos la dirección del siguiente contacto de la lista. El programa podría empezar:

```
struct contacto *primero;
primero = (struct contacto *)
                    malloc(sizeof(struct
contacto));
primero->siguiente = NULL;
printf("Introduce nombre: ");
scanf("%s", &primero->nombre);
…
//Así con todos los campos de la estructura.
```

En este punto ya hemos creado un contacto con cada uno de sus datos y un puntero que vale NULL.
Habrás notado que para cambiar los campos de la estructura no utilizamos un punto "." como hacemos con una estructura normal. Si tenemos un puntero que apunta a una estructura, la manera de acceder a uno de sus campos es con el operador "->", muy importante.

Si queremos crear otro contacto, lo que haremos será lo mismo, pero haciendo que malloc devuelva el puntero al campo siguiente del último contacto definido. Conviene siempre tener un puntero apuntando al primer elemento y que este puntero no cambie, de modo que podamos volver al comienzo de la lista con facilidad, así que el puntero "primero" lo dejaremos tal cual, cuidado con cambiar su valor porque perderemos la lista.

Necesitamos también un puntero apuntando al último elemento de la cola, de modo que al crear un nuevo contacto podamos ponerlo tras el último rápidamente, así que creamos ese puntero, que ahora mismo, como sólo tenemos uno, apuntará al mismo sitio:

```
struct contacto *ultimo = primero;
```

Ahora, crearemos una función que automatice la creación de un nuevo contacto:

```
void crear_contacto(){
       ultimo->siguiente=(struct contacto *)
                      malloc(sizeof(struct
contacto));
       ultimo = ultimo->siguiente;
      //Ahora ultimo apunta al nuevo contacto
       ultimo->siguiente = NULL;
       fflush(stdin);
       scanf("%s", ultimo->nombre);
       …
      //Lo mismo para los demás campos.
}
```

Con esta función crearíamos un nuevo contacto y el puntero "ultimo" apuntaría a él.
¿Cómo podríamos hacer para recorrer la lista entera de contactos mostrándolos por pantalla?
Ya tenemos dos punteros, uno que apunta al principio de la lista y otro que apunta al final, y cada elemento tiene un puntero que apunta al siguiente elemento. Podemos crear un tercer puntero que usaremos para recorrer la lista de manera de fácil. Una función que muestre toda la lista podría ser así:

```
void mostrar_lista() {
       struct contacto *apuntador = primero;
        //Lo ponemos en el primer elemento.
       while (apuntador->siguiente != NULL)
       {
         printf("Nombre...
         …
                    //Mostramos los campos con
printfs
         apuntador=apuntador->siguiente;
       }
}
```

Así de simple, cuando lleguemos al último, en el que el puntero siguiente = NULL, se cerrará el bucle.

Hasta aquí este uso, mira a ver si eres capaz de construir el programa completo que te permita crear borrar y listar contactos, y si surge alguna duda, ya estás preguntando en el foro.

Ahora veremos un último uso de la memoria dinámica que puede no ser muy intuitivo al principio, y con esto, acabaremos el documento. Si has llegado hasta aquí, ya no te puedes rendir, y no le puedes tener miedo a nada.

Vemos cómo los arrays son similares a los punteros en algunos aspectos. Si tenemos un puntero que apunta a varios elementos que están seguidos en memoria, podemos usar el puntero como si fuese un array. Vamos a verlo con un ejemplo.

Vimos que en memoria los elementos de más de un byte se guardaban al revés, a esto se llama Little-Endian, y es una manera en que trabaja el sistema que facilita el acceso a datos. Esta manera de guardar datos es la más común y es la que usan por ejemplo Intel y AMD, que son de los más comunes. Lo contrario, llamado Big-Endian, consiste en guardar las cosas en memoria tal cual, en el sentido convencional, y este método lo usan por ejemplo procesadores Motorola.

Vamos a ver el uso de un puntero como un array en un ejemplo sencillo para saber si nuestro sistema es Little-Endian o Big-Endian.

```
#include <stdio.h>
int main()
{
  int numero = 1;
//Esto en hexadecimal, en los cuatro bytes
//que ocupa, quedará: 0x00000001
  char *puntero = (char *)&numero;
//Hacemos casting para que interprete el int
//como 4 chars
  if (p[0] == 1)
    printf("Little-Endian\n");
  else
    printf("Big-Endian\n");
}
```

El ejemplo está claro, tenemos cuatro bytes, que suponiendo un little-endian, se guardarán así:

0x01
0x00
0x00
0x00

Como el char ocupa un byte y hemos hecho que interprete este int como un char, podemos usar el puntero como un array de cuatro elementos char, que son los cuatro bytes.
Si es little endian, p[0] valdrá 1, si es big endian, p[0] valdrá 0.

Ahora que ya sabemos esto, podemos empezar.
Los punteros pueden apuntar a cualquier tipo de variable, incluido otro puntero. Como hemos visto, el uso de memoria dinámica nos permite crear arrays en tiempo de ejecución. ¿Cómo haríamos para crear un array bidimensional? ¿Podemos crear una tabla de memoria dinámica? Claro que sí.

Lo que haremos será crear un array de punteros, y cada uno de esos punteros haremos que sea otro array, ya de variables. Esto bien podría extenderse a arrays de cualquier dimensión, siemplemente habría que anidar punteros y ya está.
Vamos a crear una tabla de 5x5 de variables int en memoria dinámica.

```
int **tabla;
filas = (int **) malloc (sizeof(int *)*5);
int a;
for (a = 0; a <5; a++)
{
  tabla[a] = (int *) malloc (sizeof(int)*5);
}
```

Y ya está. Podemos acceder a cualquier campo usando tabla como array bidimensional. Por ejemplo para darle un valor al campo 3x4:

```
tabla[3][4] = 5;
```

Ya está.

Sólo hay dos cosas que hay que tener en cuenta al hacer esto, además de tener cuidado con no salirse de los límites del array.
Una, no hay ninguna razón para pensar que la filas puedan estar consecutivas en memoria, así que no lo trates como tal. Lo único que está consecutivo en memoria son los punteros de cada columna, y dentro de las filas, cada campo, ya que cada cosa se ha reservado por separado, y el sistema le habrá reservado espacio donde hubiera disponible o donde le haya dado la gana.
Dos, cuando liberes la memoria, cosa que siempre hay que hacer cuando ya no la necesitemos, hay que liberar todos los arrays, igual que se han reservado. Para liberar la tabla anterior habría que hacer.

```
for (a = 0; a <5; a++)
{
        free(tabla[a]);
}
free(tabla);
```

Así nos aseguraremos de que todos los punteros a los que se les había reservado memoria han sido liberados.
Y ya está , ya hemos acabado.

7. Notas finales.

Si alguien ha leído este manual teniendo idea, habrá notado que hay cosas que no son realmente así o que no han sido explicadas del todo. He considerado que estas cosas no son del todo necesarias para entender el tema de este documento, pero que tener una idea global nos ayuda. Un ejemplo son las tablas a la hora de definir variables. He tratado ese tema de acuerdo a un sistema little-endian, mientras que he guardado como he querido. Estos datos se guardan en la pila, una zona de la memoria reservada al programa, de tipo LIFO (Last In First Out), que crece hacia abajo, es decir, hacia posiciones de memoria menores. En los ejemplos he seguido una escritura lineal para que sea más fácil entenderlo. Puede haber más casos.
Al resto de gente, le animo a investigar y aprender más sobre el tema para saber de verdad cómo funciona la ejecución de un programa. Cuanto más avances mejor entenderás todo.

Si encuentras algún error en el documento, ya sean erratas, fallos de explicación o fallos en los códigos, por favor comunícamelo por el foro (links en la introducción) para que puedan ser resueltos.

Finalmente, quiero agradecerte que hayas leído este documento, y te animo a preguntar cualquier duda en el foro.