
APUNTADORES, repaso.

Un apuntador, o puntero, es un identificador de programa, ya sea:

una variable
una constante o
una función,

cuyo contenido es una dirección. Ejemplos de apuntadores:

```
char *pChar;  
int *pEnt;  
double *pDoble;  
etc.
```

Cuando se trabaja con punteros surgen dos conceptos:

- **Dirección**
- **Indirección**

La **dirección** es el contenido en sí del identificador: la DIRECCION a la cual apunta.
La **indirección** es aquello que se halla REFERENCIADO o ALMACENADO en dicha dirección, o sea se trata de lo que está EN (ó IN) la dirección cargada en el puntero.

Asignación de direcciones a un apuntador.

Asignación directa:

Apuntador = Valor.

Esto suele ocurrir normalmente cuando se conoce de antemano la dirección de memoria: por ejemplo la memoria de video o ciertas posiciones bajas normalmente utilizadas por el sistema operativo para el reloj, el buffer de teclado, etc. Se debe tener mucho cuidado para no ingresar datos en direcciones equivocadas que produzcan funcionamiento anormal del programa o del sistema.

Asignación a través de operadores o funciones.

Apuntador = &Identificador (operador de dirección ampersand)

Para el ejemplo anterior tendríamos: **pPeso = &Peso**, donde el operador "&" retorna la dirección de su operando: en este caso la dirección de la variable Peso.
Es de destacar que este operador devuelve una dirección del mismo tipo del operando sobre el cual actúa: en este caso retorna un puntero a int.

Las Indirecciones.

La palabra **indirección** podríamos pensarla más bien como:

in-dirección (lo que se halla "en" la dirección)

Podemos asignar o referenciar los que se halla en la dirección apuntada por el identificador.

Asignación: ***pPeso = 2500**

Está diciendo: en la dirección apuntada por el puntero pPeso, almacene la magnitud 2500.

IMPORTANTE: Note bien la diferencia entre **pPeso=....** y ***pPeso=...** En el primer caso se asigna una **DIRECCION** a la variable en sí, y en el segundo caso se asigna **UN VALOR** en la dirección apuntada por la variable.

El dato almacenado en la dirección **pPeso** (o del puntero que se tratase) goza de todas las propiedades para ese tipo de dato, en este caso un int (entero):

- **Cociente de una División Entera.**
- **Resto de una División Entera.**
- **Puede utilizarse como operando en expresiones aritméticas.**
- **Etc.**

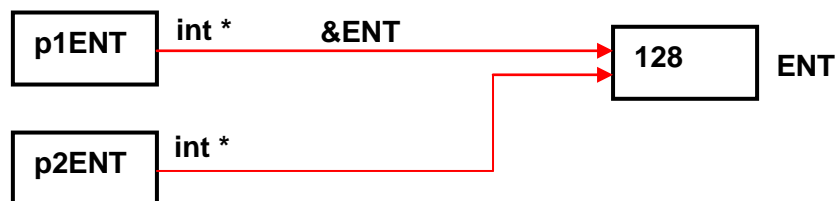
Ejemplos:

(*pPeso)*1000 Peso equivalente en gramos.
if((*pPeso) >=100) { }

En realidad el paréntesis no es necesario pero da mayor claridad a las operaciones.

Un ejemplo para aclarar dudas:

Considerar el siguiente esquema:



cuyo código vienen dado a continuación:

```
/* ----- Ejemplo de clase Nro 1 ----- */

#include<conio.h>

void main()
{
    int    *p1ENT;
    int    *p2ENT;
    int    ENT = 128;
```

```
clrscr(); highvideo();

p1ENT =&ENT;
p2ENT =p1ENT;

if(p1ENT==p2ENT) {
    cprintf("DIRECCIONES APUNTADAS IGUALES\r\n");
    getch();
}

if(*p1ENT==*p2ENT) {
    cprintf("CONTENIDOS REFERENCIADOS IGUALES\r\n");
    getch();
}
```

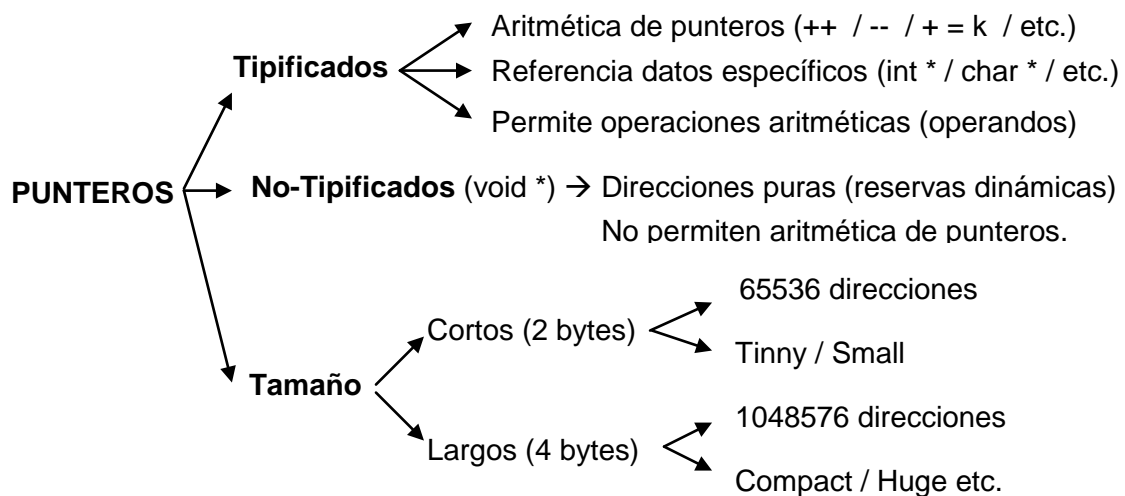
La dirección asignada al apuntador **p1ENT** se obtuvo mediante el operador "&", en tanto que la dirección de **p2ENT** fue tomada directamente del puntero **p1ENT**.

En la línea: **if(p1ENT==p2ENT)**, estamos preguntando si el contenido de la variable **p1ENT** es igual al de la variable **p2ENT**, o lo que es lo mismo, si la dirección apuntada por **p1ENT** es idéntica a la apuntada por **p2ENT**.

En cambio en la instrucción: **if(*p1ENT==*p2ENT)** averiguamos por el contenido de las direcciones apuntadas por **p1ENT** y **p2ENT**. Hay una diferencia neta con el caso anterior: aquí estamos comparando dos enteros y anteriormente comparábamos dos direcciones.

Clasificación de los punteros.

El siguiente cuadro nos permite tener una imagen general de los punteros:



¿Qué tipo de dato es una dirección?

Una dirección hace referencia a una posición específica de memoria, las cuales comienzan en 0 y se extienden hasta un cierto valor que depende de varias cosas, pero lo importante es que se trata de un **valor numérico entero**: específicamente es un **unsigned int**.

Aritmética de punteros.

La aritmética de apuntadores consiste en una sintaxis especial que produce cambios en la dirección apuntada.

Existen varias posibilidades:

int * pENT = (int *)&Vect[5]

Notación	Significado
pENT++	Incrementa en 1 la dirección.
pENT--	Decrementa en 1 la dirección.
++(*pENT)	Incrementa lo referenciado y luego utiliza dicho valor.
++(*pENT++)	Incr. lo referenciado, utiliza y vuelve a incrementar la dirección.
*pENT++	Utiliza lo referenciado e incrementa la dirección.
*(pENT++)	Usa lo referenciado e incrementa dirección.
(*pENT)++	Utiliza lo referenciado e incrementa lo referenciado.
*(pENT+i)	Utiliza lo referenciado y luego incrementa en "i" la dirección.

No conviene sobrecargar demasiado la notación debido a que va perdiendo claridad y resulta complicado cuando debemos depurar algún error.

Una diferencia importante:

Cuando hacemos:

pENT++ ó **pENT+ = i** estamos modificando el contenido del apuntador: si antes apuntaba en la dirección 10000, luego de aplicar los operadores ahora apuntan a 10004 ó a 10000+i, respectivamente. En cambio al hacer:

pENT[i]
***(pENT + i)**

El contenido del apuntador **no cambia**: continúa apuntando a la dirección original, y con respecto a ella consideramos tantos lugares hacia delante (o hacia atrás, si el caso lo requiere).

CAST's

Se denomina con este nombre a un rótulo entre paréntesis, que se coloca delante de un identificador de programa, normalmente una variable de algún tipo, a fin de forzar al compilador a realizar un cambio hacia "otro tipo" de dato, por ejemplo:

```
int    Veloc = 125;  
double Velocidad;
```

```
Velocidad=(double)Veloc;
```

Lo cual transforma a **Veloc** que era de tipo int, en un dato de tipo double.

Alguien podría considerar la aplicación de este cast totalmente innecesario debido a que "C" realiza una conversión automática de tipo...y tendría toda la razón. Sin embargo analicemos el siguiente caso:

```
int    Fuerza    = 125;  
int    Superf    = 63;  
double Presion;  
Presion = Fuerza/Superf;
```

El resultado obtenido sería 1 en lugar de 1.984 debido a que el compilador analiza que a la derecha del signo igual se realizará una operación donde todos los operandos son magnitudes enteras, para lo cual asigna buffers transitorios de ese tipo y ejecuta las operaciones basándose en enteros, en este caso un cociente entero: la parte 0.984 es directamente truncada y se queda con lo que está a la izquierda del signo igual. ¿Qué error se ha cometido por el redondeo? Nada menos que el 49,6% con respecto a lo esperado.

Estos errores son muy difíciles de detectar y producen serios dolores de cabeza a la hora de depurar. La solución surge de dos posibles acciones:

- a) Cambiar el "tipo" de las variables **Fuerza** y **Superf** a double.
- b) Utilizar cast's:

```
Presion = (double)Fuerza / (double)Superf
```

Con lo cual todo funciona bien y no se pierde la parte decimal.

Sin embargo una de las mayores aplicaciones de los cast's ocurre con los apuntadores, donde es muy común asignar a un puntero tipificado una dirección donde reside un dato de otro tipo.

```
/* ----- Ejemplo de clase Nro 3 ----- */
```

```
#include <conio.h>  
#include <stdio.h>
```

```
void main()  
{  
    int Fuerza = 125;  
    int Superficie = 63;  
    double Presion;  
  
    clrscr(); highvideo();
```

Presion = Fuerza/Superficie;

```
cprintf("Presion es:%5.2f \r\n", Presion);
```

```
getch();
```

```
}
```

```
// -----
```

Notación implícita de apuntadores.

Llamamos de esta manera a una notación especial en la cual no se ha declarado explícitamente un puntero, sino que la dirección es construida mediante una cierta sintaxis:

```
double UnDouble = M_PI;
```

```
unsigned int Direcc1;
```

```
Direcc1=(INT)&UnDouble;
```

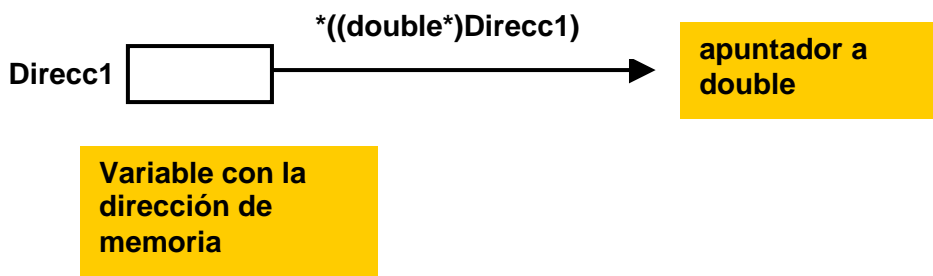
```
cprintf("Un Double = %lf\r\n",*((double *)Direcc1) );
```

Direcc1 es una variable de tipo unsigned int correspondiente de una dirección, pero falta explicitar que se trata de un puntero. Para ello utilizaremos un cast: **(double *)**. De esta manera:

(double *)Direcc1

Se trata de un apuntador a double. Si queremos tener acceso a lo que él referencia debemos indireccionar. La notación completa sería entonces:

***((double *)Direcc1)**



En el siguiente ejemplo se dispone de un conjunto de variables de distinto tipo a las cuales se le extrae su dirección mediante el operador **&**, y se las asigna a un arreglo de unsigned int. Posteriormente estas magnitudes serán reconvertidas en direcciones a través de las cuales se mostrará por pantalla los datos originales.

```
/* ----- Ejemplo de clase Nro 4 ----- */
```

```
#include <conio.h>
```

```
#include <math.h>
```

```
#include <string.h>
```

```
typedef unsigned int INT;

// -----
void main()
{
    INT      Direcc[4] = { 0,0,0,0 };
    double   UnDouble  =  M_PI;
    char     *UnaCadena = "UNA CADENA CUALQUIERA";
    int      UnEntero   =  15620;
    long     UnLong     =  6128635;

    clrscr(); highvideo();

    Direcc[0]=(INT)&UnDouble;
    Direcc[1]=(INT)UnaCadena;
    Direcc[2]=(INT)&UnEntero;
    Direcc[3]=(INT)&UnLong;

    printf(" Un Double = %lf\r\n",*((double *)Direcc[0]) );
    printf(" Una Cadena = %s\r\n",      (char *)Direcc[1] );

    printf(" Un Entero = %d\r\n",      *((int *)Direcc[2]) );
    printf(" Un Long   = %ld\r\n",     *((long *)Direcc[3]) );

    getch();
}
// -----
```

Nótese que la operación es siempre la misma:

Direcc[] = (INT)&Identificador

donde (INT) es un cast que convierte una dirección tipificada en su módulo y recién es almacenado en el arreglo. Un detalle interesante es que cuando se trata de una cadena no se requiere el operador & debido a que "C" maneja los arreglos como punteros implícitos llevando en su nombre la dirección de comienzo de la misma.

Arreglos y apuntadores.

Hemos dicho que los arreglos son un caso particular para "C", puesto que los maneja como punteros implícitos: el nombre del arreglo contiene la dirección del primer domicilio del mismo. Cuando declaramos:

int Buff[DIM];

Buff contiene la dirección de Buff[0]. Esto hace posible que un arreglo estático (el que acabamos de declarar) pueda ser manejado de la siguiente manera:

for(i=0;i<DIM;i++) *(Buff+i)=10+random(51);

cuando siempre estuvimos acostumbrados a escribir: **Buff[i]=10+random(51)**

Bueno, es exactamente lo mismo y funcionan correctamente. Con esto podemos inferir que:

Buff[i] = *(Buff + i)

La parte izquierda se denomina notación **subindexada** y la de la derecha **indexada**. Esto está íntimamente ligado a la tipificación. Cuando nosotros escribimos:

Buff + 1

Buf++

Buff+=k

siendo Buff un apuntador a int, el compilador “sabe” que debe incrementar la dirección como:

Buf++ = Buff + 1*sizeof(tipo del apuntador)

Buff + i = Buff + i*sizeof(tipo del apuntador)

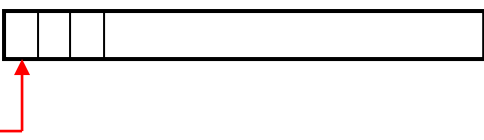
En cambio en los punteros no-tipificados (void*) el compilador no tiene forma de saber nuestras intenciones cuando le indicamos ++ ó Apuntador + i, por lo tanto a estos apuntadores no pueden aplicárseles aritmética de punteros.

Lo que **no puede hacerse** con el manejo de arreglos como punteros, es alterar la dirección del mismo. Por ejemplo hacer Buff =12500, puesto que intentaríamos **reasignar la dirección de inicio** con el riesgo de crear un gran perjuicio en el programa.

Si los elementos que van a almacenarse en el arreglo corresponden con el tipo declarado para el mismo, con la notación clásica subindexada: Buff[i] tenemos más que suficiente y no necesitamos complicarnos la vida. Sin embargo si vamos a almacenar otro tipo de dato como por ejemplo enteros, necesitamos forzosamente utilizar un puntero auxiliar o recurrir a la notación implícita mediante un cast.

Mediante un puntero auxiliar.

```
char Buff[DIM*sizeof(int)];  
int* pBuff;  
  
pBuff=(int *)&Buff[0];
```



A partir de aquí se aplica la aritmética de punteros como siempre:

Un detalle muy importante a tener en cuenta es el dimensionamiento correcto del arreglo teniendo en cuenta la cantidad de enteros que deseamos almacenar. Ello implica que si tendremos DIM domicilios de enteros, la cantidad de bytes necesarios será:

DIM*sizeof(int);

Para cargar el arreglo con enteros hacemos:

```
for(i=0;i<DIM;i+)  
    *pBuff++=10+random(41);
```


La notación `*pBuff++` posee el siguiente orden de precedencia (o prioridades):

Se utiliza la indirección

Se incrementa la dirección apuntada en 1 (ó sea `1*sizeof(int)`).

Si la dirección de comienzo de `pBuff` era 12500, la próxima será 12504, 12508, etc.

Mediante notación implícita de apuntadores.

Si no utilizamos un puntero específico a enteros tendremos que fabricarlo sintácticamente mediante cast's:

```
for(i=0;i<DIM;i++)  
    *( ((int *)&Buff[0])+i ) = 10 + random(41);
```

Hemos separado un poco los paréntesis para visualizar mejor la sintaxis:

`((int *)&Buff[0])`

transforma una dirección a char en una dirección a int. La línea anterior es, por lo tanto, un puntero a int (`int *`). En ese momento recién podemos incrementar su dirección con plena confianza sabiendo que la misma crecerá como `i*sizeof(int)` y no de otra manera. Este es un cuidado que debemos tener:

Convertir la dirección al tipo correcto y recién incrementar su dirección.

Lo trágico de esto es que si hacemos mal, o no hacemos la conversión de tipo, todo parece funcionar bien, pero no es así y el error es tan sutil que sólo un experto puede detectarlo.

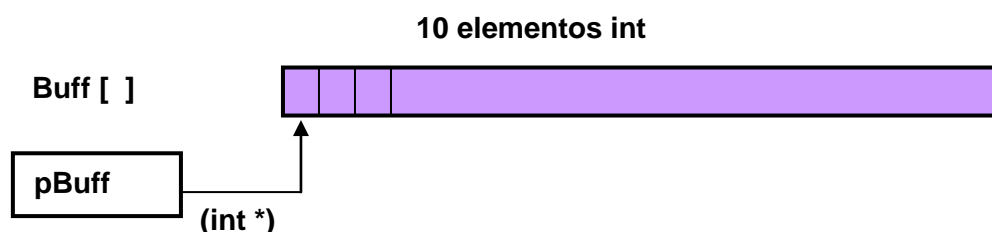
La notación: `*((int *)&Buff[0])+i` también puede reestructurarse como:

`((int *)&Buff[0])[i]`

que representa exactamente lo mismo, según vimos anteriormente. Aquí hemos mezclado un poco de notaciones puramente de apuntadores con la clásica subindexada de arreglos.

El siguiente ejemplo engloba todo lo visto hasta aquí

Siempre es conveniente hacer un esquema para facilitar la visualización del problema:



```
#include<conio.h>
#include<stdlib.h>

const DIM = 10;

// -----

void main()
{
    char Buff[DIM*sizeof(int)];
    int* pBuff;
    int i;

    clrscr(); highvideo(); randomize();

    pBuff=(int *)&Buff[0];

    // --- PRIMERA FORMA DE RECORRER UN ARREGLO -----
    cprintf("\r\nusando puntero auxiliar\r\n");
    for(i=0;i<DIM;i++)
    {
        // *(pBuff+i)=10+random(41);
        // cprintf("%4d", *(pBuff+i));
        cprintf("%4d", *(pBuff+i)=10+random(41));
    }
    cprintf("\r\nusando puntero implicito\r\n");
    //usando puntero implicito
    for(i=0;i<DIM;i++)
    {
        // *(pBuff+i)=10+random(41);
        // cprintf("%4d", *(pBuff+i));
        cprintf("%4d", *((int*) &Buff[0]+i));
    }
    cprintf("\r\nusando puntero implicito subindexado\r\n");
    //usando puntero implicito subindexado
    for(i=0;i<DIM;i++)
    {
        // *(pBuff+i)=10+random(41);
        // cprintf("%4d", *(pBuff+i));
        cprintf("%4d", ((int*)&Buff[0])[i]);
    }
    cprintf("\r\n\r\n");
    // --- SEGUNDA FORMA DE RECORRER UN ARREGLO -----

    for(i=0;i<DIM;i++)
        cprintf("%4d", ((int *)Buff)[i]);
    cprintf("\r\n\r\n");

    // --- TERCERA FORMA DE RECORRER UN ARREGLO -----

    for(i=0;i<DIM;i++)
```

```
cprintf("%4d", *((int *)Buff+i));  
  
getch();  
}
```

La instrucción: **cprintf("%4d",*(pBuff+i)=10+random(41));**

Realiza dos tareas a la vez: asigna un valor a lo que está referenciando el apuntador y a continuación lo muestra por pantalla. Esta propiedad de anidar varias instrucciones una dentro de otra, se denomina **implicitación**.

La asignación en sí es: **" , *(pBuff+i)=10+random(41)) ;**

Esta notación para incrementar la posición de un apuntador se llama **indexada** puesto que utiliza un índice.

Números aleatorios.

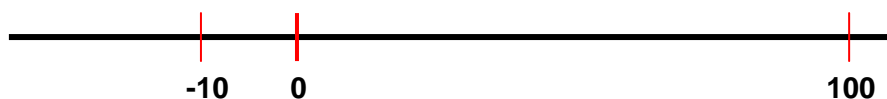
La librería **stdlib.h** posee funciones especiales para la generación de números aleatorios:

randomize()	Inicializa el secuenciador aleatorio.
random(int N)	Genera un entero aleatorio entre 0 y (N-1).

La instrucción **randomize()** debe utilizarse una sola vez, normalmente como una de las primeros comandos del **main()**, y asegura que cada vez que se corra el programa se generará una secuencia distinta al utilizar **random()**.

En cuanto a **random()** puede utilizarse de varias maneras útiles:

Imaginemos que deseamos una secuencia de aleatorios entre -10 y 100. Conviene siempre hacer un sencillo esquema que nos guíe:



Cuando el aleatorio sea igual a 0 (cero) deberemos tener el valor 10, y cuando el aleatorio sea máximo deberemos tener 100. Ello nos lleva a:

$$N = -10 + \text{random}(x) \rightarrow 100 = -10 + \text{random}(x) \rightarrow \text{random}(x) = 100 + 10 = 110$$

lo cual equivale a decir que $x = 111$ puesto que genera valores entre 0 y (N-1). Finalmente: **N = 10 + random(111).**

Este razonamiento deberá aplicarse para cualquier rango.

Ahora cabe la pregunta por el millón ¿podrán generarse valores aleatorios decimales?
Respuesta: SI.

```
double RandDec(int Linf, int Lsup, int NDec)
{
    int    Divisor=1;
    int    i;
    double xRand;

    for(i=1;i<=NDec;i++) Divisor*=10;

    xRand=(double)Linf+random(Lsup-Linf+1)+
        (double)random(Divisor+1)/(double)Divisor;

    return(xRand);
}
```

La generación aleatorio deberá hacerse por partida doble: para la parte entera y para la parte decimal:

Parte Entera = (double)(Linf + random(Lsup-Linf+1))

Parte Decimal = (double)random(Divisor + 1) / (double) Divisor

y luego sumarse a fin de adicionar la parte decimal a la parte entera.