**PUNTEROS**

     Un puntero es una variable que contiene la dirección de memoria de un dato o de otra variable que contiene al dato. Quiere esto decir, que el puntero apunta al espacio físico donde está el dato o la variable.

     Un puntero puede apuntar a un objeto de cualquier tipo, como por ejemplo, a una estructura o a una función. Los punteros se pueden utilizar para referenciar y manipular estructuras de datos, para referenciar bloques de memoria asignados dinámicamente y para proveer el paso de argumentos por referencia en las llamadas a funciones.

     Muchas funciones de la biblioteca de C++ tienen parámetros que son punteros y devuelven un puntero. Como ejemplo, recuerde la sintaxis de la función **scanf**o de la función **strcpy**.

     Cuando se trabaja con punteros son frecuentes los errores debidos ala creación de punteros que apuntan a alguna parte inesperada, produciéndose una violación de memoria. Por lo tanto, debe ponerse la máxima atención para que esto no ocutra, inicializando adecuadamente cada uno de los punteros que utilicemos.

CREACION DE PUNTEROS:

     Un puntero se declara anteponiendo al identificador que nombra al puntero, el modificador x, el cual significa "puntero a". Un puntero inicializado correctamente siempre apunta a un objeto de un tipo particular.

     Un puntero no inicializado no se sabe a donde apunta. La sintaxis para declarar un puntero es:

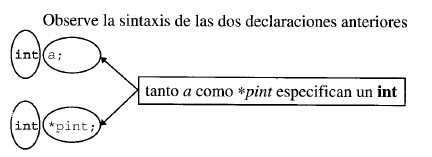
tipo \*var-puntero;

     Donde tipo especifica el tipo del objeto apuntado, puede ser cualquier tipo incluyendo tipos definidos por el usuario, y var-puntero es el identificador de la variable puntero.

*Por ejemplo:*

int a;        /\* a es una variable entera \*/  
int \*pint;   /\* pint es un puntero a un entero \*/  
pint= &a   /\* pint igual a la dirección de a; entonces, pint apunta a la variable a \* /

     La declaración de "a" ya nos es familiar. La declaración del puntero pint, aunque también la hemos visto en más de una ocasión, es nueva en lo que se refiere a su utilización como una variable más.



     Observamos que \*pint es un nemotécnico que hace referencia a un objeto de tipo int, por lo tanto puede aparecer en los mismos lugares donde puede aparecer un entero. Es decir, si pint apunta al entero a, entonces \*pint puede aparecer en cualquier lugar donde pueda hacerlo a.

     En conclusion \*pint es un entero que está localizado en la dirección de memoria especificada por pint.

     Este razonamiento es aplicable a cualquier declaración por complicada que sea.

*Otro ejemplo:*

double sqrt (double) , \*p, d, x;

p=&x;

//...

d = sqrt(x) + \*p;

printf("%g", \*p);    /\* imprime el valor actual de x \*/

      Este ejemplo pone de manifiesto que sqrt(x) y \*p representan valores de tipo double en la expresión donde aparecen; es decir, son equivalentes a variables de tipo double.

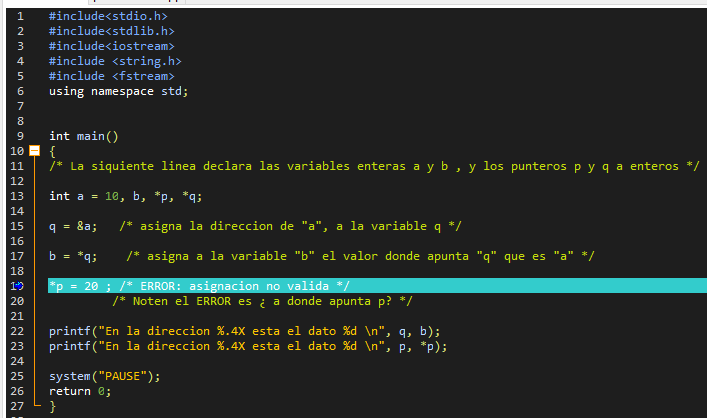
     El espacio de memoria requerido para un puntero, es el número de bytes necesarios para especific¿rr una dirección máquina. Son valores típicos 2 o 4bytes.

**Operadores**:

     Los ejemplos que hemos visto hasta ahora, ponen de manifiesto que en las operaciones con punteros intervienen frecuentemente el operador dirección de (&) y el operador de indirección (\*).

     El operador unitario &, devuelve como resultado la dirección de su operando y el operador unitario \*, toma su operando como una dirección y nos da como resultado su contenido.

*Por ejemplo:*



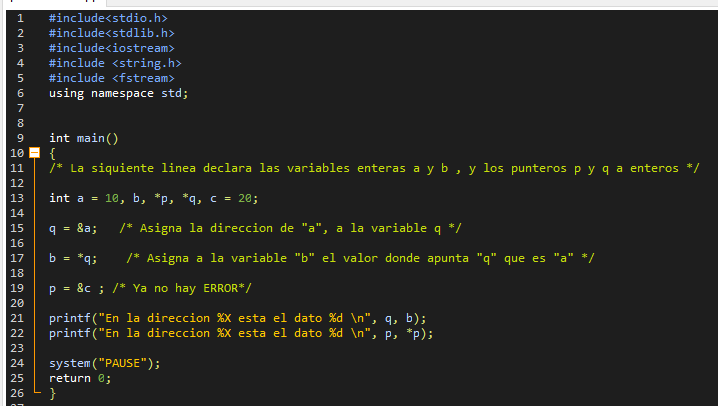
     En teoria, si ejecutamos este programa el resultado seria analogo al siguiente, donde %.4X se refiere a su valor hexadecimal:

En la direccion hexadecimal 6FFE04 esta el dato 10.

En la direccion hexadecimal 6FFE00 deberia estar el dato 20.

     Pero, en la prácfica estamos cometiendo un error por utilizar un puntero p sin saber a dónde apunta. Sabemos que "q" apunta a la variable "a"; dicho de otra forma, "q" contiene una dirección válida, pero no podemos decir lo mismo de "p" ya que no ha sido inicializado y por lo tanto su valor es desconocido para nosotros; posiblemente sea una dirección ocupada por el sistema y entonces lo que estamos haciendo es sobreescribir el contenido de esa dirección con el valor 20, lo que seguramente nos ocasionará problemas.

     Para solucionar este problema vamos a apuntar el puntero "p" a una direccion valida, en este caso creo la variable "c"= 20 y apunto p a dicha dirección de memoria:



**Operaciones con PUNTEROS:**

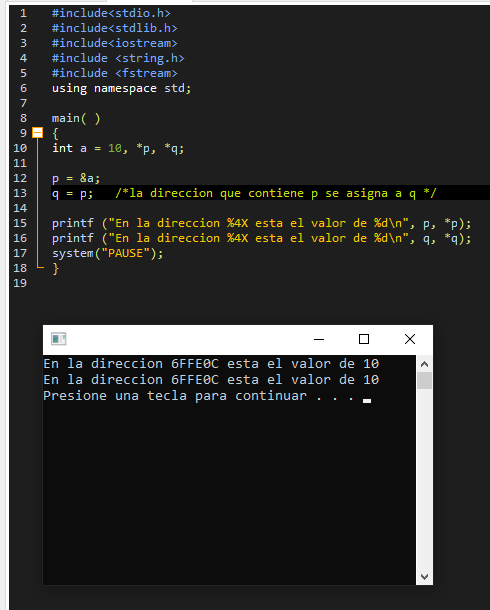
     Si **p**es un puntero, p++ incrementa **p**para que apunte al siguiente elemento de los que apunta p; es decir, C++ sabe que tiene que avanzar un número de bytes igual al tamaño de un objeto de los que apunta p. La aritmética de direcciones es una de las principales virtudes de C++. A un puntero se le puede asignar otro puntero.

     Tengamos en cuenta que si:

int i, j, \*p;  // p es un puntero.  
p=&i;         // p apunta a la direccion de i.  
\*p=10;      // i toma el valor 10.  
p=&j;        // p apunta a la direccion de j.  
\*p=11;      // j toma el valor 11.

\*El mismo puntero "p" primero apunta a una direccion, las variables "i" y "j", y luego toma primero un valor 10 y luego otro valor 11 mientras que esta apuntando a esas variables.

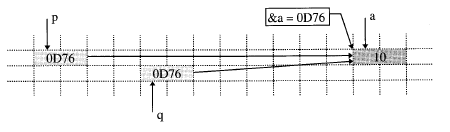
*Por ejemplo:*



     Supongamos que el resultado de este programa fuera análogo al siguiente ejemplo:

En la dirección 0D76 está el valor 10.

     Gráficamente puede observar que después de ejecutarse la asignación, "p" y "q" apuntan a la misma localización de memoria, a la variable "a". Por lo tanto, a, \*p y \*q son el mismo dato; es decir, 10.



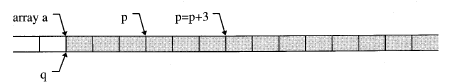
Operaciones aritméticas:

     A un puntero se le puede sumar o restar un entero.

*Por ejemplo:*

int a[100];  
int \*p, \*q;    /\* declara p y q como un puntero a un entero \*/  
p = &a[3];   /\* p apunta a a[3] \*/  
q = &a[0];   /\* q apunta a a[0] \*/  
p++;           /\* hace que p apunte al siguiente entero; a a[4] \*/  
p--;             /\* hace que p apunte al entero anterior; a a[3] \*/  
p = p+3;      /\* hace que p avance tres enteros; apunta a a[6]  \*/  
p = p-3;       /\* hace que p retroceda tres enteros; apunta a a[3] \*/

     Si "p" y "q" son variables de tipo puntero y apuntan a elementos del mismo array, la operación p - q es válida en el ejemplo, el resultado es 3. No se permite sumar, multiplicar, dividir o rotar punteros y tampoco se permite sumarles un número real.



Comparación de punteros:

     Es posible comparar dos punteros en una expresión de relación. Esta operación tiene sentido si ambos punteros apuntan a elementos del mismo array.

Por ejemplo:

int n, \*p, \*q, a[100];//  
...p = a[99];q = a[0];//  
...if (q + n <= p) q += n;  
if (q != NULL && q <= p)    /\* NULL es una constante que identifica a un puntero nulo \*/  
q++;

     La primera sentencia if indica que el puntero "q" avanzará n elementos si se cumple que la dirección q+n es menor o igual que "p".

     La segunda sentencia if indica que q pasará a apuntar al siguiente elemento del array si la dirección por él especificada no es nula y es menor o igual que la especificada por "p".

Puntero nulo

     En general, un puntero se puede inicializar como cualquier otra variable, aunque los únicos valores Significativos son NULL o la dirección de un objeto previamente definido. NULL es una constante definida en el fichero stdio.h así:

#define NULL ((void \*)0)  /\* definición de NULL en C \*/  
#defíne NULL 0                /\* definición de NULL en C++ x /

     El lenguaje c++ garantiza que un puntero que apunte a un objeto válido nunca tendrá un valor cero. El valor cero se utiliza para indicar que ha ocurrido un error; en otras palabras, que una determinada operación no se ha podido realizar.

*Por ejemplo:*

     Recordemos como la función **gets** cuando lee la marca de fin de fichero retorna un puntero nulo, indicando así que no hay más datos para leer.

     En general, no tiene sentido asignar enteros a punteros porque quien gestiona la memoria es el sistema operativo y por lo tanto es él, el que sabe en todo momento qué direcciones están libres y cuáles están ocupadas.

*Por ejemplo:*  
     int \*px = 10382;    /\* se inicializa px con la direcclón 10382 \*/

     La asignación anterior no tiene sentido porque ¿qué sabemos nosotros acerca de la dirección 10382?

**Punteros constantes:**

     Una declaración de un puntero precedida por const, hace que el objeto apuntado sea una constante, no sucediendo lo mismo con el puntero.

*Por ejemplo:*

const char \*pc = "abcd";    /\* objeto "abcd', constante y pc variable\*/  
pc[0] = 'z';                        /\* error; la cadena apuntada por pc es constante \*/  
pc = "efg";                        /\* correcto; pc pasa a apuntar a una nueva cadena \*/

     Si lo que se pretende es declarar un puntero constante, procederemos así:

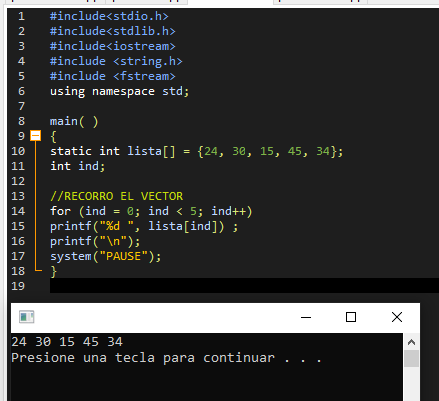
char \*const pc = "abcd"; /\* objeto "abcd" variable y pc constante \*/  
pc[0] = 'z';                     /\* correcto; la cadena apuntada por pc es variable \*/  
pc = "efg";                     /\* error; pc es constante \* /

     Para hacer que tanto el puntero como el objeto apuntado sean constantes, procederemos como se indica a continuación:

const char \*const pc = "abcd";    /\* objeto y puntero constantes \*/  
pc[0] = 'z';                                /\* error; el objeto "abcd" es contante\*/  
pc = "efg";                                /\* error; el puntero pc es constante \*/

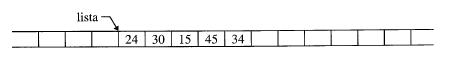
**Punteros y Arrays:**

     En C++ existe una relación entre punteros y arrays tal que cualquier operación que se pueda realizar mediante la indexación de un array, se puede realizar también con punteros.  
     Para clarificar lo expuesto, analicemos el siguiente programa, realizado primeramente con arrays y a continuación con punteros.

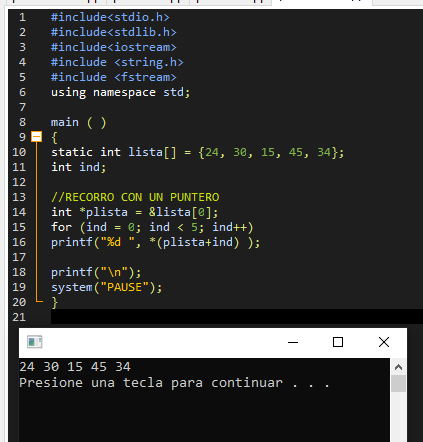


\*RECORRO NORMALMENTE UN VECTOR

     En este ejemplo se ha utilizado la indexación del array para acceder a sus elementos; expresión lista[ind]. Cuando C interpreta esta expresión sabe que a partir de la dirección de comienzo del array (a partir de lista) tiene que avanzar ind elementos para acceder al contenido del elemento especificado por ese índice.



      A continuación se expone la versión con punteros:



\*RECORRO CON UN PUNTERO (\*plista).

     Esta versión es idéntica a la anterior, excepto que la expresión para acceder a los elementos del array es ahora \*(plista+ind).

     La asignación plista = &lista[0] hace que plista apunte al primer elemento de lista; es decir, plista contiene la dirección del primer elemento, que coincide con la dirección de comienzo del array lista. Por lo tanto, en lugar de la expresión \*(plista+ind), podríamos utilizar también la expresión \*(lista+ind).

     Según lo expuesto, las siguientes expresiones dan lugar a idénticos resultados:

listat[ind]

\* (lista+ind)

plista[ind]

\*(plista+ind)

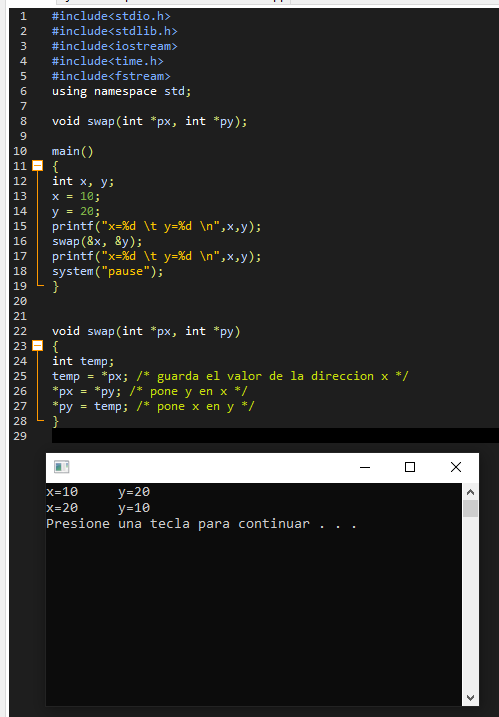
     Como pueden observar hay muchas formas de manejar la información de direccionamiento de datos al utilizar indices y punteros.

**Punteros y Funciones:**

      Cuando C pasa argumentos a funciones, los pasa por valor, es decir, si el parámetro es modificado dentro de la función, una vez que termina la función el valor pasado de la variable permanece inalterado.

     Hay muchos casos que se quiere alterar el argumento pasado a la función y recibir el nuevo valor una vez que la función ha terminado. Para hacer lo anterior se debe usar una llamada por referencia, en C++ se puede simular pasando un puntero al argumento. Con esto se provoca que la computadora pase la dirección del argumento a la función.

     Para entender mejor lo anterior consideremos la función swap() que intercambia el valor de dos argumentos enteros:



**Punteros y arreglos**

     Existe una relación estrecha entre los punteros y los arreglos. En C, un nombre de un arreglo es un índice a la dirección de comienzo del arreglo. En esencia, el nombre de un arreglo es un puntero al arreglo.

Considerar lo siguiente:

int a[10], x;

int \*ap;

ap = &a[0]; /\* ap apunta a la direccion de a[0] \*/

x = \*ap; /\* A x se le asigna el contenido de ap (a[0] en este caso) \*/

\*(ap + 1) = 100; /\* Se asigna al segundo elemento de 'a' el valor 100 usando ap\*/

     Como se puede observar en el ejemplo la sentencia a[t] es idéntica a ap+t. Se debe tener cuidado ya que C no hace una revisión de los límites del arreglo, por lo que se puede ir fácilmente más alla del arreglo en memoria y sobreescribir otras cosas.

     C sin embargo es mucho más sútil en su relación entre arreglos y apuntadores.

     Por ejemplo se puede teclear solamente:

ap = a; en vez de ap = &a[0]; y también \*(a + i) en vez de a[i], esto es, &a[i] es equivalente con a+i.

Y como se ve en el ejemplo, el direccionamiento de apuntadores se puede expresar como:

a[i] que es equivalente a \*(ap + i)

      Sin embargo los apuntadores y los arreglos son diferentes:

* Un apuntador es una variable. Se puede hacer ap = a y ap++.
* Un arreglo NO ES una variable. Hacer a = ap y a++ ES ILEGAL.

     \*Este parte es muy importante, asegúrese haberla entendido.

     Con lo comentado se puede entender como los arreglos son pasados a las funciones. Cuando un arreglo es pasado a una función lo que en realidad se le esta pasando es la localidad de su elemento inicial en memoria.

     Por lo tanto:

strlen(s) es equivalente a strlen(&s[0])

Esta es la razón por la cual se declara la función como:

int strlen(char s[]); y una declaración equivalente es int strlen(char\*s);

ya que char s[] es igual que char \*s.

    La función strlen() es una función de la biblioteca estándar que regresa la longitud de una cadena. Se muestra enseguida la versión de esta función que podría escribirse:

int strlen(char \*s)  
{  
char \*p = s;

while ( \*p != '�' )

p++;

return p s;  
}

     Se muestra enseguida una función para copiar una cadena en otra. Al igual que en el ejercicio anterior existe en la biblioteca estándar una función que hace lo mismo.

void strcpy(char \*s, char \*t)  
{  
while ( (\*s++ = \*t++) != '�' );  
}

     En los dos últimos ejemplos se emplean apuntadores y asignación por valor.

      \*Nota: Se emplea el uso del caracter nulo con la sentencia while para encontrar el fin de la cadena.

**Arreglos de Punteros**

     En C++ se pueden tener arreglos de apuntadores ya que los apuntadores son variables. El uso más común es el de formar arreglos de cadenas de caracteres, esto significa que cada entrada en el arreglo es un apuntador al primer caracter de la cadena

     Por ejemplo:

char \* mensaje[4] = {''Hola'',''Adios'',''Bye'',''Salut''}

\*Cada cadena está almacenada en memoria como una cadena de caracteres terminada en NULL.

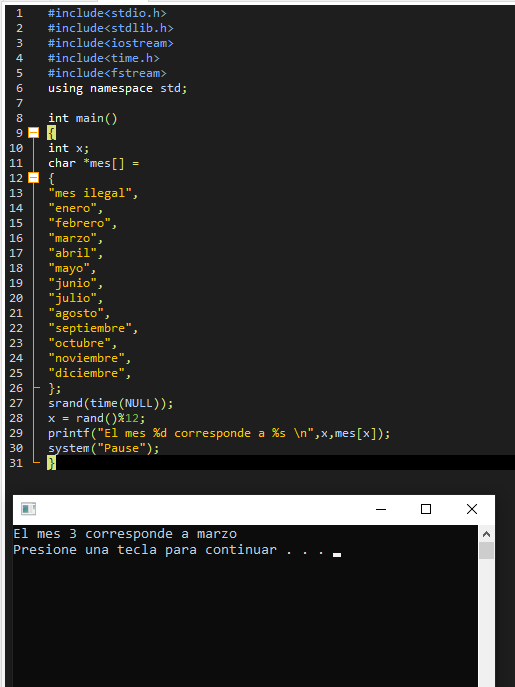
     En este el array no estan colocadas las cadenas, tan solo están almacenados los apuntadores.

    Aunque el array es de tamaño fijo, permite el acceso a cadenas de caracteres de cualquier longitud porque transforma su contenido a una variable de apuntador.

     Ejemplo2:

     En este ejemplo veremos como mi puntero \*mes puede apuntar a un valor que tiene asignado en este caso un string con los meses del año.

     Para hacerlo mas entretenido voy a utilizar un rand que me permita obtener una variable aleatoria que vaya de 0 a 12, siendo "0" el valor apuntado como "mes ilegal".



**Arreglos multidimensionales y apuntadores**

      Un arreglo multidimensional puede ser visto en varias formas en C, por ejemplo:

      Un arreglo de dos dimensiones es un arreglo de una dimensión, donde cada uno de los elementos es en sí mismo un arreglo.

Por lo tanto, la notación:

a[n][m] nos indica que los elementos del arreglo están guardados renglón por renglón.

     Cuando se pasa una arreglo bidimensional a una función se debe especificar el número de columnas el número de renglones es irrelevante.

     La razón de lo anterior, es nuevamente los apuntadores. C requiere conocer cuantas son las columnas para que pueda brincar de renglón en renglón en la memoria.

     Considerando que una función deba recibir int a[5][35], se puede declarar el argumento de la función como:

f( int a[][35] ) { ..... }  
o aún  
f( int (\*a)[35] ) { ..... }

      En el último ejemplo se requieren los parénteis (\*a) ya que [ ] tiene una precedencia más alta que \*.

Por lo tanto:

int (\*a)[35]; declara un apuntador a un arreglo de 35 enteros, y por ejemplo si hacemos la siguiente referencia a+2, nos estaremos refiriendo a la dirección del primer elemento que se encuentran en el tercer renglón de la matriz supuesta, mientras que:

int \*a[35]; declara un arreglo de 35 apuntadores a enteros.

     Ahora veamos la diferencia (sutil) entre apuntadores y arreglos. El manejo de cadenas es una aplicación común de esto.

Considera:

char \*nomb[10];  
char anomb[10][20];

En donde es válido hacer nomb[3][4] y anomb[3][4] en C.

Sin embargo:

1. anomb es un arreglo verdadero de 200 elementos de dos dimensiones tipo char.
2. El acceso de los elementos anomb en memoria se hace bajo la siguiente fórmula 20\*renglon + columna + dirección\_base
3. Encambio nomb tiene 10 apuntadores a elementos.

\*NOTA: si cada apuntador en nomb indica un arreglo de 20 elementos entonces y solamente entonces 200 chars estarán disponibles (10 elementos).

     Con el primer tipo de declaración se tiene la ventaja de que cada apuntador puede apuntar a arreglos de diferente longitud.  
Considerar:

char \*nomb[] = { "No mes", "Ene", "Feb", "Mar", .... };  
char anomb[][15] = { "No mes", "Ene", "Feb", "Mar", ... };

     Lo cual gráficamente se muestra en la figura 14.1. Se puede indicar que se hace un manejo más eficiente del espacio haciendo uso de un arreglo de apuntadores y usando un arreglo bidimensional.

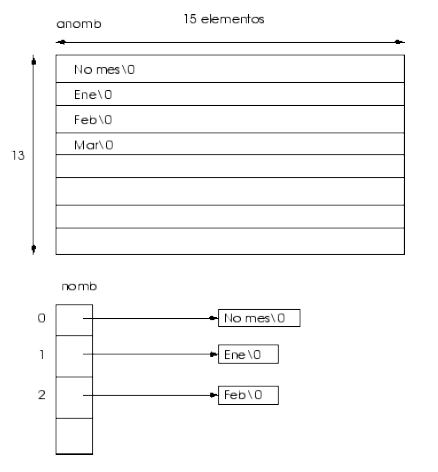


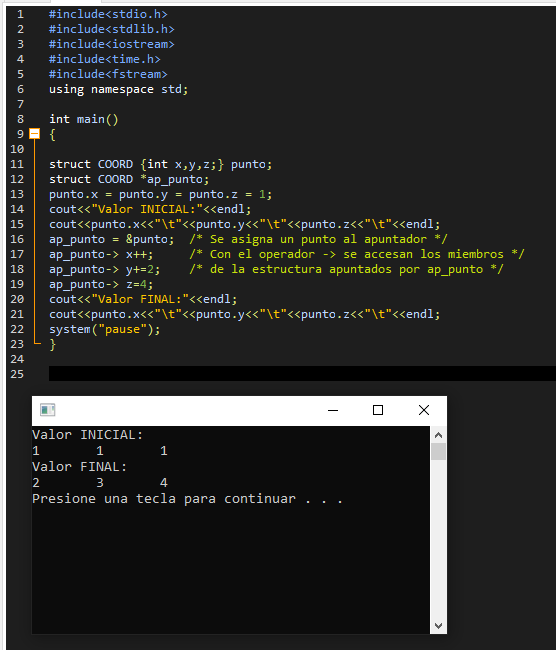
Figura 14.1: Arreglo de 2 dimensiones VS. arreglo de apuntadores.

**Punteros y estructuras**

     Los Punteros a estructuras se definen fácilmente y en una forma directa. Una estructura al contrario que los *arrays*, nos permiten agrupar varios datos, que mantengan algún tipo de relación, aunque sean de distinto tipo, permitiendo manipularlos todos juntos, usando un mismo identificador, o cada uno por separado.

     Las estructuras son llamadas también muy a menudo registros, o en inglés *records*. Tienen muchos aspectos en común con los registros usados en bases de datos. Y siguiendo la misma analogía, cada objeto de una estructura se denomina a menudo campo, o *field*.

Ejemplo:



     En este ejemplo podemos comprender sencillamente como se puede utilizar un puntero \*ap\_punto que apunta a una estructura "COORD" y cuyo valores estan denominados por el parametro "punto" para realizar asignaciones en una estructura.

     Sencillamente lo que haremos es primero apuntar ap\_punto a los elementos de la estructura llamada "punto" y luego utilizaremos el elemento de asignacion (raya flecha) para indicar a quien voy a apuntar, para finalmente realizar la asignacion de valores. De esta forma los elementos "x,y,z" que en un principio valen 1, luego de que el puntero me permita realizar ecuaciones aritmeticas valdran 2,3 y 4 sucesivamente.

     Otro ejemplo son las listas ligadas:

typedef struct { int valor;  struct ELEMENTO \*sig; } ELEMENTO;  
ELEMENTO n1, n2;  
n1.sig = &n2;

     La asignación que se hace corresponde a la figura 14.2:

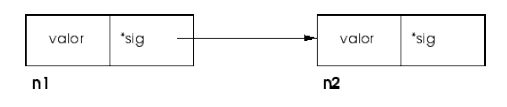


Figura 14.2: Esquema de una lista ligada con 2 elementos.

\*Nota: Solamente se puede declarar sig como un apuntador tipo ELEMENTO. No se puede tener un elemento del tipo variable ya que esto generaría una definición recursiva la cual no esta permitida. Se permite poner una referencia a un apuntador ya que los los bytes se dejan de lado para cualquier apuntador.