ARREGLO DE PUNTEROS

Un arreglo de punteros es una arreglo como cualquier otro, con la diferencia que el contenido es un puntero.
Se lo define de la siguiente manera:

<tipo> *Nombre[DIM];

Por ejemplo la declaración para un arreglo de punteros **int** de tamaño 10 es:

int *x[10];

Para asignar una dirección de una variable entera llamada **var** al tercer elemento de arreglo de punteros, se escribiría:

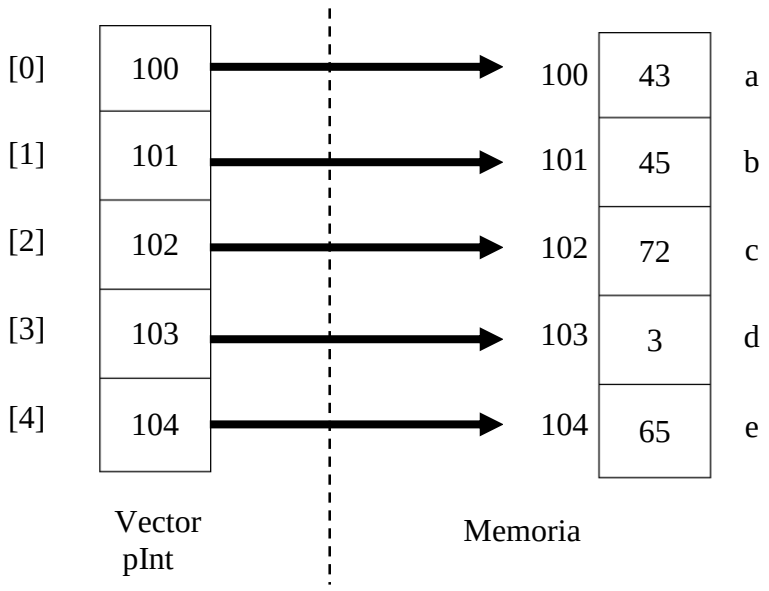
x[2]=&var;

Para modificar el valor de **var**, haría:

***x[2]=15;**

Ejemplo 1

```
int* pInt[5];

int a=43, b=45, c=72, d=3, e=65;

pInt [0] = &a;
pInt [1] = &b;
pInt [2] = &c;
pInt [3] = &d;
pInt [4] = &e;
```



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

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

const DIM=5;

void main()
{

int* pInt[5];

int i, a=43, b=45, c=72, d=3, e=65;

pInt[0] = &a;
pInt[1] = &b;
pInt[2] = &c;
pInt[3] = &d;
pInt[4] = &e;


  clrscr(); highvideo();

  for(i=0;i<DIM;i++){
          cprintf("Contenido del Vector: %d\r\n\r\n", pInt[i]);
  }

   for(i=0;i<DIM;i++){
          cprintf("Contenido de lo que apunta el Vector: %d\r\n\r\n", *pInt[i]);
  }
 }
```

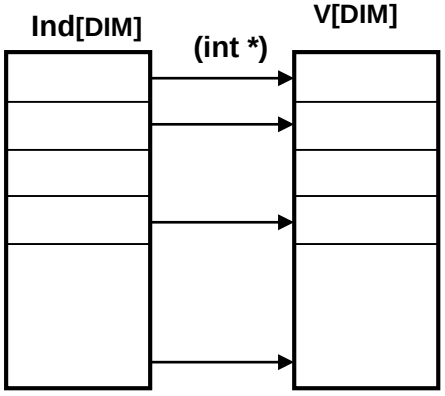
Ejemplo 2

Una utilización común de un arreglo de punteros es la de guardar los punteros a los mensajes de error.

```
/* --------- Ejemplo de clase Nro. 2---------------------------------------------------------- */

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

const DIM=4;

void main()
{
int i;
char *err[]={
                                    "no puedo abrir archivo",
                                    "error de lectura",
                                    "error de escritura",
                                    "fallo de soporte"
                  };

 clrscr(); highvideo();

 for(i=0;i<DIM;i++){
        cprintf("posibles errores: %s\r\n\r\n", err[i]);
 }
}
```

Ordenamiento a través de un índice.

Este es otro ejemplo clásico de la utilización de un arreglo de apuntadores. Consideremos la siguiente situación:



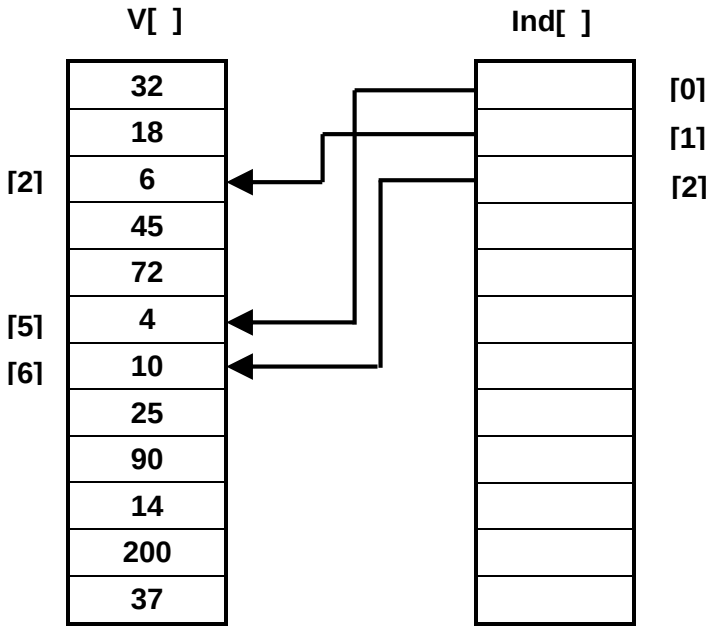
donde **V[DIM]** es un vector de enteros que contiene magnitudes entre 10 y 200, pero en forma totalmente desordenada.

El vector **Ind[ ]** es un arreglo de punteros a enteros que provisoriamente ha sido cargado con la dirección de los domicilios homónimos de **V[DIM]**.

El arreglo **V[DIM]** puede leerse de dos maneras:

a) **for(i=0;i<DIM;i++) cprintf("%4d",V[i]);**
b) **for(i=0;i<DIM;i++) cprintf("%4d",*Ind[i]);**

Ahora imaginemos que las direcciones señaladas por cada domicilio de **Ind[ ]** han sido alteradas de tal manera que recorriendo secuencialmente este arreglo, obtenemos un acceso **ORDENADO** en **V[DIM]**:



Los contenidos originales del vector **V[ ]** no han sufrido ninguna reubicación. Normalmente en las bases de datos se dispone de muchos campos que pueden incluso ser extensos, y reubicarlos para ordenarlos insume una cantidad enorme de tiempo. En cambio con un índice el problema se simplifica significativamente: en ese caso se elige un campo adecuado por el cual ordenar todos los registros.

He aquí el código para nuestro sencillo ejemplo:

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

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

const DIM = 12;

// ------------------------------------------------------------

void main()
{
  int  V[DIM]   = { 32,18,6,45,72,4,10,25,90,14,200,37 };
  int *Ind[DIM];
  int  i,j,k;
  int *Aux;

  clrscr(); highvideo();
```

```
  // --- Todos los apuntadores a domicilios homonimos ----

  for(i=0;i<DIM;i++)
    Ind[i]=&V[i];

  for(k=0;k<DIM-1;k++){
    for(i=0;i<(DIM-1-k);i++){
       if(*Ind[i+1]<*Ind[i]){
         Aux     =Ind[i];
         Ind[i]  =Ind[i+1];
         Ind[i+1]=Aux;
         }
    }
  }

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

  cprintf("\r\narreglo ordenado\r\n");

  for(i=0;i<DIM;i++)
    cprintf("%4d",*Ind[i]);

  getch();

}
```

Otros ejemplos aplicación.

Una cadena de caracteres estática contendrá el texto:

LA ACUARELA HABIA DESAPARECIDO

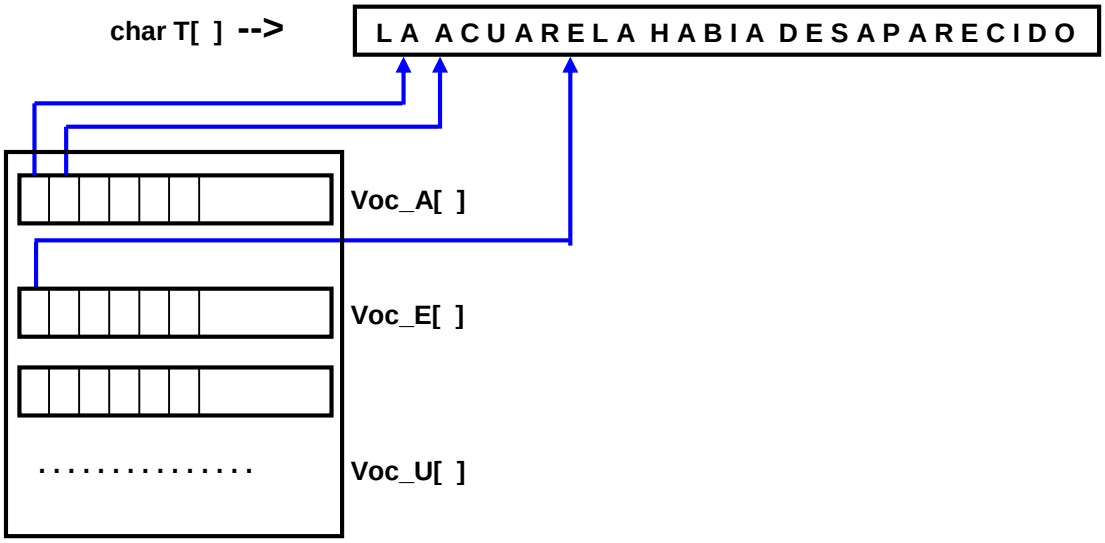
Una estructura denominada Voc que contiene 5 miembros declarados como arreglos de apuntadores a char (char *), señalaran la posición de cada 'A', cada 'E', etc. en la cadena anterior.

Finalmente mostrará por pantalla la cadena original y el orden en que se halla cada una de las vocales, primero las A luego las E, etc..

Por ejemplo:

LA ACUARELA HABIA DESAPARECIDO
2 4 7 ..... etc.

Siempre conviene partir de un esquema que nos oriente en el problema:



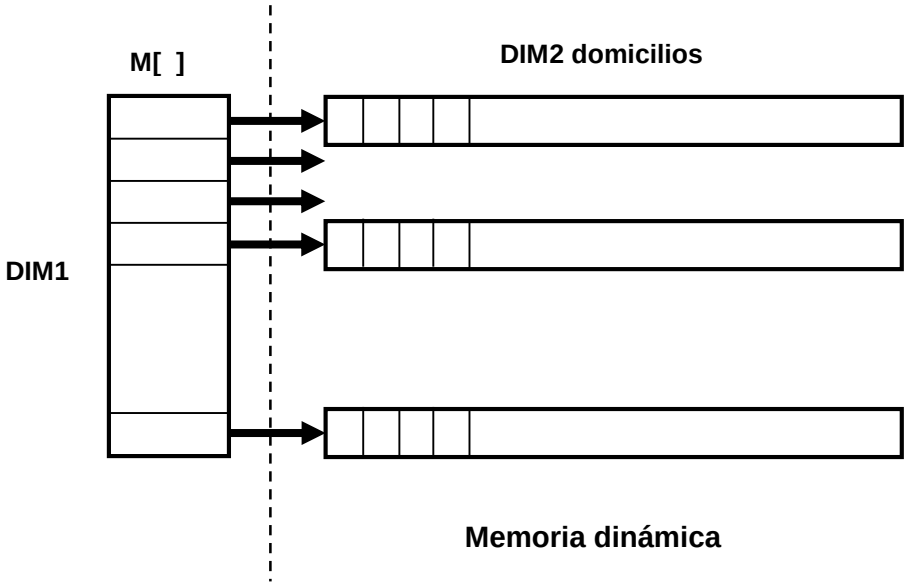

```
/* --------- Ejemplo de clase Nro. 4---------------------------------------------------------- */

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

typedef struct   {    char * Voc_A[10];
                           char * Voc_E[10];
                           char * Voc_I[10];
                           char * Voc_O[10];
                           char * Voc_U[10];
                      }TVoc;

// ------------------------------------------------------------
void main()
{
  char  T[64] = "LA ACUARELA HABIA DESAPARECIDO";
  TVoc  Voc   = {     { 0,0,0,0,0,0,0,0,0,0 },
                      { 0,0,0,0,0,0,0,0,0,0 },
                      { 0,0,0,0,0,0,0,0,0,0 },
                      { 0,0,0,0,0,0,0,0,0,0 },
                      { 0,0,0,0,0,0,0,0,0,0 }  };

  int i,i_A,i_E,i_I,i_O,i_U;

  clrscr(); highvideo();

  i_A=i_E=i_I=i_O=i_U=0;

  for(i=0;T[i];i++)

      switch(T[i]) {
        case 'A' : Voc.Voc_A[i_A++]=&T[i]; break;
        case 'E' : Voc.Voc_E[i_E++]=&T[i]; break;
```

```
        case 'I' : Voc.Voc_I[i_I++]=&T[i]; break;
        case 'O' : Voc.Voc_O[i_O++]=&T[i]; break;
        case 'U' : Voc.Voc_U[i_U++]=&T[i]; break;
       }

  cprintf(" %s\r\n\r\n",T);
  cprintf(" La cadena comienza en %u\r\n\r\n",&T[0]);

     cprintf("\r\nPosiciones de las A:");
     for(i=0;i<i_A;i++)
          cprintf(" %d",Voc.Voc_A[i]-&T[0]+1);



  getch();

}
```

La operación Voc.Voc_A[i]-&T[0]+1 encuentra el índice correspondiente donde se halla cada vocal. Es de hacer notar que esta diferencia tiene en cuenta el sizeof(tipo) al que apunta el apuntador.

Arreglos bidimensionales en memoria dinámica

Con un arreglo de punteros podríamos implementar un arreglo bidimensional en memoria dinámica:



El código para realizar las reservas y hacer las asignaciones vienen dado por:

```
/* --------- Ejemplo de clase Nro. 5 ------------------------------------------------------------ */

#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <process.h>

const DIM1 = 7;
const DIM2 = 8;

void *ReservarMemoria(int TotBytes);

void main( )
{
  int *M[DIM1];
  int  i,j;

  clrscr( ); highvideo( ); randomize( );
  for(i=0;i<DIM1;i++) {
    M[i]=(int *)ReservarMemoria(DIM2*sizeof(int));
    for(j=0;j<DIM2;j++){
    M[i][j]=50+random(50);
    cprintf("%3d",M[i][j]);
    }
    cprintf("\r\n");
```

```
  }
}

void *ReservarMemoria(int TotBytes)
{
void *pAux;

  if((pAux=malloc(TotBytes))==NULL) {
    cprintf("No pudo reservar memoria dinámica");
    getch( ); exit(1);
  }
  return(pAux);
}
```

Cabe destacar que en el momento en que el compilador ejecuta la instrucción:

int *M[DIM1]

Sólo se reservan los DIM1 domicilios que almacenarán idéntica cantidad de direcciones, pero ninguna de estas posee aún una dirección válida a la cual apuntar. Para subsanar esta situación hacemos:

M[i] = (int *)RerservarMemoria(DIM2*sizeof(int));

El lector no debe perder de vista que la notación tan sencilla para acceder a un domicilio cualquiera de la matriz dinámica:

M[i][j]

es una propiedad de los apuntadores que permite su notación subindexada. Si quisiéramos también podríamos haber escrito:

***(M[i]+j) = 50 + random(50);**

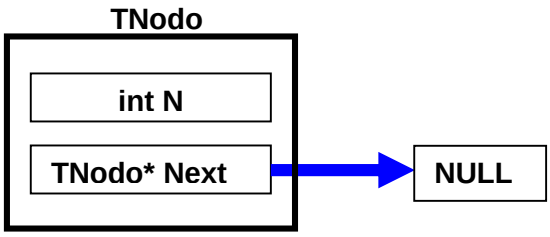
Listas enlazadas (Estructuras Auto referenciadas)

Una de las bondades más importantes que posee trabajar con apuntadores a estructuras, es que dicha estructura puede poseer como miembro un puntero a una estructura similar: ya sea a sí misma o a otra semejante, pero separada de la primera.

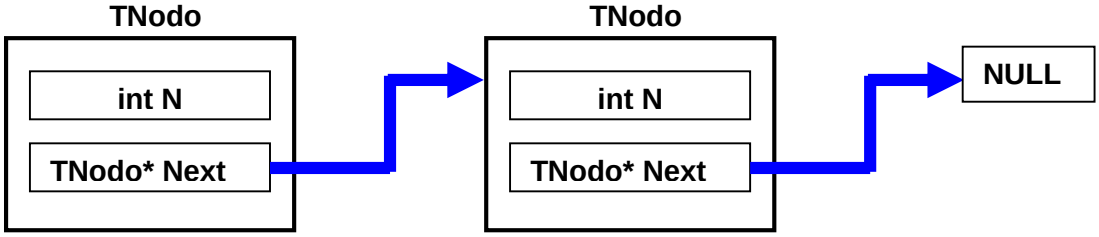
La sintaxis sería de la siguiente manera (en el caso más simple posible):

typedef struct TNodo { int N; TNodo* Next; }

que podríamos visualizar en el siguiente esquema:



Esto quiere decir que si generamos otra estructura similar utilizando nuestra conocida función **ReservarMemoria( )**, podríamos "enganchar" el apuntador **Next** de la primera con la segunda. Un conjunto de estos Nodos unidos o "ligados" a través del puntero **Next**, se denomina una "lista enlazada":



Para que la lista se halle completa, debemos poseer un apuntador extra (normalmente denominado **Start**) que señale siempre el comienzo de la lista, y en el último nodo el puntero **Next** debe quedar cargado con la dirección **NULL**, indicando que allí finaliza la lista.

Lo dicho, seguramente se verá más claro con el siguiente ejemplo de generación de lista:

Generar una lista simplemente encadenada que posea 15 nodos cada uno de los cuales será cargado con una magnitud entera aleatoria en el rango 50 ... 99.

/* --------- Ejemplo de clase Nro. 6-- */

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

```
#include <process.h>

typedef struct Nodo{
                      int N;
                      Nodo *Next;
                    }TNodo;

const   NNodos = 15;

void *ReservarMemoria(int TotBytes);

// ---------------------------------------------------------

void main()
{
  TNodo *Start;
  TNodo *pAux;
  int    i;

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

  // --- Generación de la lista enlazada ----------

  for(i=0;i<NNodos;i++)

      if(i==0) {
     Start=pAux=(TNodo*)ReservarMemoria(sizeof(TNodo));
     pAux->N=50+random(50);
     pAux->Next=NULL;
      }
      else {
       pAux->Next=(TNodo*)ReservarMemoria(sizeof(TNodo));
       pAux=pAux->Next;
       pAux->N=50+random(50);
       pAux->Next=NULL;
     }

  for(pAux=Start;pAux;pAux=pAux->Next)
    cprintf("%4d",pAux->N);

  getch();

}


void *ReservarMemoria(int TotBytes)
{
void *pAux;

  if((pAux=malloc(TotBytes))==NULL) {
    cprintf("No pudo reservar memoria dinámica");
    getch( ); exit(1);
  }
```

```
  return(pAux);
}
```

Nótese que si i=0 se produce un caso especial: debe inicializarse por única vez el nodo **Start** que señala el comienzo de la lista encadenada.

Una aclaración a hacer es que la generación de cada nodo normalmente está a cargo de una función específica que recibe la solicitación de reservar espacio para un nodo nuevo, pero dejamos su implementación para un apartado especial de esta clase que estudia específicamente a las funciones que retornan una dirección.

Listas doble enlazadas.

Las listas enlazadas están diseñadas para ser recorridas sólo en un sentido: hacia adelante, y los enlaces sólo unen el nodo anterior con el próximo y así sucesivamente. Sin embargo nada más simple que agregar un puntero al nodo para permitir generar una lista con conexiones no sólo hacia adelante sino también hacia atrás:

```
/* --------- Ejemplo de clase Nro. 7---------------------------------------------------------- */
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <process.h>

const   NNodos = 15;
typedef struct Nodo {
                  int N;
                  Nodo *Next;
                  Nodo *Back;
                 }TNodo;

void* ReservarMemoria(int TotBytes);

void main()
{
  TNodo* Start;
  TNodo* pAux;
  TNodo* End;
   int   i;

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


  //--- GENERA LA LISTA DOBLE ENLAZADA --------------------

  for(i=0;i<NNodos;i++)
     if(i==0) {
   Start=pAux=(TNodo*)ReservarMemoria(sizeof(TNodo));
   pAux->N=50+random(50);
   pAux->Back=NULL;
   pAux->Next=NULL;
```

```
   End=pAux;
     }
     else { pAux->Next=(TNodo*)ReservarMemoria(sizeof(TNodo));
      pAux->Next->Back=pAux;
      pAux=pAux->Next;
      pAux->N=50+random(50);
      pAux->Next=NULL;
         End=pAux;
    }

  //--- MUESTRA LISTA DE IZQUIERDA A DERECHA ----------------

  for(pAux=Start;pAux;pAux=pAux->Next)
     cprintf("%5d",pAux->N);

  cprintf("\r\n\r\n");

  //--- MUESTRA LISTA DE DERECHA A IZQUIERDA ----------------

  for(pAux=End;pAux;pAux=pAux->Back)
     cprintf("%5d",pAux->N);


  getch();

}


void *ReservarMemoria(int TotBytes)
{
void *pAux;

  if((pAux=malloc(TotBytes))==NULL) {
    cprintf("No pudo reservar memoria dinámica");
    getch( ); exit(1);
  }
  return(pAux);
}
```

Con un sólo puntero (pAux) ha sido suficiente para generar la lista doble
`

Punteros como parámetros de funciones

En C cuando queremos pasar alguna información a una función determinada podemos hacerlo mediante dos maneras:

- Parámetros por Valor
- Parámetros por Referencia

Al pasar una variable a una función mediante **valor** lo que se hace es pasar una copia de la información contenida en esa variable por lo que tendremos dos instancias

diferentes de la misma variable, una dentro de la función y otra fuera de ella, así si se modifica la información de la variable enviada esta solo será cambiada en ese ambiente.
Al pasar una variable a una función por **referencia** lo que hacemos en realidad es pasar un puntero a la dirección en memoria en la que se localiza la variable en si, por lo que al modificar la información mediante el apuntador en la función al que fue enviada esta será modificada en todos los ámbitos ya que en realidad modificamos la variable original, de hecho la única ya que mediante esta forma no se hace otra copia de la variable.

Normalmente "C", salvo indicación expresa, trabaja con parámetros de entrada o por valor, a excepción de los arreglos que, como ya sabemos, son tratados como apuntadores y al pasarlos a una función, automáticamente lo hace por referencia (su dirección). Un caso intermedio puede ser el siguiente:

```
void CargarVector(int *V)
{
  int i;
  for(i=0;i<DIM;i++)  V[ i ] = 50 + random(50);
}
```

En la línea de invocación iría: **CargarVector(V)**, donde V fue declarado en la forma:
int V[DIM]

Pero si en cambio redefiniéramos la función para que cargue sólo un domicilio por vez en lugar del arreglo completo, tendríamos:

```
void CargarDomicilio(int *p)
{ *p=50+random(50); }
```

Y desde el punto de invocación escribiríamos:

```
for(i=0;i<DIM;i++) CargarElemento(&V[ i ]);
```

Se sigue cumpliendo que la función requiere la dirección de un entero, pero esta vez desde el punto de invocación tendremos que fabricar el puntero con el operador & ya que por defecto todo lo que no sea un arreglo es pasado por valor.

FUNCIONES QUE DEVUELVEN PUNTEROS

Cuando definíamos un apuntador en la clase 1, especificamos que "era todo elemento de programa: variable, constante o función, cuyo contenido intrínseco era una dirección". Llegó el momento de estudiar a las funciones que son capaces de retornar una dirección.

Un caso típico es la generación de un nodo en una lista enlazada:

```
TNodo *NodoNuevo(int Total)
{
  TNodo *Direcc;
```

```
  if((Direcc=(TNodo *)malloc(Total))==NULL) {
      cprintf("NO PUDO RESERVAR PARA NODO");
      getch(); exit(1);
  }
  return(Direcc);
}
```

En realidad se trata de una función similar a nuestra conocida **ReservarMemoria( )**, con la excepción que en lugar de devolver un **void *,** retorna un **TNodo ***.

Otros casos característicos de funciones que retornan apuntadores, son las funciones que manipulan cadenas de caracteres, como por ejemplo la siguiente: insertar una subcadena dentro de una cadena.

```
char* strinsert(char *Fuente, char *Subc, int Desde)
{
  static char     Final[120];   // convendría un malloc( ) a medida.
         char * pFinal = Final;
         char * pChar = Fuente;
         int        i;

  if(Desde>strlen(Fuente)) return(NULL);

  for(i=0;i<Desde;i++) *pFinal++ = *pChar++;
  for(pChar=Subc;*pChar;) *pFinal++ = *pChar++;
  for(pChar=&Fuente[Desde];*pChar;) *pFinal++ = *pChar++;

  return(Final);
}
```

El esquema de trabajo es sencillo: por un lado tenemos la cadena original denominada **Fuente**. En esta cadena y a partir de un cierto caracter (en la posición **Desde**), insertaremos una subcadena de nombre **Subc**. Esto quiere decir que “debemos hacer lugar” para esta inserción. Por ejemplo:

Fuente = BORLANDC **E**L MAS VELOZ
Subc = ES SENCILLAMENTE
Desde = 9

Una posible solución, no la única, es la propuesta en el programa de más arriba: **Final** es una cadena auxiliar capaz de contener la suma de **Fuente** + **Subc** (esto es importante).

Final = BORLANDC **ES SENCILLAMENTE** EL MAS VELOZ

Primeramente se copia en Final los caracteres netre 0 y (Desde-1), se agrega la subcadena ES SENCILLAMENTE y por último se termina de copiar lo que quedaba de la cadena original. Esto queda evidenciado en:

```
for(i=0;i<Desde;i++) *pFinal++ = *pChar++;
for(pChar=Subc;*pChar;) *pFinal++ = *pChar++;
for(pChar=&Fuente[Desde];*pChar;) *pFinal++ = *pChar++;
```