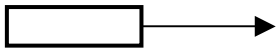
Indice de contenidos.

DEFINICION

Un puntero es un elemento de programa (constante o variable) ***cuyo contenido es una dirección***:



Esta dirección puede ser:

- *La ubicación en memoria de alguna otra variable (dirección de dicha variable).*
- *La dirección de entrada de una función (puntero a función).*
- *La dirección de comienzo de una reserva dinámica en el "heap".*
- *La dirección de comienzo de la memoria de video.*
- *etc.*

Notación formal de un puntero.

La sintaxis de un puntero es la siguiente :

```
int     *pENT;
char    *pCHAR;
float   *pFLOAT;
double  *pDOUBLE;
```

donde la presencia del asterisco (*) le confiere la característica de puntero. Sin embargo no debe pensarse que los punteros sólo pueden apuntar a datos simples como los mostrados. También pueden tenerse situaciones más complejas, como la siguiente:

```
typedef struct TArma {
                        char Marca[12];
                        float  Calibre;
                        char  Modelo[12];
                        char  Origen[3];
                      };
void main( )
{
  TArma *Arma;
```

e incluso veremos más adelante que los miembros de la estructura también pueden ser punteros a otros elementos, generando arquitecturas de datos tan complicadas como sea necesario.
Un detalle interesante es que todos los ejemplos mostrados tienen algo en común:

apuntan a un dado "tipo" de dato

y ello los engloba en la categoría de **PUNTEROS TIPIFICADOS**.

Obviamente, si hablamos de punteros tipificados pensamos en el acto que podría existir una categoría de punteros "no tipificados", y ello es correctísimo: existe dicha variedad, y recibe el nombre ***punteros void*** :

*void *Dirección;*

Estos punteros contienen una dirección "pura", no asociada a ningún tipo de dato.

Un ejemplo clásico de esto lo constituye la función "malloc( )" de la librería <alloc.h> cuya finalidad es la de realizar una reserva dinámica en el memoria masiva y regresar la dirección de comienzo de dicha reserva. El prototipo completo de esta función es :

*void *malloc(int CantidadDeBytes);*

y su sentido es bastante genérico : nosotros como usuarios de dicha reserva decidimos cómo utilizarla (almacenando enteros, float, char, ... o algún tipo especial declarado en la línea typedef).

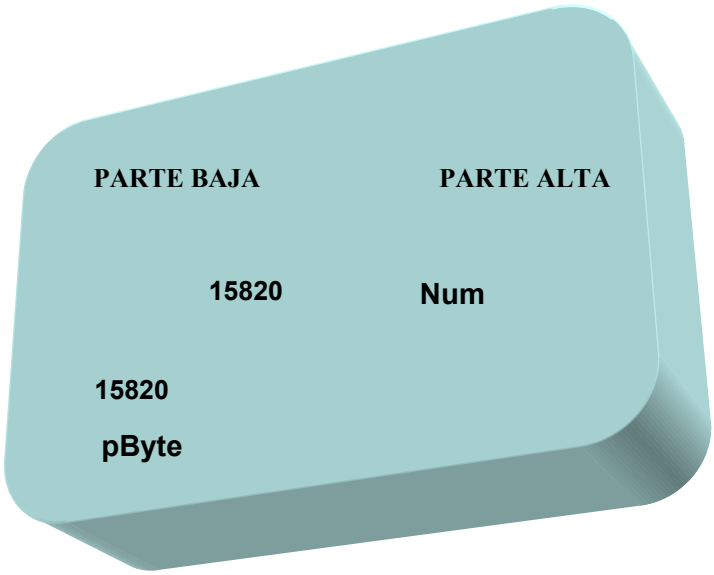
Primeros pasos con punteros.

Determinar la parte ALTA y la parte BAJA de un entero utilizando punteros.

```
#include <conio.h>

typedef unsigned char byte;

/* ------------------------------------------------------------------ */
void main()
{
  int      Num       = 14520;
  int      ParteBaja;
  int      ParteAlta;
  byte *pByte       = (byte *)&Num;

  clrscr( ); highvideo( );

  cprintf("\r\n");
  ParteBaja=*pByte++;
  ParteAlta=*pByte;
  cprintf("BYTE BAJO DE Num = %d\r\n",ParteBaja);
  cprintf("BYTE ALTO DE Num = %d\r\n",ParteAlta);

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

Visualizar el problema

Una excelente práctica en problemas que involucren punteros, es trazar un diagrama en lápiz y papel de manera de "ver" como se realizan las conexiones:



"Num" es una variable de tipo "int", que en la mayoría de los compiladores de "C" ocupa 2 bytes, denominándose **PARTE BAJA** al byte de más a la izquierda y **PARTE ALTA** al byte de más a la derecha, y aquí hacemos una pequeña digresión.

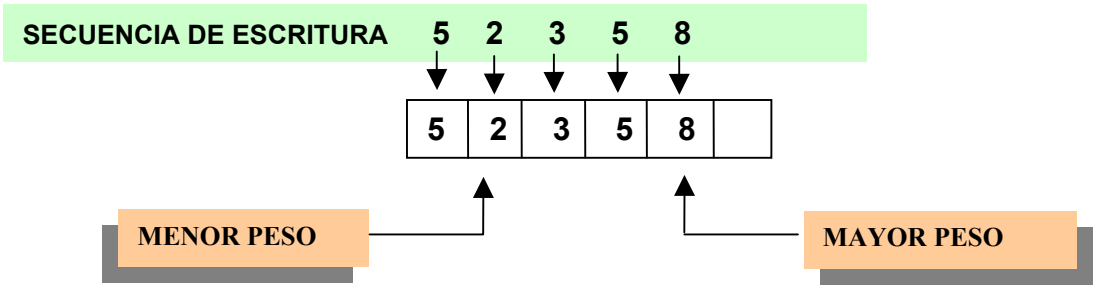
Seguramente el lector tendrá en mente que la parte baja es la de la derecha y la alta la de la izquierda, y ello es correcto. Los sistemas de numeración ***posicionales*** establecen que el dígito de ***menor peso*** es le de más a la derecha y a partir de allí cada dígito adicional ***hacia la izquierda*** posee un orden de magnitud mayor.

El ejemplo más sencillo es nuestro archiconocido sistema decimal, por ejemplo:

$$85325 = 8 .10^4 + 5 . 10^3 + 3 . 10^2 + 2 . 10^1 + 5 . 10^0$$

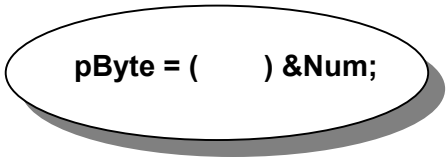
MAYOR PESO — MENOR PESO

Sin embargo la memoria del computador crece al revés: de izquierda a derecha. Por lo tanto al escribir los números, los dígitos de menor peso van quedando más a la izquierda y los de mayor peso a la derecha:



Esta notación se denomina **backword** (hacia atrás o invertida).
Volviendo a nuestro problema inicial, esto explicaría porqué para extraer el byte de menor peso o byte bajo, apuntamos el pByte en la posición inicial de la dirección de la variable **Num**.

El operador (&) tienen la particularidad de devolver la dirección de aquello a lo cual se halla aplicado:



sin embargo existe un pequeño inconveniente : este operador devuelve una dirección con el formato del tipo de dato al cual se está aplicando, en este caso devuelve la dirección de un entero con formato ***int ** y esto es incompatible con el formato esperado por la variable receptora : ***byte ***

Para solucionar esto se utiliza lo que se denomina un "cast", que es una especie de etiqueta que fuerza al compilador a cambiar un tipo de dato por otro. En este caso:

pByte = (byte *)&Ent

significa : cargue **pByte** con la dirección de la variable **Ent** con un formato de puntero a byte.

NOTA

NO debe perderse de vista que una variable puntero, al igual que cualquier otra variable (int, char, etc.), ocupa un lugar físico en el segmento de datos (o en al ámbito de trabajo que se trate) y desde este punto de vista no posee mayores privilegios. Si embargo la particularidad consiste en que su contenido es interpretado por el compilador como una dirección de memoria.

Dirección e Indirección.

Un puntero tiene dos aspectos:

- ✓ Su contenido propio (valor almacenado en la variable puntero), que es la dirección a la cual apunta.
- ✓ El dato que está refenciando en la dirección apuntada, (para lo cual utiliza la tipificación).

La primer instancia se denomina ***dirección*** y la segunda ***indirección***.

La indirección es lo que hace posible el acceso al dato referenciado. La expresión:

```
cprintf("BAYTE BAJO DE Num = %d \r\n", *pByte);
```

le indica al cprintf que el dato que saldrá con formato %d es *pByte (lo que está referenciando el puntero). Desde este punto de vista, *pByte se comporta como un dato simple de tipo byte (unsigned char), y le son válidas todas las propiedades y aplicación de operadores propias de este tipo. Podríamos tener por ej. expresiones como:

```
if(*pByte)==5) cprintf(".....");
Resto=(*pByte) % 5;
etc.
```

Una modificación mínima del programa de arranque nos permite cosas más interesantes:

Utilizando punteros determinar qué valores de un entero poseen:
PARTE ALTA = PARTE BAJA

```
#include <conio.h>

typedef unsigned char byte;

/* ------------------------------------------------------------------ */
void main( )
{
  int      Num;
  byte *pBaja      = (byte *)&Num;
  byte *pAlta      = (byte *)(&Num)+1;
                   // =  pBaja+1;

  clrscr( ); highvideo( );

  for(Num=1;Num<32767;Num++)
     if(*pBaja==*pAlta) { cprintf("%d\r\n",Num); getch(); }

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

Se hace notar la importante diferencia de hacer:

(byte *)(&Num+1) ó
(byte *)(&Num+1)

ya que el operador (&) retorna un puntero del mismo tipo que su operando.

El concepto intrínseco del problema implica que la distribución de "1" y "0" en cada byte **es el mismo**. Por ej. :

BYTE BAJO	BAYTE ALTO	
0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1	Num = 513
0 0 0 0 0 0 1 0	0 0 0 0 0 0 1 0	Num = 1026
0 0 0 0 0 0 1 1	0 0 0 0 0 0 1 1	Num = 1539
0 0 0 0 0 1 0 0	0 0 0 0 0 1 0 0	Num = 2052
etc.		

Un concepto más profundo de las direcciones.

Una dirección en sí (contenido de un puntero) ***es una magnitud unsigned*** si bien no es exactamente este el concepto. Pero tomándolo de esta manera podemos realizar algunas manipulaciones que afianzarán nuestro manejo de punteros. El siguiente programa plantea esta situación:

Tomar una cadena de caracteres y transformar su dirección de comienzo en una magnitud entera asignándola a una variable afín. Luego tomar un segundo puntero a char y cargarlo con la dirección almacenada en la variable entera y mostrar que lo apuntado por este segundo puntero es la misma cadena original.

```
#include <conio.h>

typedef unsigned int word;
/* ------------------------------------------------------------------ */

void main( )
{
 char  *Texto       = "TALLER DE LENGUAJES I";
 word   Direccion = (word)Texto;
 char  *pTexto      = (char *)Direccion;

 clrscr( ); highvideo( );
 cprintf("TEXTO : %s\r\n",pTexto);
 getch( );
}
/* ------------------------------------------------------------------ */
```

En este problema "Texto" es un puntero a char que señala el comienzo de una cadena de caracteres. De haber sido un arreglo clásico de char, el nombre del arreglo señalaría también la dirección de inicio del mismo.
Si Texto es una dirección a char, mediante un "cast" podemos convertirla en otro tipo de dato, no sólo como dirección con formato, sino algo que no sea un puntero. La expresión :

word   Direccion = (word)Texto;

toma la dirección de la cadena y la transforma en una magnitud unsigned int (word), asignándola a una variable compatible. Ahora ya no estamos manejando dirección como tal, sino una simple magnitud numérica.

¿Será posible volverla a transformar en la dirección original? Sí, si lo es:

char *pTexto = (char *)Direccion;

Esta operación toma la magnitud numérica y la transforma en una dirección que señala un dato de tipo char. Dicho en otras palabras: pTexto que originalmente carecía de una dirección válida, ahora es asignado con la dirección original de la cadena de caracteres.

Los punteros como conversores de "tipo".

Una característica que no podemos perder de vista en los punteros tipificados, es que ellos "se las ingenian" para armar el dato que referencian, tomando los bytes necesarios de acuerdo al "tipo apuntado". Dicho de manera más comprensible: si un puntero a **int** es cargado con una dirección dada, al operar sobre la indirección ***pENT**, el puntero toma 2 bytes y ***compone*** una magnitud de tipo int de forma completamente transparente para el programador. Si hubiese sido un puntero a long tomaría 4 bytes, etc.
Esto permite crear o extraer datos de un buffer que simule un archivo con cabecera, donde la información se halla inscripta a partir de determinadas posiciones.

Sin embargo bajaremos por el momento nuestras pretensiones y atacaremos un problema más sencillo:

Convertir una cadena de caracteres en un arreglo de 'long', mostrar dicho arreglo por pantalla y luego recorrer el arreglo byte a byte reconstruyendo la cadena original.

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

/* ------------------------------------------------------------------------------------ */
void main( )
{
  char   Texto[ ]     = "BORLANDC ES VELOZ";
  long   Vector[10]  = { 0,0,0,0,0,0,0,0,0,0 };
  long  *pLONG       = (long *)Texto;
  char  *pChar       = (char *)Vector;
  int    i,j;

  clrscr( ); highvideo( );

  for(i=j=0;i<strlen(Texto);i+=sizeof(long)) Vector[j++]=*pLONG++;
  for(i=0;i<10;i++) cprintf("%ld ",Vector[i]);
  for(cprintf("\r\n");*pChar;pChar++) cprintf("%c",*pChar);

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

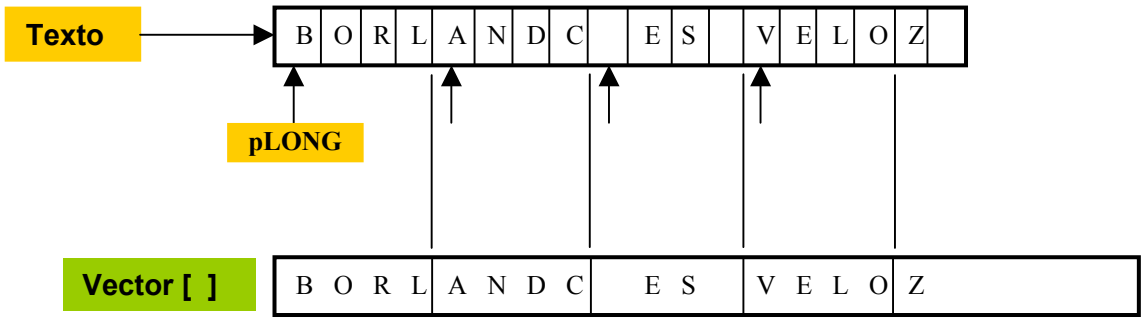
No perder de vista que el vector de long de 10 domicilios posee en realidad 40 bytes de memoria, suficiente para almacenar la cadena. La operación:

long *pLONG = (long *)Texto;

carga al puntero de tipo long con la dirección de comienzo de la cadena, pero verá a la misma no como cadena sino como si se tratase de elementos de tipo long. De esta suerte, la instrucción :

for(i=j=0;i<strlen(Texto);i+=sizeof(long)) Vector[j++]=*pLONG++;

realiza en realidad la siguiente tarea :



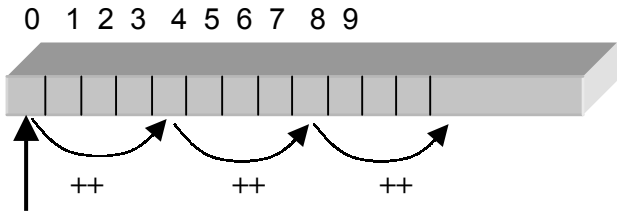
Aritmética de punteros.

Habrá notado la expresión: ***Vector[j++]=*pLONG++;*** en la cual se ha hecho uso del operador incremento y esto requiere de una muy importante aclaración.
Seguramente recordará que en variables ordinarias como por ejemplo:

long **x** = 10;

al hacer **x**++, **x** se incrementaba en una unidad.
Aquí la cosa cambia, porque al hacer **pLONG++** la magnitud almacenada en él (la dirección apuntada) cambia en :

Dirección = Dirección + sizeof(TipoApuntado)

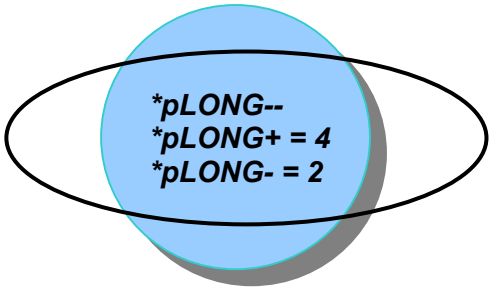


En la operación ***pLONG++** existe un orden de precedencia bien definido :

- *Primero, toma la indirección **pLONG***
- *Segundo, incrementa la dirección apuntada.*

De nuevo repetimos que, al tomar la ***indirección***, el puntero "sabe" que al ser de tipo long debe tomar 4 bytes consecutivos y componer la magnitud de ese tipo.

Un puntero no sólo puede ***incrementarse***, sino también ***decrementarse*** y moverse varios lugares en una sola operación:



Para el caso particular de ***pLONG+ = 4** la operación interna realizada es:

Contenido de pLONG = Contenido de pLONG + 4 x sizeof(long)

Notaciones particulares :

- ***pLONG++** Utiliza lo referenciado y luego incrementa la dirección.
- ***(*pLONG)++** Incrementa lo referenciado.
- ***(++pLONG)** Incrementa la dirección y luego utiliza lo referenciado.
- ***(pLONG+2)** Utiliza lo referenciado y luego incrementa dos lugares el puntero

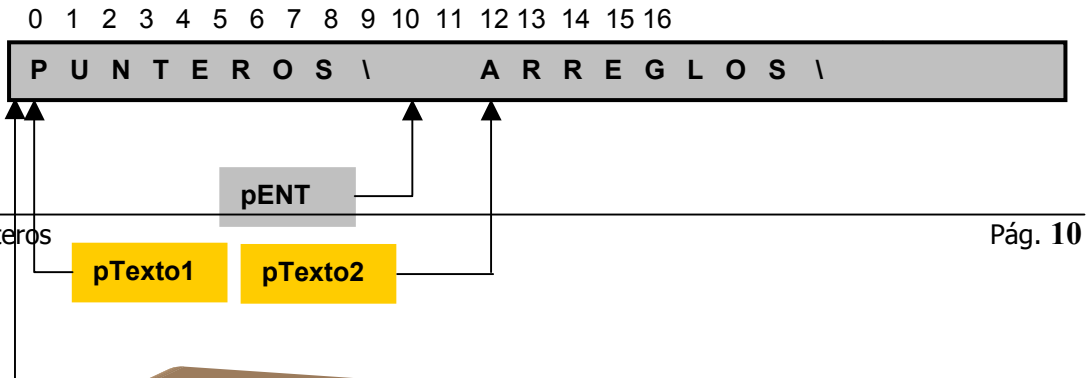
- ***(*pLONG++)++*** Utiliza lo referenciado, incrementa la dirección y luego incrementa lo referenciado en la nueva posición.
- ***++(*pLONG)*** Incrementa lo referenciado y luego utiliza dicho valor.
- ***++(*pLONG)++*** Incrementa lo referenciado, lo utiliza y vuelve a incrementar lo referenciado.

Punteros a estructuras.

Este problema se halla destinado a demostrar que la reserva de memoria que el compilador realiza para una estructura se trata de un bloque secuencial de bytes (todos en posiciones consecutivas, similar a lo que ocurre con los arreglos).

*Sobre un **arreglo de char** generar dos cadenas y un entero mediante la utilización de sendos punteros. Luego definir un **puntero a estructura**, cuya plantilla deberá tener el **mismo tamaño** del arreglo y asignarle la **dirección del arreglo**. Los datos ingresados en el array deben accederse mediante los miembros de la estructura.*

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

typedef struct TDatos {
                         char  Cad1[10];
                         int     Entero;
                         char   Cad2[14];
                        };
/* ------------------------------------------------------------------ */
void main( )
{
  char       Vect[26];
  TDatos  *Datos     = (TDatos *)Vect;
  char       *Texto1    = (char *) Vect;
  int          *ENT       = (int  *)&Vect[10];
  char       *Texto2    = (char *)&Vect[12];

  clrscr( ); highvideo( );

  strcpy(Texto1,"PUNTEROS.");
  strcpy(Texto2,"ARREGLOS.");
  *ENT=14520;

  cprintf("TEXTO1 = %s\r\n",Datos->Cad1);
  cprintf("TEXTO2 = %s\r\n",Datos->Cad2);
  cprintf("ENTERO = %d\r\n",Datos->Entero);

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



El complemento de los ejemplos anteriores donde cargábamos un puntero con una dirección dada y extraíamos datos (el puntero sabía cuántos bytes tomar) es la operación : ***ENT=14520**; en la cual en lugar de extraer, colocamos información. Aquí sucede lo mismo : el puntero "sabe" cuántos bytes asignar con el dato numérico.

Lo interesante de este problema es que al cargar el puntero a estructura con la dirección de comienzo del arreglo de char

equivale a haber realizado una asignación

a cada uno de sus miembros, al menos eso es lo que intentamos probar. Para determinar si nuestra teoría es correcta, las instrucciones:

```
cprintf("TEXTO1 = %s\r\n",Datos->Cad1);
cprintf("TEXTO2 = %s\r\n",Datos->Cad2);
cprintf("ENTERO = %d\r\n",Datos->Entero);
```

deberán reproducir los datos introducidos al arreglo mediante los punteros individuales. Si corre este programa verá que funciona a la perfección.

A modo de recordatorio:

La operación ***&Vect[10]*** retorna la dirección del domicilio 10 del vector pero en un formato de ***char ** debido a que el tipo de datos que almacena el arreglo son char. Debido a ello, la inicialización del puntero **ENT** requiere de un "cast" que haga la conversión de formato en el puntero: ***(int *)&Vect[10]***

He aquí otro ejemplo interesante:

Tomar una cadena de caracteres y transformar (dentro de un vector de unsigned) la dirección de cada caracter en un entero. Luego mostrar el contenido de la cadena pero trabajando con las direcciones del vector.

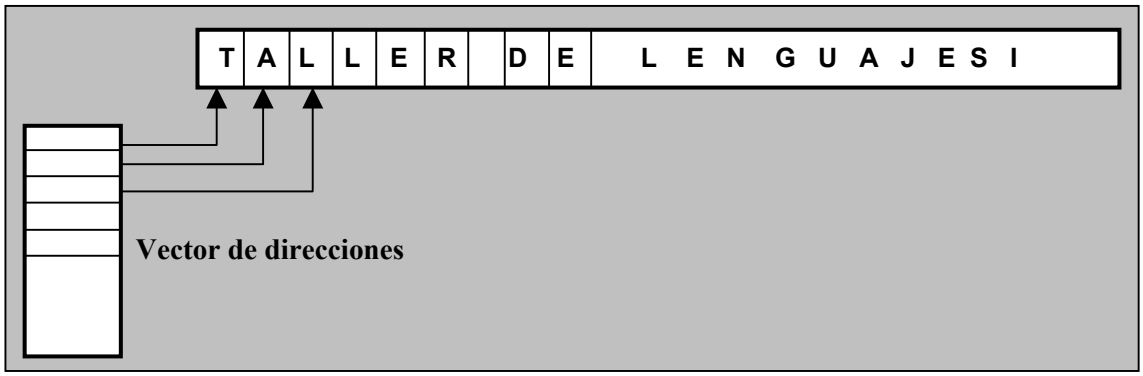
Finalmente y utilizando un puntero a char, modificar la 'R' de TALLER por 'r' y volver a mostrar el contenido de la cadena, pero a través de la variable "Texto".

```
#include <conio.h>

typedef unsigned int word;

/* ------------------------------------------------------------------------------------------------------ */
void main( )
{
  char   *Texto = "TALLER DE LENGUAJES I";
  word    Direcc[21];
  char   *pCar;

  clrscr( ); highvideo( );

  // --- TRANSFORMA DIRECCIONES DE CARACTERES EN ENTEROS ---------
  for(int i=0;*(Texto+i);Direcc[i]=(word)(char *)(Texto+i),i++);

  // --- MOSTRAR LO APUNTADO POR EL VECTOR DE ENTEROS ----------------
  for(i=0;i<21;i++) cprintf("%c",*((char *)Direcc[i]));

  // --- CAMBIA LA 'R' POR 'r' ---------------------------------------
   cprintf("\r\n");
  *(pCar=(char *)Direcc[5])='r';
   cprintf("%s\r\n",Texto);

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

El vector ***Direcc[21]*** (definido de tipo unsigned int) contendrá direcciones pero no en el formato de las mismas, sino como magnitudes numéricas. De ahí que la expresión :

Direcc[i]=(word)(char *)(Texto+i),i++);

toma el formato ***char ** de la dirección de cada elemento de la cadena y lo transforma en un dato de tipo ***word*** mediante un cast. El esquema sería el siguiente :



Para poder mostrar en pantalla las direcciones referenciadas por este arreglo de enteros, tendremos que realizar el proceso inverso : transformar las magnitudes numéricas en direcciones de tipo char :

cprintf("%c",*((char *)Direcc[i]));

Reservas Dinámicas de Memoria.

Declaración : ***void *malloc(size_t size);*** (size_t es un unsigned int)

malloc( ) emplaza un bloque de "size" bytes en el ***heap***. Esto permite a un programa emplazar memoria explícitamente en el momento y en la cantidad exacta que necesite.

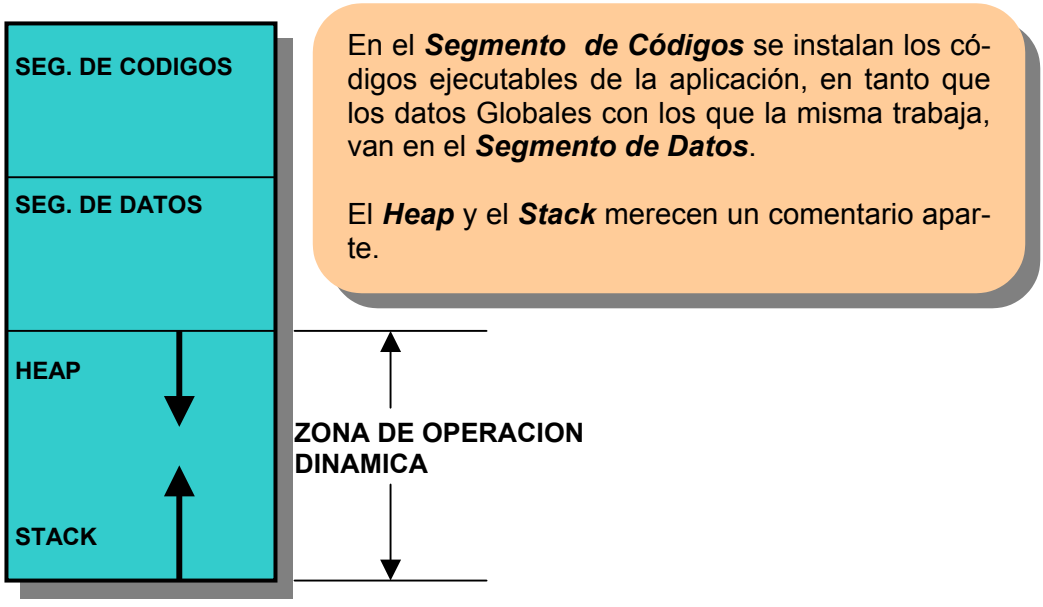
El heap es utilizado para reservas dinámicas de bloques de memoria de ***tamaños variables***. Muchas estructuras de datos, tales como árboles y listas emplean de manera natural reservas de memoria.

La totalidad del espacio entre el final del segmento de datos y la cima del *stack* del programa se halla disponible para utilizarlo en los modelos pequeños de datos, excepto para un margen pequeño inmediatamente antes de la cima del ***stack***. Este margen se halla para permitir a la aplicación (ejecutable) disponer de algún espacio adicional para el ***stack.*** En suma : para ser utilizado por el **DOS**.

En los modelos grandes de memoria, todo el espacio más allá del ***stack*** del programa hasta el final de la memoria disponible puede ser utilizado para ***heap***.

Los términos *heap* y *stack* :

Normalmente el espacio de trabajo de una aplicación ejecutable es el siguiente :



El ***stack*** (pila) es una zona de memoria donde se crea el espacio de trabajo necesario para las funciones. A medida que las mismas son invocadas se genera lo que se denomina ***registros de activación*** en el cual se definen las ***variables locales*** de la fun-

ción, la ***zona de parámetros***, los ***transitorios de operaciones*** matemáticas, la dirección de retorno al punto de invocación, etc.
Cuando una función finaliza con sus operaciones, la zona de memoria que ella ocupaba es liberada y se considera memoria libre para futuras invocaciones, o sea que esta porción de memoria crece y decrece durante la ejecución de la aplicación. No debe confundirse nunca el código que gobierna la función, con el espacio de trabajo de la misma.

El ***heap*** en cambio, está destinado para otro tipo de requerimiento : las reservas de memoria ***durante la ejecución*** (reservas dinámicas). Estas reservas tienen la particularidad de que el compilador ***desconoce*** su tamaño ***antes*** de la ejecución, por eso el nombre de dinámicas, y al igual que la zona del stack, los bloques reservados pueden liberarse una vez utilizados y luego volver a reasignarse.

De esta manera el ***heap*** y el ***stack*** crecen y decrecen enfrentados en el mapa de memoria del computador, surgiendo de inmediato en nuestra mente la idea de que tales zonas podrían invadirse mutuamente y que seguramente habría algún problema. Así es : se produce un desastre llamado ***colisión del heap con el stack*** produciendo normalmente que el sistema se ***cuelgue***.

Volviendo a nuestro malloc( ) aún nos falta los :

Valores regresados.

- *En caso de éxito, regresa un puntero con la dirección del bloque emplazado en memoria.*
- *En caso de error (si no existe suficiente espacio), **malloc( )** retorna una dirección nula (que normalmente se utiliza para verificación de reserva).*
- *Si el argumento **size** es cero, **malloc( )** regresa **null**.*

Los modelos de memoria.

Un detalle que jamás debemos perder de vista cuando hablamos de reservas dinámicas y manejo del stack, son los modelos de memoria. Un modelo de memoria es la forma en que "C" administra la memoria para la ejecución de las aplicaciones.
Esto permite la generación de códigos que permiten utilizar con la mayor eficiencia el hardware disponible de acuerdo a los requerimientos del problema.
La práctica nos enseña que existen :

- *Programas pequeños que realizan tareas específicas donde se requieren una gran velocidad de ejecución.*
- *Programas con una enorme cantidad de código que procesan complejamente pequeñas cantidades de datos.*
- *Programas pequeños que operan sobre una masa muy grande de información.*
- *Programas con mucho código que trabajan sobre una gran cantidad de datos.*

Obviamente cada uno de ellos requerirá una exigencia distinta de memoria y del procesador. El siguiente cuadro presenta los modelos de memoria disponibles en "C" :

MODELO DE MEMORIA	SEGMENTOS			PUNTEROS	
	CODIGOS	DATOS	STACK	CODIGOS	DATOS
Tinny	64 Kb			near	near
Small	64 Kb	64 Kb		near	near

Habrá notado que en la sección que dice ***Punteros***, aparecen rótulos que dicen ***near*** y ***far***. Por el momento, y asumiendo que estamos trabajando con los modelos ***Tinny*** o ***Small***, nos despreocuparemos de este detalle. Más adelante cuando analicemos el concepto de punteros largos, entonces sí nos tomaremos el tiempo necesario para aclarar estas ideas.

Ejemplos de utilización de malloc( ).

A través del siguiente ejemplo tomaremos conocimiento de importantes detalles que pueden dar serios dolores de cabeza.

Definir en el main( ) un puntero a entero y a través de él realizar una reserva dinámica cuya magnitud trataremos de variar en cada corrida para ver qué sucede.

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

typedef unsigned int INT;
const INT DIM = 30000;
/* ------------------------------------------------------------------------------------- */
void main( )
{
  int *pENT;
  int  i;
  INT  Size = DIM * sizeof(int);

  clrscr( ); highvideo( );

  // cprintf("Size=%u\r\n",Size);



  if((pENT=(int *)malloc(Size))==NULL) {
        cprintf("NO PUDO RESERVAR MEMORIA DINAMICA. \r\n");
  else cprintf("RESERVA  Ok DIRECCION %p \r\n",pENT);
```

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

Vamos a correr este programa para armar la siguiente tabla :

Size	Modelo de Memoria	Mensaje
60000	Tinny	NO PUDO RESERVAR MEMORIA DINAM.
60000	Small	RESERVA Ok DIRECCION : 05DA
40000	Tinny	RESERVA Ok DIRECCION : 2152
80000	Tinny	RESERVA Ok DIRECCION : 2152

Ahora comencemos a interpretar para formarnos un criterio de si los resultados obtenidos son correctos o no, y porqué.

En primer lugar recordemos la tabla de los modelos de memoria de "C": el modelo ***Tinny*** era el más pequeño de todos y disponía de 64 Kb para los tres segmentos (códigos, datos y memoria dinámica). Ello significa que si le estamos solicitando 60.000 bytes, y aparte de esto debe alojar los códigos compilados, con toda seguridad se quedará corto y no habrá memoria suficiente.

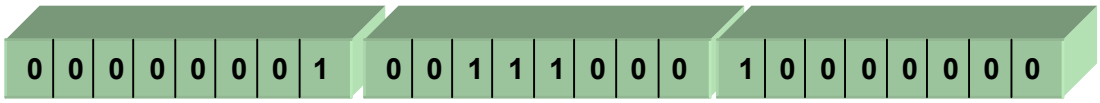
Si a la misma reserva la solicitamos pasando a un modelo superior (***Small***, segunda línea de la tabla), éste dispone de 64 Kb para códigos y 64 Kb para datos y memoria dinámica. Y aquí sí funciona todo bien.

En la tercera línea retornamos al modelo ***Tinny*** pero bajamos nuestras pretensiones de memoria a 40000 bytes y como el código del programa es corto, los 64 Kb de este modelo cubren sin problemas el pedido.

Y ahora la parte extraña: la cuarta línea solicita ***80.000 bytes*** nada menos que al modelo ***Tinny***, y para sorpresa nuestra resulta que sí hace la reserva ¿Qué pasó? Simplemente lo siguiente:

Si activamos la línea que figura comentada: // ***cprintf("Size=%u\r\n",Size);*** veremos que el resultado que sale a pantalla es 14.464 y tratemos de explicar porqué :

Para representar la magnitud 80.000, necesitamos al menos 3 bytes:



pero como la función malloc( ) se halla definida en su librería como :

void *malloc(size_t size);

o sea con su argumento como un ***unsigned int***, lo cual significa que tomará únicamente 2 bytes de la magnitud que tratemos de pasarle. Si en la representación anterior nos quedamos con los 2 bytes de menor peso (correspondientes a un unsigned int), veremos que conforman exactamente el número 14.464:

8.192 + 4.096 + 2.048 + 128 = 14.464

Y si de todas maneras se nos ocurre "forzar" un poco las cosas haciendo:

pENT = (int *)malloc((long)Size)

como la función se halla definida internamente para unsigned y no para long, vuelve a realizar el recorte y nos da una dirección válida para una reserva de 14.464 bytes.

NOTA : Los modelos de mayor porte que sin duda solucionarían en el acto nuestro problema lo veremos al estudiar los punteros far.

El tema de los errores por ***overflow***, como el rebasamiento que produce la magnitud ***DIM*sizeof(int)*** en una representación de 2 bytes, no debe perderse nunca de vista, puesto que genera errores ***indetectables*** : la sintaxis es correcta y la lógica perfecta, pero sin embargo el error está... ***y tan escondido!*** que habrá que hilar muy fino para descubrirlo.

Punteros y Arreglos.

Como recordará el lector, un arreglo convencional, por ejemplo del tipo:

int Vector[DIM]

es tratado por "C" como una especie de puntero que señala ***el comienzo*** del bloque reservado. Esto hace posible que la siguiente sintaxis sea válida:

for(i=0;i<DIM;i++) ***Vector[i]***=random(100); ó bien
for(i=0;i<DIM;i++) ***(Vector+i)***=random(100);

pero obsérvese que el truco utilizado es ***(Vector+i)*** y no ***Vector++** puesto que esto último implicaría perder la dirección de inicio del vector, y el compilador naturalmente no lo permite!

Otra implicancia que tiene el considerar a los arreglos como punteros, es que al trabajar con funciones, si un arreglo es pasado como parámetro, el mismo pasa por referencia (dirección original del arreglo).

Existen lenguajes como Pascal que permite la asignación directa entre arreglos, es decir operaciones como Arreglo1 = Arreglo2 (cargue el primer arreglo con el contenido del segundo). En "C", debido a que toma los nombres del los arreglos como direcciones, esto no es posible, perdería al dirección original de uno de ellos.

Punteros y Funciones.

La forma natural de "C" de pasar un parámetro es por valor, o sea realizar una copia local del parámetro dentro de la función (excepción ya vista con los arreglos). Sin embargo en este último caso a pesar de que en la invocación a la función la dirección del arreglo pasa desapercibida, no ocurre lo mismo en el prototipo de la función ni en la cabecera de su definición:

Definir en el main( ) un arreglo de enteros de 5 elementos e inicializarlos, luego asignarle valores mediante una función que recibirá dicho arreglo como parámetro. Esto demostrará que los arreglos se pasan por referencia (su dirección).

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

const int DIM = 5;
void CargarVector ( int *);
/* ------------------------------------------------------------- */
void main( )
{
  int Vect[DIM] = { 0,0,0,0,0 };
  int i;

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

  CargarVector(Vect);
  for(i=0;i<DIM;i++) cprintf("%d  ",Vect[i]);
  getch( );
}
/* ------------------------------------------------------------- */
void CargarVector(int *Vect)
{ for(int i=0;i<DIM;i++) Vect[i]=random(10); }
/* ------------------------------------------------------------- */
```

Nótese la notación ***void CargarVector (int *)***; en la cual le indicamos a la función que le pasaremos la dirección de un entero (puede ser un datos individual o el comienzo de una reserva de datos int), en la formalización del código de la función :

void CargarVector(int *Vect)

Sin embargo no es la única, el compilador también acepta y traduce perfectamente la siguiente sintaxis : ***void CargarVector (int[ ])*** al igual que en la definición de la función : void ***CargarVector(int Vect[ ])***
En el programa anterior el hecho de que la instrucción

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

muestre perfectamente los valores asignados en la función demuestra que los arreglos son pasados por referencia (deben aparecer valores entre 0 y 10).

Arreglos y punteros.

Para el procesamiento de cadenas de caracteres es una práctica muy normal el uso de punteros a char. Veremos una tanda de ejemplos donde esto se nota claramente.

Desarrollar una función de usuario que determine el largo de una cadena.

```
#include <conio.h>

int longtxt  ( char * );
/* ------------------------------------------------------------------ */
void main( )
{
  char  Texto[80] = "BORLANDC";

  clrscr( ); highvideo( );

  cprintf("Longitud = %d\r\n",longtxt(Texto));
  getch( );
}
/* ------------------------------------------------------------------ */
int longtxt(char *Texto)    // --- FORMA I
{
  char *pTexto = Texto;

  while(*pTexto) pTexto++;
  return(pTexto-Texto);
}
/* ------------------------------------------------------------------ */
int longtxt(char *Texto)   // --- FORMA II
{ for(int i=0;*(Texto+i);i++); return(i); }
/* ------------------------------------------------------------------ */
```

En la resolución FORMA I, hemos utilizado un puntero auxiliar ***pTexto*** que toma la dirección original de la cadena y a este sí podemos incrementarlo sin problemas porque seguimos teniendo la dirección original en ***Texto***.

Lo realmente destacable de este ejemplo es la diferencia de punteros :

return(pTexto-Texto);

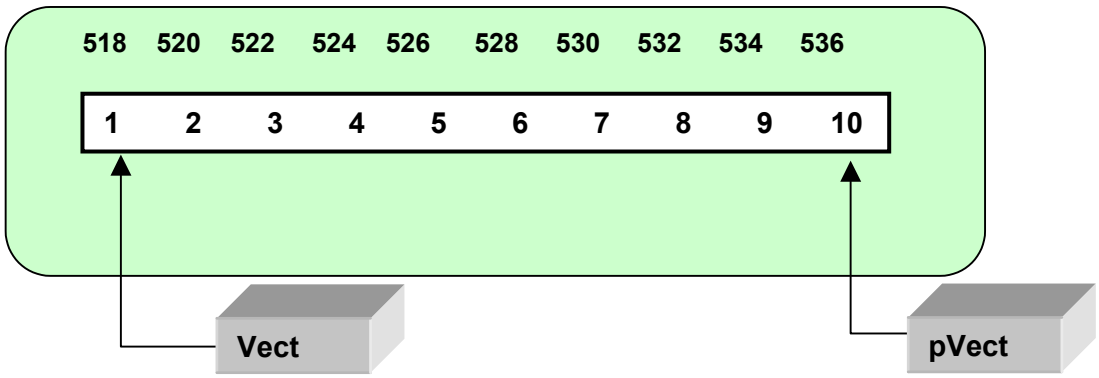
que posee un sentido totalmente distinto al de las direcciones en sí : nos da la ***cantidad de bloques*** de tamaño ***sizeof(tipo_apuntado)*** entre las dos direcciones. En nuestro caso como los sizeof( ) son 1 porque se trata de char, la diferencia es directamente ***la longitud de la cadena***.

Tal vez un ejemplo numérico mejore la idea de este concepto:

```
#include <conio.h>

/* ------------------------------------------------------------------------ */
```

```
void main( )
{
  int Vect[10] = { 1,2,3,4,5,6,7,8,9,10 };
  int *pVect   = (int *)&Vect[9];

  clrscr( ); highvideo( );
  cprintf("%d\r\n",pVect-(int *)Vect);
  getch( );
}
/* ------------------------------------------------------------------------ */
```

Aquí la expresión : ***pVect-(int *)Vect)*** arroja un resultado 9, aunque si nos fijamos bien en realidad existen 18 bytes entre ambas posiciones. O se que el resultado de la diferencia de punteros en realidad debe interpretarse como :



Los números por encima del arreglo suponen posibles direcciones de memoria. De esta manera: ***pVect - (int *)Vect = (536 - 518)/sizeof(int) = 9***

NOTA: Para obtener la cantidad correcta de domicilios hacer Vect –(int *)pVect +1

Funciones que devuelven un puntero.

Una función que lamentablemente "C" o dispone es la de insertar una subcadena dentro de otra, si bien su implementación no es demasiado complicada:

Implementar una función que permita insertar una subcadena dentro de una cadena de caracteres.

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

char *strinsert   ( char *Fuente, char *Subcadena, int Desde);



/* ------------------------------------------------------------------------------------------------ */
void main( )
{
```

```
  char *Texto = "BORLANDC EL MAS VELOZ.";

  clrscr(); highvideo();
  Texto=strinsert(Texto,"++ ES SENCILLAMENTE",9);
  cprintf("%s\r\n",Texto);
  getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
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);
}
/* ------------------------------------------------------------------------------------------------------------ */
```

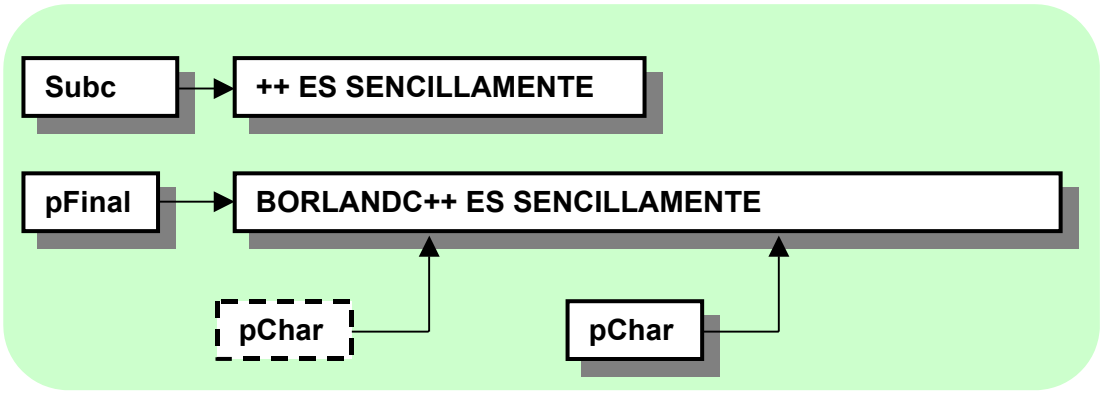
En primer lugar el porqué de la declarativa ***static*** : ello se debe a que la función devolverá como resultado la dirección de algo que ha sido declarado como ***identificador local***, por lo tanto una vez desocupada la función éste se perdería al considerarse libre esta zona de memoria. Sin embargo lo que se declara como ***static*** permanece como una zona protegida de memoria durante toda la ejecución del programa.

La expresión: ***for(i=0;i<Desde;i++) *pFinal++=*pChar++;***

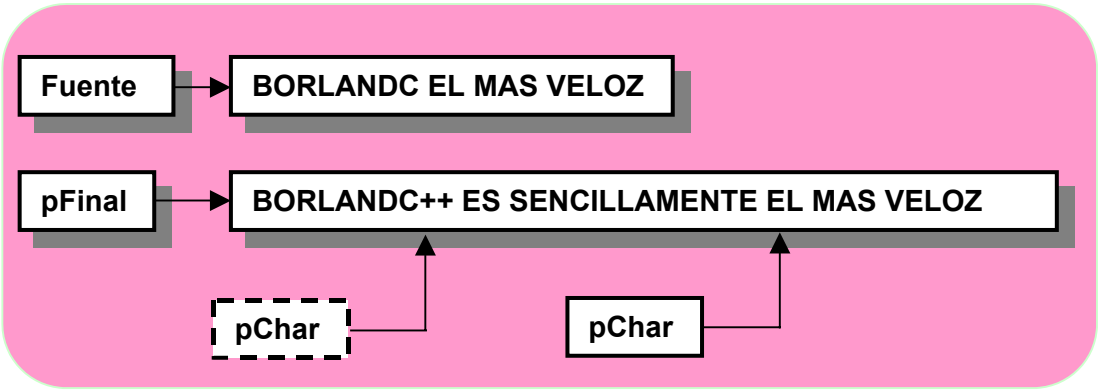
trasvasa la parte de la subcadena original desde su comienzo hasta el punto exacto en que debe insertarse la otra. De ahí que ***i*** se incrementa hasta ***i<Desde***.

La expresión siguiente: ***for(pChar=Subc;*pChar;) *pFinal++=*pChar++;***

Pasa ***todo*** el contenido de la subcadena a insertar a partir del punto exacto en que había quedado la anterior.



La última instrucción: ***for(pChar=&Fuente[Desde];*pChar;) *pFinal++=*pChar++;*** termina de pasar el resto de la cadena fuente luego de la inserción de la subcadena :



Otra función que retorna un puntero es la que permite extraer una subcadena de otra:

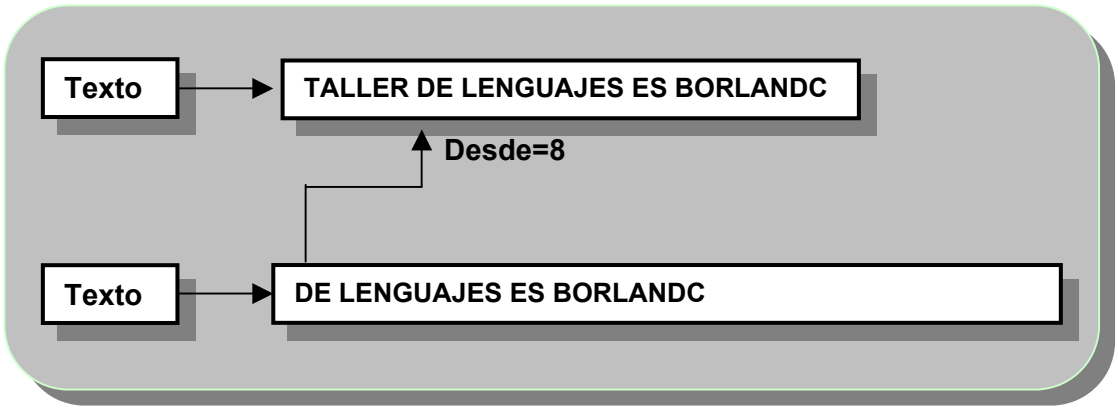
*Desarrollar una función denominada **strcopy( )** que permita "extraer" una subcadena de una cadena.*

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

char *strcopy  ( char *Cadena, int Desde, int Total);
/* ------------------------------------------------------------------------------------------ */
void main( )
{
  char *Texto = "TALLER DE LENGUAJES ES BORLANDC";

  clrscr( ); highvideo( );



  cprintf("EXTRACCION1 : %s\r\n",strcopy(Texto,1,3));
  cprintf("EXTRACCION2 : %s\r\n",strcopy(Texto,strlen(Texto),1));
```

```
  cprintf("EXTRACCION3 : %s\r\n",strcopy(Texto,8,40));
  cprintf("ORIGINAL    : %s\r\n",Texto);

  getch( );
}
/* ------------------------------------------------------------------------------------------------ */
char *strcopy(char *Texto, int Desde, int Total)
{
  static char Subc[255];
         int  i;

  // --- PROTECCIONES ----------------------------------------
  if(Desde>strlen(Texto)) return(NULL);
  if((Desde+Total)>strlen(Texto)) Total=strlen(Texto)-Desde+1;

  for(i=0;i<Total;i++) Subc[i]=*(Texto+Desde+i-1);
  Subc[i]='\0';

  return(Subc);
}
/* ------------------------------------------------------------------------------------------------ */
```

Este problema es mucho más sencillo que el anterior. La instrucción:

for(i=0;i<Total;i++) Subc[i]=*(Texto+Desde+i-1);



extrae a partir del índice ***Desde*** (8) una cantidad ***Total*** (40) de caracteres. Si bien la magnitud 40 excede lo que resta de caracteres en la cadena original, ello no es problema, pues está previsto en la zona de protecciones:

if((Desde+Total)>strlen(Texto)) Total=strlen(Texto)-Desde+1;

que ajusta ***Total*** al valor correcto hasta el final de la cadena.

Otra función que devuelve un puntero : ReservarMemoria ( )

Se trata de una función de usuario. Normalmente estamos tan acostumbrados a chequear el valor que devuelve ***malloc( )*** que seguramente una primera versión de nuestra función sería:

```
void *ReservarMemoria ( unsigned Size )
{
  void *pRes;
  if((pRes=malloc(Size))==NULL) return(NULL);
  return(pRes);
}
```

y nos parecería muy buena, pero en realidad estamos desperdiciando esfuerzo puesto que en definitiva el siguiente código :

```
void *ReservarMemoria ( unsigned Size )
{ return(malloc(Size)); }
```

Hace exactamente lo mismo, devolviendo una dirección ***void*** que puede contener un valor **NULL** o distinto de **NULL**, según haya podido o no realizar la reserva. El chequeo del resultado queda a cargo del punto de invocación. También debe notarse que al retornar un tipo de puntero ***void*** es casi seguro que la parte del programa que llama a la función deberá utilizar un "cast", como por ejemplo:

if((pENT=(int *)ReservarMemoria(100*sizeof(int)))==NULL) .....

También podríamos haber hecho que ***RerservarMemoria( )*** retornase un puntero a char, un puntero a byte, o a int, etc. Siempre puede recurrirse al cast desde la invocación para compatibilizar el formato. Un poco más adelante cuando analicemos los punteros dobles, propondremos otra versión de esta función.

De nuevo punteros a estructuras.

El caso de las estructuras es muy interesante porque el hecho de que sus miembros pueden también ser apuntadores, le confiere una enorme potencialidad sobre las que se terminan construyendo estructuras de datos tan potentes como:

- Listas encadenadas simples.
- Listas encadenadas dobles.
- Listas encadenadas múltiples.
- Pilas.
- Colas.
- Arboles.

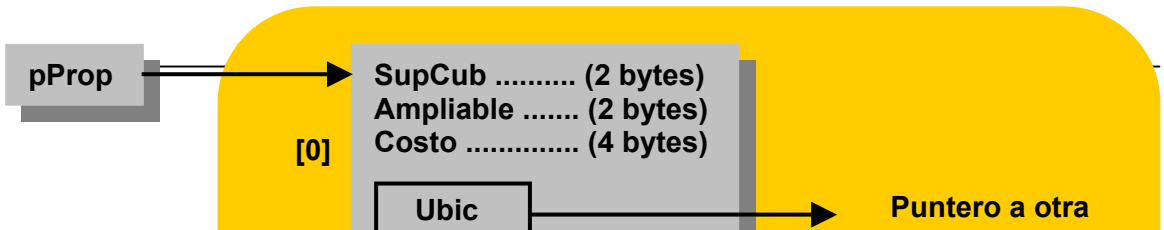
Si bien el estudio de estos ítems es en sí mismo una especialidad que va más allá de los punteros. Por el momento veremos otras instancias, como la siguiente:

Este programa no necesita realizar reserva de memoria para las estructuras en sí, debido a que al declarar un arreglo de estructuras, la reserva se realiza automáticamente (salvo para los punteros internos que serán motivo de nuestro estudio).

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

```
#include <string.h>

typedef enum boolean { false, true };

typedef struct T_Ubic {
                          char Domic[30];
                          char Barrio[20];
                        };

typedef struct T_Prop {
                          T_Ubic * Ubic;
                          int      SupCub;
                          boolean  Ampliable;
                          float    Costo;
                         };

/* ------------------------------------------------------------------------------------------ */
void main( )
{
  T_Prop  Prop[2];
  T_Prop *pProp = Prop;
  int     i;

  clrscr( ); highvideo( );

  for(i=0,pProp=Prop;i<2;pProp++)
     if((pProp->Ubic=(T_Ubic *)malloc(sizeof(T_Ubic)))==NULL) {
          cprintf("NO PUDO HACER RESERVA PARA ANIDAMIENTO.\r\n");
          getch( ); exit(1);
     }

  // --- CARGADO DE DATOS ( para un solo domicilio ) --------

  pProp=Prop;
  strcpy(pProp->Ubic->Domic, "BOLIVIA 3750");
  strcpy(pProp->Ubic->Barrio,"LAS AMERICAS");
  pProp->SupCub=120;
  pProp->Costo=45000.00;

  // --- MOSTRAR DATOS POR PANTALLA -------------------------

  cprintf("DOMICILIO ..... %s\r\n",pProp->Ubic->Domic);
  cprintf("BARRIO ..... %s\r\n",pProp->Ubic->Barrio);
  cprintf("SUP. CUBIERTA .... %d\r\n",pProp->SupCub);
  cprintf("COSTO ..... %.2f\r\n",pProp->Costo);

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

Siempre insistiremos en la conveniencia de visualizar las estructuras:



Si observamos con cuidado veremos que la reserva que el compilador hace para el arreglo es de 20 bytes (10 por cada domicilio) cuando en realidad deberíamos necesitar 120. El puntero llamado ***Ubic*** deberá referenciar una estructura adicional de 50 bytes (30 bytes para ***Domicilio*** + 20 bytes para ***Barrio***), pero en la estructura TProp sólo tenemos el puntero (que es una variable de 2 bytes).

Esto significa que necesitaremos realizar reservas posteriores para la estructura señalada por miembros punteros. Para mejorar nuestro manejo de punteros hemos declarado ***T_Prop *pProp = Prop;*** que es un puntero del mismo tipo que cada domicilio del arreglo (y lo inicializamos precisamente con la dirección de comienzo del array). A través de este puntero concretaremos las reservas del miembro ***Ubic***:

```
for(i=0,pProp=Prop;i <2;pProp++)
    if((pProp->Ubic=(T_Ubic *)malloc(sizeof(T_Ubic)))==NULL) {
          cprintf("NO PUDO HACER RESERVA PARA ANIDAMIENTO.\r\n");
          getch( ); exit(1);
    }
```

Esto significa que el arreglo y las estructuras referenciadas mediante ***Ubic***, residen en lugares distintos de memoria: el array se halla ubicado en el segmento de datos y las estructuras en el ***Heap***.

El siguiente también es un caso de estructuras que disponen de miembros punteros:

Una estructura dada contendrá dos miembros punteros: una que señalará el inicio de una cadena que almacene la hora del sistema y otro que señale otra cadena que almacene la fecha del sistema.

Existe dos estructuras predefinida en la librería <*dos.h*> llamadas

struct date
struct time

de las cuales reproduciremos solo una :

```
struct date {
                int da_year;
                int da_day;
                int da_mon;
             };
```

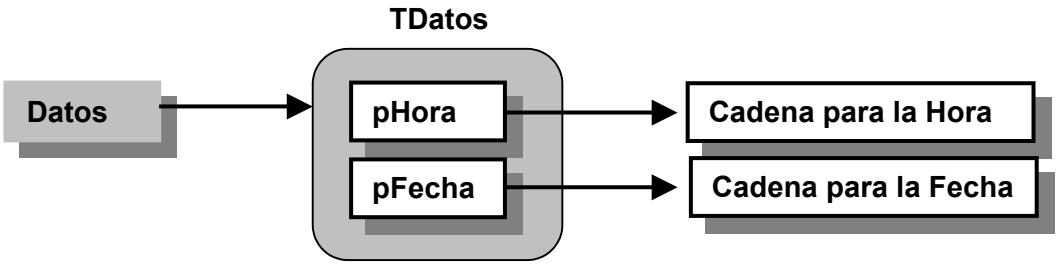
la otra la dejamos para el lector la inspeccione a través del help de BorlandC.

```
#include <conio.h>
#include <dos.h>
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>
#include <process.h>

typedef struct time TTime;
typedef struct date TDate;
typedef struct TDatos {
                        char *pHora;
                        char *pFecha;
                       };

/* ------------------------------------------------------------------------------------------ */
void main( )
{
  TTime   *Hora;
  TDate   *Fecha;
  TDatos *Datos;

  clrscr( ); highvideo( );

  // --- RESERVA PARA LA ESTRUCTURA EN SI ----------------------
        if((Datos=(TDatos *)malloc(sizeof(TDatos)))==NULL) {
            cprintf("NO PUDO HACER RESERVA EN EL HEAP .\r\n");
            getch(); exit(1);
        }

  // --- RESERVA PARA LOS PUNTEROS DE LA ESTRUCTURA -------------
        Datos->pHora =(char *)malloc(12);
        Datos->pFecha=(char *)malloc(12);

  gettime(Hora); getdate(Fecha);

  // --- ASIGNA LOS VALORES A LOS MIEMBROS RESERVADOS ---------
        sprintf(Datos->pHora ,"%02d:%02d:%02d",
                Hora->ti_hour,Hora->ti_min,Hora->ti_sec);
        sprintf(Datos->pFecha,"%02d:%02d:%02d",
                Fecha->da_day,Fecha->da_mon,Fecha->da_year);

  cprintf("HORA  DEL SISTEMA : %s\r\n",Datos->pHora);
  cprintf("FECHA DEL SISTEMA : %s\r\n",Datos->pFecha);

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

Si nos hemos planteado los formatos de fecha y hora como:

dd/mm/aaaa
hh:mm:ss

significa que necesitaremos 11 bytes para la fecha y 9 bytes para la hora. Adoptamos 12 para cada una.



La reserva para la estructura en sí (de 4 bytes), la hacemos con:

if((Datos=(TDatos *)malloc(sizeof(TDatos)))==NULL).....

y para cada miembro puntero :

```
Datos->pHora =(char *)malloc(12);
Datos->pFecha=(char *)malloc(12);
```

Nótese que los punteros para ***Hora*** y ***Fecha*** alimentan directamente la función predefinida sprintf( ) :

```
sprintf(Datos->pHora ,"%02d:%02d:%02d",
        Hora->ti_hour,Hora->ti_min,Hora->ti_sec);

sprintf(Datos->pFecha,"%02d:%02d:%d",
       Fecha->da_day,Fecha->da_mon,Fecha->da_year);
```

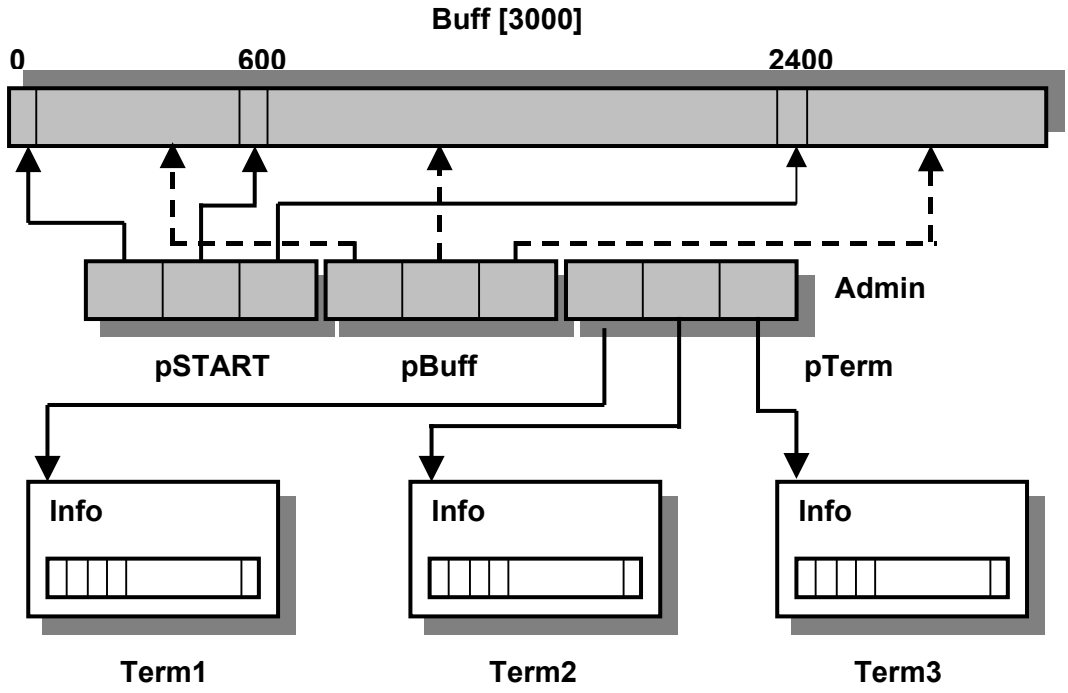
Algunos ejemplos más complejos con punteros a estructuras:

Un arreglo de char de 3000 elementos simulará un banco de memoria de un Servidor Multiusuario, y la misma se hallará particionada en 3 bloques, uno para cada usuario (Terminal) según: 600 - 1800 - 600 bytes, respectivamente.

El administrador de memoria tendrá que conocer la dirección de comienzo de cada SubBuffer y poseer además un indicador de ocupación de cada uno a medida que vaya llegando información.

Las Terminales poseerán únicamente un indicador de existencia de información a llevar a su SubBuffer (SI - NO) y un Buffer local para texto.

Un esquema nos ayudará a comprender el problema :



Los miembros ***pSTART***, ***pBuff*** y ***pTerm***, son arreglos de punteros en los cuales ***pSTART*** tiene formato ***char ** al igual que ***pBuff***, en tanto que ***pTerm*** tiene formato ***pTerm ****

Estos punteros son inicializados en:

```
TAdmin   Admin = { {  Buff,       &Buff[600],  &Buff[2400] },
                   {  Buff,       &Buff[600],  &Buff[2400] },
                   { &Term1,  &Term2,      &Term3       } };
```

```
#include <conio.h>
#include <string.h>
#include <math.h>
#include <dos.h>

typedef enum  boolean  { NO=0, SI=1 };
        const int TERMS = 3;

typedef struct TTerm  {
                          boolean Info;
                          char    Dato[256];
                        };
typedef struct TAdmin {
                          char     *pSTART[TERMS]; // Dir. de comienzo del SubBuff.
                          char     *pBuff[TERMS];       // Dirección actual de c/SubBuff.
                          TTerm *pTerm[TERMS];     // Dirección de la terminal.
                        };

typedef   struct time   TTime;
typedef unsigned char   byte;
          const  int   TIEMPO = 10;
```

```
            const  int   DIM    = 3000;



void AdministrarTerminales(char *Buffer, TAdmin *Admin);
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
   char   Buff[DIM];

  TTerm    Term1 = { SI, "Texto de prueba de la Terminal Nro 1" };
  TTerm    Term2 = { NO, "" };
  TTerm    Term3 = { SI, "TEXTO DE PRUEBA DE LA TERMINAL Nro 2" };

  TAdmin   Admin = { {  Buff,      &Buff[600],  &Buff[2400] },
                     {  Buff,      &Buff[600],  &Buff[2400] },
                     { &Term1, &Term2,      &Term3      }  };

   byte    IndTerm;  // --- Para ir secuenciando cada terminal.
   int     i,j;

  clrscr( ); highvideo( ); IndTerm=0;

  AdministrarTerminales(Buff, &Admin);

  for(IndTerm=0;IndTerm<TERMS;IndTerm++) {
          cprintf("TEMINAL %d\r\n",IndTerm+1);
          if(Admin.pBuff[IndTerm]!=Admin.pSTART[IndTerm])
               cprintf("%s\r\n",Admin.pSTART[IndTerm]);
  }

  while(kbhit( )) getch( );
  getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
void AdministrarTerminales(char *Buff, TAdmin *Admin)
{
  byte IndTerm;
  int  i;

  for(i=0;i<DIM;i++) Buff[i]=0;
  do {
      if(Admin->pTerm[IndTerm]->Info) {
             for(i=0;i<=strlen(Admin->pTerm[IndTerm]->Dato);i++)
                   *Admin->pBuff[IndTerm]++=Admin->pTerm[IndTerm]->Dato[i];
             Admin->pTerm[IndTerm]->Info=NO;
             strcpy(Admin->pTerm[IndTerm]->Dato,"");
      }
      ++IndTerm%=TERMS;

  } while(!kbhit( ));
}
```

Algo llamativo de este programa es que no es necesario realizar ninguna reserva dinámica de memoria, puesto que los punteros se hallan únicamente como miembros de

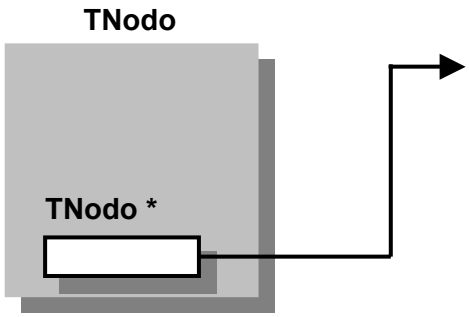
una estructura y las direcciones válidas para ellos es sobre otros datos que también se han generado estáticamente.

El miembro que dice ***Info*** en las Terminales, es un elemento de tipo lógico : si el Terminal dispone de información para enviar al Buffer del Servidor, se hallará en ***SI***, caso contrario en ***NO***.

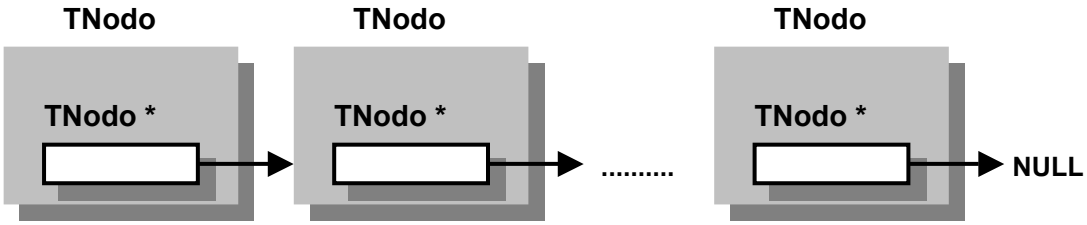
El Administrador de Terminales chequeará permanentemente si algún Terminal posee mensajes para enviar al Buffer. Si es así lo colocará donde corresponda y pasará el indicador ***Info*** a valor ***NO***. Obviamente, como se trata de un pequeño programa de simulación, una o dos pasadas serán suficientes para transmitir los mensajes, luego el chequeo continuará hasta tanto se oprima una tecla cualquiera.

Estructuras Autoreferenciadas : Listas encadenadas

Una particularidad de gran potencia, es el hecho de que un miembro de la estructura pueda apuntar a un objeto ***del mismo tipo*** al cual él pertenece. Puesto esquemáticamente :



Si mediante algún mecanismo vamos generando nuevas estructuras de TNodo y las vamos conectando al puntero indicado arriba, obtendremos :



y a esto se denomina ***lista simplemente enlazada***.

Nuestra primera lista encadenada.

Crear una lista simple enlazada que almacene los 10 primeros términos de la Serie de Fibonacci.

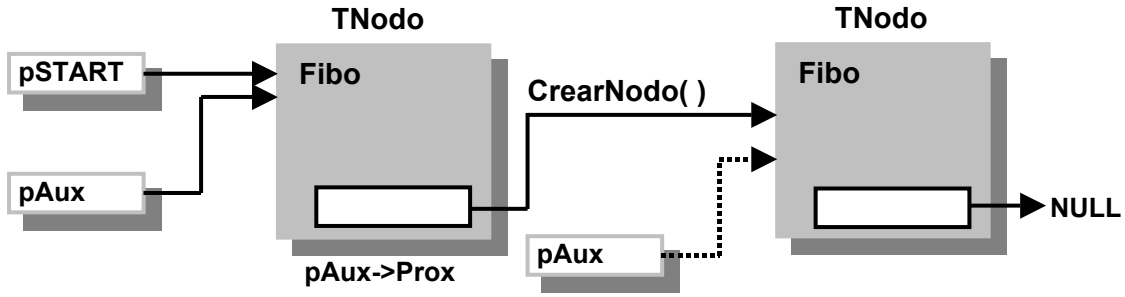
Un detalle de las listas encadenadas es que no debemos perder bajo ninguna circunstancia la **DIRECCION DE INICIO** de la misma. Debido a ello, para su generación se utilizan normalmente dos punteros: uno fijo de comienzo de lista y el otro móvil de generación.

La instrucción:

if((pAux=pSTART=CrearNodo( ))!=NULL) pSTART->Fibo=1; else exit(1);

- Crea el nodo cabecera de la lista.
- Inicializa el puntero auxiliar.
- Asigna el primer valor de Fibonacci al miembro Fibo.

Los próximos nodos se generan:



a partir de ***pAux->Prox*** y luego se reactualiza ***pAux*** a ***pAux->Prox*** como se indica en línea de trazos.

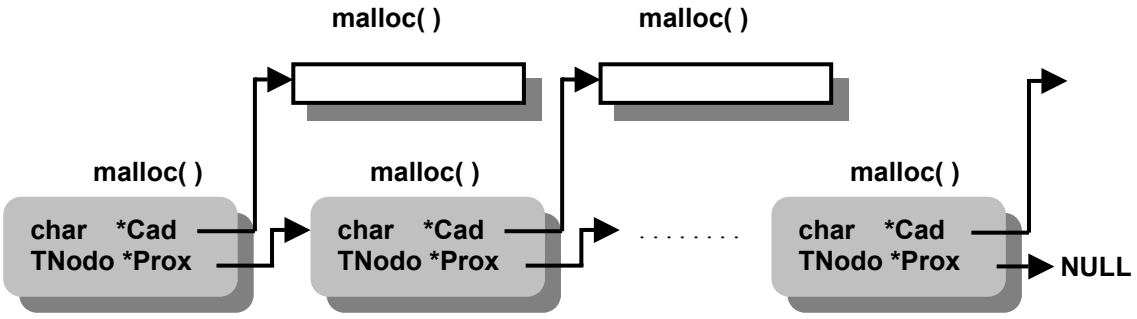
Es importante finalizar siempre al lista con una dirección **NULL**, que nos servirá de referencia para recorrerla. Una buena práctica sería hacer pAux->Prox=NULL luego de cada reactualización de pAux (ver código).

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

typedef struct TNodo {
                        int       Fibo;
                        TNodo *Prox;
                      };

TNodo *CrearNodo  ( void );
/* ------------------------------------------------------------------------------------------------------------ */



void main( )
{
```

```
  TNodo  *pSTART;
  TNodo  *pAux;
  int         Ant,Actual,Fibo;
  int         i;

  clrscr( ); highvideo( );

  // --- CREA EL NODO INICIAL ------------------------------------------
          if((pAux=pSTART=CrearNodo())!=NULL) pSTART->Fibo=1; else exit(1);

  // --- CREA LOS NODOS SIGUIENTES DE LA LISTA -------------
  for(Ant=0,Actual=1,Fibo=1,i=0;i<9;i++) {
     Ant=Actual; Actual=Fibo; Fibo=Ant+Actual;
     if((pAux->Prox=CrearNodo())!=NULL) {
           pAux=pAux->Prox; pAux->Fibo=Fibo;
           pAux->Prox=NULL;
     }
     else { cprintf("FALLA EN CREACION DEL NODO.\r\n"), getch(); exit(1); }
  }
  // --- MUESTRA CONTENIDO DE LOS NODOS POR PANTALLA -----------------
  for(pAux=pSTART;pAux;pAux=pAux->Prox) cprintf("%d  ",pAux->Fibo);
  getch();
}
/* ------------------------------------------------------------------------------------------------------------ */
TNodo *CrearNodo( )
{  return((TNodo *)malloc(sizeof(TNodo))); }
/* ------------------------------------------------------------------------------------------------------------ */
```

Lista simple encadenada "a medida".

Generar la siguiente lista simple encadenada :



de acuerdo a la siguiente modalidad :

- *Ingresar por teclado una cadena de caracteres (por ej. apellidos).*
- *Invocar una función llamada CrearNodo( ) definida según :*

TNodo *CrearNodo(char *CadenaIngresadaPorTeclado);

*que efectuará una reserva dinámica de memoria **A MEDIDA** tanto para el Nodo en sí como para la cadena apuntada por uno de sus miembros. Además incorporará la cadena ingresada por teclado al miembro char ***Cad.***

- *La lista se irá construyendo hasta el momento en que ingrese una cadena vacía.*

Finalmente listar por pantalla el contenido completo de la lista.

NOTA

Si ingresa cadenas normales que contengan espacios en blanco, utilice la función ***gets( )*** *y recuerde que le agrega un carácter extra (new line) que deberá eliminar haciendo :* ***Cadena[strlen(Cadena)]='\0'*** *para transformarla en una cadena normal.*

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <process.h>
#include <alloc.h>

typedef struct TNodo {
                        char    *Apellido;
                        TNodo *Prox;
                      };

TNodo *CrearNodo    ( char *Cad          );
void      MostrarLista  ( TNodo *pSTART );

/* ------------------------------------------------------------------------------------------------ */
void main( )
{
  TNodo  *pSTART;
  TNodo  *pAUX;
  char     *Apell;

  clrscr( ); highvideo( );

  cprintf("APELLIDO : "); gets(Apell);
  Apell[strlen(Apell)]='\0';

  if((pSTART=CrearNodo(Apell))!=NULL) {
      pAUX=pSTART;
      do {
            cprintf("APELLIDO : "); gets(Apell);
            Apell[strlen(Apell)]='\0';
            if(strlen(Apell)) {
                  if((pAUX->Prox=CrearNodo(Apell))==NULL) {
                     cprintf("NO PUDO CREAR NODO.\r\n");
                     getch(); exit(1);
                  }
                  pAUX=pAUX->Prox; pAUX->Prox=NULL;
            }
          } while(strlen(Apell));
  }
  MostrarLista(pSTART);
  getch( );
}
```

```
/* ------------------------------------------------------------------------------------------------------ */
TNodo *CrearNodo(char *Cad)
{
  TNodo *pLocal;

  if((pLocal=(TNodo *)malloc(sizeof(TNodo)))==NULL) return(NULL);
  if((pLocal->Apellido=(char *)malloc(strlen(Cad)+1))==NULL) return(NULL);
  strcpy(pLocal->Apellido,Cad);

  return(pLocal);
}
/* ------------------------------------------------------------------------------------------------------ */
void MostrarLista(TNodo *pSTART)
{
   while(pSTART)  { cprintf("%s\r\n",pSTART->Apellido); pSTART=pSTART->Prox; }
}
/* ------------------------------------------------------------------------------------------------------ */
```

Aquí la función CrearNodo( ) no es tan simple como las vistas anteriormente por la sencilla razón que la misma realiza más de una tarea:

- Emplazar memoria para el Nodo.
- Emplazar memoria para la cadena.
- Cargar la cadena pasada como parámetro.

Como la lista enlazada siempre finaliza con un ***NULL***, el recorrerla resulta trivial :

while(pSTART) { cprintf("%s\r\n",pSTART->Apellido); pSTART=pSTART->Prox; }

y el hecho de mover la posición inicial de ***pSTART*** carece de importancia, puesto que al ser un parámetro dentro de una función, y haber sido pasado ***POR VALOR***, resulta una ***copia local*** de la dirección de inicio que se halla a salvo en el punto de invocación.

Simulación de comunicación TOKEN RING.

*Utilizando una **lista simple encadenada circular**, simular seis puestos de trabajo cuya configuración, utilizando estructuras autoreferenciadas, será la siguiente:*

***Destino** y **Buff** se utilizan para **ENVIAR** mensajes entre puestos (a través del **Token**), Destino indica la máquina o Puesto al que se envía el mensaje y **Buff** (que es un arreglo de char) contiene la información a enviar.*

***Origen** y **Buff** (un poco más abajo) representan un complemento de lo anterior y son utilizados para **RECIBIR** mensajes de otros Puestos.*

*El indicador que dice **Estado** es una variable booleana que puede tener dos valores: "**ON/OFF**". Si se halla en **ON** significa que desea enviar un mensaje al Puesto indicado en **Destino**. Caso contrario el **Puesto** no pide ninguna acción al **Token**.*

IMPORTANTE

*Independientemente del contenido de **Estado** un **Puesto** puede **SIEMPRE** recibir un mensaje que le envíe algún otro **Puesto**.*

EL "TOKEN"

*El **Token** es un paquete o bandeja que recorre permanentemente el anillo de Puestos transportando mensajes de un lugar a otro. Naturalmente el **Token** arranca **VACIO** y va chequeando si el indicador **Estado** de cada **Puesto** se halla activo. Si no está activo pasa al **Puesto** siguiente, caso contrario sabe que en ese Puesto existe un mensaje a transportar y realiza la siguiente tarea:*

- ***Cambia su indicador propio de "Estado" a ON.***
- ***Asigna "Destino" con el nombre del Puesto que recibirá el mensaje.***
- ***Transfiere el contenido de Buff (del Puesto) a su propio Buff.***
- ***Cambia el indicador "Estado" del Puesto a OFF.***
- ***Sigue su camino al Puesto siguiente.***

*Una vez que el **Token** posee activado su indicador de **Estado**, no puede tomar ningún mensaje nuevo hasta tanto no se haya descargado su **Buff** en el **Puesto** de destino.*

*La descarga del **Token** consiste en lo siguiente:*

- ***Detecta el Puesto destino para el mensaje.***
- ***Independientemente si el Puesto tiene o no activado su indicador "Estado".***
- ***Transfiere su Buff propio al Buff de Recepción del Puesto.***
- ***Cambia su indicador propio de "Estado" a OFF. (Token disponible).***
- ***Pasa al Puesto siguiente.***

A partir de allí comienza nuevamente a chequear si existen nuevos mensajes a transportar.

***NOTA** : El **Token** se hallará recorriendo el anillo hasta tanto se pulse una tecla.*

VALORES INICIALES PARA CADA PUESTO.

PUESTO 'B'

*Estado = **ON***
*Dest = **'C'***
*Buff = **"LA RECEPCION FUE EXITOSA"***

PUESTO 'C'

*Estado = **ON***
*Dest = **'E'***
*Buff = **"PROBLEMAS DE HARDWARE"***

Todos los demás Puestos se inicializarán con valores nulos, o sea :

Estado = OFF
Dest = ' '
Buff = "" (de Envío)

Orig = ' '
Buff = "" (de Recepción).

*Si observa bien estos valores y las reglas de funcionamiento del **Token**, notará que el mismo necesita de más de una vuelta al anillo para cumplir con su tarea de mensajería.*

Finalmente muestre por pantalla si los mensajes se han distribuido correctamente.

```
#include <conio.h>
#include <alloc.h>
#include <string.h>
#include <process.h>
#include <dos.h>

typedef enum boolean { OFF=0, ON=1 };



typedef struct TRecepc {
                          char Orig;
                          char Buff[120];
                        };
```

```
typedef struct TEnvio    {
                           char Dest;
                           char Buff[120];
                         };

typedef struct TPuesto   {
                           char     Puesto;
                           boolean  Estado;
                           TEnvio   Envio;
                           TRecepc  Recep;
                           TPuesto  *Prox;
                         };

typedef struct TToken    {
                           boolean  Estado;
                           TEnvio   Envio;
                           TPuesto *Puesto;
                         };

/* ------------------------------------------------------------------------------------------------------- */
void main( )
{
  TToken     Token;
  TPuesto   *PuestoA;
  TPuesto   *pAux;
   char      Puesto;

  clrscr( ); highvideo( );

  // --- GENERA EL PRIMER PUESTO DE TRABAJO ----------------------
  if((pAux=PuestoA=(TPuesto *)malloc(sizeof(TPuesto)))==NULL) {
     cprintf("NO PUDO RESERVAR MEMORIA PARA PUESTO.\r\n");
     getch(); exit(1);
  }
  PuestoA->Puesto='A';  strcpy(PuestoA->Recep.Buff,"");

  // --- GENERA LOS PUESTOS RESTANTES --------------------------------
  for(Puesto='B';Puesto<='E';Puesto++) {
     if((pAux->Prox=(TPuesto *)malloc(sizeof(TPuesto)))==NULL) {
           cprintf("NO PUDO RESERVAR PARA PUESTO.\r\n");
           getch(); exit(1);
     }
     pAux=pAux->Prox;
     pAux->Puesto=Puesto;          // --- NOMINA EL PUESTO ACTUAL.
     strcpy(pAux->Recep.Buff,"");  // --- INICIALIZA BUFFER DE RECEPCION.
     strcpy(pAux->Envio.Buff,"");  // --- INICIALIZA BUFFER DE ENVIO.



     switch(pAux->Puesto) {
           case 'B' : pAux->Estado=ON;
                      pAux->Envio.Dest='C'; // --- PUESTO 'B' ENV A PUESTO 'C'
                      strcpy(pAux->Envio.Buff,"LA RECEPCION FUE EXITOSA");
```

```
                    break;
          case 'C' : pAux->Estado=ON;
                     pAux->Envio.Dest='E'; // --- PUESTO 'C' ENV A PUESTO 'E'
                     strcpy(pAux->Envio.Buff,"PROBLEMAS DE HARDWARE");
                     break;
    }
  }

  pAux->Prox=PuestoA;            // --- COMPLETA EL CIRCUITO DE PUESTOS.

  // --- COMIENZA A CIRCULAR EL TOKEN --------------------------------

  Token.Puesto=PuestoA;      // --- CARGA DIRECCION DE PUESTO INICIAL.
  Token.Estado=OFF;          // --- ARRANCA CON EL TOKEN DISPONIBLE.

  do {
     if(Token.Estado==OFF) {   // --- TOKEN DISPONIBLE PARA TOMAR MENSAJE.
          if(Token.Puesto->Estado==ON) { // PUESTO CON MENSAJE A ENVIAR.
                    Token.Estado=ON;
                    Token.Envio.Dest=Token.Puesto->Envio.Dest;
                    strcpy(Token.Envio.Buff,Token.Puesto->Envio.Buff);
                    strcpy(Token.Puesto->Envio.Buff,"");
                    Token.Puesto->Estado=OFF; // PUESTO SIN MESSG A ENVIAR.
          }
     } else { // --- TOKEN OCUPADO CON MENSAJE PARA UN PUESTO.
               if(Token.Puesto->Puesto==Token.Envio.Dest) {
                         strcpy(Token.Puesto->Recep.Buff,Token.Envio.Buff);
                         strcpy(Token.Envio.Buff,"");
                         Token.Puesto->Recep.Orig=Token.Envio.Dest;
                         Token.Estado=OFF;
               }
          }
     Token.Puesto=Token.Puesto->Prox;
     cprintf("%c",'.'); delay(50);

  } while(!kbhit( ));

  // --- MUESTRA LOS MENSAJE RECIBIDOS POR LOS PUESTOS ----------------

  cprintf("\r\n");
  for(pAux=PuestoA,Puesto='A';Puesto<='E';Puesto++,pAux=pAux->Prox) {
     cprintf("PUESTO %c : ",pAux->Puesto);
     cprintf("%s\r\n",pAux->Recep.Buff);
  }

  while(kbhit( )) getch( );  getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Generación de una lista ordenada.

Generar una lista simple enlazada ORDENADA, de tal suerte que al ir leyendo un arreglo numérico (de int), vaya generando Nodos con los valores ya ordenados.

NOTA

*En este programa se ha hecho algo indebido : definir **pSTART** (comienzo de la lista) en forma GLOBAL (no es lo más práctico), pero ello se debe a que en el momento de desarrollar este problema el lector aún no ha visto **PUNTEROS DOBLES**, elemento necesario para pasar una variable puntero **POR REFERENCIA**. Más adelante volveremos sobre este programa.*

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

const  int DIM = 10;
typedef struct TNodo {
                          int    Dato;
                        TNodo *Prox;
                      };

TNodo *pSTART;

TNodo *CrearNodo     ( int Size                          );
TNodo *BuscarNodo    ( TNodo *pSTART, int Dato     );
void     InsertarNodo    ( TNodo *pOld, TNodo *pNew );
void     MostrarLista    ( TNodo *pSTART                );
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  int         Vector[DIM] = { 1420,562,875,460,932,155,720,100,348,20 };
  TNodo  *pAUX;
  TNodo  *pINS;
  int         i;

  clrscr( ); highvideo( );

  (pSTART=CrearNodo(sizeof(TNodo)))->Dato=Vector[0]; pSTART->Prox=NULL;

  for(i=1;i<DIM;i++) {
         pAUX=CrearNodo(sizeof(TNodo)); pAUX->Dato=Vector[i];
         pINS=BuscarNodo(pSTART,Vector[i]);
         InsertarNodo(pINS,pAUX);
  }
  MostrarLista(pSTART);

  getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */


TNodo *CrearNodo(int Size)
{
  TNodo *pLocal;
```

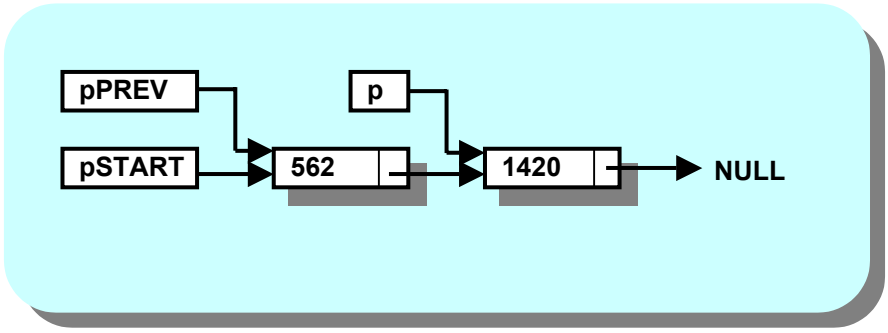
```
  if((pLocal=(TNodo *)malloc(Size))==NULL) {
     cprintf("CrearNodo( ) NO PUDO RESERVAR NODO.\r\n");
     getch(); exit(1);
  }
  return(pLocal);
}
/* ------------------------------------------------------------------------------------------------------------ */
TNodo *BuscarNodo(TNodo *pSTART,int Dato)
{
  TNodo *p     = pSTART;
  TNodo *pPREV = pSTART;

  while(Dato>(p->Dato) && p) { pPREV=p; p=p->Prox; }
  if(p!=pSTART) return(pPREV+1);  // PARA EVITAR INDETERMINACION SI DEBE
  return(pSTART);                 // INSERTAR LUEGO DEL PRIMERO.
}
/* ------------------------------------------------------------------------------------------------------------ */
void InsertarNodo(TNodo *pINS, TNodo *pNuevo)
{
  if(pINS==pSTART) { pNuevo->Prox=pSTART; pSTART=pNuevo; }
  else { pINS--; pNuevo->Prox=pINS->Prox; pINS->Prox=pNuevo; }
}
/* ------------------------------------------------------------------------------------------------------------ */
void MostrarLista(TNodo *pSTART)
{
  TNodo *p = pSTART;
  while(p) { cprintf("%d  ",p->Dato); p=p->Prox; }
}
/* ------------------------------------------------------------------------------------------------------------ */
```

La idea es tomar cada elemento del Vector y recorrer los elementos de la lista formada con los elementos previos y ver dónde debe insertarse el término actual. Para ello disponemos de la función BuscarNodo( ) que describiremos más en detalle.

```
TNodo *p     = pSTART;
TNodo *pPREV = pSTART;

while(Dato>(p->Dato) && p) { pPREV=p; p=p->Prox; }
if(p!=pSTART) return(pPREV+1);   // PARA EVITAR INDETERMINACION SI DEBE
return(pSTART);                  // INSERTAR LUEGO DEL PRIMERO.
```

Esta función trabaja con dos punteros, puesto que al hallar que el elemento a insertar es ***menor*** que el dato actual de la lista, lo que en realidad necesita es la dirección del Nodo anterior para poder hacer la inserción. Sin embargo se presenta una situación inconsistente cuando el elemento nuevo debe insertarse en la cabecera de la lista (nunca entra al while), o cuando debe insertarse luego del primer elemento : en ambos casos la dirección del nodo previo es el valor de ***pSTART***. ¿Cómo diferenciamos uno de otro? Chequeando el puntero p : Si se ha movido al menos un lugar, su dirección es

diferente a ***pSTART*** (el que es igual a ***pSTART*** es ***pPREV***). En ese caso retornamos ***pPREV+1***, caso contrario devolvemos ***pSTART***. Una situación donde se ve claramente esto, es la siguiente:



De esta manera el valor devuelto por ***BuscarNodo( )*** tiene dos posibilidades:

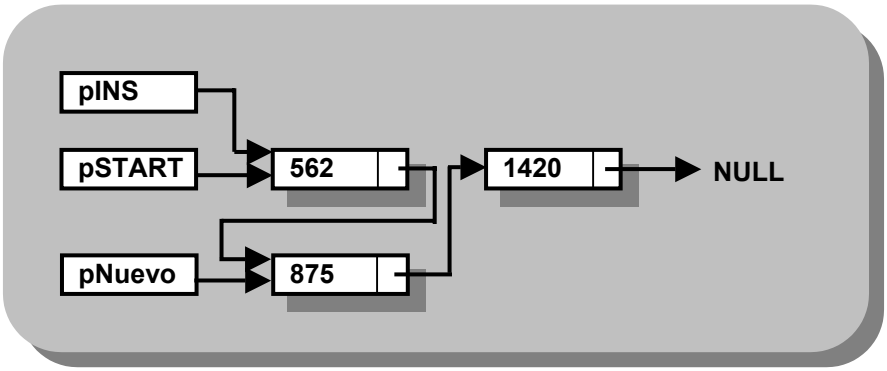
- *Es **pSTART**.*
- *Es cualquier otro valor pero... **incrementado en 1**!*

Este hecho debemos tenerlo bien presente en la función ***InsertarNodo( )*** que es un poco el complemento de ésta.

```
void InsertarNodo(TNodo *pINS, TNodo *pNuevo)
{
  if(pINS==pSTART) { pNuevo->Prox=pSTART; pSTART=pNuevo; }
  else { pINS--; pNuevo->Prox=pINS->Prox; pINS->Prox=pNuevo; }
}
```

Esta función chequea si la posición a insertar es ***pSTART***, como un caso especial (insertar en la cabecera de la lista), caso contrario "sabe" que la posición de inserción se halla incrementada en 1, por lo que debe decrementarla en igual cantidad ***antes*** de utilizarla.

Nótese que si continuamos analizando el esquema superior, al decrementar ***pINS*** se transforma en ***pSTART***, pero el tratamiento de la inserción es distinto:



Cuando se trabaja con punteros, la inserción se transforma simplemente en ***cambiar*** la magnitud almacenada por los punteros, ***sin necesidad*** de mover físicamente los datos.

Punteros a punteros (punteros dobles).

Definición :

Un puntero doble es un puntero cuyo contenido es la dirección de otro puntero que a su vez señala el lugar donde se almacena el dato en sí.

Su declaración formal es :

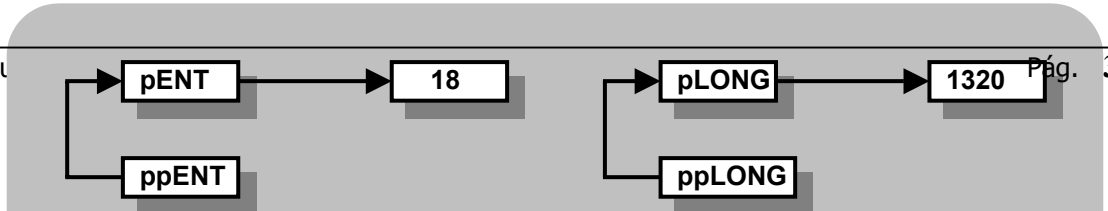
long **pLONG;
float **pFLOAT;
TDato **pDATO;
etc.

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

/* ------------------------------------------------------------------------------ */
void main( )
{
  int      *pENT     = (int *)malloc(sizeof(int));
  long    *pLONG   = (long *)malloc(sizeof(long));
  int     **ppENT;
  int     **ppLONG;

  clrscr( ); highvideo( );

  *pENT=18;
  ppENT=&pENT;
  cprintf("ENTERO=%d\r\n",**ppENT);

  *pLONG=1320;
 cprintf("LONG=%d\r\n",**ppLONG);

 ppENT=&((int *)pLONG);
 cprintf("ENTERO=%d\r\n",**ppENT);


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

Como siempre, ***antes*** de utilizar un puntero debemos asegurarnos que el mismo disponga de una dirección válida. Recuerde el lema ***apunte*** .... y ***luego dispare*** :

int   *pENT   = (int *)malloc(sizeof(int));
long *pLONG = (long *)malloc(sizeof(long));

el esquema funcional del programa anterior quedaría de la siguiente manera :



A pesar de la sencillez de la operación : ***ppENT=&pENT;*** debemos tener cuidado de que la dirección devuelta por el operador (**&**) ***sea compatible*** con lo esperado por la variable receptora, en este caso : la dirección de una variable que apunte a un entero. Idénticamente ocurre otro tanto con : ***ppENT=&((int *)pLONG);***

Para hacer referencia al dato numérico final, utilizamos la sintaxis:

```
cprintf("ENTERO=%d\r\n",**ppENT);
cprintf("LONG=%d\r\n",**ppLONG);
```

esta notación con los dos asteriscos se denomina una ***doble indirección***.
Al igual que con los punteros simples, los punteros dobles admiten la utilización de un ***cast*** para conversiones de tipo:

```
ppENT=&((int *)pLONG);
cprintf("ENTERO=%d\r\n",**ppENT);
```

Sin embargo aquí existe una pequeña trampa escondida. Al hacer la conversión de ***long*** a ***int*** solamente se tomarán los dos bytes menos significativos del tipo más grande. Por esta razón hemos asignado al puntero a ***long*** un valor pequeño como 1320, de manera que al hacer la conversión no se pierda información por recorte.

Debe tenerse cuidado en la secuencia del cast : primero debe convertirse el tipo de dirección devuelto por el operando y recién aplicar el operador &.

Un caso particular donde la doble indirección se toma con un solo asterisco, es el siguiente:

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

```
/* ------------------------------------------------------------------------------------------------ */
void main( )
{
  char    *Car    = (char *)malloc(sizeof(char));
  char    *Texto = "BORLANDC PARA PROFESIONALES.";
  char  **ppFrase;
  char  **ppCar;

  clrscr( ); highvideo( );

  *Car='C';
  ppCar=&Car;
  cprintf("CARACTER : %c\r\n",**ppCar);

  ppFrase=&Texto;
  cprintf("FRASE    : %s\r\n",*ppFrase);
  getch( );
}
/* ------------------------------------------------------------------------------------------------ */
```

A pesar de que ***Car*** y ***Texto*** son ambas punteros a char, su sentido es distinto: ***Car*** está apuntando a un dato individual en tanto que ***Texto*** señala un conjunto de caracteres que finalizarán con terminador **\0**

Si bien el mecanismo de la asignación de direcciones a los punteros dobles sigue siendo el mismo:

ppCar=&Car;
ppFrase=&Texto;

al tomar las indirecciones tendremos :

cprintf("CARACTER : %c\r\n",**ppCar);
cprintf("FRASE   : %s\r\n",*ppFrase);

en el caso de la Frase es correcto poner ****ppFrase*** debido a que no vamos a acceder a un dato individual, sino al inicio (***dirección***) de una secuencia de datos hasta hallar el finalizador, y esa dirección de inicio la posee casualmente la dirección apuntada por ***ppFrase***.

Punteros por referencia en funciones.

Aquí aparece una aplicación sumamente importante de los punteros dobles. Hemos dicho que a excepción de los arreglo, las funciones en "C" asumen por defecto que los parámetros pasan por ***VALOR***.

Observe detenidamente el siguiente ejemplo:

*Declarar un arreglos de punteros a enteros en el **main( )** y pasar cada uno de sus domicilios a una función que se encargará de asignarle memoria y un valor numérico adecuado. Verificar luego desde el main( ) si la operación funcionó correctamente.*

NOTA :
*En pantalla deberá aparecer basura y ello se debe a que hemos pasado **por valor** un puntero, de tal suerte que la dirección asignada en la función **muere con ella** al finalizar la invocación.*

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

const int DIM = 3;
void ReservarMemoria(int *pENT);

/* ------------------------------------------------------------------------------------------------------ */
void main()
{
  int  *Vect[DIM];
  int   i;

  clrscr( ); highvideo( );

  for(i=0;i<DIM;i++)
     { ReservarMemoria(Vect[i]); cprintf("%d\r\n",*Vect[i]); }

  getch();
}
/* ------------------------------------------------------------------------------------------------------ */
void ReservarMemoria(int *pENT)
{
   static int Count = 100;
   *(pENT=(int *)malloc(sizeof(int)))=Count++;
}
/* ------------------------------------------------------------------------------------------------------ */
```

El error ha sido puesto ex profeso para destacar una equivocación de interpretación bastante común en "C" :

Si bien es cierto que hemos pasado un puntero a la función ***ReservarMemoria( ),*** le estamos pasando una ***copia local*** del puntero original (que aún no tiene asignada una dirección válida). La función con toda corrección la asigna un valor en el Heap y le carga una magnitud numérica, pero como se trata de un ***valor local*** el mismo desaparece al finalizar la invocación de la función.

Moraleja: la variable puntero ***pENT*** debe ser pasada por ***REFERENCIA*** y e aquí la versión correcta del programa anterior:

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

```
const int DIM = 3;
void ReservarMemoria(int *(*pENT));

/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  int  *Vect[DIM];
  int   i;

  clrscr( ); highvideo( );

  for(i=0;i<DIM;i++)  { ReservarMemoria(&Vect[i]); cprintf("%d\r\n",*Vect[i]); }
  getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
void ReservarMemoria(int *(*pENT))
{
   static int Count = 100;
   *(*pENT=(int *)malloc(sizeof(int)))=Count++;
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Ahora en el prototipo de la función hemos declarado:

void ReservarMemoria(int *(*pENT));

que también podría haber sido escrita como:

void ReservarMemoria(int **pENT);

pero la primera arroja un poco más de claridad lógica :

Es la DIRECCION de una variable que a su vez contiene la DIRECCION de un entero.

También debe ponerse atención en la asignación del puntero doble:

(*pENT=(int *)malloc(sizeof(int)))=Count++;

en realidad aquí existe una implicitación con una doble asignación:

pENT=(int *)malloc(sizeof(int)) para la asignación de dirección del puntero y el otro paréntesis y asterisco para la asignación de la magnitud referenciada.

Ahora retornaremos a nuestro problema de la lista ordenada y en especial a la función InsertarNodo( ).

void InsertarNodo(TNodo **pSTART, TNodo *pOld, TNodo *pNew);

Se recordará que ***pSTART*** se había definido en forma ***GLOBAL*** porque en ese momento no sabíamos trabajar con punteros dobles. En realidad ésta es la forma correcta en que debía resolverse el problema: pasando el puntero ***pSTART*** por referencia.

En la definición de la función en sí tendremos:

```
void InsertarNodo(TNodo **pSTART,TNodo *pINS, TNodo *pNuevo)
{
   if(pINS==*pSTART)    { pNuevo->Prox=*pSTART; *pSTART=pNuevo; }
   else { pINS--; pNuevo->Prox=pINS->Prox; pINS->Prox=pNuevo; }
}
```

que modificará efectivamente la dirección original de ***pSTART*** cuando deba insertar un ***Nodo*** por la cabecera de la lista. Notará que el bloque ejecutable es el mismo, salvo que en lugar de trabajar sobre la definición global del puntero, lo hace sobre un parámetro por referencia.

Arreglo de cadenas : otro caso de punteros dobles.

Pasar un arreglo de punteros a cadena como parámetro de una función que será la encargada de mostrarlo por pantalla.

Si observamos la siguiente declaración :

```
char *Messgs[ ] = { "DISKETTERA NO PREPARADA",
                    "DISCO PROTEGIDO CONTRA ESCRITURA",
                    "VERSION DE S.O. NO COMPATIBLE",
                    "ERROR FATAL",
                    ""                          };
```

podremos notar dos cosas : cada uno de los domicilios del arreglo es una dirección hacia una cadena (char *) y el arreglo en sí también es una dirección (a la zona reservada por el compilador para el vector de direcciones).
Esto ya nos hace sospechar que si deseamos pasar estos datos a una función seguramente estará involucrado un puntero doble. Veamos el código resuelto:

FORMA I

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

void ImprimirMensajes ( char **Messgs );
/* ------------------------------------------------------------------------------------------------------ */



void main( )
{
  char *Messgs[ ] = { "DISKETTERA NO PREPARADA",
                      "DISCO PROTEGIDO CONTRA ESCRITURA",
```

```
                        "VERSION DE S.O. NO COMPATIBLE",
                        "ERROR FATAL",
                        ""                          };

  clrscr( ); highvideo( );
  ImprimirMensajes(Messgs);
  getch( );
}
/* ------------------------------------------------------------------------------------------------------ */
void ImprimirMensajes(char **Messgs)
{  while(strlen(*Messgs)) cprintf("%s\r\n",*Messgs++); }
/* ------------------------------------------------------------------------------------------------------ */
```

En el prototipo de la función : ***void ImprimirMensajes (char **Messgs);***
que también podemos ponerla como : ***void ImprimirMensajes (char *(*Messgs));*** vemos claramente que le pasamos la dirección de ***Messgs*** donde cada uno de sus elementos es a su vez un puntero a char (***char ****).

Idénticamente la función debe declararse como :

void ImprimirMensajes(char **Messgs)
{ while(strlen(*Messgs)) cprintf("%s\r\n",*Messgs++); }

o también :

void ImprimirMensajes(char *(*Messgs))
{ while(strlen(*Messgs)) cprintf("%s\r\n",*Messgs++); }

La expresión : ***while(strlen(*Messgs))*** dice que "mientras la longitud de cada cadena del vector sea distinta de nula, deberá mostrarla por pantalla.

Un detalle destacable :

A pesar de que ***Messgs*** es recibido como un puntero doble ***no se trata del puntero original***, sino de una ***copia local*** de un puntero hacia el inicio del vector de direcciones. Lo prueba el hecho de que a pesar de que estamos modificando su dirección mediante el operador ++ : ***cprintf("%s\r\n",*Messgs++); }*** la dirección original no se pierde.

Esto es fácil de comprobar agregando en el main( ) el siguiente código :

for(int i=0;strlen(*(Messgs+i));i++) cprintf("%s\r\n",*(Messgs+i));

que reproduce correctamente las cadenas originales. Es muy importante que el lector tenga en mente esta sutileza que acabamos de mostrar.

Si en lugar de utilizar la sintaxis mostrada hubiésemos hecho :

while(strlen(*Messgs)) cprintf("%s\r\n",*Messgs++);

El compilador hubiese protestado seriamente durante la compilación advirtiendo que no se puede modificar la dirección original de un arreglo.

La otra forma de pasar el arreglo como parámetro era:

FORMA II

```
void ImprimirMensajes ( char *Messgs[ ] );
/* ------------------------------------------------------------------------------------------ */
void ImprimirMensajes(char *Messgs[ ])
{ int i=0; while(strlen(Messgs[i])) cprintf("%s\r\n",Messgs[i++]); }
/* ------------------------------------------------------------------------------------------ */
```

directamente con la notación de arreglos, en este caso un arreglo de punteros a char.

Obsérvese que ***Messgs[i]*** es obviamente una dirección o puntero a char, de ahí que se lo puede utilizar directamente como argumentos del ***cprintf( )*** o del ***strlen( )***.

Ahora vamos a combinar las listas enlazadas con los punteros dobles en un problema un poco más complejo que los vistos hasta ahora. He aquí el enunciado:

*Ud. dispone en el **main( )** de los siguientes arreglos de punteros :*

```
char *Apell  [DIM] = { "ROQUETA","LOCATTI","JADUR","HOFFMANN","LIVETTI" };
char *Nomb[DIM]  = { "Leandro", "Carlos", "Miguel", "Karl",   "Darío"   };
```

*En el mismo **main( )** definirá dos punteros de comienzo : **START_A** y **START_N** para elaborar una lista encadenada de apellidos y otra aparte de nombres, a cargo de la función :*

```
int GenerarListas( TNodoApell ....., TNodoNomb  .....);
```

*que recibirá como parámetros los punteros de **INICIO** de cada una.*

*Los tipos **Nodo** serán los siguientes:*

```
typedef struct TNodoNomb  {
                            char          *Nombre;
                            TNodoNomb  *Prox;
                          };

typedef struct TNodoApell {
                           char          *Apellido;
                           TNodoApell   *Prox;
                           TNodoNomb  *Nombre;
                         };
```

*Una vez generadas las listas, y siempre desde el **main( )**, invocará a la función :*

```
void  CargarListas ( TNodoApell  .....,
                     TNodoNomb   .....,
                     char        .....,
                     char        .....   );
```

que recibirá como parámetros los punteros de inicio de cada lista y los arreglos de punteros a char con los apellidos y nombres.

*Habrá notado que el **TNodoApell** posee un miembro que es un puntero a **TNodo-Nomb**. Esta cualidad tiene el siguiente propósito: Ligar cada apellido con algún nombre. Por ej. :*

```
char *Apell   [DIM] = { "ROQUETA","LOCATTI","JADUR",  "HOFFMANN","LIVETTI" };
char *Nomb [DIM] = { "Miguel", "Darío",  "Leandro", "Karl",    "Carlos"  };
```

Esta operación quedará a cargo de la función:

```
void  CrearPares ( TNodoApell  ........,
                   TNodoNomb   ........  );
```

que recibirá como parámetros las direcciones de comienzo de las listas y definirá en su interior los arreglos de pares descriptos más arriba.

Finalmente la función: ***void RecorrerListas(TNodoApell .....);***

mostrará por pantalla los conjuntos de apellidos y nombres recorriendo la lista de apellidos y agregando el nombre correspondiente mediante el puntero adicional a la otra lista.

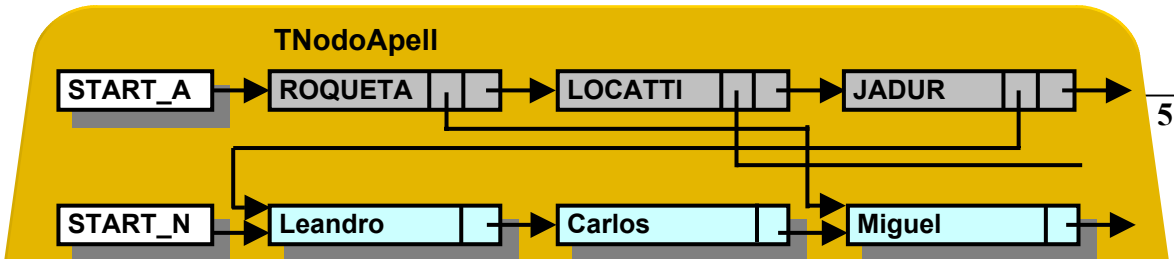
IMPORTANTE:

*En la función **CargarLista( )** observar que :*

```
pAUX_A->Apellido=(char *)malloc(strlen(Apell[i])+1);
pAUX_N->Nombre  =(char *)malloc(strlen(Nomb[i])+1);

strcpy(pAUX_A->Apellido,Apell[i]);
strcpy(pAUX_N->Nombre,Nomb[i]);
```

*los **strcpy( )** no funcionan si no se reserva primero con **malloc( )**. Caso contrario al volver al punto de invocación lo supuestamente asignado se pierde y todo funciona mal.*

Como siempre conviene ***visualizar*** lo que se busca :



Los nodos que almacenan los apellidos contienen dos punteros :

- Uno del mismo tipo ***TNodoApell*** para el próximo Nodo de apellidos.
- Otro de tipo ***TNodoNomb*** para "ligar" con el nombre que le corresponda.

Nuestro tema actual de estudio: los punteros dobles, aparecen en la función :

int GenerarListas (TNodoApell **START_A, TNodoNomb **START_N)

a la cual le pasamos ***POR REFERENCIA*** el puntero de inicio de ambas listas.
También la función :

```
void  CargarListas ( TNodoApell    *START_A,
                     TNodoNomb    *START_N,
                            char   **Apell,
                             char  **Nomb          );
```

dispone de punteros dobles, pero a diferencia de la anterior (y según aclaramos muy bien en un ejemplo anterior) ***NO LO ESTAMOS PASANDO POR REFERENCIA***, sino que se trata de un ***arreglo de punteros a char.***

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

typedef unsigned  int   INT;
typedef unsigned long LONG;

typedef struct TNodoNomb  {
                                  char *Nombre;
                             TNodoNomb *Prox;
                           };

typedef struct TNodoApell    {
                                  char  *Apellido;
                             TNodoApell   *Prox;
                             TNodoNomb  *Nombre;
                           };
const int DIM = 5;

// --- PROTOTIPOS DE FUNCIONES DE USUARIO ----------------------
// .....................................................
void *ReservarMemoria    (  INT Size                        );
```

```
// ..........................................................................................
int   GenerarListas          ( TNodoApell    **START_A,
                               TNodoNomb  **START_N  );
// ..........................................................................................
void  CargarListas           ( TNodoApell    *START_A,
                               TNodoNomb    *START_N,
                                        char  **Apell,
                                        char  **Nomb        );
// ..........................................................................................
void  CrearPares             ( TNodoApell    *START_A,
                               TNodoNomb    *START_N  );
// ..........................................................................................
void  RecorrerListas         ( TNodoApell  *START_A      );

/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  char *Apell[DIM]    = { "ROQUETA","LOCATTI","JADUR","HOFFMANN","LIVETTI" };
  char *Nomb[DIM]  = { "Leandro", "Carlos", "Miguel", "Karl",   "Darío"   };

  TNodoApell    *START_A;    // Comienzo de la lista de Apellidos.
  TNodoNomb  *START_N;    // Comienzo de la lista de Nombres.

  TNodoApell    *pAUX_A;
  TNodoNomb  *pAUX_N;

clrscr( ); highvideo( );

  if(!(GenerarListas(&START_A,&START_N))) {
      cprintf("NO PUDO GENERAR LISTAS EN EL FAR HEAP.\r\n");
      getch( ); exit(1);
  }

  CargarListas(START_A,START_N,Apell,Nomb);
  CrearPares(START_A,START_N);
  RecorrerListas(START_A);

  getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
void *ReservarMemoria(INT Size)
{ return(malloc(Size)); }
/* ------------------------------------------------------------------------------------------------------------ */
int GenerarListas(TNodoApell **START_A, TNodoNomb **START_N)
{
  TNodoApell    *pAUX_A;
  TNodoNomb  *pAUX_N;
   LONG             Size_A   = sizeof(TNodoApell);
   LONG             Size_N   = sizeof(TNodoNomb);
   int               i;


 if((pAUX_A=*START_A = (TNodoApell *   )ReservarMemoria(Size_A))==NULL)  return(0);
  if((pAUX_N=*START_N = (TNodoNomb  *)ReservarMemoria(Size_N))==NULL) return(0);
```

```
  for(i=1;i<DIM;i++) {
     // ..................................................................
     if((pAUX_A->Prox=(TNodoApell *)ReservarMemoria(Size_A))==NULL) return(0);
     pAUX_A=pAUX_A->Prox;  pAUX_A->Prox=NULL;
     // ..................................................................
     if((pAUX_N->Prox=(TNodoNomb *)ReservarMemoria(Size_N))==NULL) return(0);
     pAUX_N=pAUX_N->Prox;  pAUX_N->Prox=NULL;
  }
  return(1);
}
/* ------------------------------------------------------------------------------------------------------------ */
void CargarListas( TNodoApell      *START_A,
                   TNodoNomb     *START_N,
                             char    **Apell,
                             char    **Nomb         )
{
  TNodoApell    *pAUX_A = START_A;
  TNodoNomb  *pAUX_N = START_N;
  int                 i;

  for(i=0;pAUX_A;pAUX_A=pAUX_A->Prox,pAUX_N=pAUX_N->Prox,i++) {
          pAUX_A->Apellido=(char *)malloc(strlen(Apell[i])+1);
          pAUX_N->Nombre  =(char *)malloc(strlen(Nomb[i])+1);

          strcpy(pAUX_A->Apellido,Apell[i]);
          strcpy(pAUX_N->Nombre,Nomb[i]);
  }
}
/* ------------------------------------------------------------------------------------------------------------ */
void CrearPares (TNodoApell *START_A, TNodoNomb *START_N)
{
  char *Apell[DIM]    = { "ROQUETA","LOCATTI","JADUR",  "HOFFMANN","LIVETTI" };
  char *Nomb [DIM] = { "Miguel", "Darío",  "Leandro", "Karl",    "Carlos"  };

  TNodoApell    *pAUX_A;
  TNodoNomb  *pAUX_N;
               int    i;

  for(i=0;i<DIM;i++) {
     pAUX_A=START_A;
     pAUX_N=START_N;

     while(strcmp(pAUX_A->Apellido,Apell[i])    && pAUX_A) pAUX_A=pAUX_A->Prox;
     while(strcmp(pAUX_N->Nombre,Nomb[i])  && pAUX_N) pAUX_N=pAUX_N->Prox;

     pAUX_A->Nombre=pAUX_N;
  }
}
/* ------------------------------------------------------------------------------------------------------------ */



void RecorrerListas(TNodoApell *START_A)
{
  TNodoApell *pAUX = START_A;
```

```
  while(pAUX) {
        cprintf("%s %s\r\n",pAUX->Apellido,pAUX->Nombre->Nombre);
        pAUX=pAUX->Prox;
  }
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Punteros largos (o punteros dobles).

En un par de ocasiones recalcamos que los modelos de memoria a utilizar hasta ese momento podrían ser ***Tinny*** o ***Small***, y que postergábamos para más adelante su explicación. Ahora estamos en condiciones de dar un paso adelante:

Los modelos ***Tinny*** y ***Small*** tenían algo en común:

El ***Tinny*** disponía de un sólo bloque de memoria que englobaba el ***Segmento de Códigos***, el ***Segmento de Datos*** y el ***Stack***, y todo ello en una extensión máxima de ***64 Kb***.

El ***Small*** disponía por separado de un bloque de ***64 Kb*** para el ***Segmento de Códigos*** y otro bloque de ***64 Kb*** para el ***Segmento de Datos*** y el ***Stack***.

Ello significaba lo siguiente:

Para localizar una instrucción en el ***Segmento de Códigos***, los desplazamientos del ***IP*** (Instruction Pointer) no debían exceder nunca de 64 Kb, y para localizar un dato ya sea en el ***Segmento de datos*** o en la ***Pila*** (stack), los desplazamientos tampoco debían exceder de 64 Kb (las direcciones se toman siempre relativas al segmento de que se trate).

Por esta razón es que los punteros (por defecto) en estos modelos de memoria se denominaban punteros ***near*** (o cercanos) y tenían un sizeof( ) de 2 bytes, debido a que con un ***unsigned int*** es suficiente para desplazar hasta 65535 (la dirección del segmento es siempre conocida y sólo interesa el offset o desplazamiento con respecto a él).

Sin embargo la realidad es un poco más extensa que esta magnitud y en modelos de memoria más grandes como el ***Compact*** o el ***Large***, necesitamos desplazarnos más de 64 Kb. Lo malo es que cuando estos problemas se plantearon, el hardware disponible era el chip 8086 que utilizaba un registro de direcciones de 16 bits y de nuevo la misma limitación.

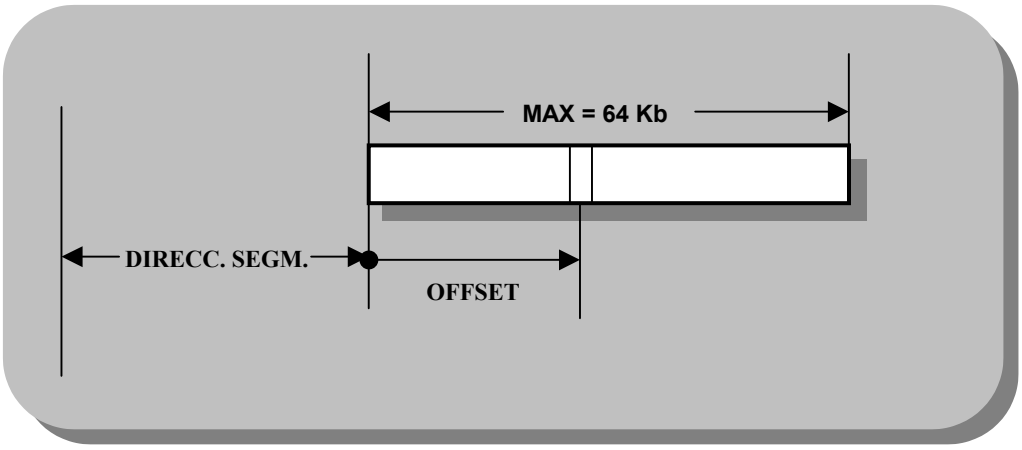
Segmentación de memoria.

El bus de direcciones de los PC históricos contaban con 20 bits, y ello permitía direccionar hasta 1.048.576 posiciones de memoria, en tanto que los registros internos del micro sólo hasta 64 Kb.

Para solucionar este inconveniente y seguir trabajando con registros de 16 bits (2 bytes), surge la idea de segmentar la memoria y direccionar cualquier byte utilizando dos elementos:

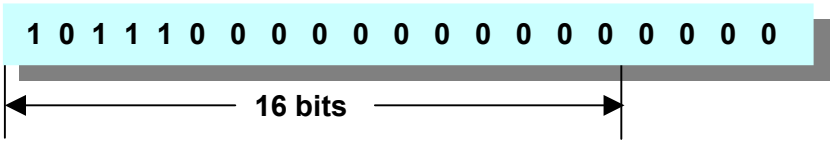
- Un segmento (de tamaño máximo 64 Kb).
- Un offset (tamaño máximo de 64 Kb) o desplazamiento dentro de dicho segmento.

Esquemáticamente sería lo siguiente:

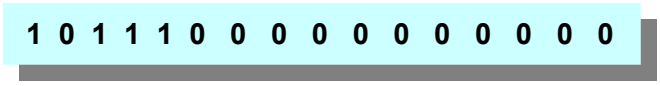


Sin embargo aún seguimos con problemas, porque para cubrir todas las direcciones segmento necesitamos 5 dígitos hexadecimales (20 bits), y nuestros registros siguen siendo de 16 bits. Para solucionar esto el sistema operativo divide la ***dirección del segmento*** por 16 (0x10), lo que equivale a desplazar ***4 bits*** a la derecha.

Por ejemplo si tomamos como referencia para la ***dirección del segmento***, el comienzo de la memoria de video, el mismo se halla ubicado en 0xB8000 = 753664



al dividir por 16 obtenemos : 47104



El resultado de dividir la dirección segmento por ***0x10*** se denomina ***valor segmento***.

Expresando matemáticamente lo dicho :

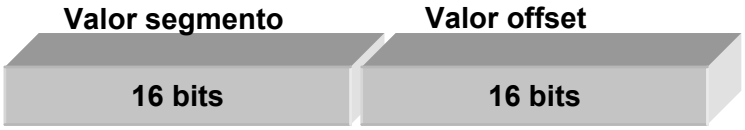
dirección segmento = valor segmento * 0x10

Esto nos indica claramente que los segmentos sólo pueden arrancar en direcciones que sean múltiplos de 16.

Lo mencionado hasta ahora señala el ***comienzo*** del segmento (que como máximo puede tener 64 Kb) y ahora falta agregar el ***desplazamiento*** con que el dato se encuentra ubicado con respecto a este origen. Este desplazamiento puede ser como máximo del tamaño del segmento (64 Kb) lo cual puede ser manejado por un registro de direccionamiento de 16 bits. Con esto llegamos a la ***dirección absoluta*** de un byte dado:

dirección absoluta = valor segmento * 0x10 + Offset

que esquemáticamente podríamos dibujar como:



o sea que para expresar una dirección segmentada necesitamos ***32 bits***. Esto es lo que se denomina un ***puntero far*** (lejano) o ***puntero largo***.

Si bien la modalidad ***far*** es tomada por defecto en los modelos grandes de memoria (sin necesidad de ninguna declarativa en especial), también podemos utilizar punteros largos en modelos pequeños agregando la palabra clave ***far*** :

El siguiente programa ilustra esta última aseveración y lo dicho anteriormente sobre direccionar una misma posición de memoria tomando diferentes segmentos y offsets:

*Variando el Segmento y el Offset de un puntero byte far * apuntado al 50 avo. caracter de la fila 1 de la memoria de video, demostrar que con 5 magnitudes diferentes se puede direccionar* ***la misma*** *posición de memoria.*

```
#include <conio.h>

typedef unsigned char byte;
```

```
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  byte far *pSCR[ ] = { (byte *)0xB8000050,
                        (byte *)0xB8010040,
                        (byte *)0xB8020030,
                        (byte *)0xB8030020,
                        (byte *)0xB8040010,
                        (byte *)0xB7FF0060 };
  char Car;

  clrscr( ); highvideo( );

  for(Car='A';Car<='F';Car++) { *pSCR[Car-'A']=Car; getch( ); }
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Aprovechamos este programa para indicar varias cosas:

Cómo asignar una ***dirección larga*** en forma inmediata:

1. La forma más sencilla y obvia es escribir los 8 dígitos hexadecimales de los cuales los 4 primeros corresponden al ***valor segmento*** y los 4 restantes al ***offset***.

Así por ejemplo: ***0xB8000050*** significa ***valor segmento*** = ***0xB800*** y ***offset = 0x50***

NOTA:
Recordar en todo momento que los dígitos son hexadecimales y que el offset no es 50 decimal, sino 50 hexa (80 decimal).

2. Otra modalidad, considerada más portable, es mediante la expresión :

(unsigned char *)(0xB8000L << 16) ¦ 0x50

donde la "**L**" significa que el compilador utilice un buffer transitorio de 4 bytes (long) para almacenar el valor segmento. Luego se ubica el valor segmento en los 2 bytes de mayor peso (moviéndolo 16 lugares a la izquierda), y a continuación se agrega el off-set en los 2 bytes de menor peso. Es un equivalente de la primera notación porque en definitiva hemos arribado a una magnitud de 8 dígitos hexadecimales.

Ahora vamos a demostrar que las 6 direcciones del vector de punteros señalan el ***mismo punto*** en el mapa de memoria:

```
byte far *pSCR[ ] = {  (byte *)0xB8000050,
                       (byte *)0xB8010040,
                       (byte *)0xB8020030,
                       (byte *)0xB8030020,
                       (byte *)0xB8040010,
                       (byte *)0xB7FF0060   };
```

Haciendo: ***dirección absoluta*** = ***valor segmento * 0x10 + Offset***, tendremos:

0xB8000050 = 0xB800 * 0x10 + 0x50 = 0xB8000 + 0x50 = 0xB8050
0xB8010040 = 0xB801 * 0x10 + 0x40 = 0xB8010 + 0x40 = 0xB8050
0xB8030030 = 0xB803 * 0x10 + 0x30 = 0xB8020 + 0x30 = 0xB8050
.....
0xB7FF0060 = 0xB7FF * 0x10 + 0x60 = 0xB7FF0 + 0x60 = 0xB8050

si esta última operación no ha quedado lo suficientemente clara, podemos descomponerla de la siguiente manera :

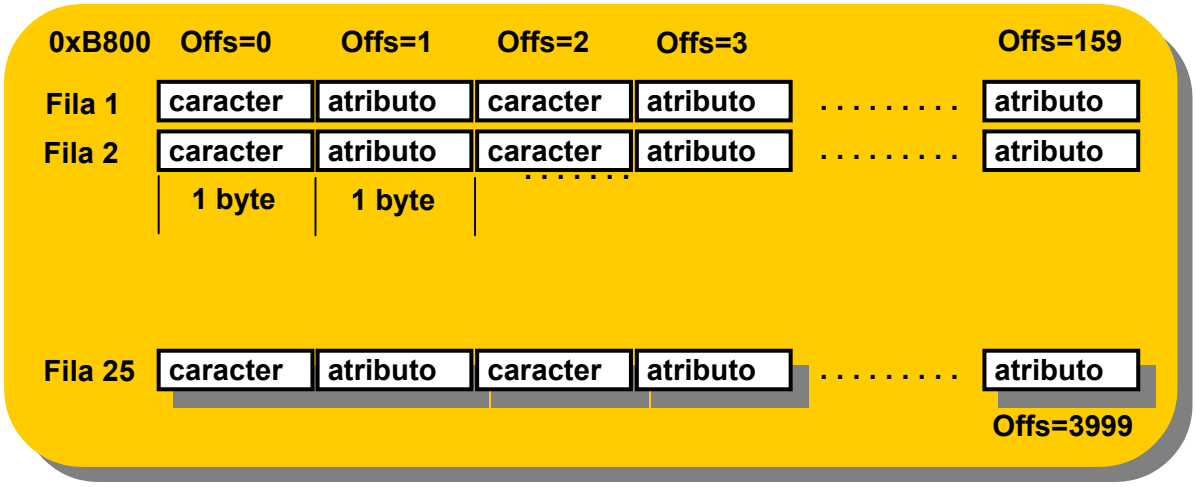
0xB7FF0 + 0x60 = (0xB7FF0 + 0x10) + 0x50 = 0xB8000 + 0x50 = 0xB8050

Finalmente falta agregar que el programa actual muestra sobre la misma posición de pantalla una secuencia de 6 caracteres que van de la 'A' a la 'Z'. Cómo es esto posible es motivo del próximo apartado.

Memoria de video en modo texto.

El PC, en modo texto, maneja la salida a pantalla inspeccionando permanentemente el contenido de una zona especial de la memoria (que depende de la tarjeta controladora) y volcando su contenido 50 veces por segundo hacia el monitor. En las tarjetas VGA esa dirección se halla a partir de 0xB800 y comprende los próximos 4000 bytes consecutivos.

La información de video se halla dispuesta de la siguiente manera :

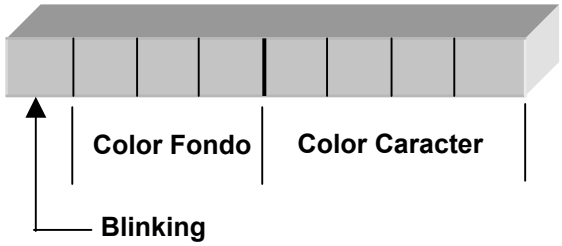


En realidad se trata de posiciones consecutivas de memoria y es un vector, no una matriz, pero estamos más familiarizado con esta última por eso lo hemos dispuesto de esta manera. Lo importante es que se van alternando: 1 byte de carácter - 1 byte de atributo, 1 byte de carácter - 1 byte de atributo y así sucesivamente hasta completar los 4000 elementos de una pantalla completa (80 columnas x 25 filas x 2 = 4000).

Se nos ocurre rápidamente una simple fórmula para hallar el Offset correspondiente a cualquier coordenada (Columna, Fila) de pantalla:

Offset = (Fila-1)*160 + (Columna-1)*2 (asumiendo que Fila y Columna arranquen de 1)

En cuanto al byte de atributo su constitución es la siguiente:



La forma más práctica de asignar un valor al byte de atributos es:

Atributo = COLOR DE FONDO << 4 | COLOR DEL CARACTER

Si observamos la zona del color de fondo veremos que sólo disponemos de 3 bits, lo cual implica que únicamente podremos asignarle los 7 primeros colores de la tabla. En cambio para el color del carácter disponemos de 4 bits y los colores permitidos son los 16 de la tabla.

He aquí un ejemplo interesante :

Generar una función que imprima mensajes directamente a través de la memoria de video.

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

typedef unsigned char byte;

void Print  ( char *Frase, int Col, int Fila, byte ATTR );
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  char     *Frase = "BORLANDC ES VELOZ.";
  byte      ATTR  = BLACK << 4 | LIGHTGREEN;
  byte      i;

  clrscr( ); highvideo( );

  Print(Frase,25,10,ATTR);
  getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
void Print(char *Frase,int Col,int Fila, byte ATTR)
{
  byte far *pSCR  = (byte *)0xB8000000;
  byte i;



  for(i=0;i<strlen(Frase);i++) {
```

```
        *(pSCR+(Fila-1)*160+(Col+i-1)*2)    =Frase[i];
        *(pSCR+(Fila-1)*160+(Col+i-1)*2+1)=ATTR;
  }
}
/* ---------------------------------------------------------------------------------------------------------- */
```

Aquí hemos desarrollado una función de usuario denominada ***Print*** que recibe como parámetros la frase a imprimir, las coordenadas de pantalla y el atributo.

La dirección de inicio de la memoria de video queda fijada en:

byte far *pSCR = (byte *)0xB8000000;

y a partir de allí y según vimos anteriormente sólo nos movemos con el offset correspondiente:

```
for(i=0;i<strlen(Frase);i++) {
        *(pSCR+(Fila-1)*160+(Col+i-1)*2)    =Frase[i];
        *(pSCR+(Fila-1)*160+(Col+i-1)*2+1)=ATTR;
}
```

Resulta más sencillo trabajar de esta manera, considerando por separado el byte de carácter y el de atributo. Sin embargo también es posible trabajar con un entero y armarlo colocando en cada byte el carácter y su atributo.

Lo bueno que tiene trabajar sobre la memoria de video es que los resultado se aprecian rápidamente... y resulta divertido!. Por ejemplo el código que viene a continuación hace invisible la pantalla y luego la restituye a la normalidad:

```
#include <conio.h>

typedef unsigned char byte;

void clrSCR   ( void );
/* ---------------------------------------------------------------------------------------------------------- */
void main( )
{
  int i;

  clrscr( ); textcolor(LIGHTGREEN);

  for(i=0;i<24;i++) cprintf("BORLANDC ES MUY VELOZ.\r\n");  getch( );
  clrSCR( );
  getch( );
}
/* ---------------------------------------------------------------------------------------------------------- */




/* ---------------------------------------------------------------------------------------------------------- */
void clrSCR( )
{
```

```
  byte       BuffAttr[2000];
  byte far *pSCR = (byte far *)0xB8000001;
  byte       HIDE =  BLACK << 4 | BLACK;
  int        i,j;

  // --- BORRA LA PANTALLA ---------------------------------------------------------
  for(i=0,j=0;i<4000;i+=2) BuffAttr[j++]=*(pSCR+i);
  for(pSCR=(byte far *)0xB8000001,i=0;i<2000;i++) { *pSCR=HIDE; pSCR+=2; }

  getch( );

  // --- RESTITUYE LA PANTALLA ANTERIOR --------------------------------------------
  for(pSCR=(byte far *)0xB8000001,i=0,j=0;i<4000;i+=2)  *(pSCR+i)=BuffAttr[j++];
}
/* ------------------------------------------------------------------------------------------------------ */
```

La técnica de ocultar la pantalla (a los fines de la práctica con punteros), consiste en cambiar el atributo de fondo por carácter negro sobre fondo negro. Esta combinación hace todo invisible. Sin embargo, antes de aplicar esta técnica debemos memorizar cuáles eran los atributos originales para restablecerlos al hacer visible nuevamente la pantalla.

Ahora bien, como los atributos pueden ser distintos en diferentes zonas de pantalla, lo más saludable es rescatarlos a todos en un arreglo de byte y recién proceder al oscurecimiento. Este rescate se lleva a cabo en:

for(i=0,j=0;i<4000;i+=2) BuffAttr[j++]=*(pSCR+i);

y el oscurecimiento en :

for(pSCR=(byte far *)0xB8000001,i=0;i<2000;i++) { *pSCR=HIDE; pSCR+=2; }

Nótese que siempre arrancamos de la posición ***0xB8000001*** y variamos el offset de 2 en 2 (puesto que allí se hallan los bytes de atributo).

La operación inversa o complementaria restituye los atributos originales:

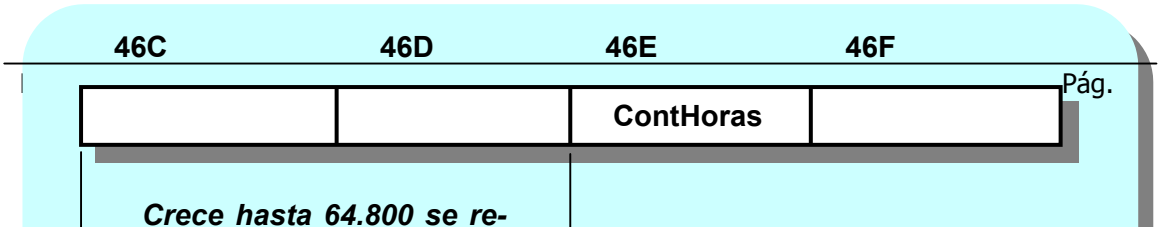
for(pSCR=(byte far *)0xB8000001,i=0,j=0;i<4000;i+=2) *(pSCR+i)=BuffAttr[j++];

Posiciones útiles en la parte baja de memoria.

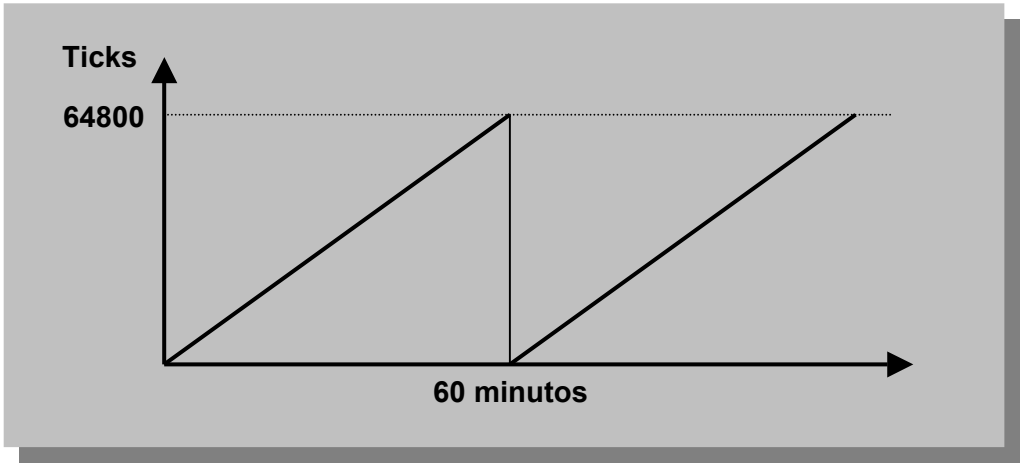
El sistema operativo DOS se reserva ciertas posiciones bajas de memoria para su trabajo propio, como por ejemplo la posición ***046C*** que acumula los ticks del reloj (tomando para ello 2 bytes : ***046C*** - ***046D***, y la posición ***046E*** mantiene la cuenta de horas transcurridas (todo ello desde las 0 hs. del día en curso).

Generar un programa que lea permanentemente dichas posiciones y construya un reloj digital.

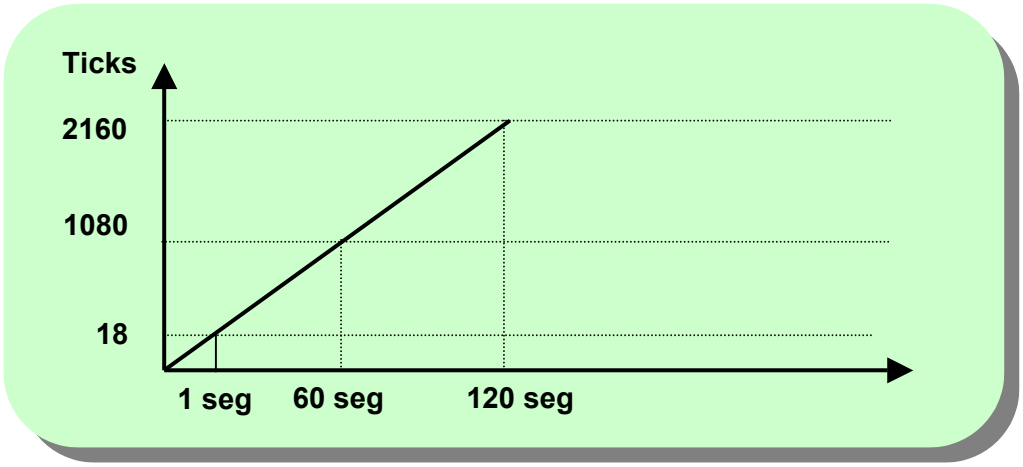
Al parecer esto funciona de la siguiente manera:



Podríamos simbolizar el comportamiento de estos bytes de la siguiente manera:



Los ticks del reloj se generan a razón de 18 pulsos por segundo, de ahí que en 60 minutos tendremos : 60 x 60 x 18 = 64800 ticks (allí se incrementan las horas y el contador de ticks comienza de cero).
A nivel segundos podríamos tener un esquema parecido:



Por lo tanto el puntero de contador de ticks lo cargamos en la posición 46C y el de contador de horas en 46E.

INT far *pContTck = (INT far *)0x0000046C; // Contador de ticks.
INT far *pContHr = (byte far *)0x0000046E; // Contador de horas.

El código sería el siguiente:

```
#include <conio.h>

typedef unsigned char byte;
typedef unsigned long LONG;
typedef unsigned int    INT;
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  INT   far *pContTck = (INT far *) 0x0000046C;    // Contador de ticks.
  INT   far *pContHr   = (INT far *) 0x0000046E;     // Contador de horas.

  clrscr( ); highvideo( ); _setcursortype(_NOCURSOR);

  do {
      gotoxy(38,14);
      cprintf("%02d:%02d:%02d\r\n",
              *pContHr,(*pContTck/1080)%60,(*pContTck/18)%60);
    } while(!kbhit( ));

  _setcursortype(_NORMALCURSOR);
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Obsérvese que una vez apuntados los punteros, lo que ellos referencian se actualizan automáticamente (el hardware se encarga de esta tarea). Esto hace posible que el lazo:

```
do {
      gotoxy(38,14);
      cprintf("%02d:%02d:%02d\r\n", ContHr,(*pContTck/1080)%60,
                                                  (*pContTck/18)%60));
    } while(!kbhit( ));
```

sea exitoso. Las expresiones tienen que ver con lo siguiente: Los ticks del reloj son producidos 18 veces por segundo y almacenados en memoria. Ello significa que por minuto se generan 60 x 18 = 1080 ticks, por lo tanto la expresión:

(*pContTck/1080)%60

se va incrementando de 1 en 1 cada 1080 ticks de reloj y cuando este incremento sea un múltiplo de 60, se pondrá en cero.

Con los segundos pasa algo similar : se incrementa de 1 en 1 cada 18 ticks de reloj y cuando el incremento sea múltiplo de 60 se pone en cero y recomienza.

Reservas en el heap lejano (far heap)

Cuando las reservas dinámicas de memoria deben exceder los 64 Kb los modelos Tinny y Small quedan chicos y por más que definamos punteros largos no podremos reservar mucha memoria si no utilizamos alguno de los modelos grandes.
Los emplazamientos que originalmente hacíamos con ***malloc( )*** ahora serán reemplazados por ***farmalloc( )***.

He aquí un programa de ejemplo :

*Trabajando con el **far heap**, realizar una reserva de **160000 bytes**, cargarla con valores aleatorios entre 20 y 100 y luego mostrar por pantalla que todo haya funcionado bien. Finalmente liberar la reserva.*

IMPORTANTE:
*Los modelos pequeños de memoria no trabajan correctamente con el **far heap**. Al comienzo dan la sensación que corren perfectamente, pero luego se quedan colgados. En consecuencia debe utilizarse el modelo **COMPACT** para compilar este programa.*

NOTA:
*En el modelo **COMPACT** los punteros de datos son far por defecto, pero por rigurosidad hemos explicitado que sean **far**.*

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


typedef unsigned long LONG;
typedef unsigned char byte;

const LONG DIM = 160000;

/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  byte far *pFAR_HEAP;
  LONG    i;

  clrscr( ); highvideo( );

  if((pFAR_HEAP=(byte far *)farmalloc(DIM))==NULL) {
      cprintf("NO PUDO RESERVAR EN EL FAR HEAP.\r\n");
      getch( ); exit(1);
  }



  randomize( );

  for(i=0;i<DIM;i++) pFAR_HEAP[i]=20+random(80);
```

```
  for(i=0;i<DIM;i++) cprintf("%d  ",pFAR_HEAP[i]);

  getch( ); free((void *)pFAR_HEAP);
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Como siempre las operaciones de asignaciones dinámicas se chequean de la forma tradicional:

```
if((pFAR_HEAP=(byte far *)farmalloc(DIM))==NULL)  {
      cprintf("NO PUDO RESERVAR EN EL FAR HEAP.\r\n");
      getch( ); exit(1);
}
```

cuando ya no necesitemos el bloque reservado, podemos liberarlo mediante :

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

Punteros "huge".

Cuando analizamos los punteros largos (far) veíamos que el desplazamiento (offset) con respecto al segmento podría ser como máximo 64 Kb y si intentamos pasarnos de esta medida vuelve a arrancar de offset cero. Esto puede resultar un verdadero problema cuando necesitamos desplazarnos grandes distancias con respecto a un segmento dado. Sin embargo "C" nos facilita este inconveniente mediante los punteros ***"huge"***, que si bien son también punteros largos, tienen la inestimable ventaja que podemos variar su offset más allá de los 64 Kb sin ningún problema. Es algo así como trabajar directamente con la ***dirección absoluta***.

La declarativa formal de un puntero huge es:

<tipo> huge nombre;

el compilador sabe perfectamente que se trata de un puntero far.

Veamos un problema interesante :

El siguiente programa hará una reserva de 60 bytes para almacenar la frase "TALLER DE LENGUAJES I". Luego mediante un puntero ***"huge"*** *realizará la búsqueda de esta frase en toda la memoria RAM y la pasará a minúsculas. Para verificar si la operación fue correcta, se mostrará por pantalla el contenido de la variable original.*

```
#include <conio.h>
#include <string.h>
#include <alloc.h>
#include <ctype.h>

typedef unsigned long LONG;
typedef enum boolean { NO, SI };


const LONG POSMAX = (LONG)700000;
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
```

```
{
  char         *Frase    = (char *)malloc(60);
  char huge *pCAR    = (char huge *)0x00000000;
  boolean      SeHalla = NO;
  LONG        i;
  int             j;

  clrscr( ); highvideo( );
  strcpy(Frase,"TALLER DE LENGUAJES I");
  cprintf("FRASE ORIGINAL : %s\r\n",Frase);

  for(i=0;i<POSMAX;i++,pCAR++) {
     if(*pCAR==Frase[0]) {
           SeHalla=SI;
           for(j=0;j<strlen(Frase);j++) if(*(pCAR+j)!=Frase[j]) SeHalla=NO;

     // --- PASA LA FRASE A MINUSCULAS ----------------------------
           if(SeHalla) {
               for(j=0;j<strlen(Frase);j++) *(pCAR+j)=tolower(*(pCAR+j));
               pCAR+=strlen(Frase);
           }
     }
  }

  cprintf("FRASE MODIFICADA : %s\r\n",Frase);
  getch( );
}
/* ------------------------------------------------------------------------------------------------------ */
```

El puntero : ***char huge *pCAR = (char huge *)0x00000000;***
será el encargado de barrer la memoria RAM hasta hallar la frase almacenada. Nótese que el offset se incrementa hasta un valor: ***const LONG POSMAX = (LONG)700000;*** y en ningún momento se presentarán problemas de segmento.

La rutina de búsqueda es simple : va desplazándose el puntero hasta que lo referenciado por él coincida con el primer carácter de la cadena buscada, a partir de allí chequea el resto de la cadena y si efectivamente la encontró la pasa a minúsculas.

Almacenamiento múltiple de pantallas.

Esta puede ser la base para manejo de helps de pantalla :
En el main( ), declarar el siguiente bloque de cadenas :

```
char *Texto[7] = {      "BorlandC++ Builder es el nuevo producto de RAD",
                        "(Desarrollo Rápido de Aplicaciones) de Borland",
                        "Con C++ Builder es posible escribir programas",
                        "para Windows de manera más rápida y sencilla",
                        "que nunca. Puede crear aplicaciones de consola",
                        "o programas de GUI ( Interfase Gráfica de Usua-",
                        "rio) para Win32.                                " };
```

donde los textos son TODOS de la misma longitud (46 caracteres).

Haciendo uso de esta función y las cadenas anteriores, procederá de la siguiente manera:

```
void MotivoDePantalla( int Offs, char **Mat);
```

Definirá la función:

que invocará desde el main( ) con valores crecientes de Offs desde 0 hasta el largo de cada cadena del arreglo :

```
for(i=0;i<strlen(Texto[0]+1;i++) {
        clrscr( ); Box(15,8,16+strlen(Texto[0])+2,16);
        MotivoDePantalla(i,Texto);
        GuardarEnMemoriaReservada
}
```

Donde ***MotivoDePantalla( )*** escribirá el texto de las cadenas a partir del mismo lugar inicial (extremo izquierdo del box), pero partiendo siempre de un carácter corrido a la derecha.

Esto irá generando pantallas con texto dentro del ***Box( )***, (que siempre se hallará emplazado en el mismo lugar de la pantalla), pero en donde el texto irá corrido un espacio hacia la izquierda:

```
BorlandC++ Builder es el nuevo producto de RAD
(Desarrollo Rápido de Aplicaciones) de Borland
Con C++ Builder es posible escribir programas
para Windows de manera más rápida y sencilla
que nunca. Puede crear aplicaciones de consola
o programas de GUI ( Interfase Gráfica de Usua-
rio) para Win32.
```

```
"orlandC++ Builder es el nuevo producto de RAD
"Desarrollo Rápido de Aplicaciones) de Borland
"on C++ Builder es posible escribir programas
"ara Windows de manera más rápida y sencilla
"ue nunca. Puede crear aplicaciones de consola
" programas de GUI ( Interfase Gráfica de Usua-
"io) para Win32.
```

```
rlandC++ Builder es el nuevo producto de RAD
esarrollo Rápido de Aplicaciones) de Borland
n C++ Builder es posible escribir programas
ra Windows de manera más rápida y sencilla
e nunca. Puede crear aplicaciones de consola
 programas de GUI ( Interfase Gráfica de Usua-
o) para Win32.
```

etc. hasta desplazarlo por completo hacia el margen izquierdo.

Cada pantalla generada es "capturada" en la heap lejano utilizando la función _fmemcpy( ) de la librería <string.h> (inspecciónela y averigüe como emplearla).

Obviamente la primera tarea será **RESERVAR** la memoria suficiente ***PARA TODAS*** las pantallas en el far heap (***strlen(Texto[0])*4000***). Cierto puntero que Ud. definirá en el main( ), memorizará la **dirección de inicio** de este almacenamiento, y a partir de allí comenzará guardando la primera pantalla generada. Luego mediante un desplazamiento de 4000 bytes se almacenará la segunda, y así sucesivamente.

IMPORTANTE :

Esto quiere decir que el desplazamiento de almacenamiento sobrepasará en algún momento los 64 Kb, lo cual sugiere la utilización de un **puntero "huge"** para que no tenga problemas por cambios de segmento.

La reserva de memoria estará a cargo de la función:

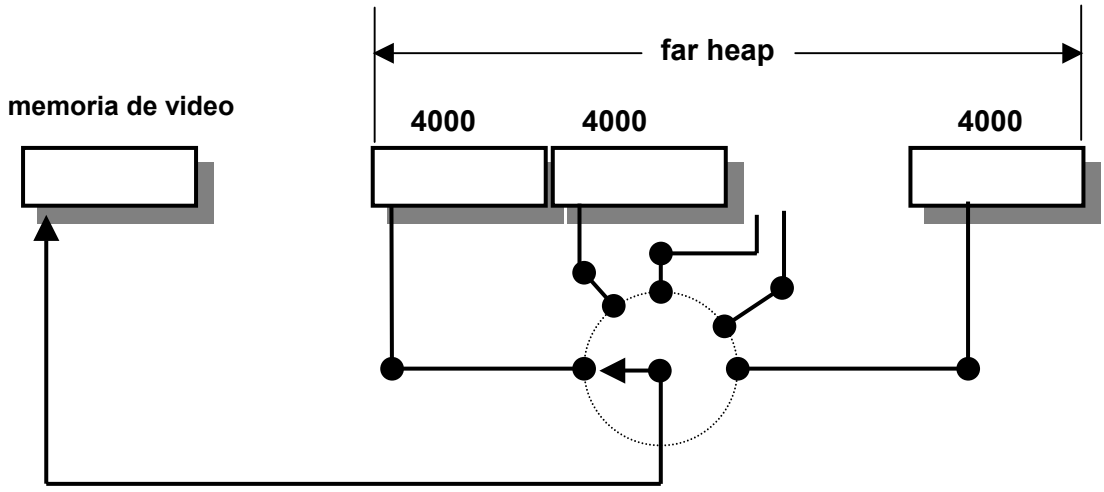
```
byte huge *ReservarMemoria(unsigned long Size)
```

que devolverá la dirección de comienzo del bloque en el far heap como un puntero "huge" a byte.

Una vez generadas y almacenadas las ***46 pantallas***, procederá a volcarlas en forma cíclica desde el far heap a la memoria de video, hasta tanto se pulse la tecla ESC. Esto creará una ilusión de desplazamiento del texto dentro del box.

Para verlo mejor consideremos la siguiente secuencia de operaciones :

- Reservar memoria dinámica en el ***far heap***.
- Generar cada pantalla e ir almacenándola a partir del inicio de la reserva.
- Una vez almacenada la última pantalla, reproducir la secuencia volcándolas una por una, con una pausa intermedia (***delay(DEMORA***)), desde el far heap al monitor :



El código sería el siguiente:

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

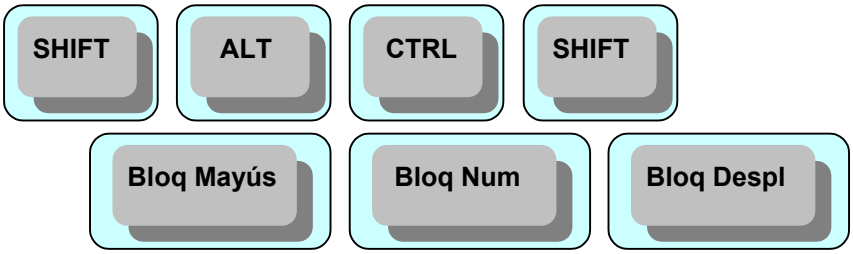
```
#include <alloc.h>
#include <process.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <string.h>

typedef unsigned long LONG;
typedef unsigned char byte;

        const int ANCHO    = 20;
        const int ALTO     = 12;
        const int DEMORA   = 120;
        const int ESC      = 27;

byte huge *ReservarHeap     ( LONG Size                    );
void       MotivoDePantalla ( int Offs, char **Mat         );
void       Box              ( int X1, int Y1, int X3, int Y3 );

/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  byte huge *pHEAP;
  byte far   *pSCREEN = (byte far *)0xB8000000;  // MEMORIA DE VIDEO VGA
  long        i;                 // Se toma long porque crear  un offset de más
  long        Size;              // de 64 Kb.

  char *Texto[7] = { "BorlandC++ Builder es el nuevo producto de RAD",
                     "(Desarrollo R pido de Aplicadiones) de Borland",
                     "Con C++ Builder es  posible escribir programas",
                     "para Windows de  manera m s r pida  y sencilla",
                     " que nunca. Puede crear aplicaciones de consola",
                     "o programs de GUI ( Interfase Gr fica de Usua-",
                     "rio) para Win32.                          "   };
  char Tecla;

  clrscr( ); highvideo( );

 // --- RESERVA EN EL  far heap PARA LAS 46 PANTALLAS -------

 Size=(long)4000*(long)(strlen(Texto[0])+1);
 if((pHEAP=ReservarHeap(Size))==NULL) {
     cprintf("NO PUDO RESERVAR EN EL FAR HEAP.\r\n");
     getch( ); exit(1);
  }

  // --- CARGA LAS PANTALLAS EN MEMORIA ------------------------

  for(i=0;i<strlen(Texto[0])+1;i++) {
     clrscr( ); Box(15,8,16+strlen(Texto[0])+2,16);

     MotivoDePantalla(i,Texto);
    _fmemcpy(pHEAP+i*4000L,pSCREEN,4000);
  }
  clrscr( ); cprintf("COMENZARA A LEER DEL HEAP..."); getch();
```

```
  do { for(i=0;i<strlen(Texto[0])+1;i++) {
               _fmemcpy(pSCREEN,pHEAP+i*4000L,4000); delay(DEMORA);
                if(kbhit()) { Tecla=getch(); break; }
       }
     } while(Tecla!=ESC);

  free((void *)pSCREEN); clrscr( );
}
/* ---------------------------------------------------------------------------------------------------------- */
byte huge *ReservarHeap(LONG Size)
{
  static byte huge *pLOCAL;

  if((pLOCAL=(byte huge *)farmalloc(Size))==NULL) return(NULL);
  return(pLOCAL);
}
/* ---------------------------------------------------------------------------------------------------------- */
void Box(int X1, int Y1, int X3, int Y3)
{
  int Col,Fila;

  gotoxy(X1,Y1); cprintf("Ú");
  for(Col=X1+1;Col<X3;Col++) cprintf("Ä");
  gotoxy(X3,Y1); cprintf("¿");
  for(Fila=Y1+1;Fila<Y3;Fila++) {
     gotoxy(X1,Fila); cprintf("³");
     gotoxy(X3,Fila); cprintf("³");
  }
  gotoxy(X1,Y3); cprintf("À");
  for(Col=X1+1;Col<X3;Col++) cprintf("Ä");
  gotoxy(X3,Y3); cprintf("Ù");
}
/* ---------------------------------------------------------------------------------------------------------- */
void MotivoDePantalla(int Offs, char **Mat)
{
  int      i;
  int      Col  = 17;
  int      Fila = 9;
  char *pTxt;

  for(i=0;i<7;i++) {
          gotoxy(Col,Fila+i);
          pTxt=&Mat[i][Offs];
         cprintf("%s",pTxt);
  }
}
/* ---------------------------------------------------------------------------------------------------------- */
```

De nuevo a las posiciones bajas de memoria.

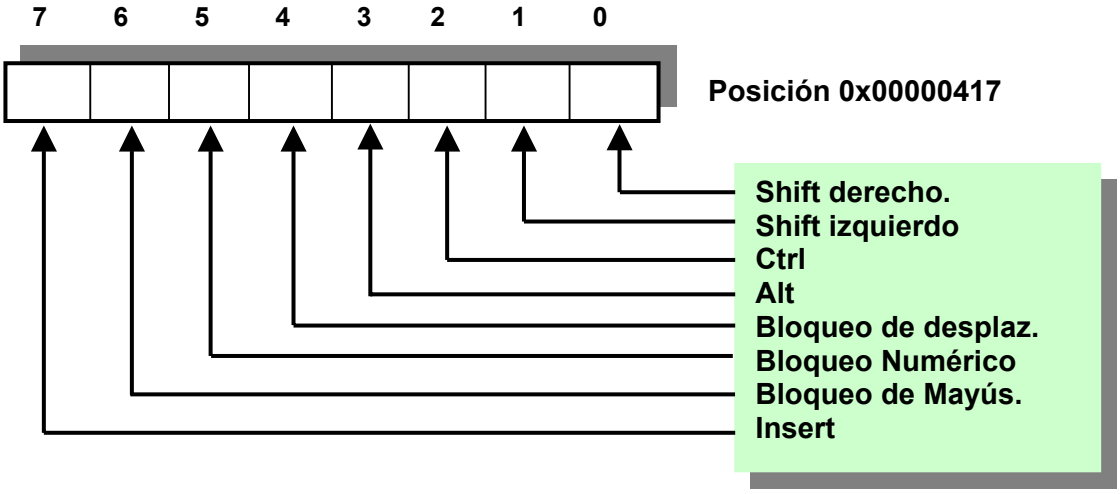
Teclas de cambio transitorio.

Existe un conjunto de 7 teclas especiales cuya detección no puede realizarse a través de la tabla de valores ***ASCII***. Ellas son :



si bien estas tres últimas se denominan teclas de ***cambio permanente***.

Para averiguar si estas teclas han sido pulsadas, el sistema operativo toma en cuenta la posición ***417 hexa*** de memoria baja según el siguiente esquema :



Con nuestros conocimientos actuales de punteros, percibimos en el acto que un puntero byte far direccionado en 0x00000417 permitirá "extraer" la información de estado, en forma permanente.

He aquí el código del programa:

```
#include <conio.h>

typedef unsigned char byte;
enum { HTDER=1,SHTIZQ=2,CTRL=4,ALT=8,SCRLCK=16,NUMLCK=32,CPSLCK=64 };
const int DIM = 7;
/* ------------------------------------------------------------------------------------------------------------ */



void main( )
{
```

```
  byte far *pByte417     =  (byte far *)0x00000417;
  byte  Estados[DIM]    = { SHTDER,SHTIZQ,CTRL,ALT,SCRLCK,NUMLCK,CPSLCK };

  char *EstadosStr[DIM] = { "SHIFT DERECHO",
                            "SHIFT IZQUIERDO",
                            "CTRL",
                            "ALT",
                            "SCROLL LOCK",
                            "NUM LOCK",
                            "CAPS LOCK"             };
  byte  ESTADO;
  byte  Col          =  20;
  byte  Fila         =  5;
  byte  i;
  byte Tecla;

  clrscr( ); highvideo( ); _setcursortype(_NOCURSOR);

  for(i=0;i<DIM;i++) { gotoxy(Col,Fila+i); cprintf("%s",EstadosStr[i]); }
  textcolor(LIGHTGREEN);
  gotoxy(Col,14); cprintf("PULSANDO CUALQUIER TECLA FINALIZA [...]");
  do {
        ESTADO=*pByte417;
        for(i=0;i<DIM;i++) {
              Tecla=Estados[i]; gotoxy(40,Fila+i);
              if(ESTADO & Tecla) { textcolor(LIGHTRED); cprintf("ON "); }
              else { textcolor(YELLOW); cprintf("OFF"); }
        }
    } while(!kbhit( ));
  _setcursortype(_NORMALCURSOR);
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Mientras no se haya pulsado ninguna tecla que no sean las especiales de nuestro estudio, el programa chequeará indefinidamente mediante un lazo for, la posición 417.
Para saber el estado de las teclas de cambio, hemos recurrido a un ***enmascaramiento*** con el operador ***and*** bit a bit. Una forma más elegante hubiera sido mediante un ***campo de bits*** (dejamos esta práctica a cargo del lector).

Nótense bien los recursos programáticos utilizados a fin de darle más consistencia al código. Por ejemplo el enumerativo que luego utilizamos para cargar el arreglo de valores de estados.

NOTA
Algunos de los nombres de las teclas han sido puestos para teclado inglés, cámbielos a español a modo de práctica.

Punteros a funciones.

Para entender cómo funcionan los punteros a funciones, debemos entender cómo se compila y se llama a una función en "C". Cuando se compila cada función el código fuente se convierte en código objeto y se establece un punto de entrada. Cuando se llama a una función durante la ejecución del programa, se hace una llamada en lenguaje máquina a ese punto de entrada. Por lo tanto si el puntero contiene la dirección de entrada de la función, puede utilizarse para ***invocar*** a la función.

La declaración formal de un puntero a función se realiza según:

<tipo> (*nombre_de_la_función)(argumentos);

y no existe ningún problema para declarar tipos de punteros a funciones.

Entre las potencialidades de los punteros a funciones podemos considerar el poder pasar ***funciones como parámetros*** de otras funciones, ***redireccionamiento*** de interrupciones a nivel de sistema operativo.

Ejemplo 1

*Una función llamada **GenerarTabla( )** recibirá como parámetros dos punteros, uno cargado con la dirección de una función que determina valores de senos utilizando la **Serie de Tylor** y el otro con otra función denominada **SenoHyp** que calcula senos hyperbólicos. Esta función generará una tabla de 10 valores.*

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

typedef double (*TpFn)(double Ang);    // Prototipo del puntero a función.

double Seno           ( double Ang   );
double SenoHyp      ( double Ang   );
void   GenerarTabla ( TpFn Funcion1, TpFn Funcion2 );
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
   TpFn Funcion1 = Seno;
   TpFn Funcion2 = SenoHyp;

   clrscr( ); highvideo( );

   GenerarTabla(Funcion1,Funcion2);
   getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
void GenerarTabla(TpFn Funcion1,TpFn Funcion2)
{
  int Ang;

  textcolor(LIGHTGREEN);
  cprintf("SENO(X)         SENOHYP(X)\r\n");


  textcolor(WHITE);
```

```
  for(Ang=0;Ang<=10;Ang+=1) {
      cprintf("%6.4lf",Funcion1((double)Ang));
      cprintf("%20.4lf\r\n",Funcion2((double)Ang));
  }
}
/* ------------------------------------------------------------------------------------------------------------ */
double Seno(double Ang)
{
  const double Sgn = -1;
  const double Eps = 0.0001;

  double Seno;
  double Term;
  double x;
  long     N;    // Orden del término actual.

  x=M_PI*Ang/180.0;
  for(Seno=Term=x,N=3;Eps<fabs((Term*=((x*x)/(N*(N-1))))*=Sgn);
      Seno+=Term,N+=2);

  return(Seno);
}
/* ------------------------------------------------------------------------------------------------------------ */
double SenoHyp(double Ang)
{ return(0.5*(exp(Ang)-exp(-Ang))); }
/* ------------------------------------------------------------------------------------------------------------ */
```

Vemos que análogamente a lo que ocurría con los arreglos, para asignar la dirección de una función a un ***puntero compatible*** con ella, sólo basta hacer:

TpFn Funcion1 = Seno;
TpFn Funcion2 = SenoHyp;

en tanto que la invocación de la función que los recibirá como parámetros es :

GenerarTabla(Funcion1,Funcion2);

Obsérvese que el prototipo de una función que recibe parámetros que a su vez son funciones, se hace como:

void GenerarTabla (TpFn Funcion1, TpFn Funcion2);

Si en otro lugar del programa se nos ocurre generar otra Tabla para otras funciones, por ej. ***coseno( )*** y ***coshyp( )***, podríamos utilizar la misma función GenerarTabla( ), simplemente redireccionando los punteros a funciones.

Ejemplo 2

Un ejemplo típico de parámetros funciones son las rutinas de ordenamiento que reciben como parámetro la función que realizará el ordenamiento en sí.

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

```
typedef int (*TFn)(int, int);
const int SIZE = 10;

void Burbuja         ( int *, const int, TFn               );
int    Ascendente    ( const int, const int                );
int    Descendente   ( const int, const int                );

/* ------------------------------------------------------------------------------------------------------ */
void main( )
{
  int  Vect[SIZE] = { 19,33,4,22,85,45,33,128,92,46 };
  int     i;
  int     Col  = 20;
  int     Fila = 10;
  char Tecla;

  clrscr( ); highvideo( );
  do {
        gotoxy(Col,Fila);      cprintf("ORDENACION ASCENDENTE    [1]");
        gotoxy(Col,Fila+1);  cprintf("ORDENACION DESCENDENTE [2]");
        gotoxy(Col,Fila+2);  cprintf("FINALIZA OPERACIONES          [3]");
        gotoxy(Col,Fila+3);  cprintf("-----------------------------------------------");
        gotoxy(Col,Fila+4);

        while((Tecla=getch( ))<'1' || Tecla>'3');

        switch(Tecla) {
            case '1' : Burbuja(Vect,SIZE,Ascendente);
                       gotoxy(1,20);
                       for(i=0;i<SIZE;i++) cprintf("%d ",Vect[i]);
                       getch(); gotoxy(1,20); clreol();
                       break;
            case '2' : Burbuja(Vect,SIZE,Descendente);
                       gotoxy(1,20);
                       for(i=0;i<SIZE;i++) cprintf("%d ",Vect[i]);
                       getch(); gotoxy(1,20); clreol();
                       break;
       }

     } while(Tecla!='3');
}
/* ------------------------------------------------------------------------------------------------------ */
void Burbuja(int *Vect, const int SIZE, TFn Comparar)
{
   int  Paso, Cuenta;
   void Swap(int *, int *);

   for(Paso=1;Paso<=SIZE-1;Paso++)
      for(Cuenta=0;Cuenta<=SIZE-2;Cuenta++)
            if((*Comparar)(Vect[Cuenta],Vect[Cuenta+1]))


                  Swap(&Vect[Cuenta],&Vect[Cuenta+1]);
}
```

```
/* ------------------------------------------------------------------------------------------------------------ */
void Swap(int *Dato1, int *Dato2)
{ int Temp;  Temp=*Dato1;  *Dato1=*Dato2;  *Dato2=Temp; }
/* ------------------------------------------------------------------------------------------------------------ */
int Ascendente(const int a, const int b)
{ return(b<a); }
/* ------------------------------------------------------------------------------------------------------------ */
int Descendente(const int a, const int b)
{ return(b>a); }
/* ------------------------------------------------------------------------------------------------------------ */
```

La rutina de ordenamiento se denomina Burbuja y en su prototipo vemos:

void Burbuja (int *, const int, TFn);

que recibe un parámetro función. Este parámetro función se cargará con la dirección de la rutina de ordenamiento que en ese momento se decida: ***Ascendente( )*** o ***Descendente( )***.
La estructura ***switch( )*** que chequea el tipo de ordenamiento deseado, decide cuál función enviará como parámetro:

case '1' : Burbuja(Vect,SIZE,Ascendente); o bien:
case '2' : Burbuja(Vect,SIZE,Descendente);

Dentro de la función:

void Burbuja(int *Vect, const int SIZE, TFn Comparar);

que recibe con el nombre local ***Comparar*** el puntero a función, se utiliza la invocación de la función apuntada:

if((*Comparar)(Vect[Cuenta],Vect[Cuenta+1]));

con los elementos de trabajo como parámetros del puntero a función.

Arreglo de punteros a funciones.

Facilita la invocación de funciones al evitar estructuras de control complicadas. El siguiente ejemplo lo prueba:

Un arreglo de punteros a funciones contendrá las direcciones de 4 funciones destinadas a mover el cursor un lugar en alguna de las 4 direcciones.

```
#include <conio.h>

enum  { F1=59, F2=60, F3=61, F4=62 };

const int DIM = 4;
const int ESC = 27;



typedef void (*TFnMov[])(int *, int *);
```

```
void MovIzq  ( int *Col,int *Fila );
void MovDer ( int *Col,int *Fila );
void MovArr  ( int *Col,int *Fila );
void MovAb   ( int *Col,int *Fila );
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  TFnMov Fn      = { MovIzq, MovDer, MovArr, MovAb };
  int         Col     = 40;
  int         Fila     = 12;
  char      Tecla;

  clrscr( ); highvideo( ); gotoxy(Col,Fila);

  do {
        if(!(Tecla=getch( )))
            if((Tecla=getch( ))>=F1 && Tecla<=F4)
                  Fn[Tecla-F1](&Col,&Fila);

      } while(Tecla!=ESC);
}
/* ------------------------------------------------------------------------------------------------------------ */
void MovIzq(int *Col, int *Fila)
{ if(*Col>1) gotoxy(--(*Col),*Fila);  }
/* ------------------------------------------------------------------------------------------------------------ */
void MovDer(int *Col, int *Fila)
{ if(*Col<80) gotoxy(++(*Col),*Fila); }
/* ------------------------------------------------------------------------------------------------------------ */
void MovArr(int *Col, int *Fila)
{ if(*Fila>1) gotoxy(*Col,--(*Fila));  }
/* ------------------------------------------------------------------------------------------------------------ */
void MovAb(int *Col, int *Fila)
{ if(*Fila<25) gotoxy(*Col,++(*Fila)); }
/* ------------------------------------------------------------------------------------------------------------ */
```

Es notoria la sencillez de la estructura :

```
do {
        if(!(Tecla=getch( )))
            if((Tecla=getch( ))>=F1 && Tecla<=F4)
                  Fn[Tecla-F1](&Col,&Fila);

      } while(Tecla!=ESC);
```

que decide cuál función activar con el toque de las flechas de edición.

En cuanto al manejo de interrupciones del DOS en las cuales se utiliza punteros a funciones, excede el alcance de este material.

Problemas que dan dolores de cabeza.

Caso 1

Este problema muestra un error muy común: no asignar a un puntero una dirección válida antes de utilizarlo. Lo llamativo de este programa es que al ejecutarlo...**CORRE BIEN!** ... pero **ESTA MAL.**

Aquí no hemos inicializado **pENT**, de manera que su contenido no es una dirección correcta sino la basura que haya en memoria, y el que funcione bien es **PURA CASUALIDAD,** ya que la dirección aleatoria que posee el puntero no afecta nada importante.

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

const int DIM = 10;
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  int *pENT;
  int   i;

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

  for(i=0;i<DIM;i++) *(pENT+i)=50+random(50);
  for(i=0;i<DIM;i++) cprintf("%d ",*(pENT+i));
  getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Caso 2

Este programa comete un error muy torpe. Pierde la dirección de comienzo de una reserva y luego pretende visualizar lo cargado en ella (números aleatorios entre 50 y 100). En pantalla se ve la basura de memoria que está mas allá de la reserva.

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

const int DIM = 10;
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  int *pENT = (int *)malloc(DIM*sizeof(int));
  int   i;

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

  for(i=0;i<DIM;i++) *pENT++=50+random(50);     // Cambia la dirección de inicio.
  for(i=0;i<DIM;i++) cprintf("%d ",*(pENT+i));
  getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Caso 3

Aquí hemos pasado **POR VALOR** una variable puntero que toma su valor de inicialización en un función y que al retornar **SE PIERDE**. Sin embargo el programa **FUNCIONA BIEN**. De nuevo funciona bien por pura casualidad, puesto que la basura que contienen el puntero como dirección válida no afecta al funcionamiento del programa.

NOTA : Este programa fue compilado con el modelo **Small**.

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

const int DIM = 10;
void ReservarMemoria  ( int *pENT);
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
   int *pENT;
   int   i;

   clrscr( ); hihgvideo( ); randomize( );

    ReservarMemoria(pENT);
    cprintf("DIRECCION DE LA RESERVA EN EL main( ) = %p\r\n",pENT);

    for(i=0;i<DIM;i++) *(pENT+i)=50+random(50);
    for(i=0;i<DIM;i++)    cprintf("%d  ",*(pENT+i));

    getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
void ReservarMemoria(int *p)
{
  p=(int *)malloc(DIM*sizeof(int));
  cprintf("DIRECCION DE LA RESERVA = %p\r\n",p);
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Caso 4

En este programa deseamos reservar 2000 enteros en memoria dinámica, pero se ha cometido el error de indicárselo mal a **malloc( )** y solo se han reservado 2000 bytes en lugar de 4000. Los **cprintf( )** acompañados de la función **coreleft( )** muestran claramente este detalle.

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

typedef unsigned int INT;
const int DIM = 2000;


/* ------------------------------------------------------------------------------------------------------------ */
```

```
void main( )
{
  int    *pENT;
  INT   Antes,Despues;

  clrscr( ); highvideo( );

  cprintf("MEMORIA ANTES DE RESERVAR = %u\r\n",Antes=coreleft());
  pENT=(int *)malloc(DIM);   // No multiplica por el sizeof(int)
  cprintf("MEMORIA LUEGO DE RESERVAR = %u\r\n",Despues=coreleft());
  cprintf("MEMORIA RESERVADA                 = %u\r\n",Antes-Despues);
  getch( );

}
/* ------------------------------------------------------------------------------------------------------------ */
```

Caso 5

En este problema se desea cargar una frase breve en la **parte baja** de cada domicilio de un arreglo dinámico de enteros, y una secuencia numérica de la misma longitud de la cadena, pero en la **parte alta** de cada domicilio.

Se ha cometido el error de **NO INCREMENTAR CORRECTAMENTE** el puntero de guardado de la cadena. Al mostrar en pantalla la cadena guardada, vemos que solo se hallan algunos caracteres.

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

typedef unsigned char byte;
const int DIM = 10;
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  int     *pENT   = (int *)malloc(DIM*sizeof(int));
  char  *Frase   = "BORLANDC++";
  char  *pAux     = (char *)pENT;
  byte  *pByte    = ((byte *)pENT)+1;
  int      i;

  clrscr( ); highvideo( );

  for(i=0;i<=strlen(Frase);i++)   *pAux++=*(Frase+i);    // Debió incrementar de 2 en 2
  for(i=0;i<=strlen(Frase);i++) { *pByte=i; pByte+=2; }

  pAux=(char *)pENT;
  for(i=0;i<strlen(Frase);i++) { cprintf("%c",*pAux); pAux+=2; }



getch( );
```

```
}
/* ------------------------------------------------------------------------------------------------------------ */
```

Caso 6

En este programa estamos compilando con el modelo **Small** y deseamos direccionar la memoria de video que comienza en **0xB0000000**, para lo cual obviamente se requiere de un puntero far. Pero nos olvidamos de esta declarativa.

Al correr el programa en pantalla no aparece nada.

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

typedef unsigned char byte;
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
  byte   *pSCR   = (byte *)0xB8000000;
  char  *Frase    = "BORLANDC++";
  int       i;

  clrscr( ); highvideo( );

  for(i=0;i<=strlen(Frase);i++) { *pSCR=*(Frase+i); pSCR+=2; }

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

Caso 7

En este programa **NO HACEMOS LA RESERVA** para la estructura (que contiene dos miembros punteros), pero en cambio sí hacemos las reservas para los miembros de la misma, y asignamos valores. Al correrlo funciona bien, a pesar de haber procedido mal.

NOTA :

Las direcciones de los miembros se están asignando sobre una dirección aleatoria de la estructura. Por casualidad no afecta nada vital y todo funciona bien.

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


typedef struct TDatos {
```

```
                              char    *Frase;
                              int     *pENT;
                             };

char *ReservarMemoria ( int Size );
/* ------------------------------------------------------------------------------------------------------------ */
void main( )
{
      char *Frase = "BORLANDC ES UNA VERSION VELOZ Y CONFIABLE";
  TDatos *Datos;
        int  i;

  clrscr( ); highvideo( );

  if((Datos->Frase=ReservarMemoria(strlen(Frase)+1))==NULL) exit(1);
  if((Datos->pENT=(int *)ReservarMemoria(5*sizeof(int)))==NULL) exit(1);

  strcpy(Datos->Frase,Frase);
  for(i=0;i<5;i++) Datos->pENT[i]=i*i;

  for(i=0;i<5;i++) cprintf("%d\r\n",Datos->pENT[i]);

  getch( );
}
/* ------------------------------------------------------------------------------------------------------------ */
char *ReservarMemoria(int Size)
{ return((char *)malloc(Size)); }
/* ------------------------------------------------------------------------------------------------------------ */
```