Arreglos bidimensionales: Matrices.

C permite arrays multidimensionales. La forma más sencilla de array multidimensionales es la de dos dimensiones comúnmente se la llama **matriz de dos dimensiones.** Por ejemplo para declarar un array de enteros bidimensional Mat de tamaño 6 x 5, se escribiría:

int Mat[6][5];

En forma general para “n” dimensiones:

<tipo> Mat[DIM1][DIM2]…[DIMn];

Sabemos que los vectores se declaraban como:

const DIM = 16;
int Vect[DIM];

No había dudas de que en memoria RAM se trataba de una secuencia consecutiva de 64 bytes (DIM x sizeof(int)). Pero ¿y si tuviésemos más de una dimensión? La notación sintáctica se expresa como:

const DIM1 = 6; //cantidad de filas
const DIM2 = 5; //cantidad de columnas
int Mat[DIM1][DIM2];

La cantidad de bytes que ocupa en memoria un arreglo bidimensional viene dada por:

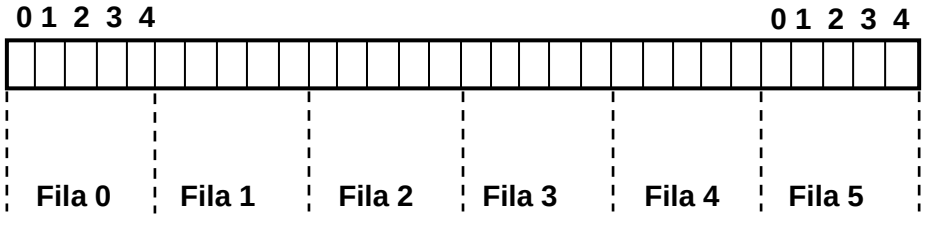
bytes = filas*columnas*sizeof(tipo de dato)

En forma general para “n” dimensiones:

(DIM1 x DIM2 x …..DIMn) x sizeof(Tipo)

Por lo tanto para este ejemplo sería **6x5xsizeof(int) = 120 bytes**

Sin embargo la memoria de la PC continúa siendo “lineal”, o sea no tiene forma de matriz de 2, de 3 ni de “n” dimensiones. Entonces ¿cómo funciona esta notación en la práctica? De la siguiente manera:



Este esquema representa una matriz de 6 x 5 (<tipo>Mat[6][5]).

Debe tomarse en cuenta que los números 0 1 2 3 4 no indican bytes, sino posiciones dentro del arreglo. En el caso que se considere un arreglo de char sí serán bytes, pero en general serán sólo domicilios.

Obviamente estamos representando un arreglo bidimensional y el compilador tendrá que ingeniárselas para saber a qué posición desde el comienzo deberá saltar para acceder a un domicilio cualquiera, por ejemplo el elemento Mat[2][3] y considerando que es una matriz de enteros, sería:

Offset = ((Fila x DIM2) + Col) x sizeof(int)
Offset = ((2 x 5) + 3) x 4
Offset = 52

Este cálculo deberá realizarlo el compilador cada vez que accedemos en forma aleatoria a cualquier domicilio. Esto nos indica que acceder aleatoriamente cuesta tiempo y esfuerzo, pues debe realizarse un cálculo, y si los accesos son miles o cientos de miles y las matrices son grandes, la situación empeora.

Sin embargo si los domicilios deben referenciarse en forma secuencial y consecutiva, y dado que en la realidad un arreglo bidimensional no existe en la memoria, sino un arreglo lineal, podemos utilizar un puntero cargado con la dirección de comienzo del arreglo y luego tan sólo ir incrementando su offset (desplazamiento). Por ejemplo:

```
int    Mat[DIM1][DIM2];
int *  pMat;
int    i;

pMat=&Mat[0][0];
for(i=0;i<DIM1*DIM2;i++) pMat[ i ] = 10 + random(41);
```

Esto equivaldría al conocido lazo:

```
for(i=0;i<DIM1;i++)
        for(j=0;j<DIM2;j++)
                Mat[ i ][ j ] = 10 + random(41);
```

O también sería hacer:

```
for(i=0;i<DIM1;i++){
    for(j=0;j<DIM2;j++){
    *(pM+((i*DIM2) + j))=100+random(900);
    }
 }
```

Con la diferencia que en el primer caso el compilador no tiene que hacer ningún cálculo para acceder a cada domicilio, sino incrementar su dirección.

Al igual que con los vectores, las matrices también comienzan en el domicilio **[0][0]** y finalizan en el domicilio **[DIM1 – 1][DIM2 – 1]** y cada elemento individual se comporta como un dato simple gozando de todas los privilegios y restricciones del tipo con que fuera declarado el arreglo.

Suele ser preferible definir una plantilla para las matrices, sobre todo si se declararán varias veces en distintas partes del programa, y luego hacer referencia a ella. Por ej.

```
const DIM1 = 8;
const DIM2 = 7;

typedef int TMat[DIM1][DIM2];

TMat Mat1;
TMat Mat2;
```

```
//---------EjeMatrices.cpp------------

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

const DIM1 = 10;
const DIM2 = 12;

typedef int TMat[DIM1][DIM2];

void main()
{
  TMat M;
  int *pM;
  int i,j;
pM=&M[0][0];//formas de asignar un puntero a una matriz
pM=(int*)M;//esta es otra forma
clrscr(); highvideo(); randomize();

//CARGA MATRIZ
 for(i=0;i<DIM1*DIM2;i++)
     pM[i]=100+random(900);

//OTRA FORMA DE CARGAR
for(i=0;i<DIM1;i++){
      for(j=0;j<DIM2;j++)
      {
```

```
    // *(pM+((i*DIM2) + j))=100+random(900); //otra forma de cargar una matriz
     cprintf("%4d",*(pM+((i*DIM2) + j)));
     }
     cprintf("\r\n");
 }
cprintf("\r\notra forma de cargar y mostrar una matriz\r\n\r\n");
//MUSTRA MATRIZ
  for(i=0;i<DIM1;i++){
     for(j=0;j<DIM2;j++)
            {
      M[i][j]=100+random(900);
      cprintf("%4d",M[i][j]);
     }
     cprintf("\r\n");
  }
  getch();
}
```

Estructuras

Las estructuras son colecciones de variables relacionadas bajo un nombre. Las estructuras pueden contener variables de muchos tipos diferentes de datos, a diferencia de los arreglos que contienen únicamente elementos de un mismo tipo de datos.

```
struct TRectang {
                    int Ancho;
                    int Largo;
                    char Letra;
                    };
```

La palabra reservada **struct** indica se está definiendo una estructura.
El identificador TRectang es el nombre de la estructura.
Las variables declaradas dentro de las llaves de la definición de estructura son los miembros de la estructura, estos tienen nombres únicos, mientras que dos estructuras diferentes pueden tener miembros con el mismo nombre.
Cada definición de estructura debe terminar con un punto y coma.
Los miembros de una estructura pueden ser variables de los tipos de datos básicos (int, char, float, etc), punteros, o agregados, como ser arreglos otras estructuras o punteros a otras estructuras (este tema lo veremos más adelante).
Una estructura no puede contener una instancia de sí misma.

Declaramos variables del tipo estructura de la siguiente manera:

```
struct  TRectang  rec1, arr[10];
```

o alternativamente sin usar la palabra struct:

TRectang rec1, arr[10];

Las declaraciones anteriores declaran variables e1 de tipo TRectang y arr de tipo arreglo de TRectang de dimensión 10.

Se pueden declarar variables de tipo estructura TRectang colocando sus nombres a continuación de la llave de cierre de la Definición de estructura y el punto y coma, en el caso anterior:

```
struct TRectang {
                    int Ancho;
                    int Largo;
                  } rec1, arr[10];
```

Una operación válida entre estructuras es asignar variables de estructura a variables de estructura del mismo tipo. Las estructuras no pueden compararse entre sí.

Como inicializar estructuras

Las estructuras pueden ser inicializadas mediante listas de inicialización como con los arreglos. Para inicializar una estructura escriba en la declaración de la variable a continuación del nombre de la variable un signo igual con los inicializadores entre llaves y separados por coma por ejemplo:

```
TRectang rec1 = { 6, 12 };
```

Si en la lista aparecen menos inicializadores que en la estructura los miembros restantes son automáticamente inicializados a 0.

Cómo acceder a los miembros de estructuras

Para tener acceso a miembros de estructuras utilizamos el operador punto.
El operador punto se utiliza colocando el nombre de la variable de tipo estructura seguido de un punto y seguido del nombre del miembro de la estructura. Por ejemplo, para imprimir el miembro Ancho de tipo int de la estructura rec1 utilizamos:

```
cprintf (" %d", rec1.Ancho);
```

Para acceder al miembro Ancho de la estructura rec1 escribimos: rec1.Ancho
En general, un miembro de una estructura particular es referenciado por una construcción de la forma:

Nombre_de_estructura.miembro

Por ejemplo para asignar el valor a un miembro podemos utilizar:

```
rec1.Ancho=6;
```

Ejemplo de clase

```
//---- EjeEstructura.cpp
#include <conio.h>
#include <string.h>

struct TRectang {
int Ancho;
int Largo;
      } ;
//----------------------------
typedef struct {
int Ancho;
int Largo;
      } TRectang1;

typedef struct TRectang Rect;

//------------------------------------------------------------
void main()
{
  TRectang Rec1;
  Rect Rec2={5,10};//inicializacion usando un "alias"

  clrscr(); highvideo();
    // --- ASIGNA DATOS A LAS ESTRUCTURAS -----------------
   Rec1.Ancho=6;
   Rec1.Largo=12;
 // --- MUESTRA DATOS POR PANTALLA ----------------------

   cprintf("Rec1.Ancho: %d\r\n",Rec1.Ancho);
   cprintf("Rec1.Largo: %d\r\n",Rec1.Largo);

   cprintf("Rec2.Ancho: %d\r\n",Rec2.Ancho);
   cprintf("Rec2.Largo: %d\r\n",Rec2.Largo);
}
```

Estructura anidada

Cuando una estructura es un elemento de otra estructura, se llama una ***estructura anidada***

Consideremos la información de una estructura fecha. y una estructura con los datos de un empleado. Declaramos toda esa información del siguiente modo:

```
 struct fecha {
                    int dia;
                    char nombre mes[9];
                    int anio;
              };


struct persona {
                    char nombre[tamanio_nombre];
                    char direccion[tamanio_dir];
                    long codigo postal;
                    long dni;
                    double salario;
                    fecha cumpleanios;
                    };
```

```
//----EjeEstructuraAnidada.cpp

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


struct fecha {
                        int dia;
                        char mes[15];
                        int anio;
                    };


struct persona {
                        char nombre[30];
                        char direccion[60];
                        long codigo_postal;
                        long dni;
                        fecha cumpleanios;
                        };

//-------------------------------------------------------
void main()
{
  persona Persona1;


  clrscr(); highvideo();

    // --- ASIGNA DATOS A LAS ESTRUCTURAS ----------------
   strcpy(Persona1.nombre,"Sergio Guardia");
   strcpy(Persona1.direccion,"Av. Independencia 1024");
```

```
  Persona1.dni=25458679;
  Persona1.codigo_postal=4000;

  Persona1.cumpleanios.dia=7;
  strcpy(Persona1.cumpleanios.mes,"Febrero");
  Persona1.cumpleanios.anio=1979;

 // --- MUESTRA DATOS POR PANTALLA ----------------------

  cprintf("Nombre: %s\r\n",Persona1.nombre);
  cprintf("DNI: %d\r\n",Persona1.dni);
  cprintf("Direccion: %s\r\n",Persona1.nombre);
  cprintf("Cod. Postal: %d\r\n",Persona1.codigo_postal);
  cprintf("Fecha de Nacimiento: %d de %s de
%d\r\n",Persona1.cumpleanios.dia,Persona1.cumpleanios.mes,Persona1.cumpl
eanios.anio);
}
```

Los bloques recuadrados muestran la notación de las estructuras anidadas.

Typedef

La palabra reservada typedef proporciona un mecanismo para la creación de sinónimos (o alias) para tipos de datos anteriormente definidos.

Por ejemplo:

typedef struct TRectangulo Rectangulo;

define Rectangulo como un sinónimo de TRectangulo.

Una forma alternativa de definir una estructura es:

```
Typedef struct {
                  int Ancho;
                  int Largo;
                } TRectang;
```

También **typedef** se utiliza para crear seudónimos para los tipos de datos básicos como lo vimos anteriormente cuando definíamos:

typedef unsigned int INT;

typedef int TMat[DIM1][DIM2];

RESERVAS DINÁMICAS

Hasta ahora cuando necesitábamos reservar un bloque consecutivo de bytes, hacíamos uso de la sintaxis:

```
int    Vector[DIM];
char   Letras[DIM2];
```
etc.

lo cual tenía la limitación de que en el momento de la compilación, el valor de **DIM** y **DIM2** debían ser **conocidos** por el compilador. Esto acarrea un problema: si no estamos seguros de la cantidad de datos que vamos a almacenar durante la ejecución, tendremos que **suponer** una magnitud, y por lo general tendemos a sobredimensionarla para no quedarnos cortos. En problemas de poca monta esto no plantea mayores inconvenientes, pero si los arreglos van a ser de estructuras donde cada unidad pueda requerir un bloque grande de bytes, los desperdicios por sobredimensionamientos pueden ser importantes.

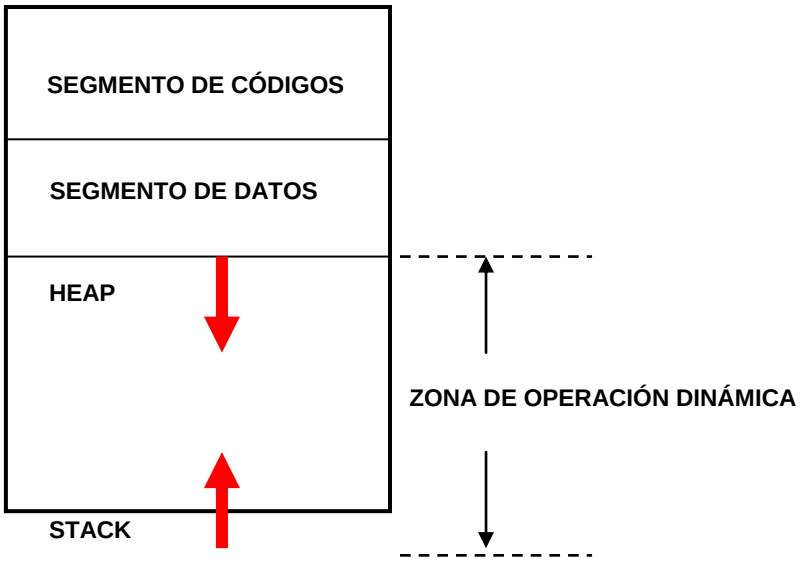
A este tipo de almacenamiento se los denomina **estáticos**, porque poseen bastante rigidez:

- **En el momento de escribir el código debemos conocer la cantidad de memoria a reservar.**
- **Sólo se admiten constantes para indicar la dimensión de la reserva.**
- **No pueden redimensionarse durante la ejecución del programa.**

La ventaja es la sencillez sintáctica y el estar familiarizados con este tipo de reservas.

La contracara de esto lo representan las **reservas dinámicas**. Todo lo que provenga de algo dinámico nos sugiere le idea de elasticidad, de adaptabilidad y de un mejor aprovechamiento de los recursos.

Para visualizar mejor el problema de las reservas, convendría echarle una mirada al esquema de memoria con que trabaja normalmente un lenguaje:



El Segmento de Datos contendrá todos aquellos identificadores declarados en forma global (fuera de toda función), por ejemplo:

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

const DIM = 512;
int Mat1[DIM];
int Mat2[DIM];
```

En cambio los identificadores que se declaran **dentro** de una función (**locales**), tienen su existencia en el **Stack**:

```
int MaxEnVector(int *V)
{
  int i, j;
  int Aux[16];
  etc.
}
```

Su alcance es local, debido a que cuando la función se extingue, la zona que ella ocupa es eliminada del **Stack** y todos los datos se pierden. El Stack indudablemente es una zona dinámica porque va ocupándose y liberándose a medida que las funciones se activan o finalizan.

El otro extremo de la memoria dinámica, el **Heap**, tiene un comportamiento parecido, pero sólo es ocupado cuando se solicitan reservas dinámicas, o sea que la ocupación de memoria se realiza durante la ejecución y recién entonces se decide cuántos bytes requeriremos para nuestro trabajo.

Es interesante destacar que el **Stack** y el **Heap** (montón) crecen en sentido contrario el uno del otro, quedando entre ellos una "tierra de nadie" que será concedida al primero que la solicite, ya sea una función que se active o una reserva dinámica. Esto nos hace pensar que puede existir un momento en el cual ambas zonas colisionen y se produzca un "cuelgue" del sistema. Normalmente con los valores que nosotros trabajaremos esto no ocurrirá nunca, pero en sistemas más complicados hay que tomar recaudos para evitar este peligro.

Cómo se realiza una reserva dinámica.

La librería **alloc.h** proporciona un abanico de funciones para el manejo de la memoria entre las cuales se encuentra:

void *malloc(int TotBytes)

malloc( ) – memory allocation o emplazamiento de memoria, se comunica con el administrador de memoria que verifica la existencia de espacio libre y si este cubre el requerimiento gestiona la reserva y retorna en su propio nombre la dirección de comienzo de la misma.

Importante:

La reserva es **SIEMPRE** una cierta cantidad de bytes. Al compilador no le interesa qué vamos a almacenar allí, él sólo entiende que debe salvaguardar "n" bytes para algún uso a cuenta del usuario. Un ejemplo aclarará el punto:

Queremos almacenar **DIM=100** valores de tipo double:

TotBytes = DIM * sizeof(double)

Posibles valores devueltos por malloc( ).

Si la solicitación de memoria resultó fallida, por ejemplo los bloques disponibles no alcanzaban a cubrir el requerimiento, la función **malloc( )** retorna **NULL**, y, si por el contrario todo anduvo bien, devuelve una dirección válida (la de comienzo de la reserva).

Esto puede resultarnos muy útil para chequear el éxito o fracaso de la gestión de memoria y a tal efecto podemos implementar la siguiente función de usuario:

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

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

Importante:

El hecho que tanto la función **malloc( )** como la func. de usuario **ReservarMemoria( )** retornen una dirección void, implica que tendremos que utilizar un cast para asignar esta dirección sin forma a una variable tipificada. Por ejemplo:

```
const DIM = 100;
double *pVect;


pVect = (double *)ReservarMemoria(DIM*sizeof(double));
```

Trabajando con las reservas dinámicas

Si bien en este ejemplo partimos de que conocemos de antemano la magnitud de la reserva, en la práctica no tiene por qué ocurrir así.

Deseamos almacenar en memoria dinámica 15 enteros aleatorios en el rango 100 y 900.

```
const DIM = 15;

int * pVect;
int     i;

pVect = (int *)ReservarMemoria(DIM*sizeof(int));

for(i=0;i<DIM;i++)  pVect[ i ] = 100+random(900-100+1);
```

Para complicar un poco las cosas, una variante es recorrer el arreglo con un puntero a char lo cual nos obligará a utilizar un cast.

```
const DIM = 128;
char *pVect;
int     I,min,max;

pVect = (char *)ReservarMemoria(DIM*sizeof(int));

for(i=0;i<DIM;i++)  ((int *)pVect)[ i ] = min+random(max-min+1);
```

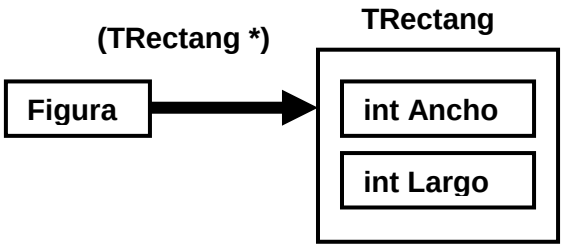
Donde la notación **((int *)pVect)[i]** es equiv. a ***(((int *)pVect) + i)**

Se recuerda la importancia de convertir a int "**antes**" de incrementar el puntero.

Punteros a estructuras

Cuando un puntero señala una reserva donde se halla una estructura, el acceso a los miembros de la misma se realiza a través del operador -> (flecha):

```
typedef struct { int Ancho; int Largo; } TRectang;

TRectang *Figura;

Figura = (TRectang *)ReservarMemoria(sizeof(TRectang));
Figura->Ancho = 25;
Figura->Largo = 30;
```



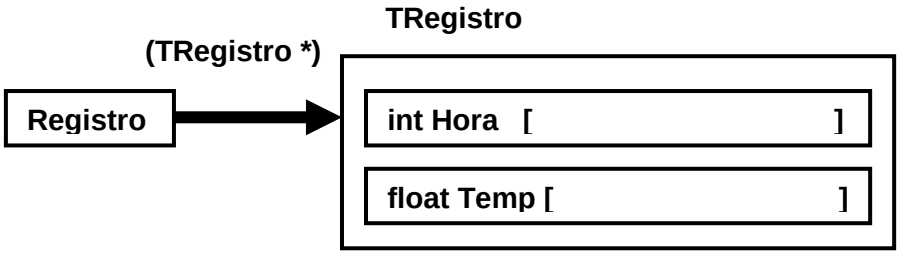
También podríamos haber escrito:

```
(*Figura).Ancho=25;
(*Figura).Largo=30;
```

Aunque es mucho más conveniente utilizar la flecha para ser coherente con la sintaxis de punteros.

Otro ejemplo:

```
const DIM = 4;
typedef struct  { int Hora[DIM]; float Temp[DIM]; } TRegistro;

TRegistro *Registro;

Registro=(TRegistro *)ReservarMemoria(sizeof(TRegistro));

Registro->Hora[0] = 14;
Registro->Temp[0] = 25.5;
etc.
```



Cuando un puntero señala el inicio de una reserva dinámica suficiente para almacenar N estructuras, el operador para acceder a los datos miembros va a depender de cómo lo escribamos:

- Si usamos notación subindexada se usa el operador . (punto)
- Si usamos notación indexada se usa el operador -> (flecha)

```
typedef struct { int Ancho; int Largo; } TRectang;
const DIM=5;
TRectang *Figura;

Figura = (TRectang *)ReservarMemoria(sizeof(TRectang)*DIM);

for(i=0;i<DIM;i++)
 {
       Figura[i].Ancho = random(100);
       Figura[i].Largo = random(100);
 }

for(i=0;i<DIM;i++)
 {
       (Figura+i)->Ancho = random(100);
       (Figura+i)->Largo = random(100);
 }
```

Cuando trabajamos con un puntero a estructuras anidada los miembros se acceden a través del operador flecha y también con el puntito:

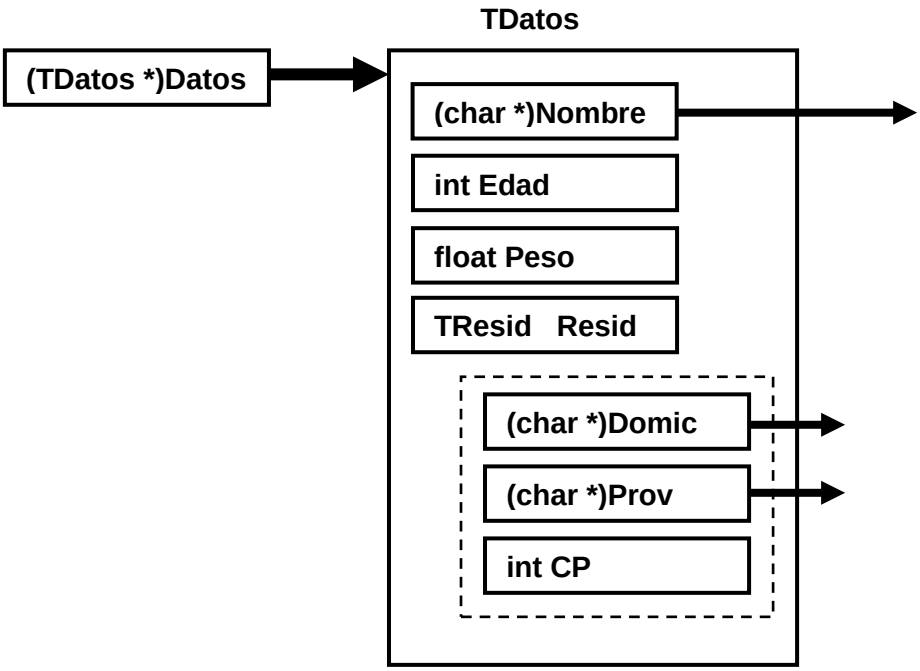
```
#include <conio.h>
#include <alloc.h>
#include <process.h>
#include <string.h>

typedef struct  { char *  Domic;
                  char *  Prov;
                  int     CP;
                } TResid;

typedef struct {  char *   Nombre;
                  int      Edad;
                  float    Peso;
                  TResid   Resid;
               }  TDatos;

void *ReservarMemoria(int TotBytes);
// ------------------------------------------------------
void main()
{
  TDatos *Datos;

  clrscr(); highvideo();

  // --- REALIZA RESERVAS DINAMICAS ---------------------

  Datos           = (TDatos *)ReservarMemoria(sizeof(TDatos));
  Datos->Nombre   = (char   *)ReservarMemoria(32);
  Datos->Resid.Domic= (char   *)ReservarMemoria(32);
  Datos->Resid.Prov = (char   *)ReservarMemoria(32);

  // --- ASIGNA DATOS A LAS ESTRUCTURAS -----------------

  strcpy(Datos->Nombre,"JUAN PEREZ");
  Datos->Edad=56;
  Datos->Peso=105;


  strcpy(Datos->Resid.Domic,"AV. BELGRANO 1240");
  strcpy(Datos->Resid.Prov,"TUCUMAN");
  Datos->Resid.CP = 4000;

  // --- MUESTRA DATOS POR PANTALLA ---------------------

  cprintf("NOMBRE    : %s\r\n",   Datos->Nombre);
  cprintf("EDAD      : %d\r\n",   Datos->Edad);
  cprintf("PESO      : %2.2f\r\n",Datos->Peso);
```

```
cprintf("DOMICILIO : %s\r\n",   Datos->Resid.Domic);
cprintf("PROVINCIA : %s\r\n",   Datos->Resid.Prov);
```

```
  getch();
}
```

Los bloques recuadrados muestran la doble notación con la flecha y el puntito.

Existe un detalle importante que no conviene pasar por alto, pero dibujemos primero el esquema de la estructura de datos:



En el momento que el compilador ejecuta la instrucción:

Datos = (TDatos *)ReservarMemoria(sizeof(TDatos));

Se efectiviza la reserva para todos los miembros de la estructura. Sin embargo los punteros **Nombre**, **Domic** y **Prov** carecen de una dirección válida ya que el compilador sólo asegura 2 bytes para cada identificador (en los modelos pequeños de memoria). Pero ahora viene el cuidado que hay que tener:

La función **strcpy(char *,cadena_de_caracteres)** trabaja mal si la reserva para la cadena no ha sido efectuada previamente. O sea:

strcpy(Datos->Nombre,"JUAN PEREZ");

u otras similares, no arrojan mensajes de error porque se trata de una instrucción válida, pero a la hora de mostrar por pantalla los datos almacenados, sale cualquier cosa o un mensaje de error. ¿Solución? Realizar una reserva previa:

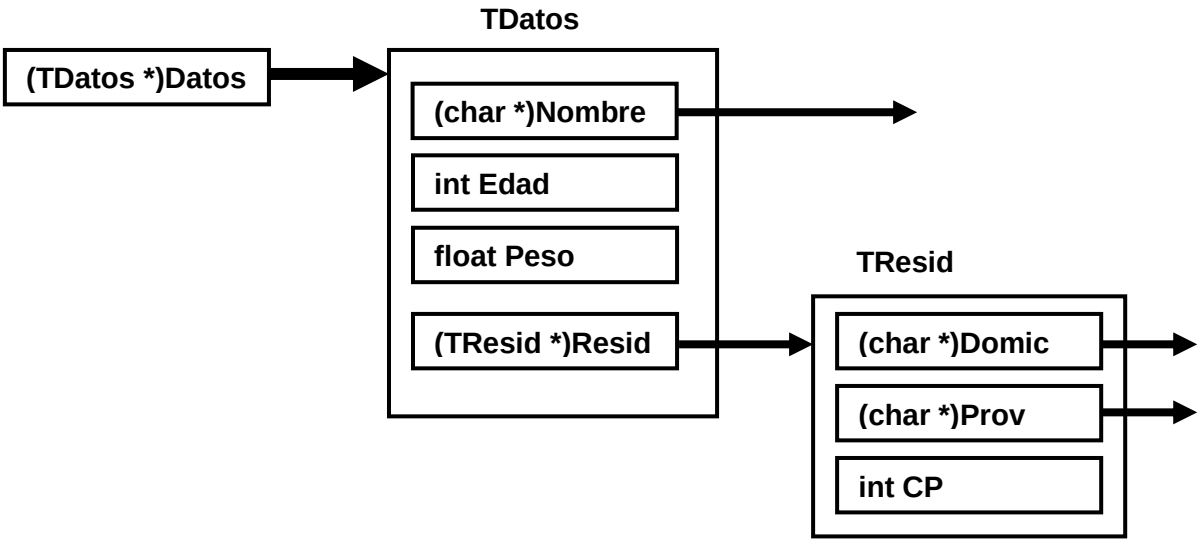
Datos->Nombre = (char *)ReservarMemoria(32);
Datos->Resid.Domic = (char *)ReservarMemoria(32);

Con toda justicia alguien podría alegar ¿de nuevo suponiendo la magnitud de la reserva, qué ganamos con hacer una reserva dinámica? Lo que pasa es que por simplificación suponemos el valor 32, pero en realidad se opera de otra manera: se ingresan las cadenas a través de variables, se determina su longitud y recién se formalizan las reservas “a medida”.

Una variante interesante es referenciar la estructura anidada a través de un puntero, en dicho caso los miembros se acceden a través del operador -> (flecha) únicamente.

```
// Ejemplo Nro.3-

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

typedef struct  {
                  char *  Domic;
                          char *  Prov;
                          int     CP;
                       }TResid;

typedef struct  {
                 char *   Nombre;
                  int      Edad;
                  float    Peso;
                  TResid * Resid;
           }TDatos;

void *ReservarMemoria(int TotBytes);

// ------------------------------------------------------
void main()
{
  TDatos *Datos;

  clrscr(); highvideo();
```

```
  // --- REALIZA RESERVAS DINAMICAS ----------------------

  Datos = (TDatos *)ReservarMemoria(sizeof(TDatos));
  Datos->Resid = (TResid *)ReservarMemoria(sizeof(TResid));

  Datos->Nombre = (char *)ReservarMemoria(32);
  Datos->Resid->Domic= (char *)ReservarMemoria(32);
  Datos->Resid->Prov = (char *)ReservarMemoria(32);

  // --- ASIGNA DATOS A LAS ESTRUCTURAS ------------------

  strcpy(Datos->Nombre,"Sergio Guardia");
  Datos->Edad=37;
  Datos->Peso=85;

  strcpy(Datos->Resid->Domic,"AV. Independencia 1700");
  strcpy(Datos->Resid->Prov,"TUCUMAN");
  Datos->Resid->CP=4000;

  // --- MUESTRA DATOS POR PANTALLA ----------------------

  cprintf("NOMBRE    : %s\r\n",   Datos->Nombre);
  cprintf("EDAD      : %d\r\n",   Datos->Edad);
  cprintf("PESO      : %2.2f\r\n",Datos->Peso);

  cprintf("DOMICILIO : %s\r\n",   Datos->Resid->Domic);

  getch();

}

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

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



Ahora los accesos a miembros son:

```
Datos->Resid->Domic = (char *)ReservarMemoria(32);
Datos->Resid->Prov  = (char *)ReservarMemoria(32);

strcpy(Datos->Resid->Domic,"AV. AMERICA 1440");
strcpy(Datos->Resid->Prov,"TUCUMAN");
```

La estructura puede ir haciéndose tan compleja como la naturaleza del problema lo requiera.