6. TIPOS DE DATOS DERIVADOS

Además de los tipos de datos fundamentales vistos en la Sección 2, en C++ existen algunos otros tipos de datos muy utilizados y que se pueden considerar derivados de los anteriores. En esta sección se van a presentar los ***punteros***, las ***matrices*** y las ***estructuras***.

6.1. Punteros

6.1.1. CONCEPTO DE PUNTERO O APUNTADOR

El valor de cada variable está almacenado en un lugar determinado de la memoria, caracterizado por una ***dirección*** (que se suele expresar en hexadecimal). El ordenador mantiene una ***tabla de direcciones*** (ver Tabla 6.1) que relaciona el nombre de cada variable con su dirección en la memoria. Gracias a los nombres de las variables (identificadores), de ordinario no hace falta que el programador se preocupe de la dirección de memoria donde están almacenados sus datos. Sin embargo, en ciertas ocasiones es más útil trabajar con las direcciones que con los propios nombres de las variables. El lenguaje C++ dispone del ***operador dirección*** (&) que permite determinar la dirección de una variable, y de un tipo especial de variables destinadas a contener direcciones de variables. Estas variables se llaman ***punteros*** o ***apuntadores*** (en inglés *pointers*).

Así pues, un ***puntero*** es una variable que puede contener la ***dirección*** de otra variable. Por tanto, los *punteros* están almacenados en algún lugar de la memoria y tienen su propia dirección (más adelante se verá que existen *punteros a punteros*). Se dice que un *puntero apunta a una variable* si su contenido es la dirección de esa variable. Un *puntero* ocupa de ordinario 4 bytes de memoria, y *se debe declarar o definir de acuerdo con el tipo del dato* al que apunta. Por ejemplo, un *puntero* a una variable de tipo ***int*** se *declara* del siguiente modo:

```
int *direc;
```

lo que quiere decir que a partir de este momento, la variable ***direc*** podrá contener la dirección de cualquier variable entera. La regla nemotécnica es que el valor al que apunta ***direc*** (es decir ****direc***, como luego se verá), es de tipo *int*. Los *punteros* a *long, char*, *float* y *double* se definen análogamente a los *punteros* a *int*.

6.1.2. OPERADORES DIRECCIÓN (&) E INDIRECCIÓN (*)

Como se ha dicho, el lenguaje C++ dispone del ***operador dirección*** (&) que permite hallar la dirección de la variable a la que se aplica. Un *puntero* es una verdadera variable, y por tanto puede cambiar la variable a la que apunta. Para acceder al valor depositado en la zona de memoria a la que apunta un *puntero* se debe utilizar el ***operador indirección*** (*). Por ejemplo, supóngase las siguientes declaraciones y sentencias,

```
int i, j, *p;      /* p es un puntero a int */
p = &i;            /* p contiene la dirección de i */
*p = 10;           /* i toma el valor 10 */
p = &j;            /* p contiene ahora la dirección de j */
*p = -2;           /* j toma el valor -2 */
```

Las constantes y las expresiones no tienen dirección, por lo que no se les puede aplicar el operador (&). Tampoco puede cambiarse la dirección de una variable. Los valores posibles para un puntero son las direcciones posibles de memoria. Un puntero puede tener valor 0 (equivalente a la

constante simbólica predefinida NULL). No se puede asignar una dirección absoluta directamente (habría que hacer un *casting*). Las siguientes sentencias son ilegales:

```
p = &34;            /* las constantes no tienen dirección */
p = &(i+1);         /* las expresiones no tienen dirección */
&i = p;             /* las direcciones de las variables no se pueden cambiar */
p = 17654;          /* habría que escribir p = (int *)17654; */
```

No se permiten asignaciones directas (sin *casting*) entre punteros que apuntan a distintos tipos de variables. Sin embargo, existe un tipo indefinido de punteros (***void ***, o ***punteros a void***), que puede asignarse y al que puede asignarse cualquier tipo de puntero. Sin embargo, no se les puede asignar ningún valor. Por ejemplo:

```
int *p;
double *q;
void *r;
p = q;              /* ilegal */
p = (int *)q;       /* legal */
r = q;              /* legal */
p = r;              /* ilegal */
*r=3;               /* ilegal */
```

6.1.3. ARITMÉTICA DE PUNTEROS

Como ya se ha visto, los *punteros* son unas variables un poco especiales, ya que guardan información –no sólo de la dirección a la que apuntan–, sino también del *tipo* de variable almacenado en esa dirección. Esto implica que no van a estar permitidas las operaciones que no tienen sentido con direcciones, como multiplicar o dividir, pero sí otras como sumar o restar. Además estas operaciones se realizan de un modo correcto, pero que no es el ordinario. Así, la sentencia:

```
p = p+1;
```

hace que **p** apunte a la dirección siguiente de la que apuntaba, teniendo en cuenta el tipo de dato. Por ejemplo, si el valor apuntado por **p** es *short int* y ocupa 2 bytes, el sumar 1 a **p** implica desplazar 2 bytes la dirección que contiene, mientras que si **p** apunta a un *double*, sumarle 1 implica desplazarlo 8 bytes.

También tiene sentido la *diferencia de punteros* al mismo *tipo* de variable. El resultado es la *distancia* entre las direcciones de las variables apuntadas por ellos, no en *bytes* sino en *datos* de ese mismo tipo. Las siguientes expresiones tienen pleno sentido en C++:

```
p = p + 1;
p = p + i;
p += 1;
p++;
```

Tabla 6.1. Tabla de direcciones.

Variable	Dirección de memoria
A	00FA:0000
B	00FA:0002
C	00FA:0004
p1	00FA:0006
p2	00FA:000A

P	00FA:000E

El siguiente ejemplo ilustra la aritmética de punteros:

```
void main(void) {
     int   a, b, c;
     int   *p1, *p2;
     int  **p;     /* p es un puntero que apuntará a otro puntero */

     p1 = &a;    /* Paso 1. La dirección de a es asignada a p1 */
     *p1 = 1;    /* Paso 2. p1 (a) es igual a 1. Equivale a a = 1; */
     p2 = &b;    /* Paso 3. La dirección de b es asignada a p2*/
     *p2 = 2;    /* Paso 4. p2 (b) es igual a 2. Equivale a b = 2; */
     p1 = p2;    /* Paso 5. El valor del p1 = p2 */
     *p1 = 0;    /* Paso 6. b = 0 */
     p2 = &c;    /* Paso 7. La dirección de c es asignada a p2 */
     *p2 = 3;    /* Paso 8. c = 3 */
     cout << a << b << c << endl;    /* Paso 9. Se imprime 103 */

     p = &p1;    /* Paso 10. p contiene la dirección de p1 */
     *p = p2;    /* Paso 11. p1= p2; */
     *p1 = 1;    /* Paso 12. c = 1 */
     cout << a << b << c << endl;    /* Paso 13. Se imprime 101 */
}
```

Supóngase que en el momento de comenzar la ejecución, las direcciones de memoria de las distintas variables son las mostradas en la Tabla 6.1.

La dirección de memoria está en hexadecimal, con el *segmento* y el *offset* separados por dos puntos (:); basta prestar atención al segundo de estos números, esto es, al *offset*.

La Tabla 6.2 muestra los valores de las variables en la ejecución del programa paso a paso. Se muestran en negrita y cursiva los cambios entre paso y paso. Es importante analizar y entender los cambios de valor.

Tabla 6.2. Ejecución paso a paso de un ejemplo con punteros.

Paso	**a** 00FA:0000	**b** 00FA:0002	**c** 00FA:0004	**p1** 00FA:0006	**p2** 00FA:000A	**p** 00FA:000E
1				***00FA:0000***		
2	***1***			00FA:0000		
3	1			00FA:0000	***000FA:0002***	
4	1	***2***		00FA:0000	000FA:0002	
5	1	2		***000FA:0002***	000FA:0002	
6	1	***0***		000FA:0002	000FA:0002	
7	1	0		000FA:0002	***000FA:0004***	
8	1	0	***3***	000FA:0002	000FA:0004	
9	1	0	3	000FA:0002	000FA:0004	
10	1	0	3	000FA:0002	000FA:0004	***000FA:0006***
11	1	0	3	***000FA:0004***	000FA:0004	000FA:0006
12	1	0	***1***	000FA:0004	000FA:0004	000FA:0006

13	1	0	1	000FA:0004	000FA:0004	000FA:0006

6.2. Vectores, matrices y cadenas de caracteres

Un ***array*** (también conocido como *arreglo*, *vector* o *matriz*) *es un modo de manejar una gran cantidad de datos del mismo tipo bajo un mismo nombre o identificador*. Por ejemplo, mediante la sentencia:

```
double a[10];
```

se reserva espacio para 10 variables de tipo ***double***. Las 10 variables se llaman **a** y se accede a una u otra por medio de un ***subíndice***, que es una *expresión entera* escrita a continuación del nombre entre corchetes [...]. La forma general de la declaración de un vector es la siguiente:

```
tipo nombre[numero_elementos];
```

Los elementos se numeran desde 0 hasta (numero_elementos-1). El tamaño de un vector puede definirse con cualquier expresión constante entera. Para definir tamaños son particularmente útiles las *constantes simbólicas*. En C++ no se puede operar con todo un vector o toda una matriz como una única entidad, sino que hay que tratar sus elementos uno a uno. Los vectores (mejor dicho, los elementos de un vector) se utilizan en las expresiones de C++ como cualquier otra variable. Ejemplos de uso de vectores son los siguientes:

```
a[5] = 0.8;
a[9] = 30. * a[5];
a[0] = 3. * a[9] - a[5]/a[9];
a[3] = (a[0] + a[9])/a[3];
```

Una ***cadena de caracteres*** no es sino un vector de tipo ***char***, con alguna particularidad que conviene resaltar. Las cadenas suelen contener texto (nombres, frases, etc.), y éste se almacena en la parte inicial de la cadena (a partir de la posición cero del vector). Para separar la parte que contiene texto de la parte no utilizada, se utiliza un *carácter fin de texto* que es el carácter nulo ('\0') según el código ASCII. Este carácter se introduce automáticamente al leer o inicializar las cadenas de caracteres, como en el siguiente ejemplo:

```
char ciudad[20] = "San Sebastián";
```

donde a los 13 caracteres del nombre de esta ciudad se añade un decimocuarto: el '\0'. El resto del espacio reservado –hasta la posición **ciudad[19]**– no se utiliza. De modo análogo, una cadena constante tal como "**mar**" ocupa 4 bytes (para las 3 letras y el '\0').

Las ***matrices*** se declaran de forma análoga, con corchetes independientes para cada subíndice. La forma general de la declaración es:

```
tipo nombre[numero_filas][numero_columnas];
```

donde tanto las *filas* como las *columnas* se numeran también a partir de 0. La forma de acceder a los elementos de la matriz es utilizando su nombre, seguido de las expresiones enteras correspondientes a los dos subíndices, entre corchetes.

En C++ tanto los vectores como las matrices admiten los *tipos* de las variables escalares (***char***, ***int***, ***long***, ***float***, ***double***, etc.), y los *modos de almacenamiento* ***auto***, ***extern*** y ***static***, con las mismas características que las variables normales (escalares). No se admite el modo *register*. Los arrays *static* y *extern* se inicializan a cero por defecto. Los arrays *auto* no se inicializan, contienen basura informática.

Las matrices en C++ *se almacenan por filas*, en posiciones consecutivas de memoria. En cierta forma, una matriz se puede ver como un *vector de vectores-fila*. Si una matriz tiene N filas (numeradas de 0 a N-1) y M columnas (numeradas de 0 a la M-1), el elemento (i, j) ocupa el lugar:

posición_elemento(0, 0) + i * M + j

A esta fórmula se le llama *fórmula de direccionamiento* de la matriz.

En C++ pueden definirse *arrays* con tantos subíndices como se desee. Por ejemplo, la sentencia,

```
double a[3][5][7];
```

declara una ***hipermatriz*** con tres subíndices, que podría verse como un conjunto de 3 matrices de dimensión (5x7). En la fórmula de direccionamiento correspondiente, el último subíndice es el que varía más rápidamente.

Como se verá más adelante, los *arrays* presentan una especial relación con los *punteros*. Puesto que los elementos de un vector y una matriz están almacenados consecutivamente en la memoria, la *aritmética de punteros* descrita previamente presenta muchas ventajas. Por ejemplo, supóngase el código siguiente:

```
int  vect[10], mat[3][5], *p;
p = &vect[0];
cout << *(p + 2) << endl;         /* se imprimirá vect[2]*/
p = &mat[0][0];
cout << *(p + 2) << endl;         /* se imprimirá mat[0][2]*/
cout << *(p + 4) << endl;         /* se imprimirá mat[0][4]*/
cout << *(p + 5) << endl;         /* se imprimirá mat[1][0]*/
cout << *(p + 12) << endl;        /* se imprimirá mat[2][2]*/
```

6.2.1. RELACION ENTRE VECTORES Y PUNTEROS

Existe una relación muy estrecha entre los vectores y los punteros. De hecho, el *nombre de un vector es un puntero* (constante, en el sentido de que no puede apuntar a otra variable distinta de aquélla a la que apunta) a la dirección de memoria que contiene el primer elemento del vector. Supónganse las siguientes declaraciones y sentencias:

```
double vect[10];  /* vect es un puntero a vect[0] */
double *p;
...
p = &vect[0];     /* p = vect; */
...
```

El identificador **vect**, es decir *el nombre del vector*, es un *puntero* al primer elemento del vector **vect[ ]**. Esto es lo mismo que decir que el valor de **vect** es **&vect[0]**. Existen más puntos de coincidencia entre los vectores y los punteros:

– Puesto que el nombre de un vector es un *puntero*, obedecerá las leyes de la aritmética de punteros. Por tanto, si **vect** apunta a **vect[0]**, (**vect+1**) apuntará a **vect[1]**, y (**vect+i**) apuntará a **vect[i]**.

– Recíprocamente (y esto resulta quizás más sorprendente), a los *punteros* se les pueden poner *subíndices*, igual que a los vectores. Así pues, si **p** apunta a **vect[0]** se puede escribir:

```
p[3]=p[2]*2.0;     /* equivalente a vect[3]=vect[2]*2.0; */
```

– Si se supone que **p=vect**, la relación entre *punteros* y *vectores* puede resumirse como se indica en las líneas siguientes:

```
*p     equivale a vect[0], a *vect     y a p[0]
*(p+1) equivale a vect[1], a *(vect+1) y a p[1]
*(p+2) equivale a vect[2], a *(vect+2) y a p[2]
```

Como ejemplo de la relación entre vectores y punteros, se van a ver varias formas posibles para sumar los N elementos de un vector **a[ ]**. Supóngase la siguiente declaración y las siguientes sentencias:

```
int a[N], suma, i, *p;

for(i=0, suma=0; i<N; ++i)          /* forma 1 */
     suma += a[i];

for(i=0, suma=0; i<N; ++i)          /* forma 2 */
     suma += *(a+i);

for(p=a, i=0, suma=0; i<N; ++i)     /* forma 3 */
     suma += p[i];

for(p=a, suma=0; p<&a[N]; ++p)      /* forma 4 */
     suma += *p;
```

6.2.2. RELACIÓN ENTRE MATRICES Y PUNTEROS

En el caso de las *matrices* la relación con los *punteros* es un poco más complicada. Supóngase una declaración como la siguiente

```
int mat[5][3], *p;
```

El *nombre de la matriz* (**mat**) es un *puntero* al primer elemento de u**n** *vector de punteros* **mat[ ]** (por tanto, existe un vector de punteros que tiene también el mismo nombre que la matriz), cuyos elementos contienen las direcciones del primer elemento de cada fila de la matriz. El nombre **mat** es pues un *puntero a puntero*. El *vector de punteros* **mat[ ]** se crea automáticamente al crearse la matriz. Así pues, **mat** es igual a **&mat[0]**; y **mat[0]** es **&mat[0][0]**. Análogamente, **mat[1]** es **&mat[1][0]**, **mat[2]** es **&mat[2][0]**, etc. La dirección base sobre la que se direccionan todos los elementos de la matriz no es **mat**, sino **&mat[0][0]**. Recuérdese también que, por la relación entre vectores y punteros, (**mat+i**) apunta a **mat[i]**. Recuérdese que la fórmula de direccionamiento de una matriz de N filas y M columnas establece que la dirección del elemento (i, j) viene dada por:

dirección (i, j) = dirección (0, 0) + i*M + j

Teniendo esto en cuenta y haciendo **p = mat;** se tendrán las siguientes formas de acceder a los elementos de la matriz:

```
*p     es el valor de mat[0]      **p          es mat[0][0]
*(p+1) es el valor de mat[1]      **(p+1)      es mat[1][0]
                                  *(*(p+1)+1)  es mat[1][1]
```

Por otra parte, si la matriz tiene M columnas y si se hace **p = &mat[0][0]** (dirección base de la matriz. Recuérdese que esto es diferente del caso anterior **p = mat**), el elemento **mat[i][j]** puede ser accedido de varias formas. Basta recordar que dicho elemento tiene por delante **i** filas completas, y **j** elementos de su fila:

```
*(p + M*i + j)      /* fórmula de direccionamiento */
*(mat[i] + j)       /* primer elemento fila i desplazado j elementos */
(*(mat + i))[j]     /* [j] equivale a sumar j a un puntero */
*((*(mat 0+ i)) + j)
```

Todas estas relaciones tienen una gran importancia, pues implican una correcta comprensión de los punteros y de las matrices. De todas formas, hay que indicar que las *matrices* no son del todo idénticas a los *vectores de punteros*: Si se define una matriz explícitamente por medio de vectores de punteros, las filas pueden tener diferente número de elementos, y no queda garantizado que estén contiguas en la memoria (aunque se puede hacer que sí lo sean). No sería pues posible en este caso utilizar la fórmula de direccionamiento y el acceder por columnas a los elementos de la matriz. La Figura 6.1 resume gráficamente la relación entre matrices y vectores de punteros.

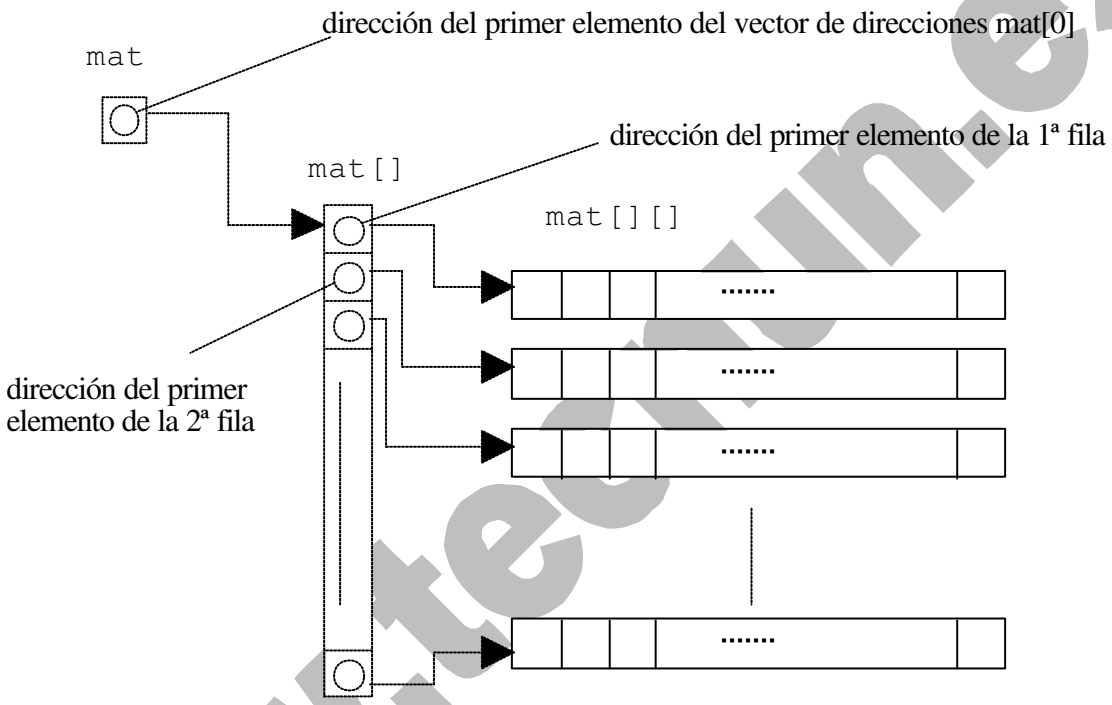

Figura 6.1. Relación entre matrices y punteros.

6.2.3. INICIALIZACIÓN DE VECTORES Y MATRICES

La inicialización de un *array* se puede hacer de varias maneras:

– Declarando el array como tal e inicializándolo luego mediante lectura o asignación por medio de un bucle ***for***:

```
double vect[N];
...
for(i = 0; i < N; i++)
     cin >> vect[i];
...
```

– Inicializándolo en la misma declaración, en la forma:

```
double   v[6] = {1., 2., 3., 3., 2., 1.};
float     d[] = {1.2, 3.4, 5.1};          /* d[3] está implícito */
int    f[100] = {0};                      /* todo se inicializa a 0 */
int     h[10] = {1, 2, 3};                /* restantes elementos a 0 */
int mat[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

6.3. Estructuras

Una ***estructura*** es *una forma de agrupar un conjunto de datos de distinto tipo bajo un mismo nombre o identificador*. Por ejemplo, supóngase que se desea diseñar una estructura que guarde los datos correspondientes a un alumno de primero. Esta *estructura*, a la que se llamará ***alumno***, deberá guardar el nombre, la dirección, el número de matrícula, el teléfono, y las notas en las 10 asignaturas. Cada uno de estos datos se denomina ***miembro*** de la estructura.
El *modelo* o *patrón* de esta estructura puede crearse del siguiente modo:

```
struct alumno {
     char nombre[31];
     char direccion[21];
     unsigned long no_matricula;
     unsigned long telefono;
     float notas[10];
};
```

El código anterior crea el *tipo* de dato **alumno**, pero aún no hay ninguna variable declarada con este nuevo tipo. Obsérvese la necesidad de incluir un carácter (;) después de cerrar las llaves. Para declarar dos variables de tipo **alumno** se debe utilizar la sentencia

```
alumno alumno1, alumno2;
```

donde tanto **alumno1** como **alumno2** son una estructura, que podrá almacenar un nombre de hasta 30 caracteres, una dirección de hasta 20 caracteres, el número de matrícula, el número de teléfono y las notas de las 10 asignaturas. También podrían haberse definido **alumno1** y **alumno2** al mismo tiempo que se definía la estructura de tipo **alumno**. Para ello bastaría haber hecho:

```
struct alumno {
     char nombre[31];
     char direccion[21];
     unsigned long no_matricula;
     unsigned long telefono;
     float notas[10];
} alumno1, alumno2;
```

Para acceder a los miembros de una estructura se utiliza el ***operador punto*** (.), precedido por el nombre de la *estructura* y seguido del nombre del *miembro*. Por ejemplo, para dar valor al **telefono** del alumno **alumno1** el valor 943903456, se escribirá:

```
alumno1.telefono = 943903456;
```

y para guardar la dirección de este mismo alumno, se escribirá:

```
strcpy(alumno1.direccion , "C/ Penny Lane 1,2-A");
```

El tipo de estructura creado se puede utilizar para definir más variables o estructuras de tipo **alumno**, así como vectores de estructuras de este tipo. Por ejemplo:

```
alumno nuevo_alumno, clase[300];
```

En este caso, **nuevo_alumno** es una estructura de tipo **alumno**, y **clase[300]** es un *vector de estructuras* con espacio para almacenar los datos de 300 alumnos. El número de matrícula del alumno 264 podrá ser accedido como **clase[264].no_matricula**.

Los *miembros* de las estructuras pueden ser variables de cualquier tipo, incluyendo vectores y matrices, e incluso otras estructuras previamente definidas. Las estructuras se diferencian de los *arrays* (vectores y matrices) en varios aspectos. Por una parte, los *arrays* contienen información

múltiple pero homogénea, mientras que los *miembros* de las estructuras pueden ser de naturaleza muy diferente. Además, *las estructuras permiten ciertas operaciones globales que no se pueden realizar con arrays*. Por ejemplo, la sentencia siguiente:

```
clase[298] = nuevo_alumno;
```

hace que se copien todos los miembros de la estructura **nuevo_alumno** en los miembros correspondientes de la estructura **clase[298]**. Estas operaciones globales no son posibles con *arrays*.

Se pueden definir también ***punteros a estructuras***:

```
alumno *pt;

pt = &nuevo_alumno;
```

Ahora, el puntero **pt** apunta a la estructura **nuevo_alumno** y esto permite una nueva forma de acceder a sus miembros utilizando el ***operador flecha*** (->), constituido por los signos (-) y (>). Así, para acceder al teléfono del alumno **nuevo_alumno**, se puede utilizar cualquiera de las siguientes sentencias:

```
pt->telefono;
(*pt).telefono;
```

donde el paréntesis es necesario por la mayor prioridad del operador (.) respecto a (*).

Las estructuras admiten los mismos modos *auto*, *extern* y *static* que los *arrays* y las variables escalares. Las reglas de inicialización a cero por defecto de los modos *extern* y *static* se mantienen. Por lo demás, una estructura puede inicializarse en el momento de la declaración de modo análogo a como se inicializan los vectores y matrices. Por ejemplo, una forma de declarar e inicializar a la vez la estructura **alumno_nuevo** podría ser la siguiente:

```
struct alumno {
     char nombre[31];
     char direccion[21];
     unsigned long no_matricula;
     unsigned long telefono;
     float notas[10];
} alumno_nuevo = {"Mike Smith", "San Martín 87, 2º A", 62419, 421794};
```

donde, como no se proporciona valor para las notas, estas se inicializan a cero.

Las estructuras constituyen uno de los aspectos más potentes del lenguaje C++. En esta sección se ha tratado sólo de hacer una breve presentación de sus posibilidades.

6.4. Gestión dinámica de la memoria

Según lo visto hasta ahora, *la reserva o asignación de memoria* para vectores y matrices se hace de forma automática con la declaración de dichas variables, asignando suficiente memoria para resolver el problema de tamaño máximo, dejando el resto sin usar para problemas más pequeños. Así, si en una función encargada de realizar un producto de matrices, éstas se dimensionan para un tamaño máximo (100, 100), con dicha función se podrá calcular cualquier producto de un tamaño igual o inferior, pero aun en el caso de que el producto sea por ejemplo de tamaño (3, 3), la memoria reservada corresponderá al tamaño máximo (100, 100). Es muy útil el poder reservar más o menos memoria *en tiempo de ejecución*, según el tamaño del caso concreto que se vaya a resolver. A esto se llama ***reserva o gestión dinámica de memoria***.

Existe en C++ un operador que reserva la cantidad de memoria deseada en tiempo de ejecución. Se trata del operador ***new***, del que ya hablamos en el apartado 4.1.6. Este operador se utiliza de la siguiente forma:

```
tipo_variable *vector;

vector = new tipo_variable [variable];
```

El operador ***new*** se utiliza para crear variables de cualquier tipo, ya sean las estándar o las definidas por el usuario. La mayor ventaja de esta gestión de memoria es que se puede definir el tamaño del vector por medio de una variable que toma el valor adecuado en cada ejecución.

Existe también un operador llamado **delete** que deja libre la memoria reservada por **new** y que ya no se va a utilizar.Recordemos que cuando la reserva es dinámica la variable creada perdura hasta que sea explícitamente borrada por este operador. La memoria no se libera por defecto.

El prototipo de este operador es el siguiente:

```
delete [] vector;
```

A continuación se presenta a modo de ejemplo un programa que reserva memoria de modo dinámico para un vector de caracteres:

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

void main()
{
    char Nombre[50];
    cout << "Introduzca su Nombre:";
    cin  >> Nombre;

    char *CopiaNombre = new char[strlen(Nombre)+1];

    // Se copia el Nombre en la variable CopiaNombre
    strcpy(CopiaNombre, Nombre);
    cout << CopiaNombre;

    delete [] CopiaNombre;
}
```

El siguiente ejemplo reserva memoria dinámicamente para una matriz de ***doubles***:

```
#include <iostream.h>

void main(){
    int    nfil, ncol, i, j;
    double **mat;
    //se pide al usuario el tamaño de la matriz
    cout << "Introduzca numero de filas y columnas: ";
    cin >> nfil >> ncol;

    // se reserva memoria para el vector de punteros
    mat = new double*[nfil];
    // se reserva memoria para cada fila
    for (i=0; i<nfil; i++)
        mat[i] = new double[ncol];

    // se inicializa toda la matriz
    for(i=0; i<nfil; i++)
```

```
            for(j=0; j<ncol; j++)
                mat[i][j]=i+j;

        // se imprime la matriz
        for(i=0; i<nfil; i++){
            for(j=0; j<ncol; j++)
                cout << mat[i][j] << "\t";
            cout << "\n";
        }

....// se libera memoria
        for(i=0; i<nfil; i++)
            // se borran las filas de la matriz
            delete [] mat[i];
        // se borra el vector de punteros
        delete [] mat;
        }
}
```