Tema 5: Tipos de datos avanzados

1 Introducción.

Hasta ahora hemos estudiado los tipos de datos simples. En este tema estudiaremos cómo combinar los tipos de datos simples para obtener a partir de ellos tipos de datos compuestos. De esta forma, entendemos por un tipo de datos compuesto por aquél que surge como unión de varios tipos de datos simples en un solo objeto.

El tipo de datos compuesto más sencillo es el **tipo vector o array**, que es una lista indexada de elementos de un mismo tipo. Por ejemplo, un listado telefónico se almacenaría en un vector.

A partir de los vectores unidimensionales se pueden formar vectores de dos o más dimensiones, esto es, **vectores multidimensionales o matrices**. Por ejemplo, una imagen digital en niveles de gris está compuesta por un vector bidimensional de enteros en la que cada elemento simboliza el nivel de gris del píxel correspondiente.

También estudiaremos determinadas operaciones básicas que pueden realizarse sobre vectores, tales como la **búsqueda** de un elemento en un vector o la **ordenación** de vectores.

Por otro lado, existen otros tipos de datos compuestos que se forman a partir de elementos de distinto tipo, estos son los **registros** o **uniones**. Por ejemplo, la ficha de un cliente en un programa de agenda consta de su nombre, su dirección, su número de teléfono y su DNI. En ocasiones es interesante agrupar todos esos campos bajo un mismo tipo de datos.

Es posible combinar todos estos elementos para formar, por ejemplo, vectores o matrices compuestas de registros o registros que contienen vectores o matrices. Por ejemplo, una imagen digital en color está compuesta por una matriz bidimensional donde cada pixel o elemento de la matriz está compuesto por tres enteros. Cada entero simboliza el nivel de rojo, verde y azul del que está compuesto el color del píxel (modelo RGB).

Finalmente introduciremos unos tipos de datos mucho más avanzados como son las **listas**, las **colas** y las **pilas**. Para ello, será necesario explicar cómo se utiliza y para qué sirve la **memoria dinámica**.

2 Vectores y matrices

Un vector es una lista indexada de elementos del mismo tipo. Para declarar una variable de tipo vector utilizamos la siguiente sintaxis:

Tipo Nombre_vector[Num_elem];

Como vemos, coincide con la sintaxis de definición de cualquier variable, a excepción de que, en este caso, tenemos de especificar de cuántos elementos consta la lista.

Para acceder a cualquier elemento del vector simplemente especificamos, junto al nombre del vector, a qué elemento queremos acceder entre corchetes:

Nombre_vector[posicion]

Esto es a lo que nos referíamos al decir que se trata de listas indexadas. Como vemos, cada elemento en la lista posee una posición numérica. La numeración de los elementos del vector comienza en **0** (primer elemento) y termina en **Num_elem-1** (último elemento) por lo que *posicion* tendrá que estar en dicho rango. Cualquier referencia a una posición fuera de ese rango provocará un desbordamiento de memoria en tiempo de ejecución.

Veamos un ejemplo muy sencillo en el que creamos un vector de cinco elementos de tipo entero. A continuación solicitamos al usuario cada uno de los elementos. Finalmente listamos todos los elementos en el orden en el que los introdujo el usuario:

```
void main(void)
{
	int elem[5], i;
	for(i=0;i<5;i++){
		printf("Introduzca dato %d: ",i);
		scanf("%d",&elem[i]);
	}
	for(i=0;i<5;i++){
		printf("El dato colocado en %d es %d\n",i,elem[i]);
	}
}
```

Sin utilizar vectores tendríamos que declarar 5 variables enteras y no podríamos utilizar bucles para solicitarlas ni para listarlas posteriormente. El resultado sería el mismo pero ¿y si fuesen 500 variables?. Como vemos, la utilización de este tipo de datos resulta muy conveniente.

Veamos un nuevo ejemplo en el que solicitamos 500 números al usuario. A continuación solicitamos un nuevo número y tenemos que determinar si se encuentra o no en la lista:

```
void main(void)
{
	int elem[500], i,elemento, posicion;
	for(i=0;i<500;i++){
		printf(“Introduzca dato %d: ”);
		scanf(“%d”,&elem[i]);
	}
	printf("Elemento a buscar: "); scanf("%d",&elemento);
	posicion=-1;
	for(i=0; i<500 ;i++){
		if(elemento == elem[i]){
			posicion=i;
		}
	}
	if(posicion==-1)
		printf("El elemento NO esta en la lista\n");
	else
		printf("Posicion del elemento en la lista: %d\n",posicion);
}
```

Como vemos, en el ejemplo se declara una variable entera llamada *posicion* que se utiliza para almacenar el lugar en el que se encuentra el elemento en el vector. Si dicha posición sigue valiendo -1 (posición inválida) después de recorrer todo el vector significa que el elemento no se encuentra en el mismo.

Tanto los vectores unidimensionales como los multidimensionales (que veremos más adelante) tienen una característica común a la hora de pasarlos por una función, esto es, que el paso de parámetros de vectores y matrices se realiza siempre por referencia, sin necesidad de especificarlo con los operadores * y &. Esto significa que al pasar un vector o matriz a una función, estaremos trabajando siempre con los elementos originales, no con copias, por lo que los cambios que realicemos en la función sobre estos elementos se verán reflejados en las variables originales.

Veamos un ejemplo en el que pasamos un vector de enteros a una función y sustituimos cada elemento del mismo por el cuadrado del número que contenía:

```
void Cuadrados( int datos[], int tam)
{
	int i;
	for(i=0;i<tam;i++){
		datos[i]=datos[i]*datos[i];
	}
	return;
}
void  main(void)
{
	int vdatos[100],i;
```

```
    for(i=0;i<100;i++){
        printf("Introduzca numero:");
        scanf("%d",&(vdatos[i]));
    }
    Cuadrados(vdatos,100);
    for(i=0;i<100;i++) printf("Cuadrado(%d) = %d\n",i,vdatos[i]);
}
```

Como vemos, en la cabecera de la función no aparece el tamaño del vector en la definición del mismo. Puesto que dicho dato es necesario, nos hemos visto obligados a introducirlo en otro parámetro. De esta forma, la función se define de forma genérica para cualquier tamaño de vector. También podemos observar que aunque el vector se pasa por referencia no resulta necesario especificarlo implícitamente.

En el siguiente ejemplo hemos representado una función que tiene por objetivo invertir el orden de un vector de elementos reales que se le pasa como parámetro:

```
void Invertir( float datos[], int tam)
{
    int i;
    float aux;
    for(i=0;i<tam/2;i++){
        aux = datos[i];
        datos[i] = datos[tam-i-1];
        datos[tam-i-1] = aux;
    }
    return;
}
```

El procedimiento consiste en recorrer la primera mitad del vector intercambiando en cada momento el valor del vector en la posición actual *i* por su simétrica al final del vector *tam-i-1*. Este mismo problema se podría haber resuelto igualmente utilizando un vector auxiliar del mismo tamaño pero el consumo de memoria habría sido mayor.

Hasta ahora hemos estudiado los vectores unidimensionales, esto es, listas de elementos de una sola dimensión. Esta definición se puede ampliar definiendo vectores compuestos por varias dimensiones, esto es, **vectores multidimensionales o matrices**. La declaración de una variable vector multidimensional es la misma que para el vector simple pero especificando el tamaño de cada una de las dimensiones:

Tipo Nombre_matriz [Tam_d1] [Tam_d2]… [Tam_dN];

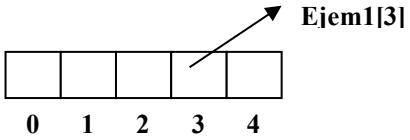
De igual forma, para acceder a cada elemento del vector es necesario especificar la posición dentro de cada una de las dimensiones:

Nombre_matriz [pos1] [pos2]… [posN]

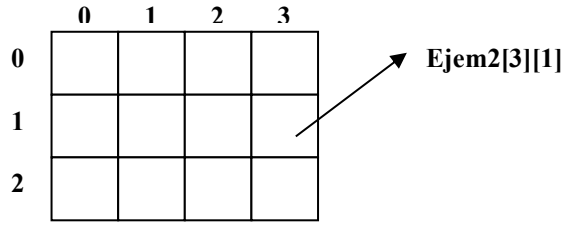
Como en el caso de los vectores simples, el rango sobre cada una de las dimensiones estará entre *0* y *Tam_dI-1*.

En la siguiente figura podemos ver la representación geométrica de tres vectores: Ejem1, Ejem2 y Ejem3 de una, dos y tres dimensiones respectivamente:

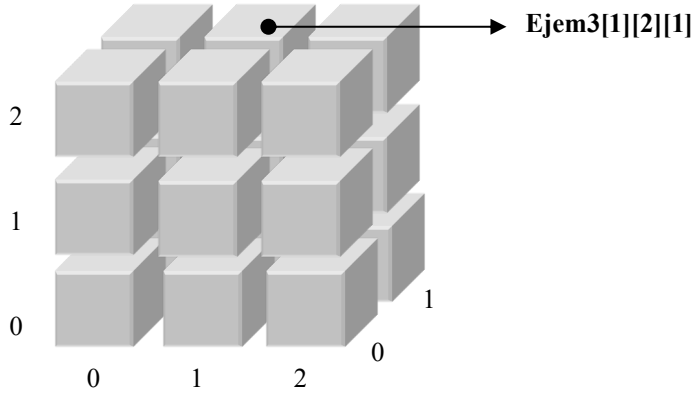
Vector unidimensional. int Ejem1[5];



Vector bidimensional. int Ejem2[4][3];



Vector tridimensional. int Ejem3[3][3][2];



Veamos un ejemplo de utilización de vectores y matrices en el que queremos calcular el producto de una matriz de tamaño NFIL X NCOL por un vector de tamaño NCOL, almacenando el resultado en un vector de tamaño NFIL:

```
#define NFIL 3
#define NCOL 4

void PMatrizVector( int M[NFIL][NCOL], int V[NCOL], int Resul[NFIL])
{
    int i,j;
    for(i=0;i<NFIL;i++){
        Resul[i]=0;
        for(j=0;j<NCOL;j++){
            Resul[i]=Resul[i]+M[i][j]*V[j];
        }
    }
    return;
}
```

```
void main(void)
{
	int Matriz[NFIL][NCOL];
	int Vector[NCOL];
	int Resultado[NFIL];
	int i,j;
	for(i=0;i<NFIL;i++){
		for(j=0;j<NCOL;j++){
			printf("Elemento %d,%d de la matriz:",i,j);
			scanf("%d",&Matriz[i][j]);
		}
	}
	for(i=0;i<NCOL;i++){
		printf("Elemento %d del vector:",i);
		scanf("%d",&Vector[i]);
	}

	PMatrizVector(Matriz,Vector,Resultado);

	printf("El resultado es:\n");
	for(i=0;i<NFIL;i++)
		printf("Resultado(%d)=%d\n",i,Resultado[i]);
	return;
}
```

Como vemos, en el programa se solicitan los valores de cada celda tanto de la matriz como del vector, cuyos tamaños se han especificado en las constantes *NFIL* y *NCOL*. Como sabemos, para multiplicar una matriz por un vector, el número de columnas de la matriz ha de coincidir con el número de elementos del vector y el resultado será un vector con tantos elementos como filas tiene la matriz. La operación de multiplicación se encapsula dentro de la función *PMatrizVector*.

Por ejemplo, vamos a realizar el siguiente producto:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \\ 194 \end{bmatrix}$$

Para ello, ejecutamos el programa anterior obteniendo:

```
Elemento 0,0 de la matriz:1
Elemento 0,1 de la matriz:2
Elemento 0,2 de la matriz:3
Elemento 0,3 de la matriz:4
Elemento 1,0 de la matriz:5
Elemento 1,1 de la matriz:6
```

```
Elemento 1,2 de la matriz:7
Elemento 1,3 de la matriz:8
Elemento 2,0 de la matriz:9
Elemento 2,1 de la matriz:10
Elemento 2,2 de la matriz:11
Elemento 2,3 de la matriz:12
Elemento 0 del vector:3
Elemento 1 del vector:4
Elemento 2 del vector:5
Elemento 3 del vector:6
El resultado es:
Resultado(0)=50
Resultado(1)=122
Resultado(2)=194
```

Una interesante aplicación de las matrices es el tratamiento de imagen digital. Una imagen en niveles de gris se puede almacenar en una matriz de enteros *int M[nfil][ncol]* donde cada elemento almacena el nivel de gris del pixel correspondiente. Sobre dicha matriz se pueden realizar distintas operaciones, como por ejemplo, determinados filtros. Vamos a ver un ejemplo de implementación de un filtro sencillo sobre una imagen digital, el *filtro de media*, que sirve para suavizar una imagen:

El filtro de media consiste en calcular, para cada píxel de la imagen, la media de los niveles de gris de sus píxeles vecinos. Los píxeles vecinos a un píxel [i,j] son los que están dentro de un cuadrado de tamaño $(2tm+1)^2$ centrado en [i,j]. Esto es lo que se conoce como una máscara de tamaño *tm*. Cuanto mayor sea *tm* mayor será el suavizado de la imagen. De esta forma, cada píxel de la imagen destino se forma a partir de la imagen origen aplicando la fórmula:

$$I_{Dest}[i,j] = \frac{\sum_{u=i-tm}^{i+tm} \sum_{v=j-tm}^{j+tm} I_{Orig}[u,v]}{(2tm+1)^2}$$

Para implementar el filtro es necesario recorrer la imagen original calculando, para cada píxel de la misma, el valor de su media. La función sería la siguiente:

```
//Ancho y alto de la imagen
#define W 640
#define H 480

void MediarImagen(int Orig[H][W], int Dest[H][W], int tm)
{
    int i,j,ii,jj,valor,TamMasc;
    //Inicializacion de la imagen destino
    for(i=0;i<H;i++){
        for(j=0;j<W;j++){
            Dest[i][j]=0;
```

```
            }
        }
        //Tamanyo de mascara
        TamMasc=(tm*2+1)*(tm*2+1);
        for(i=tm;i<H-tm;i++){
            for(j=tm;j<W-tm;j++){
                //Media para el pixel [i][j]
                valor=0;
                for(ii=i-tm;ii<=i+tm;ii++){
                    for(jj=j-tm;jj<=j+tm;jj++){
                        valor+=Orig[ii][jj];
                    }
                }
                valor/=TamMasc;
                Dest[i][j]=valor;
            }
        }
        return;
    }
```

Como vemos, hemos definido el tamaño de las imágenes en dos constantes *W* (anchura) y *H* (altura). La función recibe dos imágenes *Orig* (imagen original) y *Dest* (imagen destino) y el tamaño de la máscara (*tm*).

Lo primero que se ha de hacer es inicializar la imagen destino con un doble bucle que recorre todos sus píxeles poniéndolos a 0. A continuación se calcula el área de vecindad (*TamMasc*) que viene en función del tamaño de la más cara (*tm*).

Los dos bucles anidados siguientes nos sirven para recorrer la imagen y los dos bucles anidados que hay dentro de ellos nos sirven para calcular la media de cada píxel. Como puede verse, no se calcula la media de los bordes de la imagen, ya que éstos no tienen vecinos. El valor de la media se almacena en la variable entera *valor*, que después ha de mediarse dividiéndola por *TamMasc*. Una vez obtenida la media del píxel se almacena en la imagen de destino.

En la siguiente figura podemos observar la aplicación de este filtro sobre una imagen utilizando distintos tamaños de máscara:

Imagen original

Imagen suavizada (*tm*=2)

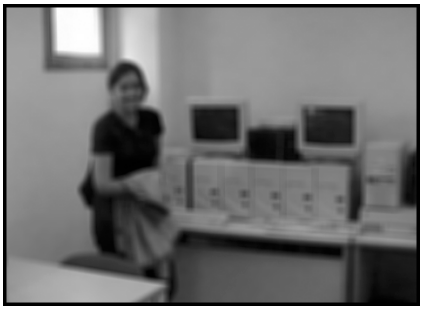


Imagen suavizada (*tm*=4)

Imagen suavizada (*tm*=6)

3 Búsqueda en vectores

En este apartado estudiaremos distintas técnicas para buscar un elemento dentro de un vector unidimensional. Distinguiremos entre dos técnicas: la búsqueda secuencial, en el caso en el que el vector se encuentra desordenado y la búsqueda binaria, que puede realizarse únicamente cuando el vector se encuentre ordenado y que, como veremos, posee ventajas computacionales con respecto a la primera.

3.1 Búsqueda secuencial

La solución más directa para buscar un elemento dentro de un vector consiste en recorrer todos sus elementos sin seguir ninguna estrategia. Esto puede hacerse sea cual sea el orden de los elementos del vector. Ya introdujimos este problema en el ejemplo del segundo punto de este mismo capítulo pero vamos a formalizarlo mejor:

Vamos a construir una función de búsqueda que reciba un vector y un elemento y devuelva un entero correspondiente a la posición del elemento dentro del vector. En el caso de que dicho elemento no se encuentre en el vector, devolverá una posición inválida, por ejemplo, el valor –1. Puesto que los elementos del vector pueden ser de cualquier tipo estudiado, construiremos la función de forma genérica, llamando a ese tipo *Tipo_base*. Para construir un algoritmo de búsqueda para vectores de tipo entero, bastaría con sustituir la palabra *Tipo_base* por *int*. El algoritmo sería el siguiente:

```
int BusquedaSecuencial1(Tipo_base vector[], int tvector, Tipo_base
elemento)
{
    int i,posicion=-1;
    for(i=0;i<tvector;i++){
        if(vector[i]==elemento)
            posicion=i;
    }
    return(posicion);
}
```

Como vemos, la posición se guarda en una variable inicializada con el valor –1. Si el elemento no está en la lista, dicho valor no cambiará y la función devolverá el valor –1. En otro caso devolverá la posición en la que se encuentra el elemento.

En esta primera solución es necesario recorrer todo el vector en todos los casos. Imaginemos que el vector contiene 3000 elementos y que el tercero coincide con el elemento a buscar. Una vez encontrado no sería necesario continuar, con lo que nos ahorraríamos 2997 iteraciones del bucle. Para evitar esto, veamos una segunda versión del algoritmo en la que añadimos una nueva condición en el bucle:

```
int BusquedaSecuencial2(Tipo_base vector[], int tvector, Tipo_base elemento)
{
    int i,posicion=-1;
    for(i=0;i<tvector && posicion==-1;i++){
        if(vector[i]==elemento)
            posicion=i;
    }
    return(posicion);
}
```

En la anterior función el bucle se repite mientras no lleguemos al final del vector y mientras la posición siga siendo inválida. En cuanto se cambie el valor de la posición (se encuentra un elemento coincidente) en la siguiente vuelta de bucle se incumplirá la segunda condición y terminará. De esta forma conseguimos que en el mejor de los casos, esto es, cuando el elemento a buscar esté en la primera posición, el algoritmo realice una única iteración.

Sin embargo, pese a que esta segunda versión es, sin duda, mejor que la primera, en el peor de los casos, esto es, cuando el elemento no se encuentra en el vector, es necesario realizar tantas iteraciones como elementos contiene el vector. En el siguiente apartado veremos una solución mucho más efectiva que podremos aplicar cuando el vector se encuentre previamente ordenado.

3.2 Búsqueda Binaria

La búsqueda secuencial se realiza únicamente cuando no conocemos el orden de los elementos dentro del vector. Si dichos elementos se encuentran ordenados, podemos realizar una búsqueda mucho más inteligente, descartando directamente rangos completos del vector. El procedimiento sería el siguiente:

Suponemos que los elementos en el vector están ordenados previamente en orden ascendente. Comenzamos buscando en la posición central de la lista. Si el elemento mitad es igual al que buscamos terminamos. Si el elemento es mayor al que está en dicha posición solo tendremos que buscar en la segunda mitad de la lista, puesto que todos los elementos de la primera mitad son menores al que buscamos. Si el elemento es menor al elemento mitad solamente tendremos que buscar en la primera mitad de la lista. A continuación realizamos el mismo procedimiento en la mitad elegida

(comprobamos el elemento central, etc...). De esta forma, vamos descartando subintervalos del vector y no es necesario comprobar todos los elementos del mismo. Este procedimiento se conoce como **búsqueda binaria** o **dicotómica**.

A continuación describimos el algoritmo correspondiente al procedimiento, encapsulado en una función que devuelve la posición del elemento en el caso de encontrarse en la lista y –1 en caso contrario:

```
int BusquedaBinaria(Tipo_base vector[], int tvector, Tipo_base elemento)
{
    int alto,bajo,centro;
    alto=tvector-1;
    bajo=0;
    centro=(alto+bajo)/2;
    while( bajo<=alto && vector[centro]!=elemento ){
        if(elemento < vector[centro])
            alto=centro-1;
        else
            bajo=centro+1;
        centro=(alto+bajo)/2;
    }
    if( elemento==vector[centro]) return(centro);
    else                          return(-1);
}
```

El algoritmo utiliza tres variables: *alto* que representa el extremo superior del intervalo, *bajo* que representa el extremo inferior y *centro* que representa la posición media del intervalo. Al principio se inicializan dichas variables tomando el intervalo completo (de cero a *tvector-1*). El bucle principal termina cuando encontramos el elemento en la posición central o bien cuando el intervalo sea vacío (*alto<bajo*). En cada iteración se comprueba si el elemento que buscamos está por encima o por debajo del elemento central del intervalo actual y se modifican los valores del mismo en consecuencia. Si el elemento a buscar es menor al elemento central colocamos el extremo superior del rango de búsqueda a la izquierda del centro y si es mayor colocamos el extremo inferior a la derecha.

Utilizando esta estrategia conseguimos que, en el mejor de los casos (cuando el elemento se encuentra directamente en la posición central) se realice una única iteración y en el peor de los casos (cuando el elemento no se encuentre en el vector) se realicen $\log_2(tvector)$ iteraciones.

Veamos un ejemplo de búsqueda utilizando dicho algoritmo: a partir de un vector de 11 elementos de tipo entero vamos a buscar el valor 18 (representamos en negrita el elemento central y el rango entre el extremo superior y el inferior entre corchetes):

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
[2	3	8	9	10	**14**	15	18	20	33	110**]**
2	3	8	9	10	14	**[**15	18	**20**	33	110**]**
2	3	8	9	10	14	**[15**	18**]**	20	33	110
2	3	8	9	10	14	15	**[18]**	20	33	110

Como vemos, el algoritmo termina al encontrar el elemento. Veamos qué sucede al intentar buscar el elemento 7, el cual no se encuentra en el vector:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
[2	3	8	9	10	**14**	15	18	20	33	110**]**
[2	3	**8**	9	10**]**	14	15	18	20	33	110
[2	3**]**	8	9	10	14	15	18	20	33	110
2	**[3]**	8	9	10	14	15	18	20	33	110
2	**3]**	**[**8	9	10	14	15	18	20	33	110

En este caso el algoritmo finaliza al cumplirse la condición alto<bajo, rompiendo el bucle. Puesto que 3 (elemento central) no coincide con 7, la función devolvería –1.

Este procedimiento, como hemos dicho al principio, tiene sentido cuando el vector se encuentra ordenado. En el siguiente apartado veremos distintos algoritmos para ordenar un vector. Una vez ordenado, podremos realizar todas las operaciones de búsqueda sobre el mismo en tiempo logarítmico.

4 Ordenación en vectores

En ocasiones resulta necesario realizar ordenaciones de vectores. Por ejemplo, imaginemos un programa de gestión encargado de mantener un conjunto de fichas de clientes. Para listar los datos es posible que sea necesario ordenarlos por nombre y apellidos, o por DNI, etc. También puede darse el caso de que en nuestro programa sea necesario realizar muchas búsquedas, por ejemplo por DNI. Como hemos visto en el apartado anterior, la operación de buscar un elemento en un vector resulta mucho más rápida si éste se encuentra previamente ordenado.

En este apartado veremos algunos métodos utilizados frecuentemente para realizar la tarea de ordenar un vector. En todos los casos supondremos que queremos hacer una ordenación ascendente. Para realizar la operación contraria (ordenar descendentemente) los cambios a realizar en los algoritmos son mínimos.

En particular veremos tres algoritmos básicos: método del intercambio o burbuja, método de la inserción y método de la selección. Se trata de algoritmos in-situ, esto es, en los que la ordenación se realiza sobre el propio vector, sin necesidad de utilizar un vector auxiliar.

4.1 Método de intercambio o burbuja

Se trata de uno de los algoritmos más sencillos que existen de ordenación de vectores. Básicamente consiste en comparar dos a dos todos los elementos adyacentes del vector intercambiándolos si es necesario. El vector se recorre tantas veces como elementos tenga.

```
void OrdenacionBurbuja(Tipo_base vector[], int tam)
{
    Tipo_base aux;
    int i,j;
    for(i=0;i<tam;i++){
        for(j=1;j<tam;j++){
            if(vector[j-1]>vector[j]){
                aux=vector[j-1];
                vector[j-1]=vector[j];
                vector[j]=aux;
            }
        }
    }
    return;
}
```

El bucle más interior recorre todo el vector comparando los elementos adyacentes. Si dos elementos adyacentes no se encuentran ordenados se intercambian sus valores (swap). Este procedimiento se ha de repetir *tam* veces para asegurar el orden de todo el vector (bucle exterior).

Veamos un ejemplo en el que representamos la evolución del algoritmo para un determinado vector de enteros. Representamos el vector en cada iteración del bucle principal:

0	120	20	12	7	14	122	3	99	67
1	20	12	7	14	120	3	99	67	122
2	12	7	14	20	3	99	67	120	122
3	7	12	14	3	20	67	99	120	122
4	7	12	3	14	20	67	99	120	122
5	7	3	12	14	20	67	99	120	122
6	3	7	12	14	20	67	99	120	122
7	3	7	12	14	20	67	99	120	122
8	3	7	12	14	20	67	99	120	122

Como vemos, a partir de la iteración 6 no se produce ningún cambio en el vector, puesto que ya está ordenado. Por tanto, las iteraciones 7 y 8 sobrarían. En

general, cuanto mejor sea la ordenación inicial del vector más iteraciones sobrantes tendremos. Veamos otro ejemplo:

0	2	3	25	11	4	89	33	100	110
1	2	3	11	4	25	33	89	100	110
2	2	3	4	11	25	33	89	100	110
3	2	3	4	11	25	33	89	100	110
4	2	3	4	11	25	33	89	100	110
5	2	3	4	11	25	33	89	100	110
6	2	3	4	11	25	33	89	100	110
7	2	3	4	11	25	33	89	100	110
8	2	3	4	11	25	33	89	100	110

Como vemos, en este caso a partir de la iteración 2 no se realizan cambios sobre el vector. En este sentido, el algoritmo tanto en el mejor como en el peor de los casos realiza del orden de tam^2 iteraciones.

Existe la posibilidad de cambiar el algoritmo para que finalice en el mismo momento en el que el vector se encuentre ordenado, introduciendo un centinela en el que almacenamos si ha habido o no cambios en la última iteración:

```
void OrdenacionBurbujaMejorada(Tipo_base vector[], int tam)
{
    Tipo_base aux;
    int i,j,centinela=1;
    for(i=0;i<tam && centinela==1;i++){
        centinela=0;
        for(j=1;j<tam;j++){
            if(vector[j-1]>vector[j]){
                aux=vector[j-1];
                vector[j-1]=vector[j];
                vector[j]=aux;
                centinela=1;
            }
        }
    }
    return;
}
```

Como vemos, el bucle principal continúa mientras que el centinela contenga el valor 1 (hay cambios). En el bucle interior, cuando hacemos algún cambio ponemos el centinela a 1. Antes de comenzar el bucle interior dicho centinela vale 0, luego si no hay ningún cambio a lo largo de una iteración el bucle principal se cortará en la siguiente.

Aplicando este algoritmo sobre los ejemplos anteriores, terminaríamos en las iteraciones 6 y 2 respectivamente. En general, en el peor de los casos (el vector se encuentra inicialmente en orden inverso) el algoritmo seguirá realizando tam^2 iteraciones. Pero en el mejor de los casos (el vector se encuentra ya ordenado inicialmente) el algoritmo realizará *tam* iteraciones (1 vuelta del bucle principal).

4.2 Ordenación por inserción

En este apartado veremos un algoritmo más eficiente para la ordenación de vectores. Consiste en dividir el vector en dos partes: una ordenada y la otra no. En cada iteración se toma el primer elemento de la parte desordenada y se introduce en su posición correspondiente en la parte ordenada. Veamos el algoritmo:

```
void OrdenacionInsercion(Tipo_base vector[], int tam)
{
    Tipo_base aux;
    int i,j;
    for(i=1;i<tam;i++){
        aux=vector[i];
        j=i-1;
        while(aux<vector[j] && j>=0){
            vector[j+1]=vector[j];
            j--;
        }
        vector[j+1]=aux;
    }
    return;
}
```

Los elementos que están en el intervalo [*0,i*[representan la parte ordenada del vector en cada iteración. El resto, esto es, los que están en el intervalo [*i,tam-1*[representan la parte desordenada. En cada iteración se coge el elemento que hay en la posición *i* (primer elemento desordenado) y se coloca en su posición correspondiente en la parte ordenada. Como vemos, al principio la parte ordenada del vector consta de un único elemento (el que está en la posición 0) y el resto se considera desordenado. En el mejor de los casos (el vector se encontraba originalmente ordenado) el algoritmo realiza *tam* iteraciones. En el peor de los casos (el vector se encontraba en orden inverso) realizaremos tam^2 iteraciones. Este algoritmo realiza menos operaciones que el anterior puesto que simplemente "mueve" elementos, no realiza el intercambio completo de las posiciones luego, en términos generales, resulta más rápido.

Veamos cómo se comporta este nuevo algoritmo con los mismos ejemplos anteriores (representamos en negrita la lista actual ordenada. El resto representa la lista desordenada):

inicio	**120**	20	12	7	14	122	3	99	67
1	**20**	**120**	12	7	14	122	3	99	67
2	**12**	**20**	**120**	7	14	122	3	99	67
3	**7**	**12**	**20**	**120**	14	122	3	99	67
4	**7**	**12**	**14**	**20**	**120**	122	3	99	67
5	**7**	**12**	**14**	**20**	**120**	**122**	3	99	67
6	**3**	**7**	**12**	**14**	**20**	**120**	**122**	99	67
7	**3**	**7**	**12**	**14**	**20**	**99**	**120**	**122**	67
8	**3**	**7**	**12**	**14**	**20**	**67**	**99**	**120**	**122**

Veamos cómo se comporta el segundo ejemplo:

inicio	**2**	3	25	11	4	89	33	100	110
1	**2**	**3**	25	11	4	89	33	100	110
2	**2**	**3**	**25**	11	4	89	33	100	110
3	**2**	**3**	**11**	**25**	4	89	33	100	110
4	**2**	**3**	**4**	**11**	**25**	89	33	100	110
5	**2**	**3**	**4**	**11**	**25**	**89**	33	100	110
6	**2**	**3**	**4**	**11**	**25**	**33**	**89**	100	110
7	**2**	**3**	**4**	**11**	**25**	**33**	**89**	**100**	110
8	**2**	**3**	**4**	**11**	**25**	**33**	**89**	**100**	**110**

En ambos ejemplos se puede observar cómo el siguiente elemento desordenado se coloca en su posición correspondiente en cada iteración.

4.3 Ordenación por selección

Este método es una interesante variante del anterior. Partimos del mismo esquema, esto es, una lista ordenada y otra desordenada. Lo que vamos a hacer ahora es elegir el elemento más pequeño de la lista desordenada para colocarlo en la lista ordenada. Como es evidente, el elemento más pequeño de la lista desordenada se colocará en la posición más alta de la lista ordenada porque, a su vez es el mayor de ésta. Veamos el algoritmo:

```
void OrdenacionSeleccion(Tipo_base vector[], int tam)
{
    int i,j,pos_menor;
    Tipo_base aux;
    for(i=0;i<tam;i++){
            pos_menor=i;
            for(j=i+1;j<tam;j++){
                    if(vector[j]< vector[pos_menor])
                            pos_menor=j;
            }

            aux=vector[i];
            vector[i]= vector[pos_menor];
            vector[pos_menor]=aux;
    }
    return;
}
```

Como vemos, se selecciona el elemento menor de la lista desordenada y se sustituye por el siguiente elemento, con lo que nos ahorramos mover todos los elementos intermedios una posición. Esto provoca que el algoritmo sea mucho más rápido que el anterior. Veamos cómo se comportan los ejemplos anteriores utilizando este algoritmo:

Inic.	120	20	12	7	14	122	3	99	67
1	**3**	**20**	12	7	14	122	120	99	67
2	**3**	**7**	**12**	20	14	122	120	99	67
3	**3**	**7**	**12**	**20**	14	122	120	99	67
4	**3**	**7**	**12**	**14**	**20**	122	120	99	67

5	**3**	**7**	**12**	**14**	**20**	**122**	120	99	67
6	**3**	**7**	**12**	**14**	**20**	**67**	**120**	99	122
7	**3**	**7**	**12**	**14**	**20**	**67**	**99**	**120**	122
8	**3**	**7**	**12**	**14**	**20**	**67**	**99**	**120**	**122**

La evolución del segundo ejemplo sería:

Inic.	2	3	25	11	4	89	33	100	110
1	**2**	**3**	25	11	4	89	33	100	110
2	**2**	**3**	**25**	11	4	89	33	100	110
3	**2**	**3**	**4**	**11**	25	89	33	100	110
4	**2**	**3**	**4**	**11**	**25**	89	33	100	110
5	**2**	**3**	**4**	**11**	**25**	**89**	33	100	110
6	**2**	**3**	**4**	**11**	**25**	**33**	**89**	100	110
7	**2**	**3**	**4**	**11**	**25**	**33**	**89**	**100**	110
8	**2**	**3**	**4**	**11**	**25**	**33**	**89**	**100**	**110**

En cada momento se selecciona el menor y se sustituye por el último elemento de la lista ordenada, ahorrándonos con ello el movimiento del resto de los elementos.

5 Registros

Hasta ahora los tipos de datos compuestos se constituyen por elementos del mismo tipo. En este apartado veremos los registros, que son tipos de datos compuestos que pueden albergar elementos de distinto tipo.

Para utilizar una variable registro es preciso especificar primero la forma de dicho registro. O sea, declararemos un tipo de datos definido por nosotros mismos con la forma que deseamos.

La sintaxis es la siguiente:

```
typedef struct{
    //Definición de las variables del registro;
    Tipo1 var11,var21..var1N;
    Tipo2 var21,var22..var1K;
    ….
    TipoR varR1,varR2..varRI;
}NombreTipoRegistro;
```

Una vez definido el tipo de datos podemos utilizarlo para declarar una variable de ese tipo. Dentro de dicha variable vendrán encapsulados los campos de los que consta el registro.

Para acceder a un campo de una variable registro especificaremos el nombre de la variable y el nombre del campo separados por un punto. Veamos un ejemplo de utilización de registros en el que nos definimos un tipo de datos que representa un vector en 3D (x,y,z). En el programa se utiliza dicho tipo para calcular la suma vectorial:

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

typedef struct{
	float x;
	float y;
	float z;
}TVector3D;

TVector3D
SumaVectorial(TVector3D v1, TVector3D v2){
	TVector3D resul;
	resul.x=v1.x+v2.x;
	resul.y=v1.y+v2.y;
	resul.z=v1.z+v2.z;
	return(resul);
}

void main(void)
{
	TVector3D a,b,r;
	printf("Componente X del vector a:"); scanf("%f",&a.x);
	printf("Componente Y del vector a:"); scanf("%f",&a.y);
	printf("Componente Z del vector a:"); scanf("%f",&a.z);
	printf("Componente X del vector b:"); scanf("%f",&b.x);
	printf("Componente Y del vector b:"); scanf("%f",&b.y);
	printf("Componente Z del vector b:"); scanf("%f",&b.z);
	r=SumaVectorial(a,b);
	printf("Resultado: (%f, %f, %f)\n",r.x,r.y,r.z);
	return;
}
```

En este ejemplo se puede observar como los distintos campos pertenecientes a la estructura se utilizan por separado tal y como si fuesen variables simples.

Vamos a construir una función sobre este mismo ejemplo para normalizar el vector. Para ello tendremos que pasar el vector por referencia:

```
void NormalizarVector(TVector3D *v)
{
	float aux, modulo;
	aux = (*v).x * (*v).x + (*v).y * (*v).y + (*v).z * (*v).z;
	modulo = sqrt(aux); //funcion raiz cuadrada

	(*v).x = (*v).x / modulo;
	(*v).y = (*v).y / modulo;
	(*v).z = (*v).z / modulo;
```

```
    return;
}
```

Para llamar a esta función para normalizar, por ejemplo, el vector *a* de la función main escribiríamos: *NormalizarVector(&a);*

Dentro de la función es necesario extraer el contenido de la estructura, y sobre dicho contenido extraer el campo. Para este caso en particular, esto es, cuando queremos extraer un campo de un puntero a una estructura podemos utilizar un operador similar, esto es, el operador flecha (->):

```
void NormalizarVector(TVector3D *v)
{
    float aux, modulo;
    aux = v->x * v->x + v->y * v->y + v->z * v->z;

    modulo = sqrt(aux); //funcion raiz cuadrada

    v->x = v->x / modulo;
    v->y = v->y / modulo;
    v->z = v->z / modulo;

    return;
}
```

Este operador es mucho más legible y cómodo.

Al igual que hemos definido una variable de tipo estructura podemos definir un vector o matriz de tipo estructura.

Veamos un ejemplo en el que vamos a almacenar una lista de fichas de clientes en un vector:

```
#include <stdio.h>
typedef struct{
    char nombre[50];
    char apellido1[50];
    char apellido2[50];
    int dni;
}TFicha;

TFicha
LeerFicha(void){
    TFicha f;
    printf("Nombre    :"); scanf("%s",f.nombre);
    printf("Apellido1 :"); scanf("%s",f.apellido1);
    printf("Apellido2 :"); scanf("%s",f.apellido2);
```

```
    printf("DNI      :"); scanf("%d",&f.dni);
    return(f);
}

void ImprimirFicha(TFicha f)
{
    printf("--------------\n");
    printf("Nombre    : %s\n",f.nombre);
    printf("Apellido1 : %s\n",f.apellido1);
    printf("Apellido2 : %s\n",f.apellido2);
    printf("DNI       : %d\n",f.dni);
    return;
}

void main(void)
{
    TFicha fichas[5];
    int i;
    for(i=0;i<5;i++)  fichas[i]=LeerFicha();
    printf("Las fichas introducidas son:\n");
    for(i=0;i<5;i++) ImprimirFicha(fichas[i]);
    return;
}
```

Existe una variante de la estructura llamada **union**, en la que es posible tener una misma variable que puede ser de varios tipos distintos, con el objeto de economizar memoria. En este curso no estudiaremos esta variante por razones de tiempo.

6 Memoria dinámica

Hasta ahora, hemos realizado una administración de memoria de tipo estático. Esto significa que los vectores y matrices creados tienen un tamaño fijo que el compilador conoce en **tiempo de compilación**. Sin embargo, es posible que el tamaño de dichos vectores no se conozca a priori y por lo tanto necesitemos solicitar memoria al sistema en **tiempo de ejecución**. La administración de memoria desde este punto de vista recibe el nombre de **memoria dinámica**.

En este apartado veremos cómo utilizar la memoria dinámica para el tratamiento de vectores y matrices y veremos, aunque sin profundizar en ellas, otras estructuras más avanzadas tales como listas, colas y pilas.

6.1 Vectores dinámicos

El tratamiento de memoria dinámica sigue tres pasos fundamentales:

1) Petición de memoria (función **malloc**)

2) Utilización de dicha memoria para nuestro propósito
3) Liberación de memoria (función **free**)

Antes de poder utilizar la memoria que deseemos tenemos que solicitarla al sistema mediante la función **malloc**. Tras su utilización y con el fin de no consumir recursos innecesariamente, tendremos que liberar dicha memoria utilizando la función **free**.

La función malloc posee la siguiente sintaxis:

[direcc_inicio] malloc([tamaño en bytes]);

La función recibe el tamaño de la memoria solicitada en bytes y devuelve la dirección de inicio de la misma. A partir de este momento podremos utilizar dicha memoria tal y como si fuese estática.

Para especificar la cantidad de memoria que necesitamos se suele utilizar la función **sizeof**. Esta función devuelve un entero con el número de bytes de los que consta un tipo definido en el computador actual:

[tamaño en bytes] sizeof([tipo definido]);

Combinando estas funciones podremos reservar memoria para vectores, matrices, etc.

Para liberar memoria asignada utilizaremos la función **free**, especificando la dirección de inicio de la misma. Su sintaxis es la siguiente:

free([direcc_inicio]);

A partir de la ejecución de esta instrucción, la memoria que habíamos reservado podrá ser utilizada por otros procesos del sistema.

Para crear y manejar vectores unidimensionales dinámicos necesitaremos almacenar la dirección de memoria de comienzo del vector en una variable de tipo **puntero**. Para crear una variable de tipo puntero solamente es necesario anteponer a su nombre el operador *. El tipo de dicho puntero será el tipo base de los elementos del vector. Para acceder a un elemento del vector lo haremos con corchetes tal y como si fuese un vector estático.

Veamos un ejemplo en el que solicitamos al usuario una cierta cantidad de números para después mostrarlos por pantalla en el mismo orden en el que se introdujeron. Al principio del programa solicitaremos al usuario cuántos números desea introducir:

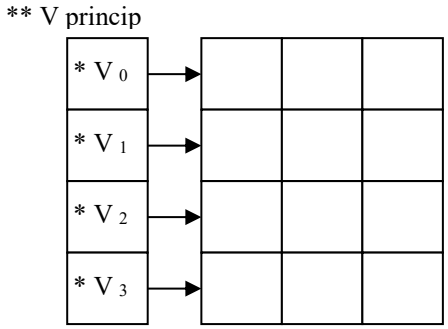
```
#include <stdio.h>
void main(void)
{
    int *vector;
    int n,i;
    printf("Elementos a introducir:"); scanf("%d",&n);
    vector=malloc(sizeof(int)*n); //peticion
```

```
        for(i=0;i<n;i++){
                printf("Elemento %d:",i);
                scanf("%d",&(vector[i]));
        }
        for(i=0;i<n;i++){
                printf("Elemento %d = %d\n",i,vector[i]);
        }
        free(vector); //liberacion
        return;
}
```

Como vemos, para almacenar el vector creamos un puntero de tipo entero. Para solicitar memoria para las *n* posiciones del vector llamamos a la función malloc solicitando tantos bytes como los que tenga un entero multiplicados por *n*. La función malloc nos devuelve la dirección de memoria de comienzo que nos almacenamos en *vector*. Una vez utilizado el vector liberamos su memoria asignada especificando la dirección de inicio de esta (*vector*). El resto del tratamiento se realiza como si de un vector estático se tratase.

6.2 Matrices dinámicas

El problema del tratamiento de matrices dinámicas es similar al del tratamiento de vectores, teniendo en cuenta que una matriz es un **vector de vectores**. En ese sentido, será necesario crear un vector principal donde cada componente del mismo es un puntero a un vector. Para ello, necesitamos reservar memoria para el vector principal y después para cada uno de los vectores que contiene. Esto es lo que se denomina **doble puntero**:



Veamos el mismo ejemplo anterior pero ahora con una matriz bidimensional. Para ello, solicitamos el número de filas y columnas de la matriz, reservamos memoria para la misma y preguntamos cada valor para después imprimirlos todos. Finalmente liberamos la memoria asignada a la matriz:

```
#include <stdio.h>
void main(void)
{
    int **matriz;
```

```
    int nfil,ncol,i,j;
    printf("Filas:"); scanf("%d",&nfil);
    printf("Columnas:"); scanf("%d",&ncol);
    //reserva de memoria
    matriz=malloc(sizeof(int*)*nfil);
    for(i=0;i<nfil;i++){
        matriz[i]=malloc(sizeof(int)*ncol);
    }
    //Preguntamos los valores
    for(i=0;i<nfil;i++){
        for(j=0;j<ncol;j++){
            printf("Elemento (%d,%d):",i,j);
            scanf("%d",&(matriz[i][j]));
        }
    }
    //Imprimimos los valores
    for(i=0;i<nfil;i++){
        for(j=0;j<ncol;j++){
            printf("Elemento (%d,%d) = %d\n",i,j,matriz[i][j]);
        }
    }
    //liberamos la memoria
    for(i=0;i<nfil;i++){
        free(matriz[i]);
    }
    free(matriz);
    return;
}
```

Como vemos, a excepción de la reserva y la liberación de memoria, el resto del tratamiento de una matriz bidimensional dinámica se realiza tal y como si fuese de tipo estático.

6.3 Tipos de datos avanzados

Combinando la definición de estructuras, memoria dinámica y punteros es posible construir tipos de datos más avanzados. Aunque no nos es posible profundizar en este campo, resulta conveniente conocer su existencia.

Veamos, por ejemplo, la siguiente estructura:

```
typedef struct x{
    char nombre[255];
    int dni;
    struct x *siguiente;
}TFicha;
```

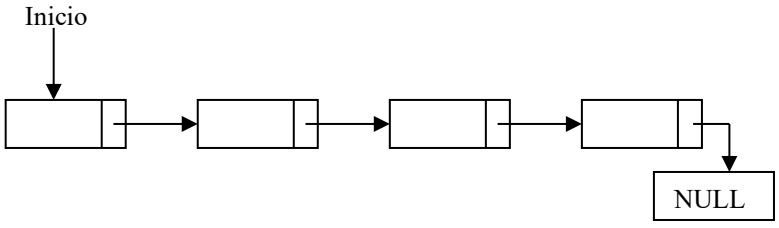
Como vemos, la estructura podría corresponderse con una ficha de clientes, de los que almacenamos el nombre y el dni. La estructura cuenta con un puntero a una estructura del mismo tipo (x es un nombre alternativo para esta estructura). De esta forma, imaginemos que en nuestro programa solicitamos memoria para dos estructuras de este tipo:

```
….
TFicha *ficha1,*ficha2;

ficha1=malloc(sizeof(TFicha));
strcpy(ficha1->nombre,"Juan Manuel");
ficha1->dni=33884499;

ficha2=malloc(sizeof(TFicha));
strcpy(ficha2->nombre,"Miguel Angel");
ficha2->dni=88337722;

ficha1->siguiente = ficha2;
ficha2->siguiente = NULL;
….
```

Como vemos, al final hacemos coincidir el campo *siguiente* perteneciente a *ficha1* con la dirección de memoria de *ficha2* y el campo *siguiente* de *ficha2* con la dirección de memoria vacía (NULL). Esta es una forma de enlazar los datos en forma de **lista dinámica**.

Esta estructura tiene la ventaja de que, en cualquier momento, podemos añadir un dato a la misma creando una nueva estructura y apuntando su puntero al primer elemento de la lista, o bien colocándolo al final de ésta.

En todo momento tenemos que guardarnos un puntero con el primer elemento de la lista:



Existen variantes del modelo dependiendo del tratamiento que les demos:

- **Colas (FIFO=First In First Out)**: La inserción de elementos se realiza por un extremo de la lista y el borrado por el otro. El primer elemento que se introduce será el primero en eliminarse.
- **Pilas (LIFO=Last In First Out)**: Insertamos y borramos los elementos por el mismo extremo de la lista, de forma que el último elemento insertado es el primero que se borra.

Dependiendo de la conectividad de las listas podemos tener:

- **Listas de enlace simple**: Cada elemento de la lista apunta al siguiente o al anterior
- **Listas doblemente enlazadas**: Cada elemento de la lista apunta al elemento siguiente y al anterior.