CAPÍTULO 7

Estructuras de datos I (arrays y estructuras)[1]

7.1. Introducción a las estructuras de datos
7.2. Arrays (arreglos) unidimensionales: los vectores
7.3. Operaciones con vectores
7.4. Arrays de varias dimensiones
7.5. Arrays multidimensionales
7.6. Almacenamiento de arrays en memoria
7.7. Estructuras *versus* registros
7.8. Arrays de estructuras
7.9. Uniones
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS
CONCEPTOS CLAVE
RESUMEN
EJERCICIOS

INTRODUCCIÓN

En los capítulos anteriores se ha introducido el concepto de datos de tipo simple que representan valores de tipo simple, como un número entero, real o un carácter. En muchas situaciones se necesita, sin embargo, procesar una colección de valores que están relacionados entre sí por algún método, por ejemplo, una lista de calificaciones, una serie de temperaturas medidas a lo largo de un mes, etc. El procesamiento de tales conjuntos de datos, utilizando datos simples, puede ser extremadamente difícil y por ello la mayoría de los lenguajes de programación incluyen características de estructuras de datos. *Las estructuras de datos básicas* que soportan la mayoría de los lenguajes de programación son los "arrays" —concepto matemático de "vector" y "matriz"—.

Un **array,** o **arreglo** *en Latinoamérica*, es una secuencia de posiciones de la memoria central a las que se puede acceder directamente, que contiene datos del mismo tipo y pueden ser seleccionados individualmente mediante el uso de subíndices. Este capítulo estudia el concepto de *arrays unidimensionales y multidimensionales*, así como el procesamiento de los mismos.

[1] El término **array** se conserva en inglés por su amplia aceptación en la comunidad de ingeniería informática y de sistemas. Sin embargo, es preciso constatar que en prácticamente toda Latinoamérica (al menos en muchos de los numerosos países que conocemos y con los que tenemos relaciones académicas y personales) el término empleado como traducción es **arreglo**. El **DRAE** (última edición, 22.ª, Madrid 2001) no considera ninguno de los dos términos como válidos, aunque la acepción 2 de la definición de **arreglo** pudiera ser ilustrativa del porqué de la adopción del término por la comunidad latinoamericana: "**Regla**, **orden**, **coordinación**".

7.1. INTRODUCCIÓN A LAS ESTRUCTURAS DE DATOS

Una *estructura de datos* es una colección de datos que pueden ser caracterizados por su organización y las operaciones que se definen en ella.

Las estructuras de datos son muy importantes en los sistemas de computadora. Los tipos de datos más frecuentes utilizados en los diferentes lenguajes de programación son:

datos simples	*estándar*	entero *(integer)* real *(real)* carácter *(char)* lógico *(boolean)*
	definido por el programador *(no estándar)*	subrango *(subrange)* enumerativo *(enumerated)*
datos estructurados	*estáticos*	*arrays (vectores/matrices)* registros *(record)* ficheros *(archivos)* conjuntos *(set)* cadenas *(string)*
	dinámicos	listas *(pilas/colas)* listas enlazadas árboles grafos

Los tipos de datos *simples* o *primitivos* significan que no están compuestos de otras estructuras de datos; los más frecuentes y utilizados por casi todos los lenguajes son: *enteros, reales* y *carácter (char)*, siendo los tipos *lógicos, subrango* y *enumerativos* propios de lenguajes estructurados como Pascal. Los tipos de datos compuestos están construidos basados en tipos de datos primitivos; el ejemplo más representativo es la *cadena (string)* de caracteres.

Los tipos de datos simples pueden ser organizados en diferentes estructuras de datos: *estáticas* y *dinámicas*. Las **estructuras de datos estáticas** son aquellas en las que el tamaño ocupado en memoria se define antes de que el programa se ejecute y no puede modificarse dicho tamaño durante la ejecución del programa. Estas estructuras están implementadas en casi todos los lenguajes: *array* (vectores/tablas-matrices), *registros, ficheros* o *archivos* (los *conjuntos* son específicos del lenguaje Pascal). Las **estructuras de datos dinámicas** no tienen las limitaciones o restricciones en el tamaño de memoria ocupada que son propias de las estructuras estáticas. Mediante el uso de un tipo de datos específico, denominado *puntero*, es posible construir estructuras de datos dinámicas que son soportadas por la mayoría de los lenguajes que ofrecen soluciones eficaces y efectivas en la solución de problemas complejos —Pascal es el lenguaje tipo por excelencia con posibilidad de estructuras de datos dinámicos—. Las estructuras dinámicas por excelencia son las *listas* —enlazadas, pilas, colas—, *árboles* —binarios, árbol-b, búsqueda binaria— y *grafos*.

La elección del tipo de estructura de datos idónea a cada aplicación dependerá esencialmente del tipo de aplicación y, en menor medida, del lenguaje, ya que en aquellos en que no está implementada una estructura —por ejemplo, las listas y árboles no los soporta COBOL— deberá ser simulada con el algoritmo adecuado, dependiendo del propio algoritmo y de las características del lenguaje su fácil o difícil solución.

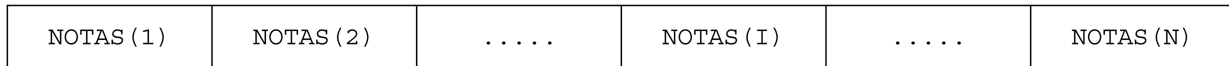
Una característica importante que diferencia a los tipos de datos es la siguiente: los tipos de datos simples tienen como característica común que cada variable representa a un elemento; los tipos de datos estructurados tienen como característica común que un *identificador* (nombre) puede representar múltiples datos individuales, pudiendo cada uno de éstos ser referenciado independientemente.

7.2. ARRAYS (ARREGLOS) UNIDIMENSIONALES: LOS VECTORES

Un *array* o *arreglo (matriz* o *vector)* es un conjunto finito y ordenado de elementos homogéneos. La propiedad *"ordenado"* significa que el elemento primero, segundo, tercero, ..., enésimo de un *array* puede ser identificado. Los

elementos de un *array* son homogéneos, es decir, del mismo tipo de datos. Un *array* puede estar compuesto de todos sus elementos de tipo cadena, otro puede tener todos sus elementos de tipo entero, etc. Los *arrays* se conocen también como *matrices* —en matemáticas— y *tablas* —en cálculos financieros—.

El tipo más simple de *array* es el *array unidimensional o vector* (matriz de una dimensión). Un vector de una dimensión denominado `NOTAS` que consta de *n* elementos se puede representar por la Figura 7.1.



NOTAS(1)	NOTAS(2)	.....	NOTAS(I)	.....	NOTAS(N)

Figura 7.1. Vector.

El *subíndice o índice* de un elemento (1, 2, ..., *i*, *n*) designa su posición en la ordenación del vector. Otras posibles notaciones del vector son:

```
a1, a2, ..., ai, ..., an                en matemáticas y algunos lenguajes (VB 6.0 y VB.Net)
A(1), A(2), ..., A(i), ..., A(n)
A[1], A[2], ..., A[i], ..., A[n]        en programación (Pascal y C)
```

Obsérvese que sólo el vector global tiene nombre `(NOTAS)`. Los elementos del vector se referencian por su *subíndice* o índice *("subscript")*, es decir, su posición relativa en el valor.

En algunos libros y tratados de programación, además de las notaciones anteriores, se suele utilizar esta otra:

```
A(L:U) = {A(I)}
para I = L, L+1, ..., U-1, U            donde cada elemento A(I) es de tipo de datos T
```

que significa: `A`, vector unidimensional con elementos de datos tipo `T`, cuyos subíndices varían en el rango de `L` a `U`, lo cual significa que el índice no tiene por qué comenzar necesariamente en 0 o en 1.

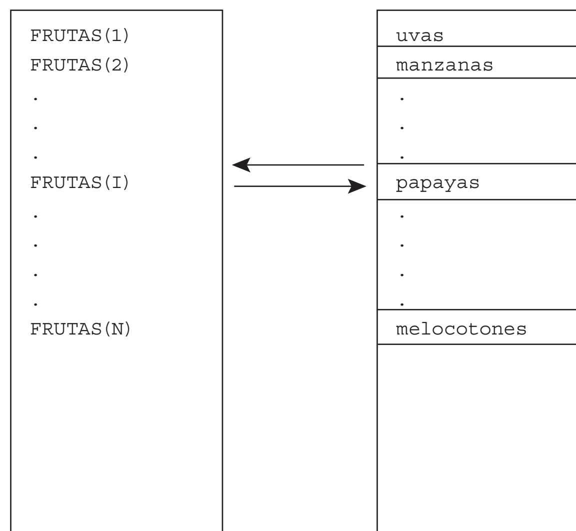
Como ejemplo de un vector o *array* unidimensional, se puede considerar el vector `TEMPERATURA` que contiene las temperaturas horarias registradas en una ciudad durante las veinticuatro horas del día. Este vector constará de veinticuatro elementos de tipo real, ya que las temperaturas normalmente no serán enteras siempre.

El valor mínimo permitido de un vector se denomina *límite inferior* del vector (`L`) y el valor máximo permitido se denomina *límite superior* (`U`). En el ejemplo del vector `TEMPERATURAS` el límite inferior es `1` y el superior `24`.

`TEMPERATURAS(I)` donde `1 <= I <= 24`

El número de elementos de un vector se denomina *rango del vector*. El rango del vector `A(L:U)` es `U-L+1`. El rango del vector `B(1:n)` es `n`.

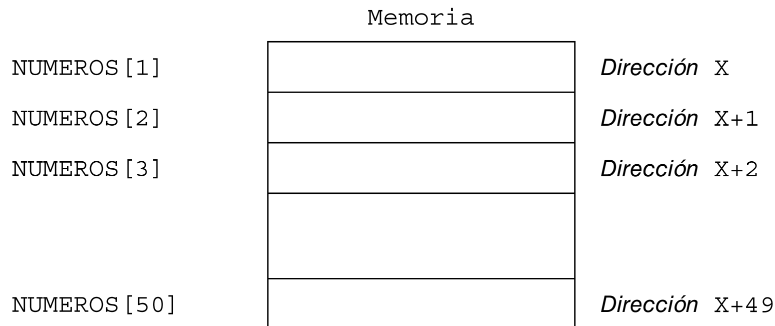
Los vectores, como ya se ha comentado, pueden contener datos no numéricos, es decir, tipo "carácter". Por ejemplo, un vector que representa las frutas que se venden en un supermercado:



Otro ejemplo de un vector pueden ser los nombres de los alumnos de una clase. El vector se denomina ALUMNOS y tiene treinta elementos de rango.

	ALUMNOS
1	Luis Francisco
2	Jose
3	Victoria
	.
i	Martin
	.
30	Graciela

Los vectores se almacenan en la memoria central de la computadora en un orden adyacente. Así, un vector de cincuenta números denominado NUMEROS se representa gráficamente por cincuenta posiciones de memoria sucesivas.



Cada elemento de un vector se puede procesar como si fuese una variable simple al ocupar una posición de memoria. Así,

```
NUMEROS[25] ← 72
```

almacena el valor entero o real 72 en la posición 25.ª del vector NUMEROS y la instrucción de salida

```
escribir (NUMEROS[25])
```

visualiza el valor almacenado en la posición 25.ª, en este caso 72.

Esta propiedad significa que cada elemento de un vector —y posteriormente una tabla o matriz— es accesible directamente y es una de las *ventajas* más importantes de usar un vector: *almacenar un conjunto de datos*.

Consideremos un vector X de ocho elementos

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
14.0	12.0	8.0	7.0	6.41	5.23	6.15	7.25
Elemento 1.º	Elemento 2.º						Elemento 8.º

Algunas instrucciones que manipulan este vector se representan en la Tabla 7.1.

Tabla 7.1. Operaciones básicas con vectores

Acciones	**Resultados**
escribir(X[1])	Visualiza el valor de X[1] o 14.0.
X[4] ← 45	Almacena el valor 45 en X[4].
SUMA ← X[1]+X[3]	Almacena la suma de X[1] y x[3] o bien 22.0 en la variable SUMA.
SUMA ← SUMA+X[4]	Añade en la variable SUMA el valor de X[4], es decir, SUMA = 67.0.
X[5] ← x[5]+3,5	Suma 3.5 a X[5]; el nuevo valor de X[5] será 9.91.
X[6] ← X[1]+X[2]	Almacena la suma de X[1] y X[2] en X[6]; el nuevo valor de X[6] será 26.5.

Antes de pasar a tratar las diversas operaciones que se pueden efectuar con vectores, consideremos la notación de los diferentes elementos.

Supongamos un vector V de ocho elementos.

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]
12	5	-7	14.5	20	1.5	2.5	-10

Los subíndices de un vector pueden ser enteros, variables o expresiones enteras. Así, por ejemplo, si

I ← 4
V[I+1] representa el elemento V(5) de valor 20
V[I+2] representa el elemento V(6) de valor 1.5
V[I-2] representa el elemento V(2) de valor 5
V[I+3] representa el elemento V(7) de valor 2.5

Los *arrays* unidimensionales, al igual que posteriormente se verán los *arrays* multidimensionales, necesitan ser dimensionados previamente a su uso dentro de un programa.

7.3. OPERACIONES CON VECTORES

Un vector, como ya se ha mencionado, es una secuencia ordenada de elementos como

X[1], X[2], ..., X[n]

El límite inferior no tiene por qué empezar en uno. El vector L

L[0], L[1], L[2], L[3], L[4], L[5]

contiene seis elementos, en el que el primer elemento comienza en cero. El vector P, cuyo rango es 7 y sus límites inferior y superior son –3 y 3, es

P[-3], P[-2], P[-1], P[0], P[1], P[2], P[3]

Las operaciones que se pueden realizar con vectores durante el proceso de resolución de un problema son:

- *asignación,*
- *lectura/escritura,*

- *recorrido* (acceso secuencial),
- *actualizar* (añadir, borrar, insertar),
- *ordenación,*
- *búsqueda.*

En general, las operaciones con vectores implican el procesamiento o tratamiento de los elementos individuales del vector.

Las notaciones algorítmicas que utilizaremos en este libro son:

```
tipo
  array [liminf .. limsup] de tipo : nombre_array
```

`nombre_array`	nombre válido del *array*
`liminf..limsup`	límites inferior y superior del rango del *array*
`tipo`	tipo de datos de los elementos del *array*: entero, real, carácter

```
tipo
  array[1..10] de carácter : NOMBRES
var
  NOMBRES : N
```

significa que `NOMBRES` es un array (vector) unidimensional de diez elementos (1 a 10) de tipo carácter.

```
tipo
  array['A'..'Z'] de real : LISTA
var
  LISTA : L
```

representa un vector cuyos subíndices son `A`, `B`, `...` y cuyos elementos son de tipo real.

```
tipo
  array[0..100] de entero : NUMERO
var
  NUMERO:NU
```

`NUMERO` es un vector cuyos subíndices van de 0 a 100 y de tipo entero.

Las operaciones que analizaremos en esta sección serán: *asignación, lectura/escritura, recorrido* y *actualización*, dejando por su especial relevancia como tema exclusivo de un capítulo la *ordenación* o *clasificación* y *búsqueda*.

7.3.1. Asignación

La asignación de valores a un elemento del vector se realizará con la instrucción de asignación:

`A[29] ← 5` *asigna el valor* `5` *al elemento* `20` *del vector* `A`

Si se desea asignar valores a todos los elementos de un vector, se debe recurrir a estructuras repetitivas (**desde, mientras** o **repetir**) e incluso selectivas (**si-entonces, segun**).

```
leer(A[i])
```

Si se introducen los valores 5, 7, 8, 14 y 12 mediante asignaciones

```
A[1] ← 5
A[2] ← 7
```

```
A[3] ← 8
A[4] ← 14
A[5] ← 12
```

El ejemplo anterior ha asignado diferentes valores a cada elemento del vector `A`; si se desea dar el mismo valor a todos los elementos, la notación algorítmica se simplifica con el formato.

```
desde i = 1 hasta 5 hacer
  A[i] ← 8
fin_desde
```

donde `A[i]` tomará los valores numéricos

```
A[1] = 8, A[2] = 8, ..., A[5] = 8
```

Se puede utilizar también la notación

```
A ← 8
```

para indicar la asignación de un mismo valor a cada elemento de un vector `A`. Esta notación se considerará con mucho cuidado para evitar confusión con posibles variables simples numéricas de igual nombre (`A`).

7.3.2. Lectura/escritura de datos

La lectura/escritura de datos en *arrays* u operaciones de entrada/salida normalmente se realizan con estructuras repetitivas, aunque puede también hacerse con estructuras selectivas. Las instrucciones simples de lectura/escritura se representarán como

leer(V[5]) leer el elemento `V[5]` del vector `V`

7.3.3. Acceso secuencial al vector (recorrido)

Se puede acceder a los elementos de un vector para introducir datos *(escribir)* en él o bien para visualizar su contenido *(leer)*. A la operación de efectuar una acción general sobre todos los elementos de un vector se la denomina *recorrido* del vector. Estas operaciones se realizan utilizando estructuras repetitivas, cuyas variables de control (por ejemplo, `I`) se utilizan como subíndices del vector (por ejemplo, `S[I]`). El incremento del contador del bucle producirá el tratamiento sucesivo de los elementos del vector.

EJEMPLO 7.1

Lectura de veinte valores enteros de un vector denominado `F`.

Procedimiento 1

```
algoritmo leer_vector
tipo
  array[1..20] de entero : FINAL
var
  FINAL : F
inicio
  desde i ← 1 hasta 20 hacer
    leer(F[i])
  fin_desde
fin
```

La lectura de veinte valores sucesivos desde el teclado rellenará de valores el vector `F`, comenzando con el elemento `F[1]` y terminando en `F[20]`. Si se cambian los límites inferior y superior (por ejemplo, 5 y 10), el bucle de lectura sería

```
desde i ← 5 hasta 10 hacer
  leer(F[i])
fin_desde
```

Procedimiento 2

Los elementos del vector se pueden leer también con bucles **mientras** o **repetir**.

i ← 1
mientras i <= 20 **hacer**
  leer(F[i])
  i ← i + 1
fin_mientras
	o bien	i ← 1
		repetir
		  leer (F[i])
		  i ← i + 1
		hasta_que i > 20

La salida o escritura de vectores se representa de un modo similar. La estructura

```
desde i ← 1 hasta i ← 20 hacer
  escribir(F[i])
fin_desde
```

visualiza todo el vector completo (un elemento en cada línea independiente).

Ejemplo 7.2

Este ejemplo procesa un array `PUNTOS`*, realizando las siguientes operaciones; a) lectura del array, b) cálculo de la suma de los valores del array, c) cálculo de la media de los valores.*

El *array* lo denominaremos `PUNTOS`; el límite superior del rango lo introduciremos por teclado y el límite inferior lo consideraremos 1.

```
algoritmo media_puntos
const
  LIMITE = 40
tipo
  array[1..LIMITE] de real : PUNTUACION
var
  PUNTUACION : PUNTOS
  real : suma, media
  entero : i
inicio
  suma ← 0
  escribir('Datos del array')
  desde i ← 1 hasta LIMITE hacer
    leer(PUNTOS[i])
    suma ← suma + PUNTOS[i]
  fin_desde
  media ← suma / LIMITE
  escribir('La media es', media)
fin
```

Se podría ampliar el ejemplo, en el sentido de visualizar los elementos del *array*, cuyo valor es superior a la media. Mediante una estructura **desde** se podría realizar la operación, añadiéndole al algoritmo anterior.

```
escribir('Elementos del array superior a la media')
desde i ← 1 hasta LIMITE hacer
  si PUNTOS[i] > media entonces
    escribir(PUNTOS[i])
  fin_si
fin_desde
```

EJEMPLO 7.3

Calcular la media de las estaturas de una clase. Deducir cuántos son más altos que la media y cuántos son más bajos que dicha media (véase Figura 7.2).

Solución

Tabla de variables

n	número de estudiantes de la clase	: entera
H[1]...H[n]	estatura de los n alumnos	: real
i	contador de alumnos	: entera
MEDIA	media de estaturas	: real
ALTOS	alumnos de estatura mayor que la media	: entera
BAJOS	alumnos de estatura menor que la media	: entera
SUMA	totalizador de estaturas	: real

7.3.4. Actualización de un vector

La operación de actualizar un vector puede constar a su vez de tres operaciones elementales:

añadir elementos
insertar elementos
borrar elementos

Se denomina *añadir datos* a un vector la operación de añadir un nuevo elemento al final del vector. La única condición necesaria para esta operación consistirá en la comprobación de espacio de memoria suficiente para el nuevo vector; dicho de otro modo, que el vector no contenga todos los elementos con que fue definido al principio del programa.

EJEMPLO 7.4

Un array TOTAL *se ha dimensionado a seis elementos, pero sólo se le han asignado cuatro valores a los elementos* TOTAL[1], TOTAL[2], TOTAL[3] *y* TOTAL[4]. *Se podrán añadir dos elementos más con una simple acción de asignación.*

```
TOTAL[5] ← 14
TOTAL[6] ← 12
```

La *operación de insertar un elemento* consiste en introducir dicho elemento en el interior del vector. En este caso se necesita un desplazamiento previo hacia abajo para colocar el elemento nuevo en su posición relativa.

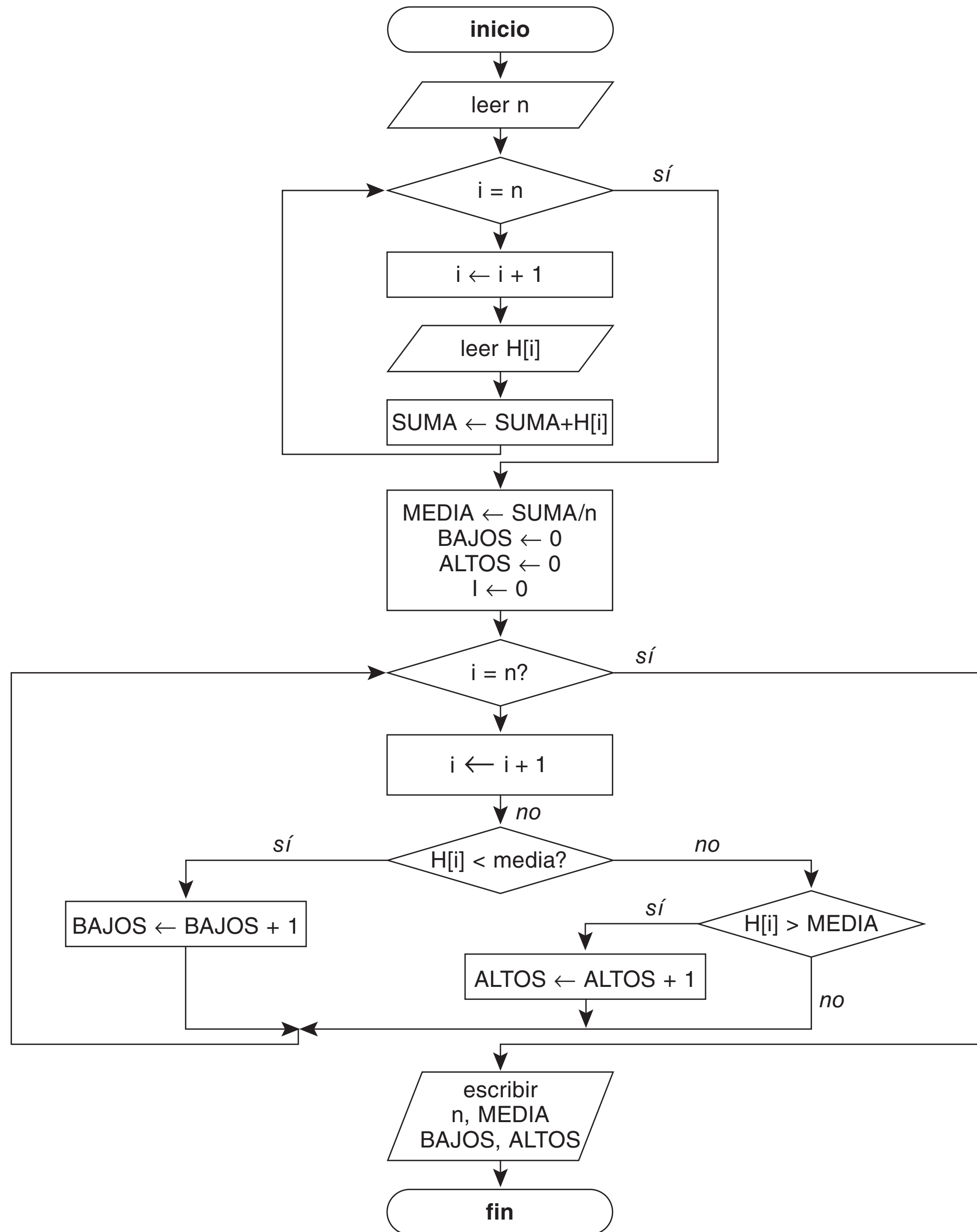


Figura 7.2. Diagrama de flujo para el cálculo de la estatura media de una clase.

EJEMPLO 7.5

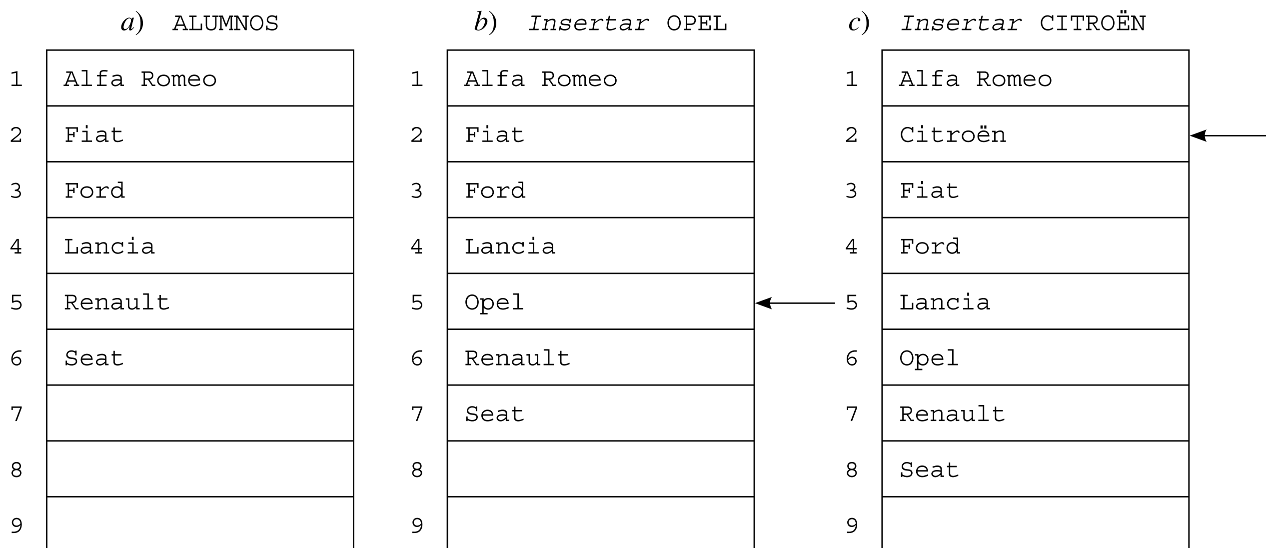
Se tiene un array `Coches`[2] *de nueve elementos de contiene siete marcas de automóviles en orden alfabético y se desea insertar dos nuevas marcas:* `Opel` *y* `Citroën`.

Como `Opel` está comprendido entre `Lancia` y `Renault`, se deberán desplazar hacia abajo los elementos 5 y 6, que pasarán a ocupar la posición relativa 6 y 7. Posteriormente debe realizarse la operación con `Citroën`, que ocupará la posición 2.

El algoritmo que realiza esta operación para un vector de *n* elementos es el siguiente, suponiendo que haya espacio suficiente en el vector.

[2] En Latinoamérica, su término equivalente es `CARRO` o `AUTO`.

```
1. //Calcular la posición ocupada por el elemento a insertar (por ejemplo, P)
2. //Inicializar contador de inserciones i ← n
3. mientras i >= P hacer
     //transferir el elemento actual i-ésimo hacia abajo, a la posición i+1
     COCHES[i + 1] ← COCHES[i]
     //decrementar contador
     i ← i - 1
     fin_mientras
4. //insertar el elemento en la posición P
     COCHES[P] ← 'nuevo elemento'
5. //actualizar el contador de elementos del vector
6. n ← n + 1
7. fin
```



Si se deseara realizar más inserciones, habría que incluir una estructura de decisión **si-entonces** para preguntar si se van a realizar más inserciones.

La operación de borrar un elemento al final del vector no presenta ningún problema; el borrado de un elemento del interior del vector provoca el movimiento hacia arriba de los elementos inferiores a él para reorganizar el vector.

El algoritmo de borrado del elemento *j-ésimo* del vector COCHES es el siguiente:

```
algoritmo borrado
inicio
//se utilizará una variable auxiliar —AUX— que contendrá el valor
//del elemento que se desea borrar
  AUX ← COCHES[j]
  desde i ← j hasta N-1 hacer
    //llevar elemento j + 1 hacia arriba
    COCHES[i] ← COCHES[i + 1]
  fin_desde
  //actualizar contador de elementos
  //ahora tendrá un elemento menos, N - 1
  N ← N - 1
fin
```

7.4. ARRAYS DE VARIAS DIMENSIONES

Los vectores examinados hasta ahora se denominan arrays unidimensionales y en ellos cada elemento se define o referencia por un índice o subíndice. Estos vectores son elementos de datos escritos en una secuencia. Sin embargo, existen grupos de datos que son representados mejor en forma de tabla o matriz con dos o más subíndices. Ejemplos típicos de tablas o matrices son: tablas de distancias kilométricas entre ciudades, cuadros horarios de trenes o aviones, informes de ventas periódicas (mes/unidades vendidas o bien mes/ventas totales), etc. Se pueden definir *tablas* o *matrices* como *arrays multidimensionales*, cuyos elementos se pueden referenciar por dos, tres o más subíndices. Los *arrays* no unidimensionales los dividiremos en dos grandes grupos:

arrays bidimensionales (2 dimensiones)
arrays multidimensionales (3 o más dimensiones)

7.4.1. Arrays bidimensionales (tablas/matrices)

El *array bidimensional* se puede considerar como un vector de vectores. Es, por consiguiente, un conjunto de elementos, todos del mismo tipo, en el cual el orden de los componentes es significativo y en el que se necesita especificar dos subíndices para poder identificar cada elemento del *array*.

Si se visualiza un *array* unidimensional, se puede considerar como una columna de datos; un *array* bidimensional es un grupo de columnas, como se ilustra en la Figura 7.3.

El diagrama representa una tabla o matriz de treinta elementos (5 × 6) con 5 filas y 6 columnas. Como en un vector de treinta elementos, cada uno de ellos tiene el mismo nombre. Sin embargo, un subíndice no es suficiente para especificar un elemento de un *array* bidimensional; por ejemplo, si el nombre del *array* es `M`, no se puede indicar `M[3]`, ya que no sabemos si es el tercer elemento de la primera fila o de la primera columna. Para evitar la ambigüedad, los elementos de un *array* bidimensional se referencian con dos subíndices: el primer subíndice se refiere a la *fila* y el segundo subíndice se refiere a la *columna*. Por consiguiente, `M[2, 3]` se refiere al elemento de la segunda fila, tercera columna. En nuestra tabla ejemplo `M[2, 3]` contiene el valor 18.

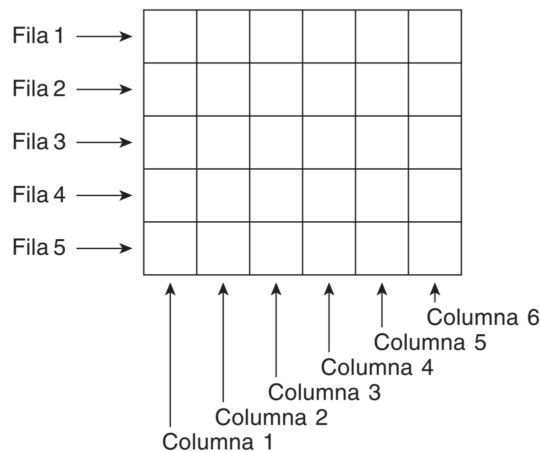


Figura 7.3. Array bidimensional.

Un *array bidimensional* `M`, también denominado *matriz* (términos matemáticos) o *tabla* (términos financieros), se considera que tiene dos dimensiones (una dimensión por cada subíndice) y necesita un valor para cada subíndice para poder identificar un elemento individual. En notación estándar, normalmente el primer subíndice se refiere a la fila del *array*, mientras que el segundo subíndice se refiere a la columna del *array*. Es decir, `B[I, J]` es el elemento de `B` que ocupa la I^a fila y la J^a columna, como se indica en la Figura 7.4.

El elemento `B[I, J]` también se puede representar por `BI, J`. Más formalmente en notación algorítmica, el *array* `B` con elementos del tipo `T` (numéricos, alfanuméricos, etc.) con *subíndices fila* que varían en el rango de `1` a `M` y *subíndices columna* en el rango de `1` a `N` es

	1	2	3	4	...	J	...	N
1								
2								
...								
I						B[I, J]		
...								
M								

Figura 7.4. Elemento `B[I, J]` del array `B`.

```
B(1:M, 1:N) = {B[I, J]}
```

donde `I = 1, ..., M` *o bien* `1 <= I <= M`
`J = 1, ..., N` `1 <= J <= N`
cada elemento `B[I, J]` *es de tipo* `T`.

El *array* `B` se dice que tiene `M` por `N` elementos. Existen `N` elementos en cada fila y `M` elementos en cada columna (`M*N`).

Los *arrays* de dos dimensiones son muy frecuentes: las calificaciones de los estudiantes de una clase se almacenan en una tabla `NOTAS` de dimensiones `NOTAS[20, 5]`, donde 20 es el número de alumnos y 5 el número de asignaturas. El valor del subíndice `I` debe estar entre 1 y 20, y el de `J` entre 1 y 5. Los subíndices pueden ser variables o expresiones numéricas, `NOTAS(M, 4)` y en ellos el subíndice de filas irá de `1` a `M` y el de columnas de `1` a `N`.

En general, se considera que un *array* bidimensional comienza sus subíndices en 0 o en 1 (según el lenguaje de programación, *0 en el lenguaje C, 1 en FORTRAN*), pero pueden tener límites seleccionados por el usuario durante la codificación del algoritmo. En general, el *array* bidimensional `B` con su primer subíndice, variando desde un límite inferior `L` (inferior, *low*) a un límite superior `U` (superior, *up*). En notación algorítmica

```
B(L1:U1, L2:U2) = {B[I, J]}
```

donde `L1 <= I <= U1`
`L2 <= J <= U2`
cada elemento `B[I, J]` *es de tipo* `T`.

El número de elementos de una fila de `B` es `U2-L2+1` y el número de elementos en una columna de `B` es `U1-L1+1`. Por consiguiente, el número total de elementos del *array* `B` es `(U2-L2+1)*(U1-L1+1)`.

EJEMPLO 7.6

La matriz T representa una tabla de notaciones de saltos de altura (primer salto), donde las filas representan el nombre del atleta y las columnas las diferentes alturas saltadas por el atleta. Los símbolos almacenados en la tabla son: x, salto válido; 0, salto nulo o no intentado.

Fila \ Columna T	2.00	2.10	2.20	2.30	2.35	2.40
García	x	0	x	x	x	0
Pérez	0	x	x	0	x	0
Gil	0	0	0	0	0	0
Mortimer	0	0	0	x	x	x

EJEMPLO 7.7

Un ejemplo típico de un array bidimensional es un tablero de ajedrez. Se puede representar cada posición o casilla del tablero mediante un array, en el que cada elemento es una casilla y en el que su valor será un código representativo de cada figura del juego.

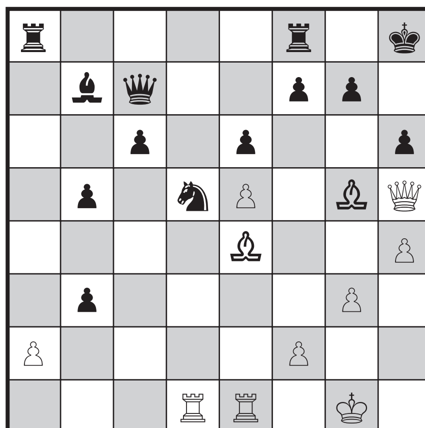


Figura 7.5. Array típico, "tablero de ajedrez".

Los diferentes elementos serán

`elemento[i, j] = 0`	si no hay nada en la casilla *[i, j]*
`elemento[i, j] = 1`	si el cuadro (casilla) contiene un peón blanco
`elemento[i, j] = 2`	un caballo blanco
`elemento[i, j] = 3`	un alfil blanco
`elemento[i, j] = 4`	una torre blanca
`elemento[i, j] = 5`	una reina blanca
`elemento[i, j] = 6`	un rey blanco

y los correspondientes números, negativos para las piezas negras.

EJEMPLO 7.8

Supongamos que se dispone de un mapa de ferrocarriles y los nombres de las estaciones (ciudades) están en un vector denominado "ciudad". El array `f` *puede tener los siguientes valores:*

`f[i, j] = 1`	si existe enlace entre las ciudades `i` y `j`, `ciudad[i]` y `ciudad[j]`
`f[i, j] = 0`	no existe enlace

> **Nota**
>
> El *array* f resume la información de la estructura de la red de enlaces.

7.5. ARRAYS MULTIDIMENSIONALES

Un *array* puede ser definido de tres dimensiones, cuatro dimensiones, hasta de *n-dimensiones*. Los conceptos de rango de subíndices y número de elementos se pueden ampliar directamente desde *arrays* de una y dos dimensiones a estos *arrays* de orden más alto. En general, un *array* de n-dimensiones requiere que los valores de los *n* subíndices

puedan ser especificados a fin de identificar un elemento individual del *array*. Si cada componente de un *array* tiene *n* subíndices, el *array* se dice que es sólo de *n-dimensiones*. El *array* `A` de n-dimensiones se puede identificar como

```
A(L1:U1, L2:U2, ... Ln:Un)
```

y un elemento individual del *array* se puede especificar por

$A(I_1, I_2, \ldots, I_n)$

donde cada subíndice I_k está dentro de los límites adecuados

$L_k <= I_k <= U_k$ donde $k = 1, 2, \ldots, n$

El número total de elementos de un *array* `A` es

$$\prod_{k=1}^{n} (U_k - L_k + 1) \qquad \Pi \text{ (símbolo del producto)}$$

que se puede escribir alternativamente como

$$(U_1-L_1+1)*(U_2-L_2+1)*\ldots*(U_N-L_N+1)$$

Si los límites inferiores comenzasen en 1, el *array* se representaría por

$A(K_1, K_2, \ldots, K_n)$ *o bien* $A_{k1, k2, \ldots, kn}$

donde

$1 <= K_1 <= S_1$
$1 <= K_2 <= S_2$
.
.
$1 <= K_n <= S_n$

EJEMPLO 7.9

Un array de tres dimensiones puede ser uno que contenga los datos relativos al número de estudiantes de la universidad `ALFA` *de acuerdo a los siguientes criterios:*

- *cursos (primero a quinto),*
- *sexo (varón/hembra),*
- *diez facultades.*

El *array* `ALFA` puede ser de dimensiones 5 por 2 por 10 (alternativamente 10 × 5 × 2 o 10 × 2 × 5, 2 × 5 × 10, etcétera). La Figura 7.6 representa el *array* `ALFA`.

El valor de elemento `ALFA[I, J, K]` es el número de estudiantes del curso `I` de sexo `J` de la facultad `K`. Para ser válido `I` debe ser 1, 2, 3, 4 o 5; `J` debe ser 1 o 2; `K` debe estar comprendida entre 1 y 10 inclusive.

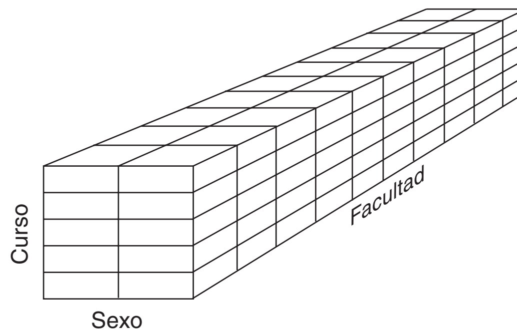


Figura 7.6. Array de tres dimensiones.

EJEMPLO 7.10

Otro array de tres dimensiones puede ser PASAJE que representa el estado actual del sistema de reserva de una línea aérea, donde

`i = 1, 2, ..., 10`	representa el número de vuelo
`j = 1, 2, ..., 60`	representa la fila del avión
`k = 1, 2, ..., 12`	representa el asiento dentro de la fila

Entonces

`pasaje[i, j, k] = 0`	*asiento libre*
`pasaje[i, j, k] = 1`	*asiento ocupado*

7.6. ALMACENAMIENTO DE ARRAYS EN MEMORIA

Las representaciones gráficas de los diferentes *arrays* se recogen en la Figura 7.7. Debido a la importancia de los *arrays*, casi todos los lenguajes de programación de alto nivel proporcionan medios eficaces para almacenar y acceder a los elementos de los *arrays*, de modo que el programador no tenga que preocuparse sobre los detalles específicos de almacenamiento. Sin embargo, el almacenamiento en la computadora está dispuesto fundamentalmente en secuencia contigua, de modo que cada acceso a una matriz o tabla la máquina debe realizar la tarea de convertir la posición dentro del *array* en una posición perteneciente a una línea.

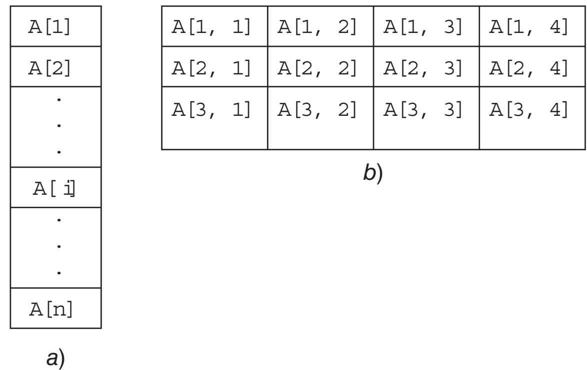


Figura 7.7. Arrays de una y dos dimensiones.

7.6.1. Almacenamiento de un vector

El almacenamiento de un vector en memoria se realiza en celdas o posiciones secuenciales. Así, en el caso de un vector `A` con un subíndice de rango 1 a *n*,

Posición B	`A[1]`
Posición B+1	`A[2]`
. . .	`A[3]`
	. . .
	`A[i]`
	. . .
Posición B+n–1	`A[n]`

Si cada elemento del *array* ocupa `S` bytes (1 *byte* = 8 bits) y `B` es la dirección inicial de la memoria central de la computadora —*posición o dirección base*—, la dirección inicial del elemento *i-ésimo* sería:

```
B+(I-1)*S
```

> ***Nota***
>
> Si el límite inferior no es igual a 1, considérese el *array* declarado como N(4:10); la dirección inicial de N(6) es B+(6–4)*S.

En general, el elemento `N(I)` de un *array* definido como `N(L:U)` tiene la dirección inicial

```
B+(I-L)*S
```

7.6.2. Almacenamiento de arrays multidimensionales

Debido a que la memoria de la computadora es lineal, un *array* multidimensional debe estar linealizado para su disposición en el almacenamiento. Los lenguajes de programación pueden almacenar los *arrays* en memoria de dos formas: *orden de fila mayor* y *orden de columna mayor*.

El medio más natural en que se leen y almacenan los *arrays* en la mayoría de los compiladores es el denominado *orden de fila mayor* (véase Figura 7.8). Por ejemplo, si un *array* es `B[1:2, 1:3]`, el orden de los elementos en la memoria es:

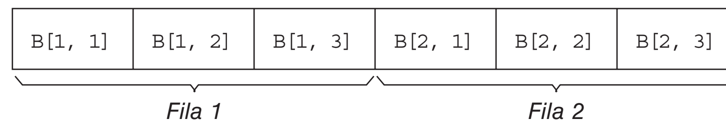


Figura 7.8. Orden de fila mayor.

C, COBOL y Pascal *almacenan los elementos por filas.*

FORTRAN emplea el *orden de columna mayor* en el que las entradas de la primera columna vienen primero.

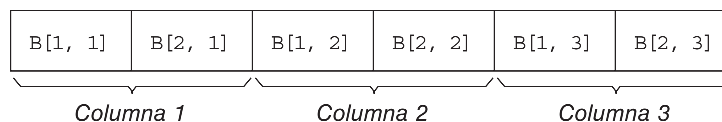


Figura 7.9. Orden de columna mayor.

De modo general, el compilador del lenguaje de alto nivel debe ser capaz de calcular con un índice *[i, j]* la posición del elemento correspondiente.

En un *array* en orden de fila mayor, cuyos subíndices máximos sean *m* y *n* (*m*, filas; *n*, columnas), la posición p del elemento [i, j] con relación al primer elemento es

```
p = n(i–1)+j
```

Para calcular la dirección real del elemento [i, j] se añade p a la posición del primer elemento y se resta 1. La representación gráfica del almacenamiento de una tabla o matriz `B[2, 4]` y `C[2, 4]`. (Véase Figura 7.10.)

En el caso de un *array* de tres dimensiones, supongamos un *array tridimensional* `A[1:2, 1:4, 1:3]`. La Figura 7.11 representa el *array* `A` y su almacenamiento en memoria.

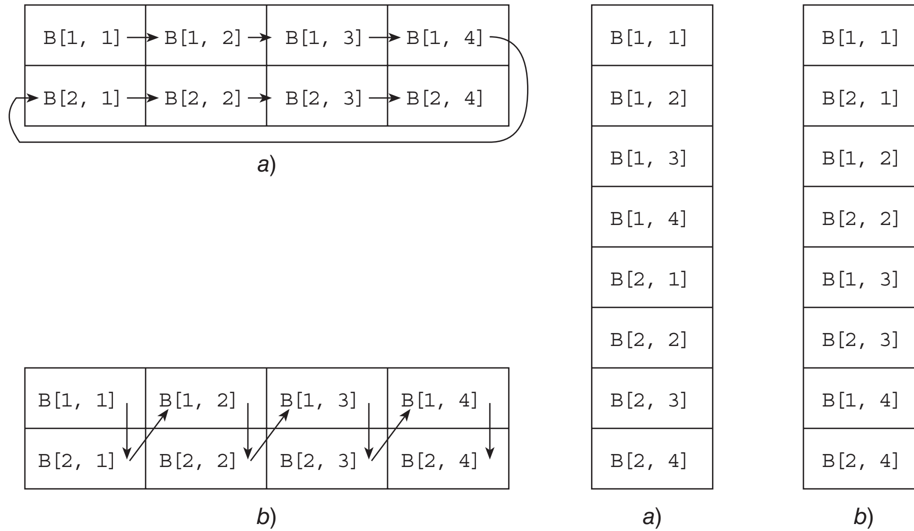


Figura 7.10. Almacenamiento de una matriz: *a*) por filas, *b*) por columnas.

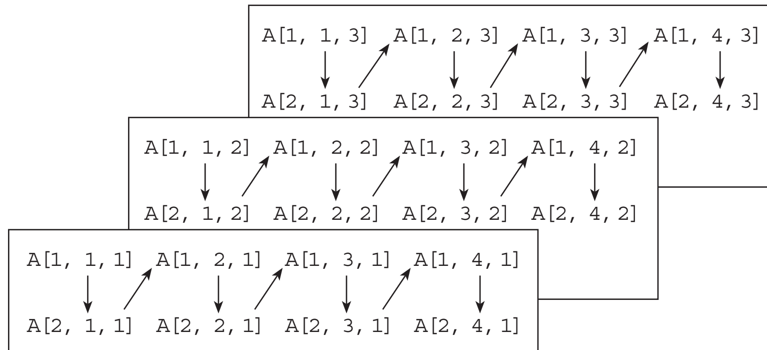


Figura 7.11. Almacenamiento de una matriz `A[2,4,3]` por columnas.

En orden a determinar si es más ventajoso almacenar un *array* en orden de columna mayor o en orden de fila mayor, es necesario conocer en qué orden se referencian los elementos del *array*. De hecho, los lenguajes de programación no le dan opción al programador para que elija una técnica de almacenamiento.

Consideremos un ejemplo del cálculo del valor medio de los elementos de un *array* `A` de 50 por 300 elementos, `A[50, 300]`. Los algoritmos de almacenamiento respectivos serán:

Almacenamiento por columna mayor

```
total ← 0
desde j ← 1 hasta 300 hacer
  desde i ← 1 hasta 50 hacer
    total ← total + a[i, j]
  fin_desde
fin_desde
media ← total / (300*50)
```

Almacenamiento por fila mayor

```
total ← 0
desde i ← 1 hasta 50 hacer
  desde j ← 1 hasta 300 hacer
```

```
    total ← total + a[i, j]
  fin_desde
fin_desde
media ← total / (300*50)
```

7.7. ESTRUCTURAS *VERSUS* REGISTROS

Un array permite el acceso a una lista o una tabla de datos del mismo tipo de datos utilizando un único nombre de variable. En ocasiones, sin embargo, se desea almacenar información de diferentes tipos, tales como un nombre de cadena, un número de código entero y un precio de tipo real (coma flotante) juntos en una única estructura. Una estructura que almacena diferentes tipos de datos bajo una misma variable se denomina *registro*.

En POO[3] el almacenamiento de información de diferentes tipos con un único nombre suele efectuarse en clases. No obstante, las clases son tipos referencia, esto significa que a los objetos de la clase se accede mediante una referencia. Sin embargo, en muchas ocasiones se requiere el uso de tipos valor. Las variables de un tipo valor contienen directamente los datos, mientras que las variables de tipos referencia almacenan una referencia al lugar donde se encuentran almacenados sus datos. El acceso a los objetos a través de referencia añade tareas y tiempos suplementarios y también consume espacio. En el caso de pequeños objetos este espacio extra puede ser significativo. Algunos lenguajes de programación como **C** y los orientados a objetos como **C++**, **C#**, ofrecen el tipo estructura para resolver estos inconvenientes. Una *estructura* es similar a una clase en orientación a objetos e igual a un registro en lenguajes estructurados como C pero es un tipo valor en lugar de un tipo referencia.

7.7.1. Registros

Un registro en **Pascal** es similar a una estructura en **C** y aunque en otros lenguajes como **C#** y **C++** las clases pueden actuar como estructuras, en este capítulo restringiremos su definición al puro registro contenedor de diferentes tipos de datos. Un registro se declara con la palabra reservada **estructura** (**struct**, en inglés) o **registro** y se declara utilizando los mismos pasos necesarios para utilizar cualquier variable. Primero, se debe declarar el registro y a continuación se asignan valores a los miembros o elementos individuales del registro o estructura.

Sintaxis

```
estructura: nombre_clase
    tipo_1:  campo1
    tipo_2:  campo2
      ...
fin_estructura
```

```
registro: nombre_tipo
    tipo_1:  campo1
    tipo_2:  campo2
      ...
fin_registro
```

Ejemplo

```
estructura: fechaNacimiento
    entero: mes  // mes de nacimiento
    entero: dia  // día de nacimiento
    entero: año  // año de nacimiento
Fin_estructura
```

La declaración anterior reserva almacenamiento para los elementos de datos individuales denominados *campos* o **miembros** de la estructura. En el caso de fecha, la estructura consta de tres campos día, mes y año relativos a una fecha de nacimiento o a una fecha en sentido general. El acceso a los miembros de la estructura se realiza con el operador punto y con la siguiente sintaxis

Nombre_estructura.miembro

Así `fechaNacimiento.mes` se refiere al miembro mes de la estructura fecha, y `fechaNacimiento.dia` se refiere al día de nacimiento de una persona. Un tipo de dato estructura más general podría ser `Fecha` y que sirviera

[3] Programación orientada a objetos.

para cualquier dato aplicable a cualquier aplicación (fecha de nacimiento, fecha de un examen, fecha de comienzo de clases, etc.).

```
estructura: Fecha
    entero: mes
    entero: dia
    entero: año
fin_estructura
```

Declaración de tipos estructura

Una vez definido un tipo estructura se pueden declarar variables de ese tipo al igual que se hace con cualquier otro tipo de datos. Por ejemplo, la sentencia de definición

```
Fecha: Cumpleaños, delDia
```

reserva almacenamiento para dos variables llamadas `Cumpleaños` y `delDia`, respectivamente. Cada una de estas estructuras individuales tiene el mismo formato que el declarado en la clase `Fecha`.

Los miembros de una estructura no están restringidos a tipos de datos enteros sino que pueden ser cualquier tipo de dato válido del lenguaje. Por ejemplo, consideremos un registro de un empleado de una empresa que constase de los siguientes miembros:

```
estructura Empleado
   Cadena: nombre
   entero: idNumero
   real: Salario
   Fecha: FechaNacimiento
   entero: Antigüedad
fin_estructura
```

Obsérvese que en la declaración de la estructura `Empleado`, el miembro `Fecha` es un nombre de un tipo estructura previamente definido. El acceso individual a los miembros individuales del tipo estructura de la clase `Empleado` se realiza mediante dos operadores punto, de la forma siguiente:

```
Empleado.Fecha.Dia
```

y se refiere a la variable `Dia` de la estructura `Fecha` de la estructura `Empleado`.

> ***Estructuras de datos homogéneas y heterogéneas***
>
> Los registros (estructuras) y los arrays son tipos de datos estructurados. La diferencia entre estos dos tipos de estructuras de datos son los tipos de elementos que ellos contienen. Un array es una estructura de datos homogénea, que significa que cada uno de sus componentes deben ser del mismo tipo. Un registro es una estructura de datos heterogénea, que significa que cada uno de sus componentes pueden ser de tipos de datos diferentes. Por consiguiente, un array de registros es una estructura de datos cuyos elementos son de los mismos tipos heterogéneos.

7.8. ARRAYS DE ESTRUCTURAS

La potencia real de una estructura o registro se manifiesta en toda su expresión cuando la misma estructura se utiliza para listas de datos. Por ejemplo, supongamos que se deben procesar los datos de la tabla de la Figura 7.12.

Un sistema podría ser el siguiente: Almacenar los números de empleado en un array de enteros, los nombres en un array de cadenas de caracteres y los salarios en un array de números reales. Al organizar los datos de esta forma,

Número de empleado	Nombre del empleado	Salario
97005	Mackoy, José Luis	1.500
95758	Mortimer, Juan	1.768
87124	Rodríguez, Manuel	2.456
67005	Carrigan, Luis José	3.125
20001	Mackena, Luis Miguel	2.156
20020	García de la Cruz, Heraclio	1.990
99002	Mackoy, María Victoria	2.450
20012	González, Yiceth	4.780
21001	Gonzáles, Rina	3.590
97005	Rodríguez, Concha	3.574

Figura 7.12. Lista de datos.

cada columna de la Figura 7.13 se considera como una lista independiente que se almacena en su propio array. La correspondencia entre elementos de cada empleado individual se mantiene almacenan los datos de un empleado en la misma posición de cada array.

La separación de cada lista completa en tres arrays individuales no es muy eficiente, ya que todos los datos relativos a un empleado se organizan juntos en un registro como se muestra en la Figura 7.13. Utilizando una estructura, se mantiene la integridad de los datos de la organización y bastará un programa que maneje los registros para poder ser manipulados con eficacia. La declaración de un array de estructuras es similar a la declaración de un array de cualquier otro tipo de variable. En consecuencia, en el caso del archivo de empleados de la empresa se puede declarar el array de empleado con el nombre Empleado y el registro o estructura lo denominamos `RegistroNomina`

```
estructura: RegistroNomina
    entero: NumEmpleado
    cadena[30]: Nombre
    real: Salario
fin_estructura
```

Se puede declarar un array de estructuras `RegistroNomina` que permite representar toda la tabla anterior

```
array [1..10] de RegistroNomina : Empleado
```

La sentencia anterior construye un array de diez elementos Empleado, cada uno de los cuales es una estructura de datos de tipo `RegistroNomina` que representa a un empleado de la empresa Aguas de Sierra Mágina. Obsérvese que la creación de un array de diez estructuras tiene el mismo formato que cualquier otro array. Por ejemplo, la creación de un array de diez enteros denominado `Empleado` requiere la declaración:

```
array [1..10] de entero : Empleado
```

En realidad la lista de datos de empleado se ha representado mediante una lista de registros como se mostraba en la Figura 7.13.

Número de empleado	Nombre del empleado	Salario
97005	Mackoy, José Luis	1.500
95758	Mortimer, Juan	1.768
87124	Rodríguez, Manuel	2.456
67005	Carrigan, Luis José	3.125
20001	Mackena, Luis Miguel	2.156
20020	García de la Cruz, Heraclio	1.990
99002	Mackoy, María Victoria	2.450
20012	González, Yiceth	4.780
21001	Verástegui, Rina	3.590
97005	Collado, Concha	3.574

Figura 7.13. Lista de registros.

7.9. UNIONES

Una **unión** es un tipo de dato derivado (estructurado) que contiene sólo uno de sus miembros a la vez durante la ejecución del programa. Estos miembros comparten el mismo espacio de almacenamiento; es decir, una unión comparte el espacio en lugar de desperdiciar espacio en variables que no se están utilizando. Los miembros de una unión pueden ser de cualquier tipo, y pueden contener dos o más tipos de datos. La sintaxis para declarar un tipo `union` es idéntica a la utilizada para definir un tipo `estructura`, excepto que la palabra `union` sustituye a `estructura`:

Sintaxis

```
union nombre
    tipo_dato1   identificador1
    tipo_dato2   identificador2
       …
fin_union
```

El número de bytes utilizado para almacenar una unión debe ser suficiente para almacenar el miembro más grande. Sólo se puede hacer referencia a un miembro a la vez y, por consiguiente, a un tipo de dato a la vez. En tiempo de ejecución, el espacio asignado a la variable de tipo `union` no incluye espacio de memoria más que para un miembro de la unión.

Ejemplo

```
union TipoPeso
  entero Toneladas
  real Kilos
  real Gramos
fin_union
TipoPeso peso       // Declaración de una variable tipo unión
```

En tiempo de ejecución, el espacio de memoria asignado a la variable `peso` no incluye espacio para tres componentes distintos; en cambio, `peso` puede contener uno de los siguientes valores: `entero` o `real`.

El acceso a un miembro de la unión se realiza con el operador de acceso a miembros (punto, **.**)

```
peso.Toneladas = 325
```

> Una unión es similar a una estructura con la diferencia de que sólo se puede almacenar en memoria de modo simultáneo un único miembro o campo, al contrario que la estructura que almacena espacio de memoria para todos sus miembros.

7.9.1. Unión *versus* estructura

Una estructura se utiliza para definir un tipo de dato con diferentes miembros. Cada miembro ocupa una posición independiente de memoria

```
estructura rectángulo
inicio
     entero: anchura
     entero: altura
fin_estructura
```

La estructura **`rectángulo`** se puede representar en memoria en la Figura 7.14.

Figura 7.14. Estructura *versus* unión.

Una unión es similar a una estructura, sin embargo, sólo se define una única posición que puede ser ocupada por diferentes miembros con nombres diferentes:

```
union valor
  entero valor_e
  real   valor_r
fin_union
```

Los miembros `valor_e` y `valor_r` comparten el mismo espacio gráficamente; se puede pensar que una estructura es una caja con diferentes compartimentos, cada uno con su propio nombre (miembro), mientras que una unión es una caja sin compartimentos donde se pueden colocar diferentes etiquetas en su interior.

En una estructura, los miembros no interactúan; el cambio de un miembro no modifica a los restantes. En una unión todos los miembros ocupan el mismo espacio, de modo que sólo uno puede estar activo en un momento dado.

EJERCICIO 7.1.

Se desea almacenar información sobre una figura geométrica estándar (círculo, rectángulo o triángulo). La información necesaria para dibujar un círculo es diferente de los datos que se necesitan para dibujar un rectángulo, de modo que se necesitan diferentes estructuras para cada figura:

```
estructura circulo
    entero: radio
fin_estructura

estructura rectángulo
    entero: altura, anchura
fin_estructura

estructura triangulo
    entero: base
    entero: altura
fin_estructura
```

El ejercicio consiste en definir una estructura que pueda contener una figura genérica. El primer código es un número que indica el tipo de figura y el segundo es una unión que contiene la información de la figura.

```
estructura figura
entero: tipo      //tipo=0, circulo; tipo=1, rectángulo; tipo=2, triángulo
  union figura_genérica
    circulo: datos_circulo
    rectabgulo: datos_rectangulo
    triangulo: datos_triangulo
  fin_union: datos
fin_estructura
```

De este modo se puede acceder a miembros de la unión, estructura específica o estructura general con el operador punto. Así el tipo de dato básico figura se puede definir y acceder a sus miembros de la forma siguiente:

```
figura: una_figura
// ...
una_figura.tipo ← 0
una_figura.datos.datos_circulo.radio ← 125
```

7.10. ENUMERACIONES

Una de las características importantes de la mayoría de los lenguajes de programación modernos es la posibilidad de definir nuevos tipos de datos. Entre estos tipos definidos por el usuario se encuentran los *tipos enumerados* o *enumeraciones.*

Un tipo enumerado o de enumeración es un tipo cuyos valores están definidos por una lista de constantes de tipo entero. En un tipo de enumeración las constantes se representan por identificadores separados por comas y encerrados entre llaves. Los valores de un tipo enumerado comienzan con 0, a menos que se especifique lo contrario y se incremente en 2. La sintaxis es:

```
enum nombre_tipo {identificador1, identificador2, ...]
```

identificador debe ser válido (**1ª, 'B', '24x'** no son identificadores válidos).

EJEMPLO

```
enum Dias {LUN, MAR, MIE, JUE, VIE, SAB, DOM}
enum Meses {ENE, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV DIC}
```

El tipo `Dias` toma 7 valores, 0 a 6, y `Meses` toma 12 valores de 0 a 11. Estas declaraciones crean un nuevo tipo de datos, `Dias` y `Meses;` los valores comienzan en 0, a menos que se indique lo contrario.

```
enum MESES {ENE←1, FEB, MAR, ABR, MAY, JUN, JUL, AGO←8, SEP, OCT, NOV, DIC)
```

Con la declaración anterior los meses se enumeran de 1 a 12.

El valor de cada constante de enumeración se puede establecer explícitamente en la definición, asignándole un valor al identificador, que puede ser el mismo o distinto entero. También se puede representar el tipo de dato con esta sintaxis:

```
enum Mes
{
  ENE ← 31, FEB ← 28, MAR ← 31, ABR ← 30, MAY ← 31, JUN ← 30, JUL ← 31, AGO ← 31,
  SEP ← 30, OCT ← 31, NOV ← 30, DIC ← 31
}
```

Si no se especifica ningún valor numérico, los identificadores en una definición de un tipo de enumeración se les asignan valores consecutivos que comienzan por cero.

EJEMPLO

```
enum Direccion { NORTE ← 0, SUR ← 1, ESTE ← 2, OESTE ← 3}
```

es equivalente a

```
enum Direccion { NORTE, SUR, ESTE, OESTE}
```

Sintaxis

```
enum <nombre> {<enumerador1>, <enumerador2>,…}
enumerador identificador = expresión constante
```

Las variables de tipo enumeración se pueden utilizar en diferentes tipos de operaciones.

Creación de variables

`enum Semaforo {verde, rojo, amarillo}`

se pueden asignar variables de tipo `Semaforo`:

```
var
  Semaforo Calle, Carretera, Plaza
```

Se crean las variables `Calle, Carretera` y `Plaza` de tipo `Semaforo`.

Asignación

La sentencia de asignación

```
Calle ← Rojo
```

no asigna a `Calle` la cadena de caracteres `Rojo` ni el contenido de una variable de nombre `Rojo` sino que asigna el valor `Rojo` que es de uno de los valores del dominio del tipo de datos `Semaforo`.

***Sentencias de selección* `caso_de (switch), si-entonces (if-then)`**

Algoritmo

```
enum Mes { ENE, FEB, MAR, ... }
algoritmo DemoEnum
var Mes MesVacaciones
inicio
MesVacaciones ← ENE
si (MesVacaciones ← ENE)
    Escribir ('El mes de vacaciones es Enero')
fin_si
fin
```

Algoritmo

Se pueden usar valores de enumeración en una sentencia `según_sea (switch)`:

```
tipo
enum Animales {Raton, Gato, Perro, Paloma, Reptil, Canario}
var
  Animales: ADomesticos
  según_sea: ADomesticos
    Raton: escribir '...'
    Gato  : escribir '...'
  fin_segun_sea
```

Sentencias repetitivas

Las variables de enumeración se pueden utilizar en bucles `desde, mientras, ...`:

```
algoritmo demoEnum2
tipo
enum meses { ENE=1, FEB, MAR, ABR, MAY, JUN, JUL. AGO, SEP, OCT, NOV, DIC}
var
    enum meses: mes
    inicio
    desde mes ← ENE hasta mes <= DIC
    ...
    fin_desde
fin
```

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

7.1. *Escribir un algoritmo que permita calcular el cuadrado de los cien primeros números enteros y a continuación escribir una tabla que contenga dichos cien números cuadrados.*

Solución

El problema consta de dos partes:

1. Cálculo de los cien primeros números enteros y sus cuadrados.
2. Diseño de una tabla `T`, `T(1), T(2), ...`, `T(100)` que contiene los siguientes valores:

```
T(1) = 1*1 = 1
T(2) = 2*2 = 4
T(3) = 3*3 = 9
...
```

El algoritmo se puede construir con estructuras de decisión o alternativas, o bien con estructuras repetitivas. En nuestro caso utilizaremos una estructura repetitiva `desde`.

```
algoritmo cuadrados
tipo
  array[1..100] de entero : tabla
var
  tabla : T
  entero : I, C
inicio
  desde I ← 1 hasta 100 hacer
    C ← I * I
    escribir(I, C)
  fin_desde
  desde I ← 1 hasta 100 hacer
    T[I] ← I * I
    escribir(T[I])
  fin_desde
fin
```

7.2. *Se tienen N temperaturas. Se desea calcular su media y determinar entre todas ellas cuáles son superiores o iguales a esa media.*

Solución

Análisis

En un primer momento se leen los datos y se almacenan en un vector (array unidimensional) `TEMP(1:N)`.

A continuación se van realizando las sumas sucesivas a fin de obtener la media.

Por último, con un bucle de lectura de la tabla se va comparando cada elemento de la misma con la media y luego, mediante un contador, se calcula el número de temperaturas igual o superior a la media.

Tabla de variables	
N	Número de elementos del vector o tabla.
TEMP	Vector o tabla de temperatura.
SUMA	Sumas sucesivas de las temperaturas.
MEDIA	Media de la tabla.
C	Contador de temperaturas >= MEDIA.

Pseudocódigo

```
algoritmo temperaturas
const
  N = 100
tipo
  array[1..N] de real : temperatura
var
  temperatura: Temp
  entero : I, C
  real : suma, media
inicio
  suma ← 0
  media ← 0
  C ← 0
  desde I ← 1 hasta N hacer
    leer(Temp[I])
    suma ← suma+Temp[I]
  fin_desde
  media ← suma/N
  para I ← 1 hasta N hacer
    si Temp[I] >= media entonces
      C ← C+1
      escribir(Temp[I])
    fin_si
  fin_para
  escribir('La media es:', media)
  escribir('El total de temperaturas >=', media, 'es:', C)
fin
```

7.3. *Escribir el algoritmo que permita sumar el número de elementos positivos y el de negativos de una tabla T.*

Solución

Sea una tabla T de dimensiones M, N leídas desde el teclado.

Tabla de variables	
I, J, M, N:	entero
SP:	real
SN:	real

Pseudocódigo

```
algoritmo suma_resta
const
  M = 50
  N = 20
```

```
tipo
  array[1..M, 1..N] de real : Tabla
var
  Tabla : T
  entero : I, J
  real : SP, SN
inicio
  SP ← 0
  SN ← 0
  desde I ← 1 hasta M hacer
    desde J ← 1 hasta N hacer
      si T[I, J] > 0 entonces
        SP ← SP + T[I, J]
      si_no
        SN ← SN + T[I, J]
      fin_si
    fin_desde
  fin_desde
  escribir('Suma de positivos', SP, 'de negativos', SN)
fin
```

7.4. *Inicializar una matriz de dos dimensiones con un valor constante dado K.*

Solución

Análisis

El algoritmo debe tratar de asignar la constante K a todos los elementos de la matriz `A[M, N]`.

```
A[1, 1] = K    A[1, 2] = K ... A[1, N] = K
.
.
A[M, 1] = K    A[M, 2] = K ... A[M, N] = K
```

Dado que es una matriz de dos dimensiones, se necesitan dos bucles anidados para la lectura.

Pseudocódigo

```
algoritmo inicializa_matriz
inicio
  desde I ← 1 hasta M hacer
    desde J ← 1 hasta N hacer
      A[I, J] ← K
    fin_desde
  fin_desde
fin
```

7.5. *Realizar la suma de dos matrices bidimensionales.*

Solución

Análisis

Las matrices `A[I, J]`, `B[I, J]` para que se puedan sumar deben tener las mismas dimensiones. La matriz suma `S[I, J]` tendrá iguales dimensiones y cada elemento será la suma de las correspondientes matrices `A` y `B`. Es decir,

```
S[I, J] = A[I, J] + B[I, J]
```

Dado que se trata de matrices de dos dimensiones, el proceso se realizará con dos bucles anidados.

Pseudocódigo

```
algoritmo suma_matrices
inicio
  desde I ← 1 hasta N hacer
    desde J ← 1 hasta M hacer
      S[I, J] ← A[I, J] + B[I, J]
    fin_desde
  fin_desde
fin
```

7.6. *Se dispone de una tabla T de dos dimensiones. Calcular la suma de sus elementos.*

Solución

Supongamos las dimensiones de `T`, `M` y `A` y que se compone de números reales.

Tabla de variables

I	Contador de filas.
J	Contador de columnas.
M	Número de filas de la tabla `T`.
N	Número de columnas de la tabla `T`.
T	Tabla.
S	Suma de los elementos de la tabla.
I, J, M, N	Enteros.
T, S	Reales.

Pseudocódigo

```
algoritmo suma_elementos
const
  M = 50
  N = 20
tipo
  array[1..M, 1..N] de real : Tabla
var
  entero : I, J
  Tabla : T
  real : S
inicio
  desde I ← 1 hasta M hacer
    desde J ← 1 hasta N hacer
      leer(T[I, J])
    fin_desde
  fin_desde
  S ← 0  {inicialización de la suma S}
  desde I ← 1 hasta M hacer
    desde J ← 1 hasta N hacer
      S ← S + T[I, J]
    fin_desde
  fin_desde
  escribir('La suma de los elementos de la matriz =', S)
fin
```

7.7. *Realizar la búsqueda de un determinado nombre en una lista de nombres, de modo que el algoritmo imprima los siguientes mensajes según el resultado:*

`'Nombre encontrado'`	*si el nombre está en la lista*
`'Nombre no existe'`	*si el nombre no está en la lista*

Solución

Se recurrirá en este ejercicio a utilizar un interruptor `SW`, de modo que si `SW = falso` el nombre no existe en la lista y si `SW = verdadero` el nombre existe en la lista (o bien caso de no existir la posibilidad de variables lógicas, definir `SW` como `SW = 0` si es falso y `SW = 1` si es verdadero o cierto).

Método 1

```
algoritmo búsqueda
const
  N = 50
tipo
  array[1..N] de cadena : Listas
var
  Listas : l
  lógico : SW
  cadena : nombre
  entero : I
inicio
  SW ← falso
  leer(nombre)
  desde I ← 1 hasta N hacer
    si l[I] = nombre entonces
      SW ← verdadero
    fin_si
  fin_desde
  si SW entonces
    escribir('Encontrado')
  si_no
    escribir('No existe', nombre)
  fin_si
fin
```

Método 2

```
algoritmo búsqueda
const
  N = 50
tipo
  array[1..N] de cadena : Listas
var
  Listas : l
  lógico : SW
  cadena : nombre
  entero : I
inicio
  SW ← 0
  leer(nombre)
  desde I ← 1 hasta N hacer
    si l[I] = nombre entonces
      SW ← 1
    fin_si
  fin_desde
  si SW = 1 entonces
    escribir('Encontrado')
  si_no
    escribir('No existe', nombre)
  fin_si
fin
```

7.8. *Se desea permutar las filas I y J de una matriz (array) de dos dimensiones (M*N):M filas, N columnas.*

Solución

Análisis

La tabla `T(M*N)` se puede representar por:

```
T[1, 1]   T[1, 2]   T[1, 3]   ...   T[1, N]
T[2, 1]   T[2, 2]   T[2, 3]   ...   T[2, N]
.
.
T[M, 1]   T[M, 2]   T[M, 3]   ...   T[M, N]
```

El sistema para permutar globalmente toda la fila I con la fila J se debe realizar permutando uno a uno el contenido de los elementos `T[I, K]` y `T[J, K]`.

Para intercambiar entre sí los valores de dos variables, recordemos que se necesitaba una variable auxiliar `AUX`. Así, para el caso de las variables `A` y `B`

```
AUX ← A
  A ← B
  B ← AUX
```

En el caso de nuestro ejercicio, para intercambiar los valores `T[I, K]` y `T[J, K]` se debe utilizar el algoritmo:

```
AUX     ← T[I, K]
T[I, K] ← T[J, K]
T[J, K] ← T[I, K]
```

Tabla de variables	
`I, J, K, M, N`	Enteras
`AUX`	Real
`Array`	Real

Pseudocódigo

```
algoritmo intercambio
const
  M = 50
  N = 30
tipo
  array[1..M, 1.. N] de entero : Tabla
var
  Tabla : T
  entero : AUX, I, J, K
inicio
  {En este ejercicio y dado que ya se han realizado muchos ejemplos de
   lectura de arrays con dos bucles desde, la operación de lectura completa del array se
   representará con la instrucción de leerArr(T)}
  leerArr(T)
  //Deducir I, J a intercambiar
  leer(I, J)
  desde K ← I hasta N hacer
    AUX ← T[I, K]
    T[I, K] ← T[J, K]
    T[J, K] ← AUX
  fin_desde
  //Escritura del nuevo array
  escribirArr(T)
fin
```

7.9. *Algoritmo que nos permita calcular la desviación estándar (SIGMA) de una lista de N números (N <= 15).*

Sabiendo que

$$\text{DESVIACIÓN} = \sqrt{\frac{\sum_{i=1}^{n}(x_i - m)^2}{n-1}}$$

```
algoritmo Calcular_desviación
 tipo
     array[1..15] de real : arr
 var
     arr    : x
     entero : n

 inicio
  llamar_a leer_array(x, n)
  escribir('La desviación estándar es ',desviacion(x, n))
 fin

 procedimiento leer_array(S arr:x  S entero:n)
   var
     entero : i
   inicio
     repetir
      escribir('Diga número de elementos de la lista ')
      leer(n)
     hasta_que n <= 15
     escribir('Deme los elementos:')
     desde i ← 1 hasta n hacer
        leer(x[i])
     fin_desde
   fin_procedimiento

 real función desviacion(E arr : x   E entero : n)
   var
     real   : suma, xm, sigma
     entero : i
   inicio
     suma ← 0
     desde i ← 1 hasta n hacer
        suma ← suma + x[i]
     fin_desde
     xm ← suma / n
     sigma ← 0
     desde i ← 1 hasta n hacer
        sigma ← sigma + cuadrado (x[i] - xm)
     fin_desde
     devolver(raiz2 (sigma / (n-1)))
   fin_función
```

7.10. *Utilizando arrays, escribir un algoritmo que visualice un cuadrado mágico de orden impar n, comprendido entre 3 y 11. El usuario debe elegir el valor de n.*
Un cuadrado mágico se compone de números enteros comprendidos entre 1 y n. La suma de los números que figuran en cada fila, columna y diagonal son iguales.

Ejemplo			
	8	1	6
	3	5	7
	4	9	2

Un método de generación consiste en situar el número 1 en el centro de la primera fila, el número siguiente en la casilla situada por encima y a la derecha y así sucesivamente. El cuadrado es cíclico, la línea encima de la primera es de hecho la última y la columna a la derecha de la última es la primera. En el caso de que el número generado caiga en una casilla ocupada, se elige la casilla que se encuentre debajo del número que acaba de ser situado.

```
algoritmo Cuadrado_magico
 var  entero : n
inicio
 repetir
  escribir('Dame las dimensiones del cuadrado (3 a 11) ')
  leer(n)
 hasta_que (n mod 2 <>0) Y (n <= 11) Y (n >= 3)
 dibujarcuadrado(n)
fin

 procedimiento dibujarcuadrado(E entero:n)
   var  array[1..11,1..11] de entero : a
        entero                       : i,j,c
   inicio
    i ← 2
    j ← n div 2
    desde c ← 1 hasta n*n hacer
        i ← i - 1
        j ← j + 1
        si j > n entonces
           j ← 1
        fin_si
        si i < 1 entonces
           i ← n
        fin_si
        a[i,j] ← c
        si c mod n = 0 entonces
           j ← j - 1
           i ← i + 2
        fin_si
    fin_desde
    desde i ← 1 hasta n hacer
      desde j ← 1 hasta n hacer
         escribir(a[i,j])
         {al codificar esta instrucción en un lenguaje será conveniente utilizar el
         parámetro correspondiente de "no avance de línea" en la salida en pantalla o
         impresora}
    fin_desde
      escribir(NL)
      //NL representa Nueva Línea, es decir, avance de línea
    fin_desde
   fin_procedimiento
```

7.11. *Obtener un algoritmo que efectúe la multiplicación de dos matrices A, B.*

$A \in M_{m,p}$ *elementos*
$B \in M_{p,n}$ *elementos*

Matriz producto: $C \in M_{m,n}$ elementos, tal que

$$C_{i,j} = \sum_{i=1}^{p} a_{i,k} * b_{k,j}$$

```
algoritmo Multiplicar_matrices
 tipo array[1..10,1..10] de real : arr
 var  entero : m, n, p
      arr    : a ,b, c

  inicio
  repetir
    escribir('Dimensiones de la 1ª matriz (filas columnas)')
    leer(m,p)
    escribir('Columnas de la 2ª matriz ')
    leer(n)
  hasta_que (n <= 10) Y (m <= 10) Y (p <= 10)
  escribir ('Deme elementos de la 1ª matriz')
  llamar_a leer_matriz(a,m,p)
  escribir ('Deme elementos de la 2ª matriz')
  llamar_a leer_matriz(b,p,n)
  llamar_a calcescrproducto(a, b, c, m, p, n)
 fin

procedimiento leer_matriz(S arr:matriz;E entero:filas,columnas)
var entero : i, j

inicio
   desde i ← 1 hasta filas hacer
    escribir('Fila ',i,':')
    desde j ← 1 hasta columnas hacer
       leer(matriz[i,j])
    fin_desde
   fin_desde
  fin_procedimiento

 procedimiento calcescrproducto(E arr: a, b, c;  E entero: m,p,n)
  var entero : i, j, k

 inicio
    desde i ← 1 hasta m hacer
      desde j ← 1 hasta n hacer
        c[i,j] ← 0
        desde k ← 1 hasta p hacer
         c[i,j] ← c[i,j] + a[i,k] * b[k,j]
        fin_desde
        escribir (c[i,j])        //no avanzar línea
      fin_desde
      escribir ( NL  )           //avanzar línea, nueva línea
    Fin_desde
  Fin_procedimiento
```

7.12. *Algoritmo que triangule una matriz cuadrada y halle su determinante. En las matrices cuadradas el valor del determinante coincide con el producto de los elementos de la diagonal de la matriz triangulada, multiplicado por –1 tantas veces como hayamos intercambiado filas al triangular la matriz.*

Proceso de triangulación de una matriz para todo i desde 1 hasta n – 1 hacer:

a) Si el elemento de lugar (i,i) es nulo, intercambiar filas hasta que dicho elemento sea no nulo o agotar los posibles intercambios.
b) A continuación se busca el primer elemento no nulo de la fila i-ésima y, en el caso de existir, se usa para hacer ceros en la columna de abajo.

Sea dicho elemento matriz[i,r]

Multiplicar fila i por matriz[i+1,r]/matriz[i,r] y restarlo a la i+1
Multiplicar fila i por matriz[i+2,r]/matriz[i,r] y restarlo a la i+2
....
Multiplicar fila i por matriz[m,r]/matriz[i,r] y restarlo a la m

```
algoritmo Triangulacion_matriz
Const m = <expresion>
      n = <expresion>
tipo  array[1..m, 1..n] de real : arr
var arr  : matriz
    real : dt

inicio
 llamar_a leer_matriz(matriz)
 llamar_a triangula(matriz, dt)
 escribir('Determinante= ', dt)
fin

procedimiento leer_matriz (S arr : matriz)
 var entero: i,j

 inicio
  escribir('Deme los valores para la matriz')
  desde i ← 1 hasta m hacer
   desde j ← 1 hasta n hacer
     leer( matriz[i, j])
   fin_desde
  fin_desde
 fin_procedimiento

procedimiento escribir_matriz (E arr : matriz)
 var entero   : i, j
     caracter : c
 inicio
  escribir('Matriz triangulada')
  desde i ← 1 hasta m hacer
   desde j ← 1 hasta n hacer
     escribir( matriz[i, j])     //no avanzar línea
   fin_desde
   escribir(NL)                  //avanzar línea, nueva línea
  fin_desde
  escribir('Pulse tecla para continuar')
  leer(c)
 fin_procedimiento

procedimiento interc(E/S real: a,b)
 var real : auxi
 inicio
  auxi ← a
  a ← b
  b ← auxi
 fin_procedimiento

procedimiento triangula (E arr : matriz; S real dt)
 var entero: signo
     entero: t, r, i, j
     real  : cs
```

```
inicio
 signo ← 1
 desde i ← 1 hasta m - 1 hacer
   t ← 1
   si matriz[i, i] = 0 entonces
     repetir
      si matriz[i + t, i] <> 0 entonces
        signo ← signo * (-1)
        desde j ← 1 hasta n hacer
          llamar_a interc(matriz[i,j],matriz[i + t,j])
        fin_desde
        llamar_a escribir_matriz(matriz)
      fin_si
      t ← t + 1
     hasta_que (matriz[i, i] <> 0) O (t = m - i + 1)
   fin_si
   r ← i - 1
   repetir
    r ← r + 1
   hasta_que (matriz[i, r] <> 0) O (r = n)
   si matriz[i, r] <> 0 entonces
       desde t ← i + 1 hasta m hacer
         si matriz[t, r] <> 0 entonces
            cs ← matriz[t, r]
            desde j ← r hasta n hacer
               matriz[t, j] ← matriz[t, j] - matriz[i, j] * (cs / matriz[i, r])
            fin_desde
            llamar_a escribir_matriz(matriz)
         fin_si
       fin_desde
   fin_si
 fin_desde
 dt ← signo
 desde i ← 1 hasta m hacer
   dt ← dt * matriz[i, i]
 fin_desde
fin_procedimiento
```

CONCEPTOS CLAVE

- Array bidimensional.
- Array de una dimensión.
- Array multidimensional.
- Arrays como parámetros.
- Arreglo.
- Datos estructurados.
- Estructura.
- Índice.
- Lista.
- Longitud de un array.
- Subíndice.
- Tabla.
- Tamaño de un array.
- Variable indexada.
- Vector.

RESUMEN

Un **array** (vector, lista o tabla) es una estructura de datos que almacena un conjunto de valores, todos del mismo tipo de datos. Un array de una dimensión, también conocido como array unidimensional o vector, es una lista de elementos del mismo tipo de datos que se almacenan utilizando un único nombre. En esencia, un array es una colección de variables

que se almacenan en orden en posiciones consecutivas en la memoria de la computadora. Dependiendo del lenguaje de programación el índice del array comienza en 0 (lenguaje **C**) o bien en 1 (lenguaje **FORTRAN**); este elemento se almacena en la posición con la dirección más baja.

1. Un array unidimensional (vector o lista) es una estructura de datos que se puede utilizar para almacenar una lista de valores del mismo tipo de datos. Tales arrays se pueden declarar dando el tipo de datos de los valores que se van a almacenar y el tamaño del array. Por ejemplo, en **C/C++** la declaración

   ```
   int num[100]
   ```

   crea un array de 100 elementos, el primer elemento es `num[0]` y el último elemento es `num[99]`.
2. Los elementos del array se almacenan en posiciones contiguas en memoria y se referencian utilizando el nombre del array y un subíndice; por ejemplo, **`num[25]`**. Cualquier expresión de valor entero no negativo se puede utilizar como subíndice y el subíndice 0 (en el caso de **C**) o 1 (caso de **FORTRAN**) siempre se refieren al primer elemento del array.
3. Se utilizan arrays para almacenar grandes colecciones de datos del mismo tipo. Esencialmente en los casos siguientes:
   - Cuando los elementos individuales de datos se deben utilizar en un orden aleatorio, como es el caso de los elementos de una lista o los datos de una tabla.
   - Cuando cada elemento representa una parte de un dato compuesto, tal como un vector, que se utilizará repetidamente en los cálculos.
   - Cuando los datos se deben procesar en fases independientes, como puede ser el cálculo de una media aritmética o varianza.
4. Un array de dos dimensiones (tabla) se declara listando el tamaño de las filas y de las columnas junto con el nombre del array y el tipo de datos que contiene. Por ejemplo, las declaraciones en **C** de

   ```
   int tabla1[5][10]
   ```

   crean un array bidimensional de cinco filas y 10 columnas de tipo entero.
5. *Arrays paralelos*. Una tabla multicolumna se puede representar como un conjunto de arrays paralelos, un array por columna, de modo que todos tengan la misma longitud y se accede utilizando la misma variable de subíndice.
6. Los arrays pueden ser, de modo completo o por elementos, pasados como parámetros a funciones y a su vez ser argumentos de funciones.
7. La longitud de un array se fija en su declaración y no puede ser modificada sin una nueva declaración. Esta característica los hace a veces poco adecuados para aplicaciones que requieren de tamaños o longitudes variables.

EJERCICIOS

7.1. Determinar los valores de I, J, después de la ejecución de las instrucciones siguientes:

```
var
  entero : I, J
  array[1..10] de entero : A
inicio
  I ← 1
  J ← 2
  A[I] ← J
  A[J] ← I
  A[J+I] ← I + J
  I ← A[I] + A[J]
  A[3] ← 5
  J ← A[I] - A[J]
fin
```

7.2. Escribir el algoritmo que permita obtener el número de elementos positivos de una tabla.

7.3. Rellenar una matriz identidad de 4 por 4.

7.4. Leer una matriz de 3 por 3 elementos y calcular la suma de cada una de sus filas y columnas, dejando dichos resultados en dos vectores, uno de la suma de las filas y otro de las columnas.

7.5. Cálculo de la suma de todos los elementos de un vector, así como la media aritmética.

7.6. Calcular el número de elementos negativos, cero y positivos de un vector dado de sesenta elementos.

7.7. Calcular la suma de los elementos de la diagonal principal de una matriz cuatro por cuatro (4 × 4).

7.8. Se dispone de una tabla T de cincuenta números reales distintos de cero. Crear una nueva tabla en la que todos sus elementos resulten de dividir los elementos de la tabla T por el elemento T[K], siendo K un valor dado.

7.9. Se dispone de una lista (vector) de N elementos. Se desea diseñar un algoritmo que permita insertar el valor x en el lugar k-ésimo de la mencionada lista.

7.10. Se desea realizar un algoritmo que permita controlar las reservas de plazas de un vuelo MADRID-CARACAS, de acuerdo con las siguientes normas de la compañía aérea:

Número de plazas del avión: 300.
Plazas numeradas de 1 a 100: fumadores.
Plazas numeradas de 101 a 300: no fumadores.

Se debe realizar la reserva a petición del pasajero y cerrar la reserva cuando no haya plazas libres o el avión esté próximo a despegar. Como ampliación de este algoritmo, considere la opción de anulaciones imprevistas de reservas.

7.11. Cada alumno de una clase de licenciatura en Ciencias de la Computación tiene notas correspondientes a ocho asignaturas diferentes, pudiendo no tener calificación en alguna asignatura. A cada asignatura le corresponde un determinado coeficiente. Escribir un algoritmo que permita calcular la media de cada alumno.

Modificar el algoritmo para obtener las siguientes medias:

- general de la clase
- de la clase en cada asignatura
- porcenaje de faltas (no presentado a examen)

7.12. Escribir un algoritmo que permita calcular el cuadrado de los 100 primeros números enteros y a continuación escribir una tabla que contenga dichos cuadrados.

7.13. Se dispone de N temperaturas almacenadas en un array. Se desea calcular su media y obtener el número de temperaturas mayores o iguales que la media.

7.14. Calcular la suma de todos los elementos de un vector de dimensión 100, así como su media aritmética.

7.15. Diseñar un algoritmo que calcule el mayor valor de una lista L de N elementos.

7.16. Dada una lista L de N elementos, diseñar un algoritmo que calcule de forma independiente la suma de los números pares y la suma de los números impares.

7.17. Escribir el algoritmo que permita escribir el contenido de una tabla de dos dimensiones (3×4).

7.18. Leer una matriz de 3×3.

7.19. Escribir un algoritmo que permita sumar el número de elementos positivos y el de negativos de una tabla T de n filas y m columnas.

7.20. Se dispone de las notas de cuarenta alumnos. Cada uno de ellos puede tener una o varias notas. Escribir un algoritmo que permita obtener la media de cada alumno y la media de la clase a partir de la entrada de las notas desde el terminal.

7.21. Una empresa tiene diez almacenes y necesita crear un algoritmo que lea las ventas mensuales de los diez almacenes, calcular la media de ventas y obtener un listado de los almacenes cuyas ventas mensuales son superiores a la media.

7.22. Se dispone de una lista de cien números enteros. Calcular su valor máximo y el orden que ocupa en la tabla.

7.23. Un avión dispone de ciento ochenta plazas, de las cuales sesenta son de "no fumador" y numeradas de 1 a 60 y ciento veinte plazas numeradas de 61 a 180 de "fumador". Diseñar un algoritmo que permita hacer la reserva de plazas del avión y se detenga media hora antes de la salida del avión, en cuyo momento se abrirá la lista de espera.

7.24. Calcular las medias de las estaturas de una clase. Deducir cuántos son más altos que la media y cuántos más bajos que dicha media.

7.25. Las notas de un colegio se tienen en una matriz de 30×5 elementos (30, número de alumnos; 5, número de asignaturas). Se desea listar las notas de cada alumno y su media. Cada alumno tiene como mínimo dos asignaturas y máximo cinco, aunque los alumnos no necesariamente todos tienen que tener cinco materias.

7.26. Dado el nombre de una serie de estudiantes y las calificaciones obtenidas en un examen, calcular e imprimir la calificación media, así como cada calificación y la diferencia con la media.

7.27. Se introducen una serie de valores numéricos desde el teclado, siendo el valor final de entrada de datos o centinela –99. Se desea calcular e imprimir el número de valores leídos, la suma y media de los valores y una tabla que muestre cada valor leído y sus desviaciones de la media.

7.28. Se dispone de una lista de N nombres de alumnos. Escribir un algoritmo que solicite el nombre de un alumno y busque en la lista (array) si el nombre está en la lista.