



## Capítulo 11

# Tipos de datos simples y compuestos

---

11.1 Tipos ordinales definidos por el programador . . . . .	234
11.2 Definición de tipos . . . . .	240
11.3 Conjuntos . . . . .	244
11.4 Ejercicios . . . . .	250

---

Ya hemos visto que los programas se describen en términos de acciones y datos. En cuanto a las acciones, se ha mostrado que admiten un tratamiento estructurado, pudiéndose combinar mediante unos pocos esquemas: la secuencia, la selección y la repetición. De igual forma se pueden estructurar los datos. Hasta ahora sólo hemos trabajado con los tipos de datos que están predefinidos en Pascal (`integer`, `real`, `char` y `boolean`), pero en muchas situaciones se manejan unidades de información que necesitan algo más que un dato predefinido, por ejemplo:

- Un color del arco iris, (rojo, naranja, amarillo, verde, azul, añil, violeta) cuya representación mediante un carácter o un entero sería forzosamente artificiosa.
- El valor de un día del mes, que en realidad no es un entero cualquiera, sino uno del intervalo  $[1,31]$ , por lo que sería impreciso usar el tipo `integer`.
- El conjunto de letras necesarias para formar una cierta palabra.

- Un vector del espacio  $\mathbb{R}^n$ , un crucigrama o un mazo de la baraja española.
- Una ficha de un alumno donde se recoja su nombre, dirección, edad, teléfono, calificación, D.N.I. y cualquier otro tipo de información necesaria.
- Una carta.

Para poder tratar con datos como los descritos y otros muchos, Pascal permite introducir *tipos de datos definidos por el programador*.

Así, podemos clasificar los datos en dos grandes grupos:

1. Los tipos de datos *simples* que son aquéllos cuyos valores representan un dato atómico. Dentro de estos tipos de datos se encuentran los tipos predefinidos *integer*, *real*, *boolean* y *char* junto con los nuevos tipos *enumerado* y *subrango*, los cuales permiten recoger datos como, por ejemplo, los colores del arco iris y los días del mes, respectivamente.
2. Los tipos de datos *compuestos* que son aquéllos cuyos valores pueden englobar a varios datos simultáneamente. Los tipos de datos compuestos son: el tipo *conjunto* (que permite expresar el caso del conjunto de letras de una palabra), el tipo *array*<sup>1</sup> (que recoge los ejemplos de vectores, crucigramas o una baraja), el tipo *registro* (con cuyos valores se pueden representar fichas de alumnos), y el tipo *archivo* (en uno de cuyos valores se puede almacenar una carta).

Una vez vista la necesidad de ampliar la gama de tipos de datos disponibles, vamos a estudiar cada uno de los tipos de datos definidos por el programador (tanto los simples como los compuestos) mencionados anteriormente. Como en todo tipo de datos, tendremos que precisar su dominio (los valores que pertenecen a él) y las operaciones sobre éste.

En este capítulo se van a estudiar los tipos de datos más sencillos (*enumerado*, *subrango* y *conjunto*) y, junto con ellos, algunos conceptos generales, válidos para todos los tipos de datos definidos por el programador.

## 11.1 Tipos ordinales definidos por el programador

Los tipos de datos simples que el programador puede definir son los tipos *enumerado* y *subrango*. Éstos, junto con los tipos de datos predefinidos, son la base para construir los tipos de datos compuestos.

---

<sup>1</sup>No existe una traducción clara al castellano del término *array*, por lo que seguiremos usando este nombre en lo que sigue.

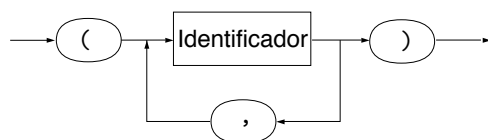


Figura 11.1.

Veamos un ejemplo en el que señalaremos la utilidad y necesidad de ampliar los tipos de datos predefinidos con estos dos nuevos tipos de datos: supongamos que deseamos hacer un horario de estudio para todos los días de la semana durante todo un año. En este caso será útil definir tipos de datos que contengan los meses (al que llamaremos tipo `tMeses`), los días de cada mes (tipo `tDiasMes`) y los días de la semana (tipo `tDiasSemana`) para poder hacer planes del tipo:

Miércoles 1 de Junio: estudiar los temas 12 y 13 de Matemáticas

Nuestro interés es definir los tipos `tDiasSemana` y `tMeses` enumerando uno a uno todos sus posibles valores y, limitar el rango del tipo `integer` a los números enteros comprendidos entre 1 y 31 para definir el tipo `tDiasMes`. Veamos que Pascal permite crear estos tipos de una forma muy cómoda.

### 11.1.1 Tipos enumerados

Un problema puede precisar de un determinado tipo de dato cuyos valores no están definidos en Pascal (como ocurre en el ejemplo anterior con los días de la semana). Podríamos optar por numerar los valores haciendo corresponder a cada valor un número entero que lo represente (por ejemplo, identificar los días de la semana con los números del 1 al 7), pero esta solución no es muy comprensible y, además, podría fácilmente conducirnos a errores difíciles de encontrar por la posibilidad de mezclarlos con los enteros “de verdad” y la posibilidad de aplicar operaciones sin sentido. Para solucionar este problema, Pascal nos proporciona la posibilidad de definir los datos de tipo *enumerado*. Es importante señalar que una vez que definamos un tipo de datos que contenga los días de la semana, este tipo va a ser completamente distinto del tipo predefinido `integer`. El diagrama sintáctico para la descripción de un tipo enumerado aparece en la figura 11.1.

Por ejemplo, para definir un tipo compuesto por los días de la semana incluimos:

```

type
  tDiasSemana = (lun, mar, mie ,jue, vie, sab, dom);
  
```

que se situará antes de la declaración de variables.

- ☉☉ Obsérvese que los valores de un tipo enumerado son identificadores, y no cadenas de caracteres (es decir, no son literales de Pascal, como los descritos en el apartado 3.6).

Como se puede observar, basta con enumerar los valores del tipo uno a uno, separándolos por comas y encerrándolos entre paréntesis. A partir de este momento Pascal reconoce el identificador `tDiasSemana` como un nuevo nombre de tipo de datos del cual se pueden declarar variables:

```
var
  ayer, hoy, manana: tDiasSemana;
```

Como se podría esperar, el dominio de un tipo enumerado está formado por los valores incluidos en su descripción.

Para asignar valores de tipo enumerado a las variables, se usa el operador de asignación habitual:

```
ayer := dom;
hoy := lun;
```

### Operaciones de los tipos enumerados

En todo tipo enumerado se tiene un orden establecido por la descripción del tipo y, por lo tanto, los tipos enumerados son tipos ordinales al igual que los tipos predefinidos `integer`, `char` y `boolean` (véase el apartado 3.6). De este modo, los operadores relacionales son aplicables a los tipos enumerados, y el resultado de su evaluación es el esperado:

```
lun < mie ~> True
jue = sab ~> False
(mar > lun) = (lun < mar) ~> True
```

Además de los operadores relacionales, son también aplicables las funciones ya conocidas `Ord`, `Pred` y `Succ`:

```
Succ(jue) ~> vie
Succ(lun) = Pred(mie) ~> True
Pred(lun) ~> error (ya que no existe el anterior del primer valor)
```

$\text{Succ}(\text{dom}) \rightsquigarrow \text{error}$  (ya que no existe el siguiente al último valor)

$\text{Ord}(\text{jue}) \rightsquigarrow 3$

Como en todo tipo ordinal, salvo en el caso de los enteros, el número de orden comienza siempre por cero.

### Observaciones sobre los tipos enumerados

- Dado que los tipos enumerados son ordinales, se pueden utilizar como índices en instrucciones **for**:

```
var
  d: tDiasSemana;
  ...
  for d:= lun to dom do
  ...
```

- Pueden pasarse como parámetros en procedimientos y funciones. Incluso, por tratarse de un tipo simple, puede ser el resultado de una función:

```
function DiaMannana(hoy: tDiasSemana): tDiasSemana;
  {Dev. el día de la semana que sigue a hoy}
begin
  if hoy = dom then
    DiaMannana:= lun
  else
    DiaMannana:= Succ(hoy)
end; {DiaMannana}
```

- Los valores de un tipo enumerado no se pueden escribir directamente. Para resolver este problema hay que emplear un procedimiento con una instrucción **case** que escriba, según el valor de la variable del tipo enumerado, la cadena de caracteres correspondiente al identificador del valor. El código de dicho procedimiento podría ser:

```
procedure EscribirDiaSemana(dia: tDiasSemana);
  {Efecto: escribe en el output el nombre de dia}
begin
  case dia of
    lun: WriteLn('Lunes');
    mar: WriteLn('Martes');
    ...
```



Figura 11.2.

```

    dom: WriteLn('Domingo')
  end {case}
end; {EscribirDiaSemana}

```

De igual forma, tampoco se puede leer directamente un valor de un tipo enumerado, por lo que, en caso necesario, habrá que desarrollar una función análoga al procedimiento `EscribirDiaSemana` que lea un valor de un tipo enumerado devolviéndolo como resultado de la función.

- No se pueden repetir valores en distintas descripciones de tipos enumerados ya que, en tal caso, su tipo sería ambiguo. Así, el siguiente fragmento es erróneo:

```

type
  tAmigos = (pepe, juan, pedro, miguel);
  tEnemigos = (antonio, pedro, enrique);

```

En este caso, la ambigüedad se refleja en que no se puede saber si el valor de `Succ(pedro)` es `miguel` o `enrique` ni si `Ord(pedro)` es 1 ó 2.

### 11.1.2 Tipo subrango

El tipo de datos subrango se utiliza cuando se quiere trabajar con un intervalo de un dominio ordinal ya existente, bien de un tipo predefinido, o bien de un tipo creado con anterioridad. A este dominio ordinal lo denominamos tipo base, ya que va a ser el tipo sobre el que se define el tipo subrango.

La descripción de un tipo subrango se hace según el diagrama sintáctico de la figura 11.2

Los valores de las constantes tienen que ser de un mismo tipo ordinal y son los que van a delimitar el intervalo con el que se trabajará. Tienen que estar en orden creciente, esto es,  $\text{Ord}(\text{constante1}) \leq \text{Ord}(\text{constante2})$ , para que el intervalo tenga sentido.

Como ejemplo, consideremos las siguientes descripciones:

```
type
  tNaturales = 1..MaxInt;
  tDiasMes = 1..31;
  tContador = 1..20;
```

El uso de tipos subrango es muy aconsejable debido a que:

- El compilador puede comprobar que el valor almacenado en una variable de un tipo subrango se encuentra en el intervalo de definición, produciendo el correspondiente error en caso contrario (véase el apartado C.3.3).
- Proporcionan una mayor claridad, ya que el compilador verifica la consistencia de los tipos usados, lo que obliga al programador a una disciplina de trabajo clara y natural.

Como es lógico, el dominio de un tipo subrango estará formado por la parte del dominio de su tipo base que indique el intervalo considerado en la descripción.

### Operaciones del tipo subrango

Un tipo subrango hereda todas las funciones y operaciones de su tipo base, por lo tanto se permiten realizar asignaciones, comparaciones, pasar como parámetros en procedimientos y funciones e incluso ser el resultado de una función. Así, aunque el procedimiento `EscribirDiaSemana` tenía como parametro una variable de tipo `tDiasSemana`, la herencia recibida por el tipo subrango permite realizar la siguiente instrucción:

```
type
  tDiasSemana = (lun, mar, mie, jue, vie, sab, dom);
  tLaborables = lun..vie;

var
  d: tLaborables;
...
  EscribirDiaSemana(d)
```

### Observaciones sobre el tipo subrango

- El tipo base puede ser cualquier tipo ordinal, es decir, `char`, `integer`, `boolean` o un tipo enumerado definido anteriormente, como ocurre en el tipo `tLaborables` y en el siguiente ejemplo:

```

type
  tMeses = (ene, feb, mar, abr, may, jun, jul, ago, sep,
           oct, nov, dic);
  tVerano = jul..sep;

```

Los valores de un tipo subrango conservan el orden de su tipo base.

- Turbo Pascal verifica las salidas de rangos siempre que se le indique que lo haga (véase el apartado C.3.3).

## 11.2 Definición de tipos

Como hemos visto, Pascal permite al programador utilizar tipos de datos propios y para ello es necesario que éste defina los tipos que quiere crear. La definición de un tipo consiste, básicamente, en dar nombre al nuevo tipo y especificar cuáles serán sus valores, esto es, nombrar el tipo y definir su dominio. Una vez que se haya definido un tipo, ya podremos declarar variables no sólo de los tipos predefinidos, sino también de este nuevo tipo definido por el programador. Naturalmente, la definición de tipos debe situarse antes de la declaración de variables.

Aunque en la mayoría de los casos se puede obviar la definición de tipos, ésta es muy recomendable para desarrollar programas más legibles, claros y no redundantes. Este hecho será explicado más adelante.

Las definiciones de tipo se sitúan entre las definiciones de constantes y de variables, como se muestra en la figura 11.3. El diagrama sintáctico para la definición de tipos es el de la figura 11.4, donde la palabra reservada **type** indica el comienzo de la definición de tipos, **identificador**<sup>2</sup> es el nombre que deseamos ponerle y **Tipo** es la descripción del tipo que vamos a nombrar.

Por ejemplo, en el caso de los tipos simples la descripción del tipo puede ser:

1. Una enumeración de sus valores (tipos *enumerados*):

```

type
  tColores = (rojo, azul, amarillo, negro, blanco);

```

pudiendo entonces hacer declaraciones de variables como:

---

<sup>2</sup>En este texto se seguirá el convenio de anteponer la letra **t** en los identificadores de los tipos de datos.

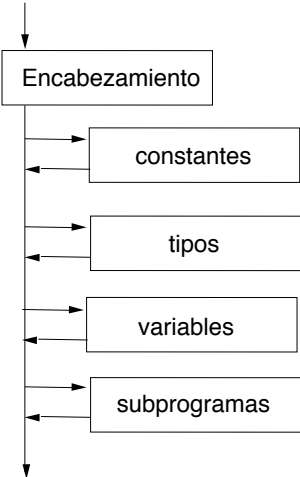


Figura 11.3.

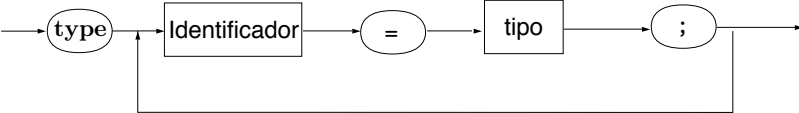


Figura 11.4.

```
var
  color: tColores;
```

2. Un intervalo de un tipo ordinal existente, sea predefinido o enumerado (es decir, cualquier tipo *subrango*):

```
type
  tColores = (rojo, azul, amarillo, negro, blanco);
  tNatural = 0..MaxInt;
  tPrimarios = rojo..amarillo;
```

pudiendo entonces declarar variables como:

```
var
  color: tPrimarios;
  contador: tNatural;
```

Como veremos más adelante, para poder definir tipos compuestos, se usarán palabras reservadas (tales como **set**, **array**, **record** o **file**) para los tipos de datos *conjunto*, *array*, *registro* y *archivo* respectivamente.

### 11.2.1 Observaciones sobre la definición de tipos

Además de las características señaladas anteriormente para las definiciones de tipos, podemos hacer las siguientes observaciones:

1. Hasta ahora, hemos estado utilizando variables de los tipos predefinidos o bien de tipos definidos anteriormente con una instrucción **type**. Sin embargo, existe otra forma de declarar variables de tipo enumerado o subrango sin necesidad de definir el tipo previamente. Para ello se incluye directamente la descripción del tipo enumerado o subrango en la zona de declaración de variables. Por ejemplo:

```
var
  dia : (lun, mar, mie, jue, vie, sab, dom);
  diasMes : 1..31;
```

Esta forma de definición de tipo recibe el nombre de *tipos anónimos* y las variables declaradas así reciben el nombre de *variables de tipo anónimo*. En cualquier caso, su utilización no es recomendable, sobre todo si se van a utilizar varias variables de un mismo tipo anónimo en distintos subprogramas del mismo programa, ya que habría que definir el tipo cada vez que necesitemos alguna variable local de estos tipos.

2. Se pueden renombrar los tipos predefinidos en una definición de tipos:

```
type
  tEntero = integer;
```

El renombramiento, aunque no es recomendable, puede ser útil para evitar memorizar algún tipo predefinido; así, en el ejemplo anterior, podremos hacer declaraciones de variables de tipo `tEntero` y no necesitaremos recordar que en Pascal sería `integer`:

```
var
  n, i, j : tEntero;
```

3. Pascal permite redefinir los tipos predefinidos:

```
type
  boolean = (falso, verdadero);
```

De todas formas, no es recomendable redefinir un tipo predefinido, ya que el código resultante puede ser confuso y difícil de entender o modificar por otra persona distinta de la que lo ha escrito.

4. No se pueden redefinir palabras reservadas como valores de un tipo enumerado ni como nombres de tipo:

```
type
  notas = (do, re, mi, fa, sol, la, si);
  while = (nada, poco, bastante, mucho);
```

En el ejemplo anterior, la primera definición de tipo enumerado es incorrecta porque uno de sus valores (`do`) es una palabra reservada; tampoco es posible hacer la segunda definición porque el nombre (`while`) utilizado para el tipo es una palabra reservada.

5. Es muy importante, como se ha comentado anteriormente, la elección del identificador de tipo a la hora de definir un tipo subrango o enumerado de forma que permita identificar claramente lo que queremos definir.
6. No se puede poner la definición de un tipo (anónimo) en el encabezamiento de un subprograma, por lo que el siguiente encabezamiento sería incorrecto:

```
function MannanaMes(d: 1..31) : 1..31;
```

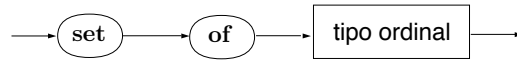


Figura 11.5.

Es preciso definir previamente el tipo y hacer referencia a su identificador en el encabezamiento, como se muestra a continuación:

```

type
  tDiaMes = 1..31;

function MannanaMes(d: tDiaMes): tDiaMes;

```

### 11.3 Conjuntos

El primer tipo de datos compuesto que vamos a estudiar en Pascal es el tipo *conjunto*, que intenta representar el concepto de los conjuntos utilizados en Matemáticas. Así, podemos definir un conjunto como una colección de objetos de un tipo ordinal. En efecto, los elementos de un conjunto no ocupan en él una posición determinada;<sup>3</sup> simplemente, se puede decir que pertenecen o no al mismo.

El tipo de los elementos que integran el conjunto se llama *tipo base*, que en Pascal debe ser ordinal (véase el apartado 3.6). La definición de un tipo de datos conjunto se hace según el diagrama sintáctico de la figura 11.5.

- ☞ No hay que confundir los elementos de un conjunto  $C$  con el dominio del tipo **set of**  $C$ , que sería  $\mathcal{P}(C)$ , es decir el conjunto formado por todos los posibles subconjuntos de  $C$ .

El cardinal máximo de un conjunto en Pascal depende del compilador que estemos utilizando. Por ejemplo, en Turbo Pascal se admiten conjuntos de hasta 256 elementos (con ordinales comprendidos entre 0 y 255), por lo que no podremos definir:

```

type
  tConjunto1 = set of 1..2000;
  tConjunto2 = set of integer;

```

---

<sup>3</sup>Nosotros vamos a considerar los conjuntos como datos compuestos (por ninguno o más elementos) aunque sin estructura, por carecer de organización entre sus elementos. Sin embargo, algunos autores clasifican los conjuntos como un dato estructurado.

En nuestros programas utilizaremos conjuntos con una cardinalidad menor. Por ejemplo:

```
type
  tConjuntoCar = set of char;
```

Una vez definido, podemos declarar variables de dicho tipo:

```
var
  vocales, letras, numeros, simbolos, vacio: tConjuntoCar;
```

Para asignar valores a un conjunto se utiliza la instrucción usual de asignación, representándose los elementos del conjunto entre corchetes. Dentro del conjunto podemos expresar los valores uno a uno o indicando un intervalo abreviadamente (con una notación similar a la utilizada para definir un tipo subrango):

```
vocales:= ['A', 'E', 'I', 'O', 'U'];
letras:= ['A'..'Z'];
simbolos:= ['#'..'&', '?'];
vacio:= []
```

### 11.3.1 Operaciones sobre el tipo conjunto

Las operaciones que se pueden realizar con los conjuntos son las que normalmente se utilizan con los conjuntos matemáticos, es decir: unión, intersección, diferencia, igualdad, desigualdad, inclusión y pertenencia. Pasamos ahora a ver la descripción y efecto de cada una de ellas:

1. *Unión* de conjuntos ( $\cup$ ): se expresa con el signo  $+$ , y su resultado es el conjunto formado por todos los elementos que pertenecen al menos a uno de los conjuntos dados:

```
['A'..'C'] + ['B'..'D', 'G'] + ['A', 'E', 'I', 'O', 'U']  $\rightsquigarrow$ 
['A', 'B', 'C', 'D', 'E', 'G', 'I', 'O', 'U']
```

2. *Intersección* de conjuntos ( $\cap$ ): se expresa con el signo  $*$ , y su resultado es el conjunto formado por los elementos comunes a todos los conjuntos dados:

```
['A'..'C'] * ['B'..'F']  $\rightsquigarrow$  ['B', 'C']
```

3. *Diferencia* de conjuntos ( $\setminus$ ): se expresa con el signo  $-$ , y su resultado es el conjunto formado por los elementos que pertenecen al primer conjunto y no pertenecen al segundo:

```
['A'..'C'] - ['B'..'F']  $\rightsquigarrow$  ['A']
```

4. *Igualdad* de conjuntos ( $=$ ): se utiliza el símbolo  $=$  y representa la relación de igualdad de conjuntos (iguales elementos):

$$['A' .. 'D'] = ['A', 'B', 'C', 'D'] \rightsquigarrow \text{True}$$

5. *Desigualdad* de conjuntos ( $\neq$ ): se utiliza el símbolo  $\langle \rangle$  que representa la relación de desigualdad de conjuntos:

$$['A' .. 'D'] \langle \rangle ['A', 'B', 'C', 'D'] \rightsquigarrow \text{False}$$

6. *Inclusión* de conjuntos ( $\subseteq, \supseteq$ ): se utilizan dos símbolos:

- El símbolo  $\leq$  que representa relación de inclusión *contenido en*:

$$['A' .. 'D'] \leq ['A', 'C', 'D', 'E'] \rightsquigarrow \text{False}$$

- El símbolo  $\geq$  que representa la relación de inclusión *contiene a*:

$$['A' .. 'Z'] \geq ['A', 'H', 'C'] \rightsquigarrow \text{True}$$

7. *Pertenencia* ( $\in$ ): se utiliza para saber si un elemento pertenece a un conjunto. Para ello se utiliza la palabra reservada **in**:

$$'A' \text{ in } ['A' .. 'D'] * ['A', 'D', 'F'] \rightsquigarrow \text{True}$$

Las operaciones unión, intersección, diferencia y los operadores relacionales y el de pertenencia se pueden combinar en una misma sentencia, en cuyo caso se mantienen las reglas de prioridad descritas en el apartado 3.5. Estas prioridades se pueden cambiar mediante el uso de paréntesis:

$$['A' .. 'D'] + ['A' .. 'C'] * ['B' .. 'H'] \rightsquigarrow ['A' .. 'D']$$

mientras que

$$(['A' .. 'D'] + ['A' .. 'C']) * ['B' .. 'H'] \rightsquigarrow ['B' .. 'D']$$

Un ejemplo típico de aplicación de conjuntos es simplificar las condiciones de un **repeat** o un **if**:

```

type
  tDiasSemana = (lun, mar, mie, jue, vie, sab, dom);
procedure LeerDiaSemana(var dia: tDiasSemana);
  {Efecto: dia es el día de la semana cuya inicial se
  ha leído en el input}
  var
    inicial : char;
begin
  repeat
    Write('Escriba la inicial del día de la semana: ');
    ReadLn(inicial)
  until inicial in ['L','l','M','m','X','x','J','j','V','v',
                   'S','s','D','d'];

```

```

case inicial of
  'L','l': dia:= lun;
  'M','m': dia:= mar;
  'X','x': dia:= mie;
  'J','j': dia:= jue;
  'V','v': dia:= vie;
  'S','s': dia:= sab;
  'D','d': dia:= dom
end {case}
end; {LeerDiaSemana}

```

Piense el lector cómo se complicaría el código de este procedimiento en caso de no usar conjuntos.

### 11.3.2 Observaciones sobre el tipo conjunto

Además de lo anteriormente dicho sobre el tipo de datos conjunto, cabe considerar las siguientes observaciones:

1. Podemos definir un conjunto de forma anónima, en el sentido explicado anteriormente:

```

var
  vocales, letras, numeros, simbolos, vacio: set of char;

```

2. Al igual que para el tipo de datos enumerado, los conjuntos no se pueden leer o escribir directamente, por lo que se tendrán que desarrollar procedimientos para tal fin como los que se muestran a continuación:

```

type
  tLetrasMayusculas = 'A'..'Z';
  tConjuntoLetras = set of tLetrasMayusculas;

procedure EscribirConjunto(letras: tConjuntoLetras);
  {Efecto: se muestran en la pantalla los elementos
  del conjunto letras}
  var
    car: char;
begin
  for car:= 'A' to 'Z' do
    if car in letras then
      Write(car, ' ')
  end; {EscribirConjunto}

```

```

procedure LeerConjunto(var conj: tConjuntoLetras);
  {Efecto: conj contiene las letras leídas del input}
  var
    car: char;
begin
  conj:= [];
  WriteLn('Escribe las letras que forman el conjunto: ');
  while not EoLn do begin
    Read(car);
    if car in ['A'..'Z']
      then conj:= conj + [car]
  end; {while}
  ReadLn
end; {LeerConjunto}

```

El primer procedimiento muestra en la pantalla todos los elementos de un conjunto formado por letras mayúsculas. Para ello recorre los valores del tipo base y comprueba la pertenencia de cada uno de ellos al conjunto, escribiéndolo en su caso.

El segundo procedimiento lee del **input** los elementos de un conjunto de letras mayúsculas. Para ello recorre los caracteres de una línea (hasta la marca ↵), incluyendo en el conjunto los del tipo base ('A'..'Z') e ignorando los demás.

### 11.3.3 Un ejemplo de aplicación

Es conveniente concretar todas estas ideas mediante un ejemplo. En este caso vamos a utilizar el tipo de datos conjunto para implementar un programa que escriba los números primos menores que 256 (esta cota viene dada por la limitación del cardinal de un conjunto en Pascal) usando el conocido método de la *criba de Eratóstenes*.<sup>4</sup> La idea de esta implementación es disponer de un conjunto inicial con todos los enteros positivos menores que 256 e ir eliminando del conjunto aquellos números que se vaya sabiendo que no son primos (por ser múltiplos de otros menores). De acuerdo con esta descripción, una primera etapa de diseño del programa podría ser:

*Calcular los números primos menores que 256*  
*Escribir los números primos menores que 256*

---

<sup>4</sup>Eratóstenes, conocido astrónomo y geógrafo griego (siglo III a. C.), es reconocido por haber ideado este método para elaborar una tabla de números primos. El nombre de *criba* se debe a que el algoritmo va eliminando los números que son múltiplos de otros menores, siendo los primos aquellos que quedan tras realizar esta criba.

Y en un segundo refinamiento de *Calcular los números primos menores que 256* se obtendría:<sup>5</sup>

*Generar el conjunto inicial primos  
para cada elemento elem mayor que 1 y menor o igual que 16  
Eliminar del conjunto todos sus múltiplos*

Y en un nivel de refinamiento inferior, *Eliminar del conjunto todos sus múltiplos* se puede desarrollar así:

*Dar valor inicial (igual a 2) al coeficiente k  
repetir  
Eliminar elem \* k del conjunto primos  
Incrementar k en una unidad  
hasta que elem \* k sea mayor o igual que 256*

Desde este nivel de refinamiento es fácil pasar a una implementación en Pascal, como puede ser la siguiente:

```

Program Criba (output);
  {Halla y escribe los primeros números primos}
  const
    N = 255;
  type
    tConjuntoPositivos = set of 1..N;
  var
    primos: tConjuntoPositivos;
    elem, k: integer;

  procedure EscribirConjuntoPositivos(c: tConjuntoPositivos);
    {Efecto: se muestran en la pantalla los elementos
    del conjunto c}
    var
      i: integer;
    begin
      for i:= 1 to N do
        if i in c then
          WriteLn(i : 3, ' es primo')
    end; {EscribirConjuntoPositivos}

```

---

<sup>5</sup>Como se comentó en el apartado 8.2.1, basta con eliminar los múltiplos de los números naturales menores o iguales que la raíz cuadrada de la cota (en nuestro ejemplo, los menores o iguales que  $\sqrt{256} = 16$ ).

```

begin {Criba}
  primos:= [1..N];
  {Se genera el conjunto inicial}
  elem:= 2;
  while elem < Sqrt(N) do begin
    {Inv.: Para todo  $i \in \text{primos}$  tal que  $1 \leq i < \text{elem}$ ,
    se tiene que  $i$  es primo}
    if elem in primos then begin
      k:= 2;
      repeat
        primos:= primos - [elem * k];
        {Se eliminan los números no primos del conjunto}
        k:= k + 1
      until elem * k > N
    end; {if}
    elem:= elem + 1
  end; {while}
  WriteLn('Los primos menores que ', N, ' son');
  EscribirConjuntoPositivos(primos)
end. {Criba}

```

## 11.4 Ejercicios

- Se desea averiguar qué letras intervienen en un texto (sin distinguir entre letras mayúsculas y minúsculas ni importar su frecuencia), y cuáles se han omitido.
  - Defina un tipo apropiado para controlar las letras que van apareciendo.
  - Escriba un subprograma que lea el `input`, formado por varias líneas, ofreciendo como resultado el conjunto de las letras encontradas.
  - Escriba un subprograma de escritura de conjuntos de letras.
  - Escriba un subprograma que averigüe el cardinal de un conjunto de letras.
  - Integre los apartados anteriores en un programa que averigüe cuáles y cuántas son las letras distintas encontradas y las omitidas en el `input`.
- Dados los conjuntos  $\text{conjNum} \subset \{1, \dots, 25\}$  y  $\text{conjLet} \subset \{ 'A', \dots, 'Z' \}$ , escriba un subprograma que escriba en la pantalla el producto cartesiano  $\text{conjNum} \times \text{conjLet}$ .
- Un modo de calcular el máximo común divisor de dos números  $m$  y  $n$ , enteros estrictamente positivos no superiores a 50, consiste en hallar los conjuntos de sus divisores, `divisDeM` y `divisDeN`, luego su intersección, `divisComunes`, y después el mayor elemento de este conjunto.  
Desarrolle un programa que realice estos pasos dando una salida detallada del proceso seguido.
- Los meses del año se conocen por su nombre, aunque a veces también se abrevian por su número. Desarrolle los siguientes apartados para ambas representaciones.

- (a) Defina un tipo para representar los meses del año, así como subprogramas apropiados para su lectura, mediante sus letras iniciales, y escritura, ofreciendo siempre su nombre (o su número).
  - (b) Defina la función `ProximoMes`, y empléela en un programa que, a partir de un mes dado, ofrece los siguientes  $n$  meses.
  - (c) Defina la función `MesAnterior`, con un argumento `esteMes`, que repasa la lista de los meses posibles hasta dar con uno, `mesAnt`, de manera que se obtenga `ProximoMes(mesAnt)  $\rightsquigarrow$  esteMes`. Ignore la existencia de la función estándar `Pred`.
  - (d) Integre los apartados anteriores en un programa que escriba una tabla con los meses del año, y para cada uno, su mes anterior y siguiente.
5. Escriba un programa que lea los caracteres del `input` hasta su final, desprecie los que no son letras mayúsculas y, con éstas, forme el conjunto de las que no aparecen, el de las que aparecen una vez y el de las que aparecen más de una.

#### 6. Partes de un conjunto

- (a) Dado un conjunto  $C$ , desarrolle un programa que escriba en la pantalla todos los posibles subconjuntos de  $C$ , esto es, el conjunto de sus partes,  $\mathcal{P}(C)$ .
- (b) Escriba un programa para mostrar que todo conjunto  $C$  tiene  $2^n$  subconjuntos, siendo  $n = \text{card}(C)$ . Para ello, modifique el programa anterior de manera que genere las partes de  $C$  y las cuente en vez de escribirlas en la pantalla.

#### 7. Combinaciones de un conjunto

- (a) Dado un conjunto  $C$ , de cardinal  $m$ , desarrolle un programa que escriba en la pantalla todas las posibles combinaciones que pueden formarse con  $n$  elementos de  $C$ , siendo  $n \leq m$ .
- (b) Escriba un programa que cuente el número de combinaciones descritas en el apartado anterior.



## Capítulo 12

# Arrays

---

<b>12.1 Descripción del tipo de datos array . . . . .</b>	<b>253</b>
<b>12.2 Vectores . . . . .</b>	<b>261</b>
<b>12.3 Matrices . . . . .</b>	<b>263</b>
<b>12.4 Ejercicios . . . . .</b>	<b>268</b>

---

En el capítulo anterior, vimos la diferencia entre tipos de datos simples y compuestos. Dentro de los datos compuestos, se estudiaron los más sencillos (los conjuntos), pero, además de éstos, existen otros datos compuestos que se construyen dotando de una estructura a colecciones de datos pertenecientes a tipos más básicos, y por esto se llaman *tipos estructurados*.

En este capítulo se presenta el tipo de datos estructurado *array*, que será útil para trabajar con estructuras de datos como, por ejemplo, vectores, matrices, una sopa de letras rectangular, una tabla de multiplicar, o cualquier otro objeto que necesite una o varias dimensiones para almacenar su contenido.

### 12.1 Descripción del tipo de datos array

Los arrays son tipos de datos estructurados ampliamente utilizados, porque permiten manejar colecciones de objetos de un mismo tipo con acceso en tiempo constante, y también porque han demostrado constituir una herramienta de enorme utilidad.

Consideremos los siguientes ejemplos en los que veremos la necesidad y la utilidad de usar el tipo de datos array:

- Imaginemos que queremos calcular el producto escalar,  $\vec{u} \cdot \vec{v}$ , de dos vectores  $\vec{u}$  y  $\vec{v}$  de  $\mathbb{R}^3$  mediante la conocida fórmula:  $\vec{u} \cdot \vec{v} = u_1v_1 + u_2v_2 + u_3v_3$ .

Con los datos simples que conocemos hasta ahora, tendríamos que definir una variable para cada una de las componentes de los vectores, es decir, algo parecido a:

```
var
  u1, u2, u3, v1, v2, v3: real;
```

y a la hora de calcular el producto escalar tendríamos que realizar la operación:

```
prodEscalar:= u1 * v1 + u2 * v2 + u3 * v3;
```

Para este caso sería más natural disponer de una “variable estructurada” que agrupe en un solo objeto las componentes de cada vector. El tipo array de Pascal permite resolver este problema.

- Imaginemos que una constructora acaba de finalizar un grupo de 12 bloques de pisos (numerados del 1 al 12), cada uno de los cuales tiene 7 plantas (numeradas del 1 al 7) y en cada planta hay 3 viviendas (A, B y C). Supongamos que el encargado de ventas quiere llevar un control lo más sencillo posible sobre qué viviendas se han vendido y cuáles no. Para ello, podríamos utilizar  $12 \times 7 \times 3 = 252$  variables de tipo `boolean` de la forma: `bloqueiPlantajLetraX` asignándole un valor `True` para indicar que la vivienda del bloque i, planta j, letra X está vendida o bien `False` para indicar que no lo está.

En este caso, sería mucho más cómodo utilizar algún tipo de datos estructurado para almacenar esta información de forma más compacta y manejable (por medio de instrucciones estructuradas). La estructura más adecuada sería la de una matriz tridimensional: la primera dimensión indicaría el número del bloque, la segunda dimensión la planta, y la tercera la letra de la vivienda. Así, para indicar que en el bloque 3, el piso 5<sup>o</sup>A está vendido, asignaremos un valor `True` el elemento que ocupa la posición `[3,5,'A']` de este dato estructurado; mientras que si en el bloque 5, el 1<sup>o</sup>C sigue estando disponible, asignaremos un valor `False` a la posición `[5,1,'C']`.

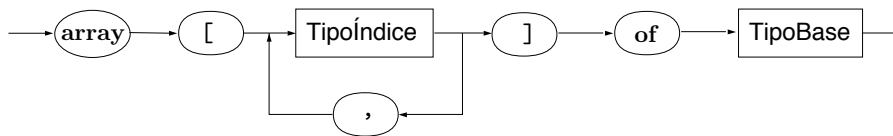
El tipo array ofrece la posibilidad de referirnos a las componentes de un modo genérico, por su posición, lo que hace más cómodo y comprensible el desarrollo de programas.

El tipo estructurado array captura la idea de los vectores y matrices del álgebra (como podrían ser  $\mathbb{R}^3$ ,  $\mathcal{M}_{2 \times 5}(\mathbb{Z})$ ), aunque sus elementos componentes no tienen que ser números: pueden ser de cualquier tipo. A semejanza de estas estructuras matemáticas, los arrays se pueden manipular fácilmente, debido a su organización regular (es fácil verlos como hileras, tablas, estructuras cúbicas, etc.):

$$v = \begin{bmatrix} 0.24 & 3.14 & -3.56 \end{bmatrix} \quad m = \begin{bmatrix} -1 & 2 & -3 & 4 & -5 \\ 6 & -7 & 8 & -9 & 0 \end{bmatrix}$$

$$c = \begin{bmatrix} 'a' & 'b' \\ 'c' & 'd' \\ 'e' & 'f' \end{bmatrix}$$

El diagrama sintáctico de la definición de un array es:



Por lo tanto, su definición en Pascal es de la siguiente forma:

```
array [TipoÍndice1, TipoÍndice2, ..., TipoÍndiceL] of TipoBase
```

o equivalentemente:

```
array [TipoÍndice1] of array [TipoÍndice2] of ...
... array [TipoÍndiceL] of TipoBase
```

Los tipos TipoÍndice1, TipoÍndice2 ... TipoÍndiceL tienen que ser de un tipo simple ordinal, es decir *integer*, *char*, *boolean*, enumerado o subrango. L es el número de dimensiones del array. Por ejemplo, se pueden realizar las siguientes definiciones de tipos y declaraciones de variables:

```
type
  tPlanetas = (mercurio, venus, tierra, marte, jupiter,
               saturno, urano, neptuno, pluton);
  tVector = array[1..3] of real;
  tMatriz = array[1..2, 1..5] of integer;
```

```

tCubo = array[1..2, 1..2, 1..3] of char;
tDistancia = array[tPlanetas, tPlanetas] of real;
tUrbanizacion = array[1..12, 1..7, 'A'..'C'] of boolean;
var
  u, v: tVector;
  m: tMatriz;
  c: tCubo;
  d: tDistancia;
  costaMar: tUrbanizacion;

```

El dominio de un array es el producto cartesiano de los dominios de los tipos de los índices.

Como se ha visto en la definición genérica anterior, los arrays multidimensionales se pueden definir de varias formas (lo vemos para el caso de dos dimensiones):

1. Como un vector de vectores:

```

type
  tMatriz = array[1..8] of array['A'..'E'] of real;

```

2. Como un vector de un tipo definido anteriormente que será otro **array**:

```

type
  tVector = array['A'..'E'] of real;
  tMatriz = array[1..8] of tVector;

```

3. Introduciendo los índices dentro de los corchetes separados por comas:

```

type
  tMatriz = array[1..8, 'A'..'E'] of real;

```

Esta última es la forma más recomendable de definir un array multidimensional, ya que evita posibles errores en el orden de los índices.

Si se quiere definir una variable de tipo `tMatriz` se realiza de la forma usual y es igual para cualquiera de las tres definiciones anteriores, esto es:

```

var
  m : tMatriz;

```

con lo que estaremos declarando una variable que será una matriz de tamaño  $8 \times 5$ . Otra posibilidad es la siguiente:

```

var
  m : array[1..8, 'A'..'E'] of real;

```

### 12.1.1 Operaciones del tipo array y acceso a sus componentes

Las operaciones permitidas con los componentes de un array son las mismas que las permitidas a su tipo base.

#### Acceso y asignación a componentes de un array

Se accede a sus elementos mediante el uso de tantos índices como dimensiones tenga el array, siguiendo el siguiente esquema:

```
idArray [expres1, expres2, ..., expresL]
```

o, equivalentemente:

```
idArray [expres1][expres2]...[expresL]
```

donde `expres1`, `expres2`, ..., `expresL` son expresiones del tipo de los `L` índices de `idArray`, respectivamente. Por ser expresiones, pueden ser literales de los tipos de los índices:

```
v[3]
m[2,3] ≡ m[2][4]
c[1,2,1] ≡ c[1][2][1]
d[venus, tierra]
costaMar[12,3,'B']
```

o resultados de alguna operación; así, por ejemplo, si  $i = 2$ , las siguientes expresiones son equivalentes a las anteriores:

```
v[i+1]
m[i,2*i-1] ≡ m[i][2*i-1]
c[i-1,i,1] ≡ c[i-1][i][1]
d[Succ(mercurio), Pred(marte)]
costaMar[6 * i, i + 1, 'B']
```

Para dar valores a las componentes de un array se usa la instrucción de asignación:

```
v[2] := 3.14
m[i,2 * i - 1] := 8
c[1][2][1] := 'b'
d[mercurio, pluton] := 3.47E38
costaMar[12,3,'B'] := True
```

Al igual que ocurre con las variables de tipo simple, una referencia a una componente de un array puede representar la posición de memoria donde se almacena su valor (como en los ejemplos anteriores), o su valor, si se utiliza dentro de una expresión (como ocurre en la instrucción `Write(v[2])`).

### Asignación y operaciones de arrays completos

Además de la asignación a las componentes de un array, se pueden realizar asignaciones directas de arrays siempre que sean del mismo tipo. Por ejemplo, dadas las definiciones

```

type
  tDiasSemana = (lun, mar, mie, jue, vie, sab, dom);
  tIndice1 = -11..7;
  tIndice2 = 'A'..'Z';
  tIndice3 = lun..vie;
  tMatrizReal = array[Indice1, Indice2, Indice3] of real;
var
  i: tIndice1;
  j: tIndice2;
  k: tIndice3;
  m1, m2: tMatrizReal;

```

la asignación:

```
m2:= m1
```

es equivalente a la instrucción:

```

for i:= -11 to 7 do
  for j:= 'A' to 'Z' do
    for k:= lun to vie do
      m2[i,j,k] := m1[i,j,k]

```

También se pueden realizar asignaciones por filas o columnas siempre que se hayan definido las filas o columnas como tipos con nombre. Por ejemplo,

```

type
  tVector = array[1..3] of real;
  tMatriz = array[1..5] of tVector;
var
  v: tVector;
  m: tMatriz;
  ...
  m[4] := v
  ...

```

En Pascal no es posible comparar arrays completos aunque sean del mismo tipo. Para realizar esta operación es necesario comparar elemento a elemento, comprobando la igualdad entre ellos. Por ejemplo, suponiendo que `m1` y `m2` son matrices de tipo `tMatriz`, la comprobación de su igualdad se puede hacer así:

```

var i, j: integer;
...
iguales := True;
i:= 0;
while iguales and (i <= 3) do begin
  j:= 0;
  while iguales and (j <= 5) do begin
    iguales := m1[i,j] = m2[i,j];
    j := Succ(j)
  end;
  i := Succ(i)
end
{PostC.: iguales indica si m1 = m2 o no}

```

Por otra parte, en cuanto a las operaciones de entrada y salida, solamente se pueden leer y escribir arrays completos cuando se trata de arrays de caracteres. En este caso, y por lo que respecta a la lectura, la cadena leída debe coincidir en longitud con el número de componentes del vector.

En los restantes tipos de componentes de arrays, se deben leer y escribir los valores de las componentes una a una siempre que sean de tipo simple. En el siguiente ejemplo se definen procedimientos para la lectura y escritura de un array de una dimensión:

```

const
  Tamanno = 5;
type
  tRango = 1..Tamanno;
  tVector = array[tRango] of real;
procedure LeerVector(var vec: tVector);
var
  i: tRango;
begin
  for i:= 1 to Tamanno do begin
    Write('Introduzca v(',i,')= ');
    ReadLn(vec[i])
  end
end; {LeerVector}

procedure EscribirVector(vec: vector);
var
  i: tRango;
begin
  for i:= 1 to Tamanno do
    WriteLn('v(',i,')= ',vec[i]);
  WriteLn
end; {EscribirVector}

```

### 12.1.2 Características generales de un array

Además de las operaciones permitidas con un array y la forma de acceder a sus elementos, hay que destacar las siguientes características comunes al tipo array, independientemente de su tamaño o de su dimensión:

- Los arrays son estructuras homogéneas, en el sentido de que sus elementos componentes son todos del mismo tipo.
- El tamaño del array queda fijado en la definición y no puede cambiar durante la ejecución del programa, al igual que su dimensión. Así, el tamaño de cualquier variable del tipo

```
type
  vector = array[1..3] of real;
```

será de 3 elementos, mientras que su dimensión será 1. Como estos valores no podrán variar, si necesitásemos utilizar un vector de  $\mathbb{R}^4$  tendríamos que definir otro tipo array con tamaño 4 y dimensión 1.

- Los datos de tipo array se pueden pasar como parámetro en procedimientos y funciones, pero el tipo que devuelve una función no puede ser un array (ya que un array es un tipo de datos compuesto). Para solucionar este problema, se debe utilizar un procedimiento con un parámetro por variable de tipo array en vez de una función. Este parámetro adicional va a ser el resultado que pretendíamos devolver con la función (se incluyen ejemplos de esto en los apartados siguientes).

Cuando los arrays son grandes y se pasan como parámetro por valor a los subprogramas, se duplica el espacio en memoria necesario para su almacenamiento, puesto que hay que guardar el array original y la copia local (véase el apartado 8.2.3). Además, el proceso de copia requiere también un cierto tiempo, tanto mayor cuanto más grande es el array. Por estos motivos, cuando el array no se modifica en el cuerpo del subprograma, es aconsejable pasarlo como parámetro por referencia, pues de esta forma no hay duplicación de espacio ni tiempo ocupado en la copia (véase el apartado 8.6.1). En cualquier caso, el programador debe cuidar extremadamente las eventuales modificaciones de esa estructura en el subprograma, ya que repercutirían en el parámetro real.

Usualmente, los arrays de una dimensión se suelen denominar *vectores*, mientras que los de más de una dimensión reciben el nombre genérico de *matrices*. En los siguientes apartados se presentan algunas de sus particularidades.

## 12.2 Vectores

En términos generales, un vector es una secuencia, de longitud fija, formada por elementos del mismo tipo. Teniendo en cuenta que un vector es un array de dimensión 1, su definición es sencilla. Por ejemplo:

```

type
  tNumeros = 1..10;
  tDiasSemana = (lun,mar,mie,jue,vie,sab,dom);
  tVectorDeR10 = array[tNumeros] of real;
  tFrase = array[1..30] of char;
  tVectorMuyGrande = array[integer] of real;

```

Hay que observar que el último vector llamado `vectorMuyGrande` sólo se podría definir si dispusiéramos de “muchísima memoria”.<sup>1</sup>

Con las definiciones anteriores se pueden realizar operaciones como:

```

var
  v: tVectorDeR10;
  refran: tFrase;

...
v[4] := 3.141516;
v[2 * 4 - 1] := 2.7172 * v[4];
refran := 'Al_que_madruga,_Dios_le_ayuda.'
...

```

### Ejemplo

Veamos ahora un ejemplo de manejo de vectores en Pascal. Para indicar cuántos viajeros van en cada uno de los 15 vagones de un tren (con una capacidad máxima de 40 personas por vagón), en lugar de utilizar 15 variables enteras (una para cada vagón), se puede y se debe utilizar un vector de la siguiente forma:

```

const
  CapacidadMax = 40;
type
  tCapacidad = 0..CapacidadMax;
  tVagones = array[1..15] of tCapacidad;
var
  vagon : tVagones;

```

---

<sup>1</sup>Por otra parte, se debe señalar que Turbo Pascal facilita un tratamiento más cómodo y directo de las cadenas de caracteres por medio de los llamados *strings* (véase el apartado B.5).

Así, haremos referencia al número de pasajeros del vagón  $i$ -ésimo mediante `vagon[i]`, mientras que el total de viajeros en el tren será

$$\sum_{i=1}^{15} \text{vagon}[i]$$

que en Pascal se calcula como sigue:

```
total:= 0;
for i:= 1 to 15 do
  total:= total + vagon[i]
```

Como todo array, un vector se puede pasar como parámetro en procedimientos y funciones:

```
const
  Tamanno = 5;
type
  tVector = array[1..Tamanno] of real;

function Norma(vec: tVector) : real;
  {Dev.  $\sqrt{\sum_{i=1}^{\text{Tamanno}} \text{vec}_i^2}$ }
  var
    i: 1..Tamanno;
    sumaCuad: real;
begin
  sumaCuad:= 0.0;
  for i:= 1 to Tamanno do
    sumaCuad:= sumaCuad + Sqr(vec[i]);
  Norma:= Sqrt(sumaCuad)
end; {Norma}
```

La función anterior devuelve la norma de un vector de  $\mathbb{R}^{\text{Tamanno}}$ . Supongamos ahora que queremos desarrollar una función tal que, dado un vector, devuelva el vector unitario de su misma dirección y sentido. En este caso no podremos utilizar una función, ya que un vector no puede ser el resultado de una función. Por esta razón tendremos que utilizar un procedimiento cuyo código podría ser el siguiente:

```
procedure HallarUnitario(v: tVector; var uni: tVector);
  {PostC.: para todo  $i$ , si  $1 \leq i \leq \text{Tamanno}$ , entonces  $\text{uni}[i] = \frac{v_i}{\text{Norma}(v)}$ }
  var
    norm: real;
    i: 1..Tamanno;
```

```

function Norma(vec: tVector): real;
  {Dev.  $\sqrt{\sum_{i=1}^{Tamanno} vec_i^2}$  }
  ...
end; {Norma}

begin
  norm:= Norma(v);
  for i:= 1 to Tamanno do
    uni[i]:= v[i]/norm
  end; {HallarUnitario}

```

## 12.3 Matrices

Como ya se dijo, los arrays multidimensionales reciben el nombre genérico de *matrices*. En este apartado se presentan algunos ejemplos de utilización de matrices.

Evidentemente, la forma de definir los tipos de datos para las matrices es la misma de todos los arrays, así como el modo de declarar y manipular variables de estos tipos. Así, por ejemplo:

```

type
  tMeses = (ene, feb, mar, abr, may, jun, jul, ago, sep, oct,
           nov, dic);
  tDiasMes = 1..31;
  tHorasDia = 0..23;
  tFrase = array[1..30] of char;
  tMCuadrada = array[1..5, 1..5] of real;
  tFiestas95 = array[tDiasMes, tMeses] of boolean;
  tAgenda95 = array[tDiasMes, tMeses, tHorasDia] of tFrase;
var
  m: tMCuadrada;
  festivos: tFiestas95;
  agenda: tAgenda95;

begin
  m[1,2]:= Sqrt(2);
  m[2,3-2]:= 2.43 * m[1,2];
  festivos[25,dic]:= True;
  festivos[28,dic]:= False;
  agenda[18,mar,17]:= 'Boda_de_Jose_Luis_y_Mavi.UUUUU'
  ...
end.

```

En el siguiente ejemplo, que calcula el producto de dos matrices reales, se utilizan las operaciones permitidas a los arrays, haciendo uso del hecho de que las matrices se pueden pasar como parámetros en funciones y procedimientos. En un primer nivel de diseño se tiene:

*Leer matrices a y b*  
*Multiplicar matrices a y b, hallando la matriz producto prod*  
*Mostrar la matriz prod*

Dada la simplicidad y familiaridad de los subprogramas a implementar, no se considera necesario un nivel inferior de diseño, razón por la cual se presenta directamente la codificación en Pascal:

```

Program MultiplicacionDeMatrices (input,output);
  {El programa lee dos matrices cuadradas de dimensión N y de
  componentes reales y las multiplica, mostrando la matriz
  producto en la pantalla}
  const
    N = 10;
  type
    tMatriz = array[1..N, 1..N] of real;
  var
    a, b, prod: tMatriz;

  procedure LeerMatriz(var mat : tMatriz);
    {Efecto: Este procedimiento lee del input una matriz
    cuadrada  $mat \in \mathcal{M}_N(\mathbb{R})$ , componente a componente}
    var
      fil, col: 1..N;
    begin
      for fil:= 1 to N do
        for col:= 1 to N do begin
          Write('Introduzca la componente ', fil, ', ', col,
            ' de la matriz: ');
          ReadLn(mat[fil,col])
        end
    end; {LeerMatriz}

  procedure MultiplicarMat(m1, m2: tMatriz; var resul: tMatriz);
    {Efecto: resul:= m1 * m2}
    var
      i, j, k: 1..N;
    {i recorre las filas de m1 y de resul, j recorre las columnas
    de m1 y las filas de m2 y k recorre las columnas de m2 y las
    de resul}

```

```

begin
  for i:= 1 to N do
    for k:= 1 to N do begin
      resul[i,k]:= 0;
      for j:= 1 to N do
        resul[i,k]:= resul[i,k] + m1[i,j] * m2[j,k]
      end {for k}
    end; {MultiplicarMat}

  procedure EscribirMatProd(m: tMatriz);
    {Efecto: escribe en la pantalla los elementos de la matriz m}
    var
      i, j: 1..N;
    begin
      for i:= 1 to N do
        for j:= 1 to N do
          {Escribe mij}
          WriteLn('m(' , i, ', ' , j, ') = ' , m[i,j]);
        end; {EscribirMatProd}
      end;

begin
  WriteLn('Lectura de la matriz A');
  LeerMatriz(a);
  WriteLn('Lectura de la matriz B');
  LeerMatriz(b);
  MultiplicarMat(a,b,prod);
  EscribirMatProd(prod)
end. {MultiplicacionDeMatrices}

```

### Un ejemplo completo

El siguiente ejemplo utiliza todos los tipos de datos definidos por el programador vistos hasta ahora, es decir, los tipos *enumerado*, *subrango* y *array*.

Se trata de construir un almanaque del siglo XX, asignando en una matriz con tres índices correspondientes a los días del mes (1 al 31), meses (ene, feb, etc.), y años desde el 1901 al 2000, los días de la semana.

El punto de partida es el día 31 de diciembre de 1900 que fue lunes. A partir de esta fecha se van recorriendo consecutivamente todos los días del siglo asignándoles el día de la semana correspondiente, teniendo especial cuidado en determinar cuántos días tiene cada mes, para lo cual se ha de comprobar si el año es bisiesto. En un primer nivel de refinamiento podríamos escribir el siguiente seudocódigo:

*El día 31 de diciembre de 1900 fue lunes*  
*Para todos los años del siglo hacer*  
*Para todos los meses del año hacer*  
*Asignar el día de la semana a cada día del mes*

donde, la acción *Asignar el día de la semana a cada día del mes* puede ser refinada en un nivel inferior de la siguiente forma:

*Calcular cuántos días tiene el mes*  
*Para todos los días del mes hacer*  
*asignar al día actual el mañana de ayer*  
*asignar a ayer el día actual*

Los bucles correspondientes a los años y los meses no presentan dificultad, sin embargo, los días del mes dependen del propio mes, y en el caso de febrero, de que el año sea o no bisiesto. Por eso hay que calcular `CuantosDias` tiene el mes y para ello es necesario conocer el mes y el año.

Durante el cálculo de `CuantosDias` se hará una llamada a `EsBisiesto`, una función que determina el número de días de febrero.<sup>2</sup> Se usa también una función `Mañana` para el cálculo correcto del día siguiente, de forma que al domingo le siga el lunes.

El programa completo es el siguiente:

```
Program AlmanaqueSigloXX (input,output);
  {Calcula el día de la semana correspondiente a cada día del siglo XX}

  type
    tMeses = (ene, feb, mar, abr, may, jun, jul, ago, sep, oct,
              nov, dic);
    tDiasSemana = (lun, mat, mie, jue, vie, sab, dom);
    tAnnos = 1901..2000;
    tDiasMes = 1..31;
    tCalendarioSigloXX = array[tDiasMes, tMeses, tAnnos] of
                        tDiasSemana;

  var
    almanaque: tCalendarioSigloXX;
    mes: tMeses;
    ayer: tDiasSemana;
    contAnnos: tAnnos;
    dias, contaDias: tDiasMes;
```

---

<sup>2</sup>En el desarrollo de esta función se tendrá en cuenta que los años múltiplos de 100 no son bisiestos, salvo que sean múltiplos de 400.

```

function Mannana(hoy: tDiasSemana): tDiasSemana;
  {Dev. el día de la semana que sigue a hoy}
begin
  if hoy = dom then
    Mannana:= lun {se evita Succ(dom)  $\leadsto$  error}
  else
    Mannana:= Succ(hoy)
  end; {Mannana}

function EsBisiesto(anno: tAnnos): boolean;
  {Efecto: Averigua si un año es bisiesto o no}
begin
  EsBisiesto:= ((anno mod 4 = 0) and (anno mod 100 <> 0))
    or (anno mod 400 = 0)
end; {EsBisiesto}

function CuantosDias(unmes: tMeses, anno: tAnnos): tDiasMes;
  {Dev. el numero de días del mes unmes del año anno}
begin
  case unmes of
    abr,jun,sep,nov :
      CuantosDias:= 30;
    feb :
      if EsBisiesto(anno) then
        CuantosDias:= 29
      else
        CuantosDias:= 28;
    ene,mar,may,jul,ago,oct,dic :
      CuantosDias:= 31
  end {case}
end; {CuantosDias}

begin {AlmanaqueSigloXX}
  {Crea un almanaque del siglo XX}
  ayer:= lun; {El dia 31 de diciembre de 1900 fue lunes}
  for contAnnos:= 1901 to 2000 do
    for mes:= ene to dic do begin
      dias:= CuantosDias (mes, contAnnos);
      {número de días del mes actual}
      for contaDias:= 1 to dias do begin
        almanaque[contaDias, mes, contAnnos]:= Mannana(ayer);
        ayer:= almanaque[contaDias,mes,contAnnos]
      end {for contaDias}
    end {for mes}
  end {for contAnnos}
end. {AlmanaqueSigloXX}

```

Para saber qué día de la semana fue el 12 de enero de 1925, se mirará la posición `almanaque[12,ene,1925]` (cuyo valor es `1un`).

## 12.4 Ejercicios

1. Complete adecuadamente los interrogantes de la definición que sigue,

```

const
  n = ?; {un entero positivo}
var
  i, j: 1..n;
  a : array[1..n, 1..n] of ?

```

e indique cuál es el efecto de la siguiente instrucción:

```

for i:= 1 to n do
  for j:= 1 to n do
    a[i, j]:= i=j

```

2. Considérese que los  $N$  primeros términos de una sucesión están registrados en un array de  $N$  componentes reales. Defina el subprograma `Sumar`, que transforma los elementos del array en los de la correspondiente serie:

$$a'_n \rightarrow \sum_{i=1}^n a_i$$

3. Escriba un programa que realice la suma de dos números positivos muy grandes (de 50 cifras, por ejemplo).
4. Para vectores de  $\mathbb{R}^3$ , defina subprogramas para calcular
  - (a) el módulo:  $\mathbb{R}^3 \rightarrow \mathbb{R}$
  - (b) un vector unitario con su misma dirección
  - (c) la suma:  $\mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3$
  - (d) el producto escalar:  $\mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}$
  - (e) el producto vectorial:  $\mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3$
  - (f) el producto mixto:  $\mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}$

Usando en lo posible los subprogramas anteriores, defina otros para averiguar lo siguiente:

- (a) dados dos vectores de  $\mathbb{R}^3$ , ver si uno de ellos es combinación lineal del otro.
- (b) dados tres vectores de  $\mathbb{R}^3$ , ver si uno de ellos es combinación lineal de los otros dos.

5. Escriba un subprograma que mezcle dos vectores (de longitudes  $m$  y  $n$  respectivamente), ordenados ascendentemente, produciendo un vector (de longitud  $m + n$ ), también ordenado ascendentemente.
6. Escriba un subprograma que averigüe si dos vectores de  $N$  enteros son iguales. (La comparación deberá detenerse en cuanto se detecte alguna diferencia.)
7. Dados dos vectores de caracteres del mismo tipo, escriba un subprograma que averigüe si el primero de ellos precede al segundo en orden alfabético.
8. Escriba un subprograma que desplace todas las componentes de un vector de  $N$  enteros un lugar a la derecha, teniendo en cuenta que la última componente se ha de desplazar al primer lugar. Generalícese el subprograma anterior para desplazar las componentes  $k$  lugares.
9. Escriba un subprograma que lea una secuencia de caracteres del teclado registrándola en un vector de caracteres. Cuando el número de caracteres escritos sea inferior a la longitud del vector, se deberá rellenar con espacios en blanco por la derecha; y cuando haya más caracteres de los que caben, se eliminarán los últimos.
10. Si representamos la ecuación de una recta en el plano como un vector de tres componentes,  $Ax + By + C = 0$ , escriba subprogramas para determinar:
  - (a) La ecuación de la recta que pasa por dos puntos dados.
  - (b) La ecuación de la recta que pasa por un punto dado y tiene una cierta pendiente.
  - (c) El ángulo que forman dos rectas dadas.
  - (d) La distancia de un punto dado a una recta dada.
11. Defina una función que averigüe el máximo elemento de una matriz de  $M \times N$  enteros.
12. Se tiene un sistema de ecuaciones lineales representado mediante una matriz de  $3 \times 4$ , donde las tres primeras columnas contienen los coeficientes del sistema (con determinante distinto de cero) y la cuarta los términos independientes. Escriba un subprograma para calcular la solución del sistema por la regla de Cramer.
13. Se tiene ahora un sistema de  $N$  ecuaciones con  $N$  incógnitas, supuestamente compatible determinado. Escriba un subprograma para calcular la solución del sistema por el método de Gauss.
14. Defina el tipo de una matriz cuadrada de dimensión  $N$  de elementos reales, y escriba un subprograma **Trasponer** que intercambie los elementos de posiciones  $(i, j)$  y  $(j, i)$  entre sí,  $\forall i, j \in \{1, \dots, N\}$ .
15. Defina un procedimiento para descomponer una matriz cuadrada  $M$  en otras dos,  $A$  y  $B$ , con sus mismas dimensiones, de manera que

$$M = S + A$$

y tales que  $S$  es simétrica y  $A$  antisimétrica. Ello se consigue forzando que

$$S_{i,j} = S_{j,i} = \frac{M_{i,j} + M_{j,i}}{2}, \forall i, j \in \{1, \dots, N\}$$

y

$$A_{i,j} = -A_{j,i} = \frac{M_{i,j} - M_{j,i}}{2}, \forall i, j \in \{1, \dots, N\}$$

16. Defina subprogramas para determinar si una matriz es simétrica y si es triangular inferior, parando cuando se detecte que no lo es.
17. (a) Escriba un programa que lea los caracteres del `input` y efectúe una estadística de las letras que aparecen, contando la frecuencia de cada una de ellas sin tener en cuenta si es mayúscula o minúscula.
  - (b) Modifique el programa del apartado (a) de manera que calcule cuántas veces aparece cada par de letras contiguas en el texto.
  - (c) Modifique el programa del apartado anterior para que se muestre también cuántas veces aparece cada trío de letras contiguas en el texto.
18. Dada una matriz real de  $3 \times 3$ , escriba un programa que calcule:
  - (a) Su determinante.
  - (b) Su matriz adjunta.
  - (c) Su matriz inversa.
19. (a) Escriba un subprograma para hallar el producto de dos matrices de  $m \times n$  y de  $n \times l$ .
  - (b) Escriba un subprograma que calcule la potencia de una matriz cuadrada de orden  $n$ . (Obsérvese que este problema admite un algoritmo iterativo y otro recursivo, como ocurre con la potencia de números.)
20. Un modo de averiguar el máximo elemento de un vector  $V$  de tamaño  $n$  es el siguiente: si el vector consta de un solo elemento, éste es el máximo; de lo contrario, se consideran los fragmentos  $V_{1, \dots, pm}$  y  $V_{pm+1, \dots, n}$  y se averigua el máximo en cada una de sus “mitades” (mediante este mismo procedimiento), resultando que el mayor de tales números es el de  $V$ .  
Desarrolle un subprograma que halle el máximo elemento de un vector en un fragmento dado mediante el procedimiento descrito.
21. Se tiene una frase en un array de caracteres. Desarrolle subprogramas para averiguar los siguientes resultados:
  - (a) El número de palabras que contiene.
  - (b) La longitud de la palabra más larga.
  - (c) De todas las palabras, la que aparece antes en el diccionario.

(Se entiende por palabra la secuencia de letras seguidas, delimitadas por un carácter que no es una letra o por los límites del array.)

## Capítulo 13

# Registros

---

<b>13.1 Descripción del tipo de datos registro . . . . .</b>	<b>271</b>
<b>13.2 Arrays de registros y registros de arrays . . . . .</b>	<b>279</b>
<b>13.3 Ejercicios . . . . .</b>	<b>282</b>

---

En el capítulo anterior se ha estudiado cómo se trabaja en Pascal con colecciones de datos del mismo tipo. Pero en los problemas relacionados con la vida real es necesario agrupar con frecuencia datos de distintos tipos: por ejemplo, el documento nacional de identidad contiene, entre otros, un entero (el número) y cadenas de caracteres (el nombre, los apellidos, etc.); una ficha de artículo en un almacén debe contener datos numéricos (código, número de unidades en “stock”, precio, etc.), booleanos (para indicar si se le aplica o no descuento) y cadenas de caracteres (la descripción del artículo, el nombre del proveedor, etc.). En este capítulo presentamos un nuevo tipo de datos, los registros, que nos van a permitir almacenar y procesar este tipo de información.

### 13.1 Descripción del tipo de datos registro

Los registros<sup>1</sup> son otro tipo de datos estructurados muy utilizados en Pascal. Su principal utilidad reside en que pueden almacenar datos de distintos tipos, a diferencia de los demás datos estructurados. Un registro estará formado por

---

<sup>1</sup>En inglés *record*.

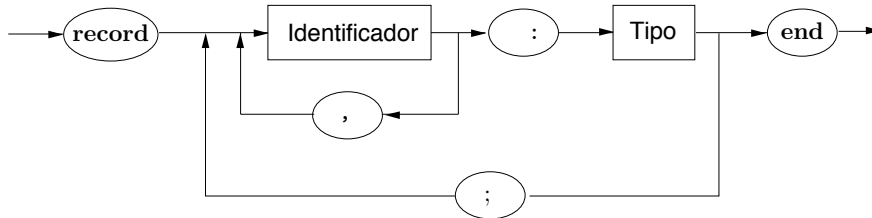


Figura 13.1.

varios datos (simples o estructurados) a los que llamaremos campos del registro y que tendrán asociado un identificador al que llamaremos nombre de campo.

La definición de un registro se hace según el diagrama sintáctico de la figura 13.1 y, por tanto, la definición de un registro genérico en Pascal es:

```

type
  tNombReg = record
    idenCampo1: idTipo1;
    idenCampo2: idTipo2;
    ...
    idenCampoN: idTipoN
  end; {tNombReg}

```

Por ejemplo, supongamos que un encargado de obra quiere tener registrados varios datos de sus trabajadores, tales como: nombre, dirección, edad y número de D.N.I. Con los tipos de datos que conocemos resultaría bastante difícil, ya que tendríamos que indicar que las variables `direccion`, `edad` y `dni` están relacionadas con el nombre de un trabajador en concreto. Para solucionarlo, se utiliza el tipo de datos estructurado *registro*, de la siguiente forma:

```

type
  tEdades = 16..65;
  tDigitos = '0'..'9';
  tFicha = record
    nombre: array[1..30] of char;
    direccion: array[1..50] of char;
    edad: tEdades;
    dni: array[1..8] of tDigitos
  end; {tFicha}

```

Como se puede observar, en este ejemplo se utilizan datos estructurados en la definición de otro dato estructurado (dentro de la estructura del dato *registro*

se utilizará un vector de caracteres para almacenar el nombre y la dirección de un empleado).

Es conveniente destacar que el tipo de datos registro, al igual que el tipo array, es un tipo estructurado de tamaño fijo; sin embargo se diferencian de ellos principalmente en que los componentes de un array son todos del mismo tipo, mientras que los componentes de un registro pueden ser de tipos distintos.

### 13.1.1 Manejo de registros: acceso a componentes y operaciones

El dominio de un registro estará formado por el producto cartesiano de los dominios de sus campos componentes.

Para poder trabajar con el tipo de datos registro es necesario saber cómo acceder a sus campos, cómo asignarles valores y que tipo de operaciones podemos realizar con ellos:

- Para acceder a los campos de los registros se utilizan construcciones de la forma `nomVarRegistro.nomCampo`, es decir, el nombre de una variable de tipo registro seguido de un punto y el nombre del campo al que se quiere acceder. Por ejemplo, si la variable `f` es de tipo `tFicha`, para acceder a sus campos `nombre`, `direccion`, `edad` y `dni` se utilizarán, respectivamente, las construcciones:

```
f.nombre  
f.direccion  
f.edad  
f.dni
```

En este punto se debe señalar que, a diferencia de los arrays, en los cuales el acceso se realiza por medio de índices (tantos como dimensiones tenga el array, que pueden ser el resultado de una expresión y por tanto calculables), en los registros se accede por medio de los identificadores de sus campos, que deben darse explícitamente.

- Los tipos de los campos pueden ser tipos predefinidos o definidos por el programador mediante una definición de tipo previa. Incluso un campo de un registro puede ser de tipo registro. Así, por ejemplo, si queremos almacenar para cada alumno, su nombre, fecha de nacimiento y nota, podríamos definir tipos y variables de la siguiente forma:

```

type
  tMeses = (ene, feb, mar, abr, may, jun, jul, ago, sep, oct, nov, dic);
  tCalificaciones = (NP, Sus, Apr, Notab, Sob, MH);
  tNombre = array[1..50] of char;
  tFecha = record
    dia: 1..31;
    mes: tMeses;
    anno: 1900..2000
  end; {tFecha}
  tFicha = record
    nombre: tNombre;
    fechaNac: tFecha;
    nota: tCalificaciones
  end; {tFicha}
var
  alumno: tFicha;

```

- La asignación de valores a los campos se hará dependiendo del tipo de cada uno de ellos. Así, en el ejemplo anterior, para iniciar los datos de la variable `Alumno` tendríamos que utilizar las siguientes asignaciones:

```

alumno.nombre:= 'Mario_Aguilera_';
alumno.fechaNac.dia:= 3;
alumno.fechaNac.mes:= feb;
alumno.fechaNac.anno:= 1973;
alumno.nota:= Notab

```

- Las operaciones de lectura y escritura de registros han de hacerse campo por campo, empleando procedimientos o funciones especiales si el tipo del campo así lo requiere. Por ejemplo:

```

procedure EscribirFicha(unAlumno: tFicha);
  {Efecto: Escribe en la pantalla el contenido del registro
  unAlumno}
begin
  WriteLn('Nombre: ', unAlumno.nombre);
  Write('Fecha de nacimiento: ', unAlumno.fechaNac.dia);
  EscribirMes(unAlumno.fechaNac.mes);
  WriteLn(unAlumno.fechaNac.anno);
  Write('Nota: ');
  EscribirNota(unAlumno.nota)
end; {EscribirFicha}

```

Obsérvese que los campos `fecha.mes` y `nota` son de tipo enumerado y necesitarán dos procedimientos especiales (`EscribirMes` y `EscribirNota`, respectivamente) para poder escribir sus valores por pantalla.

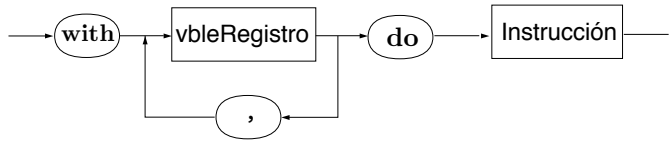


Figura 13.2.

Al igual que todos los tipos de datos compuestos, un registro no puede ser el resultado de una función. Para solucionar este problema actuaremos como de costumbre, transformando la función en un procedimiento con un parámetro por variable adicional de tipo registro que albergue el resultado de la función. Así, por ejemplo:

```

procedure LeerFicha(var unAlumno: tFicha);
  {Efecto: lee del input el contenido del registro unAlumno}
begin
  Write('Nombre del alumno:');
  LeerNombre(unAlumno.nombre);
  Write('Día de nacimiento: ');
  ReadLn(unAlumno.fechaNac.dia);
  Write('Mes de nacimiento: ');
  LeerMes(unAlumno.fechaNac.mes);
  Write('Año de nacimiento: ');
  ReadLn(unAlumno.fechaNac.anno);
  Write('Calificación: ');
  LeerNota(unAlumno.nota)
end; {LeerFicha}

```

- Como puede observarse, es incómodo estar constantemente repitiendo el identificador `unAlumno`. Para evitar esta repetición Pascal dispone de la instrucción **with** que se emplea siguiendo la estructura descrita en el diagrama sintáctico de la figura 13.2.

Por ejemplo, para el procedimiento `LeerFicha` se podrían utilizar dos instrucciones **with** anidadas de la siguiente forma:

```

procedure LeerFicha(var unAlumno: tFicha);
  {Efecto: lee del input el contenido del registro unAlumno}
begin
  with unAlumno do begin
    WriteLn('Introduce el nombre del alumno:');
    LeerNombre(nombre);
    with fechaNac do begin

```

```

    Write('Día de nacimiento: ');
    ReadLn(dia);
    Write('Mes de nacimiento: ');
    LeerMes(mes);
    Write('Año de nacimiento: ');
    ReadLn(anno)
end; {with fechaNac}
Write('Calificación: ');
LeerNota(nota)
end {with unAlumno}
end; {LeerFicha}

```

- Al igual que los arrays, si dos variables `r1` y `r2` son del mismo tipo registro, se pueden realizar asignaciones de registros completos (con la instrucción `r1 := r2`) evitando así tener que ir copiando uno a uno todos los campos del registro.

### 13.1.2 Registros con variantes

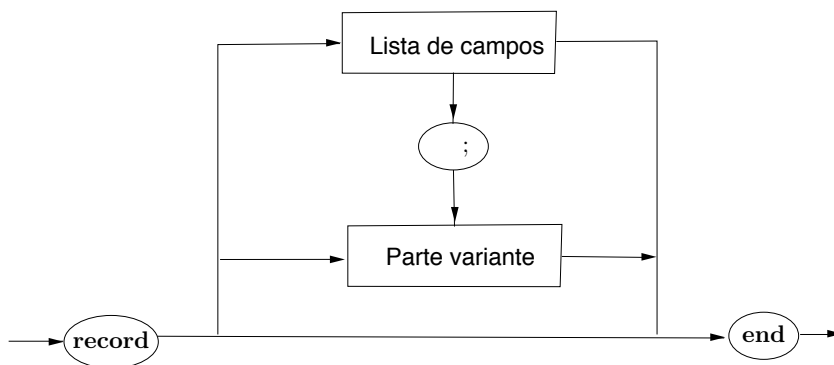
En ciertos casos es conveniente poder variar el tipo y nombre de algunos de los campos existentes en un registro en función del contenido de uno de ellos. Supongamos, por ejemplo, que en el registro `tFicha` definido anteriormente queremos incluir información adicional dependiendo de la nacionalidad. Si es española, añadiremos un campo con el D.N.I., y si no lo es, añadiremos un campo para el país de origen y otro para el número del pasaporte.

Con este objetivo se pueden definir en Pascal los registros con variantes, que constan de dos partes: la primera, llamada *parte fija*, está formada por aquellos campos del registro que forman parte de todos los ejemplares; la segunda parte, llamada *parte variable*, está formada por aquellos campos que sólo forman parte de algunos ejemplares.

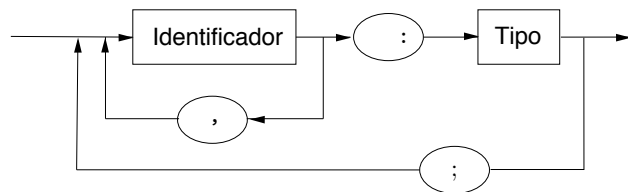
En la parte fija, debe existir un campo selector mediante el cual se determina la parte variable que se utilizará. Este campo selector debe ser único, es decir, sólo se permite un campo selector.

El diagrama sintáctico de la definición de un registro con variantes es el de la figura 13.3.

Para resolver el problema del ejemplo anterior se puede emplear un registro con variantes de la siguiente forma:



Lista de campos:



Parte variante:

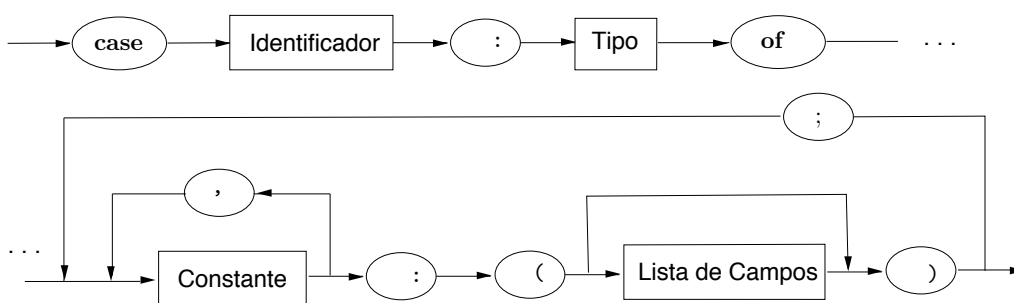


Figura 13.3.

```

type...
  tNombre = array[1..50] of char;
  tDNI = array[1..8] of '0'..'9';
  tPais = array[1..20] of char;
  tPasaporte = array[1..15] of '0'..'9';
  tFicha = record
    nombre: tNombre;
    fechaNac: tFecha;
    nota: tCalificaciones;
    case espannol : boolean of
      True :
        (dni : tDNI;
         False :
           (pais : tPais;
            pasaporte : tPasaporte
            end; {tFicha}

```

Con la definición anterior, el procedimiento LeerFicha queda como sigue:

```

procedure LeerFicha(var unAlumno: tFicha);
  {Efecto: lee del input el contenido del registro unAlumno}
  var
    c: char;
begin
  with unAlumno do begin
    Write('Introduce el nombre del alumno:');
    ReadLn(nombre);
    with fechaNac do begin
      Write('Día de nacimiento: ');
      ReadLn(dia);
      Write('Mes de nacimiento: ');
      LeerMes(mes);
      Write('Año de nacimiento: ');
      ReadLn(anno)
    end; {with fechaNac}
    Write('Calificacion: ');
    LeerNota(nota);
    repeat
      Write('¿Es ', nombre, ' español? (S/N)');
      ReadLn(c)
    until c in ['s', 'S', 'n', 'N'];
    case c of
      's', 'S' :
        begin {el alumno es español}
          espannol:= True;
          Write('DNI: ');

```

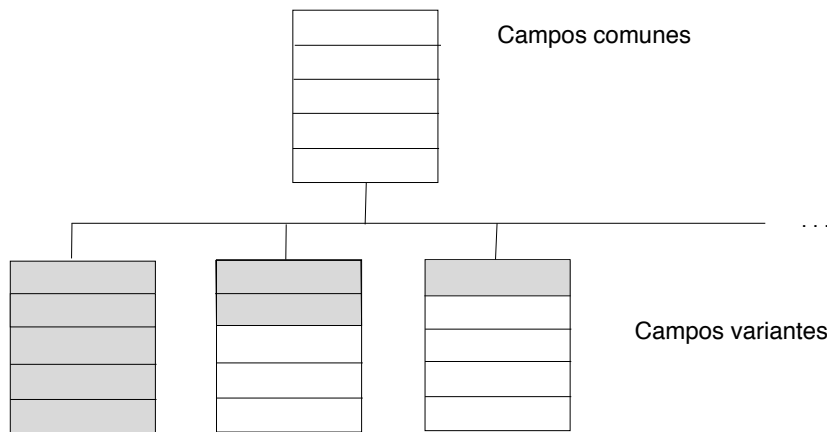


Figura 13.4.

```

    LeerDNI(dni)
  end;
'n', 'N' :
  begin {el alumno es extranjero}
    espanol:= False;
    Write('país: ');
    LeerPais(pais);
    Write('pasaporte: ');
    LeerPasaporte(pasaporte)
  end
end {case}
end {with unAlumno}
end; {LeerFicha}

```

La utilización de registros con campos variantes relaja en parte la condición de tipos fuertes de Pascal al permitir que una variable de esta clase almacene valores de diferentes tipos. Sin embargo, debe tenerse cuidado con el uso de la parte variante, ya que los compiladores no suelen comprobar que esa parte se utiliza correctamente.

Al implementar los registros con variantes el compilador hace reserva de memoria para la variante más grande, aunque no se aproveche en el caso de las variantes más pequeñas, como se muestra en la figura 13.4.

## 13.2 Arrays de registros y registros de arrays

Dado que los registros permiten almacenar datos de diferentes tipos correspondientes a una persona u objeto y que los arrays agrupan datos de un mismo

tipo, es frecuente combinarlos formando arrays de registros que permitan almacenar y gestionar la información relativa a un grupo de personas u objetos.

Por ejemplo, una vez definido el tipo `tFicha` del apartado 13.1.1 donde almacenamos los datos de un alumno, podemos definir el tipo siguiente:

```
type
  tVectorFichas = array[1..40] of tFicha;
```

y declarar las variables:

```
var
  alumno: tFicha;
  clase: tVectorFichas;
```

De esta forma, en el vector `clase` podemos almacenar los datos de los alumnos de una clase. Para acceder al año de nacimiento del alumno número 3, tendríamos que utilizar la siguiente expresión:

```
clase[3].fecha.anno
```

mientras que para acceder a su nota usaríamos:

```
clase[3].nota
```

Los arrays tienen un tamaño fijo, sin embargo, hay casos en los que el número de datos no se conoce *a priori*, pero para los que puede suponerse un máximo.

En este caso, se puede definir un registro que contenga a un vector del tamaño máximo y una variable adicional para llevar la cuenta de la parte utilizada.

Por otra parte, hay casos en los que puede resultar útil definir registros de arrays. Supongamos, por ejemplo, que queremos calcular el valor medio de una variable estadística real de una muestra cuyo tamaño no supera los cien individuos. Para ello se podrían hacer las siguientes definiciones y declaraciones:

```
const
  MaxCompo = 100;
type
  tVector = array [1..MaxCompo] of real;
  tRegistro = record
    vector: tVector;
    ocupado: 0..MaxCompo
  endtRegistro;
var
  indice: 1..MaxCompo;
  reg: tRegistro;
  valor, suma, media: real;
  fin: boolean;
```

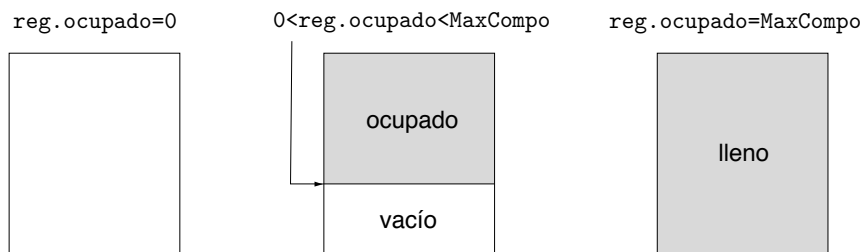


Figura 13.5.

Para introducir los datos hay que comprobar si el vector está lleno y, en caso contrario, se pide el dato y se incrementa el valor de `reg.ocupado`, que almacena el índice del último elemento introducido. Si `reg.ocupado` vale cero, el vector está vacío, si  $0 < \text{reg.ocupado} < \text{MaxCompo}$  tiene `reg.ocupado` datos y quedan  $\text{MaxCompo} - \text{reg.ocupado}$  espacios libres, y por último, cuando `reg.ocupado = MaxCompo`, el vector está lleno. Todas estas situaciones se recogen en la figura 13.5.

Para introducir los valores en el vector podríamos escribir un fragmento de programa como el siguiente:

```
reg.ocupado:= 0;
fin:= False;
while (reg.ocupado < MaxCompo) and not(fin) do begin
  Write('Introduzca el valor del individuo ', reg.ocupado + 1)
  Write(' o un valor negativo para terminar ');
  ReadLn(valor);
  if valor >= 0 then begin
    reg.ocupado:= reg.ocupado + 1;
    reg.vector[reg.ocupado]:= valor
  end
  else
    fin:= True
end {while}
```

Para calcular la media se recorre la parte ocupada del vector acumulando sus valores, como se muestra a continuación:

```
for indice:= 1 to reg.ocupado do
  suma:= suma + reg.vector[indice];
media:= suma/reg.ocupado
```

### 13.3 Ejercicios

1. Defina un tipo de datos para manejar fechas, incluyendo la información usual para un día cualquiera del calendario: el número de día dentro del mes, el día de la semana, el mes y el año, y con él, los siguientes subprogramas:
  - (a) Lectura y escritura.
  - (b) Avance (que pasa de un día al siguiente), con ayuda de una función que indica el número de días de un mes de un cierto año.
  - (c) Distancia entre fechas, usando la función avance.
2. Considérese una representación de los números reales mediante su signo, positivo o negativo, su parte entera, formada por  $N$  dígitos (por ejemplo, 25) de cero a nueve y por su parte decimal, formada por  $NDEC$  cifras (por ejemplo, 5).
  - (a) Defina en Pascal este tipo de datos.
  - (b) Defina procedimientos apropiados para su lectura y escritura.
  - (c) Defina un subprograma para sumar reales, controlando el posible desbordamiento.
  - (d) Defina un subprograma para comparar reales.
3.
  - (a) Defina un tipo de datos registro que permita almacenar un punto del plano real y un tipo vector formado por tres registros del tipo anterior.<sup>2</sup>
  - (b) Escriba un subprograma que determine si los tres puntos almacenados en una variable del tipo vector forman un triángulo.
  - (c) Escriba un subprograma tal que, si tres puntos forman un triángulo, calcule su área aplicando la fórmula de Herón (véase el ejercicio 6 del capítulo 4).
4.
  - (a) Defina un tipo registro que permita almacenar un ángulo dado en forma de grados (sexagesimales), minutos y segundos.
  - (b) Escriba dos subprogramas, el primero para convertir un ángulo dado en radianes en una variable del tipo registro anterior, y el segundo para realizar la conversión inversa. En el primero se tendrá en cuenta que el resultado debe tener el número de grados inferior a  $360^\circ$ , el de minutos inferior a  $60'$  y el de segundos inferior a  $60''$ .
5. La posición de un objeto lanzado con velocidad inicial  $v$  y con ángulo  $\alpha$  con respecto a la horizontal, transcurrido un cierto tiempo  $t$ , puede expresarse (despreciando el rozamiento) con las siguientes ecuaciones:

$$x = vt \cos \alpha \qquad y = vt \sin \alpha - \frac{1}{2}gt^2$$

donde  $g$  es la aceleración de la gravedad ( $9.8m/seg^2$ ).

- (a) Defina un registro que almacene la velocidad inicial y el ángulo  $\alpha$ .

---

<sup>2</sup>Obsérvese que ésta es una definición alternativa a la de arrays, tal vez más apropiada por ser las componentes del mismo tipo.

- (b) Defina un registro que almacene las coordenadas  $x$ ,  $y$  y el tiempo  $t$  transcurrido desde el lanzamiento.
  - (c) Escriba un subprograma que utilice un registro de cada uno de los tipos anteriores y calcule la posición del objeto a partir de  $v$ ,  $\alpha$  y  $t$ .
  - (d) Escriba un subprograma que calcule la altura máxima aproximada alcanzada por el objeto utilizando intervalos de tiempo “pequeños”, por ejemplo décimas o centésimas de segundo.
  - (e) Escriba un subprograma que calcule la distancia máxima aproximada alcanzada por el objeto, con la misma técnica del apartado anterior. Indicación: la distancia máxima se alcanza cuando la altura se hace cero.
  - (f) Redefina los registros de los apartados anteriores para utilizar ángulos en forma de grados (sexagesimales), minutos y segundos, realizando las conversiones con los subprogramas del apartado 4b.
6. Un comercio de alimentación almacena los siguientes datos de sus productos: producto (nombre del producto), marca (nombre del fabricante), tamaño (un número que indica el peso, volumen, etc. de un determinado envase del producto), precio (del tamaño correspondiente) y unidades (cantidad existente en inventario).
- (a) Defina un tipo de datos registro que permita almacenar dichos campos.
  - (b) Defina un vector suficientemente grande de dichos registros.
  - (c) Defina un registro que incluya el vector y un campo adicional para indicar la parte ocupada del vector de acuerdo con la técnica expuesta en el apartado 13.2.
  - (d) Escriba los subprogramas necesarios para realizar las siguientes operaciones:
    - **Altas:** Consiste en introducir nuevos productos con todos sus datos. Normalmente se efectuarán varias altas consecutivas, deteniendo el proceso cuando se introduzca un producto cuyo nombre comience por una clave especial (un asterisco, por ejemplo).
    - **Bajas:** Se introduce el nombre del producto a eliminar, se muestran sus datos y se pide al usuario confirmación de la baja. Para eliminar el producto se desplazan desde el producto inmediato siguiente una posición hacia delante los sucesivos productos, y se reduce en una unidad la variable que indica la posición ocupada. Indicación: se deben tratar de forma especial los casos en que no hay productos, cuando sólo hay uno y cuando se elimina el último.
    - **Modificaciones:** Se introduce el nombre del producto, se muestra y se piden todos sus datos.
    - **Consultas:** Pueden ser de dos tipos: por producto, pidiendo el nombre y mostrando todos sus datos, y total, mostrando el listado de todos los productos con sus datos. Se puede añadir el coste total por producto y el coste total del inventario. Se puede mostrar por pantalla y por impresora.

- (e) Escriba un programa completo comandado por un menú con las operaciones del apartado anterior.
7. Construya un tipo registro para manejar una hora del día, dada en la forma de horas (entre 0 y 23), minutos y segundos (entre 0 y 59). Utilizando este tipo, escriba subprogramas para:
- Leer correctamente un instante dado.
  - Mostrar correctamente una hora del día.
  - Dado un tiempo del día, pasarlo a segundos.
  - Dados dos tiempos del día, calcular su diferencia, en horas, minutos y segundos.
  - Dado un tiempo del día, calcular el instante correspondiente a un segundo después.
  - Dado un tiempo del día, mostrar un reloj digital en la pantalla durante un número de segundos predeterminado. (El retardo se puede ajustar con bucles vacíos.)
8. La posición de un punto sobre la superficie de la tierra se expresa en función de su longitud y latitud. La primera mide el ángulo que forma el meridiano que pasa por el punto con el meridiano que pasa por el observatorio de Greenwich, y toma valores angulares comprendidos entre 0 y 180°, considerándose longitud Este (E) cuando el ángulo se mide hacia el Este y longitud Oeste (W) en caso contrario. La segunda mide el ángulo que forma la línea que pasa por el punto y por el centro de la tierra con el plano que contiene al ecuador, y toma valores angulares comprendidos entre 0 y 90°, considerándose latitud Norte (N) cuando el punto está situado al Norte del ecuador y latitud Sur (S) en caso contrario. En ambos casos los valores angulares se miden en grados, minutos y segundos. Defina un tipo registro que permita almacenar los datos anteriores. Escriba subprogramas para leer y escribir correctamente variables del tipo anterior, en grados, minutos y segundos.
9. Escriba dos tipos registro, uno para almacenar las coordenadas cartesianas de un punto del plano y otro para almacenarlo en coordenadas polares con el ángulo en radianes. Escriba subprogramas para pasar de unas coordenadas a otras. Escriba un subprograma que calcule la distancia entre dos puntos en coordenadas cartesianas, polares y ambas.
10. Dada una lista de puntos del plano, en un vector no completo (véase el apartado 13.2) que se supone que definen los vértices de un polígono, determine el área del mismo mediante la fórmula siguiente:

$$A = \frac{1}{2}((X_2Y_1 - X_1Y_2) + (X_3Y_2 - X_2Y_3) + (X_4Y_3 - X_3Y_4) + \dots + (X_1Y_N - X_NY_1))$$

## Capítulo 14

# Archivos

---

14.1 Descripción del tipo de datos archivo . . . . .	285
14.2 Manejo de archivos en Pascal . . . . .	286
14.3 Archivos de texto . . . . .	294
14.4 Ejercicios . . . . .	298

---

Como se dijo en [PAO94] (véanse los apartados 4.2.2 y 6.1.1) los archivos se pueden entender como un conjunto de datos que se almacenan en un dispositivo de almacenamiento, por ejemplo, una unidad de disco.

Para expresar este concepto, Pascal dispone del tipo de datos estructurado **file** (*archivo* en inglés), que se explica en los apartados siguientes.

### 14.1 Descripción del tipo de datos archivo

Un archivo en Pascal se estructura como una *secuencia* homogénea de datos, de tamaño no fijado de antemano, la cual se puede representar como una fila de celdas en las que se almacenan los datos componentes del archivo. Una marca especial llamada *fin de archivo* señala el fin de la secuencia. La estructura del archivo se muestra gráficamente en la figura 14.1.

De hecho, hemos trabajado con archivos en Pascal desde el principio de este texto, ya que, como sabemos, los programas que producen alguna salida por pantalla necesitan el archivo estándar **output**, que se incluye en el encabezamiento



Figura 14.1.

de la siguiente forma:<sup>1</sup>

```
Program NombrePrograma (output);
```

Si el programa requiere, además, que se introduzca algún valor por teclado, necesitaremos también el archivo estándar `input` que debe entonces declararse en el encabezamiento del programa:

```
Program NombrePrograma (input, output);
```

En este apartado se detalla cómo utilizar otros archivos que sirvan para la lectura y escritura de datos. Estos nuevos tipos de archivos de entrada y salida se asociarán a archivos almacenados en unidades de disco. Pascal permite acceder a ellos para guardar datos que posteriormente podrán ser leídos por el mismo o por otro programa.

Imaginemos que se ejecuta el programa `AlmanaqueSigloXX` que aparece como ejemplo en el apartado 12.3. Una vez que hemos ejecutado dicho programa sería de gran utilidad almacenar los resultados obtenidos en un archivo y, así, poder utilizarlos posteriormente sin tener que generar nuevamente el almanaque, con el consiguiente ahorro de tiempo. En los siguientes apartados se presentan los archivos de Pascal en general, y se detalla el caso particular de los archivos de texto, por su frecuente utilización.

Los archivos tienen como limitación el que sus elementos no pueden ser archivos. Por lo tanto, no es posible la declaración

```
tArchivo = file of file of ...;
```

## 14.2 Manejo de archivos en Pascal

Un archivo es un tipo de datos estructurado que permitirá almacenar en una unidad de disco información homogénea, es decir, datos de un mismo tipo, ya sea

<sup>1</sup>Es conveniente incluir siempre en el encabezamiento del programa el archivo de salida `output` aunque no se vaya a realizar ninguna salida por pantalla, ya que esto permitirá al programa escribir en pantalla los mensajes de error que pudieran originarse.

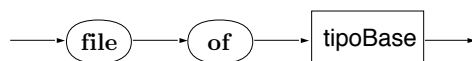


Figura 14.2.

básico o estructurado, por lo que las componentes del archivo van a ser valores de este tipo.

Como es natural, antes de trabajar con el tipo de datos archivo es necesario conocer su definición. El diagrama sintáctico de la definición de archivos es el de la figura 14.2.

Por ejemplo, si queremos trabajar con un archivo cuyos elementos sean arrays de caracteres, utilizaremos la siguiente definición:

```

type
  tTarjeta = array[1..50] of char;
  tArchivo = file of tTarjeta;
var
  unaTarjeta: tTarjeta;
  archivoTarjetas: tArchivo;

```

Si en un programa Pascal se va a utilizar un archivo externo, es necesario incluir el identificador del archivo (por ejemplo `nombreArchivo`) en el encabezamiento. Por tanto, un programa que maneje el archivo `nombreArchivo`, debe ser declarado como sigue:

```

Program TratamientoDeArchivo (input, output, nombreArchivo);

```

Con esta declaración el programa ya reconocerá al archivo con el que vamos a trabajar y estará preparado para poder realizar operaciones de acceso a dicho archivo.<sup>2</sup> Así, en el ejemplo anterior debemos realizar la siguiente declaración de programa:

```

Program TratamientoDeTarjetas (input, output, archivoTarjetas);

```

---

<sup>2</sup>Además, es necesario asociar el archivo “lógico” (declarado en el programa) con un archivo “físico” en el disco (véase el apartado B.9). Para realizar esta operación en Turbo Pascal se utiliza la instrucción `Assign (nombreArchivoLogico, NombreArchivoFisico)`. Por ejemplo, la instrucción

```
Assign (archivoTarjetas, 'C:\TARJETAS\DATOS.TXT')
```

indica que los datos de `archivoTarjetas` se almacenarán en el archivo de disco `DATOS.TXT` dentro del subdirectorio `TARJETAS` de la unidad `C:`.

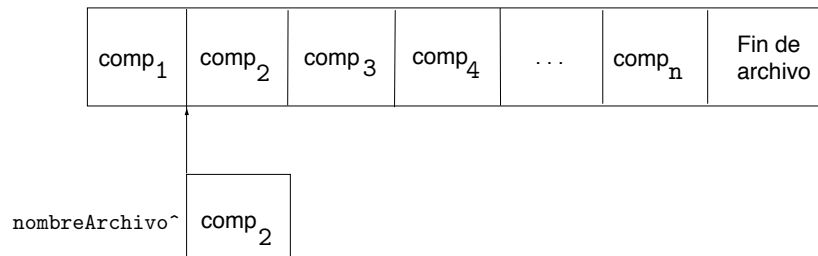


Figura 14.3.

Para acceder a las componentes de un archivo se utiliza el llamado *cursor* del archivo. Este cursor se puede entender como una “ventana” por la cual vemos una componente del archivo (aquella a la que apunta), como se muestra en la figura 14.3.

La notación utilizada en Pascal para referenciar el cursor de un archivo es:<sup>3</sup>

`nombreArchivo^`

Como se citó al comienzo de este capítulo, todo archivo tiene asociada una marca de fin de archivo. En Pascal se dispone de una función booleana llamada `EoF`.<sup>4</sup> Para utilizarla tendremos que indicarle el nombre del archivo. Así, `EoF(nombreArchivo)` devuelve el valor `False` si no se ha alcanzado el final del archivo o `True` en caso contrario (en cuyo caso el contenido del cursor `nombreArchivo^` es indeterminado).

Hemos de destacar en este momento que, como se puede comprobar, el manejo de archivos en Pascal no es muy eficiente debido a que sólo se dispone de archivos secuenciales.

### 14.2.1 Operaciones con archivos

Las operaciones más importantes que se pueden realizar con los archivos son la escritura y lectura de sus componentes. Estas operaciones se van a llevar a cabo de forma muy similar a la lectura y escritura usual (con las instrucciones `Read` y `Write`), salvo que se redireccionarán al archivo adecuado.

Un archivo se crea o se amplía escribiendo en él. Cada vez que se realice una operación de escritura, se añadirá una nueva componente al final del archivo

<sup>3</sup>En Turbo Pascal no se puede utilizar directamente el cursor `nombreArchivo^` ni las operaciones `Put` y `Get` que se presentan más adelante; posiblemente ésta es la razón por la que esta notación ha caído en desuso. (Véase el apartado B.9 para ver las operaciones equivalentes.)

<sup>4</sup>Del inglés *End Of File* (fin de archivo).

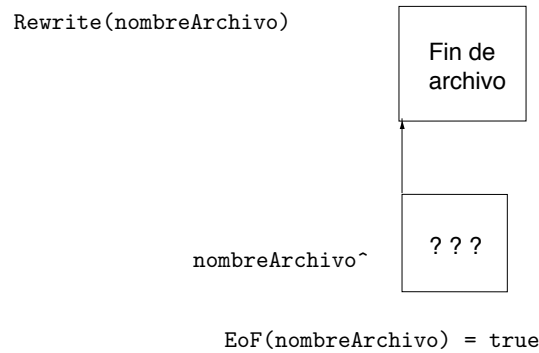


Figura 14.4.

secuencial. El cursor del archivo avanzará una posición cada vez que se escriba o se lea en el archivo.

La creación de un archivo se hace mediante la siguiente instrucción:

```
Rewrite(nombreArchivo)
```

que sitúa el cursor al principio del archivo `nombreArchivo` y, además, destruye cualquier posible información existente en él, como se muestra en la figura 14.4. Una vez ejecutada la instrucción anterior, el archivo está preparado para recibir operaciones de escritura como la siguiente:

```
Put(nombreArchivo)
```

que añade una componente más al archivo en la posición que indique el cursor y avanza éste un lugar en el archivo `nombreArchivo`, como puede verse en la representación gráfica de la figura 14.5. Después de ejecutar dicha instrucción, `nombreArchivo^` queda indefinido.

Así, teniendo en cuenta la declaración hecha anteriormente y suponiendo que la variable `unaTarjeta` posee la información que queremos almacenar en el archivo `archivoTarjetas`, realizaremos las siguientes instrucciones para añadir una nueva componente al archivo:

```
archivoTarjetas^:= unaTarjeta;
Put(archivoTarjetas)
```

Estas dos instrucciones son equivalentes a la instrucción:

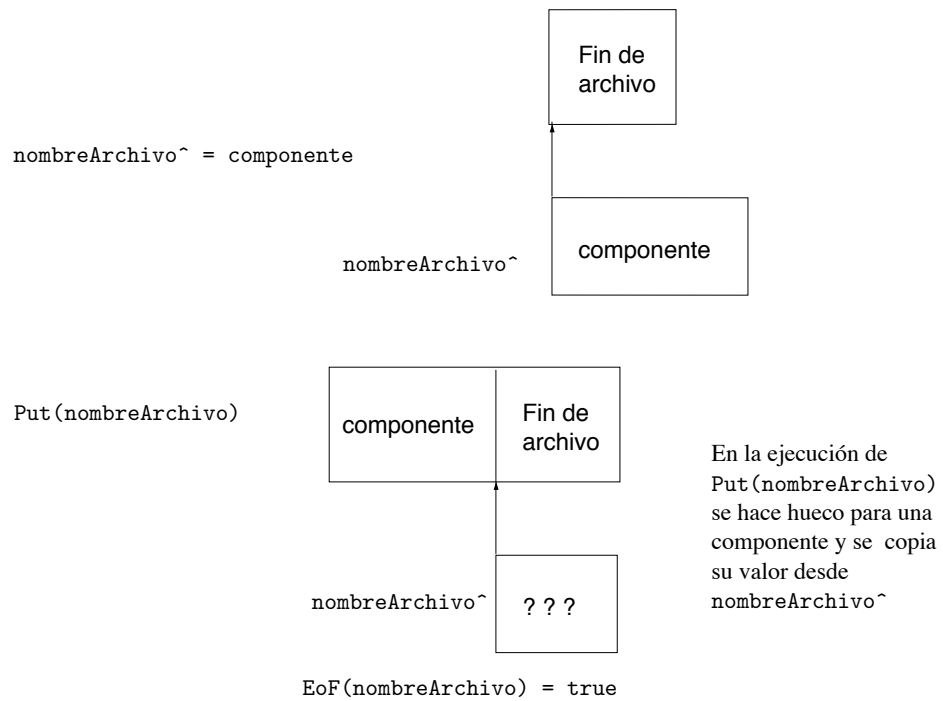


Figura 14.5.

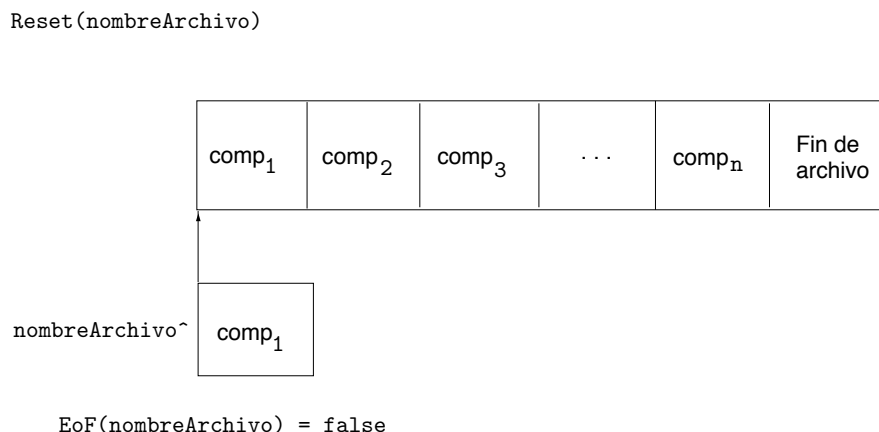


Figura 14.6.

```
Write(archivoTarjetas, unaTarjeta);
```

Dado que Pascal trabaja con archivos secuenciales, el cursor está siempre situado al final del archivo cuando se va a realizar una operación de escritura. Las sucesivas componentes se van añadiendo por el final del archivo, desplazando la marca de fin de archivo.

Una vez que hemos creado un archivo, es importante poder leer sus componentes. Dado que el acceso se hace de forma secuencial, hay que situar, nuevamente, el cursor al principio del archivo. Para ello, se ejecutará la instrucción:

```
Reset(nombreArchivo)
```

Con esta instrucción también se coloca el cursor en la primera componente del archivo. Si el archivo no está vacío, su primera componente está disponible en la variable `nombreArchivo^`, como puede comprobarse en la figura 14.6.

- ☞☞ Obsérvese que las funciones `Rewrite` y `Reset` son muy parecidas, ya que ambas sitúan el cursor al principio del archivo. La diferencia existente entre ambas es que la primera prepara el archivo exclusivamente para escritura (destruyendo la información existente), mientras que la segunda lo prepara exclusivamente para lectura.

Una vez que el cursor apunte a la primera componente, se puede mover el cursor a la siguiente posición y copiar la información de la siguiente componente de `nombreArchivo` utilizando la instrucción

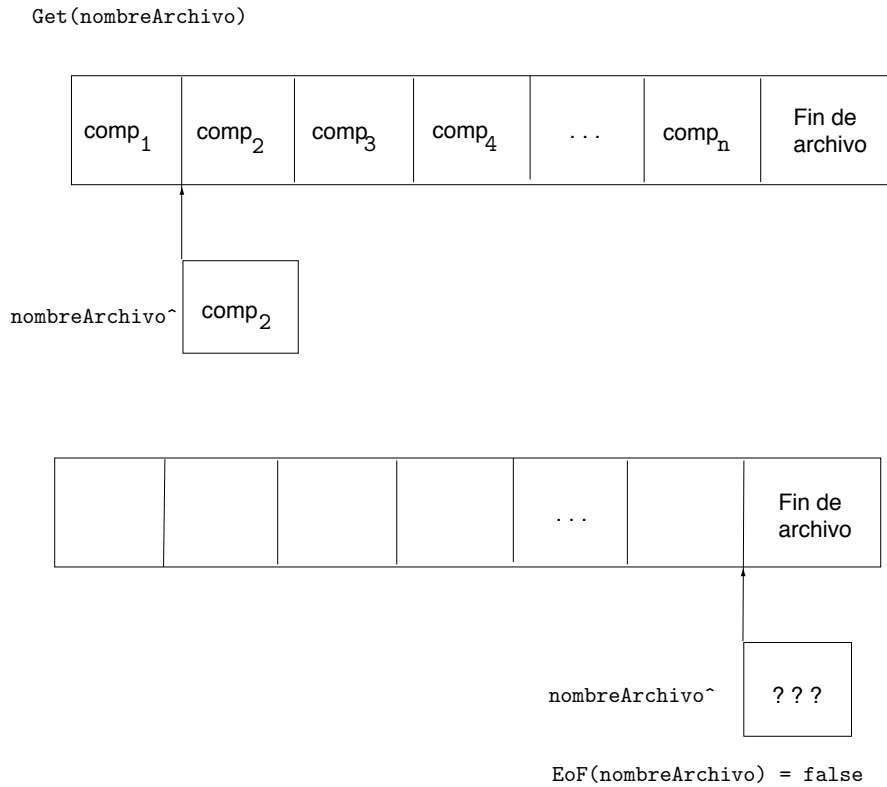


Figura 14.7.

Get(nombreArchivo)

Su efecto se muestra en la figura 14.7.

Siguiendo con el ejemplo anterior, si deseamos leer el contenido de la componente del archivo apuntada por el cursor realizaremos las siguientes instrucciones:

```
unaTarjeta:= archivoTarjetas^;
Get(archivoTarjetas)
```

o, equivalentemente,

```
Read(archivoTarjetas, unaTarjeta)
```

Es muy importante tener en cuenta que antes de leer de un archivo se ha de comprobar, mediante la función EoF, que quedan componentes por leer. Esta

función es imprescindible para realizar la lectura de archivos en los que desconocemos a priori el número de componentes o para detectar archivos vacíos.

En el siguiente ejemplo se presenta el esquema de un programa que lee un archivo completo controlando el final de archivo con EoF:

```

Program LeerArchivo (input, output, archivoTarjetas);

  type
    tTarjeta = array[1..50] of char;
    tArchivo = file of tTarjeta;
  var
    archivoTarjetas: tArchivo;
    una Tarjeta: tTarjeta
begin {LeerArchivo}
  ...
  Reset(archivoTarjetas);
  while not EoF(archivoTarjetas) do begin
    Read(archivoTarjetas, unaTarjeta);
    Procesar unaTarjeta
  end; {while}
end. {LeerArchivo}

```

El principal inconveniente que presentan los archivos en Pascal es que no se pueden alternar las operaciones de lectura y escritura en un archivo. Por tanto, si deseamos escribir y leer un archivo, en primer lugar se tendrá que escribir en él, y posteriormente situar el cursor al principio del archivo para leerlo.

Para realizar una copia de un archivo no se puede utilizar la asignación, a diferencia de los demás tipos de datos. Para poder copiar un archivo en otro se debe desarrollar un procedimiento cuyo código podría ser:

```

type
  tTarjeta = array[1..50] of char; {por ejemplo}
  tArchivoTarjetas = file of tTarjeta;
  ...
procedure CopiarArchivo(var archiEnt, archiSal: tArchivoTarjetas);
  {Efecto: archiSal:= archiEnt}
  var
    unaTarjeta: tTarjeta;
begin
  Reset(archiEnt);
  Rewrite(archiSal);
  while not EoF(archiEnt) do begin
    Read(archiEnt, unaTarjeta);
    Write(archiSal, unaTarjeta)
  end {while}
end; {CopiarArchivo}

```

### 14.3 Archivos de texto

Son muy frecuentes los programas en los que es necesario manejar textos, entendidos como secuencias de caracteres de una longitud usualmente grande, como, por ejemplo, una carta, un formulario, un informe o el código de un programa en un lenguaje de programación cualquiera. Estos textos se almacenan en archivos de caracteres que reciben el nombre de archivos de texto.

Los archivos de texto en Pascal se definen utilizando el tipo predefinido `text`. Este tipo de datos es un archivo con tipo base `char` al que se añade una marca de fin de línea. La generación y tratamiento del fin de línea se realiza con los procedimientos `WriteLn(archivoDeTexto)` y `ReadLn(archivoDeTexto)` y la función `EoLn(archivoDeTexto)`, que no se pueden utilizar con el tipo `file`.

El tipo `text` es un tipo estándar predefinido en Pascal, como `integer` o `char`, y por lo tanto, se pueden declarar variables de este tipo de la siguiente forma:

```
var
  archivoDeTexto: text;
```

En particular, los archivos `input` y `output` son de texto y representan la entrada y salida estándar, que en vez de emplear un disco utilizan, normalmente, el teclado y la pantalla como origen y destino de las secuencias de caracteres. Siempre que se utiliza una instrucción `Read` o `Write` sin indicar el archivo, se asume que dichas operaciones se realizan sobre los archivos `input` y `output`, respectivamente.

Ambos archivos, como es sabido, deben ser incluidos en el encabezamiento de todo programa sin redeclararlos dentro del programa, ya que se consideran declarados implícitamente como:

```
var
  input, output : text;
```

Además de esta declaración implícita se asumen, para los archivos `input` y `output`, las instrucciones `Reset(input)` y `ReWrite(output)` efectuadas al empezar un programa.

Debido a que los archivos de texto son muy utilizados, Pascal proporciona, además de las instrucciones comunes a todos los archivos con tipo genérico, funciones específicas de gran utilidad para los archivos de texto. Si `archivoDeTexto` es una variable de tipo `text`, según lo que vimos en el apartado anterior, sólo podríamos realizar instrucciones de la forma `Read(archivoDeTexto, c)` o bien `Write(archivoDeTexto, c)` siempre que `c` fuese de tipo `char`. Sin embargo, para los archivos `input` y `output` se permite que `c` pueda ser también de tipo `integer`, `real` o incluso `boolean`<sup>5</sup> o un array de caracteres para el caso de

---

<sup>5</sup>Sólo para archivos de salida.

`Write`.<sup>6</sup> Pascal permite este hecho no sólo a los archivos `input` y `output`, sino a todos los archivos de tipo `text`.

Pascal también permite que las instrucciones `Read` y `Write` tengan varios parámetros cuando se usan archivos de texto. Así, la instrucción

```
Read(archivoDeTexto, v1, v2, ..., vN)
```

es equivalente a:

```
Read(archivoDeTexto, v1);
Read(archivoDeTexto, v2);
...
Read(archivoDeTexto, vN)
```

y la instrucción

```
Write(archivoDeTexto, e1, e2, ..., eM)
```

es equivalente a:

```
Write(archivoDeTexto, e1);
Write(archivoDeTexto, e2);
...
Write(archivoDeTexto, eM)
```

donde `archivoDeTexto` es un archivo de tipo `text`, los parámetros `v1, v2, ..., vN` pueden ser de tipo `integer`, `real` o `char` y los parámetros `e1, e2, ..., eM` pueden ser de tipo `integer`, `real`, `char`, `boolean` o un array de caracteres. Las operaciones de lectura y escritura en archivos de texto son similares a los de `input` o `output` (véanse los apartados 4.3.2 y 4.3.3, donde se detalla su funcionamiento).

Los archivos de texto pueden estructurarse por líneas. Para manejar este tipo de organización existen la función `EoLn(archivoDeTexto)` para detectar el fin de línea y las instrucciones de lectura y escritura `ReadLn(archivoDeTexto)` y `WriteLn(archivoDeTexto)`. Su funcionamiento se describe a continuación:

- La función booleana `EoLn`<sup>7</sup> devuelve el valor `True` si se ha alcanzado la marca de fin de línea o `False` en otro caso. Cuando `EoLn(ArchivoDeTexto) ~> True` el valor de la variable apuntado por el cursor (`archivoDeTexto^`)

---

<sup>6</sup>En estos casos se ejecutarán automáticamente subprogramas de conversión.

<sup>7</sup>Del inglés *End Of LiNe* (fin de línea.)

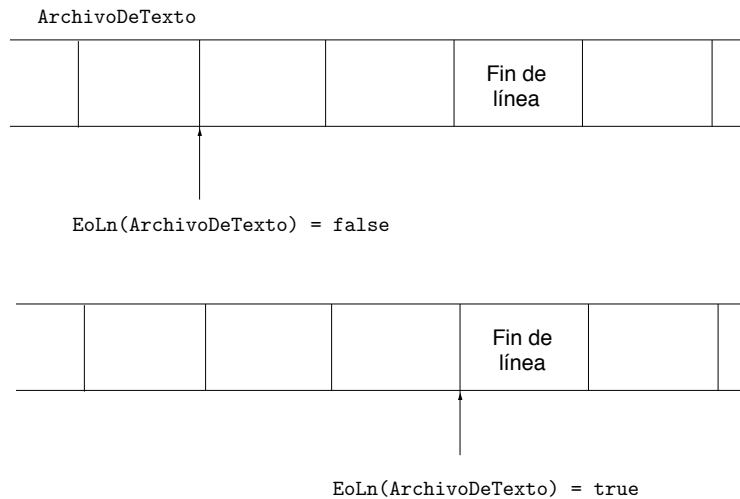


Figura 14.8.

es un carácter especial de fin de línea.<sup>8</sup> En la figura 14.8 puede verse una representación gráfica.

- La instrucción `ReadLn(archivoDeTexto, v)` lee el siguiente elemento del archivo, lo almacena en `v` y salta todos los caracteres hasta llegar al carácter especial de fin de línea, es decir, la instrucción es equivalente a:

```
Read(archivoDeTexto, v);
while not EoLn(archivoDeTexto) do
  Get(archivoDeTexto);
  {se avanza el cursor hasta encontrar el fin de línea}
Get(archivoDeTexto)
  {se salta el fin de línea}
```

con lo que con la siguiente llamada a `Read` se leerá el primer carácter de la siguiente línea. El parámetro `v` es opcional. Si se omite, el efecto de la instrucción `Read(archivoDeTexto)` sería el mismo salvo que no se almacena la componente que esté actualmente apuntada por el cursor.

De la misma forma que `Read`, la instrucción `ReadLn` puede utilizarse con el formato:

---

<sup>8</sup>En realidad este carácter especial depende del compilador que se utilice. Así, por ejemplo, en Pascal estándar se devuelve un espacio en blanco, mientras que Turbo Pascal devuelve el carácter de alimentación de línea (*Line Feed*, número 10 del juego de caracteres ASCII) seguido de un retorno de carro (*Carriage Return*, número 13 del juego de caracteres ASCII).

```
ReadLn(archivoDeTexto, v1, v2, ..., vN)
```

que es equivalente a:

```
Read(archivoDeTexto, v1);
Read(archivoDeTexto, v2);
...
ReadLn(archivoDeTexto, vN)
```

- La instrucción `WriteLn(archivoDeTexto, expresión)` se usa para escribir el valor de `expresión` en el `archivoDeTexto` y poner la marca de fin de línea. Asimismo, podemos omitir la `expresión`, produciéndose entonces simplemente un salto de línea.

Existe también la posibilidad de utilizar la instrucción:

```
WriteLn(archivoDeTexto, e1, e2, ..., eM)
```

que es equivalente a:

```
Write(archivoDeTexto, e1);
Write(archivoDeTexto, e2);
...
WriteLn(archivoDeTexto, eM)
```

Como una generalización de lo dicho en este tema sobre el uso de las funciones `EoLn` y `EoF`, se observa que la estructura general de los programas que procesan archivos de texto constan frecuentemente de dos bucles anidados: uno controlado por `EoF` y otro por `EoLn`, como se muestra a continuación, en un programa genérico de procesamiento de archivos de texto.

```
Program LecturaArchivoTexto (input, output, archivoDeTexto);
```

```
    Definición de tipos y declaración de variables
```

```
begin {Proceso de archivo de texto}
    ...
    Reset(archivoDeTexto);
    while not EoF(archivoDeTexto) do begin
        while not EoLn(archivoDeTexto) do begin
            Read(archivoDeTexto, dato);
```

```

    Procesar dato
  end; {while not EoLn}
  ReadLn(archivoDeTexto)
end {while not EoF}
...
end. {LecturaArchivoTexto}

```

En cualquiera de los casos considerados anteriormente, si se omite el archivo `archivoDeTexto` se supondrá por defecto el archivo `input` para el caso de `Read` o el archivo `output` para el caso de `Write`, produciéndose, en tal caso, una lectura por teclado o una escritura por pantalla respectivamente.

Finalmente se debe destacar que los archivos, al igual que todo tipo de datos compuesto, se pueden pasar como parámetros en funciones y procedimientos, pero no pueden ser el resultado de una función. No obstante, siempre se deben pasar como parámetros por variable.

## 14.4 Ejercicios

1. Basándose en el ejercicio 2 del capítulo 10 desarrolle un programa para las siguientes tareas:
  - (a) Invertir una serie de líneas, manteniendo su orden.
  - (b) Copiar una serie de líneas en orden inverso.
2. Escriba un programa que tabule los coeficientes binomiales  $\binom{n}{k}$  (véase el ejercicio 10 del capítulo 6), confeccionando el archivo `BINOM.TXT` de la siguiente forma:

```

1
1_1
1_2_1
1_3_3_1
1_4_6_4_1
1_5_10_10_5_1
...

```

Escriba una función que, en vez de hallar el coeficiente  $\binom{n}{k}$ , lo consulte en la tabla creada.

3. Se desea tabular los valores de la función de distribución normal

$$f(t) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \int_0^t e^{-x^2/2} dx$$

para los valores de  $t$  entre 0'00 y 1'99, aumentando a pasos de una centésima (véase el ejercicio 11 del capítulo 6).

Desarrolle un programa que construya un archivo de texto `NORMAL.TXT` con veinte filas de diez valores, así:

$$\begin{array}{cccc} f(0.00) & f(0.01) & \dots & f(0.09) \\ f(0.10) & f(0.11) & \dots & f(0.19) \\ \dots & \dots & \dots & \dots \\ f(1.90) & f(1.91) & \dots & f(1.99) \end{array}$$

Escriba una función que extraiga de la tabla `NORMAL.TXT` creada el valor correspondiente en vez de calcularlo.

4. Se ha efectuado un examen de tipo test, con 20 preguntas, a los alumnos de un grupo. En el archivo `EXAMENES.TXT` se hallan las respuestas, con arreglo al siguiente formato: en los primeros 25 caracteres se consigna el nombre, a continuación sigue un espacio en blanco y, en los 20 siguientes, letras de la 'A' a la 'E', correspondientes a las respuestas. En la primera línea del archivo `SOLUCION.TXT` se hallan las soluciones correctas consignadas en las 20 primeras casillas.

Se considera que una respuesta válida suma cuatro puntos, una incorrecta resta un punto y un carácter distinto a los posibles anula esa pregunta, no puntuando positiva ni negativamente. La nota se halla usando la expresión `Round(puntuación/8)`.

Escriba un programa que confeccione otro archivo `RESULT.TXT` con la lista de aprobados junto con la puntuación obtenida.

5. (a) Partiendo del conjunto de los primos entre 2 y 256 creado mediante el algoritmo de la criba de Eratóstenes (véase el apartado 11.3.3), escriba un programa que los guarde en un archivo de disco, `PRIMOS.TXT`.
  - (b) Defina una función que compruebe si un número menor que 256 es primo, simplemente consultando la tabla `PRIMOS.TXT`. (Al estar ordenada ascendentemente, con frecuencia será innecesario llegar al final de la misma.)
  - (c) Defina una función que compruebe si un número  $n$  entre cien y  $256^2$  es primo, tanteando como posibles divisores los números de la tabla `PRIMOS.TXT` entre 2 y  $\lfloor \sqrt{n} \rfloor$  que sea necesario.
  - (d) Integre los apartados anteriores que convengan en un programa que genere los primos menores que  $256^2$ .
6. Escriba un programa que convierta en archivo de texto, de nombre dado, las líneas introducidas desde el `input`, de forma similar al funcionamiento de la orden `copy ... con:` del DOS. Escriba igualmente un programa que muestre un archivo de texto, de nombre dado, por pantalla, tal como hace la orden `type` del DOS.
7. Escriba un subprograma que reciba dos archivos de texto y los mezcle, carácter a carácter, en un tercer archivo de texto. Si alguno de los archivos origen terminara antes que el otro, el subprograma añadirá al archivo destino lo que quede del otro archivo origen.
8. Añada al programa del ejercicio 6 del capítulo anterior una opción que permita almacenar y recuperar los datos de los productos en un archivo de texto o con tipo.



## Capítulo 15

# Algoritmos de búsqueda y ordenación

---

15.1 Algoritmos de búsqueda en arrays . . . . .	301
15.2 Ordenación de arrays . . . . .	306
15.3 Algoritmos de búsqueda en archivos secuenciales . . . . .	320
15.4 Mezcla y ordenación de archivos secuenciales . . . . .	322
15.5 Ejercicios . . . . .	329
15.6 Referencias bibliográficas . . . . .	330

---

Una vez vistos los distintos tipos de datos que el programador puede definir, se presentan en este capítulo dos de las aplicaciones más frecuentes y útiles de los tipos de datos definidos por el programador: la búsqueda y la ordenación. En particular, estas aplicaciones afectan directamente a los dos tipos de datos estudiados hasta ahora que permiten el almacenamiento de datos: los arrays, para datos no persistentes en el tiempo, y los archivos, para datos que deben ser recordados de una ejecución a otra de un determinado programa.

### 15.1 Algoritmos de búsqueda en arrays

Es evidente que, si tenemos datos almacenados, es interesante disponer de algún mecanismo que permita saber si un cierto dato está entre ellos, y, en caso afirmativo, localizar la posición en que se encuentra para poder trabajar con

él. Los mecanismos que realizan esta función son conocidos como algoritmos de búsqueda.

El problema que se plantea a la hora de realizar una búsqueda (concretamente en un array) puede ser enunciado de la siguiente forma:

Supongamos que tenemos un vector  $v$  con  $n$  elementos (los índices son los  $1 \dots n$ ) y pretendemos construir una función **Busqueda** que encuentre un índice  $i$  de tal forma que  $v[i] = \text{elem}$ , siendo  $\text{elem}$  el elemento que se busca. Si no existe tal índice, la función debe devolver un cero, indicando así que el elemento  $\text{elem}$  buscado no está en el vector  $v$ . En resumen,

$$\text{Busqueda} : \mathcal{V}_n(\text{tElem}) \times \text{tElem} \longrightarrow \{0, 1, \dots, n\}$$

de forma que

$$\text{Busqueda}(v, \text{elem}) = \begin{cases} i \in \{1, \dots, n\} & \text{si existe } i \text{ tal que } \text{elem} = v_i \\ 0 & \text{en otro caso} \end{cases}$$

En todo el capítulo se definen los elementos del vector con el tipo **tElem**:

```

const
  N = 100; {tamaño del vector}
type
  tIntervalo = 0..N;
  tVector = array [1..N] of tElem;

```

Para simplificar, digamos por ahora que **tElem** es un tipo ordinal, siendo por tanto comparables sus valores.

Los algoritmos más usuales que se pueden desarrollar para tal fin son los algoritmos de búsqueda secuencial y búsqueda binaria.<sup>1</sup>

### 15.1.1 Búsqueda secuencial

La búsqueda secuencial consiste en comparar secuencialmente el elemento deseado con los valores contenidos en las posiciones  $1, \dots, n$  hasta que, o bien encontremos el índice  $i$  buscado, o lleguemos al final del vector sin encontrarlo, concluyendo que el elemento buscado no está en el vector.

La búsqueda secuencial es un algoritmo válido para un vector cualquiera sin necesidad de que esté ordenado. También se puede aplicar con muy pocas

---

<sup>1</sup>Conocida también con los nombres de búsqueda dicotómica o por bipartición.

variaciones a otras estructuras secuenciales, como, por ejemplo, a los archivos (véase el apartado 15.3).

El primer nivel en el diseño de la función `BusquedaSec` puede ser:

```
ind:= 0;
Buscar elem en v;
Devolver el resultado de la función
```

Refinando *Buscar elem en v*, se tiene:

```
repetir
  ind:= ind + 1
hasta que v[ind] = elem o ind = n
```

Por último, refinando *Devolver el resultado de la función* se obtiene:

```
si v[ind] = elem entonces
  BusquedaSec:= ind
si no
  BusquedaSec:= 0
```

Una posible implementación de la función `BusquedaSec` siguiendo el esquema de este algoritmo secuencial es:

```
function BusquedaSec(v: tVector; elem: tElem): tIntervalo;
  {Dev. 0 (si elem no está en v) ó i (si v[i] = elem)}
  var
    i: tIntervalo;
begin
  i:= 0; {se inicia el contador}
  repeat
    {Inv.:  $\forall j, 0 \leq j \leq i \Rightarrow v[j] \neq \text{elem}$ }
    i:= i + 1
  until (v[i] = elem) or (i = N);
    {v[i] = elem}
  if v[i] = elem then {se ha encontrado el elemento elem}
    BusquedaSec:= i
  else
    BusquedaSec:= 0
end; {BusquedaSec}
```

### 15.1.2 Búsqueda secuencial ordenada

El algoritmo de búsqueda secuencial puede ser optimizado si el vector  $v$  está ordenado (supongamos que de forma creciente). En este caso, la búsqueda secuencial desarrollada anteriormente es ineficiente, ya que, si el elemento buscado  $elem$  no se encuentra en el vector, se tendrá que recorrer todo el vector, cuando se sabe que si se llega a una componente con valor mayor que  $elem$ , ya no se encontrará el valor buscado.

Una primera solución a este nuevo problema sería modificar la condición de salida del bucle **repeat** cambiando  $v[i]=elem$  por  $v[i]>=elem$ , debido a que el vector se encuentra ordenado de forma creciente:

```

function BusquedaSecOrd(v: tVector; elem: tElem): tIntervalo;
  {PreC.: v está ordenado crecientemente}
  {Dev. 0 (si elem no está en v) ó i (si v[i] = elem)}
  var
    i: tIntervalo;
begin
  i:= 0;
  repeat
    {Inv.:  $\forall j, 0 \leq j \leq i, \Rightarrow v[j] \neq elem$ }
    i:= i + 1
  until (v[i]>=elem) or (i=N);
    {v[i]=elem}
  if v[i] = elem then {se ha encontrado el elemento elem}
    BusquedaSecOrd:= i
  else
    BusquedaSecOrd:= 0
end; {BusquedaSecOrd}

```

Esta solución también se puede aplicar a archivos secuenciales ordenados (véase el apartado 15.3).

### 15.1.3 Búsqueda binaria

El hecho de que el vector esté ordenado se puede aprovechar para conseguir una mayor eficiencia en la búsqueda planteando el siguiente algoritmo: comparar  $elem$  con el elemento central; si  $elem$  es ese elemento ya hemos terminado, en otro caso buscamos en la mitad del vector que nos interese (según sea  $elem$  menor o mayor que el elemento mitad, buscaremos en la primera o segunda mitad del vector, respectivamente). Posteriormente, si no se ha encontrado el elemento repetiremos este proceso comparando  $elem$  con el elemento central del subvector seleccionado, y así sucesivamente hasta que o bien encontremos el valor  $elem$  o bien podamos concluir que  $elem$  no está (porque el subvector de búsqueda está

vacío). Este algoritmo de búsqueda recibe el nombre de búsqueda binaria, ya que va dividiendo el vector en dos subvectores de igual tamaño.

Vamos ahora a realizar una implementación de una función siguiendo el algoritmo de búsqueda binaria realizando un diseño descendente del problema, del cual el primer refinamiento puede ser:

```
Asignar valores iniciales extInf, extSup, encontrado;
Buscar elem en v[extInf..extSup];
Devolver el resultado de la función
```

Refinando *Asignar valores iniciales* se tiene:

```
extInf:= 1;
extSup:= N;
  {se supone que N es el tamaño del array inicial}
encontrado:= False;
```

En un nivel más refinado de *Buscar elem en* v[extInf..extSup] se tiene:

```
mientras el vector no sea vacío y
no se ha encontrado el valor c hacer
  calcular el valor de posMed;
  si v[posMed] = elem entonces
    actualizar el valor de encontrado
  si no
    actualizar los valores extInf o extSup según donde esté elem
```

Refinando *devolver el resultado de la función* obtenemos:

```
si se ha encontrado el valor entonces
  BusquedaBinaria:= posMed;
si no
  BusquedaBinaria:= 0;
```

Con todo esto, una posible implementación sería:

```
function BusquedaBinaria(v: tVector; elem: tElem): tIntervalo;
  {PreC.: v está ordenado crecientemente}
  {Dev. 0 (si elem no está en v) ó i (si v[i] = elem)}
  var
    extInf, extSup, {extremos del intervalo}
    posMed: tIntervalo; {posición central del intervalo}
    encontrado: boolean;
```

```

begin
  extInf:= 1;
  extSup:= N;
  encontrado:= False;
  while (not encontrado) and (extSup >= extInf) do begin
    {Inv.: si elem está en v, ⇒ v[extInf] ≤ elem ≤ v[extSup]}
    posMed:= (extSup + extInf) div 2;
    if elem = v[posMed] then
      encontrado:= True
    else if elem > v[posMed] then
      {se actualizan los extremos del intervalo}
      extInf:= posMed + 1
    else
      extSup:= posMed - 1
  end; {while}
  if encontrado then
    BusquedaBinaria:= palMed
  else
    BusquedaBinaria:= 0
end; {BusquedaBinaria}

```

La necesidad de acceder de forma directa a las componentes intermedias no permite la aplicación de este tipo de soluciones a archivos secuenciales.

## 15.2 Ordenación de arrays

En muchas situaciones se necesita tener ordenados, según algún criterio, los datos con los que se trabaja para facilitar su tratamiento. Así, por ejemplo, en un vector donde se tengan almacenados los alumnos de cierta asignatura junto con la calificación obtenida, sería interesante ordenar los alumnos por orden alfabético, o bien, ordenar el vector según la calificación obtenida para poder sacar una lista de “Aprobados” y “Suspensos”.

En este apartado se presentan los algoritmos de ordenación más usuales, de los muchos existentes. Su objetivo común es resolver el problema de ordenación que se enuncia a continuación:

Sea  $v$  un vector con  $n$  componentes de un mismo tipo,  $tElem$ . Definimos la función ordenación como:

$$\text{ordenación} : \mathcal{V}_n(tElem) \longrightarrow \mathcal{V}_n(tElem)$$

de tal forma que  $\text{ordenación}(v) = v'$  donde  $v'=(v'_1, \dots, v'_n)$  es una permutación de  $v=(v_1, \dots, v_n)$  tal que  $v'_1 \leq v'_2 \leq \dots \leq v'_n$  donde  $\leq$  es la relación de orden elegida para clasificar los elementos de  $v$ .

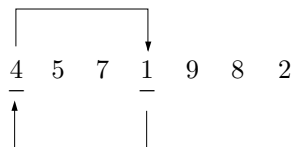
- ☉☉ En todos los algoritmos que se van a estudiar a continuación, se supondrá que la ordenación se llevará a cabo en forma creciente. Es obvio que las modificaciones que se tendrían que realizar para ordenar un vector con otra relación de orden son inmediatas y se dejan para el lector.

### 15.2.1 Selección directa

Este algoritmo resuelve el problema de la ordenación recorriendo el vector y seleccionando en cada recorrido el menor elemento para situarlo en su lugar correspondiente. El esquema básico del algoritmo de selección directa es:

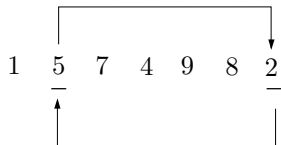
1. Se sitúa en  $v_1$  el menor valor entre  $v_1, \dots, v_n$ . Para ello se intercambian los valores de  $v_1$  y  $v_m$  siendo  $v_m = \min_{k=1, \dots, n} \{v_k\}$ .

Por ejemplo, para el vector (4, 5, 7, 1, 9, 8, 2), este primer paso produce el intercambio representado en la siguiente figura:



2. Se sitúa en  $v_2$  el menor valor entre  $v_2, \dots, v_n$ . Para ello se intercambian los valores de  $v_2$  y  $v_m$  siendo  $v_m = \min_{k=2, \dots, n} \{v_k\}$ .

En el ejemplo, se produce el intercambio:



...

- (j-1). Se sitúa en  $v_{j-1}$  el menor valor entre  $v_{j-1}, \dots, v_n$ . Para ello se intercambian los valores de  $v_{j-1}$  y  $v_m$  siendo  $v_m = \min_{k=j-1, \dots, n} \{v_k\}$ .

...

- (n-1). Se sitúa en  $v_{n-1}$  el menor valor entre  $v_{n-1}$  y  $v_n$ . Para ello se intercambian los valores de  $v_{n-1}$  y  $v_n$  si es necesario.

La situación final en el ejemplo es representada en la siguiente figura:

1   2   4   5   7   8   9  
                                   \_   \_

El primer nivel de diseño del algoritmo es:

*para cada i entre 1 y n - 1 hacer*  
     *Colocar en v<sub>i</sub> el menor entre v<sub>i</sub>, ..., v<sub>n</sub>;*  
*Devolver v, ya ordenado*

donde *Colocar en v<sub>i</sub> el menor entre v<sub>i</sub>, ..., v<sub>n</sub>* es:

```
valMenor:= v[i];
posMenor:= i;
para cada j entre i + 1 y n hacer
    si v[j] < valMenor entonces
        valMenor:= v[j];
        posMenor:= j
    fin {si}
fin {para}
Intercambiar v[i] con v[PosMenor];
```

Por lo tanto, una posible implementación de ordenación de un vector, siguiendo el algoritmo de selección directa, podría ser la siguiente:

```
procedure SeleccionDirecta(var v: tVector);
    {Efecto: se ordena v ascendentemente}
    var
        i, j, posMenor: tIntervalo;
        valMenor, aux: integer;
begin
    for i:= 1 to N-1 do begin
        {Inv.:  $\forall j, 1 \leq j \leq i-1, \Rightarrow v[j] \leq v[j+1]$ 
        y además todos los v[i]...v[N] son mayores que v[i-1]}
        valMenor:= v[i];
        {se dan valores iniciales}
        posMenor:= i;
        for j:= i + 1 to n do
            if v[j] < valMenor then begin
                {se actualiza el nuevo valor menor y la
                posición donde se encuentra}
                valMenor:= v[j];
                posMenor:= j
            end {if}
    end {if}
```

```

    if posMenor <> i then begin
        {Si el menor no es v[i], se intercambian los valores}
        aux:= v[i];
        v[i]:= v[posMenor];
        v[posMenor]:= aux
    end {if}
end; {for i}
end; {SeleccionDirecta}

```

### 15.2.2 Inserción directa

Este algoritmo recorre el vector  $v$  insertando el elemento  $v_i$  en su lugar correcto entre los ya ordenados  $v_1, \dots, v_{i-1}$ . El esquema general de este algoritmo es:

1. Se considera  $v_1$  como primer elemento.
2. Se inserta  $v_2$  en su posición correspondiente en relación a  $v_1$  y  $v_2$ .
3. Se inserta  $v_3$  en su posición correspondiente en relación a  $v_1, \dots, v_3$ .
- ...
- i. Se inserta  $v_i$  en su posición correspondiente en relación a  $v_1, \dots, v_i$ .
- ...
- n. Se inserta  $v_n$  en su posición correspondiente en relación a  $v_1, \dots, v_n$ .

En el diagrama de la figura 15.1 se muestra un ejemplo de aplicación de este algoritmo.

Atendiendo a esta descripción, el diseño descendente de este algoritmo tiene como primer nivel el siguiente:

*para cada  $i$  entre 2 y  $N$  hacer*  
*Situar  $v[i]$  en su posición ordenada respecto a  $v[1], \dots, v[i-1]$*

Donde *Situar  $v[i]$  en su posición ordenada respecto a  $v[1], \dots, v[i-1]$*  puede refinarse de la siguiente forma:

*Localizar la posición  $j$  entre 1 e  $i-1$  correspondiente a  $v[i]$ ;*  
*Desplazar una posición las componentes  $v[j+1], \dots, v[i-1]$ ;*  
 $v[j] := v[i]$

Por lo tanto, una implementación de este algoritmo será:

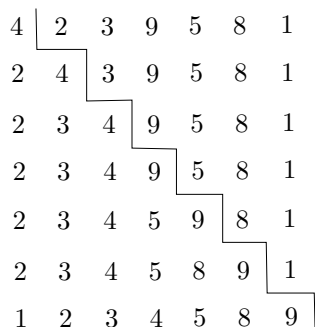


Figura 15.1.

```

procedure InsercionDirecta(var v: tVector);
  {Efecto: se ordena v ascendentemente}
  var
    i, j: tIntervalo;
    aux: tElem;
begin
  for i:= 2 to N do begin
    {Inv.:  $\forall j, 1 \leq j < i, \Rightarrow v[j] \leq v[j+1]$ }
    aux:= v[i];
    {se dan los valores iniciales}
    j:= i - 1;
    while (j >= 1) and (v[j] > aux) do begin
      v[j+1]:= v[j];
      {Desplazamiento de los valores mayores que v[i]}
      j:= j-1
    end; {while}
    v[j+1]:= aux
  end {for}
end; {InsercionDirecta}

```

### 15.2.3 Intercambio directo

El algoritmo por intercambio directo recorre el vector buscando el menor elemento desde la última posición hasta la actual y lo sitúa en dicha posición. Para ello, se intercambian valores vecinos siempre que estén en orden decreciente. Así se baja, mediante sucesivos intercambios, el valor menor hasta la posición deseada. Más concretamente, el algoritmo consiste en:

1. Situar el elemento menor en la primera posición. Para ello se compara el último elemento con el penúltimo, intercambiando sus valores si están

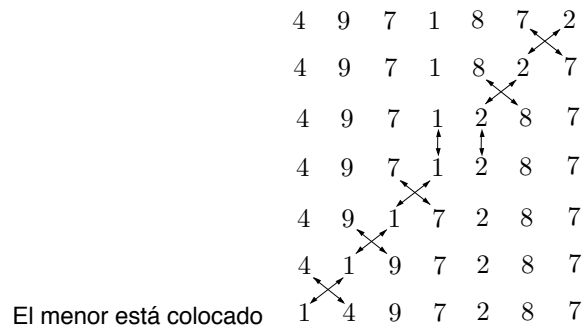


Figura 15.2.

en orden decreciente. A continuación se comparan el penúltimo elemento con el anterior, intercambiándose si es necesario, y así sucesivamente hasta llegar a la primera posición. En este momento se puede asegurar que el elemento menor se encuentra en la primera posición.

Así, para el vector  $(4, 9, 7, 1, 8, 7, 2)$ , se realiza el proceso representado en la figura 15.2.

2. Situar el segundo menor elemento en la segunda posición. Para ello se procede como antes finalizando al llegar a la segunda posición, con lo que se sitúa el elemento buscado en dicha posición.

En el ejemplo, se obtiene en este paso el vector  $(1, 2, 4, 9, 7, 7, 8)$ .

... Se repite el proceso para las posiciones intermedias.

- (n-1). Se comparan los dos últimos valores, intercambiándose si están en orden decreciente, obteniéndose así el vector ordenado. En el caso del ejemplo se obtiene el vector  $(1, 2, 4, 7, 7, 8, 9)$ .

El pseudocódigo correspondiente al primer nivel de diseño de este algoritmo es:

```

para i entre 1 y n-1 hacer
    Desplazar el menor valor desde  $v_n$  hasta  $v_i$ , intercambiando
    pares vecinos, si es necesario
Devolver  $v$ , ya ordenado

```

Por lo tanto, una implementación del algoritmo puede ser:

```

procedure OrdenacionPorIntercambio(var v: tVector);
  {Efecto: se ordena v ascendentemente}
  var
    i, j: tIntervalo;
    aux: tElem;
  begin
    for i:= 1 to N-1 do
      {Inv.:  $\forall j, 1 \leq j < i, \Rightarrow v[j] \leq v[k], \forall k$  tal que  $j \leq k < N$ }
      for j:= N downto i + 1 do
        {Se busca el menor desde atrás y se sitúa en  $v_i$ }
        if v[j-1] > v[j] then begin {intercambio}
          aux:= v[j];
          v[j]:= v[j-1];
          v[j-1]:= aux
        end {if}
      end; {OrdenacionPorIntercambio}
  
```

### 15.2.4 Ordenación rápida (*Quick Sort*)

El algoritmo de *ordenación rápida*<sup>2</sup> debido a Hoare, consiste en dividir el vector que se desea ordenar en dos bloques. En el primer bloque se sitúan todos los elementos del vector que son menores que un cierto valor de  $v$  que se toma como referencia (valor pivote), mientras que en el segundo bloque se colocan el resto de los elementos, es decir, los que son mayores que el valor pivote. Posteriormente se ordenarán (siguiendo el mismo proceso) cada uno de los bloques, uniéndolos una vez ordenados, para formar la solución. En la figura 15.3 se muestran gráficamente las dos fases de ordenación.

Evidentemente, la condición de parada del algoritmo se da cuando el bloque que se desea ordenar esté formado por un único elemento, en cuyo caso, obviamente, el bloque ya se encuentra ordenado.

También se puede optar por detener el algoritmo cuando el número de elementos del bloque sea suficientemente pequeño (generalmente con un número aproximado de 15 elementos), y ordenar éste siguiendo alguno de los algoritmos vistos anteriormente (el de inserción directa, por ejemplo).

- ☉☉ El número elegido de 15 elementos es orientativo. Se debería elegir dicha cantidad mediante pruebas de ensayo para localizar el valor óptimo.

Aunque cualquier algoritmo de ordenación visto anteriormente sea “más lento” (como veremos en el capítulo de complejidad algorítmica) que el *Quick Sort*, este último pierde gran cantidad de tiempo en clasificar los elementos en

---

<sup>2</sup>*Quick Sort* en inglés.

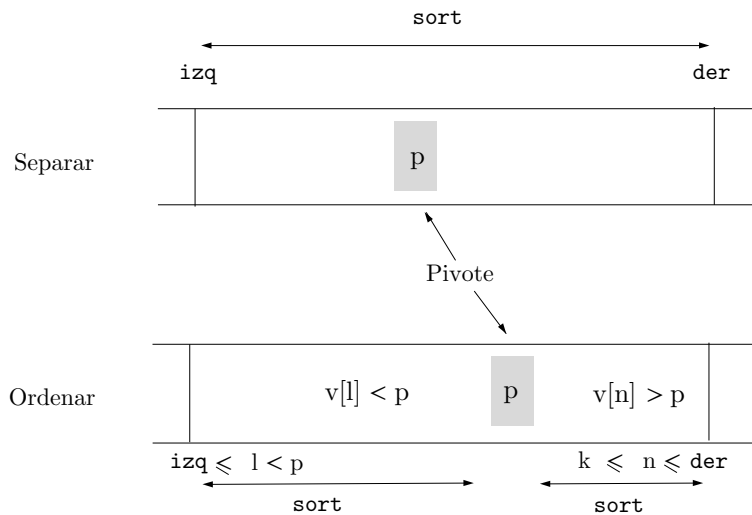


Figura 15.3.

los dos bloques, por lo que, cuando el número de elementos es pequeño, no es “rentable” utilizar *Quick Sort*.

En este apartado vamos a desarrollar el algoritmo *Quick Sort* con la primera condición de parada, dejando al lector el desarrollo de la segunda versión.

Este algoritmo sigue el esquema conocido con el nombre de *divide y vencerás* (véase el apartado 20.2) que, básicamente, consiste en subdividir el problema en dos iguales pero de menor tamaño para posteriormente combinar las dos soluciones parciales obtenidas para producir la solución global.

El seudocódigo correspondiente al primer nivel en el diseño descendente del algoritmo *Quick Sort* es:

```

si v es de tamaño 1 entonces
  v ya está ordenado
si no
  Dividir v en dos bloques A y B
  con todos los elementos de A menores que los de B
fin {si}
  Ordenar A y B usando Quick Sort
  Devolver v ya ordenado como concatenación
  de las ordenaciones de A y de B

```

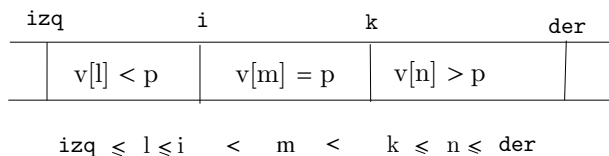


Figura 15.4.

Donde *Dividir v en dos bloques A y B* se puede refinar en:

*Elegir un elemento p (pivote) de v*  
*para cada elemento del vector hacer*  
*si elemento < p entonces*  
*Colocar elemento en A, el subvector con los elementos de v*  
*menores que p*  
*en otro caso*  
*Colocar elemento en B, el subvector con los elementos de v*  
*mayores que p*

- ☉☉ De cara a la implementación, y por razones de simplicidad y ahorro de memoria, es preferible situar los subvectores sobre el propio vector original v en lugar de generar dos nuevos arrays.

Con todo lo anterior, una implementación de este algoritmo podría ser:

```

procedure QuickSort(var v: tVector);
  {Efecto: se ordena v ascendentemente}
  procedure SortDesdeHasta(var v: tVector; izq,der: tIntervalo);
    {Efecto: v[izq..der] está ordenado ascendentemente}
    var
      i,j: tIntervalo;
      p,aux: tElem;
  begin
    i:= izq;
    j:= der;
    {se divide el vector v[izq..der] en dos trozos eligiendo como
     pivote p el elemento medio del array}
    p:= v[(izq + der) div 2];
    {si i >= d el subvector ya está ordenado}
    {Inv.:  $\forall s, izq \leq s < i, \Rightarrow v[s] < p$ 
     y  $\forall t$  tal que  $j < t \leq der \Rightarrow v[s] > p$ }
  
```

```

while i < j do begin
  {se reorganizan los dos subvectores}
  while v[i] < p do
    i:= i + 1;
  while p < v[j] do
    j:= j - 1;
  if i <= j then begin
    {intercambio de elementos}
    aux:= v[i];
    v[i]:= v[j];
    v[j]:= aux;
    {ajuste de posiciones}
    i:= i + 1;
    j:= j - 1
  end {if}
end; {while}
if izq < j then
  SortDesdeHasta(v,izq,j);
if i < der then
  SortDesdeHasta(v,i,der)
end; {SortDesdeHasta}

begin {QuickSort}
  SortDesdeHasta(v,1, n)
end; {QuickSort}

```

Para facilitar la comprensión del método, se ha realizado un ejemplo de su funcionamiento provocando algunas llamadas sobre el vector inicial  $v = [0, 3, 86, 20, 27]$  y mostrando el vector tras cada llamada:

```

                                v = [0, 3, 86, 20, 27]
SortDesdeHasta(v,1,5)          v = [0, 3, 27, 20, 86]
SortDesdeHasta(v,1,4)          v = [0, 3, 27, 20, 86]
SortDesdeHasta(v,1,4)          v = [0, 3, 27, 20, 86]

```

A propósito de este algoritmo, es necesaria una observación sobre su eficiencia. El mejor rendimiento se obtiene cuando el pivote elegido da lugar a dos subvectores de igual tamaño, dividiéndose así el vector en dos mitades. Este algoritmo también proporciona un coste mínimo cuando los elementos del vector se distribuyen aleatoriamente (equiprobablemente).

Sin embargo, puede ocurrir que los elementos estén dispuestos de forma que, cada vez que se elige el pivote, todos los elementos queden a su izquierda o todos a su derecha. Entonces, uno de los dos subvectores es vacío y el otro carga con todo el trabajo. En este caso, el algoritmo da un pésimo rendimiento, comparable a

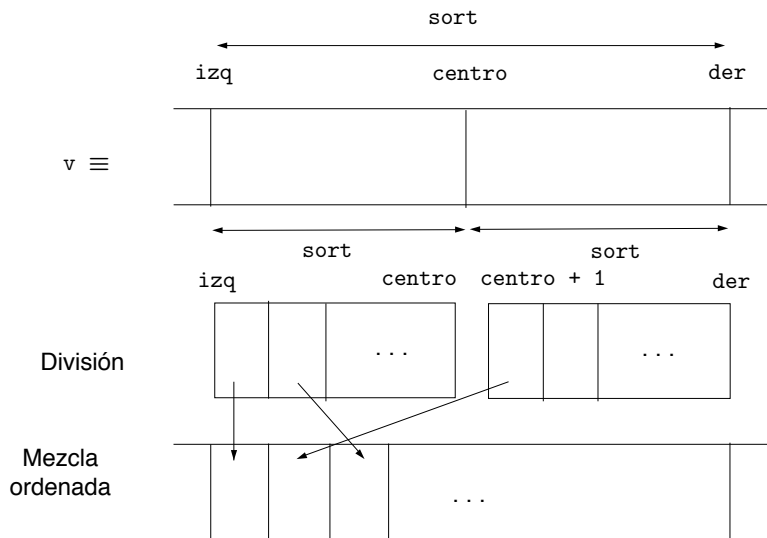


Figura 15.5.

los algoritmos de selección, inserción e intercambio. Un ejemplo de esta situación se tiene cuando el vector por ordenar tiene los elementos inicialmente dispuestos en orden descendente y el pivote elegido es, precisamente, el primer (o el último) elemento.

### 15.2.5 Ordenación por mezcla (*Merge Sort*)

Al igual que *Quick Sort*, el algoritmo de ordenación *Merge Sort* va a utilizar la técnica divide y vencerás. En este caso la idea clave del algoritmo consiste en dividir el vector  $v$  en dos subvectores A y B (de igual tamaño, si es posible), sin tener en cuenta ningún otro criterio de división.

Posteriormente se mezclarán ordenadamente las soluciones obtenidas al ordenar A y B (aplicando nuevamente, a cada uno de los subvectores, el algoritmo *Merge Sort*). En la figura 15.5 se muestra un esquema de este algoritmo.

Para *Merge Sort* también se pueden considerar las condiciones de parada descritas en el apartado anterior para el algoritmo *Quick Sort*, desarrollándose a continuación el diseño descendente y la implementación para el primer caso de parada (cuando se llega a subvectores de longitud 1). Se deja como ejercicio para el lector el desarrollo de la implementación de *Merge Sort* con la segunda condición de parada, es decir, la combinación de *Merge Sort* con otro algoritmo de ordenación cuando se llega a subvectores de una longitud dada.

El algoritmo *Merge Sort* puede esbozarse como sigue:

```

si v es de tamaño 1 entonces
  v ya está ordenado
si no
  Dividir v en dos subvectores A y B
fin {si}
  Ordenar A y B usando Merge Sort
  Mezclar las ordenaciones de A y B para generar el vector ordenado.

```

En este caso, el paso *Dividir v en dos subvectores A y B* consistirá en:

```

Asignar a A el subvector  $[v_1, \dots, v_{n \text{ div } 2}]$ 
Asignar a B el subvector  $[v_{n \text{ div } 2 + 1}, \dots, v_n]$ 

```

mientras que *Mezclar las ordenaciones de A y B* consistirá en desarrollar un procedimiento (que llamaremos **Merge**) encargado de ir entremezclando adecuadamente las componentes ya ordenadas de A y de B para obtener el resultado buscado.

De acuerdo con este diseño se llega a la implementación que se muestra a continuación. Al igual que en la implementación de *Quick Sort*, se sitúan los subvectores sobre el propio vector original *v* en lugar de generar dos nuevos arrays.

```

procedure MergeSort(var vector: tVector);
  {Efecto: se ordena vector ascendentemente}
  procedure MergeSortDesdeHasta(var v: vector; izq, der: integer);
    {Efecto: se ordena v[izq..der] ascendentemente}
    var
      centro : tIntervalo;

  procedure Merge(vec: tVector; iz, ce, de: tIntervalo;
    var w: tVector);
    {Efecto: w := mezcla ordenada de los subvectores v[iz..ce] y
      v[ce+1..de]}
    var
      i, j, k: 1..N;
  begin {Merge}
    i := iz;
    j := ce + 1;
    k := iz;
    {k recorre w, vector que almacena la ordenación total}
    while (i <= ce) and (j <= de) do begin
      {Inv.:  $\forall m, iz \leq m < k, \Rightarrow w[m] \leq w[m+1]$ }
      if vec[i] < vec[j] then begin

```

```

        w[k]:= vec[i];
        i:= i + 1
    end {Then}
    else begin
        w[k]:= vec[j];
        j:= j + 1
    end; {Else}
    k:= k + 1;
end; {While}
for k:= j to de do
    w[k]:= vec[k]
for k:= i to ce do
    w[k+de-ce]:= vec[k]
end; {Merge}

begin {MergeSortDesdeHasta}
    centro:= (izq + der) div 2;
    if izq < centro then
        MergeSortDesdeHasta(v, izq, centro);
    if centro < der then
        MergeSortDesdeHasta(v, centro+1, der);
    Merge(v, izq, centro, der, v)
end; {MergeSortDesdeHasta}

begin {MergeSort}
    MergeSortDesdeHasta(vector, 1, N)
end; {MergeSort}

```

Para facilitar la comprensión del método se ha realizado un ejemplo, provocando algunas llamadas sobre el vector inicial  $v = [8, 5, 7, 3]$  y mostrando el vector tras cada llamada:

	$v = [8, 5, 7, 3]$
<code>MergeSortDesdeHasta(v,1,2)</code>	$v = [5, 8, 7, 3]$
<code>MergeSortDesdeHasta(v,3,4)</code>	$v = [5, 8, 3, 7]$
<code>Merge(v,1,2,4,v)</code>	$v = [3, 5, 7, 8]$

### 15.2.6 Vectores paralelos

Supóngase que se desea ordenar un vector `vectFich` de la forma

`array[1..N] of tFicha`

siendo el tipo `tFicha` de un gran tamaño.

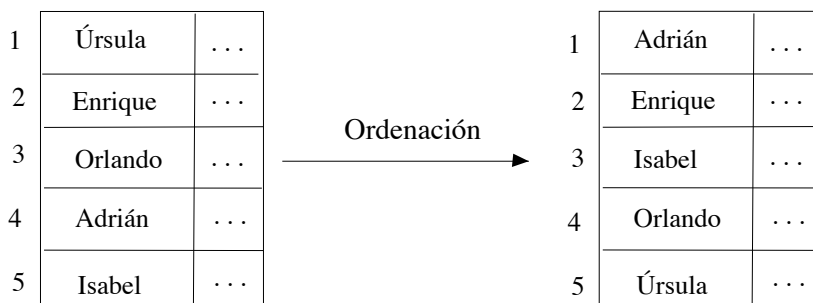


Figura 15.6.

Una posibilidad interesante consiste en crear otro vector `vectPosic` de la forma

**array[1..N] of [1..N]**

cuyas componentes refieren una ficha completa, mediante su posición, siendo inicialmente

$$\text{vectPosic}[i] = i \quad \forall i \in \{1, \dots, N\}$$

El interés de esta representación indirecta reside en aplicar este método a cualquiera de los algoritmos de ordenación estudiados, con la ventaja de que la ordenación se puede efectuar intercambiando las posiciones en vez de los registros (extensos) completos.

Por ejemplo, supongamos que el tipo de datos `tFicha` es un registro con datos personales, entre los que se incluye el nombre. La figura 15.6 muestra el vector `vectFich` antes y después de la ordenación.

Si no se emplea el método de los vectores paralelos, el algoritmo de ordenación empleado debería incluir instrucciones como las siguientes:

```

if vectFich[i].nombre > vectFich[j].nombre then begin
  {Intercambiar vectFich[i], vectFich[j]}
  elemAux:= vectFich[i];
  vectFich[i]:= vectFich[j];
  vectFich[j]:= elemAux
end

```

Como se puede comprobar, en las instrucciones de intercambio es necesario mantener tres copias de elementos del tipo `tElemento`. Sin embargo, si se utiliza un vector paralelo (como se muestra en la figura 15.7), sólo se producen cambios en el vector de posiciones. El fragmento de código correspondiente a los intercambios quedaría como sigue:

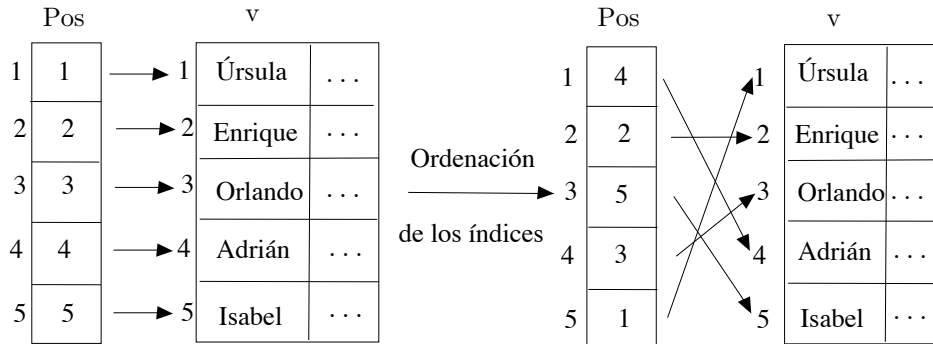


Figura 15.7.

```

if vectFich[vectPosic[i]].nombre > vectFich[vectPosic[j]].nombre
then begin
  {Intercambiar vectPosic[i], vectPosic[j] ∈ [1..N]}
  posAux:= vectPosic[i];
  vectPosic[i]:= vectPosic[j];
  vectPosic[j]:= posAux
end

```

Así se manipulan únicamente tres copias de posiciones, que son de tipo integer, con el consiguiente ahorro de tiempo y espacio, ya que es más eficiente intercambiar dos índices que dos elementos (siempre y cuando éstos sean grandes).

Además, el método de los vectores paralelos tiene la ventaja añadida de que permite mantener varios índices, que en el ejemplo permitirían realizar ordenaciones por nombre, DNI, etc.

Finalmente, también conviene indicar que la idea de manejar las posiciones en lugar de los elementos mismos se explota ampliamente en programación dinámica (véanse los capítulos 16 y 17).

### 15.3 Algoritmos de búsqueda en archivos secuenciales

La variedad de algoritmos de búsqueda en archivos se ve muy restringida en Pascal por la limitación a archivos de acceso secuencial. Este hecho obliga a que la localización del elemento buscado se haga examinando las sucesivas

componentes del archivo. Por tanto, la estrategia de búsqueda será la misma que se emplea en los algoritmos de búsqueda secuencial en arrays (véanse los apartados 15.1.1 y 15.1.2).

En este breve apartado nos limitaremos a mostrar la adaptación directa de estos algoritmos de búsqueda para archivos arbitrarios y para archivos ordenados.

### 15.3.1 Búsqueda en archivos arbitrarios

Supongamos que se dispone de un archivo definido de la siguiente forma:

```

type
  tElem = record
    clave: tClave;
    resto de la información
  end;
  tArchivoElems = file of tElem;

```

Con esto, el procedimiento de búsqueda queda como sigue (por la simplicidad de la implementación se han obviado los pasos previos de diseño y algunos aspectos de la corrección):

```

procedure Buscar(var f: tArchivoElems; c: tClave; var elem: tElem;
  var encontrado: boolean);
  {PostC: si c está en f entonces encontrado = True y elem es
  el elemento buscado; en otro caso encontrado = False}
begin
  encontrado:= False;
  while not EoF(f) and not encontrado do begin
    Read(f, elem);
    if c = elem.clave then
      encontrado:= True
  end {while}
end; {Buscar}

```

### 15.3.2 Búsqueda en archivos ordenados

Al igual que ocurre con la búsqueda secuencial en arrays, el algoritmo puede mejorarse si las componentes del archivo están ordenadas por sus claves. En este caso, se puede evitar recorrer inútilmente la parte final del archivo si hemos detectado una clave mayor que la buscada, ya que entonces tendremos la certeza de que el elemento no está en el archivo. La modificación en el código es muy sencilla, puesto que basta con variar la condición de salida del bucle while para que contemple la ordenación del archivo. El nuevo procedimiento es el siguiente:

```

procedure BuscarOrd(var f: tArchivoElems; c: clave;
                    var elem: tElem; var encontrado: boolean);
{PostC.: si c está en f entonces encontrado = True y elem es
 el elemento buscado; en otro caso encontrado = False}
var
  ultimaClave: clave;

begin
  encontrado:= False;
  ultimaClave:= cota superior de las claves;
  while not EoF(f) and ultimaClave > c do begin
    Read(f, elem);
    if c = elem.clave then
      encontrado:= True;
      ultimaClave:= elem.clave
    end {while}
  end; {BuscarOrd}

```

## 15.4 Mezcla y ordenación de archivos secuenciales

Una de las operaciones fundamentales en el proceso de archivos es el de su ordenación, de forma que las componentes contiguas cumplan una cierta relación de orden.

Cuando los archivos tengan un tamaño pequeño pueden leerse y almacenarse en un array, pudiéndose aplicar las técnicas de ordenación estudiadas para arrays. Estas técnicas son también aplicables a archivos de acceso directo (véase el apartado B.9).

Por último, cuando el único acceso permitido a los archivos es el secuencial, como en los archivos de Pascal, es necesario recurrir a algoritmos específicos para dicha ordenación.

El método más frecuente de ordenación de archivos secuenciales es en realidad una variante no recursiva del algoritmo *Merge Sort* estudiado para arrays en el apartado anterior. Consiste en dividir el archivo origen en dos archivos auxiliares y después mezclarlos ordenadamente sobre el propio archivo origen, obteniendo de esta forma, al menos, pares de valores ordenados. Si ahora se vuelve a dividir el archivo origen y se mezclan otra vez, se obtienen, al menos, cuádruplas de valores ordenados, y así sucesivamente hasta que todo el archivo esté ordenado.

A este proceso que usa dos archivos auxiliares se le denomina de *mezcla simple*. Si se usan más archivos auxiliares, se llama de *mezcla múltiple*, con el que se puede mejorar el tiempo de ejecución.

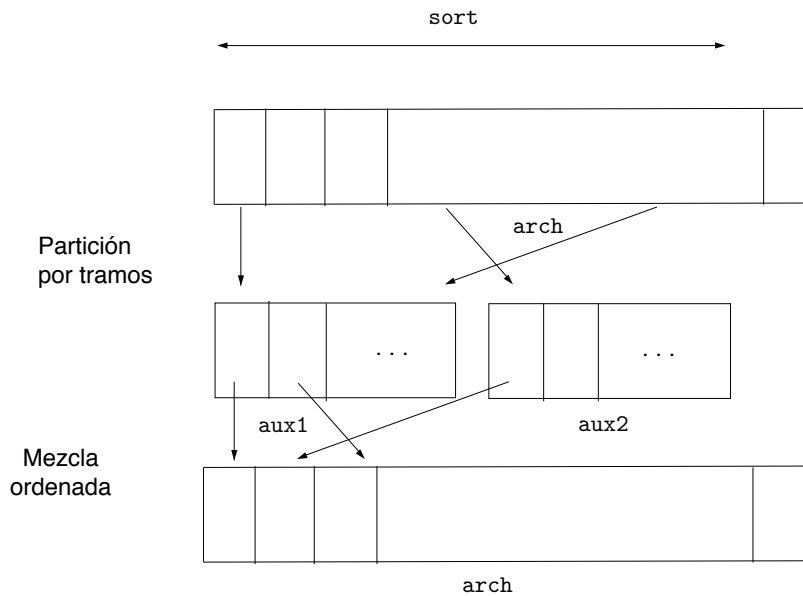


Figura 15.8.

Por lo tanto se tienen dos acciones diferentes: por una parte hay que dividir el archivo original en otros dos, y por otra hay que mezclar ordenadamente los dos archivos auxiliares sobre el archivo original, como se puede ver en la figura 15.8.

Vamos a concentrarnos primero en esta segunda acción de mezcla ordenada, ya que tiene entidad propia dentro del proceso de archivos: en el caso de disponer de dos archivos ordenados, situación que se da con relativa frecuencia, la mezcla ordenada garantiza que el archivo resultante también esté ordenado.

Para realizar la mezcla se ha de acceder a las componentes de los dos archivos, utilizando el operador  $\wedge$ , que aplicaremos a los archivos `aux1` y `aux2`. Si la primera componente de `aux1` es menor que la de `aux2`, se escribe en el archivo `arch` y se accede al siguiente valor de `aux1`. En caso contrario, se escribe en `arch` la primera componente de `aux2` avanzando en este archivo. En el caso de que se llegara, por ejemplo, al final del archivo `aux1`, hay que copiar las restantes componentes de `aux2` en `arch`. Si por el contrario, terminara `aux2`, habremos de copiar las restantes componentes de `aux1` en `arch`.

Dados los valores de `aux1` y `aux2`, la secuencia de lecturas y escrituras sería la que se muestra en la figura 15.9.

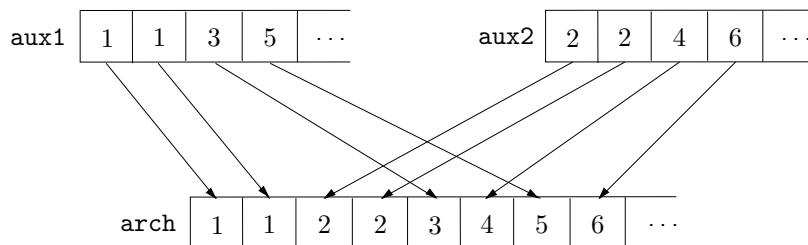


Figura 15.9.

Veamos el diseño descendente de *Mezcla*:

```

mientras no se acabe aux1 y no se acabe aux2 hacer
  si aux1^ < aux2^ entonces
    arch^ := aux1^
    Avanzar aux1
    Poner en arch
  fin {si}
  si aux2^ < aux1^ entonces
    arch^ := aux2^
    Avanzar aux2
    Poner en arch
  fin {si}
fin {mientras}
mientras no se acabe aux1 hacer
  arch^ := aux1^
  Avanzar aux1
  Poner en arch
fin {mientras}
mientras no se acabe aux2 hacer
  arch^ := aux2^
  Avanzar aux2
  Poner en arch
fin {mientras}

```

A continuación se ha escrito el procedimiento *Mezcla* en Pascal:

```

procedure Mezcla(var aux1, aux2, arch: archivo);
  {Efecto: arch := mezcla ordenada de aux1 y aux2}
begin
  Reset(aux1);
  Reset(aux2);
  Rewrite(arch);

```

```

while not EoF(aux1) and not EoF(aux2) do
  if aux1^ < aux2^ then begin
    arch^:= aux1^;
    Put(arch);
    Get(aux1)
  end {if}
  else begin
    arch^:= aux2^;
    Put(arch);
    Get(aux2)
  end; {else}
while not EoF(aux1) do begin
  {Se copia en arch el resto de aux1, si es necesario}
  arch^:= aux1^;
  Put(arch);
  Get(aux1)
end; {while}
while not EoF(aux2) do begin
  {Se copia en arch el resto de aux2, en caso necesario}
  arch^:= aux2^;
  Put(arch);
  Get(aux2)
end; {while}
Close(arch);
Close(aux1);
Close(aux2)
end; {Mezcla}

```

Como puede apreciarse el procedimiento `Mezcla` que se propone utiliza el cursor de archivo, lo que no está permitido en Turbo Pascal (véase el apartado B.9). Para poder utilizar dicho procedimiento en Turbo Pascal hay que modificarlo, utilizando en su lugar el procedimiento `Mezcla` que se detalla seguidamente.

El nuevo procedimiento `Mezcla` dispone, a su vez, de dos procedimientos anidados: el primero, llamado `LeerElemDetectandoFin`, comprueba si se ha alcanzado el final del archivo, y en caso contrario lee una componente del archivo. El segundo, `PasarElemDetectando`, escribe una componente dada en el archivo de destino, y utiliza el procedimiento `LeerElemDetectandoFin` para obtener una nueva componente. A continuación se muestra el procedimiento `Mezcla` modificado:

```

procedure Mezcla(var aux1, aux2, arch: tArchivo);
{Efecto: arch := mezcla ordenada de aux1 y aux2}
var
  c1,c2: tComponente;
  finArch1, finArch2: boolean;

```

```

procedure LeerElemDetectandoFin(var arch: tArchivo;
                                var comp: tComponente; var finArch: boolean);
begin
    finArch:= EoF(arch);
    if not finArch then
        Read(arch, comp)
    end; {LeerElemDetectandoFin}

procedure PasarElemDetectandoFin(var archOrigen, archDestino:
                                tArchivo; var comp: tComponente; var finArchOrigen: boolean);
begin
    Write(archDestino, comp);
    LeerElemDetectandoFin(archOrigen, comp, finArchOrigen)
end; {PasarElemDetectandoFin}

begin {Mezcla}
    Reset(aux1);
    Reset(aux2);
    Rewrite(arch);
    LeerElemDetectandoFin(aux1, c1, finArch1);
    LeerElemDetectandoFin(aux2, c2, finArch2);
    while not finArch1 and not finArch2 do
        if c1 < c2 then
            PasarElemDetectandoFin (aux1, arch, c1, finArch1)
        else
            PasarElemDetectandoFin (aux2, arch, c2, finArch2);
    while not finArch1 do
        PasarElemDetectandoFin (aux1, arch, c1, finArch1);
    while not finArch2 do
        PasarElemDetectandoFin (aux2, arch, c2, finArch2);
    Close(arch);
    Close(aux1);
    Close(aux2)
end; {Mezcla}

```

Abordamos ahora el desarrollo de la otra acción a realizar en este algoritmo, que es la división de `arch` en los dos archivos auxiliares `aux1` y `aux2`. Para optimizar esta tarea conviene tener en cuenta los posibles tramos ordenados que existan ya en `arch` para no desordenarlos. Así, mientras los sucesivos valores que se van leyendo de `arch` estén ordenados, los vamos escribiendo, por ejemplo, en `aux1`. En el momento en que una componente de `arch` esté desordenada pasamos a escribir en `aux2` donde seguiremos escribiendo los sucesivos valores de `arch` que formen otro tramo ordenado. Al aparecer un nuevo valor fuera de orden cambiamos de nuevo a `aux2`.

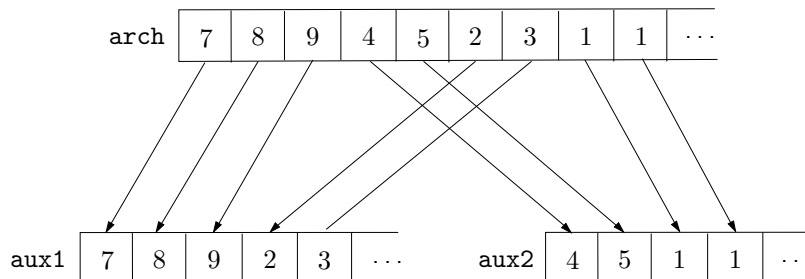


Figura 15.10.

Dados los valores de `arch`, un ejemplo de división sería la que aparece en la figura 15.10.

Este método de división hace que los archivos tengan el mismo número de tramos o a lo sumo que difieran en uno, por lo que su mezcla se denomina *mezcla equilibrada*. Además, se detecta inmediatamente si `arch` está ordenado, ya que existirá un único tramo que se escribirá en `aux1`, quedando `aux2` vacío.

Para saber cuándo cambiar de archivo auxiliar hay que comprobar si el `valorAnterior` leído es mayor que el `valorActual`, lo que obliga a realizar una lectura inicial, sin comparar, para tener un valor asignado a `valorAnterior`, y a almacenar en una variable booleana `cambio` el destino actual. Al escribir al menos una componente en `aux2`, éste ya no está vacío, lo que señalizaremos con una variable booleana `esVacio2`.

Veamos cómo podría ser un primer esbozo de `Division`:

```

si no se ha acabado arch entonces
  Leer valorActual de arch
  Escribir valorActual en aux1
  valorAnterior:= valorActual
fin {si}
mientras no se acabe arch hacer
  Leer valorActual de arch
  si valorAnterior > valorActual entonces
    Cambiar cambio
  si cambio entonces
    Escribir valorActual en aux1
  si no
    Escribir valorActual en aux2
    esVacio2:= False
  fin {si no}
  valorAnterior:= valorActual
fin {mientras}

```

A continuación se ha implementado el procedimiento `Division` en Pascal:

```

procedure Division(var arch, aux1, aux2: tArchivo;
                   var esVacio2: boolean);
  {Efecto: arch se divide en dos archivos aux1 y aux2,
   copiando alternativamente tramos ordenados maximales}

  var
    valorActual, valorAnterior: tComponente;
    cambio: boolean;
    {conmuta la escritura en aux1 y aux2}

begin
  Reset(arch);
  ReWrite (aux1);
  ReWrite (aux2);
  cambio:= True;
  esVacio2:= True;
  if not EoF(arch) then begin
    Read(arch, valorActual);
    Write(aux1, valorActual);
    valorAnterior:= valorActual
  end;
  while not EoF(arch) do begin
    {se buscan y copian los tramos ordenados}
    Read(arch, valorActual);
    if valorAnterior > valorActual then
      cambio:= not cambio;
    if cambio then
      Write(aux1, valorActual)
    else begin
      Write(aux2, valorActual);
      esVacio2:= False
    end;
    valorAnterior:= valorActual
  end; {while}
  Close(arch);
  Close(aux1);
  Close(aux2)
end; {Division}

```

## 15.5 Ejercicios

1. Se desea examinar el funcionamiento del método de búsqueda por bipartición del siguiente modo:

```
Número buscado: 27
 1  4  5 12 25 27 31 42 43 56 73 76 78 80 99
 [                               <                               ]
 [           >           ]
 [           =           ]
El 27 está en la 6ª posición
```

Modifique el programa dado en el apartado 15.1.3 para que dé su salida de esta forma.

2. Para examinar igualmente la evolución de los métodos de ordenación por intercambio, también se pueden insertar los siguientes subprogramas:
  - (a) Uno de escritura que muestre el contenido de un vector (que, supuestamente, cabe en una línea) después de cada intercambio que se produzca.
  - (b) Otro que sitúe las marcas '>' y '<' debajo de las componentes que se intercambian para facilitar su seguimiento visual.

Desarrolle los subprogramas descritos e incorpórelos en un programa que muestre de este modo los intercambios que lleve a cabo.

### 3. Ordenación por el método de la burbuja

Un método de ordenación consiste en examinar todos los pares de elementos contiguos (intercambiándolos si es preciso), efectuando este recorrido hasta que, en una pasada, no sea preciso ningún intercambio. Desarrolle un procedimiento para ordenar un vector siguiendo este algoritmo.

4. Desarrolle un procedimiento para ordenar una matriz de  $m \times n$  considerando que el último elemento de cada fila y el primero de la siguiente son contiguos.
5. Dados dos archivos secuenciales ordenados ascendentemente, escriba un programa que los intercale, produciendo otro con los componentes de ambos ordenados ascendentemente.

### 6. Ordenación por separación

Sea un archivo de texto compuesto por una línea de enteros separados por espacios.

- (a) Escriba un subprograma que construya otro archivo a partir del original aplicando la siguiente operación a cada línea:
  - Si la línea está vacía, se ignora.
  - Si la línea tiene un solo elemento, se copia.
  - De lo contrario, se extrae su primer elemento y se construyen tres líneas en el archivo nuevo: la primera formada por los elementos menores que el primero, en la segunda el primero, y en la tercera los mayores que el primero.

- (b) Escriba un programa que repita el proceso anterior hasta obtener un archivo formado tan sólo por líneas unitarias, copiándolas entonces en una sola línea.
- (c) Establezca las semejanzas que encuentre entre este algoritmo y uno de ordenación de vectores.

### 7. Ordenación por mezcla

Sea un archivo de texto compuesto por una línea de enteros separados por espacios.

- (a) Escriba un subprograma que construye otro archivo, a partir del original, de la siguiente forma:
    - Primero se separan los enteros, situando uno en cada línea.
    - Luego se intercalan las líneas de dos en dos (véase el ejercicio 5). Este paso se repetirá hasta llegar a un archivo con una sola línea.
  - (b) Establezca las semejanzas que encuentre entre este algoritmo y uno de ordenación de vectores.
8. Complete el programa del ejercicio 6 con las siguientes opciones:
- (a) Ordenar Inventario: que ordene el vector donde se almacenan los productos, utilizando alguno de los métodos de ordenación presentados en el capítulo, de las siguientes formas:
    - De forma ascendente por su nombre, y en caso de que el nombre coincida, por su marca, y en caso de que ésta también coincidiera, de forma descendente por su tamaño.
    - De forma descendente por el valor obtenido al multiplicar su precio por el número de unidades inventariadas.
  - (b) Ordenar Archivo: que ordene un archivo dado, donde se almacenan los productos, utilizando el método de División y Mezcla, en las formas señaladas en el apartado primero

## 15.6 Referencias bibliográficas

Con relación al contenido de esta parte del libro, en [Aho88] encontramos un tratamiento a un nivel elevado del tipo compuesto conjunto y de los algoritmos de búsqueda y ordenación; en particular, se recomienda la lectura del capítulo 8 donde se abordan los principales esquemas de clasificación interna, y el capítulo 11 donde se recogen los de clasificación externa.

Otro texto interesante es [Coll87], con una clara orientación a Pascal. Los capítulos de mayor interés son el 4 dedicado a los conjuntos, el 5 dedicado a la estructura vectorial, el 12 sobre ordenación de vectores y el 14 que trata sobre archivos. El texto presenta un enfoque moderno y utiliza especificaciones formales.

En esta bibliografía no podemos dejar de citar el texto clásico [Wir86], y en particular sus dos primeros temas sobre estructuras fundamentales de datos y ordenación de vectores y archivos. Utiliza técnicas de refinamientos progresivos, pero no las especificaciones formales.

Otro de los textos más utilizados sobre estos temas es [Dale89], que se caracteriza por la gran cantidad de ejemplos resueltos. Los capítulos 9 y del 11 al 15 son los que tratan sobre tipos de datos.

Un texto más reciente es [Sal93], que en su parte cuatro refleja las estructuras de datos. Hace una presentación progresiva de los contenidos con ejercicios de complejidad creciente. Faltan por concretar algunos métodos de ordenación de vectores y de archivos secuenciales.

El método de los vectores paralelos (apartado 15.2.6) no es nuevo; por ejemplo, se estudia en [DL89], aunque con el nombre de *punteros de ordenación*.



**Tema V**

**Memoria dinámica**