# 8. ARCHIVOS EN TURBOPASCAL

**Conceptos**: *Archivo, Fichero, Estructura secuencial, Estructura aleatoria, Variable archivo, Archivo con tipo, Archivo de texto, Archivo sin tipo, Apertura, Cierre, Escritura, Lectura, Fin de archivo, Fin de línea.*

**Resumen**: Los datos estructurados vistos hasta ahora se denominan internos porque se almacenan en la memoria principal. Sin embargo, en ocasiones es preciso almacenar los datos de forma permanente en algún sistema de almacenamiento masivo. Los archivos o ficheros son el único tipo de estructura que permiten almacenar datos en estos dispositivos o memoria externa. Otra característica asociada a los ficheros es que su tamaño sólo está limitado por el espacio libre disponible en el dispositivo correspondiente. Se introducen los tipos de ficheros que pueden emplearse en TurboPascal: los archivos con tipo o de acceso aleatorio, los archivos de texto o secuenciales y los archivos sin tipo. La manipulación de estas estructuras se realiza mediante un conjunto de rutinas predefinidas: assign, reset, rewrite, read, write, close… La detección de errores en las operaciones de entrada y salida se efectúa mediante la función predefinida ioresult y de las directivas del compilador a las que se tiene acceso en el programa.

**Objetivos específicos**. Al finalizar el tema, el alumno deberá ser capaz de:

a) Definir el concepto de archivo y describir los diferentes tipos de archivos y sus características fundamentales (*Conocimiento*)
b) Describir las operaciones típicas y las rutinas predefinidas correspondientes que se emplean en TurboPascal para la manipulación de variables archivo (*Comprensión*)
c) Interpretar el resultado de la ejecución de un programa que emplea variables archivo (*Comprensión*)
d) Determinar los formatos adecuados para almacenar los datos en un archivo con tipo o en un archivo de texto en función de las necesidades de una aplicación (*Aplicación*)
e) Codificar una tarea sencilla convenientemente especificada utilizando variables archivo (*Aplicación*)

## 8.1. INTRODUCCIÓN

Como se ha indicado anteriormente una estructura de datos es un conjunto de datos más simples. Las estructuras de datos pueden clasificarse según se realice su almacenamiento en:

a) **internas**, que se almacenan en memoria. Por ejemplo, variables de tipo `set`, `array`, `string` o `record`, y

b) **externas**, que se almacenan en dispositivos de almacenamiento masivo como puede ser el disco duro. La forma habitual de almacenar datos en un disco es mediante un *archivo* o *fichero*.

Un *archivo* o *fichero* es una estructura o colección secuencial de datos de tamaño variable que puede guardarse de forma permanente en un sistema de almacenamiento masivo. El acceso a los datos contenidos en el archivo puede ser **secuencial** (esto quiere decir que los datos se acceden en el orden en el que fueron introducidos) o **aleatorio** o directo (cada dato tiene asociado una posición de almacenamiento).

## 8.2. CARACTERÍSTICAS DE LOS ARCHIVOS DE DISCO

Las principales características de los archivos de disco como estructura de datos son las siguientes:

a) un archivo de disco es un almacén de datos **no volátil**,
b) no tiene un tamaño **predeterminado**: el tamaño de un archivo puede ir variando durante la ejecución de un programa según las necesidades. Por otro lado, el tamaño está sólo limitado por el espacio de almacenamiento existente en disco y,
c) en un archivo todos los datos están almacenados en **formato determinado**, que no tiene porqué coincidir con el formato empleado en otros archivos.

El uso de archivos de disco como almacén de datos tiene las siguientes **ventajas**:

a) la posibilidad de manejar **grandes** cantidades de datos (tanto de entrada como de salida),
b) los datos **no se pierden** al terminar de ejecutar un programa o al apagar el ordenador, es decir, pueden quedar almacenados permanentemente y,
c) su empleo facilita la **transmisión** de datos **entre programas**, aplicaciones, ordenadores o sistemas diferentes. Esta ventaja afecta especialmente a los archivos de tipo `text`.

Aunque su empleo también tiene algunos **inconvenientes**, tanto en la construcción como durante la ejecución de un programa, así:

a) la **manipulación** de un archivo en un programa es algo más **complicada** que el de una estructura de datos interna y,
b) las operaciones de entrada y salida (el acceso para escritura y lectura en los archivos en disco) son relativamente **más lentas**.

## 8.3. VARIABLES ARCHIVO EN TURBOPASCAL

Para poder trabajar y manipular datos con archivos de disco dentro de un programa escrito en TurboPascal es necesario utilizar una variable de un nuevo tipo: el tipo archivo. Este tipo de dato se estructura como una secuencia lineal de componentes. Como ya se detallará más adelante, mediante esta variable, que hace las veces de *intermediario*, se podrán crear o eliminar archivos en el disco, introducir o sacar datos, etc. durante la ejecución de un programa.

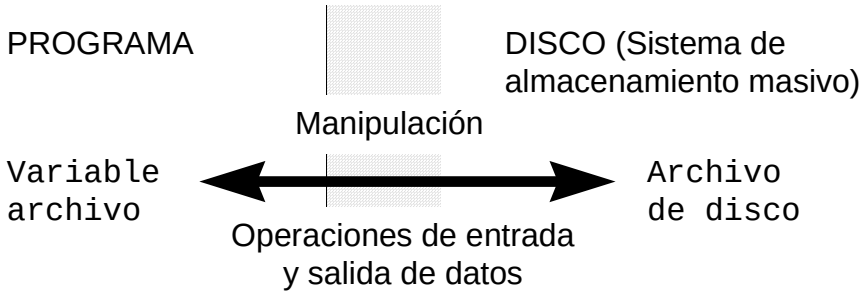


Figura 30. Utilización de variables archivo para la manipulación de archivos de disco

## 8.4. TIPOS DE VARIABLE ARCHIVO

En TurboPascal existen diferentes tipos genéricos de variables archivo con los que se puede trabajar en los programas:

a) **Archivo con tipo** (`file of` ... ): opera con datos de un tipo determinado (`integer`, `real`, `char`, `string`, `array`, `record`,... excepto tipo `file`). Los datos se almacenan en el archivo de disco con el formato binario correspondiente a ese tipo de dato tal y como se almacenan en memoria RAM. Cada dato de un archivo se denomina *registro* (no confundir con datos de tipo `record`). Debido a esto, por ejemplo, cada registro de un `file of integer` necesita una secuencia de 16 bits (2 bytes), mientras que un registro de un `file of real` ocupará 48 bits (6 bytes).

b) **Archivo de texto** (`text`): trabaja con datos de tipo `char`: es decir, con caracteres del formato ASCII. Así, los valores numéricos se convierten en una secuencia de caracteres antes de escribirse en el archivo de disco. Cada registro, en este caso carácter, ocupa 1 byte. Los datos almacenados en este tipo de archivos son fácilmente accesibles desde otras aplicaciones como editores de texto, bases de datos, ...

c) **Archivo sin tipo** (`file`): trabaja con bloques de bytes, desconociéndose (en principio no interesa) qué se almacena o cómo están estructurados los datos en el archivo de disco. Se utilizan para transferir de forma rápida grandes bloques de información.

A continuación, se detallarán las características, estructura y empleo de cada uno de ellos.

## 8.5. VARIABLES ARCHIVO CON TIPO

Las variables *archivo con tipo* permiten trabajar con archivos *de acceso directo*, es decir, cada dato tiene asociado dentro del archivo una posición de almacenamiento mediante la cual puede accederse directamente a él.

### 8.5.1. Declaración de variables archivo con tipo

Como cualquier otro elemento que se emplee en un programa, antes que nada, es necesario declararlas. Previo a la declaración de las variables archivo se recomienda hacer la correspondiente declaración del tipo de dato *archivo con tipo*. La declaración de un tipo archivo se hace de la siguiente manera:

Sintaxis: `type` *`tipo_archivo`* `= file of` *`tipo_de_dato`*`;`

*`tipo_de_dato`* ha de ser o bien un tipo predefinido de TurboPascal o bien un tipo ya definido por el usuario. En este último caso, es necesario declarar previamente el tipo de dato.

Ej.:
```
type  notas_1 = file of integer; {integer, char y real }
      notas_2 = file of char; { son tipos predefinidos }
      notas_3 = file of real;          { de TurboPascal }
      cadena = string[30];     { Definido por el usuario }
      agenda_1 = file of cadena;
      vector = array[1..9] of integer;
      agenda_2 = file of vector;
      ficha  = record
            nombre : string[30];
            numero : integer
            end;
      listin = file of ficha;
```

Y, a continuación, la correspondiente declaración de variables archivo:

Sintaxis: `var` *`ident_1 : tipo;`*

Dadas las declaraciones de tipo anteriores:

Ej.:
```
var a : notas_1;
{ o tambien directamente pero menos recomendable: }
          b : file of integer;
          c : agenda_1;
          d : agenda_2;
          e : listin;
          f : file of ficha;
```

Se recomienda realizar primero la definición del tipo archivo y posteriormente utilizar ésta en la declaración de la variable archivo. Los registros o componentes de un archivo pueden ser de cualquier tipo excepto `file`. Sólo podrán realizarse operaciones de entrada y salida de datos que sean de tipo idéntico o compatible al `tipo_de_dato` declarado en la variable archivo. Es decir, por ejemplo, en la variable archivo `b` declarada anteriormente como `file of integer`, no podrán almacenarse datos de tipo `real` (en general, ningún otro dato que no sea de tipo `integer`).

### 8.5.2. Estructura de una variable archivo con tipo

La estructura de cualquier variable archivo utilizada en un programa sigue de forma general el esquema de la Figura 31:

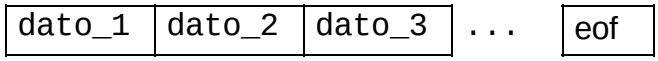


Figura 31. Estructura de una variable archivo con tipo

Un *archivo con tipo* es una secuencia de datos de un determinado tipo almacenados uno a continuación de otro. Tras el último registro del archivo se encuentra una señal, `eof` (**e**nd **o**f **f**ile), que es la *marca* o *carácter* de fin de archivo. Esta marca es un carácter especial que no puede ser leído y que indica el final del archivo. El acceso, tanto de entrada como de salida, a los datos o registros del archivo se permite a través de un *puntero* o *ventana* asociado a cada variable archivo. Este elemento indica la posición en la que se realiza el siguiente acceso de lectura o de escritura en el archivo en disco correspondiente.

### 8.5.3. Rutinas estándar para manejar variables archivo con tipo

En general, las variables de tipo archivo (`file of` ..., `file` o `text`) tienen una manera característica de ser manipuladas durante la ejecución de un programa, que las distinguen del resto de variables de otros tipos. Las variables archivo no pueden aparecer **directamente** ni en asignaciones ni en expresiones, pero pueden ser parámetros por variable de funciones y procedimientos. Así, aunque se declaren dos variables archivo del mismo tipo `var f,g: file of integer;` no es correcta la asignación `f:=g;`

En TurboPascal existe un grupo de rutinas declaradas en la unidad estándar `System` que permiten manipular variables de tipo archivo. A continuación, se comentan algunas de estas rutinas incluidas en la unidad estándar `System` para poder trabajar con variables *archivo con tipo*. Algunas de ellas sirven para otros tipos de variable archivo.

#### 8.5.3.1 ASSIGN

Cabecera[13]: `assign(var f:<tipoarchivo>;archivodisco:string);`

El procedimiento `assign` asigna o asocia el nombre del archivo externo `archivodisco` al identificador de la variable archivo `f`. Para poder trabajar con una variable archivo es **necesario** en primer lugar asociar a la variable archivo TurboPascal correspondiente un archivo del disco. Esto permite la manipulación del archivo externo con el que se trabaja. El nombre del archivo externo debe ser admisible por el sistema operativo DOS y no tiene porqué coincidir con el de un archivo ya existente en disco. En caso necesario puede indicarse la unidad (`A:`, `B:`, `C:`) y el camino o vía de acceso (`\TURBO\`, `\WP\TEXTOS\`,...). La operación que se haga con la variable archivo del programa, en la práctica se realizará en el archivo externo asociado.

Ej. 1.:
```
const archivo = 'archivo.dat';
type enteros = file of integer;
var f : enteros;
begin
assign(f,archivo);
```

Ej. 2.:
```
type caracteres = file of char;
var g : caractetes;
    archivo:string[30];
begin
writeln('Escribe el nombre del archivo:');
readln(archivo);
assign(g,archivo);
```

Ej. 3.:
```
type enteros = file of integer;
var f : enteros;
begin
assign(f,'A:\PASCAL\datos1.dat');
```

#### 8.5.3.2 REWRITE

Cabecera: `rewrite(var f:<tipoarchivo>);`

El procedimiento `rewrite` es el *procedimiento de apertura* de un archivo nuevo o no existente en disco. La ejecución de la llamada a este procedimiento **crea** un nuevo archivo en el disco con el nombre previamente asignado mediante el procedimiento `assign` a la variable fichero. Hay que llevar cuidado con su empleo, ya que si existía un archivo con ese nombre se borrarán los datos contenidos en él. Una vez creado, lleva el puntero asociado a la variable archivo al inicio para comenzar operaciones de escritura. Por lo tanto, el archivo abierto con este procedimiento no posee ningún registro inicialmente.

Ej.:
```
type enteros = file of integer;
var f : enteros;
begin
assign(f,'miarchi.dat');
rewrite(f);
```

#### 8.5.3.3 RESET

Cabecera: `reset(var f:<tipoarchivo>);`

[13] Como ya se ha visto la estructura de los procedimientos y funciones, se ha considerado de mayor utilidad mostrar la forma cómo se ha declarado la cabecera de las rutinas relacionadas con el uso de los archivos

El procedimiento `reset` es el *procedimiento de apertura* de un archivo que ya existe en disco. La ejecución de la llamada a este procedimiento lleva el puntero al inicio o principio del archivo (registro nº 0) para posteriores operaciones de escritura (entrada de datos) o de lectura (salida de datos).

Ej.:
```
type enteros = file of integer;
var f : enteros;
begin
assign(f,'miarchi.dat');
reset(f);
```

El archivo externo asociado a la variable fichero debe existir en el disco o se producirá un error durante la ejecución de esta sentencia en el programa.

#### 8.5.3.4 WRITE

Cabecera: `write(var f:<tipoarchivo>,var_1,var_2,...,var_n);`

El procedimiento `write` es el *procedimiento de entrada* o *escritura de datos* en un archivo ya existente en disco y previamente abierto. La ejecución de la llamada a este procedimiento con las variables archivo es parecida a lo ya visto anteriormente con la pantalla o dispositivo estándar de salida de datos. En este caso, al ejecutarse el procedimiento, se escribe cada uno de los datos almacenados en las variables correspondientes en el archivo en disco asociado, moviendo a continuación el puntero asociado a la variable archivo a la siguiente posición de registro.

La variable archivo debe haber sido abierta previamente en el programa. Las variables que se indiquen en la llamada al procedimiento han de ser idénticas al tipo de dato declarado para la variable archivo. El identificador de la variable archivo siempre debe colocarse después del primer paréntesis, en caso contrario, la escritura de datos se realiza por el *archivo de salida estándar* asociado por defecto: la pantalla del ordenador.

Ej.:
```
type enteros = file of integer;
var   a,b,c : integer;
      f : enteros;
begin
assign(f,'archivo.dat');
rewrite(f);
a:=16;
b:=-1;
c:=0;
write(f,a,b,c);
{ resto de sentencias del programa . . . }
end.
```

Al ejecutar la sentencia `write(f,a,b,c);` se almacenan tres registros o datos de tipo `integer` en el archivo de disco asociado y el puntero asociado a la variable archivo se mueve a la posición siguiente según indica la Figura 32:

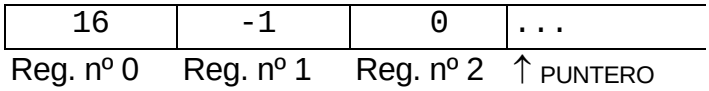


Figura 32. Contenido del archivo y situación del puntero asociado

Como ya se ha mencionado anteriormente los datos son almacenados en el archivo de la misma manera que se almacenan en memoria, es decir, en el formato binario correspondiente al tipo del dato del registro. En este caso, al ser los datos de tipo `integer`, cada registro ocupará 2 bytes. Si pudiera verse el contenido bit a bit del archivo en disco asociado se vería lo que muestra la Figura 33 (la razón de incluir esta figura es la de mostrar la diferencia en cuanto a

formato de almacenamiento entre este tipo de archivo y el archivo de texto que se verá posteriormente).

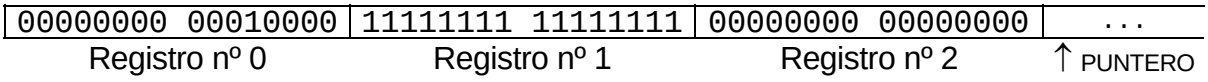


Figura 33. Contenido del archivo en representación binaria y situación del puntero asociado una vez realizada la operación de escritura

#### 8.5.3.5 EOF

Cabecera: `eof(var f:<tipoarchivo>): boolean;`

La función `eof` devuelve el valor lógico `true` o `false` dependiendo si el puntero asociado a la variable archivo `f` está apuntando a la marca de fin de archivo (también llamada eof) o a cualquier otro registro del archivo, respectivamente. Por ejemplo, inmediatamente después de ejecutarse la llamada al procedimiento `rewrite(f)`, una llamada a la función `eof(f)` devolvería siempre el valor `True`, ya que el archivo está vacío. Después de la ejecución `reset(f)`, la función `eof(f)` devolvería el valor `True` si el archivo estuviera vacío y `False` en caso contrario.

Ej.:
```
type enteros = file of integer;
var f : enteros;
begin
assign(f,'miarchi.dat');
reset(f);           { miarchi.dat debe existir en disco }
if eof(f) then writeln('Archivo vacio')
else writeln('Archivo no vacio')
```

#### 8.5.3.6 READ

Cabecera: `read(var f:<tipoarchivo>,var idvar1,...,var idvar_n);`

El procedimiento `read` es el *procedimiento de salida* o *lectura de datos* de un archivo. La ejecución de la llamada al procedimiento `read` lee uno o más datos del archivo en disco asociado y lo/s almacena en la/s variable/s correspondientes. Estas variables han de ser de un tipo idéntico al declarado para la variable archivo con tipo. Cada vez que se lee un registro del archivo se lleva al puntero a la siguiente posición.

Ej.:
```
type enteros = file of integer;
var a,b,c   : integer;
    f: enteros;
begin
assign(f,'archivo.dat');
reset(f);       { Archivo.dat debe existir en disco }
read(f,a,b,c);
```

Con el archivo en disco creado en el ejemplo anterior se almacenarían en las variables de tipo `integer a`, `b` y `c` los valores enteros 16, –1 y 0 respectivamente.

La ejecución de este procedimiento produce un error si se intenta leer cuando el puntero señala el carácter o marca de fin de archivo, eof, de un archivo. Una manera muy práctica y habitual de leer secuencialmente todos los datos de un archivo sería mediante el siguiente bucle `while`:

```
type enteros=file of integer;
var a,b,c   : integer; f: enteros;
begin
assign(f,'miarchi.dat');
reset(f);
while not eof(f) do read(f,a);
close(f);
writeln('El ultimo dato de miarchi.dat es: ',a);
end.
```

#### 8.5.3.7 CLOSE

Cabecera: `close(var f:<tipoarchivo>);`

El procedimiento `close` es el *procedimiento de cierre* de un archivo previamente abierto. Es recomendable utilizar este procedimiento al dejar de trabajar con el archivo en disco. La ejecución de este procedimiento transfiere los posibles datos almacenados en el *buffer* o almacén intermedio al archivo del disco y lo cierra liberando al sistema operativo del control de archivo. Si no cerramos el archivo los datos que quedan en el *buffer* o almacén temporal intermedio de la memoria podrían perderse sin quedar almacenados en el archivo.

Ej.:
```
type enteros = file of integer;
var a,i : integer;
    f : enteros;
begin
assign(f,'miarchi.dat');
rewrite(f);
{ Algunas operaciones de escritura de datos }
for i:=1 to 10 do write(f,i);
close(f);
reset(f);
{ Operaciones de escritura }
while not eof(f) do read(f,a);
close(f);
writeln('El ultimo dato de miarchi.dat es: ',a);
end.
```

Aunque se cierre la variable archivo, el archivo de disco correspondiente sigue asociado a la variable archivo, después de la ejecución de las llamadas al procedimiento `close(f)`.

#### 8.5.3.8 RENAME

Cabecera: `rename(var f:<tipoarchivo>;nombrearchivo:string);`

El procedimiento `rename` permite cambiar el nombre del archivo de disco asociado a una variable archivo. El contenido del archivo de disco no sufre ninguna modificación. La llamada al procedimiento de cambio de nombre puede hacerse con una variable archivo abierta, no abierta todavía o cerrada.

Ej.:
```
type reales = file of real;
var f : reales;
begin
assign(f,'antiguo.dat');
rename(f,'nuevo.dat');
```

#### 8.5.3.9 ERASE

Cabecera: `erase(var f:<tipoarchivo>);`

El procedimiento `erase` permite borrar el archivo del disco asignado a la variable archivo. La variable archivo archivo debe estar cerrada.

Ej.:
```
type reales = file of real;
var f : reales;
begin
assign(f,'miarchi.dat');
erase(f);
```

#### 8.5.3.10 SEEK

Cabecera: `seek(var f:<tipoarchivo>;posicion:longint);`

Este procedimiento mueve el puntero asociado a la variable archivo previamente abierta a la posición indicada por el segundo parámetro entero. Se considera que la posición del primer registro del archivo es la 0. Por ejemplo, en el caso del archivo de disco de la Figura 34.

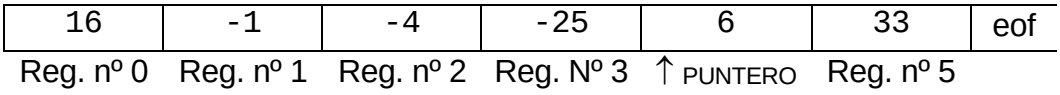


Figura 34. Contenido del archivo y situación del puntero asociado a la variable archivo f

La ejecución de las sentencias

```
type enteros = file of integer;
var f : enteros;
begin
assign(f,'miarchi.dat');
reset(f);
seek(f,2);
```

dejaría al puntero asociado señalando el tercer registro del archivo, es decir, el valor entero `-4`, según indica la Figura 35.

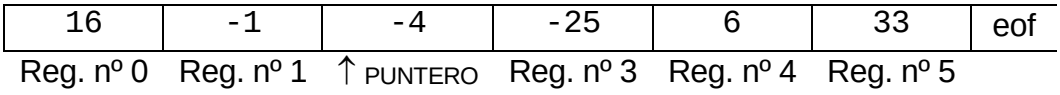


Figura 35. Contenido del archivo de disco y situación del puntero asociado a la variable archivo f una vez ejecutada la llamada a la sentencia `seek(f,2)`

A esta operación es a la que se conoce como *acceso directo* a un registro de un archivo. Aunque puede utilizarse la sentencia `seek(f,0)`, lo habitual para situar el puntero señalando al primer registro de un archivo es usar el procedimiento `reset(f)`, aunque la variable archivo estuviera previamente abierta.

### 8.5.3.11 FILEPOS

Cabecera: `filepos(var f:<tipoarchivo>):longint;`

La llamada a la función `filepos` devuelve la posición actual del puntero asociado a una variable archivo. Por ejemplo, inmediatamente después de ejecutar la llamada al procedimiento `reset(f)`, la función `filepos(f)` devolvería el valor entero 0.

Ej.:
```
type enteros=file of integer;
var f: enteros;
begin
assign(f,'miarchi.dat');
reset(f);
write(f,a);
{ N° indeterminado de operaciones de escritura . . .}
write(filepos(f));
```

### 8.5.3.12 FILESIZE

Cabecera: `filesize(var f:<tipoarchivo>):longint;`

La función `filesize` devuelve el número de componentes o registros almacenados en un archivo. El valor devuelto es de tipo entero. Si el archivo está vacío devuelve un cero. Así, después de la llamada al procedimiento `rewrite(f)`, la función `filesize(f)` devuelve el valor 0. Por ejemplo, si a la variable archivo `f` de tipo `file of integer` se le asocia el archivo de disco de la Figura 36:

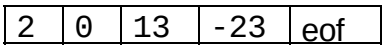


Figura 36. Contenido del archivo de disco.

la sentencia `Write(FileSize(f))` escribiría el valor 4 por pantalla. Como otro ejemplo de uso, para añadir datos a un archivo se mueve el puntero al final de archivo...

```
seek(f,filesize(f));
```

que deja al puntero señalando la marca de fin de archivo.

### 8.5.4. Programas ejemplo con archivos con tipo

A continuación, se muestra el código de dos programas ejemplo que trabajan con un archivo con tipo que almacena datos de tipo `integer`:

```
{ Programa que introduce varios datos enteros
en un archivo de disco }
program intdatos;
type enteros = file of integer;
var dato,k,total:integer;
    f:enteros;
begin
assign(f,'valores.dat');
rewrite(f);
write('¿Cuantos valores? ');
readln(total);
for k:=1 to total do
      begin
      write('Dato n° ',k,': ');
      readln(dato);
      write(f,dato)
      end;
close(f)
end.
```

Ejemplo de ejecución del programa `intdatos`:

```
c:\tp\archivos>intdatos↵
¿Cuantos valores? 5↵
Dato n° 1: 12↵
Dato n° 2: 4↵
Dato n° 3: 7↵
Dato n° 4: 0↵
Dato n° 5: 5↵
```

Una vez ejecutado, crea el archivo `valores.dat`:

| 12 | 4 | 7 | 0 | 5 | eof |
|---|---|---|---|---|---|

Figura 37. Contenido del archivo de disco valores.dat

En el siguiente programa ejemplo, se trabaja con el archivo de disco generado con el programa anterior, leyendo los datos almacenados y visualizando por pantalla el número de caracteres correspondiente al valor entero leído.

```
program grafica;
type archivo = file of integer;
var  valor, i : integer;
      f : archivo;
begin
assign(f,'valores.dat');
reset(f);
i:=1;
while not eof(f) do
      begin
      read(f,valor);
      write('Valor n° ',i,': ');
      while valor>0 do
            begin
            write('*');
            dec(valor)
            end;
      writeln;
```

```
      inc(i)
      end;
close(f)
end.
```

La salida por pantalla del ejemplo de ejecución del programa `grafica` supuesto que el contenido del archivo `'valores.dat'` es el mostrado en la Figura 37:

```
c:\tp\archivos>grafica↵
Valor nº 1: ***********
Valor nº 2: ****
Valor nº 3: *******
Valor nº 4:
Valor nº 5: *****
```

## 8.6. VARIABLES ARCHIVO DE TEXTO

El tipo de dato `text` es un tipo predefinido en TurboPascal. Con variables de este tipo pueden manipularse, es decir, realizar operaciones de lectura y escritura con datos de tipo entero, real, `boolean` (sólo escritura), `char` y `string` que se almacenan en el archivo de disco correspondiente en el formato de caracteres del código ASCII.

Por poseer un formato estándar son los más adecuados para intercambiar información con otros programas y sistemas: un archivo ASCII puede crearse y modificarse con casi cualquier otro editor o procesador de texto. Por ejemplo, los editores del sistema operativo DOS `EDLIN` y `EDIT`, el editor Block de Notas del sistema Windows y el propio editor del entorno integrado de TurboPascal. También pueden manipularse con otros comandos del sistema operativo. Por ejemplo, puede visualizarse su contenido en pantalla de un archivo ASCII con el comando `TYPE` del sistema operativo DOS o enviarse a la salida por impresora con el comando `PRINT`.

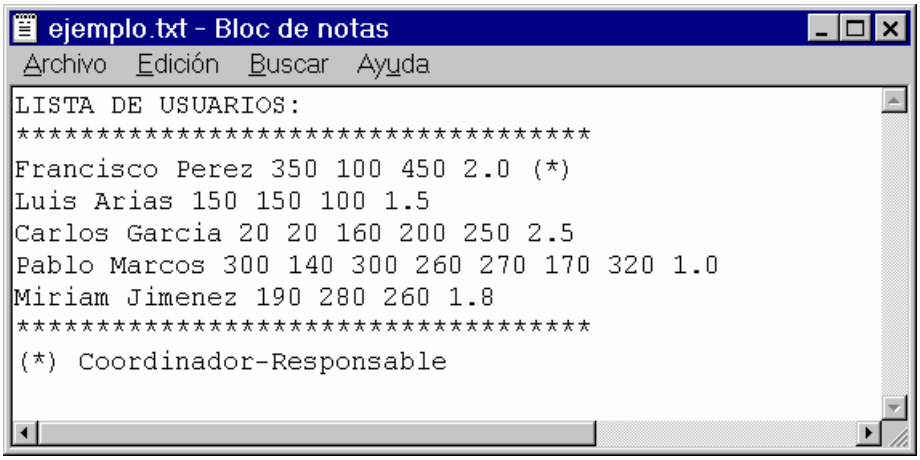


Figura 38. Visualización de un archivo ASCII con el Block de Notas de Windows

### 8.6.1. Declaración de variables archivo de texto

Al ser el tipo `text` un tipo predefinido en TurboPascal puede utilizarse directamente en la declaración de variables:

Sintaxis: `var` *archivo*`:text;`

Ej.: `var f,g:text;`

### 8.6.2. Estructura de las variables archivo de texto

Las variables archivo de tipo `text` están estructuradas por líneas que contienen caracteres, cadenas o frases separadas por una marca que indica el fin de la línea: `eoln` (**e**nd **o**f **lin**e). Para TurboPascal (en realidad, para el sistema operativo) `eoln` es una secuencia de dos caracteres ASCII: el carácter de retorno de carro o CR (carácter nº 13 del código ASCII) y el carácter de avance de línea o LF (ASCII nº 10). El final de un archivo de texto se indica como con los archivos con tipo: con una marca de fin de archivo `eof`. Al emplear una variable archivo de tipo `text`, el archivo de disco a manipular quedará estructurado, por ejemplo, siguiendo el esquema de la Figura 39:

| '1' | '3' | '.' | '4' | *#13* | *#10* | ' ' | ' ' | 'O' | 'k' | *#13* | *#10* | '$' | '$' | *#27* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | *eoln* | | | | | | *eoln* | | | | *eof* |

Figura 39. Ejemplo de contenido y estructura de una variable archivo de texto

En realidad, ya se han utilizado dos dispositivos que tienen una estructura similar a los archivos de tipo `text`: el teclado, denominado `input` en Pascal estándar, que sirve sólo para lectura (entrada de datos), y la pantalla, denominada `output`, que sólo es de escritura (salida de datos).

En Pascal estándar estos archivos hay que declararlos obligatoriamente en la cabecera del programa, no así en TurboPascal. En estas dos *variables archivo* la marca de fin de línea se obtiene pulsando la tecla INTRO, cerrándose con una marca de final de archivo: `eof` (**e**nd **o**f **f**ile) que se obtiene pulsando la combinación de teclas CTRL + Z.

### 8.6.3. Rutinas estándar para manipular variables de tipo TEXT

Para trabajar con variables de tipo `text` pueden emplearse todas las subrutinas comentadas anteriormente para los *archivos con tipo*, excepto `Seek`, `FilePos` y `FileSize`. Como características diferenciadoras, siendo `f` una variable de tipo `text`, la ejecución del procedimiento `rewrite(f)` lo convierte en archivo de **sólo escritura**, mientras que la ejecución de `reset(f)` lo convierte en archivo de **sólo lectura**. Además, con las variables de tipo `text` pueden utilizarse otras subrutinas que se detallan a continuación:

#### 8.6.3.1 WRITELN

Cabecera: `writeln(`*`var f:text;expresion_1,...,expresion_n`*`);`

El procedimiento `writeln` escribe en el archivo `f` los valores de los datos (constantes o variables) y, a continuación, una marca de fin de línea. Esta última operación adicional diferencia al procedimiento `writeln` del otro procedimiento análogo de escritura de datos en una variable archivo de tipo `text` (`write`).

Ej.:
```
var a,b,c:integer;
    f:text;
begin
assign(f,'miarchi.dat');
rewrite(f);
a:=16;
b:=-1;
c:=0;
writeln(f,a,b,c);
```

Los datos de tipos numéricos se convierten en caracteres del código ASCII y quedan almacenados como tales en los archivos de disco asociados al emplear los procedimientos

`write` o `writeln`. Con la última sentencia del ejemplo anterior se escribiría en el archivo de disco `miarchi.dat`... como muestra la Figura 8.11.

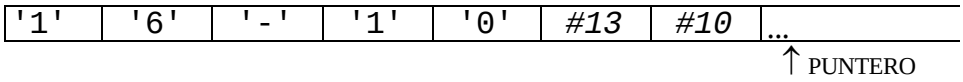


Figura 40. Contenido del archivo de disco miarchi.dat.

Estos datos almacenados como caracteres del código ASCII, se codificarían en formato binario de la siguiente manera:

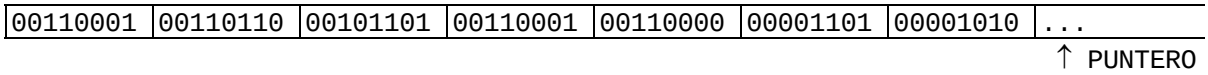


Figura 41. Contenido en formato binario del archivo de disco miarchi.dat.

Es importante observar la diferencia en cómo se almacenan datos de tipo `integer` en un archivo de disco mediante una variable tipo `text`, con respecto a cómo lo hacen mediante una variable archivo de tipo `file of integer`. El primero los almacena en formato de caracteres ASCII y el segundo almacena la representación binaria del número entero (obsérvese la Figura 33).

En variables de tipo `text`, los procedimientos `write` y `writeln` pueden escribir en un archivo de disco, datos de tipo `char`, `string`, entero, real y `boolean` de manera similar a cómo lo hacen en la pantalla, es decir utilizando un determinado número de columnas (caracteres en blanco en un archivo de texto), por ejemplo:

```
var r:real; i:integer; c:char; f:text;
begin
assign(f,'miarchi.txt');
rewrite(f);
r:=8.123;
i:=-2;
c:='*';
writeln(f,r:4:1,i:5,c:3);
```

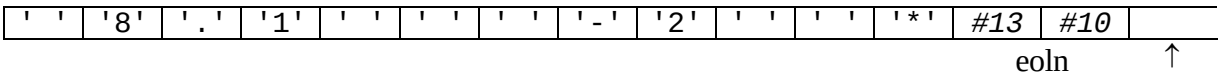


Figura 42. Contenido del archivo de texto resultante de la ejecución de las sentencias anteriores.

Si en lugar del identificador de la variable archivo `f`, se coloca el identificador `lst`, la salida de datos se realiza a la impresora.

Ej.: `writeln(lst,a);`

Nota: el identificador `lst` está declarado como una variable archivo de tipo `text` en la unidad `Printer` estándar de TurboPascal.

### 8.6.3.2 EOLN

Cabecera: `eoln(`*`var archivo:text`*`):boolean;`

La función `eoln` devuelve el valor `true` o `false` si el puntero apunta o no a la posición de la marca de fin de línea eoln (en concreto, sólo si apunta al primer carácter que la forma) o de la marca de fin de archivo `eof`.

Ej.:
```
var f:text;
    b:boolean;
begin
assign(f,'miarchi.txt');
reset(f);
if eoln(f) then ...
```

En la figura siguiente se indica el valor que devuelve esta función si el puntero está apuntando al carácter situado en la parte superior:

| '1' | '6' | '-' | '1' | '0' | *#13* | *#10* | … |
|---|---|---|---|---|---|---|---|
| false | false | false | false | false | **true** | **false** | |

Figura 43. Contenido del archivo miarchi.txt y resultado (abajo) de la llamada a la función `eoln` según la posición del puntero

### 8.6.3.3 READLN

Cabecera: `readln(var f:text;var idvar_1,...,var idvar_n);`

El procedimiento `readln` lee los datos a los que apunta el puntero de la variable archivo `text`, los almacena en la/s variable/s correspondiente/s y, finalmente, mueve el puntero hasta el primer carácter después de la siguiente marca de fin de línea que se encuentre. Esta última operación adicional diferencia al procedimiento `readln` del otro procedimiento análogo de lectura de datos en una variable archivo de tipo `text` (`read`).

Ej.:
```
var  d:string[10];
     f:text;
begin
assign(f,'miarchi.txt');
reset(f);
readln(f,d);
```

Con el puntero de la variable archivo `f` del ejemplo anterior apuntando al primer carácter `'1'` se almacenaría la cadena formada por los caracteres `'16-10'` en la variable `d` y el puntero quedaría apuntando al primer carácter a continuación de la marca de fin de línea.

Las variables donde se almacena la información leída pueden ser de tipo `char`, cadena, entera o real. Si la variable donde se almacena el dato leído es de tipo numérico, los procedimiento `readln` realiza una conversión de datos (de ASCII al tipo numérico correspondiente) antes de almacenarlo en la variable.

### 8.6.3.4 Nota importante sobre los procedimientos `read` y `readln`

Al utilizar los procedimientos de salida de datos `read` y `readln` con variables de tipo `text`, es importante tener siempre muy en cuenta el tipo de variable en la que vamos a almacenar el dato que se acaba de leer y el/los posibles caracteres que se pueden encontrar en los archivos de texto. Pueden aparecer errores durante la ejecución de un programa si se intentan algunos tipos de asignaciones incompatibles.

Por ejemplo, surgiría un error durante la ejecución de un programa si se intenta asignar a una variable de tipo numérico una secuencia de caracteres no numéricos.

Para intentar dar una idea de los distintos comportamientos que puede tener el procedimiento `read`, se ha construido la Tabla 21, que muestra, en función del **tipo de la variable** `a` y de la **posición del puntero** cuando se realiza la llamada al procedimiento `read(f,a)`, el valor que toma la variable `a` y la posición a la que apunta posteriormente el puntero.

```
program ejemplo;
var   a : ? ;       { Declaracion de la variable a }
      f : text;     { Declaracion de la variable archivo }
begin
assign(f,'archivo.dat');
reset(f);
{ operaciones diversas ... de lectura y escritura }
{ Lectura de un dato de la variable archivo }
read(f,a);
```

donde `archivo.dat` es un archivo de disco que almacena la siguiente sucesión de caracteres mostrada en la Figura 44:

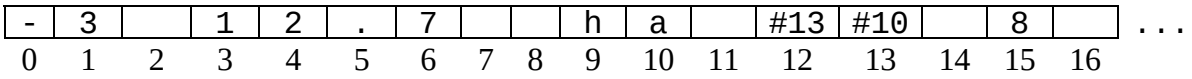


Figura 44. Contenido del archivo de texto archivo.dat.

Otra aclaración importante es la siguiente: Los procedimientos `read` y `readln` no producen un error de ejecución cuando se intenta leer una marca de fin de fichero (`eof`) en variables de tipo TEXT.

Ejemplo de lectura de la Tabla 21:

Si el puntero señala al carácter nº 6 y la variable `a` es de tipo `string[4]` entonces tras la sentencia `read(f,a);`, la variable `a` almacena la cadena `'7 h'` y el puntero queda señalando el carácter nº 10.

| Si el puntero señala al caracter nº | ... y la variable `a` es de tipo... | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | `char` | | `string[1]` | | `string[4]` | | `integer` | | `real` | |
| | ... `a` almacena el valor... | | | | y el puntero queda señalando el caracter nº... | | | | | |
| 0 | `'-'` | 1 | `'-'` | 1 | `'-3 1'` | 4 | `-3` | 2 | `-3.0` | 2 |
| 1 | `'3'` | 2 | `'3'` | 2 | `'3 12'` | 5 | `3` | 2 | `3.0` | 2 |
| 2 | `' '` | 3 | `' '` | 3 | `' 12.'` | 6 | `Error` | | `12.7` | 7 |
| 3 | `'1'` | 4 | `'1'` | 4 | `'12.7'` | 7 | `Error` | | `12.7` | 7 |
| 4 | `'2'` | 5 | `'2'` | 5 | `'2.7 '` | 8 | `Error` | | `2.7` | 7 |
| 5 | `'.'` | 6 | `'.'` | 6 | `'.7 '` | 9 | `Error` | | `0.7` | 7 |
| 6 | `'7'` | 7 | `'7'` | 7 | `'7 h'` | 10 | `7` | 7 | `7.0` | 7 |
| 7 | `' '` | 8 | `' '` | 8 | `' ha'` | 11 | `Error` | | `Error` | |
| 8 | `' '` | 9 | `' '` | 9 | `' ha '` | 12 | `Error` | | `Error` | |
| 9 | `'h'` | 10 | `'h'` | 10 | `'ha '` | 12 | `Error` | | `Error` | |
| 10 | `'a'` | 11 | `'a'` | 11 | `'a '` | 12 | `Error` | | `Error` | |
| 11 | `' '` | 12 | `' '` | 12 | `' '` | 12 | `8` | 16 | `8.0` | 16 |
| 12 | `#13` | 13 | `''` | 12 | `''` | 12 | `8` | 16 | `8.0` | 16 |
| 13 | `#10` | 14 | `#10` | 14 | `#10' 8 '` | 16 | `8` | 16 | `8.0` | 16 |
| 14 | `' '` | 15 | `' '` | 15 | `' 8 ?'` | 17 | `8` | 16 | `8.0` | 16 |
| 15 | `'8'` | 16 | `'8'` | 16 | `'8 ??'` | 18 | `8` | 16 | `8.0` | 16 |

Tabla 21. Resultados de la ejecución de la sentencia `read(f,a);` en función de la posición de la posición del puntero asociado a la variable archivo y del tipo de variable donde se almacena la información leída

#### 8.6.3.5 Observaciones sobre la lectura mediante variables `text`

Hay una serie de puntos que es necesario tener en cuenta para evitar problemas y errores de ejecución cuando se realizan operaciones de lectura en variables de tipo `text`, y posteriormente, el dato leído se almacena en una variable de un tipo determinado.

a) Con una variable de tipo `char` la operación nunca da problemas: almacena siempre el carácter leído y deja el puntero apuntando a la siguiente posición. Esto ocurre incluso con los caracteres de control (caracteres ASCII nº 13, nº 10, etcétera) siendo la única

excepción la marca de fin de archivo, cuyo intento de lectura provoca un error en la sentencia `readln` durante la ejecución de un programa.

b) Con una variable de tipo `string` almacena como máximo un número de caracteres igual a su tamaño declarado. La cadena leída y posteriormente almacenada se trunca al encontrar una marca de fin de línea o de archivo. La ejecución del procedimiento no almacena ningún dato si el puntero apuntaba al carácter de retorno de carro (carácter ASCII nº 13), dejando al puntero en la misma posición.

c) Con una variable de tipo entero el puntero puede saltar caracteres en blanco (carácter ASCII nº 32), tabuladores (carácter ASCII nº 9) y caracteres de fin de línea (secuencia de caracteres ASCII nº 13 y nº 10) hasta encontrar una secuencia de caracteres numéricos que sea entera. A esta secuencia debe seguir inmediatamente un carácter en blanco, un tabulador o una marca de fin de línea o fin de archivo. En caso contrario, se produce un error de ejecución.

d) Con una variable de tipo real funciona de manera similar al anterior, permitiendo además, que la secuencia de caracteres sea un valor real (dígitos separados por un punto).

Se recuerda que los procedimientos `read` (con las variables de tipo `text`) y `readln` admiten variables de tipo entero, real, `char` y cadena. El procedimiento `readln` difiere del anterior en que una vez hecha la salida de datos deja al puntero señalando al primer carácter a continuación de la siguiente marca de fin de línea que se encuentra. Se propone como ejercicio realizar la tabla equivalente a la anterior resultante de la llamada al procedimiento `readln(f,a)`

#### 8.6.3.6 APPEND

Cabecera: `append(var f:text);`

El procedimiento `append` abre una variable `text` asociada a un archivo en disco existente para añadir datos al final del mismo. Deja el puntero asociado, apuntando a la marca de fin de archivo, permitiendo posteriores operaciones de escritura en el archivo de disco.

Ej.:
```
var f:text;s:string;
begin
assign(f,'archivo.dat');
append(f);
s:='Ultima linea';
writeln(f,s);
```

#### 8.6.3.7 SEEKEOLN

Cabecera: `seekeoln(var f:text):boolean;`

La función `seekeoln(f)` es similar a la función `eoln(f)` excepto que salta espacios en blanco y tabuladores **antes** de hacer la verificación. El puntero no se mueve de su posición al realizar la verificación. En la Figura 45 se indica el valor que devuelve esta función si el puntero está apuntando al carácter superior:

| ' ' | '6' | '1' | ' ' | ' ' | #13 | #10 | … |
|---|---|---|---|---|---|---|---|
| false | false | false | **true** | **true** | **true** | false | |

Figura 45. Contenido de un archivo de texto y valor devuelto por la función `eoln` en el caso de que el puntero estuviera apuntando al carácter superior.

#### 8.6.3.8 SEEKEOF

Cabecera: `seekeof(var f:text):boolean;`

La función `seekeof(f)` es similar a `eof(f)` aunque salta blancos, tabuladores y marcas de fin de línea antes de hacer la verificación de que el puntero asociado a la variable archivo está apuntando a la marca de fin de archivo. El puntero no se mueve de su posición al realizar la verificación.

### 8.6.4. Diferencias entre una variable archivo de tipo `file of char` y otra variable archivo de tipo `text`

Trabajar con una variable de tipo `file of char` en un programa no es lo mismo que hacerlo con una de tipo `text`. Aquella es una variable archivo de acceso directo, tiene subrutinas de manipulación específicas y no se considera estructurada por líneas (aunque pueda tratar las marcas de fin de línea sin ningún tipo de problemas, como si fueran dos caracteres más del archivo de disco). La variable de tipo `text` se trata como un archivo secuencial (por ejemplo, no puede emplear el procedimiento `seek`), también tiene subrutinas de uso propio y está estructurada por líneas. De forma que la declaración en un programa de la variable archivo con tipo

```
var   f: file of char;
```

es distinta de la declaración de la variable archivo de texto

```
var   f: text;
```

Sin embargo, un archivo de disco cuyos datos se almacenen en formato ASCII puede ser manipulado por cualquiera de las dos variables anteriormente declaradas en un programa de TurboPascal, teniendo en cuenta las peculiaridades en la manipulacion de cada una de los tipos de archivo.

Como resumen se subraya que en archivos de tipo `text` sólo pueden manipularse (escribir y leer) variables de tipo entero, real, `char` o `string` y cualquier dato que se trate se hará en formato ASCII.

### 8.6.5. Programas ejemplo con archivos de texto

Seguidamente se muestra otro ejemplo de la utilización de variables archivo de texto en un programa:

```
program archivador;
type ficha = record
       alumno : string[20];
       matricula: word;
       nota: real
     end;
var g:text;
procedure cf(var f:text; c:string);
  var i,n:integer; p:ficha;
  begin
  assign(f,c);
  rewrite(f);
  write('¿Cuantos alumnos?: ');
  readln(n);
  for i:=1 to n do
    with p do
      begin
      write('Nombre: ');
      readln(alumno);
      writeln(f,alumno);
      write('N° matricula: ');
      readln(matricula);
      write(f,matricula:6);
```

```
      write('Nota: ');
      readln(nota);
      writeln(f,nota:6:2)
      end;
  close(f)
  end;
{ Otras declaraciones ... }
function media(var f:text):real;
  var r,a:real; n,i:word;
  begin
  i:=0;r:=0;
  reset(f);
  while not eof(f) do
    begin
    readln(f);
    read(f,n);
    readln(f,a);
    r:=r+a;
    inc(i);
    end;
  close(f);
  media:=r/i
  end;
begin
{ Cuerpo del programa . . . }
end.
```

Al ejecutar en el programa principal la llamada al procedimiento

```
cf(g,'fichas');
```

se visualiza por pantalla:

```
¿Cuantos alumnos? 5↵
Nombre: Francisco Perez↵
N° matricula: 13245↵
Nota: 7.5↵
Nombre: Maria Lopez↵
N° matricula: 14074↵
Nota: 6.25↵
Nombre: Mario Garcia↵
N° matricula: 17721↵
Nota: 4↵
Nombre: Sergio Sebastian↵
N° matricula: 18555↵
Nota: 5.125↵
Nombre: Ana Martin↵
N° matricula: 19229↵
Nota: 8.2↵
```

creándose además el archivo de texto `fichas` en disco con el siguiente contenido:

```
Francisco Perez#13#10
 13245  7.50#13#10
Maria Lopez#13#10
 14074  6.25#13#10
Mario Garcia#13#10
 17721  4.00#13#10
Sergio Sebastian#13#10
 18555  5.13#13#10
Ana Martin#13#10
 19229  8.20#13#10
```

Al ejecutarse a continuación en el programa la sentencia `writeln(media(f));` se visualiza por pantalla:

```
6.2160000000E+00
```

## 8.7. LA FUNCIÓN IORESULT

La función estándar de TurboPascal `ioresult` permite la prevenir y gestionar errores que se puedan producir durante la ejecución de un programa al manipular variables archivo de cualquier tipo.

Cabecera de la función `ioresult : integer;`

Esta función sirve para todo tipo de archivos: devuelve un valor entero, que es cero si no se ha producido un error, y en caso de que haya ocurrido, un número entre 1 y 255 en función del error cometido.

Un ejemplo de utilización de esta función se da en el siguiente programa. Se trata de un programa que quiere añadir una serie de datos al final de un archivo ya existente o bien crear uno, si éste no existiera previamente, e introducir posteriormente la serie de datos en el nuevo archivo. Surge una cuestión: ¿qué procedimiento de apertura utilizar para abrir la variable archivo? Si se emplea el procedimiento `reset` y el archivo no existe se producirá un error de ejecución. Si se emplea el procedimiento `rewrite`, se creará un archivo nuevo, perdiéndose los datos que estuvieran en el archivo si éste existiera. El problema se ha solucionado empleando conjuntamente una directiva del compilador, {$I}, y la función `ioresult` de la siguiente manera:

```
program ejemplo;
type enteros=file of integer;
var f:enteros; i:integer;
begin
assign(f,'enteros.dat');
{$I-}
reset(f);
{$I+}
if ioresult<>0 then begin
   writeln('Se crea un archivo nuevo');
   rewrite(f)
   end
   else seek(f,filesize(f));
for i:=1 to 3 do write(f,i);
reset(f);
while not eof(f) do
   begin
   read(f,i);
   writeln(i)
   end
end.
```

La directiva de compilación {$I} controla la generación de errores en las operaciones de Entrada y Salida de datos (por ejemplo, en la escritura de datos en un archivo) en tiempo de ejecución. Por defecto, su estado es activa `{$I+}`. En el ejemplo anterior, la directiva {$I-} evita que se detenga la ejecución del programa al tratar de ejecutarse la sentencia `reset(f)` y no encontrarse el archivo asociado con el procedimiento `assign` en el disco. En este caso, aunque el programa pueda seguir ejecutándose con la siguiente sentencia, no puede volver a realizarse una llamada a una sentencia de manipulación de archivos. La directiva {$I+} permite de nuevo estas operaciones. La función `ioresult` determina si se ha producido un error o no en la operación anterior (`reset(f);`) y en el caso de que se haya producido (valor devuelto distinto de 0), presumiblemente por no existir dicho archivo, permite crear un archivo nuevo en disco.

En general, el control de esta directiva consiste en que después de cada operación de E/S el programa realiza internamente una llamada a una subrutina que supervisa este tipo de operaciones: la función estándar `ioresult`. Si el valor numérico entero devuelto por `ioresult` es igual a cero significa que no ha habido ningún error de E/S y el programa sigue su curso. En caso contrario la función devuelve un valor distinto de cero dependiendo del tipo

de error generado (ver siguiente sección), se genera un error de ejecución y el programa detiene su curso.

Si la directiva desconecta el control de las operaciones de E/S: `{$I-}`, al producirse un error de ejecución no se detiene el programa, pero no puede volverse a realizar otra operación de E/S hasta que no se llama explícitamente en el programa a `ioresult` lo cual inicializa el valor devuelto por la función (le da valor 0) permitiendo una nueva operación de E/S.

### 8.7.1. Errores de entrada/salida en decimal

Los valores que devuelve la función `ioresult` y el tipo de error asociado se indican en la Tabla 22.

Tabla 22. Código de error devuelto por la llamada a la función `ioresult`

| Código | Significado |
|---|---|
| `0` | No hay error |
| `1` | No existe el archivo. |
| `2` | No se abrió el archivo de entrada. Por ejemplo, falta la sentencia `reset(f)` |
| `3` | No se abrió el archivo de salida. |
| `4` | Archivo sin abrir. |
| `10` | Error en la conversión a formato numérico tras la lectura de un dato no numérico mediante una variable archivo `text`. |
| `32` | Operación no permitida en el dispositivo lógico. |
| `33` | No permitido en modo directo. |
| `34` | Asignación a archivo estándar no permitido. |
| `144` | Discrepancia en la longitud de un registro. |
| `145` | Búsqueda más allá del fin de archivo. |
| `153` | Fin de archivo inesperado. |
| `240` | Error de escritura en el disco. |
| `241` | Directorio lleno. |
| `242` | Tamaño de archivo demasiado grande. |
| `243` | Demasiados archivos abiertos. |
| `255` | Archivo desaparecido. |

## 8.8. VARIABLES ARCHIVOS SIN TIPO

El archivo sin tipo es tratado como una sucesión de bytes que se manipulan como una serie de bloques de bytes sin importar como estuvieran estructurados los datos almacenados en el archivo de disco. Permite trabajar con cualquier tipo de archivo cualesquiera sea su contenido, tipo o estructura.

### 8.8.1. Declaración de variables archivo sin tipo

La declaración de variables archivo sin tipo se realiza de la siguiente manera:

Sintaxis: `var f:file;`

donde `file` es el tipo de dato correspondiente a un archivo sin tipo

### 8.8.2. Estructura de las variables archivo sin tipo

Al emplear una variable archivo de tipo `file`, el archivo de disco asociado a manipular quedará formalmente estructurado en **n** bloques de **T** bytes cada uno.


Figura 46. Estructura de una variable archivo sin tipo.

En general, el tamaño del último bloque, F, puede ser distinto del tamaño definido para el bloque, T, es decir, puede no coincidir el producto n·T con el tamaño en bytes del archivo.

### 8.8.3. Subrutinas específicas de manipulación de archivos sin tipo

Si bien algunas de las subrutinas vistas anteriormente pueden emplearse en este tipo de archivo, existen una serie de subrutinas que son específicas para su manipulación.

#### 8.8.3.1 Procedimientos de apertura

El formato de los procedimientos de apertura de variables archivo `reset` y `rewrite` es distinto al usado anteriormente. Sus cabeceras de declaración, en el caso de ser empleadas con variables archivo sin tipo, corresponden respectivamente a:

```
procedure reset(var f:file; bloque: word);
procedure rewrite(var f:file; bloque: word);
```

donde *`bloque`* es un número entero de tipo `word` que indica el tamaño (en bytes) o longitud de los bloques en los que queda estructurada la variable archivo. Este segundo parámetro es opcional; si se omite se configura por defecto un tamaño de bloque de 128 bytes.

Ej.:
```
var f:file;
begin
assign(f,'datos.dat');
reset(f,16);
```

En el ejemplo anterior se abre la variable archivo `f` dejando estructurado *formalmente* para su manipulación en bloques de 16 bytes el archivo `datos.dat`.


Figura 47. Variable archivo sin tipo estructurada en bloques de 16 bits (con $F \leq 16$).

#### 8.8.3.2 Procedimientos de entrada y salida de datos

Para escribir o leer datos en una variable archivo sin tipo no se utilizan los procedimientos `write`, `writeln`, `read` o `readln` vistos anteriormente, sino otros específicos para este tipo de variables archivo.

El procedimiento de escritura de datos `blockwrite` escribe uno o más bloques de datos procedente de una variable genérica auxiliar en una variable archivo sin tipo. Su cabecera es la siguiente:

```
procedure blockwrite(var f:file;var aux; m:word [;var r:word]);
```

El procedimiento de lectura de datos `blockread` lee uno o más bloques de datos de una variable archivo sin tipo y los almacena en una variable genérica auxiliar. Su cabecera es la siguiente:

```
procedure blockread(var f:file;var aux; m:word [;var r:word]);
```

Los parámetros formales indicados anteriormente tienen el siguiente significado:

a) *`f`* es una **variable archivo** de tipo `file`,

b) *`aux`* es una variable de cualquier tipo, **de donde se toman los datos** que se escriben en la variable archivo (`blockwrite`) o **en donde se van a introducir los datos** que se

lean de la variable archivo (`blockread`). Se suele emplear habitualmente una variable de tipo `array` de un tamaño adecuado al total de los bloques manipulados,

c) *m* es una expresión de tipo `word` que indica el **número máximo** de bloques a manipular en cada llamada al procedimiento,

d) *r* es un parámetro **opcional** (por eso se ha indicado entre corchetes en la cabecera) que almacena el nº de bloques que se han terminado de escribir o leer tras la ejecución del procedimiento. Si se especifica en la llamada, el parámetro real ha de ser una variable de tipo `word`.

Aunque ambos procedimiento parezcan tener parámetros similares, mientras que el procedimiento `BlockWrite` escribe uno o más bloques (sucesiones de bytes almacenados en la variable genérica `aux`) en el archivo asociado a la variable `f`, el procedimiento `BlockRead` lee uno o más bloques de bytes del archivo y los almacena en la variable `aux`. Por lo tanto, es importante tener siempre presente el tamaño de la variable genérica `aux` y el número y tamaño de los bloques con los que se trabajan, para evitar problemas en la escritura y lectura de los datos.

Como continuación del ejemplo de la sección anterior, en el programa:

```
var f:file: a:array[1..64] of char;
begin
assign(f,'datos.dat');
reset(f,16);
blockread(f,a,4);
```

la ejecución de la sentencia `blockread(f,a,4);` almacenaría 4 bloques (4 x 16 bytes) en la variable `a`, que podría haberse definido previamente, por ejemplo, como una variable de tipo `array[1..64] of char`, es decir, con el tamaño adecuado para almacenar todo lo leído del archivo (64 bytes).

### 8.8.4. Otros procedimientos y funciones

Algunos de los procedimientos y funciones empleados en los tipos de archivo vistos anteriormente pueden asimismo emplearse con las variables archivo sin tipo. Entre estos se encuentran los procedimientos `Close`, `Reset`, etc.

Como ejemplo de utilización de las rutinas anteriores, los siguientes programas realizan una copia de un archivo de disco en otro:

```
(* Realiza una copia del archivo antiguo.dat *)
program copia_archivo_1;
var f,g: file; res_f,res_g: word;
    aux: array[1..2048] of byte;
begin
assign(f,'antiguo.dat');
reset(f,1);
assign(g,'nuevo.dat');
rewrite(g,1);
repeat
  blockread(f,aux,SizeOf(aux),res_f);
  blockwrite(g,aux,res_f,res_g)
until (res_f=0) or (res_f<>res_g);
close(f);
close(g)
end.
(* Tambien realiza una copia del archivo antiguo.dat *)
program copia_archivo_2;
var f,g: file; res_f,res_g: word;
    aux: array[1..1024] of real;
begin
assign(f,'antiguo.dat');
reset(f,256);
assign(g,'nuevo.dat');
```

```
rewrite(g,256);
repeat
  blockread(f,aux,24,res_f);
  blockwrite(g,aux,res_f,res_g)
until (res_f=0) or (res_f<>res_g);
close(f);
close(g)
end.
```

La única diferencia entre los dos programas anteriores estriba en el tamaño de los bloques de datos que manipulan. En el primer programa el tamaño de 1 byte y en el segundo es de 256 bytes.

Nota: `SizeOf(aux)` es una función estándar de TurboPascal que devuelve el tamaño de la variable `aux` en bytes.

**Bibliografía básica**


- **García-Beltrán, A., Martínez, R.** y **Jaén, J.A.** *Métodos Informáticos en TurboPascal*, Ed. Bellisco, 2ª edición, Madrid, 2002
- **Wirth, N.** *Algoritmos + Estructuras de Datos = Programas*, Ediciones del Castillo, Madrid, 1986
- **Kruse, R.** *Estructuras de Datos y Diseño de Programas*, Prentice-Hall, 1988
- **Joyanes, L.** *Fundamentos de programación, Algoritmos y Estructuras de Datos*, McGraw-Hill, Segunda edición, 1996
- **Duntemann, J.** *La Biblia de TurboPascal*, Anaya Multimedia, Madrid, 1991