Departamento de Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

Apuntes para la asignatura

Informática

Facultad de Ciencias (Matemáticas)

http://www.lcc.uma.es/personal/pepeg/mates

Tema 9. Almacenamiento externo de la información

Bibliografía

- *Programación 1*. José A. Cerrada y Manuel Collado. Universidad Nacional de Educación a Distancia.
- *Programming with TopSpeed Modula-2*. Barry Cornelius. Addison-Wesley.
- *Fundamentos de programación*. L. Joyanes. McGraw-Hill.

Introducción

En este tema estudiamos el tipo de dato `File` de *TopSpeed Modula-2* y las operaciones para el manejo de éste que se encuentran en la biblioteca `FIO`.

9.1. Ficheros

En el tema 1 estudiamos los tipos de memoria existentes en un ordenador: memoria *principal* (RAM) y memoria *masiva* (discos, cintas magnéticas,...). Vimos que para que un programa se ejecutase era necesario que éste y los datos que manejase estuviesen cargados en memoria principal. También estudiamos las características de ambos tipos de memoria, que podemos resumir en:

- memoria principal: rápida, capacidad limitada y volátil.
- memoria masiva: más lenta, mayor capacidad y no volátil.

Todos los datos que se han manipulado hasta el momento en los algoritmos presentados tienen una característica común: están almacenados en memoria principal. Al ser la memoria principal volátil, la consecuencia inmediata es que el contenido de estos datos se pierde al apagar el ordenador (e incluso cuando termina el programa que se está ejecutando). Sin embargo, los ordenadores suelen utilizarse para almacenar datos que serán recuperados posteriormente. Para conseguir esto son necesarios mecanismos que permitan:

- copiar el contenido de la información situada en memoria principal sobre la memoria secundaria (antes de terminar la ejecución del programa)
- recuperar el contenido de información almacenada previamente en memoria secundaria.

Para trabajar con datos almacenados en memoria secundaria es necesario declarar variables donde temporalmente se copien éstos durante la ejecución del programa y operaciones adicionales que establecen la conexión entre dichas variables en memoria principal y los datos en memoria secundaria.

En *Modula-2* esta conexión se consigue mediante el tipo de datos fichero (File) y sus operaciones asociadas. En *TopSpeed Modula-2*, el módulo de biblioteca FIO contiene todos los servicios para el tratamiento de ficheros, incluyendo la definición del tipo File, que no es un tipo predefinido en *Modula-2*.

Nota: el modulo FIO es propio de *TopSpeed Modula-2* y no es estándar. Otros compiladores pueden no contar con las operaciones que veremos en este tema, aunque poseerán algunas similares.

En la memoria secundaria la información está organizada en archivos o ficheros. Un *fichero* se puede definir como una colección de información o elementos lógicamente relacionados y almacenados en memoria secundaria. En realidad un fichero es tan solo una colección de bytes, pero éstos pueden estar organizados jerárquicamente en estructuras que facilitan y aumentan la eficiencia de las operaciones sobre el mismo.

9.1.1 Ficheros en *MS-DOS*

Dentro de un mismo dispositivo de almacenamiento secundario (disco duro, disquete, cinta magnética, CD-ROM, ...) pueden almacenarse varios ficheros. Cuando queremos acceder a un fichero lo hacemos mediante su nombre. El *nombre* del fichero es una cadena de caracteres que debe seguir ciertas reglas, las cuales dependen del sistema operativo que utilice el ordenador. En nuestro caso, el sistema operativo *MS-DOS* exige que el nombre del fichero esté

formado por no más de ocho caracteres. El primer carácter deberá ser obligatoriamente una letra (no podrá ser un número ni un signo). Tampoco podremos utilizar el espacio en blanco como parte del nombre de un fichero. Adicionalmente podremos añadir no más de tres letras al fichero precedidas del carácter punto. A esto se llama extensión del fichero y se suele usar para, siguiendo cierto convenio, indicar el tipo de datos almacenado en el fichero. Por último, hemos de comentar que *MS-DOS* no distingue entre letras mayúsculas y minúsculas en el nombre de un fichero. Así, los nombres de fichero "Datos", "DATOS" y "datos" se refieren todos a un mismo fichero.

Algunos ejemplos de nombres de fichero son:

- `"Datos"`: nombre de fichero de cinco letras sin extensión
- `"Programa.mod"`: nombre de fichero con ocho letras y extensión de tres letras. La extensión `"mod"` se suele usar para ficheros cuyo contenido es el texto de un programa escrito en el lenguaje *Modula-2*.
- `"Programa.exe"`: nombre de fichero con ocho letras y extensión de tres letras. La extensión `"exe"` se suele utilizar para ficheros cuyo contenido es un programa ejecutable, resultado de compilar un programa.

9.2. Tipos de ficheros

Simplificando mucho, un fichero es tan solo una secuencia de bytes que contienen cierta información. Consideramos a continuación un par de criterios para clasificar ficheros: su contenido y el modo en que se puede acceder a la información que contienen.

9.2.1 Clasificación de ficheros según su contenido

Uno de los criterios que podemos seguir para clasificar ficheros es el tipo de información que almacenan. De este modo podemos distinguir entre ficheros de *texto* y *binarios*.

9.2.1.1 Ficheros de texto

El contenido del fichero es una secuencia de caracteres pertenecientes a un determinado código de entrada/salida, normalmente el código ASCII. Suelen estar compuestos por una serie de líneas.

Un ejemplo típico de fichero de texto es aquel que almacena el texto correspondiente a un programa escrito en *Modula-2*. Las líneas del fichero corresponden a las distintas líneas del programa que contiene.

Cuando desde un programa escrito en *Modula-2* se almacena el contenido de un valor cadena de caracteres en un fichero de texto, sólo es necesario copiar los caracteres desde memoria interna al fichero correspondiente. Algo similar sucede para operaciones de lectura.

Sin embargo, se necesita más trabajo cuando se pretenden escribir valores de otros tipos (INTEGER, CARDINAL, REAL, ...). En este caso hay que realizar una conversión de la representación interna en memoria del valor (una serie de bytes que representan el número en binario como puede ser 00011001) a la correspondiente secuencia de caracteres que lo representan externamente (en este caso podría ser "25"). En las operaciones de lectura se realiza la conversión inversa. Los subprogramas en la biblioteca FIO se encargan de esta tarea.

La principal ventaja que presenta este tipo de ficheros es que una persona puede entender directamente su contenido si lo inspecciona.

Nota: la conversión también ocurre en las lecturas/escrituras en dispositivos estándares (teclado/pantalla).

9.2.1.2 Ficheros binarios

En lugar de realizar las conversiones anteriormente comentadas para almacenar caracteres en un fichero, las propias representaciones internas de los datos pueden copiarse en un fichero directamente. Un fichero creado de esta forma se denomina *binario*.

La principal ventaja de este tipo de ficheros es que las operaciones con ellos son más rápidas (no es necesario realizar la transformación desde la representación interna de memoria a la cadena de texto correspondiente). La principal desventaja es que su contenido no puede ser entendido directamente por una persona al inspeccionarlo.

Modula-2 también provee operaciones en el modulo de biblioteca FIO para el manejo de este tipo de ficheros, aunque no las estudiaremos en este curso.

9.2.2 Clasificación de ficheros según su acceso

Si atendemos al modo en que se puede acceder a la información almacenada en el fichero podemos clasificar éstos como: ficheros *secuenciales* y ficheros con *acceso directo*.

9.2.2.1 Ficheros secuenciales

Un fichero secuencial es aquel en el cual para acceder al elemento del contenido que ocupa la posición *i*-ésima hay que acceder previamente a los *i*-1 elementos previos.

9.2.2.2 Ficheros con acceso directo

Un fichero con acceso directo es aquel que permite acceder al elemento que ocupa la posición *i*-ésima directamente (al estilo del acceso en un ARRAY) sin pasar necesariamente por los elementos previos.

La biblioteca FIO de *TopSpeed Modula-2* posee operaciones para el manejo de ficheros secuenciales y con acceso directo, aunque en este curso sólo estudiaremos las primeras.

9.3 Gestión de ficheros

Los ficheros se suelen utilizar de tres modos:

- para escribir datos en ellos
- para leer datos contenidos en ellos
- para modificar o almacenar información adicional a la que ya contenían

A la hora de utilizar un fichero debemos indicar de qué modo vamos a utilizarlo. Veremos en los puntos siguientes como hacerlo.

La gestión básica de ficheros en *TopSpeed Modula-2* se realiza a través de la biblioteca FIO. Para utilizar los servicios ofrecidos por esta biblioteca, es necesario importar dicho módulo.

9.3.1. Manejadores de ficheros

Hemos visto que un fichero se identifica mediante su nombre y que antes de utilizarlo hay que indicar de qué modo va a ser utilizado (para lectura, escritura o ambas). En *Modula-2*, para utilizar un fichero hay que abrirlo previamente mediante una de las siguientes operaciones: `Open`, `OpenRead`, `Create` o `Append`. Al utilizar las operaciones anteriores se especifica el nombre del fichero que se va a utilizar. Estas operaciones devuelven un valor de tipo `File`, que es conocido como *manejador de fichero*. Posteriormente cuando queramos leer o escribir información en el fichero NO utilizaremos el nombre de éste, sino el manejador que obtuvimos

al abrirlo. Cada fichero abierto tiene un manejador de fichero diferente. Cuando hayamos terminado de trabajar con el fichero debemos cerrarlo mediante la operación `Close`.

Asociada a cada fichero abierto existe una *cabeza de lectura/escritura*. Ésta indica la posición dentro del fichero de donde se leerá (o en donde se escribirá) información la próxima vez que se ejecute una operación de lectura (o escritura). En el apartado siguiente veremos que, dependiendo de la operación de apertura utilizada, la cabeza se colocará inicialmente al principio o al final del fichero. También veremos que las operaciones de lectura y escritura avanzan automáticamente esta cabeza cada vez que se ejecutan.

9.3.2 Apertura y cierre de ficheros

Para la apertura de un fichero existen diversas posibilidades. Hay cuatro formas de apertura.

- Creación de un nuevo fichero: en este caso usaremos la función `Create`:

  ```
  PROCEDURE Create(Nombre: ARRAY OF CHAR): File;
  ```

  El parámetro de esta función es el nombre del fichero que se desea crear.

  - Si el fichero no existe, se crea uno nuevo vacío. Después de esta operación el fichero queda abierto, listo para ser usado por el programa.
  - Si el fichero indicado ya existe, su contenido se borra y se abre como si se hubiese creado uno nuevo.

  El valor devuelto por dicha función es el manejador asociado a dicho fichero. Este manejador sirve para poder referenciar al fichero que se acaba de crear. La cabeza de lectura/escritura se sitúa al principio del fichero.

- Apertura de un fichero ya existente para lectura o escritura. En este caso usamos la función `Open`:

  ```
  PROCEDURE Open(Nombre: ARRAY OF CHAR): File;
  ```

  - Si el fichero ya existe, abre el fichero especificado para lectura o escritura, e inicializa la cabeza de lectura/escritura al principio del mismo.
  - Si el fichero no existe se produce un error en tiempo de ejecución.

- Apertura de un fichero ya existente sólo para lectura. En este caso se usa la función `OpenRead`:

  ```
  PROCEDURE OpenRead(Nombre: ARRAY OF CHAR): File;
  ```

  En este caso, sólo podemos realizar operaciones que lean datos del fichero. Si usamos operaciones que intenten modificar el contenido de éste se produce un error en tiempo de ejecución.

- Apertura de un fichero ya existente para añadir datos al final. En este caso usamos la función `Append`:

  ```
  PROCEDURE Append(Nombre: ARRAY OF CHAR): File;
  ```

  - Si el fichero ya existe, `Append` abre el fichero especificado para lectura o escritura, e inicializa la cabeza de lectura/escritura al final del mismo. Por lo tanto, se utiliza cuando se quiere añadir datos detrás de los ya existentes.
  - Si el fichero no existe se produce un error en tiempo de ejecución.

Siempre hay que cerrar un fichero cuando se termina de trabajar con él. El procedimiento `Close` se utiliza para ello.

```
PROCEDURE Close(Fich: File);
```

Este procedimiento toma como parámetro el manejador del fichero abierto que queremos cerrar.

Si hemos de determinar si un fichero existe o no antes de elegir la operación con la cual abrirlo, podemos utilizar la función `Exists`:

```
PROCEDURE Exists(Nombre: ARRAY OF CHAR): BOOLEAN;
```

Esta función toma como parámetro el nombre de un fichero y devolverá TRUE si el fichero ya existe, o FALSE en caso contrario.

Veamos un ejemplo de uso de las operaciones anteriores. Se trata de un programa para añadir nueva información al final del fichero "`Datos.txt`". Previamente comprobamos si el fichero existe (en cuyo caso lo abrimos con `Append`) o no (en este caso se crea con `Create`). Una vez añadida la información al fichero, se cierra:

```
MODULE Fichero;
FROM FIO IMPORT File, Create, Append, Exists, Close;
CONST
  NOMBREFICHERO = "Datos.txt";

VAR
  Manejador : File;

BEGIN
  IF Exists(NOMBREFICHERO) THEN
    Manejador := Append(NOMBREFICHERO)
  ELSE
    Manejador := Create(NOMBREFICHERO)
  END;
  ...
  (* Operaciones de escritura *)
  ...
  Close(Manejador)
END Fichero.
```

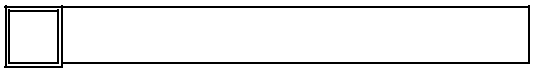
9.3.3 Escritura de ficheros

Las operaciones de escritura en ficheros son similares a las de escritura por pantalla, pero toman un parámetro adicional que es el manejador del fichero en el cual queremos escribir el dato. Algunas de estas operaciones son:

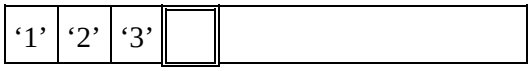
```
PROCEDURE WrChar(Fich:File, V:CHAR);
PROCEDURE WrInt(Fich:File, V:INTEGER, Long:INTEGER);
PROCEDURE WrLngInt(Fich:File, V:LONGINT, Long:INTEGER);
PROCEDURE WrCard(Fich:File, V:CARDINAL, Long:INTEGER);
PROCEDURE WrLngCard(Fich:File, V:LONGCARD, Long:INTEGER);
PROCEDURE WrReal(Fich:File, V:REAL, Pre:CARDINAL, Lon:INTEGER);
PROCEDURE WrLngReal(Fich:File, V:LONGREAL, Pre:CARDINAL,Lon:INTEGER);
PROCEDURE WrFixReal(Fich:File, V:REAL, Pre:CARDINAL, Lon:INTEGER);
PROCEDURE WrFixLngReal(Fich:File, V:LONGREAL, Pre:CARDINAL,Lon:INTEGER);
PROCEDURE WrStr(Fich: File, V:ARRAY OF CHAR);
PROCEDURE WrLn(Fich: File);
```

Estos procedimientos escriben los datos que toman como parámetros en el fichero indicado a partir de la posición actual de la cabeza de lectura/escritura. La posición de la cabeza se incrementa automáticamente para que apunte a la siguiente posición a los caracteres escritos.

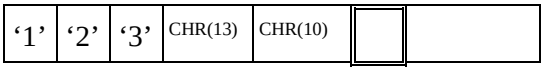
Veamos un ejemplo. Supongamos que hemos abierto un fichero con la operación `Create`. Esta operación deja la cabeza al principio del fichero por lo que tenemos la siguiente situación:



Si a continuación utilizamos la operación de escritura de enteros en ficheros `WrInt(Manejador,123,0)` la situación del fichero pasa a ser:



El procedimiento `WrLn` escribe una indicación de fin de línea en el fichero. Esta indicación consiste en escribir los caracteres **`CHR`**`(13)` y **`CHR`**`(10)` en el fichero en este orden exactamente. Si llamamos a `WrLn(Manejador)` el estado del fichero queda:



Los procedimientos `WrFixReal` y `WrFixLngReal` se usan para escribir valores reales en notación fija (no científica).

Obsérvese que el nombre de los procedimientos coincide con el nombre de los procedimientos de salida por pantalla del módulo IO. Si nos interesa utilizar un procedimiento del módulo IO y otro de FIO con el mismo nombre en un mismo programa, podemos utilizar la importación de módulos completos y referirnos a la operación concreta precediéndola del nombre de la biblioteca y un punto:

```
MODULE Importar;
IMPORT IO, FIO;
...
BEGIN
  ...
  (* Escribe por pantalla *)
  IO.WrStr(“A pantalla”);
  ...
  (* Escribe en un fichero *)
  FIO.WrStr(Manejador, “A fichero”);
  ...
END Importar.
```

Nota Importante: La lectura de los datos almacenados en el fichero debe respetar una serie de convenios, por lo tanto, dichos datos se deben escribir (tanto con un programa en *Modula-2* como de cualquier otra forma, por ejemplo con un editor) teniendo en cuenta que dichos datos serán posteriormente leídos. Estos convenios tienen que ver con el funcionamiento de las funciones de lectura/escritura de *Modula-2*.

9.3.4 Lectura de ficheros

El módulo FIO también contiene operaciones que permiten leer datos desde un fichero. Algunas de éstas son:

```
PROCEDURE RdChar(Fich:File): CHAR;
PROCEDURE RdInt(Fich:File): INTEGER;
PROCEDURE RdLngInt(Fich:File): LONGINT;
PROCEDURE RdCard(Fich:File): CARDINAL;
PROCEDURE RdLngCard(Fich:File): LONGCARD;
PROCEDURE RdReal(Fich:File): REAL;
PROCEDURE RdLngReal(Fich:File): LONGREAL;
PROCEDURE RdStr(Fich: File, VAR V:ARRAY OF CHAR);
```

Estos subprogramas se comportan de forma parecida a sus homólogos del módulo IO, salvo que los caracteres se toman del fichero especificado como primer parámetro, a partir de la posición indicada por la posición de lectura/escritura. Tras la lectura, dicha posición se incrementa para que apunte a la siguiente posición a los caracteres leídos.

El procedimiento `RdStr` se comporta de forma diferente al homólogo de IO, por lo que lo estudiaremos con más detenimiento.

9.3.4.1 Lectura de múltiples datos desde teclado

Hasta ahora hemos usado siempre la instrucción `RdLn` tras cualquier operación de entrada desde teclado. Vamos a explicar en este punto porqué ha sido esto necesario y cómo funciona la lectura desde teclado más detalladamente.

Sabemos que cuando un programa escrito en *Modula-2* llega a una operación de entrada desde teclado se detiene, muestra un cursor parpadeante y espera a que el usuario introduzca un dato. El programa no reanuda su ejecución hasta que el usuario pulsa la tecla especial <ENTER>. Los datos tecleados por el usuario se almacenan en una zona llamada *buffer* del teclado. La pulsación de la tecla <ENTER> hace que se almacenen los caracteres **CHR**(13) y **CHR**(10) al final del buffer del teclado. Ahora bien, nada impide que el usuario teclee más de un dato cuando aparece el cursor parpadeante. Supongamos que un programa ejecuta la instrucción `N:=RdInt()` y que el usuario escribe 12, un espacio en blanco, 15 y entonces pulsa <ENTER>. El contenido del buffer del teclado queda como se indica:

‘1’	‘2’	‘␣’	‘1’	‘5’	CHR(13)	CHR(10)

Como `RdInt` lee del teclado hasta encontrar un carácter que no forme parte de un número, `N` toma el valor 12 y la cabeza de lectura/escritura queda en el espacio en blanco (la lectura desde teclado no salta el separador final):

‘1’	‘2’	‘␣’	‘1’	‘5’	CHR(13)	CHR(10)

El resto de la información queda almacenada en el buffer del teclado, de modo que si el programa ejecuta otra operación `M:=RdInt()` no llega a detenerse ya que hay información suficiente en el buffer. En este caso `M` tomaría el valor 15 y el buffer del teclado quedaría:

‘1’	‘2’	‘␣’	‘1’	‘5’	CHR(13)	CHR(10)

La ejecución de un programa no se detiene en una operación de lectura desde teclado hasta que se agota el contenido del buffer. En ese momento el programa se detendrá a la espera de que el usuario introduzca más datos.

Lo único que hace la operación de lectura `RdLn` es vaciar el buffer del teclado. Si llamamos a `RdLn` tras cualquier operación de lectura desde teclado, borramos cualquier información extra que el usuario haya podido introducir.

A veces puede ser interesante no llamar a `RdLn` tras una operación de entrada. El siguiente programa pide al usuario que introduzca diez números de una vez y los lee en un ARRAY:

```
MODULE SinRdLn;
FROM IO IMPORT RdInt, RdLn, WrStr;
CONST
  MAXIMO = 10;
VAR
  Vector : ARRAY [1.. MAXIMO] OF INTEGER;
  i      : CARDINAL;
BEGIN
  WrStr(“Dame 10 nºs separados por espacio y pulsa ENTER: ”);
  FOR i := 1 TO MAXIMO DO
    Vector[i] := RdInt()
```

```
    END;
    RdLn
    ...
END SinRdLn.
```

9.3.4.2 Diferencia entre la lectura de teclado y de fichero

Supongamos que el buffer de teclado contiene la siguiente información (el próximo carácter a leer es el enmarcado en trazo doble).

‘␣’	‘␣’	‘1’	‘2’	‘3’	‘␣’	‘A’	...

Después de realizar la operación `N:=RdInt()`, `N` valdrá 123 y el buffer queda como sigue:

‘␣’	‘␣’	‘1’	‘2’	‘3’	‘␣’	‘A’	...

si ahora realizamos la operación `C:=RdChar()`, `C` contiene el carácter blanco `‘␣’`.

Veamos ahora lo que sucede en el caso de lectura de ficheros (la posición de lectura/escritura se muestra en trazo doble):

‘␣’	‘␣’	‘1’	‘2’	‘3’	‘␣’	‘A’	...

Después de realizar la operación `N:=RdInt(Manejador)`, `N` también vale 123 pero la posición de lectura/escritura queda como sigue:

‘␣’	‘␣’	‘1’	‘2’	‘3’	‘␣’	‘A’	...

con lo que si realizamos la operación `C:=RdChar(Manejador)` `C` contendrá el carácter `‘A’`.

La diferencia fundamental entre la lectura desde el teclado y la lectura desde fichero es que las operaciones de lectura desde teclado no saltan el separador (en este caso el blanco) que origina el fin del número, mientras que las operaciones de lectura desde fichero sí lo hacen (avanzan una posición más).

Nótese que las funciones para leer enteros, cardinales y reales saltan los separadores (`‘␣’`, **CHR**(9), **CHR**(10), **CHR**(13), **CHR**(26)) que se encuentran antes de los datos a leer. En las bibliotecas IO y FIO se encuentra una variable de tipo SET OF CHAR que contiene los valores que se consideran separadores. Esta variable es `Separators` y se puede modificar utilizando los procedimientos INCL y EXCL.

Si el dato introducido por el usuario no representa un número válido del tipo que se intenta leer el valor devuelto por las funciones de lectura es indefinido. Se puede importar la variable booleana `OK` del módulo IO (o de FIO si estamos leyendo desde ficheros) que se pone a FALSE si el dato introducido por el usuario no fue válido. El siguiente programa pide un dato de tipo CARDINAL por teclado y no cesa hasta conseguirlo:

```
MODULE Leer;
IMPORT IO;
VAR
  c : CARDINAL;
BEGIN
  LOOP
    IO.WrStr("Dame un valor CARDINAL: ");
    c := IO.RdCard();
    IF IO.OK THEN
      EXIT
    END;
    IO.WrStr("Valor no válido. Prueba de nuevo");
    IO.WrLn
  END;
  IO.WrStr("El valor leído es: ");
  IO.WrCard(c, 0)
```

```
END Leer.
```

9.3.4.3 Detección del fin de fichero

Para controlar correctamente la lectura de datos de un fichero, necesitamos algún mecanismo para saber si existen más datos en éste o no, es decir, si la posición de lectura/escritura se encuentra al final del fichero.

Para ello se utiliza la variable booleana EOF (siglas de *End Of File*) que se importa del módulo FIO. Su utilización es similar a la de la variable OK ya vista. Si la variable EOF es TRUE, indica que en la última operación de lectura de fichero que se intentó, la cabeza de lectura/escritura ya estaba sobre el final del fichero. Obsérvese que HAY QUE INTENTAR LEER ALGO Y LUEGO COMPROBAR SI LO LEÍDO ES VÁLIDO en este orden exactamente.

Veamos un programa que muestra el contenido de un fichero de texto por pantalla:

```
MODULE Fichero;
IMPORT IO, FIO;
CONST
  MAX = 20;
TYPE
  Nombre = ARRAY [1..MAX] OF CHAR;
VAR
  C           : CHAR;
  Fich        : Nombre;
  Manejador   : FIO.File;
BEGIN
    IO.WrStr("Introduce el nombre del fichero: ");
    IO.RdStr(Fich);
    Manejador := FIO.Open(Fich);     (* Apertura de fichero  *)
    LOOP
      C := FIO.RdChar(Manejador);    (* Lectura de datos    *)
      IF FIO.EOF THEN                (* Fin de fichero ?    *)
        EXIT
      END;
      IO.WrChar(C);                  (* Escribir en pantalla *)
    END;
   FIO.Close(Manejador);
END Fichero.
```

9.3.4.4 Lectura de cadenas de caracteres. RdStr

El comportamiento del procedimiento `RdStr` del módulo FIO es diferente a su homólogo del módulo IO. Su comportamiento es como se indica a continuación:

Se leen caracteres del fichero especificado hasta que:

1) Se encuentra los caracteres correspondientes a un fin de línea (**CHR**(13) y **CHR**(10)). Estos caracteres, sin embargo, no se almacenan en la cadena leída. Se coloca en su lugar un carácter **CHR**(0).
2) Se encuentra el fin de fichero. En este caso también se añade un carácter **CHR**(0) al final de la cadena.
3) Se llena la cadena argumento.

Algunas consideraciones sobre esta operación son:

- Si los primeros caracteres leídos son **CHR**(13) y **CHR**(10), la cadena contiene la cadena vacía (una cadena con el carácter **CHR**(0) en su primera posición).

- Si la llamada a `RdStr` es seguida por una llamada a `RdChar`, ésta lee el primer carácter del fichero que no leyó `RdStr`. Este carácter dependerá de la causa que motivó el fin de la operación `RdStr`. Para cada uno de los casos anteriores tenemos:
  1) Se leerá el primer carácter de la siguiente línea. Nótese que los caracteres de fin de línea son saltados.
  2) Se producirá fin de fichero.
  3) Se leerá el siguiente carácter a los que se leyeron.

9.3.4.5 Convenios para la escritura en ficheros de texto

Las funciones para leer enteros, cardinales y reales desde un fichero leen hasta encontrarse un carácter que no corresponda a un número de dicho tipo. Este carácter (que llamaremos separador) es leído también. Como consecuencia de esto, resulta que es necesario escribir algún carácter especial (como puede ser un espacio en blanco) siempre que escribamos un dato numérico en un fichero y queramos que éste pueda ser leído posteriormente. Si no seguimos este convenio no podremos recuperar la información almacenada posteriormente con éxito. En efecto, supongamos que NO lo hacemos y que escribimos dos enteros (10 y 25) en un fichero:

```
...
Manejador := FIO.Open(Fich);
FIO.WrInt(Manejador, 10, 0);
FIO.WrInt(Manejador, 25, 0);
...
```

el contenido del fichero sería:

‘1’	‘0’	‘2’	‘5’	‘b’	...

Por lo cual, si posteriormente abrimos el fichero para lectura e intentamos leer un número obtendremos 1025. Sin embargo, si seguimos el convenio indicado:

```
...
Manejador := FIO.Open(Fich);
FIO.WrInt(Manejador, 10, 0);
FIO.WrChar(Manejador, ‘ ‘);
FIO.WrInt(Manejador, 25, 0);
FIO.WrChar(Manejador, ‘ ‘);
...
```

el contenido del fichero es ahora

‘1’	‘0’	‘b’	‘2’	‘5’	‘b’	‘b’	...

y los dos enteros almacenados pueden ser leídos con éxito:

```
...
Manejador := FIO.OpenRead(Fich);
a := FIO.RdInt(Manejador);

b := FIO.RdInt(Manejador);
...
```

Por otro lado, la función de lectura de cadenas de caracteres `RdStr` lee hasta fin de línea, por lo tanto se debe escribir el fin de línea después de escribir en el fichero una cadena de caracteres, si queremos que sea posible leerla posteriormente con `RdStr`:

```
...
Manejador := FIO.Open(Fich);
FIO.WrStr(Manejador, “Hola”);
FIO.WrLn(Manejador);
...
```

Relación de problemas (Tema 9)

1. Escribe un programa que lea del teclado el nombre de tres ficheros y escriba en el tercero el contenido del primero seguido del contenido del segundo.

2. Escribe un programa que codifique y decodifique ficheros de texto usando el método de *cifrado César*. Para codificar un fichero de texto (cuyo nombre y extensión se leerá de teclado) se generará otro fichero de texto con el mismo nombre y con extensión `".COD"`. La codificación consistirá en reemplazar cada carácter del fichero original por el tercero siguiente según la tabla ASCII (Por ejemplo, el carácter `'a'` será sustituido por el carácter `'d'`). Si el carácter a codificar es **`CHR`**`(10)` o **`CHR`**`(13)` no se cambiará al codificarlo. La opción de decodificación deberá leer de teclado el nombre de un fichero codificado y recuperar en otro fichero con mismo nombre y con extensión `".DEC"` la información original.

NOTA: El desplazamiento en tres caracteres debe ser circular tanto en un sentido como en otro. Por ejemplo, el último carácter de la tabla ASCII (**`CHR`**`(255)`) es codificado como **`CHR`**`(2)`.

3. Diseña un programa que cree un fichero que contenga los datos relativos a los artículos de un almacén. Para cada artículo hay que guardar la siguiente información:

Código del artículo	(Numérico)
Nombre del artículo	(Cadena de caracteres)
Existencias actuales	(Numérico)
Precio	(Numérico).

Los datos de los distintos artículos se pedirán del teclado y se almacenarán previamente en un array. Supondremos que el número máximo de artículos que se pueden introducir es treinta. Una vez leídos, los datos se ordenarán en el array (el orden será ascendente según el código del artículo) y se grabarán en el fichero.

4. Escribe un programa que a partir de dos ficheros generados por el programa anterior (recordar que los artículos están ordenados por código) genere un tercer fichero que sea el resultado de mezclar de formar ordenada los dos primeros. Escribe otro programa que haga lo mismo pero con *n* ficheros (*n* será un número positivo leído de teclado).

5. Escribe un programa que tome como entrada un fichero generado por el programa del ejercicio **3** y una condición sobre el campo referente a las existencias actuales. La condición podrá ser:

[<,<=,>,>=, <>, =] <número>

Es decir, que las referencias actuales sean menores a un número dado, o menores o iguales, o mayores, etc.

El programa debe generar como salida un fichero llamado `"salida.dat"` que contenga todos aquellos artículos para los que se cumple la condición de entrada.

6. Escribe un programa que tenga como entrada el nombre de un fichero que contenga un texto y muestre en pantalla una estadística de la longitud de las distintas palabras que contiene (número de palabras de cada longitud), así como cuántas veces aparecen los caracteres separadores (`","`, `"."`, `":"`, `";"`, `"~~b~~"`).

7. Escribe un programa que a partir de un fichero de entrada que contiene números enteros positivos expresados en cualquier base entre 2 y 10, genere un listado por pantalla y un fichero de salida que contenga las mismas cantidades del fichero de entrada, pero

expresadas en base 10.

El fichero de entrada contendrá en cada línea la base y el número separados por el carácter '/'.

Ejemplo:

Fichero de Entrada:

```
2/101
5/412
9/111
10/923
```

Salida por pantalla: Se deberá generar un listado por pantalla con las siguientes líneas:

```
Número de entrada      2/101      Resultado    5
Número de entrada      5/412      Resultado  107
Número de entrada      9/111      Resultado   91
Número de entrada     10/923      Resultado  923
```

El contenido del fichero de salida será:

```
5  107  91  923
```

8. Escribe un programa que busque en un archivo de texto, que contiene números enteros separados por un espacio, los valores máximo y mínimo y los muestre por pantalla.

9. Escribe un programa que calcule el número de líneas que han de leerse de un archivo de texto hasta que se hayan leído del fichero todas las letras minúsculas del alfabeto al menos una vez. El programa debe mostrar también el número total de caracteres leídos hasta ese momento. Si se alcanza el final del fichero y no se consigue el objetivo, el programa mostrará un mensaje por pantalla que lo indique.

10. Escribe un programa que a partir de un fichero de texto, cree otro que sólo contenga aquellas líneas del primero en las que figure una palabra introducida previamente por el usuario desde el teclado.

NOTA: Suponer que la longitud máxima de una línea del fichero es 128 caracteres, que las palabras están separadas por espacios en blanco o saltos de línea y que los únicos caracteres que aparecen en el texto son letras, espacios en blanco y los caracteres que forman el salto de línea.

Relación complementaria (Tema 9)

11. Escribe un programa que mezcle dos archivos de texto, que contengan números enteros separados por espacios, y que estén ordenados ascendentemente. El fichero resultante deberá estar también ordenado en orden ascendente.

12. Escribe un programa que a partir de los nombres de dos ficheros de texto que contienen valores cardinales en el rango [1..1000] separados por espacios, genere dos nuevos ficheros de texto con la unión e intersección de los dos ficheros anteriores.

13. Se desea escribir un programa que resuelva el juego *Sopa d e Letras*. Para ello, se considerarán las siguiente especificaciones:

- La tabla de caracteres que forma la sopa de letras tendrá como dimensiones 15 filas por 15 columnas.
- En fichero de entrada "SOPA.DAT" contendrá las distintas líneas que forman la tabla.
- En fichero "PATRONES.DAT" contendrá las palabras a buscar dentro de la SOPA.
- Sólo será necesario buscar los patrones dentro de la SOPA por filas y por columnas en ambos sentidos (No será necesario buscar en las diagonales).

La salida del programa será un listado por pantalla mostrando cada palabra encontrada junto con las coordenadas donde fue hallada.

14. Mejora el programa anterior para que busque también en diagonales

15. Se desea ordenar un fichero de números enteros que no cabe en la memoria central del ordenador. Se opta por ordenarlo considerándolo como un array en disco. Para ello será necesario escribir los siguientes subprogramas:

1) `PROCEDURE Longitud (NombreFich : NOMBREFICH) : LONGCARD;` Devuelve la cantidad de números enteros almacenada en el fichero que toma como argumento.
2) `PROCEDURE LeerFich (NombreFich : NOMBREFICH; Posicion : LONGCARD) : INTEGER;` Devuelve el valor del número que está situado en la posición indicada dentro del fichero.
3) `PROCEDURE EscribirFich (NombreFich : NOMBREFICH; Posicion : LONGCARD; Valor : INTEGER);` Modifica el contenido del fichero, de modo que el valor del número que ocupa la posición indicada pasa a ser `Valor`. El resto de los números en el fichero permanecen iguales.

Utilizando estos subprogramas, diseña el programa que ordene el fichero y muestre el resultado por pantalla.

16. Otra alternativa al procedimiento anterior es dividir el fichero inicial en subficheros de como máximo 100 bytes. Cada uno de estos ficheros se carga en memoria en un vector de tamaño 50, se ordena en el vector y se vuelca en el correspondiente subfichero. Posteriormente se mezclan los subficheros obteniéndose el fichero original ordenado. Escribe un programa para esto.

Nota: Los problemas 14, 15 y 16 presentan un alto nivel de dificultad.