PRACTICA 1c

Elementos de programación Fortran: Entrada/salida por archivos y formatos.

```
Archivo no encontrado...  ¿Falsifico?
                                (S/N)
```

Entrada/salida por archivos. Hasta ahora hemos asumido que los datos que necesita un programa han sido entrados por teclado durante la ejecución del programa y que los resultados son mostrados por pantalla. Es claro que esta forma de proceder es adecuada sólo si la cantidad de datos de entrada/salida es relativamente pequeña. Para problemas que involucren grandes cantidades de datos resulta más conveniente que los mismos estén guardados en *archivos*. En lo que sigue veremos las instrucciones que proporciona Fortran para trabajar con archivos.

Un archivo es un conjunto de datos almacenado en un dispositivo (tal como un disco rígido) al que se le ha dado un nombre. Para la mayoría de las aplicaciones los únicos tipos de archivos que nos interesa considerar son los *archivos de texto*. Un archivo de texto consta de una serie de líneas o *registros* separadas por una marca de fin de línea (*newline*, en inglés). Cada línea consta de uno o más *datos* que es un conjunto de caracteres alfanuméricos que, en el procesamiento de lectura o escritura, se trata como una sola unidad. El acceso a los datos del archivo de texto procede en forma *secuencial*, esto es, se procesan línea por línea comenzando desde la primera línea hacia la última. Esto implica que no es posible acceder a una línea específica sin haber pasado por las anteriores.

Para fijar ideas consideremos el problema de calcular el baricentro de un conjunto de N puntos (x_i, y_i) del plano. Esto es, queremos computar las coordenadas del baricentro, definidas por

$$\bar{x} = \frac{\sum_{i=1}^{N} x_i}{N}, \qquad \bar{y} = \frac{\sum_{i=1}^{N} y_i}{N}.$$

Supongamos que las coordenadas de los N puntos se encuentran dispuestas en un archivo llamado `coordenadas.dat` que consta de una línea por cada punto, cada una de las cuales tiene dos columnas: la primera corresponde a la coordenada x y la segunda a la coordenada y. Así, este archivo consiste de N líneas, cada una de los cuales consta de dos datos de tipo real. Por otra parte, supondremos que el resultado $(\bar{x}, \bar{y})$ se quiere guardar en un archivo denominado `baricentro.sal`. El siguiente código Fortran efectúa lo pedido.

Código 1. Cálculo del baricentro de un conjunto de puntos en el plano

```
PROGRAM baricentro
  ! ----------------------------
  ! Declaración de variables
  ! ----------------------------
  IMPLICIT NONE
  INTEGER :: n, i
  REAL :: x, y, bar_x, bar_y
  ! ----------------------------
  ! Leer el número de puntos
  ! ----------------------------
  WRITE(*,*) 'Número de puntos?'
  READ (*,*) n
  ! ----------------------------
  ! Abrir archivo de datos
  ! ----------------------------
  OPEN(UNIT=8,FILE='coordenadas.dat', &
       & ACTION='READ')
  ! ----------------------------
  ! Leer los datos y procesarlos
  ! ----------------------------
  bar_x = 0.0
  bar_y = 0.0
  DO i=1,n
     READ(8,*) x, y
     bar_x = bar_x + x
     bar_y = bar_y + y
  END DO
  ! ----------------------------
  ! Cerrar el archivo de datos
  ! ----------------------------
  CLOSE(8)
  ! ----------------------------
  ! Calcular las coordenadas
  ! del baricentro
  ! ----------------------------
  bar_x = bar_x/n
  bar_y = bar_y/n
  ! ----------------------------
  ! Abrir archivo de salida
  ! ----------------------------
  OPEN(UNIT=9,FILE='baricentro.sal', &
       & ACTION='WRITE')
  ! ----------------------------
  ! Imprimir resultados en el
  ! archivo de salida
  ! ----------------------------
  WRITE(9,*) 'Baricentro:'
  WRITE(9,*) 'x = ', bar_x
  WRITE(9,*) 'y = ', bar_y
  ! ----------------------------
  ! Cerrar archivo de salida
  ! ----------------------------
  CLOSE(9)
  ! ----------------------------
END PROGRAM baricentro
```

En base a este programa podemos detallar las características básicas de la entrada/salida por archivos.

☞ En Fortran, un programa referencia indirectamente a un archivo a través de un *número de unidad lógica* (*lun*,

del inglés *logic unit number*), el cual es un entero positivo pequeño (pero distinto de 5 y 6 pues estas unidades están *pre-conectadas* a la entrada y salida estándar por teclado y pantalla, respectivamente). Así, para trabajar con un archivo, el primer paso consiste en establecer la relación entre el nombre del archivo y una unidad lógica. Esta conexión se realiza con la sentencia `OPEN` y el proceso se conoce como *abrir* el archivo (en nuestro ejemplo, el número de unidad 8 es asignado al archivo `coordenadas.dat`, mientras que el número de unidad 9 es asignado al archivo `baricentro.sal`). Nótese que excepto por esta sentencia, los archivos son referidos dentro del programa a través de su unidad lógica y *no* por su nombre.

☞ Para poder leer o escribir datos de una línea del archivo, conectado a la unidad *número*, Fortran utiliza las sentencias `READ` y `WRITE`, respectivamente, en la forma

```
READ(número,*) variables
WRITE(número,*) variables
```

Cada dato tiene su correspondiente variable del tipo apropiado en la lista de variables.

En nuestro ejemplo la lectura procede en un bucle `DO` desde el inicio hasta el final del archivo, avanzando línea por línea en cada lectura y asignando, cada vez, los dos datos del registro en sendas variables. Esta operación de lectura se comprende mejor introduciendo el concepto de *posición actual de línea*. A medida que el bucle `DO` se ejecuta imaginemos que un *puntero* se mueve a través de las líneas del archivo de modo que la computadora conoce de cual línea se deben leer los datos. Comenzando con la primera línea, la primer sentencia `READ` asigna a `x` e `y` los dos datos de dicha línea y luego *mueve el puntero a la siguiente línea*. Así, la segunda vez que la sentencia `READ` es ejecutada los datos de la segunda línea son asignados a las variables `x` e `y`. El proceso se repite N veces hasta alcanzar el final del archivo. De manera similar, cada vez que se ejecuta una sentencia de escritura `WRITE` se comienza en una nueva línea en el archivo.

☞ Así como un programa debe ser abierto para poder trabajar con el mismo, una vez finalizada la lectura o escritura el archivo debe ser *cerrado*, esto es, debe terminarse la conexión existente entre el archivo y la unidad lógica respectiva. Esta operación se realiza con la sentencia `CLOSE` seguida por el número de unidad entre paréntesis. Constituye una buena práctica de programación cerrar el archivo tan pronto no se necesita más. Nótese que mientras un archivo esté abierto su número de unidad no debe ser utilizado para abrir otro archivo. Sin embargo, una vez cerrado un archivo, el número de unidad correspondiente puede ser reutilizado. Por otra parte, un archivo que ha sido cerrado puede ser nuevamente abierto con una sentencia `OPEN`. Todos los archivos que no han sido cerrados explícitamente con una sentencia `CLOSE` serán cerrados automáticamente cuando el programa termine (salvo que un error aborte el programa).

☞ Los números de unidades son un recurso *global*. Un archivo puede ser abierto en cualquier unidad del programa, y una vez abierto las operaciones de entrada/salida pueden ser realizadas por cualquier unidad del programa (en tanto se utilice el mismo número de unidad). Nótese que los números de unidades puede ser guardados en variables enteras y ser pasados a un subprograma como argumento. Por otra parte, un programa resultará más modular si en lugar de utilizar directamente los números de unidad en las sentencias de entrada/salida se utilizan constantes con nombres (esto es, parámetros) para referirnos a los mismos.

☞ Desde el punto de vista del programa los archivos se utilizan o bien para *entrada*, o bien para *salida*. Un archivo es de entrada cuando el programa lee datos del mismo para usarlos (como el archivo `coordenadas.dat` en nuestro ejemplo), y es de salida cuando los resultados son escritos en él (como el archivo `baricentro.sal`). La cláusula `ACTION` en la sentencia `OPEN` permite indicar si el archivo será tratado como un archivo de entrada o bien de salida. Específicamente, `ACTION='READ'` indica que el archivo será abierto para lectura, con lo que cualquier intento de escribir sobre el mismo producirá un error. Esta cláusula es entonces apropiada para un archivo de entrada. Por el contrario, `ACTION='WRITE'` indica que el archivo será abierto para escritura solamente, con lo que un intento de lectura sobre el mismo conduce a un error. Esta cláusula es apropiada para un archivo de salida.

☞ Por otra parte, es claro que para que un programa funcione correctamente los archivos de entrada deben existir previamente a la ejecución del mismo. Esto puede controlarse fácilmente utilizando las cláusulas `STATUS` y `IOSTAT` en la sentencia `OPEN`:

```
OPEN ( UNIT=número, &
      & FILE='nombre del archivo', &
      & ACTION='READ', &
      & STATUS='OLD', &
      & IOSTAT=variable entera)
```

La cláusula `STATUS='OLD'` indica que el archivo a abrirse debe existir previamente. Por otra parte la asignación de una variable entera en la cláusula correspondiente a `IOSTAT`[1] permite discernir si el archivo fue abierto o no, puesto que la variable entera tomará el valor cero si el archivo se abrió exitosamente o un valor positivo si hubo un error (en este caso, el error es que el archivo no existe). Esto permite implementar una estructura de selección para manejar el error que puede producirse:

```
IF (variable entera /= 0) THEN
 WRITE(*,*) 'El archivo no puede ser leído'
 STOP
ENDIF
```

☞ En el caso de un archivo de salida, éste puede o no existir previamente. Si, en el caso que exista, nos interesa *no* sobreescribir el archivo, entonces la clásula apropiada es `STATUS='NEW'`. De este modo, si el archivo existe previamente la sentencia `OPEN` generará un error y con ello evitaremos sobreescribir los datos que contenga el archivo. Si el archivo no existe, entonces la sentencia `OPEN` lo creará (estando vacío hasta que se escriba algo en él). Nótese que aún en este caso puede un error que impida crear el archivo, por ejemplo, por falta de permisos adecuados. Nuevamente la cláusula `IOSTAT` nos permite manejar estas situaciones de error.

[1] El nombre `IOSTAT` hace referencia a *Input/Output status*, esto es estado de entrada/salida.

```
OPEN (  UNIT=número, &
      & FILE='nombre del archivo', &
      & ACTION='WRITE', &
      & STATUS='NEW', &
      & IOSTAT=variable entera)
IF (variable entera /= 0) THEN
 WRITE(*,*) 'El archivo no puede &
             & ser escrito'
 STOP
ENDIF
```

Si, por el contrario, no nos importa preservar el archivo de salida original en el caso que exista, entonces la clásula apropiada es `STATUS='REPLACE'`.

```
OPEN (  UNIT=número, &
      & FILE='nombre del archivo', &
      & ACTION='WRITE', &
      & STATUS='REPLACE', &
      & IOSTAT=variable entera)
IF (variable entera /= 0) THEN
 WRITE(*,*) 'El archivo no puede &
             & ser escrito'
 STOP
ENDIF
```

☞ Una situación que se presenta muchas veces es la necesidad de leer un archivo de entrada cuyo número de líneas es arbitrario (esto es, no está fijado de antemano por el problema). Bajo esta circunstancia un bucle `DO` no resulta adecuado. Para implementar la alternativa (un bucle `DO WHILE`) se necesita disponer de una forma de detectar el final del archivo conforme éste se va recorriendo. Esto puede lograrse agregando la cláusula `IOSTAT` a la sentencia de lectura. La correspondiente variable entera asignada tomará el valor cero si la lectura se realizó sin error, un valor positivo si se produjo un error (por ejemplo, intentar leer un dato de tipo distinto al que se está considerando) y un valor negativo si el final del archivo es encontrado. Utilizando un contador para el número de datos leídos y el bucle `DO WHILE` podemos resolver el problema con el siguiente fragmento de código.

```
n  = 0
io = 0
DO WHILE(io >= 0)
 READ(número,*,IOSTAT=io) variables
 IF(io == 0) THEN
   n = n+1
   procesar variables leídas
  ENDIF
ENDDO
```

Aquí `io` es una variable entera para controlar los errores de lectura, mientras que `n` es un contador entero cuyo valor final se corresponde con el número de datos válidos leidos.

Ejercicio 1. Modificar el código 1 parametrizando los números de unidades lógicas y permitiendo que el usuario asigne, por teclado, los nombres de los archivos de entrada y salida. (*Ayuda*: los nombres de los archivos pueden ser asignados a variables carácter).

Ejercicio 2. Modificar el código anterior para controlar que el archivo de entrada exista previamente y que el archivo de salida puede ser escrito. Detener el programa con un mensaje apropiado si la correspondiente situación no se cumplen.

Tabla 1. Descritores de formatos más comunes.

Descriptor de formato	Uso
Iw ó I$w.m$	Dato entero
F$w.d$	Dato real en notación decimal
E$w.d$	Dato real en notación científica
A ó Aw	Dato carácter
'$x\ldots x$'	Cadena de caracteres
nX	Espaciado horizontal

w: constante positiva entera que especifica el ancho del campo.
m: constante entera no negativa que especifica el mínimo número de dígitos a leer/mostrar.
d: constante entera no negativa que especifica el número de dígitos a la derecha del punto decimal.
x: un carácter.
n: constante entera positiva especificando un número de posiciones.

Ejercicio 3. Remover en el programa anterior la lectura del número de puntos N y reescribir la sección de lectura de manera que sea el propio programa quien determine el número de datos que lee.

Formatos. Otro aspecto a considerar en la entrada/salida de datos (ya sea por archivos o por teclado/pantalla) es la forma de estos datos. Hasta el momento hemos dejado que el compilador escoja automáticamente el formato apropiado para cada tipo de dato. Sin embargo es posible dar una forma precisa de la representación de los datos con una *especificación de formato* la cual consiste en una cadena de caracteres de la forma `'(` *lista de especificaciones de formatos* `)'`. Cada conjunto de especificación de formato consiste de un *descriptor de edición* de una letra, dependiente del tipo de dato, un tamaño de campo y una ubicación decimal si es pertinente. La tabla 1 muestra los descriptores de formatos de más utilidad. Para hacer efectiva esta especificación de formato se coloca la misma en la correspondiente sentencia de entrada/salida reemplazando al segundo carácter '*'.

Por ejemplo, para imprimir el resultado de nuestro programa ejemplo con las coordenadas con tres decimales, podemos escribir

```
WRITE(9,'(A)') 'Baricentro:'
WRITE(9,'(A,F7.3)') 'x = ', bar_x
WRITE(9,'(A,F7.3)') 'y = ', bar_y
```

☞ En la especificación de formato F$w.d$ de un dato real debe ser $w \geq d+3$ para contemplar la presencia del signo del número, el primer dígito y el punto decimal.

☞ Con la especificación de formato E$w.d$ de un dato real, éste es mostrado en forma *normalizada*, esto es, con

un signo menos (si es necesario) seguido de una parte entera consistente de un cero, el punto decimal, y una fracción decimal de d dígitos significativos, y una letra `E` con un exponente de dos dígitos (con su signo menos si es apropiado). Por lo tanto, debe ser $w \geq d + 7$.

☞ En la especificación de formatos de un dato real si el número de dígitos de la parte fraccionaria excede el tamaño asignado en la misma los dígitos en exceso serán eliminados *después* de redondear el número. Por ejemplo, la especificación `F8.2` hace que el número 7.6543 se escriba como 7.65, mientras que el número 3.9462 se escribe como 3.95.

☞ En el descriptor `A` para variables carácter si el ancho del campo no se especifica se considera que el mismo es igual a la longitud de la variable especificada en su declaración de tipo.

☞ La especificación n`X` hace que n posiciones no se tomen en cuenta en la lectura o bien que se llenen n espacios en blanco en la escritura.

☞ Cuando varios datos tienen las mismas especificaciones de formato, puede repetirse una sola especificación colocando un número entero frente a ella. Encerrando un conjunto de especificaciones entre paréntesis y colocando un número frente al conjunto indicamos que dicho conjunto será repetido tal número de veces.

☞ Si la especificación de formato es demasiado pequeña para el dato que se quiere procesar el campo será llenado con asteriscos. ¡Esta es una situación que suele ser muy desconcertante!

☞ La especificación de formato puede ser almacenada en una variable o parámetro de tipo carácter y ser utilizada como tal en la correspondiente sentencia de lectura o escritura.

☞ En general para *leer* datos (ya sea por teclado o archivo) la lectura por lista sin formato es el método adecuado. Sólo si los datos tienen una forma específica predeterminada debe utilizarse la lectura con formato.

Ejercicio 4. En el ejercicio 15 de la Práctica 1b establezca un formato con tres decimales para la tabla que debe crearse.

Ejercicio 5. En cierto observatorio meteorológico el promedio de la temperatura exterior en un día es anotada en un archivo día por día con el formato:

$$yyyy\text{-}mm\text{-}dd{:}\pm xxx\,.\,xx$$

Esto es, la fecha es descrita en el formato año-mes-día, con el año descrito con cuatro dígitos y el mes y el día con dos dígitos cada uno; la temperatura, por su parte, es un dato real con tres dígitos para la parte entera y dos dígitos para la parte decimal. Los dos datos están separados por el signo de dos puntos. Realizar un programa para crear un nuevo archivo en base al anterior pero con la fecha descrita en el formato día/mes/año.

Ejercicio 6. Si la especificación de formato no es adecuada para los datos considerados, entonces los mismos pueden ser mostrados incorrectamente. Pruebe los siguientes ejemplos:

```
WRITE(*,'(I4, 2X, I4)') 12, 12345
WRITE(*,'(F6.2)') 0.12345
WRITE(*,'(F6.2)') 123.45
WRITE(*,'(F6.2)') 12345.0
```

Ancho ajustable en los descriptores I y F

Con el fin de utilizar el menor espacio posible en la salida de datos, los descriptores de formato `I` y `F` permiten especificar un ancho w igual a cero. Esto no sólo elimina todo espacio en blanco precedente al dato, sino que también evita el desbordamiento con asteriscos cuando la especificación del ancho del campo es pequeña para el mismo.