Como utilizar MODULES en Fortran 90

Seminario de computación 2009

¿Qué es un MODULE?

Los modules son una forma eficiente de intercambiar información entre diferentes programas y subprogramas.

También permiten agrupar funciones y subrutinas que operan sobre los mismos datos en paquetes o “tool boxes” que pueden ser facilmente utilizados por diferentes programas.

SINTAXIS:

```
MODULE module-name
 IMPLICIT NONE
 [specification part]
 CONTAINS
 [internal-functions]
 END MODULE module-name
```

La sintaxis de los MODULES es similar a la de los programas, pero en un module no puede haber sentencias ejecutables fuera de las subrutinas o funciones, un module no puede existir por si solo, necesita estar asociado a un programa para poder funcionar.

Podemos utilizar modules para intercambiar datos entre diferentes unidades de un mismo programa. Por ejemplo entre diferentes subrutinas, sin la necesidad de estar pasándolos como argumentos.

Definimos un MODULE donde definimos el valor de diferentes constantes útiles.

```
MODULE CONSTANTES

SAVE

!DEFINO EL VALOR DE CIERTAS CONSTANTES QUE VOY A
!NECESITAR EN DIFERENTES SUBRUTINAS.
REAL, PARAMETER :: G=9.81
REAL, PARAMETER :: OMEGA=7.29E-5
REAL, PARAMETER :: RGAS=287.0
REAL, PARAMETER :: CP=1004.0
REAL, PARAMETER :: P0=1000.
REAL, PARAMETER :: PI=3.14159
REAL, PARAMETER :: DEG_TO_RAD=PI/180.

END MODULE CONSTANTES
```

Este código lo guardamos como ejemplo_mod_mod.f90 Vamos a utilizar este módulo para intercambiar el valor de diferentes constantes que intervienen en los cálculos en las subrutinas de un programa, sin que sea necesario incluirlas en los argumentos de entrada de las mismas.

¿Cómo usar un MODULE?

Una vez que escribimos un MODULE, las variables que definamos en el y las subrutinas y funciones que contenga pueden ser usadas por otros MODULES o por un programa.

Para que un programa u otro MODULE pueda utilizar el contenido de un MODULE utilizamos la sentencia USE.

SINTAXIS:

```
USE nombre_del_module    !En este caso todas las variables, funciones y subrutinas contenidos en el module están disponibles para el programa.
```

Si solo queremos usar algunas de las variables o procedimientos definidos en el module usamos

```
USE nombre_del_module ONLY: nombre1 , nombre2  !En este caso solo usamos aquellas variables o procedimientos llamados nombre1 y nombre2 que están en el module.
```

La sentencia USE debe venir a continuación del nombre del programa, subrutina o module que quiere incorporar al MODULE que escribimos. Podemos incluir tantos MODULES como queramos:

```
USE nombre_del_module1

USE nombre_del_module2    !uso una sentencia USE por cada MODULE que quiero incorporar
```

```
PROGRAM PRUEBA_MOD
!INCLUYO LOS MODULOS QUE VOY A USAR EN EL PROGRAMA
USE CONSTANTES

IMPLICIT NONE
REAL :: LATITUD=-80.
REAL :: TEMPERATURA=289.0
REAL :: PRESION=1004.0
REAL :: F , TEMPERATURA_POTENCIAL

!EN ESTE PROGRAMA SE VAN A USAR 2 SUBRUTINAS,
!UNA PARA CALCULAR EL PARAMETRO DE CORIOLIS Y
!OTRA PARA CALCULAR LA TEMPERATURA POTENCIAL.

CALL CORIOLIS(LATITUD,F)

CALL POT_TEMP(TEMPERATURA,PRESION,TEMPERATURA_POTENCIAL)


WRITE(*,*)"LA TEMPERATURA POTENCIAL ES: ",TEMPERATURA_POTENCIAL
WRITE(*,*)"EL PARAMETRO DE CORIOLIS ES: ",F

STOP
END PROGRAM PRUEBA_MOD

SUBROUTINE CORIOLIS(LAT,PARAMETRO)
USE CONSTANTES
IMPLICIT NONE

REAL, INTENT(IN) :: LAT
REAL, INTENT(OUT):: PARAMETRO

PARAMETRO=2*OMEGA*SIN(LAT*DEG_TO_RAD)

END SUBROUTINE CORIOLIS

SUBROUTINE POT_TEMP(T,P,TP)
USE CONSTANTES
IMPLICIT NONE

REAL, INTENT(IN) :: T,P
REAL, INTENT(OUT):: TP

TP=T*((P/P0)**(-RGAS/CP))

END SUBROUTINE POT_TEMP
```

Utilizamos la sentencia USE para poder utilizar el valor de las constantes definidas en el module en las diferentes subrutinas que componen este programa.

De esta manera reducimos el número de argumentos que le pasamos a cada subrutina y evitamos tener que redefinir el valor de las constantes en cada subrutina que componen nuestro programa.

Además si quisiéramos modificar el valor de alguna de las constantes, solo tenemos que modificar el MODULE y automáticamente modificaremos su valor en todo el programa.

¿Cómo compilamos un programa que utiliza MODULES?

En general los **MODULES** se escriben en archivos aparte que también llevan la extensión f90. Los **MODULES** pueden ser escritos en el mismo archivo que contiene el fuente del programa que los utiliza, pero esto reduce algunas de las ventajas de los **MODULES** como la facilidad para intercambiarlos entre un programa y otro.

Supongamos que tenemos un **MODULE** escrito en el archivo mi_module.f90 y un programa que lo utiliza en el programa mi_programa.f90

Primero tenemos que compilar el **MODULE** y luego el programa.

Si utilizamos el compilador desde la consola podemos compilar ambos en el mismo comando de la siguiente manera:

```
f90  mi_module.f90 mi_programa.f90 –o mi_ejecutable.exe
```

Donde f90 es el nombre del compilador que estemos usando. (en nuestro caso utilizamos el compilador Digital Fortran, por lo que el nombre del comando sería df)

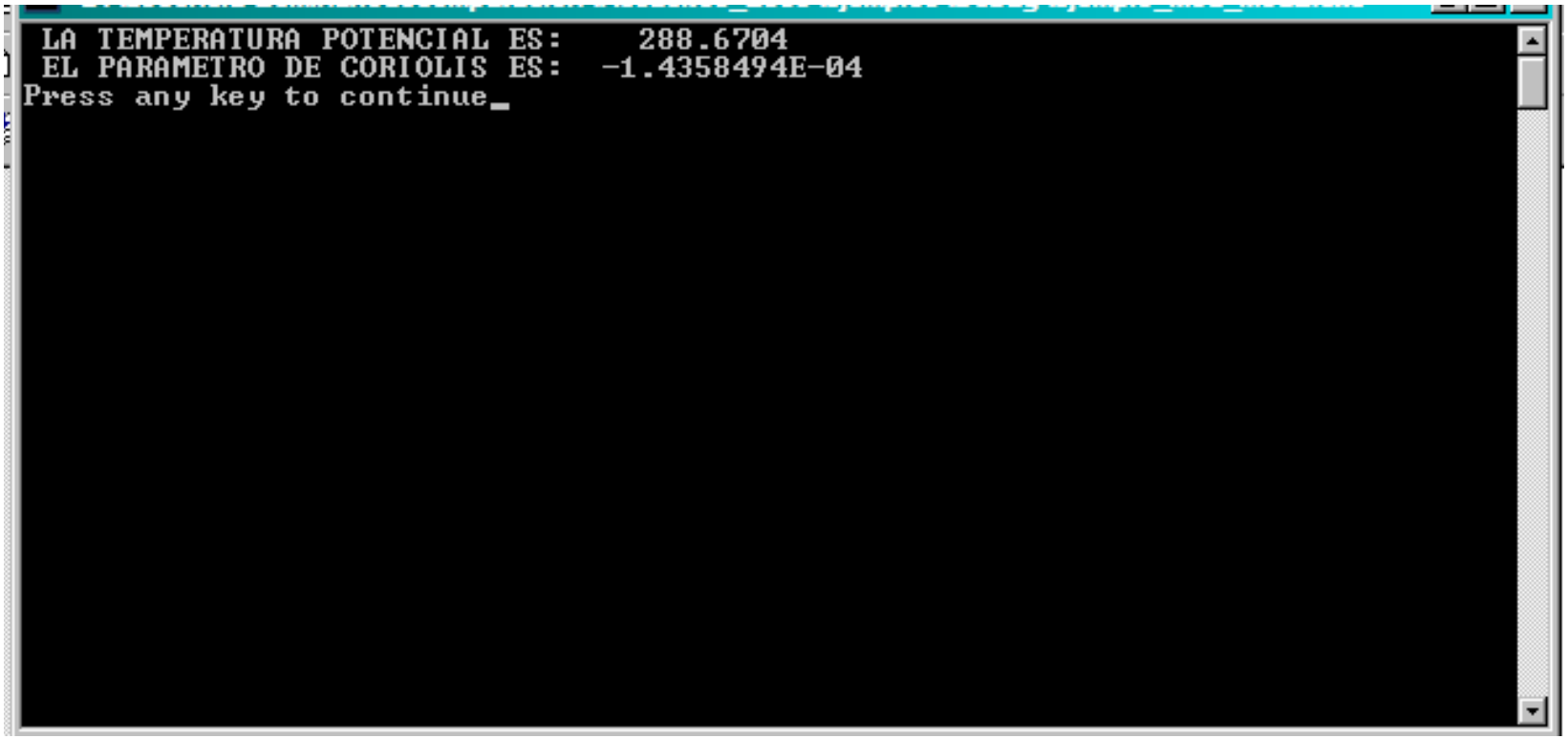
Si queremos compilar un programa que usa varios **MODULES** que a su vez se usan entre si, primero debemos compilar los **MODULES** que no dependen de otros **MODULES**, luego los **MODULES** que tienen alguna dependencia y finalmente el programa. El orden lo indicamos con el orden en el que ponemos los **MODULES** y el programa en la sentencia de compilación.

Para compilar nuestro ejemplo utilizando el Visual Digital Fortran hay que seguir los siguientes pasos:

Los fuentes del MODULE y del programa que lo utiliza los guardamos en la misma carpeta.

Primero compilamos el MODULE y a continuación compilamos y linkeamos el programa principial.

Al ejecutarlo el resultado debería ser:



Como vimos previamente, los **MODULES** también se pueden utilizar para almacenar subrutinas

¿Cuál es la ventaja de colocar las subrutinas de un programa dentro de un **MODULE**?

La ventaja es que cuando colocamos una subrutina dentro de un **MODULE** al momento de la compilación se realizarán chequeos que verifiquen que las subrutinas están siendo llamadas con el número y tipo correcto de argumentos evitando de esta manera errores frecuentes.

Armar un **MODULE** con subrutinas y datos de uso frecuente puede ser además una manera prolija y eficiente de intercambiar código entre diferentes programas.

A continuación vemos un ejemplo en donde además de algunas constantes útiles colocamos en un **MODULE** algunas subrutinas.

```
MODULE CONSTANTES2

SAVE

!DEFINO EL VALOR DE CIERTAS CONSTANTES QUE VOY A
!NECESITAR EN DIFERENTES SUBRUTINAS.
REAL, PARAMETER :: G=9.81
REAL, PARAMETER :: OMEGA=7.29E-5
REAL, PARAMETER :: RGAS=287.0
REAL, PARAMETER :: CP=1004.0
REAL, PARAMETER :: P0=1000.
REAL, PARAMETER :: PI=3.14159
REAL, PARAMETER :: DEG_TO_RAD=PI/180.

CONTAINS

SUBROUTINE CORIOLIS(LAT,PARAMETRO)
IMPLICIT NONE

REAL, INTENT(IN) :: LAT
REAL, INTENT(OUT):: PARAMETRO

PARAMETRO=2*OMEGA*SIN(LAT*DEG_TO_RAD)

END SUBROUTINE CORIOLIS

SUBROUTINE POT_TEMP(T,P,TP)
IMPLICIT NONE

REAL, INTENT(IN) :: T,P
REAL, INTENT(OUT):: TP

TP=T*((P/P0)**(-RGAS/CP))

END SUBROUTINE POT_TEMP


END MODULE CONSTANTES2
```

En este ejemplo además de haber incorporado algunas constantes, incorporamos algunos procesos que pueden ser llamados desde cualquier programa o subprograma que tenga acceso a este módulo.

El programa principal queda de esta forma….

```
PROGRAM PRUEBA_MOD
!INCLUYO LOS MODULOS QUE VOY A USAR EN EL PROGRAMA
USE CONSTANTES2

IMPLICIT NONE
REAL :: LATITUD=-80.
REAL :: TEMPERATURA=289.0
REAL :: PRESION=1004.0
REAL :: F , TEMPERATURA_POTENCIAL

!EN ESTE PROGRAMA SE VAN A USAR 2 SUBRUTINAS,
!UNA PARA CALCULAR EL PARAMETRO DE CORIOLIS Y
!OTRA PARA CALCULAR LA TEMPERATURA POTENCIAL.

CALL CORIOLIS(LATITUD,F)

CALL POT_TEMP(TEMPERATURA,PRESION,TEMPERATURA_POTENCIAL)


WRITE(*,*)"LA TEMPERATURA POTENCIAL ES: ",TEMPERATURA_POTENCIAL
WRITE(*,*)"EL PARAMETRO DE CORIOLIS ES: ",F

STOP
END PROGRAM PRUEBA_MOD
```

Otro ejemplo: Un module que contenga funciones trigonométricas para usar con grados en lugar de radianes.

```
MODULE  MyTrigonometricFunctions
   IMPLICIT   NONE

   REAL, PARAMETER :: PI         = 3.1415926        ! some constants
   REAL, PARAMETER :: Degree180 = 180.0
   REAL, PARAMETER :: R_to_D     = Degree180/PI
   REAL, PARAMETER :: D_to_R     = PI/Degree180
CONTAINS
! ------------------------------------------------------------------
   REAL FUNCTION  RadianToDegree(Radian)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: Radian
      RadianToDegree = Radian * R_to_D
   END FUNCTION  RadianToDegree
! ------------------------------------------------------------------
   REAL FUNCTION  DegreeToRadian(Degree)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: Degree
      DegreeToRadian = Degree * D_to_R
   END FUNCTION  DegreeToRadian
! ------------------------------------------------------------------
   REAL FUNCTION  MySIN(x)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: x

      MySIN = SIN(DegreeToRadian(x))
   END FUNCTION  MySIN
! ------------------------------------------------------------------
   REAL FUNCTION  MyCOS(x)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: x

      MyCOS = COS(DegreeToRadian(x))
   END FUNCTION  MyCOS
END MODULE  MyTrigonometricFunctions
```

Programa principal que utiliza el modulo que me permite utilizar funciones trigonométricas que toman el argumento en grados en lugar de radianes.

```
PROGRAM  Trigontest
   USE  MyTrigonometricFunctions          ! usar un module
   IMPLICIT  NONE
   REAL :: Begin = -180.0                 ! valor inicial
   REAL :: Final =  180.0                 ! valor final
   REAL :: Step  =   10.0                 ! intervalo
   REAL :: x

   WRITE(*,*)  'Value of PI = ', PI
   WRITE(*,*)
   x = Begin                              ! empezar en el valor inicial
   DO
      IF (x > Final)  EXIT                ! si X es mayor que el valor final EXIT
      WRITE(*,*)  'x = ',  x, 'deg   sin(x) = ', MySIN(x), &
                  '   cos(x) = ', MyCOS(x)
      x = x + Step                        ! pasamos al siguiente valor de X
   END DO

END PROGRAM  Trigontest
```

Parte de la salida del programa anterior sería:

Value of PI = 3.1415925
x = -180.deg sin(x) = 8.742277657E-8 cos(x) = -1.
x = -170.deg sin(x) = -0.173648298 cos(x) = 0.98480773
x = -160.deg sin(x) = -0.342020214 cos(x) = -0.939692616
x = -150.deg sin(x) = -0.50000006 cos(x) = -0.866025388
x = -140.deg sin(x) = -0.642787635 cos(x) = -0.766044438
x = -130.deg sin(x) = -0.766044438 cos(x) = -0.642787635
x = -120.deg sin(x) = -0.866025388 cos(x) = -0.50000006

Los programas “grandes” modelos numéricos, etc están armados por **MODULES**:

Los diferentes procesos en los que se divide el programa, están agrupados por **MODULES** según el tipo de tareas que cumplen. Así tenemos **MODULES** que se encargan de la lectura y escritura de datos, **MODULES** que se encargan de las parametrizaciones, etc.

Cada **MODULE** a su vez está compuesto por una serie de subrutinas que cumplen tareas específicas, leer un archivo, resolver un sistema lineal, calcular advecciones, etc.

En la práctica se propone el armado de un **MODULE** con subrutinas para calcular estadísticos básicos (media, desvío, correlación, etc.).

Dentro de un MODULE se pueden definir muchas variables y funciones, en ocasiones algún nombre de los definidos dentro de un module podría entrar en conflicto con los definidos en el programa que utiliza dicho MODULE. Para evitar estos problemas existen sentencias que nos permiten controlar que variables, funciones y subrutinas van a ser “visibles” para el programa que utiliza el MODULE y cuales no.

PUBLIC / PRIVATE

SINTAXIS:

PUBLIC :: name1 , name2 , name3

PRIVATE :: name4 , name5 , name6

En este caso, las variables / procesos name1, name2 y name3 podrán ser accedidos por cualquier unidad que utilice el MODULE mediante la sentencia USE.

Mientras que las variables / procesos name4, name5, name6 solo podrán ser utilizadas dentro del MODULE pero no estarán disponibles para otras unidades que utilicen el MODULE.

Veamos un ejemplo sencillo…

```
MODULE  TheSimpsons
   IMPLICIT   NONE

   INTEGER :: Homero , Marge
   REAL    :: Lisa,Bart
   LOGICAL :: Ayudante_de_santa

   CHARACTER(len=4), PARAMETER :: Duff = "BURP"

   PUBLIC  :: Homero, Marge, Bart, Lisa
   PRIVATE :: Volumen_panza_homero , Largo_pelo_Marge
   PRIVATE :: Ayudante_de_santa

CONTAINS
   REAL FUNCTION  Volumen_panza_homero()
      ..........
   END FUNCTION Volumen_panza_homero

   REAL FUNCTION  Largo_pelo_Marge()
      ..........
   END FUNCTION


      ..........
END MODULE  TheSimpsons
```

Ahora veamos un ejemplo más serio… retomemos el ejemplo de las funciones seno y coseno para grados. Sería bueno que solo estuvieran disponibles las funciones en si, y no todas las constantes que usan las funciones para hacer los cálculos intermedios. Tampoco me interesa la función que pasa de grados a radianes y viceversa, solo la función final seno y coseno para usar con grados en lugar de radianes.

```
MODULE  MyTrigonometricFunctions
   IMPLICIT   NONE

   REAL, PARAMETER :: PI        = 3.1415926       ! some constants
   REAL, PARAMETER :: Degree180 = 180.0
   REAL, PARAMETER :: R_to_D    = Degree180/PI
   REAL, PARAMETER :: D_to_R    = PI/Degree180
   PRIVATE :: Degree180 , R_to_D , D_to_R , RadianToDegree , DegreeToRadian
   PUBLIC  :: MySIN , MyCOS
CONTAINS
! ------------------------------------------------------------------
   REAL FUNCTION  RadianToDegree(Radian)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: Radian
      RadianToDegree = Radian * R_to_D
   END FUNCTION  RadianToDegree
! ------------------------------------------------------------------
   REAL FUNCTION  DegreeToRadian(Degree)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: Degree
      DegreeToRadian = Degree * D_to_R
   END FUNCTION  DegreeToRadian
! ------------------------------------------------------------------
   REAL FUNCTION  MySIN(x)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: x

      MySIN = SIN(DegreeToRadian(x))
   END FUNCTION  MySIN
! ------------------------------------------------------------------
   REAL FUNCTION  MyCOS(x)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: x

      MyCOS = COS(DegreeToRadian(x))
   END FUNCTION  MyCOS
END MODULE  MyTrigonometricFunctions
```

Veamos un ejemplo del uso de **MODULES** para intercambiar datos entre las subrutinas y el programa principal.

Vamos a ver una forma de calcular la desviación estándar y la media de una serie con datos faltantes (como en el ejercicio 8 de la práctica 3).

```
MODULE ESTADISTICA
IMPLICIT NONE
SAVE
REAL                              :: UNDEF=-999.
REAL, ALLOCATABLE, DIMENSION(:)   :: SERIE_SIN_UNDEF
INTEGER                           :: N_UNDEF=0 , N_SERIE=0

CONTAINS
!=====================================================================
SUBROUTINE SACA_UNDEF(SERIE_IN,LONGITUD)
IMPLICIT NONE
INTEGER, INTENT(IN):: LONGITUD
REAL, INTENT(IN), DIMENSION(LONGITUD) :: SERIE_IN

!CONTAMOS LA CANTIDAD DE ELEMENTOS UNDEF Y NO UNDEF
N_UNDEF=COUNT(SERIE_IN==UNDEF)
N_SERIE=LONGITUD-N_UNDEF

!GUARDAMOS ESPACIO EN LA MEMORIA PARA LA NUEVA SERIE.
ALLOCATE(SERIE_SIN_UNDEF(N_SERIE))

!NOS QUEDAMOS SOLO CON LOS ELEMENTOS DE LA SERIE QUE NO SON UNDEF
SERIE_SIN_UNDEF=PACK(SERIE_IN,SERIE_IN/=UNDEF)

END SUBROUTINE SACA_UNDEF
!=====================================================================
SUBROUTINE MEAN(PROMEDIO)
IMPLICIT NONE
REAL,INTENT(OUT)    :: PROMEDIO

PROMEDIO=SUM(SERIE_SIN_UNDEF)/REAL(N_SERIE)

END SUBROUTINE MEAN
!=====================================================================
SUBROUTINE STD(DESVIO)
IMPLICIT NONE

REAL,INTENT(OUT)    :: DESVIO

DESVIO=SQRT(SUM(SERIE_SIN_UNDEF**2)/REAL(N_SERIE)-(SUM(SERIE_SIN_UNDEF)/REAL(N_SERIE))**2)

END SUBROUTINE STD
!=====================================================================
END MODULE ESTADISTICA
```

```
PROGRAM EJEMPLO3
USE ESTADISTICA

IMPLICIT NONE
!DESARROLLO DE FUNCIONES ESTADISTICAS PARA APLICAR A UNA SERIE TEMPORAL
!EN ESTE PUNTO COMO TODAVIA NO SE INTRODUJERON LOS CONCEPTOS BASICOS DE I/O
!LA SERIE SE GENERA COMO UNA SERIE DE NUMEROS ALEATORIOS.

INTEGER, PARAMETER :: LARGO_SERIE=1000
REAL   :: SERIE(LARGO_SERIE)
INTEGER :: I,J
REAL    :: PROMEDIO,DESVIO,TEMP

!LA SERIE LA GENERAMOS COMO UNA SUCESION DE NUMEROS ALEATORIOS.
! UN 10% APROXIMADAMENTE TIENE QUE TENER DATOS FALTANTES (ESTO TAMBIEN LO HACEMOS CON NUMEROS ALEATORIOS).

CALL RANDOM_NUMBER(SERIE) !GENERAMOS UN ARRAY DE NUMEROS AL AZAR.
SERIE=SERIE*10.
DO I=1,LARGO_SERIE
   CALL RANDOM_NUMBER(TEMP)
   IF(TEMP > 0.9 )SERIE(I)=-999. !USAMOS OTRO NUMERO AL AZAR PARA DECIDIR SI EL DATO SERA O NO FALTANTE.
ENDDO

!EN ESTE PUNTO YA TENEMOS UNA SERIE DE 1000 ELEMENTOS CON APROXIMADAMENTE UN 10 % DE FALTANTES.

!LLAMO A LA SUBRUTINA SACA_UNDEF PARA QUE ELIMINE DE MI SERIE LOS VALORES UNDEF
CALL SACA_UNDEF(SERIE,LARGO_SERIE)

!UTILIZO LA SUBRUTINA MEAN QUE OPERA SOBRE LA SERIE SIN LOS VALORES UNDEF.
CALL MEAN(PROMEDIO)

!IDEM CON LA SUBRUTINA STD.
CALL STD(DESVIO)

WRITE(*,*)"EL PROMEDIO Y EL DESVIO SON:"
WRITE(*,*)"PROMEDIO : ", PROMEDIO
WRITE(*,*)"DESVIO   : ", DESVIO


END PROGRAM EJEMPLO3
```