Madrid
Julio 2004

Aprenda Fortran 8.0

como si estuviera en primero

Javier García de Jalón, Francisco de Asís de Ribera

Escuela Técnica Superior
de Ingenieros Industriales

Universidad Politécnica de Madrid

Aprenda Fortran 8.0 como si estuviera en primero

Javier García de Jalón
Francisco de Asís de Ribera

PRÓLOGO

La colección de manuales "Aprenda Informática como si estuviera en Primero" nació en la Escuela Superior de Ingenieros Industriales de San Sebastián (Universidad de Navarra) a lo largo de la década de 1990, como consecuencia de la impartición de las asignaturas Informática 1 e Informática 2, introducidas en el Plan de Estudios de 1993.

El objetivo de esta colección era facilitar a los alumnos de las asignaturas citadas unos apuntes breves y sencillos, fáciles de leer, que en unos casos ayudasen en el uso de las aplicaciones informáticas más habituales para un ingeniero industrial y en otros sirvieran de introducción a distintos lenguajes de programación.

Así pues, los destinatarios directos de estos apuntes eran los alumnos de la Escuela de Ingenieros Industriales de San Sebastián. Para facilitarles su uso, además de estar a la venta en el Servicio de Reprografía, se introdujeron versiones "online" en formato PDF (Portable Document Format, de Adobe), accesibles a través de las páginas Web de las mencionadas asignaturas. Los alumnos de cursos superiores y algunos profesores los utilizaban también para actualizar sus conocimientos cuando se instalaban nuevas versiones de las correspondientes aplicaciones.

Sin haberlos anunciado en ningún índice o buscador, al cabo de cierto tiempo se observó que eran accedidos con una frecuencia creciente desde el exterior de la Escuela, a través de Internet. Poco a poco empezaron a llegar de todo el mundo de habla hispana correos electrónicos que se interesaban por nuevos títulos, daban noticia de erratas, solicitaban permiso para utilizarlos en la docencia de otras instituciones o simplemente daban las gracias por haberlos puesto en Internet.

A la vista de estos efectos "no buscados", se estableció una página Web dedicada especialmente a esta colección y se anunció en los tres o cuatro portales más importantes de lengua española, lo que hizo que en poco tiempo se multiplicaran los accesos.

A partir del curso 2000-01 el autor principal y creador de la colección se trasladó a la Escuela Técnica Superior de Ingenieros Industriales de la Universidad Politécnica de Madrid, de la que es actualmente catedrático en el área de Matemática Aplicada. El principal punto de entrada a la colección se encuentra ahora en la dirección http://www.tayuda.com.. El número de accesos ha seguido aumentando, superando la cifra de 50.000 ficheros mensuales desde la primavera de 2001.

Aunque el mantenimiento de esta colección constituya un trabajo notable y no se saque ningún rendimiento económico de ella, da particular alegría el realizar un trabajo que tantos miles de personas consideran útil. El mantenimiento de estos manuales va a ser más difícil en los próximos años, en gran parte por el cambio en la actividad docente de su director o coordinador. Por eso serán bienvenidas todas aquellas ofertas de ayuda para mantener y crear esta colección de "Open Tutorials".

Madrid, marzo de 2005.

Javier García de Jalón de la Fuente
(jgjalon@etsii.upm.es)

ÍNDICE

1. INTRODUCCIÓN AL LENGUAJE FORTRAN 90

1.1 QUÉ ES FORTRAN 90

La palabra FORTRAN viene de FORmula TRANslation. Fortran fue el primer lenguaje científico de alto nivel utilizado en la historia de los computadores. La primera versión fue desarrollada para el IBM 704 por John Backus y colaboradores entre 1954 y 1957, pocos meses después apareció la versión llamada Fortran II. Poco a poco se empezaron a desarrollar versiones más o menos similares de Fortran para diversos computadores. En 1962 se presentó Fortran IV, que era casi por completo independiente del computador en el que se había de ejecutar. En 1962 se estableció un comité de ANSI (America Nacional Standard Institute) para definir un Fortran estándar, que estuvo muy basado en Fortran IV. Este estándar fue ratificado en 1966, y a veces se conoce como Fortran 66. En 1977 se publicó el borradord e un nuevo estándar que incorporaba los avances alcanzados en aquellos años. Este nuevo estándar fue publicado en 1978 con el nombre de Fortran 77. Finalmente, en 1991 se publicó un nuevo estándar, esta vez aprobado por la ISO (Internacional Standards Organization), que es el que se presenta en estos breves apuntes.

Fortran nació y se ha desarrollado como un lenguaje especializado en cálculos técnicos y científicos. Aunque las librerías matemáticas y numéricas existentes para Fortran son probablemente las más completas y eficientes, y aunque los compiladores de Fortran suelen producir el código ejecutable más rapido de todos, lo cierto es que el lenguaje Fortran ha ido perdiendo peso frente a lenguajes de propósito general como C/C++ que son hoy día muchísimo más utilizados. Fortran mantiene sin embargo una cierta importancia en ingeniería y métodos numéricos, y en muchos casos es la opción preferible. Fortran es más fácil de aprender que C/C++ y en las últimas versiones ha ido incorporando ideas de otros lenguajes más modernos.

1.1.1 Compiladores de Fortran 90

Existen diversos compiladores de Fortran. Una visión general de este tema se puede encontrar en la dirección http://www.lahey.com/other.htm. En este manual se ha utilizado el compilador Visual Fortran 8.0 de Intel, compilador basado en el que hace unos años comercializaba Microsoft y que luego pasó a Digital y a Compaq.

1.2 CARACTERES PERMITIDOS

Los caracteres permitidos en Fortran 90 son los siguientes:

0,1,2,3,4,5,6,7,8,9

A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z

a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z

',",(,),+,-,*,/, ,:,=,!,&,$,;,<,>,%,?,,,. (incluye blanco, punto, coma, y punto y coma)

Es importante tener en cuenta que Fortran no distingue entre mayúsculas y minúsculas.

1.3 PALABRAS RESERVADAS DE FORTRAN 90

Como todos los lenguajes de programación, Fortran 90 tiene algunas palabras reservadas, que los programadores deben abstenerse de utilizar:

Algunas palabras reservadas de Fortran 90 son las siguientes (no se incluyen las funciones de librería):

ALLOCATE, ALLOCATABLE
CALL, CASE, CHARACTER, COMPLEX, CONTAINS, CYCLE,
DEALLOCATE, DEFAULT, DIMENSION, DO,
END, ELSE, ELSEWHERE, EXIT, EXTERNAL
FUNCTION,
IF, IMPLICIT, IN, INOUT, INTEGER, INTENT, INTERFACE, INTRINSIC
KIND,
LOGICAL,
MODULE
NONE,
ONLY, OPEN, OUT,
PARAMETER, POINTER, PRINT, PROGRAM,
READ, REAL, RECURSIVE, RESULT, RETURN
SAVE, SELECT, SIZE, STAT, STOP, SUBROUTINE
TARGET, THEN, TYPE
UNIT, USE
WHERE, WRITE

1.4 CARACTERÍSTICAS GENERALES DE FORTRAN 90

Los **identificadores** son nombres para que se utilizan identificar programas, variables, constantes (simbólicas), y otras entidades. Los identificadores deben siempre empezar por una letra, y pueden ir seguidos por hasta 30 letras, dígitos y guión bajo (_).

En los identificadores no se distingue entre mayúsculas y minúsculas. Es bastante habitual escribir todo con mayúsculas las palabras clave del lenguaje (IF, DO, END, …). Para las ***variables*** es habitual utilizar minúsculas, con mayúscula para la primera letra. Con frecuencia se unen varias palabras poniendo la priomera letra en mayúscula o separádolas con un guión bajo.

Las ***líneas*** pueden tener hasta 132 caracteres. Una línea puede contener varias sentencias, separadas por el carácter punto y coma (;).

Las sentencias pueden llevar una ***etiqueta***, que debe ir al comienzo de la línea y que es una constante entera entre 1 y 99999. La etiqueta debe ir separada del resto de la sentencia por uno o más blancos.

Las ***líneas de comentario*** empiezan por el carácter (!). Se admiten comentarios al final de una sentencia ejecutable, pues el compilador ignora todo lo que aparece en una línea a continuación del carácter (!), excepto si este carácter aparece en el interior de una cadena de caracteres.

Para ***continuar*** una sentencia en la línea siguiente se termina con el carácter &. Se permiten hasta 39 líneas de continuación (40 líneas por sentencia). El carácter de continuación puede cortar una cadena de caracteres, pero en este caso es necesario poner otro carácter & al comienzo de la línea siguiente.

La sentencia STOP ***termina inmediatamente*** la ejecución de un programa. Opcionalmente se le puede añadir una cadena de caracteres que se imprime al ejecutar esta sentencia, o una cadena de hasta seis dígitos que sisve para identificar el error.

1.5 ESTRUCTURA GENERAL DE UN PROGRAMA FORTRAN 90

La primera y la última línea de un programa son respectivamente:

```
PROGRAM program_name
    ...
END PROGRAM program_name
```

Es habitual obligarse a declarar todas las variables que se vayan a utilizar poniendo cono segunda sentencia del programa:

```
IMPLICIT NONE
```

1.6 TIPOS DE VARIABLES

Fortran 90 soporta los siguientes 5 tipos de variables o constantes: INTEGER, REAL, COMPLEX, CHARACTER y LOGICAL.

Las variables se declaran en la forma:

```
INTEGER :: i, j, hora, minutos=60
REAL :: pi=3.141592654, velocidad, valorMedio=0.0
```

Es posible ***inicializar*** las variables en el momento de declararlas. Si no se inicializan contienen basura informática y su uso resulta imprevisible.

Las ***constantes reales*** siempre deben llevar un punto decimal (.), aunque su valor sea entero.

Las ***cadenas de caracteres*** van entre "—" ó '—'. El carácter de cierre debe ser el mismo que el de apertura. En una cadena definida con "-" el apóstrofo ' se considera como tal, es decir como un carácter más de la cadena.

Para las variables CHARACTER se puede declarar la ***longitud*** de la cadena que contendrán.

```
CHARACTER (LEN=20) :: nombre, primerApellido, segundoApellido
```

que es lo mismo que:

```
CHARACTER (20) :: nombre, primerApellido, segundoApellido
```

o que:

```
CHARACTER :: nombre*20, primerApellido*20, segundoApellido*20
```

También es posible definir una ***longitud general*** y establecer excepciones:

```
CHARACTER (20) :: nombre*30, primerApellido, segundoApellido, iniciales*3
```

Fortran 90 no obliga a declarar las variables que se van a utilizar. Existe desde la primera versión una ***convención de nombre implícita***, que establece que las variables que empiezan pos las letras I, J, K, L, M, N (ó i, j, k, l, m, n) son INTEGER. Si comienzan por cualquiera de las letras restantes son REAL. Sin embargo, para evitar errores es recomendable no utilizar esta característica del lenguaje, introduciendo la sentencia IMPLICIT NONE.

1.7 CONSTANTES SIMBÓLICAS

Las constantes simbólicas (constantes con nombre) se declaran de modo análogo que las constantes, introduciendo la palabra PARAMETER a continuación del tipo, separados por una coma:

```
INTEGER, PARAMETER :: nitMax=1000
```

En las constantes de tipo CHARACTER se puede definir la longitud implícitamente por medio del carácter (*):

```
CHARACTER (*), PARAMETER :: saludo="Hola", despedida="Adiós"
```

Las constantes simbólicas se emplean luego en las expresiones aritméticas exactamente igual que las variables.

1.8 OPERADORES ARITMÉTICOS

Los operadores aritméticos binarios son:

Suma (+), Resta (-), Multiplicación (*), División (/) y Potenciación (**)

Operadores unarios: Signo positivo (+) y signo negativo (-).

No se pueden poner nunca dos operadores consecutivos. Si es preciso se utilizan los paréntesis para separarlos.

1.9 EXPRESIONES ARITMÉTICAS

Las expresiones aritméticas se definen mediante una combinación de variables, constantes numéricas y simbólicas, paréntesis, llamadas a función y operadores aritméticos.

El ***orden de las operaciones aritméticas*** en una expresión es el siguiente:

1. Se evalúa en primer lugar el operador potencia (**). Si hay varios, se ejecutan de derecha a izquierda.
2. Se evalúan a continuación las multiplicaciones y divisiones. Si hay varios oparadores de este tipo se evaluan de izquierda a derecha.
3. Finalmente se evalúan las sumas y restas, también de izquierda a derecha.

Los ***paréntesis*** cambian el orden de realización de las operaciones: en primer lugar se realizan las operaciones del paréntesis más interior.

Las operaciones se realizan con el tipo de datos de los operandos. Por ejemplo, la suma de INTEGER es un INTEGER; la división de dos INTEGER es otro INTEGER, lo que quiere decir que se trunca el cociente.

Las ***operaciones mixtas*** (u operaciones en modo mixto) son operaciones entre variables o constantes de distinto tipo. Lo más freceuente es combinar INTEGER con REAL. En las operaciones mixtas el dato INTEGER se convierte antes en REAL, y el resultado es de tipo REAL. Por ejemplo, 5/2 es 2, mientras que 5/2.0 es 2.5.

Un caso particular interesante es el que concierne al ***operador potencia*** (**). Si el exponente es INTEGER el resultado se obtiene por multiplicación repetida. Por el contrario, si el exponente es de tipo REAL el resultado se obtiene mediante la aplicación de logaritmos; esto quiere decir que la base no puede ser un número negativo, porque los números negativos no tienen logaritmo.

La prioridad de los operadores unarios es la misma que la de los operadores binarios correspondientes.

1.10 FUNCIONES NUMÉRICAS

Fortran 90 dispone de funciones numéricas para evaluar las funciones matemáticas más habituales. Algunas de estas funciones se muestran en la **¡Error! No se encuentra el origen de la referencia.**, que muestra también el tipo del argumento (o de los argumentos) y del resultado o valor de retorno.

Hay otras funciones numéricas en Fortran, como por ejemplo las funciones hiperbólicas (SINH, COSH, TANH).

Todas las funciones numéricas se utilizan incluyendo su nombre, seguido de los argumentos entre paréntesis y separados por comas, en alguna expresión aritmética. La función se evalúa y el resultado se incluye en dicha expresión aritmética.

Función	**Significado matemático**	**Tipo argumento**	**Tipo resultado**
ABS(x)	Valor absoluto de x	INTEGER ó REAL	Como el argumento
COS(x)	Coseno de x (en radianes)	REAL	REAL
EXP(x)	Función exponencial	REAL	REAL
INT(x)	Parte entera de x	REAL	INTEGER
FLOOR(x)	Mayor entero ≤ x	REAL	INTEGER
FRACTION(x)	Mantisa de x	REAL	REAL
LOG(x)	Logaritmo natural de x	REAL	REAL
MAX(x1,…,xn)	Máximo de x1, x2, …xn	INTEGER ó REAL	Como el argumento
MIN(x1,…,xn)	Mínimo de x1, x2, …xn	INTEGER ó REAL	Como el argumento
MOD(x, y)	Resto de la división: x−INT(x/y)	INTEGER ó REAL	Como el argumento
NINT(x)	x redondeado al entero más próximo	REAL	INTEGER
REAL(x)	Se convierte x a REAL	INTEGER	REAL
SIN(x)	Seno de x (en radianes)	REAL	REAL
RANDOM_NUMBER(x)	Subrutina que genera nº aleatorio 0≤x<1	REAL	REAL
SQRT(x)	Raíz cuadrada de x	REAL	REAL
TAN(x)	Tangente de x (en radianes)	REAL	REAL
ACOS(x)	Arco coseno de x	REAL	REAL
ASIN(x)	Arco seno de x	REAL	REAL
ATAN(x)	Arco tangente de x	REAL	REAL
ATAN2(x,y)	Arco tangente de x/y (considera signos)	REAL	REAL

Tabla 1. Algunas funciones intrínsecas de Fortran 90 para operaciones numéricas.

1.11 OPERACIONES CON CARACTERES

Las cadenas de caracteres van entre apóstrofos o entre dobles comillas. El ***operador de concatenación*** es la doble barra //.

A partir de una variable de tipo cadena de caracteres se puede extraer una subcadena por medio del carácter (:). Por ejemplo si la variable **Especie** tiene como valor "rinoceronte", la subcadena **Especie(:4)** es "rino", **Especie(3:8)** es "nocero" y **Especie(9:)** es "nte".

Hay otras funciones para manejo de cadenas de caracteres, algunas de las cuales se muestran en la Tabla 2.

Función	Significado	Tipo argumento	Tipo resultado
ACHAR(I)	Carácter I del código ASCII	INTEGER	CHARACTER
ADJUSTL(str) ADJUSTR(str)	Se ajustan los caracteres por la izda (dcha), eliminando blancos y añadiéndolos por la dcha (izda)	CHARACTER	CHARACTER
CHAR(I, kind)	Carácter en posición I. El valor de retorno tiene la clase KIND especificada por el argumento opcional	INTEGER	CHARACTER
LEN(str)	longitud de una cadena de caracteres	CHARACTER	INTEGER
LEN_TRIM(str)	Longitud de una cadena sin contar los blancos	CHARACTER	INTEGER
LGE(str1, str2)	.TRUE. si str1 es lexicograficamente >= que str2	CHARACTER	LOGICAL
LGT(str1, str2)	.TRUE. si str1 es lexicograficamente > que str2	CHARACTER	LOGICAL
LLE(str1, str2)	.TRUE. si str1 es lexicograficamente <= que str2	CHARACTER	LOGICAL
LLT(str1, str2)	.TRUE. si str1 es lexicograficamente < que str2	CHARACTER	LOGICAL
REPEAT(str, n)	Formar una cadena resultado de concatenar str n veces	CHAR., INTEG.	CHARACTER
TRIM(str)	Elimina los blancos que van por delante	CHARACTER	CHARACTER
VERIFY(str, s, bk)	Chequea que una cadena contiene a otra, dando la posición del primer carácter que no pertenece a ella.	CHARACTER	INTEGER

Tabla 2. Funciones intrínsecas para el manejo de cadenas de caracteres.

1.12 SENTENCIAS DE ASIGNACIÓN

La forma general de las sentencias de asignación hace uso del operador (=) en la forma:

```
Variable = expresión
```

Si **Variable** es de tipo REAL y la expresión es de tipo INTEGER, el resultado de la expresión se convierte a REAL antes de hacer la asignación.

En el caso contrario de que el resultado de una expresión de tipo REAL deba ser asignado a una variable INTEGER, se trunca la parte decimal o fraccional de la expresión y la parte entera se asigna a la variable INTEGER.

Estas sentencias de asignación entre distintos tipos de variables, con cambios de tipo implícitos, se consideran una metodología de programación peligrosa. Se considera preferible incluir funciones de cambio de tipo explícito, tales como REAL() ó INTEGER().

También se puede utilizar sentencias de asignación con variables o constantes que representan cadenas de caracteres, por ejemplo en la forma:

```
Nombre = "Rigoberto"
```

Es posible realizar la asignación a unos caracteres concretos de la variable **Nombre** definiendo un rango por medio del ***operador dos puntos***, como por ejemplo:

```
Nombre(3:8) = "gobert"
```

También se podrían utilizar las formas **Nombre(:4)** y **Nombre (6:)**.

1.13 SENTENCIAS DE ENTRADA/SALIDA SENCILLAS

Fortran permite utilizar sentencias de entrada/salida muy sencillas, en las que casi todas las opciones son las de defecto. Existen otras sentencias de entrada/salida más complejas que se verán en un capítulo posterior. Las primeras están ***controladas por la lista de variables*** a leer o escribir. Las segundas tienen ***formatos específicos*** que controlan todos los detalles.

1.13.1 Sentencias PRINT y WRITE

La sentencia PRINT permite imprimir variables y constantes en la salida estándar, que de ordinario es la consola desde la que se está ejecutando el programa (por ejemplo, una ventana MS-DOS).

A continuación se muestra la forma general y algunos ejemplos de sentencia PRINT sin formatos:

```
PRINT *, lista_de_variables_y_constantes_separadas_por_comas
PRINT *, "Mi nombre es ", nombre, "."
PRINT *, "Velocidad = ", velocidad, "m/s."
PRINT *
```

Cada sentencia PRINT empieza en una nueva línea. Si se utiliza sin ninguna variable o constante a imprimir, simplemente produce una línea en blanco.

La sentencia WRITE es similar a PRINT, con la posibilidad adicional de elegir otras unidades de salida tales como impresoras y ficheros. También se puede utilizar WRITE sin variables, en cuyo caso imprime una línea en blanco.

```
WRITE (*, *) lista_de_variables_y_constantes_a_imprimir
WRITE (*, *)
```

La sentencia WRITE se verá con más detalle en el Tema 4.

1.13.2 Sentencia READ

La sentencia READ hace que se adquieran ciertos valores (normalmente desde teclado) desde una o más ***líneas de entrada*** y que se asignen a la lista de variables de la sentencia. Cuando el programa llega a una sentencia READ espera a que el usuario introduzca los valores de las variables que deben ser leídas. Por este motivo siempre es conveniente, antes de llamar a la sentencia READ, imprimir un mensaje por la consola solicitando los datos. A continuación se muestra una forma general y un ejemplo de sentencia READ:

```
READ *, lista_de_variables_separadas_por_comas
READ *, velocidad, tiempo
```

Una línea de entrada es un conjunto de valores o constantes de distinto tipo que terminan con un carácter ***Intro***. Una línea de entrada se puede introducir desde teclado, o se puede leer desde un fichero. Se pueden leer cadenas de caracteres poniéndolas entre comillas simples o dobles.

Cada vez que se ejecuta la sentencia READ se procesa una nueva línea de datos de entrada. Los distintos datos se separan mediante comas, o mediante uno o más espacios en blanco. Si en la línea de entrada (hasta que se pulsa ***Intro***) hay menos datos que en la lista, se continúan procesando nuevas líneas de entrada hasta obtener valores para todas las variables de la lista. Si hay más valores en la línea de entrada que variables a leer, se asignan valores a todas las variables y se ignoran los restantes valores de la línea.

Las constantes numéricas de la línea de entrada deben ser del mismo tipo que las variables de la lista, aunque a las variables reales es posible asignarles constantes enteras.

1.14 EJEMPLO COMPLETO DE PROGRAMA EN FORTRAN 90

A continuación se muestra un ejemplo sencillo pero completo de programa en Fortran 90. Este programa, que se guarda en un fichero llamado ***ecgrado2f.f90***, resuelve la ecuación de segundo grado.

Los coeficientes A, B y C se leen de un fichero de datos llamado ***datos.d***, que se supone está en el mismo directorio que el programa. A continuación se muestra el listado del programa numerando las sentencias para poderlas comentar más fácilmente:

```
1.    PROGRAM ecgrado2f
2.       IMPLICIT NONE
3.       INTEGER :: ierror=2
4.       REAL (KIND=8) :: A, B, C
5.       REAL (KIND=8) :: X, X1,X2,DISCR
6.       REAL (KIND=8), PARAMETER :: TOL=1e-10
7.       COMPLEX (KIND=8) :: XA,XB, PIM
8.
9.       ! Los datos se leen en el fichero datos.d
10.      OPEN(UNIT=10, FILE="datos.d", STATUS="OLD", ACTION="READ", &
              IOSTAT=ierror)
11.      IF (ierror /= 0) THEN
12.         PRINT *, "Error de lectura de fichero"
13.         STOP 123456
14.      END IF
15.      READ (10,*) A, B, C
16.      PRINT *, " ", "A=", A, " ", "B=", B, " ", "C=", C
17.      DISCR = (B**2-4.0*A*C)
18.
19.      IF (DISCR>TOL) THEN
20.         X1 = (-B+SQRT(DISCR))/(2.0*A)
21.         X2 = (-B-SQRT(DISCR))/(2.0*A)
22.         PRINT *, "Las soluciones son reales: X1=",X1," X2=",X2
23.      ELSE IF (DISCR<-TOL) THEN
24.         PIM = SQRT(CMPLX(DISCR))
25.         XA = (-B+PIM)/(2.0*A)
26.         XB = (-B-PIM)/(2.0*A)
27.         PRINT *, "Las soluciones son complejas: XA=",XA," XB=",XB
28.      ELSE
29.         X = -B/(2.0*A)
30.         PRINT *, "La raiz doble es: X=", X
31.      END IF
32.   END PROGRAM
```

A propósito de este programa se pueden hacer los siguientes comentarios:

1. Las sentencias 1 y 32 definen el comienzo y el final del programa, que se llama como se indica en la línea 1: ecgrado2f.

2. Las sentencias 3-7 definen las variables que se van a utilizar. Es imprescindible definirlas, porque en la sentencia 2 se ha establecido así con IMPLICIT NONE. El parámetro KIND=8 indica que las variables REAL y COMPLEX van a ser de 8 bytes (64 bits).

3. La palabra PARAMETER en la línea 6 indica que TOL es una constante simbólica, cuyo valor por tanto no puede cambiar a lo largo de la ejecución del programa.

4. La sentencia 10 abre para lectura (READ), un fichero ya existente (OLD), llamado **datos.d** (FILE) como unidad 10 (UNIT). Si la lectura es correcta, la variable INTEGER ierror tomará valor 0, pero si se produce un error tomará un valor positivo. En la sentencia 11 se comprueba que no ha habido error; si lo hay, la sentencia STOP detiene el programa imprimiendo el número que va a continuación.

5. En la sentencia 24 se calcula la raíz cuadrada de un número negativo. Para evitar un error es necesario convertir el argumento de REAL a COMPLEX con la función CMPLX().

2. CONSTANTES Y VARIABLES EN FORTRAN 90

2.1 TIPOS DE VARIABLES: REAL, INTEGER, CHARACTER Y LOGICAL

Como ya se ha explicado en la Introducción, Fortran permite trabajar con datos de distinto tipo. Los tipos básicos son REAL, INTEGER, CHARACTER y LOGICAL, cuyo significado se ha mencionado ya.

Dentro de un mismo tipo REAL o INTEGER se pueden definir variables de distinta clase o precisión.

2.2 CLASES DE VARIABLES DE UN TIPO DETERMINADO

Fortran permite trabajar con constantes y variables de tipo INTEGER o REAL con distinta precisión, es decir, con distinto número de cifras o rango. Por ejemplo, si se utilizan variables INTEGER con 16 bits se pueden representar números entre −32768 y 32767; con 32 bits se pueden representar enteros entre −2147483648 y 2147483647. También las variables REAL se pueden representar en PCs con 32 ó 64 bits; en el primer caso se tienen unas 7 cifras dcimales equivalentes y en el segundo caso 16.

La clase de variable INTEGER o REAL que se desea utilizar se puede especificar en el momento de declarar la variable, utilizando en parámetro KIND. Estas clases de variables dependen del ordenador y del compilador. A continuación se ofrecen algunos ejemplos de declaración de variables con KIND y posteriormente se dan los valores más habituales.

```
INTEGER (KIND=3) :: i, j, k
REAL (KIND=4), DIMENSION(100,100) :: A, B, C
```

2.2.1 Clases o rangos de variables INTEGER

La mayor parte de los compiladores Fortran 90 ofrecen las siguientes posibilidades para las variables INTEGER:

INTEGER	KIND=1	Enteros de 8 bits: entre −128 y 127
INTEGER	KIND=2	Enteros de 16 bits: entre −32768 y 32767
INTEGER	KIND=4	Enteros de 32 bits: entre −2147483648 y 2147483647

Con objeto de hacer que los programas sean portables entre computadores que utilizan sistemas diferentes, Fortran 90 dispone de la función intrínseca SELECTED_INT_KIND(r), en la que el argumento **r** es un entero. Esta función devuelve una clase −un valor de KIND− tal que todos los números enteros entre 10^{-r} y 10^{r} pueden ser representados. En el caso de que sea imposible representar ese intervalo completo se devuelve el valor (−1). Por ejemplo, una forma de garantizar suficiente rango para la constante simbólica **N** podría ser como sigue:

```
INTEGER, PARAMETER :: r12 = SELECTED_INT_KIND(12)
INTEGER (KIND = r12) :: N
```

También es possible definer el rango o la clase para las constantes. Esto se hace por medio de un carácter ***guión bajo*** (_) seguido de la clase de la constante, como por ejemplo:

```
123456789_3, 987654321012_r12
```

donde **r12** es la constante calculada previamente mediante la función SELECTED_INT_KIND(12).

2.2.2 Clases o rangos de variables REAL

Para las variables y constantes REAL se suelen ofrecer las siguientes posibilidades:

REAL KIND=4 Variables de ***simple precisión***, con 24 bits para la mantisa y 8 para el exponente. Se pueden representar números decimales con unas 7 cifras, entre 10^{-38} y 10^{38}.

REAL KIND=8 Variables de ***doble precisión***, con 53 bits para la mantisa y 11 para el exponente. Se pueden representar números decimales con unas 15~16 cifras, entre 10^{-308} y 10^{308}.

Fortran 90 dispone de la función SELECTED_REAL_KIND(p,r), donde **p** y **r** son enteros y **r** es opcional. Esta función devuelve la clase que garantiza **p** cifras decimales en el rango -10^r y 10^r. Esta función devuelve (−1) si no hay una clase con el rango solicitado, (−2) si no hay una clase con la precición solicitada, y (−3) si ni el rango ni la precición son alcanzables. Considérese el siguiente ejemplo de utililización de esta función:

```
REAL (KIND = SELECTED_REAL_KIND(12,30)
```

Las constants REAL también permiten especifical la clase mediante el **guión bajo** (_), como por ejemplo:

```
INTEGER, PARAMETER :: pr15 = SELECTED_REAL_KIND(15)
0.12345678901234_2, 0.12345678901234_pr15
```

2.3 VARIABLES COMPLEX

Fortran 90 permite trabajar con constantes y variables complejas. Las constantes complejas se definen mediante dos constantes reales entre paréntesis, separadas por una coma:

```
(1.23, -3.12), (1e-02, .324e-02)
```

Las variables complejas se declaran mediante la sentencia COMPLEX, que en muchos aspectos es similar a REAL:

```
COMPLEX :: x, y
COMPLEX (KIND=8), DIMENSION(100,100) :: A, B
```

Las variables complejas se pueden combinar con los ***operadores*** +, −, *, / y **. De todos los operadores relacionales, sólo los operadores == y /= se pueden utilizar con variables complejas.

También muchas funciones intrínsecas se pueden aplicar a variables complejas, como por ejemplo SQRT, ABS, SIN, COS, LOG y EXP. Estas funciones son generales, en el sentido que que su argumento puede ser una variable REAL o COMPLEX. Por lo general, para que se aplique la versión COMPLEX de la función, el argumento debe ser también COMPLEX.

Algunas funciones intrínsecas están especialmente dirigidas a variables complejas, como por ejemplo CONJG(c), que calcula el complejo conjugado; REAL(c) e AIMAG(c) que devuelven las partes real e imaginaria de un complejo **c** manteniendo la clase; CMPLX(x, KIND=k) y CMPLX(x, y, KIND=k), que convierten una o dos variables numéricas de cualquier tipo en un número complejo de la clase especificada, con parte imaginaria nula cuando sólo se da un argumento numérico.

Las variables complejas pueden ser leídas con READ. Cuando no se incluye formato, las partes real e imaginaria deben ir entre paréntesis, separadas por una coma. Si la lectura es con formato no se incluye el paréntesis. En la escritura sin formato con PRINT * o WRITE (*,*) los valores

complejos se imprimen asimismo entre paréntesis, separados por una coma. Los valores numéricos de la parte real e imaginaria se definen como los números reales.

2.4 ESTRUCTURAS

Al igual que en otros lenguajes de programación, las ***estructuras*** son agrupaciones de datos de naturaleza diferente. A cada uno de los datos que se agrupan bajo un mismo nombre se les llama ***componentes*** o ***campos***. Por ejemplo, la estructura **persona** puede tener dos componentes CHARACTER que sean el **nombre** y los **apellidos**; un componente INTEGER que sea el número de **DNI**, un componente REAL que sea la **estatura** en metros, etc.

Las estructuras se tratan como nuevos tipos de variables, con los que pueden formarse vectores y matrices, pasarse como argumentos a funciones y subrutinas, etc. Una estructura se define en la forma:

```
TYPE nombre_estructura
    declaracion_de_variables_1
    declaracion_de_variables_2
    declaracion_de_variables_3
    ...
END TYPE nombre_estructura
```

Esta definición se puede realizar después de IMPLICIT NONE y antes de la declaración de las variables. El nombre de la estructura debe de satisfacer las reglas de los identificadores de Fortran 90. Las estructuras pueden tener otras estructuras como componentes.

Una vez definida una estructura, pueden crearse constantes simbólicas o variables de esa estructura con algunas de las sentencias siguientes:

```
TYPE(nombre_estructura) :: lista_de_variables
TYPE(nombre_estructura), DIMENSION(10) :: A
TYPE(nombre_estructura) :: A(10)
```

Para inicializar una variable de este tipo se utiliza el nombre de la estructura seguido, entre paréntesis, por los valores adecuados para las distintas componentes, en el orden en que se definieron:

```
unaPersona = persona("Juan", "García García", 23458762, 1.82)
```

Por supuesto, el valor de una variable de una estructura puede ser asignado a otra variable del mismo tipo de estructura:

```
otraPersona = unaPersona
```

También se puede ***pasar como argumento*** a un subprograma una variable de tipo estructura, aunque es necesario en este caso que el argumento actual y el formal sean estructuras del mismo tipo.

Las estructuras, sin embargo, no se pueden ***leer o imprimir*** de un modo global, sino que hay que hacerlo componente a componente.

Se ***accede a una componente*** particular de una estructura por medio del carácter ***tanto por ciento*** (%), como en el siguiente ejemplo:

```
PRINT *, unaPersona%nombre, " ", unaPersona%apellidos
```

Esta forma de acceder a los componentes permite utilizarlos o modificarlos como se desee.

2.5 PUNTEROS

Fortran 90 ha introducido por primera vez punteros en el lenguaje, aunque los punteros de Fortran no tienen la versatilidad y generalidad de los punteros de otros lenguajes como C/C++. Por ejemplo, en Fortran ***no hay aritmética de punteros*** y tampoco existe la peculiar relación de los punteros con los nombres de vectores y matrices que existe en C/C++.

Los punteros son variables relacionadas con el contenido de una determinada dirección de memoria. El tipo de puntero está relacionado con el tipo de información almacenada en esa zona de la memoria.

2.5.1 Declaración y definición de punteros

Las variables puntero se declaran con el atributo POINTER, como por ejemplo:

```
TYPE fecha
    INTEGER :: day, month, year
END TYPE date
REAL, POINTER :: xptr, yptr, zptr
TYPE (fecha), POINTER :: fechaptr
```

Puede ser conveniente indicar de alguna manera en el nombre de una variable que se trata de un puntero (por ejemplo, terminando el nombre con las letras "ptr", como se ha hecho en los ejemplos anteriores).

Aunque se haya creado una variable puntero, dicha variable ***no está asociada*** todavía (no apunta) a una zona concreta de memoria. Se dice que está ***sin definir***. Para reservar una zona de memoria a la que apunte esa variable puntero se utiliza la sentencia ALLOCATE, en una de las dos formas siguientes:

```
ALLOCATE (lista_de_variables_puntero)
ALLOCATE (lista_de_variables_puntero, STAT = var)
```

En la segunda de estas formas la variable **var** es cero si la reserva de memoria re realiza sin dificultades y adquiere un valor positivo si se produce una ***situación de error***, como por ejemplo el que no haya memoria suficiente o que ya se haya reservado memoria previamente.

Además del estado ***inicial*** o ***sin definir***, los punteros pueden estar ***asociados*** o ***no-asociados*** (a este estado también se llama ***nulo***). Un puntero ***asociado*** es el que está relacionado con una zona de memoria concreta y ***no-asociado*** es el que ha perdido dicha asociación. La sociación se pierde por medio de la función NULLIFY:

```
NULLIFY(lista_de_variables_puntero)
```

Por otra parte, la función ASSOCIATED(varptr) devuelve .TRUE. o .FALSE. según el estado de **varptr** sea ***asociado*** o ***no-asociado***.

La función DEALLOCATE se utiliza para ***liberar la memoria*** reservada con ALLOCATE. Se puede utilizar en cualquiera de las formas siguientes:

```
DEALLOCATE (lista_de_variables_puntero)
DEALLOCATE (lista_de_variables_puntero, STAT = var)
```

2.5.2 Asignación de punteros

Hay que distinguir entre “asignación de punteros” y “asignación de las variables asociadas a los punteros”.

La ***asignación de punteros*** significa que un puntero pasa a estar asignado a la zona de memoria asociada a otro puntero del mismo tipo. Se realiza mediante los caracteres (=>) en la forma:

```
ptr2 => ptr1
```

Después de ejecutarse esta sentencia ambos punteros están ***asociados a la misma variable*** en la memoria. La asociación que tuviera **ptr2** con otra variable se pierde al ejecutar esta sentencia.

Fortran 90 permite crear variables con el atributo TARGET. Estas variables tienen como finalidad el asociar punteros a ellas, como por ejmplo:

```
REAL, TARGET :: xptr
REAL, POINTER :: yptr

ALLOCATE(xptr)
yptr => xptr
```

La function ASSOCIATED(xptr, yptr) devuelve .TRUE. si ambos punteros están asociados con la misma zona de memoria. Análogamente, ASSOCIATED(pointer, target) devuelve .TRUE. si **pointer** está aociado con **target**.

2.5.3 Utilización de punteros en expresiones y en sentecias de E/S

Cuando una variable puntero se utiliza en una expresión aritmética o lógica, o bien en una sentencia de entrada/salida, automáticamente se sustituye por el contenido de la memoria asociada con dicho puntero[1]. La sentencia,

```
xptr = xptr + w*m -3
```

utiliza el valor asociado con el puntero **xptr** para evaluar la expresión aritmética del miembro derecho y el resultado se almacena en la memoria asociada con **xptr**. En resumen, **xptr** se refiere a la misma zona de memoria antes y después de ejecutar esa sentencia, pero cambia el valor almacenado en esa zona de memoria.

En este sentido, ***los operadores (=) y (=>) se comportan de modo diferente con punteros***. Considérense dos punteros **xptr** e **yptr** del mismo tipo pero apuntando o asociados con zonas de memoria diferentes. La sentencia,

```
yptr => xptr
```

hace que **yptr** deje de apuntar a donde apuntaba antes y apunte a la misma zona de memoria que **xptr** (a partir de esta sentencia ambos nombres representan el mismo valor en cualquier expresión en que se utilicen, porque están asociados con la misma zona de memoria). Si se ejecuta la sentencia **xptr = xptr−1** también cambia el valor representado por **yptr** en cualquier otra expresión. Sin embargo, la sentencia,

```
yptr = xptr
```

copia el contenido de memoria apuntado por **xptr** a la zona de memoria apuntada por **yptr**. Si uno de ellos cambia su valor, el otro permanece con el valor anterior.

2.5.4 Paso de punteros como argumentos a subprogramas

Los punteros se pueden pasar como argumentos a funciones y subrutinas, pero con algunas condiciones: 1) Los argumentos actual y formal correspondientes deben ser punteros del mismo tipo. 2)

---

[1] Como Fortran 90, a diferencia de C/C++, no tiene aritmética de punteros, no tiene sentido utilizar un puntero como tal en una expresión y lo que se hace es utilizar el valor de la variable asociada con él.

Si un puntero se utiliza como argumento formal, no puede tener atributo INTENT. 3) Si un argumento formal es un puntero (o un target), el correspondiente subprograma debe tener definida una ***interface*** de modo explícito.

El ***valor de retorno*** de una función puede ser un puntero definido localmente, pero en este caso hay que utilizar un RETURN para devolver el valor asociado con dicho puntero.

3. SENTENCIAS DE CONTROL: BIFURCACIONES Y BUCLES

En ausencia de otros tipos de indicaciones, las sentencias de un programa Fortran se ejecutan de modo ***secuencial***, es decir cada una a continuación de la anterior. Las sentencias de control permiten modificar esta ejecución secuencial. Las principales sentencias de control corresponden a estas dos categorías:

1. ***Bifurcaciones***: Dependiendo de que se cumpla o no cierta condición, permiten ejecutar o no una parte del código, o bien permiten ejecutar alternativamente unas u otras sentencias.
2. ***Bucles***: Los bucles ejecutan repetidamente un conjunto de sentencias. Algunos bucles utilizan un contador para ejecutarlas un número determinado de veces y otros utilizan una condición de salida o de terminación.

En ambas sentencias de control las expresiones lógicas constituyen una parte importante, porque permiten controlar la ejecución de dichas sentencias.

3.1 EXPRESIONES LÓGICAS Y OPERADORES RELACIONALES

Las expresiones lógicas pueden ser ***simples*** o ***compuestas***. Se llaman ***expresiones lógicas simples*** las que constan de una constante lógica (.TRUE. y .FALSE.) o de una variable lógica, o bien de una expresión relacional en la forma:

```
expresión1 operador_relacional expresión2
```

donde tanto ***expresión1*** como ***expresión2*** son expresiones aritméticas, de caracteres o lógicas.

Los operadores relacionales de Fortran se muestran en la Tabla 3.

Símbolo	Significado
< o .LT.	Menor que
> o .GT.	Mayor que
== o .EQ.	Igual que
<= o .LE.	Menor o igual que
>= o .GE.	Mayor o igual que
/= o .NE.	Distinto de

Tabla 3. Operadores relacionales de Fortran 90.

Cada uno de los operadores relacionales tiene dos formas, que se pueden usar indistintamente. Para comparar cadenas de caracteres se utiliza el código ASCII de cada carácter; si es necesario, la cadena más corta se completa con blancos por la derecha.

Las ***expresiones lógicas compuestas*** constan una o más expresiones lógicas combinadas mediante los ***operadores lógicos***.

3.2 OPERADORES LÓGICOS

Los operadores lógicos de Fortran se muestran en la Tabla 4.

Operador	Significado
.NOT.	Negación
.AND.	Conjunción lógica
.OR.	Disyunción lógica
.EQV.	Equivalencia (.TRUE. si coinciden)
.NEQV.	Negación de equivalencia (.TRUE. si no coinciden)

Tabla 4. Operadores lógicos.

3.3 SENTENCIAS IF

Existen diversas formas de la sentencia IF para realizar bifurcaciones en la ejecución. Estas formas se presentan a continuación

3.3.1 Sentencia IF simple

La forma más sencilla de la bifurcación IF es la siguiente:

```
IF (expresión_lógica) THEN
    sentencias
END IF
```

Las ***sentencias*** que están dentro del bloque se ejecutan sólo si ***expresión_lógica*** es .TRUE.

Existe todavía ***un IF más sencillo*** para cuando se trata de ejecutar o no ***una sola sentencia***:

```
IF (expresión_lógica) sentencia
```

3.3.2 Sentencia IF compuesta

La forma general de la sentencia IF es la siguiente:

```
IF (expresión_lógica) THEN
    sentencias1
ELSE
    sentencias2
END IF
```

Si ***expresión_lógica*** es .TRUE. se ejecutan ***sentencias1*** y si no lo es se ejecutan ***sentencias2***.

3.3.3 Sentencia IF-ELSE IF

Esta es la forma más general de la sentencia IF, que permite encadenar varias condiciones lógicas.

```
IF (expresión_lógica1) THEN
    sentencias1
ELSE IF (expresión_lógica2) THEN
    sentencias2
ELSE IF (expresión_lógica3) THEN
    sentencias3
ELSE
    sentencias4
END IF
```

Pueden encadenarse tantas condiciones lógicas como se desee. Si ninguna de las expresiones lógicas es .TRUE. se ejecutan las sentencias que siguen a ELSE. Esta parte de la construcción es opcional; si no está presente y no se cumple ninguna de las condiciones, se ejecutan las sentencias que siguen al END IF.

3.3.4 Sentencias IF con nombre

Las sentencias IF pueden ***anidarse***, es decir, dentro de las sentencias de un IF puede haber otras sentencias IF. En estos casos puede ser útil poner un nombre a cada IF, nombre que aparece al prinncipio y final del bloque:

```
unNombre: IF (expresión_lógica) THEN
    sentencias1
ELSE
    sentencias2
END IF unNombre
```

En las bifurcaciones IF-ELSE IF el nombre se puede porner también después de cada THEN o detrás del ELSE:

```
otroNombre: IF (expresión_lógica1) THEN
    sentencias1
ELSE IF (expresión_lógica2) THEN otroNombre
    sentencias2
ELSE IF (expresión_lógica3) THEN otroNombre
    sentencias3
ELSE otroNombre
    sentencias4
END IF otroNombre
```

3.3.5 Sentencia CASE

La sentencia CASE equivale en cierta forma a una sentencia IF-ELSE IF compleja. Su forma general es la siguiente:

```
SELECT CASE (selector)
    CASE (lista_de_valores1)
        Sentencias1
    CASE (lista_de_valores2)
        Sentencias2
    CASE DEFAULT
        Sentencias3
END SELECT
```

En este caso ***selector*** es una expresión entera, de caracteres o lógica. Cada ***lista_de_valores*** es un conjunto de ***posibles valores para el selector***, entre paréntesis y separados por comas. Estos valores pueden ser simples (**valor**) o representar rangos (**valor1:valor2**, **:valor** ó **valor:**).

Cuando la ejecución llega a la sentencia SELECT CASE se evalúa la expresión ***selector***, y a continuación las sentencias que siguen a la ***lista_de_valores*** que contiene el resultado de dicha evaluación. Si ninguna ***lista_de_valores*** contiene dicho resultado se evalúan las sentencias que siguen a DEFAULT (si está presente).

3.3.6 Sentencia CASE con nombre

La sentencia CASE también puede tener nombres, que se utilizan en la forma:

```
unNombre: SELECT CASE (selector)
    CASE (lista_de_valores1)
        Sentencias1
    CASE (lista_de_valores2)
        Sentencias2
    CASE DEFAULT
        Sentencias3
END SELECT unNombre
```

3.3.7 Constantes y variables lógicas

En Fortran hay dos constantes lógicas: .TRUE. y .FALSE. (los puntos antes y después de la correspondiente palabra son obligatorios).

Las variables lógicas se declaran con la palabra LOGICAL y se les asigna valor por medio de expresiones lógicas.

Las variables lógicas pueden imprimirse con la sentencia PRINT sin formato, y en este caso se imprimen como una T ó una F, precedidas normalmente por un espacio en blanco.

También es posible leerlas con una sentencia READ sin formato. En este caso los puntos son opcionales y sólo se tiene en cuenta la primera letra de la palabra (es lo mismo ".F" que "F", que "FALSE" y que ".FALSO.").

3.4 BUCLES

Los bucles se utilizan para ***repetir la ejecución*** de un conjunto de sentencias un determinado número de veces o hasta que se cumpla una determinada condición. Es posible referirse a estos casos como ***bucles controlados por contador*** o ***bucles controlados por expresión lógica***, respectivamente.

3.4.1 Bucles DO controlados por contador

La primera forma de este bucle es la siguiente:

```
DO variable_de_control = valor_inicial, valor_final, incremento
    Sentencias
END DO
```

El modo de ejecutarse este bucle es el siguiente: la ***variable_de_control*** toma el ***valor_inicial*** y se compara con ***valor_final***. Las ***Sentencias*** se ejecutan si dicha ***variable_de_control*** es menor o igual que ***valor_final*** si ***incremento*** es positivo, o si ***variable_de_control*** es mayor o igual que ***valor_final*** si ***incremento*** es negativo.

Como el control se realiza ***al comienzo del bucle***, es posible que el bucle no se llegue a ejecutar ninguna vez. En DO, ***valor_inicial***, ***valor_final*** e ***incremento*** pueden ser expresiones.

Los valores inicial, final y el incremento de la variable de control ***se determinan antes*** de que el bucle comience a ejecutarse, y no deben ser modificados durante la ejecución del bucle. La ***variable_de_control*** puede ser utilizada en las ***Sentencias***, pero no debe ser modificada en ellas.

3.4.2 Bucles DO generales (controlados por expresión lógica)

La forma general del bucle DO controlado mediante una expresión lógica es la siguiente:

```
DO
    Sentencias1
    IF (expresión_lógica) EXIT
    Sentencias2
END DO
```

La ejecución del bucle se termina cuando ***expresión_lógica*** se hace .TRUE. Con esta construcción es posible que ***Sentencias1*** se ejecuten sólo una vez y que ***Sentencias2*** no se ejecuten ninguna. Según las sentencias que se pongan delante o detrás del IF se puede hacer que el control se relice al principio del bucle, al final o en un punto intermedio.

También es posible bloquear el ordenador si la condición de salida no se cumple nunca. Para evitar esto es conveniente introducir un ***contador de iteraciones*** e incluir en la condición de salida el que el contador haya alcanzado un valor máximo previamente fijado.

3.4.3 Sentencia CYCLE

La sentencia CYCLE introducida entre las demás sentencias del bucle hace que ***se termine la iteración actual*** (saltando las sentencias que están entre ella y el END DO) y se comience una nueva.

Esta sentencia se puede utilizar con cualquiera de los bucles DO.

3.4.4 Bucles DO con nombre

Los bucles con nombre no solamente son más fáciles de identificar, sino que también otras importantes ventajas prácticas en el caso de los ***bucles anidados***, pues el nombre puede también asociarse a las sentencias EXIT y CYCLE. Considérese el siguiente ejemplo:

```
bucle_1: DO
    Sentencias1
    bucle_2: DO
        Sentencias2
        IF (expresión_lógica1) THEN
            EXIT bucle_2
        ELSE IF (expresión_lógica2) THEN
            EXIT bucle_1
        END IF
    END DO bucle_2
END DO bucle_1
```

Con esta construcción, si ***expresión_lógica1*** es .TRUE. se abandona el ***bucle_2***, mientras que si es ***expresión_lógica2*** la que se hace .TRUE. se abandonaría el ***bucle_1***.

4. SENTENCIAS DE ENTRADA/SALIDA

Fortran tiene dos tipos de sentencias de E/S (Entrada/Salida): con ***formato fijo*** y con ***formato libre***, también llamadas ***controladas por formato***, o ***controladas por lista de constantes y variables***. En este Capítulo se verán principalmente las sentencias de E/S con formato fijo, incluyendo algunas alusiones a las sentencias con formato libre.

De ordinario existen unas unidades de entrada y de salida estándar, que suelen ser el teclado y la consola, respectivamente. Ya se verá cómo estas unidades pueden ser especificadas en el momento de leero y/o escribir.

4.1 SENTENCIA PRINT

La forma general de la sentencia PRINT es la siguiente:

```
PRINT formato, lista_de_constantes_y_variables
```

donde ***formato*** es una de las opciones siguientes:

a) Un asterisco (*): En este caso se trata de un formato libre, donde las variables de la lista se imprimen de un modo "razonable", dependiente del compilador utilizado.

b) Una constante o variable de tipo CHARACTER (o una expresión de este tipo, o bien un vector de caracteres), que especifica los formatos con los que se imprimirán las variables de la lista.

c) La etiqueta numérica (de 1 a 99999) de una sentencia FORMAT que especifica el formato.

Por lo general, cada sentencia PRINT da comienzo a una nueva línea, de tal forma que si se ejecuta una sentencia PRINT vacía, simplemente se deja una línea en blanco. A continuación se ofrecen las dos formas generales de sentencias PRINT con formato:

```
PRINT "descriptores_de_formato", lista_de_constantes_y_variables

PRINT label, lista_de_constantes_y_variables
label  FORMAT(descriptores_de_formato)
```

4.1.1 Espaciados y saltos de página

Tradicionalmente en Fortran el primer carácter de cada línea no se imprime y es considerado como un carácter de control que determina el espaciado vertical. Las posibles opciones son:

- ' ' (blanco): Espaciado normal; comenzar siempre una nueva línea.
- '0': Doble espacio; saltar una línea antes de empezar a imprimir.
- '1': Comenzar a imprimir en nueva página.
- '+': Comenzar a imprimir sin avanzar línea, en la última línea impresa.

Como la opción más fecuente es la primera, se suele introducir un carácter en blanco al comienzo de cada lista de variables:

```
PRINT "A1,F12.4,A4", " ", Velocidad, " m/s"
```

El significado de los formatos incluidos en el ejemplo anterior se entenderá en los apartados siguientes. Por ahora basta prestar atención al formato A1 y al carácter blanco " " al comienzo de la lsita de variables (ambos en negrita).

4.1.2 Descriptores de formato

Fortran dispone de numerosos descriptores de formato adaptados a los distintos tipos de variable y a las distintas formas de representarlos. Estos formatos se detallan en la **¡Error! No se encuentra el origen de la referencia.**.

Formatos		Significado
Iw	Iw.m	Constantes y variables INTEGER
Bw	Bw.m	Datos INTEGER en formato binario
Ow	Ow.m	Datos INTEGER en formato octal
Zw	Zw.m	Datos INTEGER en formato hexadecimal
Fw.d		Datos REAL con notación decimal
Ew.d	Ew.dEe	Datos REAL con notación exponencial
ESw.d	ESw.dEe	Datos REAL con notación científica
ENw.d	ENw.dEe	Datos REAL con notación ingenieril
Gw.d	Gw.dEe	Datos REAL con formato adaptable
A	Aw	Caracteres
"*…*"	'*…*'	Cadenas de caracteres
Lw		Datos LOGICAL
Tc	TLn TRn	Tabuladores
nX		Espacios en horizontal
/		Espaciado vertical
:		Control para lectura de caracteres

Tabla 5. Distintos especificadores de formato de la sentencia PRINT.

El significado de los "campos" que aparecen en la Tabla 5 es el siguiente:

w nº entero positivo que indica la anchura del campo (por ejemplo, 6 espacios)
m nº entero igual o mayor que cero que indica un nº mínimo de dígitos a leer o imprimir
d nº entero positivo que indica el númedo de cifras a la derecha del punto decimal
e nº entero igual o mayor que cero que indica el nº de cifras en un exponente
* un carácter
c nº entero positivo que representa la posición de un carácter
n nº entero positivo que indica un nº de caracteres

4.1.3 Formato para números enteros

Por ejemplo, el formato I5 indica que se dispone de cinco espacios para imprimir un número entero; si el nº es positivo y tiene menos de cinco cifras se ajusta por la derecha y se completa con blancos por la izquierda. Si es lo primero que se imprime en una línea el primer carácter por la izda se toma como carácter de control y no se imprime. En principio el signo se imprime sólo si es negativo. Si el número a imprimir precisa más de cinco espacios se imprimen cinco asteriscos (*****), como forma de advertir que se ha producido un error.

4.1.4 Formatos para números reales

Los formatos reales −descriptores F, E, ES y EN− también ajustan el contenido por la derecha. Por ejemplo, el formato F10.4 indica que se reserva un total de 10 espacios para imprimir el número y que aparecerán 4 cifras a la derecha del punto decimal. Si tiene menos de cuatro cifras decimales se

completa con ceros por la derecha. Si necesita más de 10 espacios se imprimen asteriscos (*) en toda la anchura del campo.

El formato E12.6 imprime el número en forma exponencial normalizada: un signo − opcional, un cero, un punto decimal seguido de 6 cifras, la letra E y el exponente de 10 con 4 cifras (con 2 cifras si se hubiera definido como E12.6e2). También se completa con blancos por la izquierda y con asteriscos si la anchura del campo es insuficiente para representar el número a imprimir.

La notación científica ES es similar a la E, pero la mantisa se normaliza de otra forma, de modo que mayor o igual que 1 y menor que 10 (a no ser que sea cero). La notación ingenieril EN es también similar, pero en este caso la mantisa es mayor o gual que 1 y menor que 1000 (salvo que sea cero) y el exponente siempre ha de ser múltiplo de tres.

4.1.5 Formatos para cadenas de caracteres

Las cadenas de caracteres se imprimen mediante el descriptor A, que tiene dos formas: Si se especifica A se imprime la cadena ocupando los espacios que necesita; si se especifica A20 la salida impresa tendra 20 caracteres. Si la cadena tiene menos caracteres se ajusta por la derecha y se rellena con blancos por la izquierda. Si la cadena tiene más de 20 caracteres, se imprimen los 20 caracteres que están más a la izquierda.

4.1.6 Formatos de espaciado horizontal (X y T)

El formato 12X deja 12 espacios en blanco en la línea de salida.

Por otra parte, el formato T32 indica que el siguiente dato se imprimirá en esa línea a partir de la posición 32.

4.1.7 Factor de repetición de formatos

El factor de repetición se antepone a uno de los formatos de enteros o reales vistos previamente. Por ejemplo, los formatos 3I6 y 5F10.3 permiten respectivamente imprimir tres enteros con seis espacios para cada uno, y cinco números reales con una anchura de 10 posiciones y 3 cifras decimales.

Es posible también repetir un conjunto de descriptores encerrándolos entre paréntesis y poniendo el factor de repetición antes del paréntesis, como por ejemplo: 3(I5,2X,F12.4).

4.1.8 Descriptor de cambio de línea /

El descriptor barra / hace que se produzca un salto a la línea siguiente. Se puede poner varias veces seguidas (///), o se puede utilizar un factor de repetición (3/).

4.1.9 Correspondencia entre la lista de variables y los descriptores de formato

Fortran establece una correspondencia entre la lista de variables a imprimir y la lista de descriptores de formato. Esta correspondencia sigue las siguientes reglas:

1. Si la lista de variables se agota antes que la lista de formatos, la lista de descriptores se continúa ejecutando, imprimiéndose las constantes, los espaciados y los tabuladores, hasta que se produce una de las condiciones siguientes:
    a. Se llega al cierre de paréntesis que determina el fin de la lista de descriptores
    b. Se encuentra un descriptor de uno de los tipos I, F, E, ES, EN, A, L, G, B, O ó Z.

c. Se encuentra una coma (,)

2. Si la lista de descriptores de formato se agota antes que la lista de variables, se comienza una nueva línea de salida y se vuelve a empezar con la lista de descriptores desde el principio si ésta no tiene paréntesis. Si hay paréntesis internos se repiten los descriptores contenidos en el paréntesis interno más a la derecha, incluyendo el factor de repetición si lo tuviera. Considérese el siguiente ejemplo:

```
PRINT 100, I, A, I+1,A+1.0, I+2,A+2.0, I+3,A+3.0
100 FORMAT(1X, I10,F14.4, / (1X, I6, F10.2))
```

Si I=−100 y A=3.141592654, la salida que se obtiene es la siguiente:

```
      -100         3.142
-99      4.14
-98      5.14
-97      6.14
```

4.2 ENTRADA DE DATOS CON FORMATO. SENTENCIA READ

La lectura de datos se realiza con la sentencia READ. Su forma general, con formato, es:

```
READ especificadores_de_formatos, lista_de_variables
```

En este caso los especificadotes o descriptores de formatos son muy similares a los vistos anteriormrnte. La principal diferencia es que en los formatos de entrada no pueden aparecer constantes con cadenas de caracteres, y que la presencia de comas (,) como separador es ignorada.

A continuación se presentan algunos ejemplos de distintos tipos de variables.

4.2.1 Lectura de variables INTEGER

La lectura de variables INTEGER se hace mediante campos del tipo **rIw**, como por ejemplo:

```
READ '(I5,I10,I5)', I, J, K
```

que es equivalente a:

```
READ 100, I, J, K
100 FORMAT(I5,I10,I5)
```

Los datos pueden disponerse separados por más o menos blancos, pues los blancos por defect se ignoran en las líneas de entrada (salvo que se utilice el descriptor BZ, que hace que se interpreten como ceros). Los datos purden prepararse también sin ningún tipo de separadores (ni blancos, ni comas), pero en este caso deben ajustarse exactamente a la anchura de campor prevista.

4.2.2 Lectura de variables REAL

Los valores correspondientes a variables REAL pueden prepararse con o sin puntos decimales. Si no se introducen puntos decimales, los puntos se suponen exactamente donde lo indican los formatos F, E, ES ó EN. Si se definen los puntos en los valores de entrada, esta definición tiene prioridad sobre la indicada en los formatos F, E, ES ó EN. Si no se utilizan separadores (comas y/o blancos) la anchura de los distintos campos debe coincidir exactamente con la indicada en los descriptores de formato. La anchura indicada en el descriptor de entrada debe ser suficiente para el número que se desae leer, incluyendo el punto, el signo, etc.

Los datos preparados con formato exponencial pueden también ser leídos con descriptores F.

4.2.3 Lectura de variables CHARACTER

Cuando la lista de variables de una sentencia READ contiene variables CHARACTER, se leen tantos caracteres de la línea de entrada como longitud o nº de caracteres tienen las variables correspondientes.

Por ejemplo, considérense las variables **frase1** y **frase2** y la siguiente sentencia READ:

```
CHARACTER (8) :: frase1, frase2
READ 100, frase1, frase2
100 FORMAT(2A)
```

Si en la línea de entrada se tiene la frase "La liga de las estrellas", el contenido de **frase1** será "La liga " y el de **frase2** "de las e". Si los formatos fueran (A8,A8) el resultado sería el mismo, pero si fueran (A6,A12) los contenidos de **frase1** y **frase2** serían respectivamente "La ligbb" y " las est", donde la 'b' representa un carácter en blanco. Obsérvese que **frase1**, asociada con un formato A6, recoge los seis primeros caracteres y, como tiene 8 espacios, completa con dos blancos por la derecha; la variable **frase2** está asociada con un formato A12, por lo que toma los 12 siguientes caracteres (del 7 al 18) y almacena sólo los 8 últimos.

4.2.4 Salto de caracteres

Los descriptores de posición horizontal X y T pueden utilizarse para ignorar caracteres en la línea de entrada. Por ejemplo, supóngase que se tiene una línea con valores para las variables I, J, K que incluye los nombres de dichas variables:

```
I=10 J=-20  K=314
```

Las siguientes sentencias READ asignan los valores correctos a las variables I, J, K:

```
READ '(2X, I2, 3X, I3, 4X, I3)', I, J, K
READ '(T2, I2, T8, I3, T14, I3)', I, J, K
```

Es obvio que la utilización de estas sentencias implica un conocimiento muy preciso de cómo se hallan preparados los datos.

4.2.5 Líneas de entrada múltiples

Cada vez que se ejecuta una sentencia READ se comienza a leer una nueva línea. También se comienza a leer una nueva lñinea si se encuentra el descriptor salto de línea (/), que puede ser utilizado para saltar líneas con comentarios y sin datos de interés.

4.3 SENTENCIA WRITE

La sentencia WRITE es más complicada y tiene más posibilidades que la sentencia PRINT. Su forma general es la siguiente:

```
WRITE (lista_de_control) lista_de_constantes_y_variables
```

La **lista_de_control** puede contener los elementos siguientes:

1. Una ***unidad de escritura***, que es una expresión entera que que indica un dispositivo de salida. Si es un ***asterisco*** (*) se utiliza la unidad de salida estándar (la consola o una impresora). Esta unidad se puede poner también en la forma UNIT=expresion_entera, lo que hace que el contenido quede más claro.

2. Un conjunto de especificadotes de formatos, tales como los presentados para PRINT. Pueden escribirse sin más, como en PRINT, pero en ese caso deben ser el 2º item de la **lista_de_control**; también pueden incluirse en la forma FMT=especificadotes_formato.
3. Una especificación ADVANCE en la forma ADVANCE=conjunto_de_caracteres, donde **conjunto_de_caracteres** es "NO" o "YES" después de eliminar blancos y pasar a mayúsculas. Esta especificación se utiliza para indicar si al final de la actual operación de lectura hay que avanzar o no a la siguiente línea. La opción por defecto es 'YES'.
4. Otros elementos de utilidad para operaciones de lectura/escritura de ficheros.

A continuación se muestran algunos ejemplos típicos de uso de la sentencia WRITE:

```
WRITE (*, *) A, B, C   ! equivalente a PRINT *; A, B, C
WRITE (6, FMT='3F10.4') A, B, C
WRITE (UNIT=unidadSalida, *) A, B, C
```

4.4 SENTENCIA READ GENERAL

La forma general de la sentencia READ es similar a la de WRITE, y puede utilizarse por ejemplo para leer de ficheros, en lugar de la unidad de entrada estándar. Dicha forma general es la siguiente:

```
READ (lista_de_control) lista_de_constantes_y_variables
```

La **lista_de_control** es similar a la de la sentencia WRITE, añadiendo una cláusula IOSTAT= que permite detectar y gestionar condiciones de error, tales como una marca de fin de fichero encontrada antes de lo previsto.

4.5 SENTENCIAS PARA LECTURA/ESCRITURA DE FICHEROS

Uno de los principales usos de las formas generales de READ y WRITE es para operaciones de lectura y/o escritura de ficheros.

La primera operación que se debe realizar es la de abrir el fichero, asociándolo con un dterminado número de unidad. Los ficheros se abren por medio de la sentencia OPEN, que tiene la forma:

```
OPEN (lista_de_apertura)
```

donde **lista_de_apertura** incluye los siguientes especificadotes:

1. Un número especificador de unidad, que será luego la unidad a la que se hará referencia en las sentencias READ y WRITE.
2. Una cláusula FILE = nombre_de_fichero.
3. Una cláusula STATUS = cadena_de_caracteres, donde **cadena_de_caracteres** es (tras eliminar blancos y convertir a mayúsculas) una de estas posibilidades: 'OLD', 'NEW', 'REPLACE' ó 'UNKNOWN'. La primera se utiliza si el fichero ya existe; la segunda cuando el fichero debe ser creado por el programa, en cuyo caso OPEN crea un fichero vacío y cambia el estado a 'OLD'; con 'REPLACE' se crea un nuavo fichero, borrando uno anterior con el mismo nombre si existe, y cambiando el estado a 'OLD'.
4. Una cláusula ACTION = operacion, donde **operacion** es una cadena de caracteres cuyos valores pueden ser 'READ', 'WRITE' ó 'READWRITE' (o equivalentes), lo que determina

si el fichero se abre sólo para lectura, sólo para escritura, o para ambas operaciones. Si se omite esta cláusula, lo normal es que el fichero se abra para lectura y escritura.

5. Una cláusula POSITION = cadena_de_caracteres, donde **cadena_de_caracteres** tiene como posibles los valores siguientes: 'REWIND', 'APPEND' ó 'ASIS'. En el primer caso el cursor se situaría al comienzo del fichero, en el segundo al final y en el tercero se quedaría donde estaba. Si se omite esta cláusula el cursor sigue donde está si el fichero estaba ya abierto, o se crea el fichero y el cursor se coloca al principio si el fichero no existía.
6. Una cláusula IOSTAT = variable_de_estado, donde **variable_de_estado** es una variable entera a la que se asigna un valor según haya transcurrido la operación. De ordinario se asigna un cero si el fichero se abre sin problemas, un nº positivo si se produce alguna condición de error, y un nº negativo si se ha llegado al fin del fichero sin que se produzca un error.

Por ejemplo, a continuación se indica cómo se abre un fichero llamado **datos.d** para lectura:

```
OPEN(UNIT=10, FILE="datos.d", STATUS="OLD", ACTION="READ", &
     POSITION="REWIND", IOSTAT=ierror)
```

Otro ejemplo de apertura de ficheros es el siguiente:

```
OPEN(UNIT=10, FILE=miFichero, STATUS="NEW", ACTION="WRITE", &
     POSITION="REWIND", IOSTAT=ierror)
```

Después de abrir el fichero es conveniente comprobar que la variable **ierror** es cero, es decir, que el fichero ha sido abierto sin problemas. Existen algunás otras cláusulas tales como BLANK (que indica cómo interpretar los blancos en los campos numéricos; DELIM, que indica si las cadenas de caracteres deben ser incluidas entre delimitadores; ERR, que indica una sentencia que se debe ejecutar en caso de error; FORM, que indica si el fichero está o no formateado; PAD, que indica si las cadenas de caracteres leídas deben ser completadas con blancos; y RECL, que indica la longitud de los registros para operaciones de acceso directo (no secuencial, como e slo más normal).

Una vez terminadas las operaciones de lectura y/o escritura, se debe proceder a cerrar el fichero, lo cual se realiza con la sentencia CLOSE, cuya forma general es la siguiente:

```
CLOSE (lista_de_clausura)
```

donde **lista_de_clausura** incluye necesariament el número de unidad, y opcionalmente puede incluir las cláusulas IOSTAT, ERR y STATUS. Los ficheros no cerrados explícitamente se cierran de modo automático al llegar a una sentencia STOP ó END.

Existen otras sentencias auxiliares, como REWIND y BACKSPACE. La sentencia **REWIND unidad** permite volver a poner el cursor al comienzo de un fichero abierto. La sentencia **BACKSPACE unidad** pone el cursor al comienzo de la línea precedente.

5. FUNCIONES Y SUBROUTINAS

A diferencia de otros lenguajes de programación como C/C++ y Java, Fortran dispone de dos tipos de subprogramas: las ***funciones*** y las ***subrutinas***. La principal diferencia es que las funciones tienen un ***valor de retorno*** y pueden por tanto ser utilizadas en expresiones aritméticas o de otro tipo. Por el contrario, las subrutinas no devuelven ningún valor y todos sus resultados los transmiten a través de ***modificaciones de sus argumentos***.

Los subprogramas responden a los requerimientos de ***encapsulación*** y ***reutilización*** de código. El código de un subprograma es independiente del código del programa principal y del de cualquier otro subprograma. La comunicación con un subprograma se realiza exclusivamente a través de los argumentos y −en el caso de las funciones− también a través de su resultado o valor de retorno. Esto quiere decir que las variables del subprograma son independientes de las variables de cualquier otro programa, también aunque tengan el mismo nombre.

Una función se puede utilizar desde muchos programas diferentes y en muchas condiciones diferentes. Una vez que su código se ha depurado y se tiene una razonable seguridad de que está libre de errores, puede utilizarse con gran confianza en cualquier otro programa.

5.1 FUNCIONES

El propio lenguaje Fortran proporciona un gran número de ***funciones de librería***, algunas de las cuales como el ***seno*** y el ***coseno*** ya se han visto previamente. El usuario puede programar sus propias funciones que se comportan igual que éstas.

5.1.1 Partes o componentes de una función

Las funciones de Fortran constan de las partes siguientes:

1. Encabezamiento o sentencia FUNCTION.
2. Sentencias de especificación
3. Sentencias ejecutables
4. Sentencia END FUNCTION

La sentencia FUNCTION es la primera sentencia de cualquier función y tiene la forma siguiente:

```
[tipo_retorno] FUNCTION nombre_funcion(lista_de_argumentos)
```

donde **tipo_retorno** es un identificador de tipo opcional (REAL, INTEGER, …), **nombre_funcion** es un identificador válido de Fortran, y **lista_de_argumentos** es una lista de variables que son utilizadas para pasar información a la función. A estos argumentos que aparecen en el encabezamiento de la función se les llama ***argumentos formales***, para distinguirlos de los ***argumentos actuales***, que son los que aparecen en la llamada a la función desde otro programa.

En la parte de las sentencias de especificación deben determinarse:

1. El tipo del valor de retorno, si no se ha especificado en la sentencia FUNCTION.
2. Los tipos de todos y cada uno de los argumentos formales. En esta especificación puede aparecer el cualificador INTENT, que sirve para indicar cómo se usa ese argumento. Por ejemplo, INTENT(IN) indica que esa variable es sólo ***de entrada*** y que no se deberá modificar en la función. Por ejemplo:

```
REAL FUNCTION EspacioRecorrido(velocidad, tiempo)
REAL, INTENT(IN) :: velocidad, tiempo
```

Las sentencias ejecutables son distintas sentencias de Fortran que realizarán diversas operaciones, pero que siempre deberán calcular el valor de retorno antes de devolver el control. Esto se hace ***asignando valor*** a una variable cuyo nombre y tipo es el de la función.

La última sentencia de la función tiene la forma:

```
END FUNCTION nombre_funcion
```

o bien, para las ***funciones externas*** (las definidas después de la sentencia END PROGRAM):

```
END
```

Téngase en cuenta que ***la forma de llamar a la función*** en el programa principal es por medio del ***nombre sequido de los argumentos actuales*** entre paréntesis, en una expresión que sea compatible con el tipo del valor de retorno de la función.

5.1.2 Sentencia RETURN

La ejecución de la función termina cuando se llega a la sentencia END FUNCTION o cuando se encuentra una sentencia RETURN en la parte correspondiente a las sentencias ejecutables. Se puede decir que la sentencia RETURN es una forma de adelantar la devolución del control al programa que ha llamado a la función, sin esperar a llegar a la sentencia END FUNCTION.

5.1.3 Dónde se definen las funciones

Las funciones deben definirse de modo que el programa que las llama o ***programa principal*** tenga acceso a ellas. Esto se puede hacer de una de las formas siguientes:

1. Colocando el código de la función en el mismo fichero que el programa principal, justo ***antes*** de la sentencia END PROGRAM. En este caso se dice que la función es ***interna***.
2. El código de la función se introduce en un ***módulo***, al que el programa principal deberá tener acceso para poder importarlo. En este caso se dice que la función es un **módulo**.
3. El código de la función puede ser también situado en el mismo fichero que el programa principal, pero ***después*** de la sentencia END PROGRAM. En este caso se dice que la función es ***externa***.

Las ***funciones internas*** se definen al final de un programa (o de otra función externa). La definición de las funciones −puede haber más de una− se realiza después de la palabra clave CONTAINS. No se puede definir una función dentro de otra función interna.

5.1.4 Argumentos actuales y formales

Una función se llama utilizando su ***nombre***, seguido de una lista de ***argumentos actuales*** entre paréntesis, en una expresión que normalmente será del tipo del valor de retorno de la función.

Los ***argumentos actuales*** pueden ser variables o expresiones. Si son expresiones se evalúan antes de llamar a la función. Después se realiza una ***asociación*** entre la lista de ***argumentos actuales*** y la lista de ***argumentos formales***, que contiene las variables en las cuales la función recibe los valores de los argumentos actuales. Mientras los argumentos actuales pueden ser expresiones, los argumentos actuales siempre son variables. En cualquier caso, el número y tipo de los argumentos actuales debe coincidir con el número y tipo de los argumentos formales.

Además del valor de retorno (que es siempre un único resultado), las funciones tienen otra forma de transmitir resultados al programa que la ha llamado. Si el argumento actual es una variable y el correspondiente argumento formal ***se modifica dentro de la función***, esa modificación se transmite a la variable que constituía el argumento actual.

Esta modificacion de los argumentos actuales se impide si el argumento formal se declara dentro de la función con el cualificador INTENT(IN).

5.1.5 Reglas de visibilidad y permanencia de las variables

Es importante considerar qué programas tienen acceso a una variable o a una función que se ha declarado en unas determinadas circunstancias. La ***regla general*** es que *una variable o una función sólo son visibles (sólo se pueden utilizar) en el programa o subprograma en el que se han declarado*.

Como consecuencia de la regla anterior, una variable declarada en una función (en un subprograma) sólo es accesible dentro de esa función. Por eso estas variables se llaman ***locales***.

Asimismo, *una variable o subprograma declarado dentro de un programa principal (entre las sentencias PROGRAM y END PROGRAM) son visibles dentro de dicho programa principal y de sus subprogramas internos, excepto si en dichos subprogramas hay una variable local con el mismo nombre que la oculta*. A estas variables o subprogramas se les llama ***globales***.

En el caso de las funciones que se llaman a sí mismas (***funciones recursivas***) esta llamada oculta también otras variables o funciones que pudieran tener ese mismo nombre.

La sentencia IMPLICIT al comienzo de un programa principal tiene también carácter ***global***.

En la práctica, aunque las variables globales constituyen otro mecanismo de comunicación entre el programa principal y los subprogramas, ***se desaconseja vivamente su uso***, pues es considerado como una técnica de programación peligrosa.

Con respecto a la persistencia de variables hay que señalar que por defecto las ***variables locales*** de un subprograma ***se crean y se inicializan*** cada vez que el subprograma se ejecuta de nuevo. Si se desea que algunas variables locales ***conserven su valor*** entre las sucesivas ejecuciones sel subprograma hay dos posibilidades:

1. Declararlas con el atributo SAVE como por ejemplo:

```
INTEGER, SAVE :: numeroLlamadas
```

2. Utilizar la sentencia SAVE en la forma (si se omite **lista_de_variables** se guardan todas las variables locales del subprograma):

```
SAVE lista_de_variables
```

5.1.6 Módulos

Los módulos son ***ficheros de código Fortran 90*** que contienen ***declaraciones de variables con sus tipos***, ***subprogramas*** y ***definiciones de nuevos tipos de variables*** (se verán posteriormente). Los módulos definen ***librerías*** de utilidad, que pueden ser incorporadas a muchos programas diferentes.

Los módulos se definen en la forma siguiente:

```
MODULE nombre_modulo
CONTAINS
    Subprograma_1
    Subprograma_2
END MODULE nombre_modulo
```

Los módulos se almacenan en ***ficheros fuente especiales***, que deben ser compilados de forma independiente que el programa principal. En el momento de crear el ejecutable hay que proporcionar también los ficheros objeto resultantes de la compilación de los módulos. La forma concreta de hacerlo depende del compilador que se utilice.

Para que un programa principal pueda hacer uso de las funciones de un módulo hay que incluir en él (al comienzo de la parte de especificación) la sentencia USE, en una de las formas siguientes:

```
USE nombre_modulo
```

o bien:

```
USE nombre_modulo ONLY: lista_de_identificadores
```

En este segundo caso sólo se podrán utilizar las variables y subprogramas que hayan sido especificados. Existe la posibilidad de ***cambiar el nombre*** (para usar un nombre distinto en el programa principal) de algunos identificadores. Esto se hace construyendo la lista en la forma:

```
USE nombre_modulo ONLY: nuevoNombre1 => nombre1, nuevoNombre2 => nombre2, nombre3
```

5.1.7 Funciones externas e interfaces

Las ***funciones externas*** −o en general, los ***subprogramas externos***− son accesibles desde un programa principal porque se incluyen en el mismo fichero, después de la sentencia END PROGRAM.

En este caso ni el programa principal accede a variables de la función externa, ni ésta a las variables o funciones declaradas en el programa principal. El único "canal" de comunicación abierto entre ambos son los ***argumentos*** con los que el programa principal llama a la función externa.

Al compilar un programa principal, gracias a que las ***funciones internas*** se definen entre las sentencias PROGRAM y END PROGRAM, y a que las ***funciones de los módulos*** se especifican mediante la sentencia USE, el compilador puede chequear fácilmente que los argumentos actuales y formales de las funciones internas y de los módulos coinciden en número y tipo, así como que el valor de retorno de la función se usa de modo correcto. En estos casos se dice que la ***interface*** entre el programa que llama y las funciones llamadas está ***definida explícitamente***.

Con las funciones externas la situación es diferente, ya que el compilador no tiene noticia de ellas mientras compila las sentencias contenidas entre PROGRAM y END PROGRAM. En este caso se dice que la interface está definida ***implícitamente***. Para asegurar que el compilador tenga información información completa sobre las funciones externas conviene definir también en este caso la interface explícitamente. Fortran 90 introduce para ello los bloques INTERFACE (opcionales en algunos compiladores, obligatorios en otros). Estos bloques se introducen en la parte de especicficación del programa principal y contienen información sobre cada función externa que va a ser llamada: nombre, argumentos, tipo de los argumentos. La forma concreta de dar esta información para el caso de una sola función la muestra el siguiente ejemplo:

```
INTERFACE
    FUNCTION espacio(vel, tiempo)
        REAL, INTENT(IN) :: vel
        REAL :: tiempo
    END FUNCTION espacio
END INTERFACE
```

5.1.8 Funciones recursivas

Las funciones recursivas son ***funciones que se llaman a sí mismas*** (obviamente con distintos datos, porque en otro caso el ordenador entraría en un bucle infinito). Las funciones recursivas simplifican en gran medida la programación de ciertos algoritmos, por lo que son muy utilizadas en esos casos.

El ejemplo más típico es el cálculo del ***factorial de un número natural***, que se define de la forma siguiente:

$$n! = n(n-1)(n-2)...3 \cdot 2 \cdot 1, \qquad (n \geq 0,\ 0! = 1)$$

Utilizando recursividad, el factorial de *n* se puede calcular en la forma:

$$\begin{aligned} n! &= n \cdot (n-1)!, \quad \text{si } n > 0 \\ &= 1, \qquad\qquad \text{si } n = 0 \end{aligned}$$

En la fórmula anterior la primera línea representa el ***caso general***, mientras que la segunda representa el ***caso básico***. La presencia y la detección del caso básico son fundamentales, porque si no el proceso no terminaría nunca.

Fortran 90 se diferencia de otros lenguajes como C y Java en que *el valor de retorno se devuelve a través de una variable con el nombre y tipo de la función* (en C y Java el valor de retorno se devuelve por medio de la sentencia ***return***, pudiendo estar guardado en cualquier variable o ser el resultado de una expresión). Esto crea algunas dificultades para las funciones recursivas, que deben utilizar su nombre para almacenar el resultado y también para llamarse a sí mismas. Por ejemplo, el cálculo recursivo del factorial de *n* se podría hacer en una función de la forma:

```
FUNCTION factorial(n)  !ESTE PROGRAMA NO ES CORRRECTO
    INTEGER factorial
    INTEGER, INTENT(IN) :: n

    IF (n==0) THEN
        factorial = 1
    ELSE
        factorial = factorial * factorial(n-1)
    END IF

END FUNCTION factorial
```

Este programa ***no es correcto*** porque el mismo identificador **factorial** se está utilizando de dos formas distintas: como simple variable y como nombre de función.

Para resolver esta dificultad Fortran 90 introduce el cualificador RECURSIVE, que se introduce al comienzo del encabezamiento de la función. El compilador ya sabe que tiene que trata a esta función de una forma especial, porque a las funciones recursivas se les permite dar el valor de retorno a través de una ***variable con un nombre distinto*** del nombre de la función. Esta variable se especifica al final del encabezamiento con la palabra RESULT seguida del nombre de dicha variable entre paréntesis.

En este caso, el programa Fortran 90 podría tener las sentencias siguientes:

```
RECURSIVE FUNCTION factorial(n) RESULT (fac)
    INTEGER :: fac.
    INTEGER, INTENT(IN) n
```

```
        IF (n==0) THEN
            fac = 1
        ELSE
            fac = n*factorial(n-1)
        END IF

    END FUNCTION factorial
```

5.2 SUBRUTINAS

5.2.1 Analogías y diferencias entre funciones y subrutinas

Las ***subrutinas*** son subprogramas con muchas cosas en común y algunas diferencias con las ***funciones***. De modo análogo que las funciones, las subrutinas:

- Son llamadas desde un ***programa principal*** o desde otra subrutina para realizar una tarea determinada.
- Constan de un ***encabezamiento*** con la palabra SUBROUTINE y una lista de argumentos entre paréntesis, unas declaraciones, un cuerpo de sentencias ejecutables y la sentencia END.
- Pueden ser ***internas***, formar parte de un ***módulo*** o ser ***externas***.
- Las reglas de ***visibilidad*** y ***permanencia*** de variables son las mismas que para las funciones.
- Su nombre puede ser pasado como argumento a otro subprograma y pueden llamarse a sí mismas, es decir pueden ser ***recursivas***.

Sin embargo, las subrutinas se diferencias de las funciones en que:

- Las funciones tienen siempre un valor de retorno, que toma valor a través de una variable con el mismo nombre que la función. Las subrutinas ***no tienen valor de retorno*** propiamente dicho y siempre devuelven los resultados a través de ***modificaciones de los argumentos formales*** que se transmiten a los argumentos actuales. En ocasiones las subrutinas no devuelven nada y se limitan por ejemplo a imprimir ciertos valores.
- Las funciones se llaman incluyendo su nombre seguido de los argumentos en una expresión. Las subrutinas se llaman siempre a través de una sentencia CALL.

5.2.2 Encabezamiento y llamadas a una subrutina

El encabezamiento de una subrutina tiene la siguiente forma general:

```
SUBROUTINE nombre_de_subrutina(lista_de_argumentos_formales)
```

Para el caso de una ***subrutina recursiva*** (no tiene el problema de confusión de nombres):

```
RECURSIVE SUBROUTINE nombre_de_subrutina(lista_de_argumentos_formales)
```

La llamada a una subrutina se hace a través de una sentencia CALL, que no consta de nada más que de la llamada a la subrutina:

```
CALL nombre_de_subrutina(lista_de_argumentos_actuales)
```

5.2.3 Relación entre argumentos formales y actuales

Al igual que en las funciones, al llamar a una subrutina ***se evalúan los argumentos actuales*** que aparecen en la sentencia CALL y ***se asocian con los argumentos formales*** que aparecen en el encabezamiento de la subrutina.

Como en las subrutinas no hay otro modo (excluyendo el uso siempre peligroso de variables globales) de transmtir resultados al programa que ha hecho la llamada que a través de los argumentos, conviene especificar con más detalle en la declaración de los argumentos formales qué se espera de cada argumento. Las posibilidades del cualificador INTENT son las siguientes:

- INTENT(IN): Indica que el correspondiente argumento es de entrada (INput), y que no debe ser modificado dentro de la subrutina.
- INTENT(OUT): Indica que el correspondiente argumento se utiliza como salida (Output), es decir para devolver resultados al programa principal.
- INTENT(INOUT): El correspondiente argumento se emplea para introducir datos en la función y también para transmitir resultados.

Por supuesto, los argumentos formales que son INTENT(OUT) e INTENT(INOUT), es decir, los que van a ser utilizados para devolver resultados, se deberán corresponder con argumentos actuales que sean variables, para que puedan almacenar los resultados de la función.

5.2.4 Paso de subprogramas como argumentos

Fortran 90 permite pasar a un subprograma argumentos que representan ***subprogramas de módulo***, ***subprogramas externos*** y ***funciones intrínsecas***. Estas funciones pueden estar destinadas a tareas como el cálculo de integrales definidas, la integración de ecuaciones diferenciales ordinarias, etc., problemas que pueden ser resueltos mediante varios métodos. Según qué método se desee utilizar se pasará al subprograma como argumento uno u otro nombre de subprograma de cálculo.

Los argumentos formales que reciben del argumento actual un subprograma no deben tener ningún cualificador INTENT.

Si el subprograma que se pasa como argumento es una ***función de módulo***, lo único que hay que hacer es incluir ese módulo con la sentencia USE y recibir el nombre de esa función en una variable del tipo del valor de retorno de dicha función. Después, en la parte de las sentencias ejecutables, se utiliza el nombre del argumento formal seguido de los argumentos entre paréntesis.

Si se desea pasar como argumento una ***función externa***, en la especificación del tipo del argumento formal se debe utilizar el atributo EXTERNAL, por ejemplo en la forma:

```
REAL, EXTERNAL :: nombre_de_funcion
```

También se puede utilizar la sentencia EXTERNAL, por ejemplo en la forma:

```
EXTERNAL lista_de_subprogramas
```

En el caso de las ***subrutinas*** es necesario utilizar esta segunda forma.

También se pueden pasar como argumentos las ***funciones intrínsecas*** de Fortran 90. Para ello, en el programa principal se puede utilizar el especificador INTRINSIC o la sentencia INTRINSIC, por ejemplo como se muestra a continuación:

```
TIPO, INTRINSIC :: lista_nombres_funciones_intrinsecas
INTRINSIC lista_nombres_funciones_intrinsecas
```

Dentro de la función que recibe estos nombres como argumentos y se utilizan en las expresiones correspondientes en la forma habitual.

Hay que tener en cuenta que ciertas funciones intrínsecas tienen ***nombres genéricos*** (SIN, COS, …) y ***nombres específicos*** (DSIN, CSIN, DCOS, CCOS, …) para determinados tipos de ar-

gumentos. En el paso de funciones intrínsecas como argumentos ***hay que utilizar los nombres específicos***.

Cuando un subprograma se pasa como argumento a otro subprograma, por lo general el compilador no tiene información suficiente para garantizar que el subprograma pasado como argumento se utilice correctamente dentro del subprograma que lo ha recibido, esto es, ***con el número y tipo de argumentos correctos***, y −en el caso de las funciones− asegurar también que el resultado se utiliza de modo correcto. Esta dificultad se puede resolver por medio de un bloque INTERFACE.

En la sección de declaraciones del programa principal, que va a pasar un subprograma a otro subprograma, se incluye un bloque INTERFACE, que describe con detalle las características de la función que va a ser pasada como argumento. Este bloque podría ser de la siguiente forma:

```
INTERFACE
    FUNCTION funcion_a_pasar(lista_argumentos)
        REAL :: funcion_a_pasar
        INTEGER :: lista_argumentos_integer
    END FUNCTION funcion_a_pasar
END INTERFACE
```

El cualificador EXTERNAL no se utiliza simultáneamente con un bloque INTERFACE. Tampoco se utilizan estos bloques con ***subprogramas de módulo*** o con f***unciones intrínsecas***.

6. VECTORES Y MATRICES

Una característica fundamental de cualquier lenguaje de programación es la de poder trabajar con vectores y matrices, a los que se hace referencia con un nombre genérico seguido de uno o dos subíndices.

6.1 DECLARACIÓN DE UNA VARIABLE COMO VECTOR

Para declarar un ***vector*** de nombre **x** con 1000 elementos de tipo REAL se utiliza el atributo DIMENSION seguido del nº de elementos entre paréntesis, en alguna de las formas siguientes (cuando sólo se establece un límite se trata del límite superior, y se supone que el límite inferior es igual a 1):

```
REAL, DIMENSION(1000) :: x
REAL :: y(1000)
INTEGER DIMENSION(-50:50) :: in
INTEGER :: im(-50:50)
```

La definición de ***matrices*** es similar, utilizando dos subíndices separados por comas:

```
REAL, DIMENSION(100,100) :: A
INTEGER, DIMENSION(100,4) :: ElemCuad
```

La definición de las dimensiones de los vectores y matrices se puede hacer por medio de ***constantes simbólicas***, como por ejemplo:

```
INTEGER, PARAMETER :: ini=-100, fin=100, n=400
REAL, DIMENSION(ini:fin) :: x
REAL, DIMENSION(n, n) :: A, B
```

Luego, en las sentencias ejecutables, se hace referencia a los distintos elementos de **x** y de **A** poniendo a continuación el o los índices entre paréntesis: **x(1)**, **x(j)**, **x(1000)**, **A(1,1)**, **A(i,j)**, etc. Es muy frecuente acceder a u operar con los elementos de un vector dentro de un bucle DO, y en el caso de las matrices con dos bucles DO, uno dentro de otro.

Fortran 90 admite un máximo de 7 índices, es decir, permite trabajar con hipermatrices hasta de orden 7.

6.2 RESERVA DINÁMICA DE MEMORIA

En muchas ocasiones se sabe ya en ***tiempo de compilación*** el tamaños que deben tener los vectores y matrices que se van a utilizar en un determinado programa. En otras muchas ocasiones sólo en ***tiempo de ejecución***, después de leer los datos del problema, se sabe de qué tamaño deben ser los vectores y matrices para los que hay que reservar mamoria. Una posibilidad −y así se hacía en los primeros años de los computadores− es reservar memoria para el caso más grande que se pueda presentar y que pueda ser estudiado por el ordenador disponible, pero es evidente que ésta no es una buena solución.

La mejor solución consiste en ***reservar memoria en tiempo de ejecución***, una vez que ya se conoce el tamaño del problema que se va a resolver. A esto se llama ***reserva dinámica de memoria***.

Para declarar uno o más vectores y/o matrices dejando para más adelante la definición de su dimensión se utiliza el cualificador ALLOCATABLE, como en las sentencias siguientes:

```
REAL DIMENSION(:), ALLOCATABLE :: x, y
REAL DIMENSION(:,:), ALLOCATABLE :: A, B
```

Después, en tiempo de ejecución, se da valor a la dimensión con la sentencia ALLOCATE. Las variables que se utilizan para asignar la dimensión deben ser INTEGER. Considérese el ejemplo siguiente:

```
READ *, n
ALLOCATE(x(n), y(-n:n), STAT=var1)
ALLOCATE(A(n,n), B(-n:n, -n:n), STAT=var2)
```

El argumento STAT es opcional: si está presente, la variable INTEGER llamada **var** recibe un valor cero si la reserva de memoria se hace correctamente, y un valor distinto dependiente del sistema si se produce algún error, como por ejemplo que no haya memoria suficiente.

6.3 LIBERACIÓN DE LA MEMORIA RESERVADA DINÁMICAMENTE

Cuando una variable para la que se ha reservado memoria dinámicamente ya no se va a utilizar, es conveniente liberar esa memoria con objeto de que quede disponible para otros posibles usos que puedan surgir. Para liberar la memoria se utiliza la sentencia DEALLOCATE, como por ejemplo:

```
DEALLOCATE(x, y, A, STAT=var)
```

El arumento STAT es de nuevo opcional; si está presente, la variable entera var toma valor cero si la liberación de memoria se ha hecho correctamente, y un valor distinto de cero si se ha producido algún error.

6.3.1 Inicialización de vectores: conjuntos de constantes

Un vector puede inicializarse a partir de un conjunto o vector de constantes, que es un conjunto de valores definidos entre (/ y /). Los valores pueden definirse mediante valores separados por comas o con DO implícitos, o con cualquier combinación de ambos sistemas, como por ejemplo:

```
in1 = (/ 1,3,5,7,9,11,13,15 /)
in2 = (/ (i,i=1,15,2) /)
in3 = (/ 1,3,(2*i-1, i=3,7),15 /)
```

6.4 OPERACIONES CON VECTORES Y MATRICES

Fortran 90 permite ejecutar algunas operaciones con vectores o matrices en forma parecida a como lo hace Matlab o como se haría con C++, sobrecargando los operadores aritméticos y lógicos para que pudieran realizar operaciones matriciales. Las operaciones permitidas son las siguientes:

1. Asignación de valor a partir de un escalar.
2. Asignación de valor a partir de otro vector, matriz o expresión del mismo tamaño.
3. Multiplicación por un escalar.
4. Combinación lineal de vectores o matrices.

No se permiten otras operaciones, tales como:

1. Producto escalar de vectores (se puede hacer con la función DOT_PRODUCT())
2. Producto de matrices (se puede hacer con la función MATMUL())

3. Otras operaciones matriciales (como la inversión, las diversas factorizaciones, etc.) que deben realizarse con funciones o subrutinas programadas por el usuario o importadas de distintas librerías.

Considérense los siguientes ejemplos:

```
z = x + y
w = x + ABS(y)*2
v = (x < 0) .AND. (MOD(y,3) == 1)
```

Es posible asignar a un vector o matriz el resultado de una operación vectorial o matricial (que debe ser un vector o matriz de las mismas dimensiones) o un escalar (en este caso se asigna el escalar a todos los elementos del vector). Véanse los ejemplos siguientes:

```
vector = expresion_con_vectores
vector = 6
Amat = 10
```

A la hora de operar con vectores y matrices, tanto a la derecha como a la izquierda del operador de asignación (=) se pueden utilizar partes del vector o ***subvectores*** y partes de la matriz o ***submatrices***. Para ello se pueden utilizar los tres números o variables típicos de direccionamiento de Fortran: **valor_inicial:valor_final:salto**.

```
x = y(2:20:2)
B = A(2:20:2, 1:15)
```

También se puede utilizar ***direccionamiento indirecto***, de modo que los elementos del vector o matriz que son accedidos son aquellos cuyos índices están almacenados en otro vector de enteros:

```
i = (/ 1,2,4,7,11,16,22 /)
x = y(i)
z = y( (/ 1,2,4,7,11,16,22 /) )
B = A(i, i)
```

Los subvectores y submatrices también se pueden utilizar en las sentencias de entrada/salida. En cualquier caso las dimensiones de los distintos operandos y del resultado deben ser las correctas.

6.5 SENTENCIA WHERE

La sentencia WHERE permite asignar valores a los elementos de un vector o matriz según se cumplan o no ciertas condiciones. Existen dos formas, una más sencilla y otra más general:

```
WHERE (expresión_logica_vectorial) vector1 = expresión_vectorial1
```

o bien:

```
WHERE (expresión_logica_vectorial)
    vector1 = expresión_vectorial1
    vector2 = expresión_vectorial2
    vector3 = expresión_vectorial3
ELSEWHERE
    vector4 = expresión_vectorial4
    vector5 = expresión_vectorial5
END WHERE
```

En cada una de las expresiones anteriores el tamaño del vector o matriz a la izquierda del operador de asignación debe ser igual al tamaño del resultado de la expresión vectorial o matricial a la derecha.

6.6 PASO DE VECTORES Y MATRICES COMO ARGUMENTO A FUNCIONES Y SUBRUTINAS

Existen algunas funciones intrínsecas de Fortran 90 que admiten vectores o matrices como argumentos. Algunas de las más utilizadas son las mostradas en la Tabla 6.

Función	**Significado matemático**
ALLOCATED(x)	Devuelve .TRUE. si se ha reservado memoria para x
DOT_PRODUCT(x,y)	Devuelve el producto escalar de los vectores x e y
MAXVAL(x)	Devuelve el valor máximo del vector x
MAXLOC(x)	Devuelve un vector con un elemento cuyo valor indica la posición de la 1ª vez que aparece el valor máximo de x
MINVAL(x)	Devuelve el valor mínimo del vector x
MINLOC(x)	Devuelve un vector con un elemento cuyo valor indica la posición de la 1ª vez que aparece el valor mínimo de x
PRODUCT(x)	Devuelve el valor del producto de los elementos de x
SIZE(x)	Devuelve el nº de elementos de x
SUM(x)	Devuelve el valor de la suma de los elementos de x
MATMUL(A,B)	Devuelve el resultado del producto matricial de matrices y/0 vectores.

Tabla 6. Funciones intrínsecas que admiten argumentos vectoriales o matriciales.

Existen algunas otras funciones intrínsecas en esta categoría.

Las funciones de usuario también pueden tener vectores o matrices como argumentos formales. En este caso los argumentos vectores y/o matrices deben ser declalarados tanto en el programa principal como en la función o subrutina. También los valores de retorno pueden ser un vector o una matriz.

Fortran 90 permite también pasar como argumento, además de vectores y matrices, las ***dimensiones*** de estos vectores y matrices. Por ejemplo, la función **normaVect(x,n)** puede recibir como argumentos el vector cuya norma tiene que calcular y el número de elementos de dicho vector:

```
FUNCTION normaVect(x,n)
    INTEGER, INTENT(IN) :: n
    REAL DIMENSION(n), INTENT(IN) :: x
    REAL :: normaVect=0.0
    INTEGER :: i

    DO i=1,n
        normaVect = normaVect + x(i)*x(i)
    END DO
    normaVect = SQRT(normaVect)
END FUNCTION normaVect
```

Los vectores y matrices que se pasan como argumentos pueden haber sido creados estática o dinámicamente, por medio de la sentencia ALLOCATE. Sin embargo, dentro de la función o subrutina, nunca se utiliza reserva dinámica de memoria para los argumentos.

El paso de vectores y matrices como argumentos es tan frecuente, que Fortran 90 dispone de una forma sencilla para hacer que los vectores y matrices tomen automáticamente dentro de la función o subrutina el mismo tamaño que tenían los argumentos actuales en el programa principal. Para ello basta utilizar el carácter ***dos puntos*** (:) en la sentencia DIMENSION, como por ejemplo:

```
SUBROUTINE GaussElim(A, b, x)
    REAL DIMENSION(:,:), INTENT(INOUT) :: A
    REAL DIMENSION(:), INTENT(INOUT) :: b, x
    ...
END SUBROUTINE GaussElim
```

El uso del carácter (:) supone que los índices empiezan a numerarse a partir de 1. Si el límite inferior es distinto de 1, dicho límite inferior se pone inmediatamete antes del carácter (:). Por ejemplo, para empezar a contar desde cero:

```
REAL DIMENSION(0 :), INTENT(INOUT) :: b, x
```

Si en el programa principal se define la INTERFACE de la función o subrutina, es posible utilizar también este sistema de definición automática de la DIMENSION.

Para definir dentro de una función o subrutina vectores o matrices locales que tengan el mismo tamaño que otros vectores o matrices que hayan llegado como argumentos, es posible utilizar la sentencia SIZE, como por ejemplo:

```
SUBROUTINE GaussElim(A, b, x)
    REAL DIMENSION(:,:), INTENT(INOUT) :: A
    REAL DIMENSION(SIZE(A)) :: B
    REAL DIMENSION(:), INTENT(INOUT) :: b, x
    ...
END SUBROUTINE GaussElim
```

Para que el valor de retorno de una función sea una matriz o un vector se procede de la forma estándar.

6.7 SENTENCIAS DE ENTRADA/SALIDA CON VECTORES Y MATRICES

Se pueden utilizar expresiones abreviadas en la lectura o escritura de un vector o de una matriz:

```
READ (10,*) (x(i), i=1,n)
PRINT (100, *), x(1:n)
READ *, ((A(i,j), i=1,m), j=1,n)
DO i=1,m
    READ *, (B(i,j), j=1,n)
END DO
```

En estas sentencias se han presentado dos formas equivalentes de leer las matrices **A** y **B**.

6.8 EJEMPLOS

6.8.1 Ordenar los elementos de un vector

El siguiente programa lee un vector de enteros desde el teclado y lo ordena de menos a mayor por medio del método de la burbuja, que consiste en comparar cada elemento con todos los siguientes y permutarlos cuando el orden no es el correcto. De esta forma, en cada paso se asegura que el elemento más pequeño de los que restan ocupa la primera posición entre ellos. No es un método muy eficiente, pero es muy fácil de programar.

También es fácil mantener información sobre la posición que cada elemento ocupaba en el vector inicial. Para ello basta con disponer de un segundo vector que inicialmente contiene los N primeros números naturales y realizar en él las mismas permutaciones que en los elementos del vector que se está ordenando.

```
program Reordena
   implicit none
   ! Definición de variables
   INTEGER, PARAMETER :: SIZE = 7
   INTEGER, DIMENSION(SIZE) :: vector, copia, orden
   INTEGER :: i, j, temp

   ! Lectura de datos
   PRINT '(A,I2,A)', "Introduzca los ", SIZE, " valores para ordenar:"
   READ *, (vector(i), i=1,SIZE)
   copia = vector
   DO i=1, SIZE
      orden(i) = i
   END DO

   ! Ordenación con el método de la burbuja
   DO i=1, SIZE-1
      DO j=i+1, SIZE
         IF (vector(j)<vector(i)) then
            ! se intercambian vector(i) y vector(j)
            temp = vector(j)
            vector(j) = vector(i)
            vector(i) = temp
            ! se intercambian las posiciones iniciales respectivas
            temp = orden(j)
            orden(j) = orden(i)
            orden(i) = temp
         END IF
      END DO
   END DO

   PRINT '(A)', "El vector inicial es: {"
   PRINT '(10I5)', copia
   PRINT '(A)', "}"

   PRINT '(A)', "El vector ordenado es: {"
   PRINT '(10I5)', vector
   PRINT '(A)', "}"

   PRINT '(A)', "que ocupaban los lugares: {"
   PRINT '(10I5)', orden
   PRINT '(A)', "}, respectivamente"

end program Reordena
```

7. APÉNDICE A: COMPILADOR INTEL VISUAL FORTRAN 8

Se supone que se ha instalado correctamente el compilador Intel Viasual Fortran 8. Este compilador se puede utilizar de dos formas:

1. En una ventana MS-DOS, que se abre ejecutando "Inicio, Todos los programas, Intel® software Development Tools, Intel® Fortran Compiler 8.0, Build Environment for IA-32 applications".
2. En el entorno Microsoft Visual Studio .NET Development Environment.

7.1 UTILIZACIÓN DE LA VENTANA DE COMANDOS

La compilación se realiza con el comando **ifort**, bien directamente en la línea de comandos, bien a través de un fichero ***makefile***.

Como ejemplo, créese un fichero llamado **hello.f90** en un directorio adecuado que contenga las siguientes líneas:

```
PROGRAM hello
   INTEGER I
   DO I=1,3
      WRITE (*,*) 'Hello World!'
   END DO
END PROGRAM
```

A continuación, en la ventana de commandos, cámbiese al directorio en el que ha guardado el fichero anterior y ejecute los comandos:

```
> ifort /Zi /Od hello.f90
> hello.exe
```

Respecto a la primera sentencia se pueden hacer los siguientes comentarios:

/Zi Genera información simbólica para el Debugger incluyendo números de línea en el código objeto. La opción /Zi desactiva la opción por defecto /O2 y hace que el defecto sea /Od.

/Od Desactiva la optimización del código.

Para compilar con más opciones de debugger y de modo que dé todo tipo de avisos:

```
> ifort /Zi /Od /warn:all hello.f90
```

Para compilar con el mayor nivel de optimización:

```
> ifort /O3 /G2 hello.f90
```

Para compilar con más opciones de debugger y de modo que dé todo tipo de avisos:

```
> ifort /Zi /Od /warn:all hello.f90
```

Por defecto el compilador compila y monta. El compilador es **fortcom** y el montador LINK. El compilador es capaz de manejar todas las fases correctamente, y puede ser llamado también sólo para montar.

7.2 QUÉ HACE POR DEFECTO EL COMPILADOR

Por defecto el compilador genera el ejecutable de los ficheros indicados, realizando las siguientes acciones:

1. Da mensajes de error y avisos.
2. Para aplicaciones IA-32, utiliza la opción /G7 para optimizar el código para los procesadores Pentium® 4 e Intel® Xeon™.
3. Busca los ficheros fuente en el directorio actual (el directorio en el está definido el proyecto) o en el path explícitamente especificado antes del nombre del fichero (por ejemplo, mirar en "src" cuando el path del directorio es src/test.f90).
4. Busca ficheros include y módulos en:
   a) El directorio cuyo path se define explícitamente antes del nombre del fichero (por ejemplo, busca en "src" cuando los ficheros a incluir están en src/test.f90)
   b) El directorio actual.
   c) El directorio especificado con la opción /Idir (para todos los módulos e include)
   d) El path de include definido con la variable de entorno INCLUDE (para todos los módulos e include).
   e) Cualquier directorio especificado de modo explícito en cualquier INCLUDE dentro de un fichero incluido.
5. Transmite al montador las opciones de montaje, así como las librerías definidas por el usuario. El montador busca librerías en los directorios especificados en la variable LIB, en el caso de que no se encuentren en el directorio actual.
6. Para las opciones no especificadas, el compilador utiliza las opciones por defecto o bien no hace nada.