Pedro Cuesta (C.P.D., Servicios Informáticos U.C.M)

Fortran 90

CONTENIDO

1. INTRODUCCION
Historia, Características nuevas,
Características mejoradas, Características obsoletas

2. ESTRUCTURA DEL PROGRAMA Y FORMATO FUENTE

3. TIPOS DE DATOS
3.1 Especificación
3.2 Estructuras. Tipo derivado
3.3 Punteros
3.4 Datos carácter
3.5 Matrices

4. ESTRUCTURAS DE CONTROL
Estructuras IF, CASE, DO

5. ENTRADA / SALIDA
Nuevos parámetros en sentencia OPEN, Control de no avance,
Nuevos descriptores.

6. SUBPROGRAMAS EXTERNOS
INTENT, Matriz automática, Recursión.

7. MÓDULOS

8. SUBPROGRAMAS INTERNOS.

9. INTERFACE
Interface bloks. Interface body. Overloading.
Definición de operadores. Precedencia de operadores

10. PROCEDIMIENTOS INTRÍNSECOS

11. ANEXOS
Documentación
Opciones del compilador DEC f90

Fortran 90

1. INTRODUCCIÓN

Historia

1954: Proyecto para desarrollar un sistema de programación automático que convirtiera programas escritos en notación matemática a instrucciones máquina.

1957: Primer compilador FORTRAN de IBM.

Estandarización:

En 1960 hay muchos compiladores con nuevas características, no siempre compatibles entre distintos sistemas informáticos. Un proyecto de estandarización dio lugar al FORTRAN-66, inicialmente llamado FORTRAN-IV.

Una revisión posterior creó el FORTRAN-77 (1978). Lo más característico de esta versión es

- Tipo de datos carácter.
- IF-THEN-ELSE
- Facilidades de I/O: Ficheros de acceso directo, sentencia OPEN, ...

El siguiente objetivo, que tuvo en cuenta las sugerencias de los numerosos usuarios del FORTRAN-77 e intentaba proporcionar la potencia de los nuevos lenguajes C++ y Ada, fue el Fortran-90.

Fortran sigue siendo el lenguaje de programación más ampliamente usado en aplicaciones científicas y de ingeniería.

Características nuevas en Fortran 90:

- Formato de fuente libre:

  No posiciones fijas ni columnas reservadas, comentarios, ...

- Módulos:

  Nuevas unidades de programas; para compartir datos, especificaciones, etc.

- Tipos y operadores derivados:

  Definidos por el usuario, por combinación de otros tipos.

- Operaciones en matrices:
  - Funciones intrínsecas operando en todos o parte de los elementos de una matriz.
  - Asignación dinámica de memoria a matrices (Atributo `ALLOCATABLE, POINTER`).
  - Procedimientos intrínsecos de creación, manipulación y cálculos con matrices (ej: `SUM`)

- Definiciones genéricas para bloques de procedimientos. Ej: operador `.SUMA.` genérico.

- Punteros:

  Permiten acceso dinámico a memoria. Dimensionamiento dinámico de matrices.

- Recursión:

  `RECURSIVE` en sentencia `FUNCTION` ó `SUBROUTINE`

- Especificaciones de Interface:

  Interface bloks: Describe características de un procedimiento externo, un nombre genérico, un nuevo operador ó un nuevo tipo de asignación.

Observación. Algunos compiladores de FORTRAN 77 admiten, como extensiones, varias de las características nuevas o mejoradas en Fortran-90, si bien al no ser estándar en FORTRAN-77 hay notables diferencias entre ellos.

Características mejoradas

- Formato fuente:
  Nombres de variables, lineas multisentencia.
- Nuevas funciones intrínsecas.
- Argumentos:
  Argumentos opcionales, palabras clave y alteración de orden
- I/O adicional:
  Parámetro nuevos en `OPEN`, `INQUIRE`. Carácter de no-avance
- Control adicional :
  `CASE`, `DO-ENDDO` con `CYCLE` y `EXIT`, `WHILE`. Bucles con nombres.
- Nuevos procedimientos intrínsecos:
  Operaciones matemáticas en matrices, manipulación de bits, precisión numérica,...
- Especificación adicional:
  `INTENT, OPTIONAL, POINTER, PUBLIC, PRIVATE, TARGET`

Características obsoletas

- Return alternativo (etiquetas en la lista de argumentos)
  Alternativa: Usando una variable y `GOTO` calculado ó `CASE`.
- `PAUSE`. Alternativa: (`READ`).
- `ASSIGN` y `GOTO` asignado. Alternativa: Procedimeintos internos.
- `FORMAT` asignado. Alternativa: Expresiones de caracteres.
- Descriptor `H`
- `IF` aritmético
- Variables de control Real y Double en bucles `DO. (DO R=10,50).`
- Múltiples bucles `DO` terminando en la misma sentencia. Final de bucle `DO` en una sentencia diferente de `ENDDO` ó `CONTINUE`.
- Salto a un `ENDIF` desde fuera de su bloque `IF`.

Se mantienen en Fortran 90 por compatibilidad.

2. ESTRUCTURA DE PROGRAMA Y FORMATO FUENTE

Elementos del lenguaje

Los componentes básicos del lenguaje FORTRAN son su *conjunto de caracteres*:

- Las letras `A ...Z` y `a ... z`
- Los números `0 ... 9`
- Guión underscore `_`
- Caracteres especiales
  `= : + blanco - * / ( ) , . $ ' (antiguos)`
  `! " % & ; < > ?`

Con estos caracteres se construyen *tokens* (marcas), que tienen un significado sintáctico para el compilador. Hay seis clases de marcas

Etiqueta(Label):	123
Constante:	123.456789_long
Palabras clave(Keyword):	ALLOCATABLE
Operador	.add.
Nombre:	solucion_raiz
Separador	`/ ( ) (/ /) , = => : :: ; %`

A partir de los tokens se construyen sentencias y un conjunto de sentencias forma una *unidad de programa*.

Estructura del programa

Una o más unidades de programa :

- *Programa principal* (main)
- *Subprogramas externos*: no contenidos en un *host* (principal, otro subprograma, módulo). Pueden llamarse desde otras unidades de programa. Se corresponden con los subprogramas de FORTRAN-77.
- *Módulos*: contiene definiciones, inicializaciones,...
- *Programa block data*: especifica valores iniciales.
- *Subprogramas internos*. Están contenidos dentro de un programa principal, subprograma externo ó un módulo. La unidad que le contiene es su "*host*". Sólo pueden llamarse por su *host* o por otros subprogramas internos de su host.

Formato fuente

- `INCLUDE 'fichero.f'`
  Incluye texto fuente
- Formatos:
  Dos tipos: Libre ó fijo. No deben mezclarse en un programa.
- Nombres simbólicos:
  Hasta 31 caracteres (letras, números, _) sin distinguir entre mayúsculas y minúsculas.
  `Presion_maxima_hora_1730`
- Comentarios:
  `A = B ! después de exclamación`
  Pueden usarse líneas en blanco. En formato fijo pueden usarse ‘`c`’ ó ‘`*`’ en la columna 1.
- Líneas multisentencia (en formato libre):
  `Temp = X ; X = Y ; Y = Temp`
- Operadores simbólicos relacionales:

==	.EQ.	/=	.NE.
<	.LT.	>	.GT.
<=	.LE.	>=	.GE.

```
IF (I==1) THEN; I = 2; ELSE IF (I >= 2) THEN; I = 1; END IF
```

`salida = a ==b` ! para la variable lógica salida

- Líneas de continuación:
  39 líneas de continuación (99 en DEC) en formato libre (`ffree`), con carácter & al final de la línea

Formato libre:

- Líneas de 132 caracteres
- Los blancos son significativos (`IMPLICIT NONE`, `DO WHILE`, `CASE DEFAULT`). Son opcionales en
  - Doble palabra clave comenzando por `END` o `ELSE`
  - `DOUBLE PRECISION`
  - `GO TO` `IN OUT` `SELECT CASE`
- Indicador de continuación: Carácter &

```
TCOSH(Y) = EXP(Y) +  &   ! Linea inicial de la sentencia
           EXP(-Y)       ! Continuación
TCOSH(Y) = EXP(Y) +  &   ! Linea inicial de la sentencia
        &  EXP(-Y)       ! Continuación
TCOSH(Y) = EXP(Y) + EX&
           &P(-Y)
```

Ejemplo

```
! ffree 132 car. por linea
 IF ( label .NE. 0 .AND.                                                &
  lchar .GE. 11 .AND. (buffer(:7) .EQ. 'FORMAT('  .OR.                  &
                        buffer(:7) .EQ. 'format(') THEN
  IF (len-lenst.GE.2) stamnt(lenst+1:lenst2) = ' very long char&
      &acter string '
  lenst = MIN(lenst+2,len)
  GO TO 99
 ENDIF

! ffixed 72 car. por linea (132 con opcion de compilacion)
      IF ( label .NE. 0 .AND.
     x  lchar .GE. 11 .AND. (buffer(:7) .EQ. 'FORMAT('  .OR.
     x                       buffer(:7) .EQ. 'format(') THEN
         IF (len-lenst.GE.2) stamnt(lenst+1:lenst2) =   ' very long char
     xacter string '
         lenst = MIN(lenst+2,len)
         GO TO 99
      ENDIF
```

Ejemplo válido para todos los formatos

```
Columna  1         2                                                   7
12345678901234567890                                                   3
------------------------------------------------------------------------

! define funcion de usuario my_sin
      DOUBLE PRECISION FUNCTION my_sin(X)
          my_sin = x - x**3/factor(3) + X**5/factor(5)                 &
     &               - x**7/factor(7)
      CONTAINS
        INTEGER FUNCTION factor(N)
         factor = 1
         DO 10 i = n, 1, -1
 10      factor = factor*I
        END FUNCTION factor
      END FUNCTION my_sin
```

Compilación

`f90 -ffree   file.for` (defecto para ficheros con ext `.f90`)

`f90 -ffixed  file.for` (defecto para ficheros con ext `.f`)

3. TIPOS DE DATOS

3.1 Especificación

A las sentencias de especificación

```
REAL, INTEGER, CHARACTER, DIMENSION, LOGICAL, COMPLEX
```

se añaden

```
POINTER, TARGET, ALLOCATABLE, PUBLIC, PRIVATE, INTENT,
OPTIONAL
```

Las sentencias

```
    REAL  VAR1,VAR2
    DIMENSION VAR1(10), VAR2(10)
    TARGET VAR1, VAR2
```

se pueden sustituir por el nuevo formato de especificación:

```
REAL, DIMENSION(10), TARGET       :: VAR1, VAR2
Tipo, lista de atributos (comas)  :: Lista de variables
```

Este formato es requerido en algunos casos, como en los tipos derivados.

Ejemplos de declaraciones válidas

```
    DOUBLE PRECISION b(6)
    INTEGER(KIND=2) i
    REAL(KIND=4) x, y
    REAL(4) x, y
    LOGICAL, DIMENSION(10,10) :: array_a, array_b
    INTEGER,                 PARAMETER                 ::
    smallest=SELECTED_REAL_KIND(6,70)
    REAL(KIND (0.0)) m
    COMPLEX(KIND=8) :: d
    TYPE(EMPLOYEE) :: manager
    REAL, INTRINSIC :: cos
    CHARACTER(15) prompt
    CHARACTER*12, SAVE :: hello_msg
    INTEGER COUNT, MATRIX(4,4), sum
    LOGICAL*2 switch
    REAL :: x = 2.0                  ! valores iniciales
```

La tabla siguiente muestra las 12 propiedades o **atributos** que pueden especificarse entre el tipo de dato y ::

Atributo	**En F-77**	**Uso**
ALLOCATABLE	NO	Las cotas de los subíndices no se determinarán hasta el momento de la ejecución.
DIMENSION(espec.)	SI	Indica cotas para los subíndices o que éstas se determinarán en ejecución.
EXTERNAL	SI	Una función se define por (1) subprograma function no contenido en un programa principal o un módulo; (2) no por Fortran.
INTENT(espec.)	NO	Tipo de argumento: Entrada y/o Salida.
INTRINSIC	SI	Función o subrutina implementada.
OPTIONAL	NO	Puede omitirse el argumento.
PARAMETER	SI	Nombre para un valor constante que no puede modificarse en ejecución.
POINTER	NO	Variable que no contiene datos sino que apunta a un área de memoria que almacena el dato de interés.
PRIVATE	NO	Un nombre dentro de un módulo no es accesible fuera del módulo.
PUBLIC	NO	Un nombre dentro de un módulo si es accesible fuera del módulo.
SAVE	SI	Una variable retiene todas sus propiedades, incluido su valor, después de salir del subprograma.
TARGET	NO	Indica que un nombre refiere a un área de memoria que puede ser apuntada por una variable con atributo POINTER.

Parámetro KIND

Especifica tipo de dato. Si no se especifica se asume el tipo por defecto.

Por ejemplo DEC Fortran 90 provee tres tipos de parámetro KIND para datos de tipo real:

```
REAL(KIND=4) (or REAL*4)
REAL(KIND=8) (or REAL*8)       (DOUBLE PRECISION)
REAL(KIND=16) (or REAL*16)
```

Puede usarse como en los ejemplos de especificación anteriores o en conjunción con las funciones KIND y Selected_Real_Kind

```
INTEGER, PARAMETER :: K10 = Selected_Real_Kind(10)
INTEGER, PARAMETER :: Dbl = KIND(1.0D0)
REAL (KIND=k10) a,b,c
REAL (KIND=Dbl) d1, d2, d3     ! puede omitirse KIND
d1=1.0_Dbl
```

La constante 1.0 se almacena usando la representación del tipo inidicado por `Dbl`. Equivalente a 1.0_8

Función KIND

Devuelve el tipo de dato del argumento

what_kind.f90

```
PROGRAM what_kind
REAL x
DOUBLE PRECISION xx
COMPLEX cc
ix = KIND(0)         ; PRINT *,'def integer kind    = ',ix
iy = KIND(x)         ; PRINT *,'def real    kind    = ',iy
id = KIND(xx)        ; PRINT *,'def double  kind    = ',id
ic = KIND(cc)        ; PRINT *,'def complex kind    = ',ic
iz = KIND(.false.)   ; PRINT *,'def logical kind    = ',iz
iw = KIND("a")       ; PRINT *,'def charact kind    = ',iw
END
```

Salida:

```
def integer kind  = 4
def real    kind  = 4
def double  kind  = 8
def complex kind  = 4
def logical kind  = 4
def charact kind  = 1
```

Nota. Existen 5 tipos de datos: INTEGER, REAL, COMPLEX, LOGICAL, CHARACTER. Cada uno tiene distintas representaciones, cada una especificada por un KIND.

INTEGER: KIND=2, KIND=4.
REAL: KIND=4, KIND=8, KIND=16.

Función Selected_Real_Kind(*P*,*R*)

Devuelve el tipo de dato real adecuado a la precision y rango especificados en los argumentos

`SELECTED_REAL_KIND(6,70)=8`
P= precisión decimal de al menos P dígitos
R= rango del exponente

Si ese tipo no es posible devuelve
−1 si no es posible la precisión
−2 si no es posible el rango
−3 ninguno de los dos

select_real.f90

```
 PROGRAM select_real
 lix = 0
 DO i = 1,100
   ix = SELECTED_REAL_KIND(i,1)
   IF (ix.ne.lix) THEN
     PRINT *,'digitos mantisa=',i,' kind= ',ix
   ENDIF
   lix = ix
ENDDO
END
```

```
 digitos mantisa=   1  kind=  4
 digitos mantisa=   8  kind=  8
 digitos mantisa=  16  kind= 16
 digitos mantisa=  34  kind= -1
```

Función Selected_Int_Kind(*R*)

Devuelve el tipo de dato entero adecuado para representar todos los valores n en el rango $-10^R < n < 10^R$. Vale -1 si no es posible ese tipo.

`SELECTED_INT_KIND (6) = 4`

select_int.f90

```
 PROGRAM select_int
 lix = 0
 DO i = 1,20
   ix = SELECTED_INT_KIND(i)
   IF (ix.ne.lix) THEN
     PRINT *,'nuevo tipo para i=',i,' kind= ',ix
   ENDIF
   lix = ix
```

```
ENDDO
END
```

Salida:

```
 nuevo tipo para i=  1  kind=  1
 nuevo tipo para i=  3  kind=  2
 nuevo tipo para i=  5  kind=  4
 nuevo tipo para i= 10  kind=  8
 nuevo tipo para i= 19  kind= -1
```

Ejercicio: Programa tipo Select_real.f90 para determinar el tipo de dato real adecuado para exponentes en el rango de 1 a 10000

3.2 Estructuras. Tipo derivado

Son entidades agregadas que contienen uno o más elementos.
(Elementos ≡ Campos ≡ Componentes).

Diferencias con un array:

- Su creación es un proceso en dos pasos
    1. Definición de la forma con una declaración de estructura (TYPE) multisentencia. Puede haber subestructuras.
    2. Declaración del registro con un nombre simbólico estableciendo su estructura en memoria.
- Pueden tener campos de diferentes tipos.
- Cada elemento del registro tiene su propio nombre.

Ejemplo

```
PROGRAM tipo                                ! Tipo derivado: programador
 TYPE alumno
   CHARACTER (len=12) :: nombre ! Tipo intrínseco
   REAL               :: nota(2)
 END TYPE alumno
```

Definición de registro:

```
TYPE (alumno) alum(30)
TYPE (alumno), DIMENSION(20) :: alumrep
```

Referencias:

```
alum(8)%nombre,   alum(8)%nota(2)

alum(1) = alumno('PEREZ' , 6.5 , 7.3)
```

Podemos referenciar a las componentes individuales o al tipo agregado global.

3.3 Punteros

- Son variables a las que se les asigna el atributo "pointer". Una variable con el atributo pointer puede usarse como una variable ordinaria y además de otras formas adicionales.

- Una variable puntero se asocia (alias) con un objeto destino (target) de un tipo específico: contiene su dirección de memoria y otra información descriptiva.

- En fortran, se debe pensar en los punteros como "alias" en lugar de como posiciones de memoria. Su declaración crea y asigna memoria al puntero, pero no crea un objeto destino.

- Cada puntero puede tener tres estados
  1. **Indefinido**, que es la situación al comienzo del programa.
  2. **Null**, que significa que no es alias de ningún dato objeto.
  3. **Asociado** a alguna variable destino.

- NULLIFY(*ptr*) hace que *ptr* apunte a NULL, ningún destino

- ASSOCIATED es una función intrínseca que devuelve *false* si su argumento es un puntero que apunta a NULL.

  También ASSOCIATED(*P*1,*P*2), ASSOCIATED(*P*1,*R*).

- ALLOCATE(*P*1) crea espacio para un número real y hace que *P*1 sea el alias para ese espacio. *P*1 = 9.2 almacena el número real 9.2 en el espacio previamente reservado y no inicializado.

- DEALLOCATE(*P*1) libera el espacio y deja a *P*1 indefinido.

Ejemplo

```
REAL, TARGET :: r,s
REAL, POINTER :: p1,p2
r = 5.3 ; s = 8.1
p1 => r ! guarda en p1 la dirección de r y otra información descriptiva
p2 => p1
PRINT *,p2+4,p1,r
p1 => s
PRINT *,p2+4,p1,r
END
```

```
    p2+4 = 9.3, p1 = 5.3, r = 5.3
    p2+4 = 9.3, p1 = 8.1, r = 5.3
```

`r` siempre está asociado con la misma área de memoria. Es una variable ordinaria.

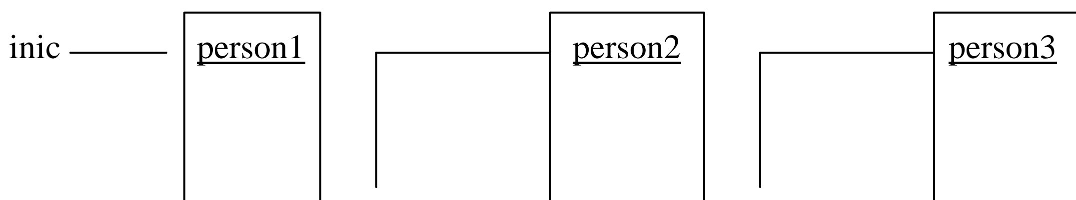
`p1` puede asociarse con distintas posiciones de memoria a lo largo de la ejecución del programa

- Una variable apuntada por un puntero debe tener el atributo "target"·
- En una asignación de punteros (=>) se transfiere el status de un puntero a otro.
- En una asignación ordinaria entre punteros (=) estos se consideran como "alias" de sus destinos.
- Usualmente apuntan a un array de valores o a un tipo derivado, pues rara vez merece la pena la complicación que supone usar punteros para apuntar a valores escalares de tipos intrínsecos.

Ejemplo. Uso de punteros para almacenar datos en memoria sin dimensión previa. (Listas enlazadas).

punter.f90

```
PROGRAM punter
 TYPE person
   INTEGER :: edad
   CHARACTER (len=50) :: nomb
    TYPE(person), POINTER :: hacia    ! apunta a un tipo
person
 END TYPE person

 TYPE (person), pointer :: inic,sig
 TYPE (person) :: temp
 INTEGER :: ios

 temp%edad = 0
 temp%nomb = ' '
 NULLIFY(temp%hacia)
 ALLOCATE(inic)       !   reserva memoria para el primer
elemento
 IF (.NOT. ASSOCIATED(inic)) STOP 'error de memoria'

! lectura de lista

  sig => inic    ! puntero => destino. El puntero es un
alias
               ! para su destino
 DO
   sig = temp    ! inicializa sig , puede suprimirse
   read(*,*,iostat=ios) sig%edad,sig%nomb   ! lee sig
   IF (ios < 0) EXIT
   IF (sig%edad < 0) EXIT
   ALLOCATE(sig%hacia)
   IF  (.NOT.  ASSOCIATED(sig%hacia))  STOP  'error  de
memoria'
   sig => sig%hacia
 END DO

! escribe lista

 sig => inic
 DO
   IF (.NOT. ASSOCIATED(sig%hacia)) EXIT
   WRITE (*,*) sig%edad,sig%nomb
   sig => sig%hacia
 END DO
END
```



edad nomb sig ——	edad nomb sig ——	edad nomb null

Con dos punteros (derecha, izquierda) se construyen árboles binarios.

3.4 Datos carácter

- La longitud de una constante carácter puede ser de 0 a 2000.
- Pueden usarse como delimitadores un apóstrofo (') o comillas (").

funcar.f90

```
PRINT "(' ', I3, ' isn''t', I3)", i,j
```

- Se permiten asignaciones del tipo

```
res(3:5) = res(1:3)  ! solapamiento
res(3:3) = res(3:2) ! no asignación de una subcadena nula
```

Funciones intrínsecas de caracteres

`CHAR(I,[kind])` Carácter en la posición I en la secuencia del procesador.

`ICHAR(C)` Posición que ocupa el carácter C en la secuencia del procesador.

`INDEX(STR,SUB_STR,[back])` Posición de comienzo de SUB_STR en STR.

`LGE, LGT, LLE, LLT(STR_A,STR_B)` Comparaciones lexicograficas.

Nuevas:

`ACHAR(I)` Carácter en la posición I en la secuencia ASCII

`IACHAR(C)` Posición que ocupa el carácter C en la secuencia ASCII.

`ADJUSTL(STR)` Ajusta a la izquierda moviendo los blancos iniciales al final.

`ADJUSTR(STR)` Ajusta a la derecha moviendo los blancos finales al principio.

`LEN_TRIM(STR)` Longitud de STR sin los blancos finales.

`REPEAT(STR,N)` Concatena STR consigo mismo N veces

`SCAN(STR,SET,[BACK])` Posición del primer caracter por la izquierda de STR que está en SET. 0 si ninguno está.

`TRIM(STR)` Subcadena inicial obtenida al suprimir los blancos finales.

`VERIFY(STR,SET,[BACK])` Posición del primer carácter por la izquierda de STR que no está en SET. 0 si están todos.

`ACHAR` coincide con `CHAR`

3.5 Matrices

Declaración

```
REAL, DIMENSION (1:9)                  :: x,y
CHARACTER (LEN=8), DIMENSION (0:17) :: lista
REAL, DIMENSION (:,:), ALLOCATABLE  :: a,b
```

a y b , matrices de dos dimensiones (rango 2), de tamaño establecido posteriormente con una sentencia `ALLOCATE`

Ejemplo

```
 PROGRAM matri
  DIMENSION a(100), b(100), c(200)
  n = 100
  . . .
  CALL sub(a,b,c,n)
  . . .
 END
 SUBROUTINE sub(x,y,z,n)
   DIMENSION x(100)
   DIMENSION y(n)
   DIMENSION z(*)  !
!    REAL, DIMENSION(:) :: z
   REAL, DIMENSION(SIZE(z)) :: locz
   DIMENSION temp(2*n+1)                 ! error en FORTRAN-77
   REAL, DIMENSION(:), ALLOCATABLE :: work
   . . .
   ALLOCATE (work(2*n))
 . . .
 END
```

- `x` matriz (dummy) de tamaño constante
- `y` matriz (dummy) de tamaño ajustable. El tamaño se determina por el argumento `n`.
- `z` matriz (dummy) de tamaño asumido. El tamaño se determina por el del correspondiente argumento real `c`.
- `locz` matriz (actual, local) cuyo tamaño es el mismo que `z`. Matriz automática.
- `temp` matriz (actual, local) de tamaño ajustable. Determinado por el argumento `n`. Matriz automática.
- `work` matriz (actual,local) de tamaño (*deferred*: demorado, retardado) determinado en cualquier punto de la subrutina y en función del argumento `n`. Reserva de memoria dinámica.

Constructores de matrices:

Inicialización o asignación de un conjunto de valores de la matriz. Sintaxis similar a la de la sentencia `DATA` pero con más posibilidades. Hay tres formas posibles:

1. Expresión escalar (REAL X(4))

   ```
   X = (/ 1.2, 3.5, 1.1, 1.5 /)
   ```

2. Expresión matricial

   ```
   X = (/ A(I,1:2), A(I+1,2:3) /)
   ```

   Asigna por columnas en `A(1:2,1:2)`

3. DO implícito

   ```
   X = (/ (SQRT (REAL (I)), I=1,4) /)
   ```

Estos constructores se restringen a matrices de rango 1 (vectores). Para utilizarlos en matrices de mayor rango puede usarse la función intrínseca `RESHAPE`

El ejemplo siguiente muestra como usar RESHAPE para crear una matriz multidimensional D(2,3)

```
E = (/2.3, 4.7, 6.6/)
D = RESHAPE(SOURCE = (/3.5, (/2.0, 1.0/), E/), SHAPE = (/2,3/))
```

D es una matriz de rango 2 con dimensiones (shape) (2,3) conteniendo los elementos:

```
3.5     1.0     4.7
2.0     2.3     6.6
```

Se pueden utilizar para una linea (o parte) de la matriz::

construct.f90

```
PROGRAM constr
 REAL, DIMENSION(2,3) :: a2
 a2(1,1:3) = (/11,12,13/)
 a2(2,1:3) = (/21,22,23/)
 PRINT *,a2
END
```

Secciones y asignaciones:

```
C=(/4,8,7,6/)  C=[4,8,7,6]
D=(/ (I,I=1,5,2) /)  D=(/1:5:2/)
A( (/1,7,3,2/), 1) = (/1,2,3,4/)  ! vector de subíndices.

DIMENSION  A(10), B(10), C(5),  X(5,5)
INTEGER  IND(10)
```

EXPRESION	EQUIVALE A
`A = B`	`DO I=1,10` `A(I) = B(I)`
`C = A(3:7)`	`DO I=1,5` `C(I)=A(I+2)`
`C = X(:,3)`	`DO I=1,5` `C(I)=X(I,3)`
`B(1:5) =` `X(3,1:5)`	`DO J=1,5` `B(J)=X(3,J)`
`A(2:10:2) = C`	`DO I=1,5` `A(2*I)=C(I)`
`A = SIN(B)`	`DO I=1,10` `A(I)=SIN(B(I))`

`X(1,:) = 1.0`
`X(2,:) = 2.0`
	`DO J=1,5`
	`X(1,J)=1.0`
	`X(2,J)=2.0`

`A = B(IND)`	`DO I=1,10` `A(I)=B(IND(I))`

- Asignaciones de matrices se permiten bajo dos circunstancias: cuando la expresión de la derecha es un escalar o cuando las dos expresiones son matrices (o secciones) de igual forma.

  `A(2:4,5:8) = A(3:5,1:4)` ! válido pues $3\times4 \equiv 3\times4$.
  `A(1:4,1:3) = A(1:2,1:6)` ! no válido pues $4\times3 \neq 2\times6$.

- Pueden hacerse correspondencias de asignación entre secciones dispersas

```
DIMENSION S(17), T(10,10), S1(6), T2(5,3)
...
S1(::1) = S(2:17:3)
! ó S1 = S(2:17:3)
T2(::1,::1) = T(1:10:2,2:10:3)
```

Sustituir un bucle por una operación matricial no siempre es equivalente. Por ejemplo

```
      DIMENSION a(10)
      n=10
      DO 5 i=1,n
 5      A(i)=i

      DO 10 i=2,n-1
        a(i)=a(i-1)+a(i+1)   !      Operacion escalar
 10   CONTINUE

      PRINT *,a
      END
```

Resultado:

```
1,. 4., 8., 13., 19., 26., 34., 43., 53., 10.
```

no es equivalente a

```
      DIMENSION a(10)
      n=10
      DO 5 i=1,n
 5      A(i)=i

      a(2:n-1)=a(1:n-2)+a(3:n))   !    Operacion vectorial

      PRINT *,a
      END
```

Resultado:

```
1,. 4., 6., 8., 10., 12., 14., 16., 18., 10.
```

La expresión matricial se evalua resolviendo previamente el lado derecho completamente. El programador deberá determinar si una operación con sintaxis matricial es equivalente a un bucle DO.

Sentencia WHERE

Se pueden asignar valores en aquellos elementos que cumplan una condición:

```
WHERE (x < 0) yb = 0
WHERE (c /= 0)
   a = b/c
ELSEWHERE
   a = 0
   c = 1
END WHERE
```

Ejemplo:

```
PROGRAM demo_where
INTEGER, DIMENSION(5)      :: a
INTEGER, DIMENSION(-1:3)   :: b
a(1)  = 2; a(2) = -4; a(3) = 6; a(4) = -8; a(5) = 10
b(-1) = 1; b(0) =  2; b(1) = 3; b(2) =  4; b(3) =  5

WHERE (a < 0) b = 0                ! sentencia WHERE
PRINT "(5I4)", b

WHERE (b /= 0)                     ! bloque WHERE
   a = a/b
ELSEWHERE
   a = 0
END WHERE
PRINT "(5I4)", a
END PROGRAM demo_where
```

Operadores intrínsecos

Los operadores intrínsecos (+, –, /, ...) y funciones se pueden aplicar a matrices y operan independientemente en cada elemento.

`a*b` multiplica los elementos correspondientes de dos matrices de la misma forma.

`m(k,k:n+1) = m(k,k:n+1)/3.5` divide cada elemento por la constante.

Funciones matriciales intrínsecas

Una máscara (`MASK`) es una array que especifica los elementos de otro array sobre los que se opera.

MULTIPLICACIÓN (numérica o lógica)

`DOT_PRODUCT (VEC_A,VEC_B)`	Producto escalar de dos vectores.
`MATMUL (MAT_A,MAT_B)`	Multiplicación matricial.

REDUCCION (completa ó secciones)

`MAXVAL (MAT[,DIM][,MASK])`	Máximo valor en una matriz.
`MINVAL (MAT[,DIM][,MASK])`	Mínimo valor en una matriz.
`PRODUCT (MAT[,DIM][,MASK])`	Producto de los elementos.
`SUM (MAT[,DIM][,MASK])`	Suma de los elementos.
`ALL, ANY, COUNT.`	

INTERROGACIÓN

`ALLOCATED (MAT)`	`TRUE` si tiene reserva de memoria
`LBOUND (MAT[,DIM])`	Cota inferior en una dimensión específica o vector de cotas inferiores.
`UBOUND (MAT[,DIM])`	Cota superior o vector de cotas superiores.
`SHAPE (SOURCE)`	Vector que contiene la extensión de cada dimensión.
`SIZE (MAT[,DIM])`	Número total de elementos en la matriz o en una dimensión.

CONSTRUCCIÓN (nuevas matrices a partir de elementos de mat. existentes)

`PACK (MAT,MASK[,VECTOR])`	Crea un vector a partir de los elementos de una matriz seleccionados por `MASK`.
`UNPACK (MAT,MASK[,VECTOR])`	Crea una matriz a partir de un vector.
`MERGE, SPREAD.`	

MANIPULACIÓN

`CSHIFT (MAT,SHIFT[,DIM])`	Movimiento circular.
`RESHAPE (SOURCE, SHAPE [,PAD][,ORDER])`	Toma los elementos de `SOURCE` en la secuencia de almacenamiento y los combina en una matriz de forma `SHAPE`
`TRANSPOSE (MAT)`	Transpuesta de una matriz de rango 2.

LOCALIZACION

`MAXLOC (MAT[,MASK])`	Localización (subíndices) del máximo
`MINLOC (MAT[,MASK])`	Localización del mínimo

Programar un ejemplo de cada función

4. ESTRUCTURAS DE CONTROL

- Tres tipos de estructuras de control: IF, CASE, DO
- Un conjunto de sentencias controladas por una estructura de control es un *bloque*.
- No se permite transferir el control dentro de un bloque desde fuera, pero se permite abandonar un bloque con sentencias como EXIT y CYCLE
- Puede asignarse un nombre a la estructura

Estructuras IF (Como en FORTRAN-77)

```
PROGRAM pr_if
  INTEGER :: c1,c2,c3
  READ *, c1,c2,c3
  context: IF (c1 < 0) THEN
     contint1: IF (c2 >= 0 ) THEN
          PRINT *,"c1 < 0 y c2 >= 0"
     END IF contint1
  ELSE IF (c1 == 0 ) THEN context
     PRINT *,"c1 es cero"
  ELSE context
     contint2: IF (c3 /= 0 ) THEN
        PRINT *,"c1 > 0 y c3 no cero"
     END IF contint2
  END IF context
END PROGRAM pr_if
```

No se ejecuta más de un bloque de sentencias.

Estructuras CASE

```
PROGRAM pr_case
  INTEGER :: dia
  READ *,dia

  SELECT CASE  (dia)  ! expresión escalar de tipo
                      ! entero,caracter o lógico
  CASE (2:5)
     PRINT *,"Lunes a Jueves: 8 AM 9 PM"
  CASE (6)
     PRINT *,"Viernes: 9 AM 8 PM"
  CASE (7,1)
     PRINT *,"Sabado y Domingo: Cerrado"
  CASE DEFAULT
     PRINT *,"Codigo de entrada erroneo"
  END SELECT

END PROGRAM pr_case
```

Los rangos de valores en los bloques deben ser disjuntos. Pueden usarse como rango expresiones del tipo `(:-1)`, que incluye todos los enteros negativos.
Pueden usarse rangos de valores caracter, como "0000":"9999"

El selector `CASE` puede ser de tipo caracter de cualquier longitud.
Cada valor en `CASE` debe ser del mismo tipo que el de la expresión `SELECT CASE`

pr_casec.f90

```
PROGRAM pr_casec

  CHARACTER(9) :: dia_nom
  CHARACTER(2) :: dia_abr
  READ "(A)",dia_nom
  CALL nomabr(dia_nom,dia_abr)    ! Abreviatura de dos caracteres

  SELECT CASE  (dia_abr)
  CASE ("LU","MA","MI","JU")
     PRINT *,"Lunes a Jueves: 8 AM 9 PM"
  CASE ("VI")
     PRINT *,"Viernes: 9 AM 8 PM"
  CASE ("SA","DO")
     PRINT *,"Sabado y Domingo: Cerrado"
  CASE DEFAULT
     PRINT *,"Codigo de entrada erroneo"
  END SELECT

END PROGRAM pr_casec

SUBROUTINE nomabr(dia_nom,dia_abr)

  CHARACTER(*), INTENT (IN)  :: dia_nom
  CHARACTER(2), INTENT (OUT) :: dia_abr
  CHARACTER(27), PARAMETER ::                       &
      Minus = "abcdefghijklmnñopqrstuvxyz" ,        &
      Mayus = "ABCDEFGHIJKLMNÑOPQRSTUVXYZ"
  INTEGER :: pos,i

  dia_abr = ADJUSTL (dia_nom)
  DO i=1,2
     pos = INDEX (Minus,dia_abr(i:i))
     IF (pos /= 0) dia_abr(i:i)=Mayus(pos:pos)
  END DO

END SUBROUTINE nomabr
```

Si la estructura `CASE` tiene un nombre, éste debe aparecer tanto en `SELECT CASE` como en `END SELECT` pero su presencia en las sentencias `CASE` es opcional.

Estructuras DO

Ejemplos:

```
      DO 10 I=1 ,M
        ...
 10   CONTINUE

      DO 20 I=1 ,M
        ...
 20   END DO
```

No deben usarse como índices variables reales, si se quiere compatibilizar con posteriores versiones.

```
CONTADOR: DO I=1 ,M
   ...
ENDDO CONTADOR


BUCLE_WHILE: DO WHILE (Expresion logica)
   ...                                     !DO[,]WHILE coma opcional
   IF (A(I) .LE. 0.0) CYCLE   !  nueva iteración
   ...
ENDDO BUCLE_WHILE


BUCLE_SIEMPRE: DO
   ...
   IF (IER <  0.0) EXIT   !   sale fuera
   ...
ENDDO BUCLE_SIEMPRE
```

Cuando se utilizan EXIT, CYCLE en bucles anidados puede usarse su forma general

```
EXIT  [Nombre]
CYCLE [Nombre]
```

Cuando no aparece Nombre, se refiere al bucle más interior.

Ejemplo

```
DO
  READ(…,IOSTAT=st) buffer
  IF(st==End_of_file) EXIT
  ! procesa buffer
  …
ENDDO
```

```
READ(…,IOSTAT=st) buffer
DO WHILE(st/=End_of_file)
  !procesa buffer
  …
  READ(…,IOSTAT=st) buffer
ENDDO
```

<table>
<tr><th>OBSOLETO</th><th>EQUIVALENTE</th></tr>
<tr><td colspan="2">DO CONTROL REAL</td></tr>
<tr><td>REAL :: x
DO x = 0.1,0.5,0.1
  PRINT “(F5.1)”,x
ENDDO</td><td>INTEGER :: i
DO i = 1,5
PRINT “(F5.1)”,0.1*i
ENDDO</td></tr>
<tr><td colspan="2">IF ARITMETICO</td></tr>
<tr><td>INTEGER :: n=1
IF (n-1) 10,20,30
10 PRINT *,’n es menor que 1’
   GOTO 40
20 PRINT *,’n es igual a 1’
   GOTO 40
30 PRINT *,’n es mayor que 1’
40 CONTINUE</td><td>INTEGER :: n=1
IF (n<1) THEN
  PRINT *,’n es menor que 1’
ELSE IF (n==1) THEN
  PRINT *,’n es igual a 1’
ELSE IF (n>1) THEN
  PRINT *,’n es mayor que 1’
ENDIF</td></tr>
</table>

5. ENTRADA/SALIDA

Nuevos parámetros en la sentencia open

`POSITION='REWIND','APPEND','ASIS'`

Especifica la posición en el fichero que se abre. Un fichero nuevo se posiciona en su punto inicial. Un fichero que ya existe puede posicionarse en su punto inicial si `REWIND` ó al final si `APPEND`. `ASIS` es la opción por defecto y no reposiciona el fichero.

`ACTION='READ','WRITE','READWRITE'`

Especifica el tipo de transferencia de datos permitida.

`DELIM='APOSTROPHE','QUOTE','NONE'`

Especifica el delimitador usado para los datos carácter escritos por listas directas o `NAMELIST`.

`STATUS='REPLACE'`

Es equivalente a `STATUS='UNKNOWN'`. Si el fichero no existe le crea y si existe le borra y se creará otro con el mismo nombre.

`PAD='YES' o 'NO'`

Tratamiento, en lectura formateada, del caso en que el registro tiene menos caracteres de los requeridos por la lista de entrada y su formato asociado.

Si YES: completa el registro con blancos (defecto). Si NO: error.

La sentencia `INQUIRE` se adapta para incorporar estos nuevos parámetros. Además la sentencia `INQUIRE` puede utilizarse para calcular la longitud de una lista no formateada que se especificará posteriormente en el parámetro `RECL` de una sentencia `OPEN` de un fichero de acceso directo. Esto garantiza la portabilidad de programas. (Algunas implementaciones calculan la longitud en bytes y otras en palabras).

recl.f90

```
INQUIRE (IOLENGTH=length) lista de variables
OPEN (...,RECL=length,...)
```

En un READ	si se produce un error	`IOSTAT` > 0
	si final de fichero	`IOSTAT` < 0

Control de no-avance

La lista de control de una sentencia de entrada o salida formateada puede incluir el especificador

`ADVANCE='YES' ó ADVANCE='NO'`

Con "no avance" no se cambia la posición en el fichero antes ó después de una transferencia de datos

Ejemplo

```
PROGRAM entsal
 suma=0
 DO i=1,5
  READ  (5,FMT='(I3)', ADVANCE='NO') ival
  WRITE (6,FMT='(I4)', ADVANCE='NO') ival
  suma=suma+ival
 ENDO
 WRITE(6,FMT='( " suma= ", I5)') suma
END PROGRAM entsal
```

Linea de entrada:

```
  1  2  3  4  5
---------------
```

Linea de salida:

```
   1   2   3   4   5 SUMA=    15
--------------------------------
```

En un `READ` con no avance puede especificarse el parámetro `SIZE=`número de caracteres transferidos desde el fichero de entrada durante la ejecución del `READ`, sin tener en cuenta los posibles blancos añadidos al final. Puede utilizarse para leer registros de longitud variable y determinar sus longitudes.

Matrices completas y estructuras

En las listas de entrada/salida se puede especificar el nombre de una matriz, denotando globalmente todos sus elementos. Tambien puede especificarse una sección de una matriz o una estructura.

Nuevos descriptores

Descriptores B:binario, O:octal y Z:hexadecimal para datos enteros

```
PRINT "(E11.3)", 12300.0
                            0.123E+05
                          -----------
```

Parte significativa mayor o igual que 0 y menor que 1.

```
PRINT "(ES11.3)", 12300.0
                            1.230E+04
                          -----------
```

Parte significativa mayor o igual que 1 y menor que 10.
Excepto cuando vale 0.

```
PRINT "(EN11.3)", 12300.0
                           12.300E+03
                          -----------
```

Parte significativa mayor o igual que 1 y menor que 1000. Excepto cuando vale 0. Exponente múltiplo de 3.

<table>
<tr><th colspan="2">OBSOLETO</th><th colspan="2">EQUIVALENTE</th></tr>
<tr><td colspan="4" align="center">DESCRIPTOR H</td></tr>
<tr><td colspan="2">PRINT ”(32H Don´t use
    obsolescent features!)”</td><td colspan="2">PRINT ”(’ Don’’t use
   obsolescent features!’)”</td></tr>
<tr><td colspan="4" align="center">PAUSE</td></tr>
<tr><td>…
PAUSE
…</td><td colspan="3">CHARACTER :: ignora_car
WRITE (*,”(A)”,ADVANCE=NO) “Presione <Return>”
READ (*,”(A)”) ignora_car</td></tr>
<tr><td colspan="4" align="center">FORMATO ASIGNADO</td></tr>
<tr><td colspan="2">   INTEGER :: ifmt
1000 FORMAT (‘Salida 1’)
2000 FORMAT (‘Salida 2’)
     ASSIGN 2000 TO ifmt
     PRINT ifmt,…</td><td colspan="2">CHARACTER(80) :: fmt
CHARACTER(80), PARAMETER :: &
       fmt1=”(‘Salida 1’)”, &
       fmt2=”(‘Salida 2’)”
fmt = fmt2
PRINT fmt,…</td></tr>
</table>

6. SUBPROGRAMAS EXTERNOS

- Hay tres categorias de subprogramas:

– Subprograma *interno* (contenido dentro de un programa o subprograma)
– Subprograma *módulo* (define y empaqueta datos y/o funciones, subprogramas,...)
– Subprograma *externo* (como en F77)

- Desde el punto de vista de la organización, un programa completo consiste de *unidades* que pueden ser: *programa principal*, *subprogramas externos* ó *módulos*. Estas unidades pueden compilarse separadamente.

INTENT():

atributo para declarar si un argumento es de entrada (`IN`), de salida (`OUT`) o de entrada/salida (`INOUT`)

Ejemplo

```
SUBROUTINE cambia_car (si_car,nu_car,arg_car,num_car)
 CHARACTER(1), INTENT (IN)      :: si_car     ! entrada
 CHARACTER(1), INTENT (IN)      :: nu_car     ! entrada
 CHARACTER(*), INTENT (IN OUT)  :: arg_car    ! cambiado
 CHARACTER(1), INTENT (OUT)     :: num_car    ! salida
 INTEGER :: i   ! Local

 num_car = 0
 DO i = 1, LEN(arg_car)
   IF (arg_car(i:i) == si_car) THEN
     arg_car(i:i) = nu_car
     num_car = num_car + 1
   ENDIF
 ENDO
END SUBROUTINE cambia_car

PROGRAM pru_cambia_car
 CHARACTER(6) :: var_car = '  1.2 '
 INTEGER      :: num_car

 PRINT '(2A)', '  Antes de CALL, var_car = ',var_car
 CALL cambia_car (' ','0', var_car, num_car)
 PRINT '(2A)', 'Despues de CALL, var_car = ',var_car
 PRINT  '(A,I1)',  'Numero  de  caracteres  cambiados  =
',num_car
END PROGRAM cambia_car
```

Salida:

```
  Antes de CALL, var_car =   1.2
Despues de CALL, var_car = 001.20
Numero de caracteres cambiados = 3
```

Matriz automática:

Matriz declarada en un subprograma, que no es un argumento ficticio y su dimensión depende de valores no constantes.

```
 REAL, DIMENSION(n) :: temp      ! n argumento del subprograma
```

Puede utilizarse la función SIZE `(SIZE(a,1), SIZE(a,2)`.
También es válido para las longitudes de variables carácter.

Recursion

`RESULT`:

Permite que una función devuelva el valor de una variable cuyo nombre es distinto que el de la función. Esto permite la recursión directa.
El programador debe conocer el tipo de dato que devuelve la función.

Un subprograma recursivo es aquel que se llama a sí mismo directa o indirectamente. Debe declararse, en ese caso como `RECURSIVE`. Una función recursiva requiere usar una variable `RESULT`.

Frecuentemente un subprograma recursivo puede programarse con algun método iterativo más eficiente.

```
PROGRAM Binomial
 IMPLICIT NONE
 INTEGER N, K
 READ *, N, K
 PRINT *, ' N = ', N, ' K = ', K, ' C = ', &
     Factorial(N) / (Factorial(K) * Factorial(N-K))
 CONTAINS
 RECURSIVE INTEGER FUNCTION Factorial (K) RESULT (Valor)
   ! Valor mismo tipo que Factorial
   INTEGER K
   IF (K <= 1) THEN
     Valor = 1
   ELSE
     Valor = K * Factorial (K-1)
   ENDIF
 END FUNCTION Factorial
END PROGRAM Binomial
```

`Factorial` es un subprograma interno

Notas:

- El alcance (*scope*) del nombre de una variable puede ampliarse, respecto a las reglas generales de F77, mediante el uso de un módulo. En F77 los nombres de procedimientos externos son globales, mientras que las variables escalares, matrices y constantes son locales al subprograma.

- Una rutina que llama a una función externa devolviendo un tipo derivado debe suministrar una declaración de tipo para tal función

- Son equivalentes

```
FUNCTION func (arg1,arg2)
REAL (KIND(0D0)) :: func
```

y

```
REAL (KIND(0D0)) FUNCTION func (arg1,arg2)
```

7. MÓDULOS

- Un modulo permite empaquetar definiciones de datos y compartir datos entre diferentes unidades de programas compiladas por separado.
- En su uso básico ofrece posibilades similares a `INCLUDE`. Adicionalmente permite y facilita:
  - Compartir datos en ejecución. (≈ `COMMON`).
  - Inicializar variables, por lo que no se necesitaría `BLOCK DATA`
  - Incluir subrutinas y funciones

```
MODULE Datos_globales
  REAL, PARAMETER :: Pi   = 3.14159265358979, Inch = 2.54
END MODULE
```

Ejemplo. Uso de un módulo para compartir datos

```
PROGRAM prog_compartir
   USE datos
   IMPLICIT NONE
   INTEGER :: i
   READ *, numval
   ALLOCATE (valores(numval))
   DO i = 1, numval
      valores(i)=i**i
   END DO

   CALL escribe

END PROGRAM prog_compartir

SUBROUTINE escribe
   USE datos
   IMPLICIT NONE
   INTEGER :: i
   DO i = 1, numval
      print '(f7.1)', valores(i)  ! carácter global
   END DO
END SUBROUTINE escribe

MODULE datos
   INTEGER  :: numval
   REAL, ALLOCATABLE, DIMENSION(:) :: valores
END MODULE datos
```

Global Allocatable Arrays

- Fortran-90 prohibe la presencia de arrays dinámicas en COMMON pues la longitud del COMMON se determina en compilación. Solución: Poner arrays dinámicas en un módulo y en las subrutinas que determinan tamaño, leen y escriben se pone USE. Ver ejemplo anterior.

- Tampoco está permitido poner una matriz "allocatable" en un argumento ficticio de un subprograma.

Ejemplo. Módulo que contiene una definición de tipo, un bloque interface y un subprograma de función.

```
MODULE intervalo_aritmetico
   TYPE intervalo
      REAL inf, sup
   END TYPE intervalo
   INTERFACE OPERATOR(+)
      MODULE PROCEDURE suma_intervalos
   END INTERFACE

CONTAINS
¡                                    Función que devuelve un tipo derivado
   FUNCTION suma_intervalos(a,b)
      TYPE(intervalo), INTENT(IN) :: a,b
      TYPE(intervalo)  suma_intervalos
      suma_intervalos%inf = a%inf + b%inf
      suma_intervalos%sup = a%sup + b%sup
   END FUNCTION suma_intervalos

END MODULE intervalo_aritmetico
```

Ejercicio: Programa que pruebe la función `suma_intervalos` y el operador + entre intervalos.

- El acceso del programa principal, un subprograma u otro módulo al contenido del módulo se llama ***use association*** y se realiza por la sentencia:

  ```
  USE datos
  ```

  Accede a las especificaciones y variables del módulo datos, con los valores previamente asignados, posiblemente en otra unidad de programa.

- Las variables y datos de los módulos tienen un alcance **global** en todas las unidades con *use association*, mientras que `INCLUDE`, `COMMON` tienen un caracter local en la subrutina donde aparecen. El atributo `PRIVATE` puede modificar este caracter global (ver private.f90).

```
MODULE datos
  REAL :: var1=1.1, var2=2.2
  PRIVATE var1
END MODULE datos
PROGRAM pr
  USE datos
  PRINT *,var1,var2                0,   2.2
  var1=11.1; var2=22.2
PRINT *,var1,var2                  11.1 22.2
  CALL sub
END PROGRAM pr
SUBROUTINE sub
  USE datos
PRINT *,var1,var2                  0,   22.2
END SUBROUTINE sub
```

- Pueden inicializarse variables en un módulo:
  - En BLOCK DATA

  ```
  DATA  X /1.0,2.0,3.0,4.0,5.0, 995*0.0/
  ```

  En un constructor de F90 995*0.0 sería un cero.

  - En MODULE

  ```
  REAL,DIMENSION(1000)::X=(/1.0,2.0,3.0,4.0,5.0,(0.0,i=6,1000)/)
  ```

- Un módulo puede usar otro módulo y la asociación use es transitiva.
- El uso de módulos facilita la definición de funciones que devuelven un tipo derivado o una matriz.
- El módulo debe compilarse antes o ponerse al principio del fichero fuente. Crea un fichero *.mod que es el que lee la sentencia USE $\Rightarrow$ la compilación de una unidad debe estar precedida por la compilación de todos los módulos usados por ella.

8. SUBPROGRAMAS INTERNOS

- Es un subprograma contenido en un subprograma externo.
- Su uso es adecuado, a efectos de organización, cuando sólo es llamado desde un único programa ó subprograma (**host**).
- *Ejemplo*

  Alcance de variables:

```
SUBROUTINE exterior
   REAL x,y
   :
CONTAINS

   SUBROUTINE interior
     REAL y
     y = x + 1
     :
   END SUBROUTINE interior

END SUBROUTINE exterior
```

- La subrutina interior accede a las variables de exterior por **host association** (`x`), mientras que `y` es local debido a que aparece en una declaración de tipo.

  En este ejemplo hay dos zonas de alcance (*scoping unit*):

  exterior–interior
  interior.

Ejercicio: Programa que pruebe las zonas de alcance asignando valores e imprimiendo `x` e `y` en ambas zonas del ejemplo anterior.

- Las etiquetas son locales: Si una sentencia con etiqueta, ésta debe estar en el mismo *scoping unit* que la sentencia que la referencia (⇒ no GOTO de un programa interno a un host).
- IMPLICIT NONE afecta al host y a los subprogramas internos debajo de CONTAINS.
- Un host siempre conoce todo acerca del interface con un subprograma interno ⇒ no es necesario una declaración de tipo en el host para una función interna.
- Un subprograma interno no contiene él mismo directamente un subprograma interno.

- Si se repite la definición de un tipo derivado en diferentes *scoping unit* el estándar obliga a especificar la sentencia `SEQUENCE` en cada definición. Por este motivo es preferible que la definición se ponga en un módulo.

- Fortran-90 tiene cuatro categorías de *scoping unit* que siempre residen dentro de otro *scoping unit* :

  a) Definición de tipo derivado
  b) Cuerpo interface
  c) Subprograma interno
  d) Subprograma módulo (subrutina dentro de un módulo)

  a), c) y d) siempre tienen acceso a los nombres del “host scoping unit” via “host association”.

  b) no; su entorno de datos es completamente independiente de su “host scoping unit”. Ni siquiera le afectará un IMPLICIT situado en el scoping unit donde está contenido.

- Un modulo puede contener subprogramas internos (su alcance no será global como un subprograma externo). Un ejemplo típico es un módulo con un tipo derivado y subprogramas módulo que ejecutan operaciones en datos de ese tipo derivado. Ej: tipo derivado para números racionales y operaciones *, /, +, – .

Ejemplo

Función interna que devuelve un vector

pro_sort.f90

```
PROGRAM pro_sort

  INTEGER, PARAMETER     :: num=5
  INTEGER, DIMENSION(num):: inicial=(/3,5,1,4,2/)
  INTEGER, DIMENSION(num):: ordenado

  ordenado=sort(inicial); print *,ordenado

CONTAINS

  FUNCTION sort (entrada) RESULT (salida)
    INTEGER,DIMENSION(:),INTENT(IN)   :: entrada
    INTEGER,DIMENSION(SIZE(entrada)) :: salida
    LOGICAL :: no_camb
    INTEGER :: i,t

    salida = entrada
    DO
      no_camb = .TRUE.
      DO i=1,SIZE(salida)-1
        IF (salida(i) > salida(i+1)) THEN
          t=salida(i);             salida(i)=salida(i+1);
salida(i+1)=t
          no_camb = .FALSE.
        END IF
      END DO
      IF (no_camb) EXIT
    END DO
  END FUNCTION sort

END PROGRAM pro_sort
```

9.INTERFACE

Interface bloks

Una referencia a un módulo o subprograma interno se considera un interface *explícito*: el compilador puede ver todos los detalles. Una referencia a un procedimiento externo se realiza usualmente por un interface *implícito*: el compilador asume los detalles. En este último caso se puede suministrar un interface explícito, que consiste en la cabecera, las especificaciones de los argumentos y la sentencia `END`, constituyendo un *interface body*.

Interface body

- Con el objetivo de que las nuevas características puedan emplearse tanto con los módulos y procedimientos internos como con los subprogramas externos se introduce un mecanismo conocido como "cuerpo interface".

- Permite al programa especificar explícitamente todos los detalles acerca de la comunicación con el procedimiento externo: subrutina o función, número, orden y tipo de los argumentos, tipo de resultado de función, etc. En muchos casos no es obligatorio, como en F77. (ver ejemplo `pro_opera)`.

- Por ejemplo, si con un argumento ficticio `x` de tamaño asumido se utiliza despues `SIZE(x)`, entonces no es suficiente que el compilador pase la dirección de memoria inicial de `x` si no también el tamaño ⇒ se requiere el cuerpo interface .

```
REAL FUNCTION minimo(a,b,func)
  REAL, INTENT(IN) :: a,b  ! especificaciones sobre a y b
  INTERFACE                ! especificaciones sobre func
    REAL FUNCTION func(x)
    REAL, INTENT(IN) :: x
    END FUNCTION func
  END INTERFACE
  REAL f,x
  :
  f = func(x)   ! llamada a la funcion del usuario
  :
END FUNCTION minimo

REAL FUNCTION func(x)
   REAL, INTENT(IN) :: x
   func(x) = x**2 - 6*x + 9
END FUNCTION func
```

- Se pueden formar con una copia del subprograma, eliminando aquellas partes no relevantes al interface. Los nombres de las variables usadas en el interface son totalmente independientes de los nombres de fuera. El cuerpo interface es algo aislado del resto del host.

- Un interface para un procedimiento externo es **explícito** si se suministra un cuerpo interface; en otro caso es **implícito**.

- Los "cuerpos interface" nunca se usan con procedimientos internos, intrínsecos, módulos o funciones sentencia.

- Se requieren en los siguientes casos:
  - Argumento ficticio de tamaño asumido: se especifica el rango pero no el tamaño.
  - Función que devuelve una matriz de valores
  - Función con resultado tipo caracter cuya longitud no se especifca ni por constante ni por *.
  - Función con resultado tipo POINTER.
  - Hay un argumento ficticio con atributo OPTIONAL
  - Hay un argumento ficticio con atributo POINTER ó TARGET
  - Cuando el subprograma se llama con palabras clave en los argumentos
  - Cuando se llama a un procedimiento por un nombre genérico.
  - Cuando aparecen operadores definidos por el usuario (+, AND, ...) o el signo = con un significado redefinido.

- Esto permite una comprobación completa al compilar entre los argumentos actuales y ficticios.

Ejemplo

```
PROGRAM opc

  INTERFACE
    SUBROUTINE sub1 (uno,dos,tres)
      INTEGER, OPTIONAL :: dos ; INTEGER  uno ,tres
    END SUBROUTINE
  END INTERFACE
  :
  CALL sub1(5,tres=23)
  :
END PROGRAM opc
```

Ejemplo

Argumentos opcionales. Palabras clave de argumentos.

opc_arg.f90

```
PROGRAM opc_arg
  ! interface body obligatorio debido a los argumentos  opcionales

  INTERFACE
    SUBROUTINE fecha_hora (fecha, hora, estilo)
      CHARACTER(8), INTENT (OUT), OPTIONAL :: fecha
      CHARACTER(5), INTENT (OUT), OPTIONAL :: hora
      INTEGER     , INTENT (IN) , OPTIONAL :: estilo
    END SUBROUTINE fecha_hora
  END INTERFACE
```

! En el interface los nombres fecha, hora, estilo pueden ser diferentes que los de la
! subrutina, pero en el CALL deben aparecer los nombres del INTERFACE.

```
  CHARACTER(8) ::fecha  ; CHARACTER(5) :: hora

  CALL fecha_hora (fecha)
  PRINT '(1X,A)', fecha

  CALL fecha_hora (fecha,hora)
  PRINT '(1X,A,2X,A)', fecha,hora

  CALL fecha_hora (fecha,estilo=3)
  PRINT '(1X,A)', fecha

  CALL fecha_hora (estilo=2, fecha=fecha)
  PRINT '(1X,A)', fecha

  CALL fecha_hora (hora=hora)
  PRINT '(1X,A)', hora

END PROGRAM opc_arg
```

```
SUBROUTINE fecha_hora (fecha, hora, estilo)

  CHARACTER(8), INTENT (OUT), OPTIONAL :: fecha
  CHARACTER(5), INTENT (OUT), OPTIONAL :: hora
  INTEGER     , INTENT (IN) , OPTIONAL :: estilo

  INTEGER               :: loc_estilo
  INTEGER, DIMENSION(8) :: valores
  INTRINSIC DATE_AND_TIME, MOD, PRESENT

  CALL DATE_AND_TIME (VALUES=valores)

  IF (PRESENT (fecha)) THEN
    IF (PRESENT (estilo)) THEN
      loc_estilo = estilo
    ELSE
      loc_estilo = 1     ! Defecto
    ENDIF
    valores(1) = MOD (valores(1),100)
    SELECT CASE (loc_estilo)
      CASE (1)
        WRITE (fecha,'(I2.2,A1,I2.2,A1,I2.2)')      &
               valores(2),'/',valores(3),'/',valores(1)
      CASE (2)
        WRITE (fecha,'(I2.2,A1,I2.2,A1,I2.2)')      &
               valores(3),'/',valores(2),'/',valores(1)
      CASE (3)
        WRITE (fecha,'(I2.2,A1,I2.2,A1,I2.2)')      &
               valores(1),'/',valores(2),'/',valores(3)
    END SELECT
  END IF

  IF (PRESENT (hora)) WRITE (hora,'(I2.2,A1,I2.2)') &
                      valores(5),':',valores(6)

END SUBROUTINE fecha_hora
```

Salida:

```
06/03/96
06/03/96  16.41
96/06/03
03/06/96
16:41
```

Si estilo no está presente no puedo hacer estilo=1 (segmentation fault).

Los nombres de los argumentos de la subrutina o función en el cuerpo interface no tienen que corresponder con los nombres de los argumentos en el subprograma. Sólo importa la correspondencia posicional

OVERLOADING. DEFINICIÓN DE OPERACIONES.

- Fortran90 permite que un mismo nombre pueda ejecutar diferentes procedimientos. Por ejemplo se puede extender el significado de los signos intrínsecos +, = con procedimientos definidos por el programador. El signo = se extendería a asignaciones no intrínsecamente posibles. Igualmente, pueden definirse nuevos operadores, simbolizados por una secuencia de letras entre puntos (`.suma.`).
- El programador extiende la asignación "=" con subrutinas y los operadores con funciones.
- Fortran permite llamar a una familia de funciones intrínsecas por su nombre genérico. El compilador identifica la adecuada en función del argumento. F90 extiende esta capacidad a procedimientos escritos por el programador.
- Un ***identificador genérico*** aparecerá siempre en un *bloque* `INTERFACE`, que le asociará con funciones o subrutinas.

Ejemplo. Nombre genérico definido por el operador:

mod_cambio.f90

```
MODULE mod_cambio
INTERFACE cambio
  MODULE PROCEDURE cambio_i,cambio_r, cambio_c     ! (*)
END INTERFACE

CONTAINS

 SUBROUTINE cambio_i(x,y)
   INTEGER , INTENT(IN OUT) :: x,y
   INTEGER temp
    temp = x; x = y; y = temp
 END SUBROUTINE cambio_i

 SUBROUTINE cambio_r(x,y)
   REAL , INTENT(IN OUT) :: x,y
   REAL temp
   temp = x; x = y; y = temp
 END SUBROUTINE cambio_r

 SUBROUTINE cambio_c(x,y)
   COMPLEX , INTENT(IN OUT) :: x,y
   COMPLEX temp
   temp = x; x = y; y = temp
 END SUBROUTINE cambio_c

END MODULE mod_cambio
```

(*) indica que cualquier referencia a `cambio` se tomará como una referencia a una de las 3 subrutinas especificadas. En la llamada, los argumentos deben coincidir en tipo con alguna de las tres situaciones.

pro_cambio.f90

```
PROGRAM pro_cambio
 USE mod_cambio

 INTEGER i,j ; REAL x,y ; COMPLEX c,d

  x = 1.0; y = -1.0
  print *,x,y; call cambio (x,y); print *,x,y
  i = 1  ; j = -1
  print *,i,j; call cambio (i,j); print *,i,j
  c = (1.0,1.0); d = (-1.0,-1.0)
  print *,c,d; call cambio (c,d); print *,c,d

! print *,x,d; call cambio (x,d); print *,x,d  ! error de
                                              ! compilacion

END PROGRAM pro_cambio
```

- EL modulo está en el fichero delante del programa principal. Lo mismo si se compila en un fichero aparte.
- El ejemplo es válido si en lugar de complejos ponemos vectores.
- En lugar de un módulo pueden usarse subrutinas externas.

Ejemplo

Redefinición del operador `EQ.` (y su alias ==). La sentencia `MODULE PROCEDURE` puede utilizarse cuando puede accederse al subprograma referenciado en el mismo módulo o mediante un `USE` a otro módulo.

num_raci.f90

```
MODULE num_raci                          ! módulo
  TYPE :: tipo_raci
    INTEGER :: num,denom
  END TYPE tipo_raci
  INTERFACE OPERATOR (.EQ.)           ! asocia .EQ.  y == a la
    MODULE PROCEDURE es_igual_a       ! funcion  es_igual_a
  END INTERFACE

CONTAINS
                                          ! función Módulo
  FUNCTION es_igual_a (rac_1,rac_2) RESULT (igual)
    LOGICAL     :: igual
    TYPE (tipo_raci), INTENT (IN)  :: rac_1,rac_2
    TYPE (tipo_raci)               :: tmp_1,tmp_2

    tmp_1 = reduce(rac_1)
    tmp_2 = reduce(rac_2)
    igual = (tmp_1%num   == tmp_2%num)    .AND.     &
```

```
          (tmp_1%denom == tmp_2%denom)
END FUNCTION es_igual_a
```

```
                                            ! función Módulo
  FUNCTION reduce (rac) RESULT (red)
    TYPE (tipo_raci), INTENT (IN)  :: rac
    TYPE (tipo_raci)               :: red
    INTEGER :: mcd         !  maximo comun divisor
    INTEGER :: calc_mcd    !  funcion externa para el m.c.d.

    mcd = calc_mcd(rac%num,rac%denom)
                                            ! asignación de un tipo derivado
    red = tipo_raci (rac%num/mcd, rac%denom/mcd)
  END FUNCTION reduce

END MODULE num_raci
                                            ! función entera
RECURSIVE FUNCTION calc_mcd (m,n) RESULT (mcd)
  INTEGER, INTENT (IN) :: m, n
  INTEGER              :: mcd
  IF (n == 0) THEN
    mcd = m
  ELSE
    mcd = calc_mcd (n, MOD (m,n))
  ENDIF
END FUNCTION calc_mcd
                                            ! programa principal
PROGRAM pro_raci

  USE num_raci
  TYPE (tipo_raci) :: rac_1 = tipo_raci (2,4)    ,  &
                      rac_2 = tipo_raci (3,6)

  IF (rac_1 .EQ. rac_2) THEN
    PRINT '(A)','2/4 igual a 3/6'
  END IF
  IF (rac_1 == tipo_raci (4,6)) THEN
    !
  ELSE
    PRINT '(A)','2/4 diferente de 4/6'
  END IF

END PROGRAM pro_raci
```

- Definir el significado de `.EQ.` (==) no implica nada acerca del significado de `.NE.` (\=).

- Se recomienda que operaciones y asignaciones de usuario se utilicen sólo en situaciones donde esté involucrado al menos un tipo derivado.

- *Ejercicio*: Definir un operador unario `.invert.` que devuelva (`RESULT`) el racional inverso y un programa que le pruebe.

- En Fortran-90 se extiende la asignación = (númerico, carácter, lógica) al caso en que ambos lados tenemos el mismo tipo derivado. Se puede extender a otros casos como tipo-

derivado=vector ó vector=tipo-derivado. En el ejemplo siguiente se define tipo-derivado=variable-carácter y viceversa.

Ejemplo

Redefinición de operadores. Se define un interface explícito. (El programa pro_opera.f90 es equivalente, utilizando procedimientos internos)

pro_opera2.f90

```
MODULE mod_opera   ! para definir tipo string
  TYPE string
    INTEGER lon
    CHARACTER(len=80) :: string_dat
  END TYPE string
END MODULE mod_opera

PROGRAM pro_opera

  USE mod_opera

  INTERFACE ASSIGNMENT (=)
  SUBROUTINE c_asiga_s (s,c)
    USE mod_opera
    TYPE(string), INTENT(INOUT)   :: s
    CHARACTER(len=*), INTENT (IN) :: c
  END SUBROUTINE c_asiga_s
  SUBROUTINE s_asiga_c (c,s)
    USE mod_opera
    TYPE(string), INTENT(IN)         :: s
    CHARACTER(len=*), INTENT (INOUT) :: c
  END SUBROUTINE s_asiga_c
  END INTERFACE

  INTERFACE OPERATOR (.merge.)
    FUNCTION string_merge (s1,s2) RESULT (s3)
     USE mod_opera
     TYPE(string), INTENT(IN) :: s1,s2
     TYPE(string) :: s3
    END FUNCTION string_merge
  END INTERFACE

  TYPE(string) :: str1, str2, str3
  CHARACTER(len=80) :: ch
  str1 = 'ABCDEF '          ! c_asigna_s
  str2 = '12345678'              ! c_asigna_s
  str3 = str1.merge.str2         ! string_merge  y = entre tipos
derivados
  ch = str3                 ! s_asigna_c
  PRINT *, ch
!  PRINT *, str3
```

```
END PROGRAM pro_opera
```

```
SUBROUTINE c_asiga_s (s,c)
  USE mod_opera
  TYPE(string), INTENT(INOUT)   :: s
  CHARACTER(len=*), INTENT (IN) :: c
  s%string_dat = c
  s%lon        = len(c)
END SUBROUTINE c_asiga_s

SUBROUTINE s_asiga_c (c,s)
  USE mod_opera
  TYPE(string), INTENT(IN)         :: s
  CHARACTER(len=*), INTENT (INOUT) :: c
  c = s%string_dat(1:s%lon)
END SUBROUTINE s_asiga_c

FUNCTION string_merge (s1,s2) RESULT (s3)
  USE mod_opera
  TYPE(string), INTENT(IN) :: s1,s2
  TYPE(string) :: s3
  l = MIN(79,MAX(s1%lon,s2%lon))
  j = 1
  DO i = 1, l
    s3%string_dat(j:j+1) = s1%string_dat(i:i) // &
                           s2%string_dat(i:i)
     j = j + 2
  ENDDO
     s3%lon = 2*l
END FUNCTION string_merge
```

Salida:

```
A1B2C3D4E5F6 7 8
```

A pesar del `USE mod_opera` en el host es necesario `USE mod_opera` dentro del interface pues en el estándar el cuerpo interface no tiene acceso a ningún nombre de fuera.

__Nota__. Como se ha visto en los ejemplos hay cuatro tipos de sentencia `INTERFACE`:

```
INTERFACE

INTERFACE (.EQ. ó .merge.)

INTERFACE ASSIGNMENT (=)

INTERFACE nombre-genérico (función ó subrutina)
```

- Los procedimientos que extienden la asignación `=` son subrutinas con dos argumentos ficticios. El primero es el que aparecerá a la izquierda

```
c_asigna_s(s,c) ⇒ s = c
                  ↓   ↓
              INOUT  IN
```

- Cuando `INTERFACE nombre-genérico` todos los procedimientos deben ser subrutinas ó todos funciones. `nombre-genérico` puede ser el de un procedimiento intrínseco.
- Cuando `INTERFACE OPERATOR` todos los procedimientos deben ser funciones, con uno ó dos argumentos (`IN,IN`).
- Cuando `INTERFACE ASSIGNMENT (=)` todos los procedimientos deben ser subrutinas con dos argumentos (`INOUT,IN`).
- No se puede redefinir `=` cuando ambas entidades son numéricas, caracteres ó lógicas. Pero sí se puede redefinir el significado de = entre tipos derivados.

Precedencia de operadores

Los operadores intrínsecos tienen la precedencia usual. Los operadores definidos por el usuario tienen la precedencia más alta si son unarios y la más baja si son binarios. Las dos expresiones siguientes se ejecutan con diferente precedencia si se quitan los paréntesis

```
vector3 =  mat    *   vector1  + vector2
vector3 = (mat .por. vector1) + vector2
```

Operador programado binario
**
*, /
unario +, -
binario +, -
.EQ.,==,.NE.,/=,.LT.,<,.LE.,<=,.GT.,>,.GE.,>=
.NOT.
.OR.
.EQV.,.NEQV.
Operador programado binario

Precedencia de operadores de mayor a menor

10.PROCEDIMIENTOS INTRÍNSECOS

- Las funciones y subrutinas intrínsecas son suministradas por el propio lenguaje.

- Todas pueden referenciarse con palabras clave en los argumentos y muchas tienen argumentos opcionales.

  `CALL DATE_AND_TIME (TIME=t)`

- Adiccionalmente existen funciones suministradas por el sistema operativo, independientemente del lenguaje.

  (DEC: `lib$init_timer`, `lib$show_timer`, `lib$stat_timer`).

- Se agrupan en cuatro categorías:

  1. *Elemental*: su argumento principal es un escalar o una matriz. Si el argumento es una matriz, la función actúa separadamente en cada uno de sus elementos y el resultado es una matriz de la misma forma.

     `SQRT(a), ABS(x)`

  2. *Inquiry*: El resultado depende de alguna propiedad del argumento, pero no de su valor. Por ejemplo si el argumento de `LEN` es una expresión escalar de tipo `CHARACTER`, el resultado es el máximo número de caracteres que el argumento puede contener, independientemente de los caracteres que en ese momento tenga el argumento.

     `LEN(c), PRECISION(a)`

  3. *Transformacional*: La mayoría tienen un argumento matriz y el resultado es una matriz de diferente forma

     `RESHAPE(a,b), MAXVAL(a)`

  4. *SUBROUTINE*: Se llaman mediante un `CALL`

Ejemplos

Algunas **funciones** intrínsecas nuevas que pueden ser útiles en cálculos numéricos:

`CEILING(x)`	El menor entero mayor o igual que `x`
`FLOOR(x)`	El mayor entero menor o igual que `x`
`MODULO(a,p)`	Resto de la división de a entre `p` , con el mismo signo que `p`.
`EXPONENT(x)`	Exponente de `x` en su representación de número real.
`FRACTION(x)`	Parte fraccional de `x` en su representación de número real.

`EPSILON(x)`	Devuelve el número positivo más pequeño del mismo tipo y precisión que `x`.
`HUGE(x)`	Mayor número del mismo tipo que `x`.
`PRECISION(x)`	Dígitos de precisión

Subrutinas:

`DATE_AND_TIME`	Obtiene fecha y hora
`MVBITS`	Copia bits
`RANDOM_NUMBER`	numero aleatorio
`RANDOM_SEED`	Accede o especifica semilla
`SYSTEM_CLOCK`	Accede al reloj del sistema

Ejemplo

Funciones numericas tipo **inquiry**

num_inq.f90

```
PROGRAM num_inq
  DOUBLE PRECISION double
  INTEGER(KIND=2) :: int2
  OPEN (6,FILE='num_inq.sal')

  WRITE (6,*)  '    REAL'
  WRITE (6,*)  'digits        =', digits(real)
  WRITE (6,*)  'epsilon       =', epsilon(real)
  WRITE (6,*)  'huge          =', huge(real)
  WRITE (6,*)  'maxexponent   =', maxexponent(real)
  WRITE (6,*)  'minexponent   =', minexponent(real)
  WRITE (6,*)  'precision     =', precision(real)
  WRITE (6,*)  'radix         =', radix(real)
  WRITE (6,*)  'range         =', range(real)
  WRITE (6,*)  'tiny          =', tiny(real)

  WRITE (6,*)  '    DOUBLE'
  WRITE (6,*)  'digits        =', digits(double)
  WRITE (6,*)  'epsilon       =', epsilon(double)
  WRITE (6,*)  'huge          =', huge(double)
  WRITE (6,*)  'maxexponent   =', maxexponent(double)
  WRITE (6,*)  'minexponent   =', minexponent(double)
  WRITE (6,*)  'precision     =', precision(double)
  WRITE (6,*)  'range         =', range(double)
  WRITE (6,*)  'tiny          =', tiny(double)

  WRITE (6,*)  '    INTEGER'
  WRITE (6,*)  'digits        =', digits(integer)
  WRITE (6,*)  'huge          =', huge(integer)
  WRITE (6,*)  'radix         =', radix(integer)
  WRITE (6,*)  'range         =', range(integer)
```

```
  WRITE (6,*)  '     INT2'
  WRITE (6,*)  'digits       =', digits(int2)
  WRITE (6,*)  'huge         =', huge(int2)
  WRITE (6,*)  'radix        =', radix(int2)
  WRITE (6,*)  'range        =', range(int2)

END PROGRAM
```

Salida:

```
    REAL
 digits       =          24
 epsilon      =  1.1920929E-07    !  ≈ tipo y precisión
 huge         =  3.4028235E+38    !
 maxexponent  =         128
 minexponent  =        -125
 precision    =           6       !
 radix        =           2
 range        =          37       !
 tiny         =  1.1754944E-38    !  ≈ tipo
    DOUBLE
 digits       =          53
 epsilon      =  2.220446049250313E-016
 huge         =  1.797693134862316E+308
 maxexponent  =        1024
 minexponent  =       -1021
 precision    =          15
 range        =         307
 tiny         =  2.225073858507201E-308
    INTEGER
 digits       =          31
 huge         =  2147483647
 radix        =           2
 range        =           9
    INT2
 digits       =          15
 huge         =  32767
 radix        =           2
 range        =           4
```

Ejemplo. Generacion de numeros aleatorios

random.f90

```
PROGRAM random
  REAL  z(2,2), y
  INTEGER, DIMENSION(1) :: seed = 9576591
  CALL RANDOM_SEED (PUT = seed)   ! Semilla propia
  CALL RANDOM_NUMBER(HARVEST = y)
  CALL RANDOM_NUMBER(z)
  PRINT *, y
  PRINT *, z
END PROGRAM
```

`HARVEST` es la palabra clave que nombra el argumento de `RANDOM_NUMBER`

Salida:

```
4.7963569E-03
0.2795743   0.9185119   0.6959528   0.7653494
```

DOCUMENTACIÓN

Utilizada en este manual:

1. "*INTRODUCTION OF FORTRAN 90*". CRAY RESEARCH INC.

2. "*DEC FORTRAN 90*". Language Reference Manual and User Manual. Digital Equipment Corporation 1994.

3. "*PROGRAMMER'S GUIDE TO FORTRAN 90*". BRAINERD W.S., GOLDBERG C.H., ADAMS J.C. 1990. McGraw-Hill.

4. "*UPGRADING TO FORTRAN 90*". COOPER REDWINE. 1995. Springer-Verlag.

5. CERN: http://wwwcn.cern.ch/asdoc/f90.html or via anonymous ftp from asisftp.cern.ch in the directory cnl as the file f90tutor.ps. An ASCII copy of this material as a set of slides for a six-hour course is available from metcalf@cern.ch.

11. ANEXOS

OTRA DOCUMENTACIÓN

Note: additional information on Fortran products is availble on WWW with the URL http://www.fortran.com/fortran.

LIBROS EN INGLES:

- Fortran 90 - Meissner, PWS Kent, Boston, 1995, ISBN 0-534-93372-6.
- Fortran 90 - Counihan, Pitman, 1991, ISBN 0-273-03073-6.
- Fortran 90 and Engineering Computation - Schick and Silverman, John Wiley, 1994, ISBN 0-471-58512-2.
- Fortran 90, A Reference Guide - Chamberland, Prentice Hall PTR, 1995, ISBN 0-13-397332-8.
- Fortran 90/95 Explained - Metcalf and Reid, Oxford University Press, 1996, ISBN 0-19-851888-9, about $33. This book is a complete, audited description of the Fortran 90 and Fortran 95 languages in a more readable style than the standards themselves. It incorporates all X3J3 and WG5's interpretations and has a complete chapter on Fortran 95. It has seven Appendices, including an extended example program that is available by ftp and solutions to exercises, as well as an Index. US e-mail orders may be sent to: orders@oup-usa.org. The Fortran 90 version is also available in French, Japanese and Russian (see below).
- Fortran 90 for Scientists and Engineers - Brian D. Hahn, Edward Arnold, 1994, ISBN 0-340-60034-9.
- Fortran 90 Handbook - Adams, Brainerd, Martin, Smith and Wagener, McGraw-Hill, 1992, ISBN 0-07-000406-4.
- Fortran 90 Language Guide - Gehrke, Springer, London, 1995, ISBN 3-540-19926-8.
- Fortran 90 Programming - Ellis, Philips, Lahey, Addison Wesley, Wokingham, 1994, ISBN 0-201-54446-6.
- Fortran Top 90-Ninety Key Features of Fortran 90 - Adams, Brainerd, Martin and Smith, Unicomp, 1994, ISBN 0-9640135-0-9.
- Introducing Fortran 90 - Chivers and Sleightholme, Springer-Verlag London, 1995, ISBN 3-540-19940-3.
- Introduction to Fortran 90, Algorithms, and Structured Programming, Vol. 1, R. Vowels: 93 Park Drive, Parkville 3052, Victoria, AUSTRALIA, (rav@goanna.cs.rmit.edu.au). $20 Aust, $15 US, ISBN 0-9596384-8-2.

- Introduction to Fortran 90 for Scientific Computing - Ortega, Saunders College Publishing, 1994, ISBN 0-030010198-0.

- Migrating to Fortran 90 - James F. Kerrigan, O'Reilly Associates, 1993, ISBN 1-56592-049-X.

- Programmer's Guide to Fortran 90, third edition - Brainerd, Goldberg and Adams, Springer, 1996, ISBN 0-387-94570-9.

- Programming in Fortran 90 - Morgan and Schonfelder, Alfred Waller/ McGraw-Hill, Oxfordshire, 1993, ISBN 1-872474-06-3.

- Programming in Fortran 90 - I.M. Smith, Wiley, ISBN 0471-94185-9.

- Schaum's Outline of Theory and Praxis -- Programming in Fortran 90 - Mayo and Cwiakala, Mc Graw Hill, 1996. ISBN 0-07-041156-5.

- Upgrading to Fortran 90 - Redwine, Springer-Verlag, New York, 1995, ISBN 0-387-97995-6.

LIBROS EN FRANCES:

- Fortran 90; Approche par la Pratique - Lignelet, Se'rie Informatique E'ditions, Menton, 1993, ISBN 2-090615-01-4.

- Fortran 90. Les concepts fondamentaux, the translation of "Fortran 90 Explained" M. Metcalf, J. Reid, translated by M. Caillet and B. Pichon, AFNOR, Paris, ISBN 2-12-486513-7.

- Fortran 90; Initiation a` partir du Fortran 77 - Aberti, Se'rie Informatique E'ditions, Menton, 1992, ISBN 2-090615-00-6.

- Manuel complet du langage Fortran 90, et guide d'application, LIGNELET, P., S.I. editions, Jan. 1995. ISBN 2-909615-02-2

- Programmer en Fortran 90, DELANNOY, C., Eyrolles, 1992. ISBN 2-212-08723-3

TUTORIALS

Copyright but freely available course material is available on the World Wide Web from the URLs:

- Manchester Computer Centre:
  http://www.hpctec.mcc.ac.uk/hpctec/courses/Fortran90/F90course.html
  or via ftp: ftp.mcc.ac.uk, in the directory /pub/mantec/Fortran90.

- The University of Liverpool: http://www.liv.ac.uk/HPC/HPCpage.html.

- CERN: http://wwwcn.cern.ch/asdoc/f90.html or via anonymous ftp from asisftp.cern.ch in the directory cnl as the file f90tutor.ps. An ASCII copy of this material as a set of slides for a six-hour course is available from metcalf@cern.ch.

- In French: Support de cours Fortran 90 IDRIS - Corde & Delouis (from ftp.ifremer.fr, file pub/ifremer/fortran90/f90_cours_4.ps.gz).

- A course on HPF is freely available from Edinburgh: http://www.epcc.ed.ac.uk/epcc-tec/course-packages/HPF-Package-form.html

ARTICULOS

- Appleby, D., FORTRAN First in a six-part series on languages that have stood the test of time -- BYTE, Sep. 1991, 147-150

- Bernheim, M., Fortran Mode d'emploi - Fortran 90 -- Intereditions (1991) 163-176

- Brankin, R.W., Gladwell, I., A Fortran 90 Version of RKSUITE: An ODE Initial Value Solver, Annals of Numerical Mathematics, Vol 1, 1994, in press.

- Buckley, Albert G., Conversion to Fortran 90: A Case Study -- accepted (Sep. 93) for ACM TOMS (ftp.royalroads.ca:pub/software/bbuckley/alg999)

- Buckley, Albert G., Algorithm 999: A Fortran 90 code for unconstrained non linear minimisation -- accepted (Sep. 93) for ACM TOMS

- Du Croz, Jeremy J., Building Libraries with Fortran 90 Fortran Journal 4/5, Sep./Oct 1992

- Glassy, L., Tiny-Ninety: A subset of F90 for beginning programmers --Fortran Journal 4/3, May/Jun. 1992, 2-6

- Hanson, R.J., A design of high-performance Fortran 90 Libraries -- IMSL technical report series No. 9201 (1992)

- Hanson, R.J., Matrix multiplication in Fortran 90 using Strassen's algorithm -- Fortran Journal 4/3, May/Jun. 1992, 6-7

- Iles, Robert, Fortran 90: The First Two Years -- Unicom Seminar on Fortran and C in Scientific Computing, 1993.

- Lignelet, P., Fortran -- Les Techniques de l'ingenieur, H2120, Dec 1993.

- Maine, R., Review of NAG Fortran 90 translator -- Fortran Journal 3/6, Nov/dec 1991.

- Metcalf, M., A derived data type for data analysis -- Computers in Physics, Nov/Dec 1991, 599-604.

- Metcalf, M., An encounter with F90 -- Particle World 3/3 (1993), 130-134.

- Metcalf, M., Fortran 90 Tutorial -- CERN Computer Newsletter, Nos. 206/207/208/209/210/211 (1992-1993).

- Metcalf, M., Using the f90 compiler as a software tool -- CERN Computer Newsletter, No. 209 (1992).
- Metcalf, M., Still programming after these years -- New Scientist, (12 Sep. 1992), 30-33
- Olagnon, M., Experience with NagWare f90 -- Fortran Journal 4/6, Nov/dec 1992, 2-5.
- de Polignac, Christian, Du Fortran VAX au Fortran 90 -- Decus, Paris, 7 Avril 1993.
- Prentice, John K., Fortran 90 benchmark results -- Fortran Journal 5/3, May/June 1993.
- Reid, John, The Fortran 90 Standard -- Programming environments for high level scientific problem solving, Gaffney ed., IEEE Trans., North-Holland (1992), 343-348.
- Reid, John, Fortran 90, the language for scientific computing in the 1990s --Unicom Seminar on Fortran and C in Scientific Computing, 1992
- Reid, John, The advantages of Fortran 90 -- Computing 48, 219-238.
- Robin, F., Fortran 90 et High Performance Fortran, Bulletin technique CEA, Oct. 1992, 3-7.
- Schonfelder, J.L., Semantic extension possibilities in the proposed new Fortran -- Software practice and experience, Vol.19, (1989), 529-551.
- Schonfelder, J.L., Morgan, J.S., Dynamic strings in Fortran 90 --Software practice and experience, Vol.20(12), (1990), 1259-1271.
- Sipelstein, J.M., Blelloch, G.E., Collection-oriented languages -- Proceedings of the IEEE, Vol. 79, No. 4, (1991), 504-530.
- Vignes, Jean, Vers un calcul scientifique fiable : l'arithmetique stochastique -- La Vie des Sciences, Comptes rendus, serie generale, tome 10, 1993, No 2, 81-101.
- Vignes, Jean, A stochastic arithmetic for reliable scientific computation, MATCOM 940 - Mathematics and Computers in Simulation 35 (1993) 233-261.
- Walker, D.W., A Fortran 90 code for magnetohydrodynamics. Part I: banded convolution -- Oak Ridge National Lab. report TM-12032 (1992).
- Walter, W., Fortran 90: Was bringt der neue Fortran-Standard fuer das numerische Programmieren ? -- Jahrbuch Ueberblicke Mathematik Vieweg, (1991) 151-174.
- Wampler, K. Dean, The Object-Oriented programming Paradigm and Fortran programs -- Computers in Physics, Jul/Aug 1990, 385-394.

f90 compiler

SYNOPSIS

f90 [flag] ... file ...

DESCRIPTION

The **f90** command invokes the DEC OSF/1 Fortran 90 compiler and produces extended **coff** object files and Fortran 90 module information files. The **f90** command may invoke the C compiler (**cc**(1)) or the C preprocessor (**ccp**(1)). The **f90** command accepts several types of arguments:

- files with the suffix .f90 are assumed to be free form source files.Files with the suffix **.f**, **.for**, or **.FOR**, are assumed to be fixed form source forms.
- When the **-c** flag is specified without the **-o** flag, an object file is created in the current directory that has the basename of the source file suffixed with **.o**.
- For each MODULE declaration in a source file, a module information file named module**.mod** is created.

FLAGS

Some flags have the form **–flag** keyword. The flag must be spelled out completely, but the keyword can be abbreviated to its shortest unique prefix (4 characters are recommended). For example, **–check underflow** can be specified as **–check unde**.

In instances where a keyword or flag can be prefixed with "no", only the form which alters the default value is documented.

The following flags are interpreted by **f90**. See **ld**(1) for linker and load-time flags.

–assume nozsize

Omit run-time checking for zero-sized array sections for increased performance with the **–wsf** option. In programs compiled with both the **-wsf** and **–assume nozsize** flags, references to zero-sized array sections will cause the executable to fail or produce incorrect results.

–automatic or **-recursive**
–static (default)

Local variables are allocated in static storage unless the **-automatic** or **-recursive** flag is used to specify the run-time stack.

—c Suppress linking and force an object file to be produced.

—C or **-check_bounds** or **-check bounds**
Generate code to issue a run-time error if array subscript or character substring expressions violate the declared bounds.

—check format
Issue a run-time FORVARMIS error message when the data type for an item being formatted for output does not match the FORMAT descriptor.

—check output_conversion
Issue a run-time OUTCONERR continuable error message when a data item is too large to fit in a designated FORMAT descriptor field.

—check overflow
Trap on integer overflow.

—check underflow
Produce a run-time warning that a floating-point underflow occurred.

—convert big_endian
—convert ...
Read and write unformatted files according to the specified format.

—double_size 64 (default)
—double_size 128
Use as the default size for DOUBLE PRECISION declarations and constants.

—d_lines
Compile lines having a D in column 1 of a fixed form source file.

—error_limit num (default is 30)
—noerror_limit
Limit the number of compiler messages to num.

—extend_source
Treat the statement field of each fixed form source line as ending in column 132 instead of 72.

—fast
Sets flags which can improve run-time performance: —assume noaccuracy_sensitive, -assume nozsize, -align dcommons, and —math_library fast.

–fixed
–free
Interpret Fortran source files in the specified source form. By
default, source files suffixed with **.f90** or **.i90** are assumed to be free format and files suffixed with **.f**, **.for**, **.FOR**, or **.i** are assumed to be fixed format.

–fpe0 or **-fpe** (default)
Terminates execution if the operation results in overflow or division by zero or if the operands are denormalized numbers or other exceptional values. Calculated denormalized numbers are set to zero. This is the only fpe setting which produces a core file when a floatingpoint exception occurs.

–fpe1 , **–fpe2** , **–fpe3** , **–fpe4**

–g0 Do not produce symbolic debugging or traceback information.

–g1 (default)
Produce traceback information (showing pc to line correlation).

–g or **-g2**
Produce traceback and symbolic debugging information. Unless an explicit optimization level was specified, the **–O0** flag is set.

–g3 Produce symbolic debugging and traceback information without modifying the optimization level. Debugger inaccuracies may result.

–i2 or **-integer_size 16**
–i4 or **-integer_size 32** (default)
–i8 or **-integer_size 64**
Use as the default size for integer and logical variables.

–math_library accurate (default)
–math_library fast
Select a math library based on accuracy or performance. The fast math library performs faster computations for several intrinsic procedures but with slightly less fractional accuracy and less robust exception handling than the default library.

–om Perform post-link optimizations for programs compiled **-non_shared**. The following options can be passed directly to **om** by using the **-WL** compiler option:
–O0 Disable all optimizations.

–O1 Enable local optimizations and recognize common subexpressions.

–O2 Enable global optimizations and all **-O** optimizations. Optimizations include code motion, strength reduction and test replacement, split lifetime analysis, and code scheduling.

–O3 Enable global optimizations that improve speed at the cost of increased code size and all **–O2** optimizations. Optimizations include integer multiplication and division expansion using shifts, loop unrolling, and code replication to eliminate branches.

–O4 or **-O** (default)
Enable inline expansion of small procedures and all **–O** optimizations.

–O5 Enable software pipelining for innermost loops. To determine whether using **–O5** benefits your particular program, time program execution for the same program compiled at levels **–O** and **-O5**. For programs that contain loops that exhaust available registers, longer execution times may occur. In this case, specify options **–unroll 1** or **-unroll 2** with the **–O5** option.

–o output
Name the final output file output.

–P Run only **cpp**(1) and put the result for each source file in a corresponding .i or .i90 file. The .i or .i90 file does not have line numbers (#) in it. This flag sets the **–cpp** flag.

–real_size 32 (default)
–r8 or **-real_size 64**
–r16 or **-real_size 128**
Use as the default size for REAL declarations, constants, functions, and intrinsics.

–shared
Produce a shared object. This includes creating all of the tables for run-time linking and resolving references to other specified shared objects. The object created may be used by the linker to make a dynamic executable.

–show include
Include in the listing files specified with in an INCLUDE statement.

–show map
Include in the listing file, symbol maps of all symbols used in the source program.

–syntax_only
Check for correctness without generating an object file.

–u or **-warn declarations**
Set the default type of a variable undefined (IMPLICIT NONE), causing the compiler to issue a warning for any undeclared symbols.

–V Create a source listing whose filename consists of the basename of the source file suffixed with **.l**

–version
Print the compiler version number.

–w or **-nowarn** or **-warn nogeneral**
Suppress issuing of warnings.

–warn argument_checking
Issue warning messages for mismatched procedure arguments.

–warn nouninitialized
Do not issue warning messages for a variable that is used before a value is assigned to it.

EXAMPLES

f90 ax.f90
Compiles ax.f90 producing executable file a.out. Optimizations occur by default.

f90 -o abc ax.f90 bx.f90 cx.f90
Uses flag **–o** to name the executable file abc and compiles ax.f90, bx.f90, and cx.f90 as one program. Interprocedural optimization occurs across ax, bx, and cx.

f90 -c ax.f90 bx.f90 cx.f90
Uses flag **–c** to suppress linking and produce individual object files ax.o, bx.o, and cx.o. Interprocedural optimizations are prevented.

f90 -c -o abc.o ax.f90 bx.f90 cx.f90

Uses flag **–c** to suppress linking, and flag **-o** to

produce a single object file abc.o.

Interprocedural optimization occurs.

Compilación de un fichero que sólo tiene módulos

```
f90 -c mod_ppp.f90
```

crea `mod_ppp.o` y un fichero por módulo (`mod_opera.mod`, …)

Despues se linka con el principal

```
f90   mod_ppp.o  pro_ppp.f90
```

Los `*.mod` no hacen falta durante la ejecución. Hacen falta en la compilación del principal o donde aparezca `USE`.

```
f90  -c  pro_ppp.f90
```

lee `mod_opera.mod`