ÍNDICE

CAPÍTULO 1. FORTRAN 90 3
Francisco de Sande

1. INTRODUCCIÓN 7
2. CÓDIGO FUENTE, TIPOS Y ESTRUCTURAS DE CONTROL 7
3. ENTRADA/SALIDA (E/S) 13
4. PROCEDIMIENTOS Y MÓDULOS 15
5. PROCESAMIENTO DE VECTORES 26
6. ADMINISTRACIÓN DE MEMORIA DINÁMICA 34

CAPÍTULO 1
FORTRAN 90

Francisco de Sande
DEIOC. Universidad de La Laguna

ÍNDICE

1. INTRODUCCIÓN 7
2. CÓDIGO FUENTE, TIPOS Y ESTRUCTURAS DE CONTROL 7
   - 2.1. Especificaciones 8
   - 2.2. `implicit none` 9
   - 2.3. Valores de clase 9
   - 2.4. Tipos derivados 10
   - 2.5. Estructuras de control de flujo 11
     - 2.5.1. Sentencia condicional: `if` 11
     - 2.5.2. Bucles `do` 11
     - 2.5.3. La sentencia `case` 12
3. ENTRADA/SALIDA (E/S) 13
4. PROCEDIMIENTOS Y MÓDULOS 15
   - 4.1. Unidades de programa 15
   - 4.2. Procedimientos 16
     - 4.2.1. Procedimientos externos 16
   - 4.3. Procedimientos internos 16
   - 4.4. Bloques `interface` 17
     - 4.4.1. Argumentos con nombre 18
     - 4.4.2. Argumentos opcionales 18
     - 4.4.3. Tipos derivados como argumentos 19
   - 4.5. Argumentos procedurales 19
   - 4.6. Cláusula `result` para funciones 19
   - 4.7. Funciones que devuelven vectores 20
   - 4.8. Procedimientos recursivos 20
   - 4.9. Procedimientos genéricos 20
   - 4.10. Módulos 21
     - 4.10.1. Procedimientos genéricos 22
   - 4.11. Sobrecarga de operadores 23
   - 4.12. Definición de operadores 24
   - 4.13. Sobrecarga del operador de asignación 24
   - 4.14. Estructura de los programas 25
     - 4.14.1. Cuándo utilizar bloques `interface` 25
5. PROCESAMIENTO DE VECTORES 26
   - 5.1. Terminología 26
   - 5.2. Declaración de vectores 26
   - 5.3. Operaciones sobre vectores 27
     - 5.3.1. Procedimientos intrínsecos elementales 28
   - 5.4. Sentencia `where` 28
   - 5.5. Secciones de vectores 28
   - 5.6. Vectores de tamaño cero 30
   - 5.7. Constructores de vectores 30
   - 5.8. Vectores dinámicos 31
   - 5.9. Vectores automáticos 31
   - 5.10. Vectores con forma asumida 33
6. ADMINISTRACIÓN DE MEMORIA DINÁMICA 34
   - 6.1. Significado de un puntero 34
   - 6.2. Declaración de punteros 34

6.4. Estado de una asociación de punteros 35
6.5. Memoria dinámica 36
6.6. Argumentos puntero 37
6.7. Funciones que devuelven punteros 37
6.8. Vectores de punteros 38
6.9. Listas enlazadas 38

1. INTRODUCCIÓN

En este capítulo introduciremos al lector a la programación en Fortran90 utilizando para ello ejemplos ilustrativos de las características principales del lenguaje.

Se sale por completo de la finalidad de este texto un estudio exhaustivo de Fortran90. Nuestro objetivo es más bien facilitar la transición a este lenguaje a programadores que ya tienen cierta experiencia con algún lenguaje de programación imperativo (C, Pascal, Fortran77, etc). En ese sentido no entraremos en detalles a la hora de explicar el significado de algunos conceptos que deberían ser claros para este tipo de lectores.

En la bibliografía proponemos al lector referencias suficientes que le permitirán ampliar sus conocimientos sobre el lenguaje, partiendo de la introducción que aquí presentamos. El material del curso (ejercicios y transparencias), las páginas de manual (`man`) o los manuales de referencia del compilador de Fortran90 con que se trabaje son otras fuentes complementarias de información para consolidar y comprender mejor las especificidades del lenguaje.

Después de una fase inicial de toma de contacto, entendemos que el mejor camino para comenzar a trabajar con un lenguaje de programación nuevo consiste en plantearse un proyecto (más o menos ambicioso, dependiendo de las expectativas del lector) y proceder a su desarrollo utilizando el nuevo lenguaje, y acudiendo a las fuentes de información disponibles a la hora de salvar los escollos que se plantean. En este sentido, animamos al lector a comprobar de forma práctica (con un compilador de Fortran90) todos los ejemplos que se plantean en este capítulo.

2. CÓDIGO FUENTE, TIPOS Y ESTRUCTURAS DE CONTROL

Comenzaremos nuestro recorrido por los diferentes elementos del lenguaje, estableciendo unos convenios mínimos respecto a la forma en la que recomendamos codificar los programas escritos en Fortran90 (que por otra parte se corresponde con la que aquí utilizaremos).

1. Utilizaremos minúsculas para la mayor parte del código, puesto que ello redunda en una mayor legibilidad del mismo.

   Reservaremos las mayúsculas para constantes o cualquier otra entidad que deseemos resaltar explícitamente en el código.

2. Utilizaremos un indentado de 2 espacios (o una tabulación equivalente) para el cuerpo de programas, subrutinas, bloques `interface`, bucles `do`, bloques `if`, `case`, etc.

3. Incluiremos siempre el nombre del programa, subrutina o función en su sentencia `end`.

4. En las sentencias `use` utilizaremos siempre la cláusula `only` para documentar explícitamente todos los elementos que son accedidos desde el módulo.

5. En las sentencias `call` y referencias a funciones utilizaremos nombres para los parámetros opcionales.

En Fortran90 las líneas de código pueden tener hasta un máximo de 132 caracteres y el símbolo ampersand (`&`) se usa como signo de continuación: si una línea acaba con `&`, ello significa que continúa en la siguiente.

El compilador no distingue entre letras mayúsculas y minúsculas y los identificadores pueden tener un máximo de 31 caracteres incluyendo el caracter de subrayado (que a menudo se utiliza para dar más legibilidad a los identificadores). El primer caracter de un identificador ha de ser una letra, y como siempre, resulta fundamental para una buena documentación implícita de nuestros programas utilizar identificadores significativos. En Fortran90 el símbolo punto y coma (`;`) se usa para separar diferentes sentencias en la misma línea.

Para colocar un comentario se utiliza el símbolo de exclamación (`!`). Desde ese símbolo hasta el final de la línea, el compilador ignora el texto (lo entiende como un comentario). Cuando el comentario se utiliza para aclarar el contenido de una línea de código, recomendamos que el texto se separe (lo más a la derecha posible) del texto del código, y que todos los comentarios queden alineados a la misma altura. Los operadores relacionales

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

Tabla 1.1: Operadores relacionales

```
program programa_en_formato_libre
! nombres largos con subrayado
! no hay columnas especiales
implicit none

real :: tx, ty, tz        ! comentario final

! varias sentencias por linea
tx = 1.0; ty = 2.0; tz = tx * ty;

print *, &                ! las lineas se pueden partir
   tx, ty, tz
end program programa_en_formato_libre
```

Figura 1.1: Formato del código fuente

```
program sumatorio
implicit none
integer, parameter :: MAX = 100    ! úNmero ámximo de elementos
integer :: num_elem, &             ! úNmero de elementos
           i                       ! Contador
real, dimension(MAX) :: v          ! Vector
logical :: default

writeÍ(*, *) ntroduzca el únmero de elementos: '
read(*, *) num_elem
writeÍ(*, *) ntroduzca uno a uno los elementos: '
read *, (v(i), i=1, num_elem)
print *, 'Suma = ', sum(v, num_elem)
print *, 'Todo listo.'
end program sumatorio

function sum(v, n)
real v(n)
sum = 0.0
do i = 1, n
  sum = sum + v(i)
end do
return
end function
```

Figura 1.2: Un programa que calcula un sumatorio

Los programas que aparecen en las figuras 1.2 y ?? muestran algunas de las características del código fuente en Fortran90. El programa de la figura ?? calcula la suma de una serie de números introducidos por teclado.

Respecto al formato del código en Fortran90 diremos para finalizar que los compiladores de Fortran90 aceptan cualquier código correctamente escrito en Fortran77. Es por eso que podemos encontrar códigos en Fortran90 que mantienen algunas características que son consideradas obsoletas desde el punto de vista de Fortran90.

En este mismo sentido el lenguaje conserva algunos elementos (la sentencia `goto` por ejemplo) simplemente por compatibilidad con Fortran77, aunque se desaconseja el uso de estas características.

2.1. Especificaciones

En la terminología de Fortran90 se denomina especificaciones a las declaraciones. Una especificación es una forma (extendida) de declaración donde se colocan juntas todas las características de una entidad. Una especificación tiene la forma:

`<tipo> [[, <atributo>] ... ::] <lista de elementos>`

El `tipo` puede ser cualquiera de los tipos predefinidos del lenguaje: `integer`, `real`, `complex`, `logical` o

```
integer[([kind=]<valor de clase>)]
character[(<lista de áparmetros actuales>)]
type(<identificador del tipo>)
```

El atributo puede ser cualquiera de los siguientes: `parameter`, `public`, `private`, `pointer`, `target`, `allocatable`, `dimension(`<*extensión*>`)`, `intent (`<*inout*>`)`, `optional`, `save`, `external`, `intrinsic`.

```
real :: a = 2.61828, b = 3.14159
! Dos variables reales declaradas e inicializadas

integer, parameter :: n = 100, m = 1000
! óDeclaracin e óinicializacin de dos constantes enteras

character (len = 8) :: ch
! óDeclaracin de una cadena de caracteres de longitud 8

integer, dimension(-3:5, 7) :: ia
! óDeclaracin de un vector de enteros con l\'{\i}mite
! inferior negativo

integer, dimension (-3:5, 7) :: ia, ib, ic(5, 5)
! Vectores de 9x7 componentes.
! Los \'{\i}ndices de ic var\'{\i}an en [1, 5] [1, 5]
```

Figura 1.3: Ejemplos de declaraciones

El significado de cada uno de estos elementos lo iremos introduciendo a lo largo del capítulo.

Es posible inicializar las variables a la hora de declararlas (especificarlas). La figura 1.3 muestra algunos ejemplos.

2.2. `implicit none`

En Fortran77 los tipos implícitos permiten la utilización de variables que no han sido declaradas previamente (por ejemplo, cualquier variable que no haya sido declarada y cuyo identificador comience por "i", el compilador asumirá que es de tipo entero).

Los tipos implícitos han sido la causa de muchos errores de programación. La sentencia `implicit none` de Fortran90 obliga a declarar todas las variables y puede ir precedida en una unidad de programa solamente por sentencias `use` y `format`.

Recomendamos encarecidamente que todos los programas que se desarrollen en Fortran90 utilicen `implicit none`. De hecho, todos los ejemplos de este capítulo utilizan esta sentencia.

2.3. Valores de clase

En Fortran90 hay cinco tipos básicos: `real`, `integer`, `complex`, `character` y `logical`. Cada uno de estos tipos puede tener asociado un valor entero no negativo llamado la clase (kind) del tipo.

Se trata de una característica útil para escribir código portable que requiera una determinada precisión numérica. Los valores de clase dependen de la máquina en cuestión, pero todo procesador ha de soportar al menos dos clases para `real` y `complex` y una para `integer`, `logical` y `character`. Estudiaremos alguna de las diversas funciones intrínsecas que podemos utilizar para obtener y establecer los valores de clase. En la terminología de Fortran90 se denominan funciones intrínsecas a aquellas funciones que forman parte del lenguaje (en contraposición a las funciones que implementa el programador).

El siguiente ejemplo declara una variable, ra, cuya precisión está determinada por el valor de clase wp. El valor de wp se supone que ha sido evaluado previamente.

```
real (kind = wp) :: ra        ! o bien:
real(wp) :: ra
```

Una variable real de 8 bytes (64 bits) habitualmente tiene un valor de clase 8 ó 2 mientras que una variable real de 4 bytes (32 bits) habitualmente tiene un valor de clase de 4 ó 1.

```
real(kind=2) :: x       ! x declarada de clase de tipo 2
real :: y               ! y declarada de clase de tipo por defecto
integer :: i, j
i = kind(x)             ! i = 2
j = kind(y)             ! j tiene el valor por defecto de los reales
                        ! el valor de j depende del sistema
```

Figura 1.4: Ejemplos de declaraciones

La función intrínseca kind(), que requiere un argumento de cualquier tipo básico, devuelve la clase de tipo de su argumento. La figura 1.4 muestra un ejemplo de utilización de la función kind().

La función intrínseca selected_real_kind() tiene dos argumentos enteros opcionales, p y r El valor p especifica el número de dígitos decimales y r especifica el mínimo rango de exponente que se desea. La función selected_real_kind (p, r) devuelve el valor de clase que se ajusta o que excede los requisitos especificados por p y r. Si más de un valor de clase satisface los requisitos, el valor que se devuelve es el que tenga la mínima precisión decimal. Si la precisión no está disponible se retorna -1. Si el rango no está disponible, -2.

La utilización de `kind` con esta función dota al código de una gran portabilidad. Veamos un ejemplo de utilización de `kind`:

```
integer, parameter :: idp = kind(1.0D)
real(kind = idp) :: ra
integer, parameter :: idp = kind(1.0D)
complex (kind = idp) :: raiz1, raiz2
```

La variable ra se declara de doble precisión. La portabilidad está garantizada por el hecho de que el valor de clase depende de la máquina en la que el código se ejecute.

En Fortran90 los enteros habitualmente tienen 16, 32 ó 64 bits. Para declarar un entero de forma que dependa del sistema, se debe indicar el valor de clase asociado con el rango de enteros que se desee.

Consideremos el siguiente código:

```
integer, parameter :: i8 = selected_int_kind(8)
integer(kind = i8) :: ia, ib, ic
```

Las variables ia, ib e ic tomarán valores en el rango de al menos $[-10^8, 10^8]$ si lo permite el procesador.

El siguiente código:

```
integer, parameter :: i8 = selected_int_kind(8)
integer(kind = i8) :: ia
print *, huge(ia), kind(ia)
```

imprime el mayor entero disponible para este tipo de enteros, y su valor de clase, mientras que este otro:

```
integer, parameter :: i10 = selected_real_kind(10, 200)
real(kind = i10) :: a
print *, range(a), precision(a), kind(a)
```

imprime el rango del exponente, los dígitos de precisión y el valor de clase de la variable a.

2.4. Tipos derivados

En Fortran90 se denomina tipo derivado a cualquier tipo definido por el usuario (en contraposición a los tipos básicos, que están predefinidos en el sistema). Los tipos derivados de Fortran90 juegan un papel similar al de las `struct` de C o los `record` de Pascal: permiten definir tipos estructurados.

Los tipos derivados pueden construirse a partir de diferentes tipos intrínsecos (básicos) así como otros tipos derivados que hayan sido definidos previamente. Las componentes de un tipo derivado se acceden usando el símbolo `%`. El único operador predefinido para tipos derivados es el de asignación (=), que permite asignarle un valor a una variable de ese tipo. Cuando estudiemos la sobrecarga de operadores veremos que es posible redefinir los operadores que actúan sobre un tipo derivado. Este primer ejemplo muestra cómo definir la forma de un tipo derivado al que hemos denominado "matriculas":

```
type matriculas
  character (len = 2) :: provincia
  integer :: numero
  character (len = 2) :: letras
end type matriculas
```

y la declaración de variables de este tipo:

```
type(matriculas) :: coche1, coche2
matriculas :: coche1              ! Incorrecto
```

Veamos cómo asignar un valor constante a coche1:

```
coche1 = matricula('TFÁ', 2227, G')
```

O cómo acceder directamente a una componente de coche2:

```
coche2%letra = 'BZ'
```

```
type (matriculas), dimension(n) :: miscoches
```

y ahora definimos un tipo derivado conteniendo otro:

```
type propietario
  character (len = 1) :: nombre
  character (len = 50) :: direccion
  type(matriculas) :: coche
end type propietario
```

y declaramos una variable de tipo propietario:

```
type(propietario) :: persona
```

Veamos cómo utilizar `%` para referenciar una componente del tipo derivado:

```
propietaro%coche%numero = 2227
```

y por último, definamos una constante de un tipo derivado:

```
type punto :: origen = punto(0.0, 0.0, 0.0)
```

2.5. Estructuras de control de flujo

Fortran90 posee tres sentencias que permiten costruir bloques: `if`, `do` y `case`. Todas ellas pueden ser anidadas y puede dárseles nombres para incrementar la legibilidad de los programas.

2.5.1. Sentencia condicional: `if`

La figura 1.5 muestra un ejemplo de utilización de una sentencia `if`. La forma general de esta sentencia es:

```
[<nombre>:] if (<expr. logica>) then
  <bloque>
else if (<expr. ólgica>) then [<nombre>]
  [<bloque>]...
  [else [<nombre>]
    <bloque>]
end if [<nombre>]
```

```
select: if (i < 0) then
  call negativo
else if (i == 0) then select
  call cero
  else select
    call positivo
end if select
```

Figura 1.5: Ejemplo de una sentencia `if`

2.5.2. Bucles `do`

La forma general de un bucle `do` es la siguiente:

```
[<nombre>:] do [<clausula de control>]
  <bloque>
end do [<nombre>]
```

La cláusula de control que aparece en la sentencia puede ser:

1. Una cláusula de control de iteración

   contador = inicial, final [, incremento]

2. Una cláusula de control `while`

   `while` (<expr. lógica>)

3. Ninguna

```
! Clausula de control de óiteracin:

filas: do i = 1, n
cols:   do j = 1, m
          a(1, j) = i + j
        end do cols
      end do filas

! Clausula de control while:

menor: do while (i <= 100)
  ...
  <cuerpo del bucle>
  ...
end do menor
```

Figura 1.6: Ejemplos de bucles `do`

La figura 1.6 presenta dos ejemplos de bucles `do`. En Fortran90 las sentencias `exit` y `cycle` tienen la misma semántica que en C las sentencias `break` y `continue` respectivamente: `exit` abandona la ejecución de un bucle mientras que `cycle` abandona la iteración actual, y transfiere el control a la posición del `end do`.

Tanto `exit` como `cycle` por defecto se refieren al bucle más interno pero pueden referirse a un bucle con nombre.

```
ext: do i = 1, n
med:   do j = 1, m
int:     do k = 1, l
           ...
           if (a(i, j, k) < 0) exit ext  ! salta fuera
           if (j == 5)
             cycle med       ! omite j==5 y hace j=6
           if (i == 5)
             cycle           ! salta el resto del bucle int y
                             ! pasa a su siguiente óiteracin
           ...
         end do int
       end do med
     end do ext
```

Figura 1.7: Ejemplo de utilización de `exit` y `cycle`

La figura 1.7 muestra un ejemplo de utilización de las sentencias `exit` y `cycle`.

```
do
  read(*, *) x
  if (x < 0) exit
  y = sqrt(x)
  ...
end do
```

Figura 1.8: Ejemplo de bucle `do` sin cláusula de control

La figura 1.8 presenta un bucle `do` sin cláusula de control. Nótese que esta forma puede tener el mismo efecto que un bucle `do-while`.

2.5.3. La sentencia `case`

La sentencia `case` suministra una forma estructurada de seleccionar diferentes opciones, dependiendo del valor de una única expresión. Normalmente se utiliza para sustituir a una secuencia de sentencias `if ... then ... else` o también a los `goto` calculados de Fortran77.

```
[<nombre>:] select case (<óexpresin>)
  [case (selector)[<nombre>]
    block]
                ...
end select [<nombre>]
```

Figura 1.9: Forma general de la sentencia `case`

La forma general de una sentencia `case` se muestra en la figura 1.9 donde <expresión> ha de ser de tipo `character`, `logical` o `integer` y <selector> ha de ser la palabra `default` o bien uno o más valores del mismo

1. Un único valor
2. Un rango de valores separados por : (solamente `integer` o `character`)
3. Una lista de valores o rangos. El valor mayor o menor del rango puede omitirse.

```
color: select case (ch)
  case ('c', 'd', 'g':'m')
    color = 'rojo'
  case ('x':)
    color = 'verde'
  case default
    colorá = zul'
end select color
```

Figura 1.10: Un ejemplo de sentencia `case`

La figura 1.10 ejemplifica la utilización de la sentencia `case` ilustrando estas posibilidades. Por compatibilidad con Fortran77, la sentencia `goto` sigue estando disponible en Fortran90, pero es preferible usar combinaciones adecuadas de las sentencias `if`, `do`, `case`, `exit` y `cycle` en su lugar.

3. ENTRADA/SALIDA (E/S)

Las sentencias de entrada y salida (E/S) en Fortran90 son respectivamente `read` y `print`. Cada una de las sentencias puede adoptar tres formas diferentes:

1. read <ch_var>, <lista de entrada>
2. read <etiqueta> , <lista de entrada>
3. read ∗, <lista de entrada>

para `read` y

1. print <ch_var> , <lista de entrada>
2. print <etiqueta> , <lista de salida>
3. print ∗, <lista de salida>

para `print`.

En todos los casos <ch_var> es una constante de caracteres, una variable de tipo `character`, o un vector de caracteres. <etiqueta> es la etiqueta de una sentencia del código y las listas corresponden a las variables que se desea leer o escribir. La lista de variables de una sentencia `read` sólo puede contener identificadores de variables, mientras que en un `print` también pueden contener constantes o expresiones.

En las tres formas de las sentencias de E/S el primer elemento que sigue a la palabra `read` o `print` es lo que se conoce como el especificador de formato, que suministra la información necesaria para la edición que ha de realizarse como parte del proceso de adquisición (emisión) de datos.

Esta información se denomina un formato y consiste en una lista de descriptores de edición encerrados entre paréntesis:

```
(desc_ed1, desc_ed2, ...)
```

La primera de las formas de las sentencias de E/S se denomina de formato empotrado porque el formato en sí mismo forma parte de la propia sentencia de E/S:

```
print '(<lista de descriptores de edicion>)', <lista de salida>
```

o bien

```
print "(<lista de descriptores de edicion}>)", <lista de salida>
```

(el compilador acepta cualquiera de las dos formas, aunque recomendamos la primera).

En la segunda variante de las sentencias `print` y `read` la etiqueta corresponderá con una nueva sentencia: `format` que contiene el formato apropiado para la operación de E/S.

La tercera forma de las sentencias de E/S es la que se conoce como entrada/salida dirigida por lista. Se trata de la forma más simple y es la que más utilizaremos en los ejemplos de este capítulo. En esta forma el asterisco indica que se desea un formato de salida (entrada) que se realiza teniendo en cuenta el tipo de los valores a escribir (leer).

Los descriptores de edición se dividen en dos categorías:

`Iw`	Escribe un entero en los siguientes w caracteres
`Fw.d`	Imprime un número real en los siguientes w caracteres utilizando d posiciones decimales
`Ew.d`	Imprime un número real en los siguientes w caracteres utilizando un formato de exponente con d posiciones decimales y 4 dígitos para el exponente
`Aw`	Imprime una cadena de caracteres en los siguientes w caracteres
A	Imprime una cadena de caracteres comenzando en la siguiente posición sin imprimir espacios en blanco sobrantes ni de relleno
`Lw`	Imprime L-1 espacios en blanco seguidos por `T` o `F` para indicar un valor lógico
`nX`	Ignora las siguientes n posiciones
`Tc`	Imprime el siguiente elemento comenzando en el caracter c
`TLn`	Imprime el siguiente elemento comenzando n posiciones antes (`TL`)
`TRn`	o después (`TR`) de la posición actual
"$c_1c_2\ldots c_n$"	Imprime la cadena de caracteres $c_1c_2\ldots c_n$ comenzando en la siguiente posición
'$c_1c_2c_n$'	Imprime la cadena de caracteres $c_1c_2c_n$ comenzando en la siguiente posición

Tabla 1.2: Descriptores de edición para salida

- Los que afectan al orden en que los caracteres se editan a la entrada o salida

Estudiaremos solamente los descriptores de edición de salida (los de entrada son completamente análogos) y remitimos al lector interesado a la bibliografía o bien al manual de usuario de su compilador de Fortran90.

En la tabla 1.2 aparecen los descriptores de edición de salida así como el significado de cada uno de ellos. Los descriptores para la sentencia `read` son completamente análogos.

```
program salida_tabular
  implicit none
  real, parameter :: TERCIO = 1.0 / 3.0
  real :: x
  integer :: i

  do i = 1, 5
    x = i
    print '(F15.4, F15.4, F15.4)', x, sqrt(x), x**TERCIO
  end do
end program salida_tabular
```

Figura 1.11: Salida en forma de tabla

```
123456789012345678901234567890123456789012345678901234567890
         1.0000          1.0000          1.0000
         2.0000          1.4142          1.2599
         3.0000          1.7321          1.4422
         4.0000          2.0000          1.5874
         5.0000          2.2361          1.7100
```

Figura 1.12: Salida del programa de la figura 1.11

El programa de la figura 1.11 produce la salida que vemos en la figura 1.12 (la primera linea no corresponde a la salida, sino que se ha incluido a efectos ilustrativos).

```
program ejemplo_format
  implicit none
  integer :: a, b
  real :: c, d

  read 101, a, b, c, d
  printá *,  = ', a
  print *, 'b = ', b
```

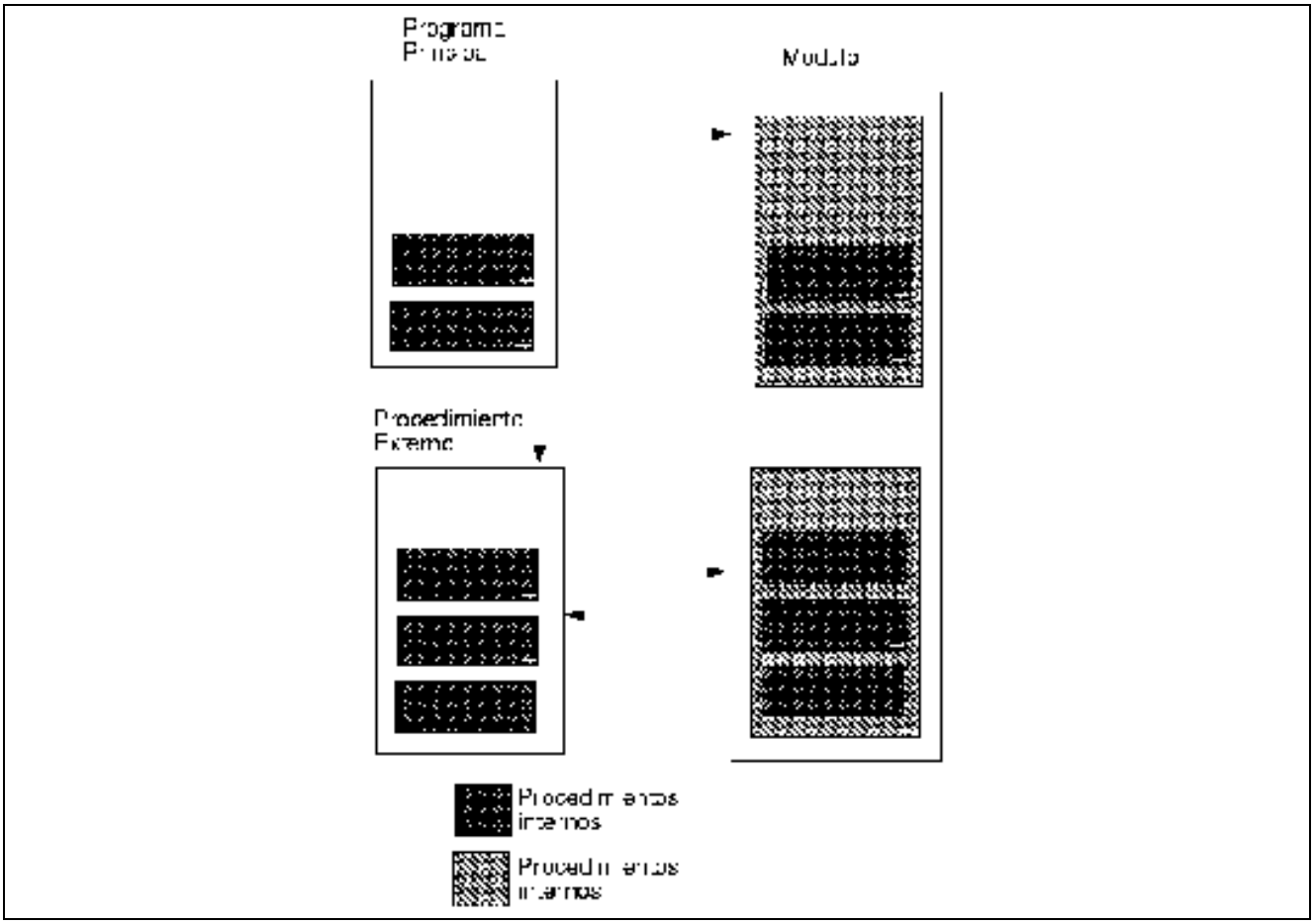

Figura 1.1: Unidades de programa

```
  200 format (5X, I4, " menos", I5, " es", I5, TR4, F6.2,  &
              " menos", F6.2, " es", F8.3)
end program ejemplo_format
```

Figura 1.13: La sentencia `format`

```
a =  6789
b =  4567
    6789 menos 4567 es 2222     234.50 menos 12.34 es 222.160
```

Figura 1.14: Salida del programa 1.13 para la entrada 123456789

Si se ejecuta el programa de la figura ?? y se le da como entrada los nueve digitos `123456789`, el resultado es el que se muestra en la figura 1.14.

4. PROCEDIMIENTOS Y MÓDULOS

4.1. Unidades de programa

En Fortran90 se denominan unidades de programa a los módulos y a los procedimientos externos. La figura 1.1 muestra las relaciones existentes entre las diferentes unidades de programa y el programa principal. Un programa sólo puede tener un único programa principal y todas las unidades de programa que se desee. Los módulos se utilizan para que diferentes programas compartan código (datos y subrutinas).

Un programa principal siempre tiene la siguiente forma:

```
program <nombre>
  <sentencias de especificacion>
  <sentencias ejecutables>
  ...
end [program [<nombre}>]]
```

```
program prueba
```

```
  ! end
  ! end program
end program prueba
```

Figura 1.15: La estructura de un programa principal

La figura 1.15 presenta un ejemplo básico de programa principal.

4.2. Procedimientos

De aquí en adelante llamaremos procedimientos de forma genérica tanto a funciones como a subrutinas. La diferencia entre funciones y subrutinas es que una función devuelve un único resultado y no suele modificar los valores de sus parámetros. Las subrutinas realizan tareas generalmente más complejas y retornan valores a través de sus parámetros.

Estructuralmente los procedimientos pueden ser:

1. Externos (autocontenidos, no necesariamente escritos en Fortran)
2. Internos (dentro de una unidad de programa)
3. De módulos (miembros de un módulo)

Fortran77 sólo dispone de procedimientos externos. Un bloque `interface` se utiliza para definir detalles relativos a los argumentos de un procedimiento. Es obligatoria su presencia para los procedimientos externos.

4.2.1. Procedimientos externos

Un procedimiento externo tiene la forma:

```
subroutine; <nombre>(<lista de áparmetros formales>)
  [<sentencias de óespecificacin>]
  [<sentencias ejecutables>]
  ...
end [subroutine [<nombre}>]]
```

o bien:

```
function <nombre>(<lista de parametros formales>)
  [<sentencias de especificacion>]
  [<sentencias ejecutables>]
  ...
end [function [<nombre>]]
```

4.3. Procedimientos internos

Cualquier unidad de programa (un programa principal, un módulo o un procedimiento externo) puede contener procedimientos internos. Los procedimientos internos se colocan todos juntos al final de una unidad de programa y precedidos por la sentencia `contains`. Tienen la misma forma que los procedimientos externos excepto en que la palabra `subroutine` o `function` ha de aparecer obligatoriamente en la sentencia `end`. Las variables definidas en la unidad de programa permanecen accesibles (definidas) en los procedimientos internos, a menos que se redefinan allí.

Es una buena práctica declarar todas las variables que se utilicen en un subprograma (parámetros y variables locales). No se permite anidar procedimientos internos.

```
program principal
implicit none
real :: a, b, c
real :: suma
...
suma = sumar()
...
  contains
  function sumar()
    real :: suma            ! a, b y c ya han sido definidas

    sumar = a + b + c
  end function sumar
...
end program principal
```

La figura 1.16 presenta un ejemplo en el que la función sumar() es un procedimiento interno.

La cláusula `implicit none` en una unidad de programa también tiene efecto sobre todos los procedimientos internos que contenga. No obstante, se recomienda volver a usarla, tanto por claridad como para evitar errores.

```
subroutine aritmetica(n, x, y, z)
  implicit none
  integer :: n
  real, dimension(100) :: x, y, z
   ...
  contains

  function sumar(a, b, c) result(suma)
    implicit none
    real, intent(in) :: a, b, c
    real :: suma

    suma = a + b + c
  end function sumar

end subroutine aritmetica
```

Figura 1.17: Una función dentro de una subrutina

La figura 1.17 muestra un ejemplo en el que se repite (a pesar de no ser estrictamente necesario) la cláusula `implicit none`.

4.4. Bloques `interface`

Si se suministra explícitamente una interface para un procedimiento, el compilador puede comprobar inconsistencias en los argumentos (en cuanto a número y tipo de los mismos). En el caso de los subprogramas intrínsecos, los subprogramas internos y los módulos, el compilador conoce siempre esta información y se dice que es explícita. Cuando se invoca un subprograma externo, esta información no está disponible y se dice que es implícita. Fortran90 utiliza los bloques interface para especificar una interface explícita para los procedimientos externos.

Hay que utilizar siempre bloques interface en las unidades de programa que invoquen procedimientos externos. La forma general de un bloque `interface` es:

```
interface
  <cuerpo de la interface>
  ...
end interface            ! Ojo: no se pone nombre
```

donde el cuerpo de la interface es una copia exacta de la especificación del subprograma, su especificación de argumentos y su sentencia `end`.

Veamos un ejemplo de `interface`:

```
interface
  real function func(x)
  real, intent(in) :: x
  end function func
end interface
```

El bloque interface se ha de colocar en la unidad de programa que realiza la llamada.

La cláusula `intent` que aparece en el ejemplo anterior sirve para especificar si un argumento de un procedimiento es para:

- entrada (in) (no puede ser modificado)
- salida (out)
- o ambos (inout)

```
integer, intent(in) :: x
real, intent(out) :: y
real, intent(inout) :: z

subroutine intercambia(a, b)
  implicit none
```

```
  real :: temp

  temp = a
  a = b
  b = temp
end subroutine intercambia
```

Figura 1.18: Ejemplo de utilización de la cláusula `intent`

El código de la figura 1.18 utiliza la cláusula `intent` en una subrutina que intercambia el valor de sus argumentos. La llamada a la función de la figura 1.18 tendría la forma:

```
call intercambia(x, y)
```

4.4.1. Argumentos con nombre

En Fortran90, los argumentos con nombre se utilizan para evitar confusión cuando un procedimiento tiene varios argumentos. Evitan el tener que recordar el orden de los parámetros. Por ejemplo, la llamada a la función:

```
real function area(inicio, final, tol)
  implicit none
  real, intent(in) :: inicio, final, tol
  ...
end function area
```

podría hacerse de cualquiera de las siguientes formas:

```
a = area(0.0, 100.0, 0.01)
b = area(inicio = 0.0, tol = 0.01, final = 100.0)
c = area(0.0, tol = 0.01, final = 100.0)
```

Una vez que se usa un argumento con nombre, el resto de parámetros también han de usar nombre. Así no es posible:

```
c = area(0.0, tol = 0.01, 100.0)    ! prohibido
```

En este ejemplo no se necesita una `interface` porque area() es una función interna. Sí haría falta si se tratara de un procedimiento externo con argumentos.

4.4.2. Argumentos opcionales

En ciertas ocasiones, no es necesario que todos los argumentos estén presentes en una llamada. Un argumento se puede declarar como opcional:

```
real function area(inicio, final, tol)
  implicit none
  real, intent(in), optional :: inicio, final, tol
  ...
end function area
```

Y la llamada podría tener cualquiera de las siguientes formas:

```
a = area(0.0, 100.0, 0.010)
b = area(inicio = 0.0, final = 100.0, tol = 0.01)
c = area(0.0)
d = area(0.0, tol = 0.01)
```

La función lógica (intrínseca) present() se utiliza para comprobar la presencia de un parámetro opcional.

```
real function area(inicio, final, tol)
  implicit none
  real, intent(in), optional :: inicio, final, tol
  real ttol
  ...
  if (present(tol)) then
    ttol = tol
  else
    ttol = 0.01
  end if
end function area
```

El código de la figura 1.19 muestra un ejemplo de la utilización de present(). En ese ejemplo, no se podría modificar tol, porque se trata de un parámetro `intent(in)` y por ello se usa una variable local, ttol.

Si el procedimiento es externo y tiene argumentos opcionales, se ha de suministrar una interface. Si la función del ejemplo anterior fuera externa sería necesario el siguiente bloque interface:

```
interface
  real function area(inicio, final, tol)
  real, intent(in), optional :: inicio, final, tol
  end function area
end interface
```

4.4.3. Tipos derivados como argumentos

Los argumentos de un procedimiento pueden ser de un tipo derivado si:

- El procedimiento es interno a la unidad de programa en la que se define el tipo derivado
- El tipo derivado se define en un módulo que es accesible desde el procedimiento

4.5. Argumentos procedurales

En Fortran90 un procedimiento que es pasado como argumento debe ser un procedimiento externo o un procedimiento de módulo. No se permite pasar como parámetro un procedimiento interno. Para procedimientos externos se recomienda suministrar siempre un bloque `interface` en la unidad de programa que realiza la llamada.

Hemos de tener en cuenta que un procedimiento de módulo tiene una interface explícita por defecto.

```
! La unidad de programa que realiza la llamada:
...
interface
  real function func(x, y)
  real, intent(in) :: x, y
  end function func
end interface
...
call area(func, start, finish, tol)

! La funcion externa:

real function func(x, y)
  implicit none
  real, intent(in) :: x, y
  ...
end function func
```

Figura 1.20: Un ejemplo de uso de parámetros procedurales

En el código de la figura 1.20 la función func() se pasa como como parámetro a la subrutina area().

4.6. Cláusula `result` para funciones

Las funciones pueden tener una variable resultado. El identificador de resultado que se utilice dentro de la función ha de ser especificado entre paréntesis al final de la sentencia `function`. Veamos un ejemplo:

```
function sumar(a, b, c) result(suma_abc)
  implicit none
  real, intent(in) :: a, b, c
  real :: suma_abc

  suma_abc = a + b + c
end function sumar
```

Las funciones directamente recursivas (la recursividad la estudiaremos en el epígrafe 4.8.) han de tener forzosamente una variable `result`.

4.7. Funciones que devuelven vectores

El resultado de una función no tiene porqué ser un escalar. El tipo de una función que devuelve un vector no se especifica en la sentencia `function` inicial sino en una declaración de tipo en el cuerpo de la función, en la que hay que especificar las dimensiones del vector.

```
function sumar_vec (a, b, n)
  implicit none
  real, dimension (n) :: sumar_vec
  integer, intent(in) :: n
  real, dimension (n), intent(in) :: a, b

  do i=1, n
    sumar_vec(i) = a(i) + b(i)
  end do
end function sumar_vec
```

Figura 1.21: La función devuelve el vector suma de a y b

La figura 1.21 muestra una función que devuelve como resultado un vector de enteros. Si la función fuera externa, habría que especificar una interface en el programa llamador.

4.8. Procedimientos recursivos

En Fortran90 los procedimientos se pueden invocar recursivamente. Si P1 y P2 son dos procedimientos (funciones o subrutinas) puede ocurrir:

- P1 invoca a P2 y P2 invoca a P1 (esta cadena de llamadas puede ser arbitrariamente larga) o bien
- P1 invoca directamente a P1 (en este caso se requiere `result`)

```
recursive function fact(n) result (res)
  implicit none
  integer intent(in) :: n
  integer :: res

  if (n == 1) then
    res = 1
  else
    res = n * fact(n - 1)
  endif
end function fact
```

Figura 1.22: La funcion factorial implementada recursivamente

Si un procedimiento es recursivo ha de declararse como tal, de la forma que muestra el código de la figura 1.22.

Antes de finalizar este apartado dedicado a la recursividad, haremos notar que siempre hemos de tener en cuenta que la recursividad es ineficiente: si un mismo algoritmo admite una solución recursiva y otra iterativa, siempre es preferible elegir la versión iterativa. En este sentido es preferible una implementación iterativa de una función para calcular un factorial que la que hemos presentado en la figura 1.22.

4.9. Procedimientos genéricos

Cuando se utilizan procedimientos genéricos, se usa un determinado identificador para el procedimiento y el código que se ejecuta depende del tipo de los argumentos (tal como ocurre por ejemplo en C++).

Un procedimiento genérico se define utilizando un bloque `interface` y usando un identificador (genérico) para todos los procedimientos definidos dentro de ese bloque `interface`. La forma general es:

```
interface <nombre generico>
  <cuerpo especifico de la interface>
  ...
end interface
```

Todos los procedimientos de un bloque `interface` genérico han de diferenciarse de forma no ambigua y por lo tanto todos han de ser subrutinas o todos funciones.

```
subroutine intercambiareal
  implicit none
  real, intent(inout) :: a, b
  real :: temp
```

```
  temp = a
  a = b
  b = temp
end subroutine intercambiareal

subroutine intercambiaint
  implicit none
  integer, intent(inout) :: a, b
  integer :: temp

  temp = a
  a = b
  b = temp
end subroutine intercambiaint

interface intercambia

  subroutine intercambiareal (a, b)
  real, intent(inout) :: a, b
  end subroutine intercambiareal

  subroutine intercambiaint (a, b)
  integer, intent(inout) :: a, b
  end subroutine intercambiaint

end interface
```

Figura 1.23: Subrutinas genéricas para intercambiar valores de diferentes tipos

El código de la figura 1.23 muestra un par de subrutinas genéricas para intercambiar valores de tipo real o entero y también un bloque `interface` para esas rutinas. La llamada para intercambiar dos valores se realizaría mediante:

```
call intercambia(x, y)
```

y se invocaría a una subrutina u otra dependiendo del tipo de los parámetros x e y.

4.10. Módulos

Los módulos son una característica del lenguaje que incide positivamente sobre la estructura de los programas escritos en Fortran90. Los módulos juegan un papel importante en la definición de tipos y operadores asociados La forma general de una definición de módulo es:

```
module <nombre>
  [<sentencias de especificacion>]
  [<sentencias ejecutables>]
contains
  [<procedimientos del modulo>]
end [module [<nombre>]]
```

Al módulo se accede a través de la sentencia `use`. En fortran las variables son habitualmente entidades locales. Utilizando módulos es posible hacer accesibles un conjunto de datos a diferentes unidades de programa:

```
module globales
  real, save :: a, b, c
  integer, save :: i, j, k
end module globales
```

El atributo `save` permite declarar datos como globales (es un sustituto del `common` de Fortran77). Los datos se utilizan en otras unidades de programa a través de la sentencia `use`:

```
use globales
```

La sentencia `use` no es ejecutable y debe aparecer al principio de la unidad de programa antes que cualquier otra sentencia y después de `program`, `function` o `subroutine`.

Una unidad de programa puede invocar código de diferentes módulos utilizando una serie de sentencias `use`. Nótese que un módulo puede usar a su vez otros módulos, pero un módulo no puede usarse a sí mismo (ni directa ni indirectamente).

```
use globales
! permite acceder a todas las variables del modulo
```

```
use globales, only : a, c
! permite acceder solamente a las variables a  y c

use globales, r=>a, s=>b
! permite acceder a las variables a y b
! a étravs de los identificadores r y s
```

Figura 1.24: La sentencia `use`

La figura 1.24 presenta un ejemplo de utilización de la sentencia `use`. Los procedimientos que se especifican dentro de un módulo se llaman procedimientos de módulo. Se trata de procedimientos que pueden ser accedidos (utilizados) por otras unidades de programa y es posible que haya varios procedimientos de módulo contenidos en el mismo módulo. Los procedimientos de módulo han de estar codificados en fortran (los externos pueden estar en otro lenguaje) y tienen la misma forma que los procedimientos externos salvo que:

- Los procedimientos de un módulo deben aparecer después de una sentencia `contains`
- La sentencia `end` ha de tener especificada una `subroutine` o `function`

Los procedimientos de módulo se invocan mediante la sentencia `call` o referencia a la función, pero sólo desde una unidad de programa que haya declarado con una sentencia `use` la utilización del módulo en cuestión.

Los procedimientos de módulo son especialmente útiles para un conjunto de tipos derivados y sus correspondientes operaciones.

```
module modulo_puntos
  type punto
    real :: x, y
  end type punto

contains
  function suma_puntos(p, q)
    type (punto), intent(in) :: p, q
    type (punto) :: suma_puntos
    suma_puntos%x = p%x + q%x
    suma_puntos%y = p%y + q%y
  end function suma_puntos

end module modulo_puntos
```

Figura 1.25: Un módulo que opera con puntos en el plano

El código de la figura 1.25 muestra un ejemplo de utilización de un procedimiento de módulo. El programa podría declarar:

```
use modulo_puntos
type (punto) :: px, py, pz
  ...
pz = suma_puntos(px, py)
```

4.10.1. Procedimientos genéricos

La utilización de módulos permite argumentos de tipos derivados y por tanto procedimientos genéricos con tipos derivados.

```
module intercambio_generico
  implicit none

  type punto
    real :: x, y
  end type punto

  interface intercambia
    module procedure intercambia\_real, intercambia\_int, &&
                     intercambia\_log, intercambia\_punto
  end interface

contains
  subroutine intercambia\_punto (a, b)
    implicit none
```

```
    temp = a
    a = b
    b = temp
  end subroutine intercambia\_punto

  subroutine intercambia\_real
    implicit none
    real, intent(inout) :: a, b
    real :: temp
    temp = a
    a = b
    b = temp
  end subroutine intercambia\_real

  ! rutinas similares para intercambiar
  ! otros tipos de datos
  ...
end module intercambio_generico
```

Figura 1.26: Un módulo con procedimientos genéricos

El código de la figura 1.26 es una extensión del procedimiento de intercambio genérico que aparece en la figura 1.23.

Por defecto, todas las entidades de un módulo son accesibles a las unidades de programa que utilicen la sentencia `use`. A veces interesa prohibir el uso de ciertas entidades al programa llamador para forzar al usuario a utilizar las rutinas del módulo en lugar de las suyas propias o también para permitir flexibilidad a la hora de realizar cambios internos sin necesidad de informar a los usuarios del módulo o cambiar la documentación. Esto se consigue a través de la sentencia `private`:

```
private :: sub1, sub2
```

o el atributo `private`:

```
integer, private, save :: fila_actual, columna_actual
```

4.11. Sobrecarga de operadores

En Fortran90 se puede extender el significado de un operador intrínseco para que actúe sobre tipos de datos adicionales: a ello se le llama sobrecarga del operador. Para sobrecargar un operador se necesita un bloque `interface` de la forma:

```
interface operator; (<operador intrinseco>)
  <cuerpo de la interface>
end interface;
```

```
module sobrecarga
  implicit none
    ...
  interface operator (+)
    module procedure concatena
  end interface
    ...
contains
  function concatena(cha, chb)
    implicit none
    character (len = *), intent(in) :: cha, chb
    character (len = (len_trim(cha) + len_trim(chb))) :: concatena

    concatena = trim(cha)//trim(chb)
  end function concatena
  ...
end module sobrecarga
```

Figura 1.27: Módulo para sobrecargar el operador '+'

Por ejemplo, el operador + podría extenderse a variables de tipo carácter para concatenaenar dos cadenas de caracteres ignorando los espacios sobrantes, y el código podría colocarse en un módulo, tal como se muestra en la figura 1.27.

Ahora la expresión `cha + chb` tiene significado en cualquier programa que '`use`' este módulo. Obsérvese en el ejemplo la presencia del bloque `interface`. La función que define el operador está en un módulo y no es

Cuando se da un nombre genérico o un operador para un conjunto de procedimientos, se necesita un bloque de interface. El bloque `interface` tiene la forma:

```
interface ...
  module procedure <lista>
end interface
```

Donde <lista> ha de contener los identificadores de los procedimientos de módulo implicados.

4.12. Definición de operadores

En Fortran90 es posible definir nuevos operadores, lo cual resulta especialmente útil para manipular tipos definidos por el usuario. Los operadores definidos han de tener '.' (punto) al principio y al final. Por ejemplo, en el ejemplo de la figura 1.27 podríamos haber definido `.mas.` en lugar de sobrecargar +

La operación se ha de definir a través de una función que tenga uno o dos argumentos no opcionales con atributo `intent(in)`.

```
program main
implicit none
use mod_distancia
type(point) :: px, py
  ...
distancia = px .dist. py
  ...
end program main

module mod_distancia
  implicit none
  ...
  type punto
    real :: x, y
  end type punto
  ...
  interface operator (.dist.)
    module procedure calcula_dist
  end interface
  ...
contains
  ...
  function calcula_dist (px, py)
    implicit none
    real :: calcula_dist
    type (punto), intent(in) :: px, py

    calcula_dist = sqrt ((px%x-py%x)**2 +(px%y-py%y)**2)
  end function calcula_dist
  ...
end module mod_distancia
```

Figura 1.28: Cálculo de la distancia euclídea entre dos puntos

El ejemplo de la figura 1.28 muestra la definición de un operador distancia (.dist.).

4.13. Sobrecarga del operador de asignación

Al usar tipos derivados puede ser necesario extender el significado del operador de asignación (=):

```
real :: ax
type (punto) :: px
...
ax = px          ! se asigna un punto a un real
...              ! no es ávlido si no se define
```

Siguiendo con el ejemplo, supongamos que pretendemos que ax tome el valor máximo de las componentes de px. Esta asignación se ha de hacer a través de una subrutina con dos argumentos no opcionales, el primero de ellos `intent(out)` o `intent(inout)` y el segundo `intent(in)`. Debe crearse un bloque interface de asignación:

```
interface assignment (=)
  <cuerpo de la interface de la subrutina>
```

Figura 1.2: Estructura de los programas

```
module mod_sobre_assign
  implicit none
  type punto
    real :: x, y
  end type punto
  ...
  interface assignment (=)
    module procedure assign_punto
  end interface
contains
  subroutine assign_punto (ax, px)
    real, intent(out) :: ax
    type (punto), intent(in) :: px
    ax = max(px%x, px%y)
  end subroutine assign_punto
  ...
end module mod_sobre_assign
```

Figura 1.29: Sobrecarga de la asignación

La definición de la asignación se podría colocar en un módulo como el que se presenta en la figura 1.29. El programa principal necesita invocar a este módulo con una sentencia `use`:

```
use mod_sobre_assign
real :: ax
type (punto) :: px
...
ax = px          ! correcto
```

4.14. Estructura de los programas

4.14.1. Cuándo utilizar bloques `interface`

- si el módulo define o sobrecarga un operador o sobrecarga la asignación; o bien
- si el módulo utiliza un identificador genérico

Se utiliza también un bloque interface cuando un procedimiento externo:

- Es invocado con un argumento con nombre y/o opcional
- Es una función que devuelve un vector o un puntero o bien una función de caracteres que no es ni constante ni de longitud conocida
- Tiene un argumento que es un vector de forma asumida o un puntero
- Es argumento de otro subprograma (en este caso no es obligatorio, pero se recomienda)

5. PROCESAMIENTO DE VECTORES

5.1. Terminología

De forma general hablaremos de vectores (arrays) para referirnos a vectores multidimensionales (matrices). Fortran90 permite vectores de hasta 7 dimensiones. Consideremos el siguiente ejemplo en el que w, x, y, z son vectores de 50 elementos:

```
real, dimension(50) :: w
real, dimension(5:54) :: x
real y(50)
real z(11:60)
```

E introduzcamos la siguiente terminología básica:

- Rango de un vector: es su número de dimensiones
- Extensión: es el número de elementos en una dimensión
- Forma: es un vector de extensiones. El vector contiene la extensión de cada dimensión
- Tamaño: es el número de elementos. es el producto de extensiones
- Dos vectores son compatibles si tienen la misma forma. Cualquier vector es compatible con un escalar

Por ejemplo, si consideramos:

```
real, dimension :: a(-3:4, 7)
real, dimension :: b(8, 2:8)
real, dimension :: d(8, 1:8)
integer :: c
```

El vector a tiene:

- rango: 2
- extensiones: 8 y 7
- forma: (/8, 7/)
- tamaño: 56

El vector a es compatible con b y c, pero no con d (d tiene forma (/8, 9/).

5.2. Declaración de vectores

La forma general de una declaración es:

```
<tipo> [[, dimension (<lista de extensiones>)] [, <atributo>] ... ::] &
  <lista de identificadores>
```

Donde:

- tipo: ha de ser un tipo básico o derivado
- `dimension` es opcional, pero necesario para definir dimensiones por defecto

Una constante entera

Una expresión entera usando argumentos o constantes ficticios

':' para mostrar que el vector es dinámico o de forma asumida

- Los atributos pueden ser los que ya hemos estudiado: `parameter`, `public`, `private`, `pointer`, `target`, `allocatable`, `intent(inout)`, `dimension(<lista de extensiones>)`, `optional`, `save`, `external` o `intrinsic`.
- Los identificadores de la lista opcionalmente pueden tener dimensiones y valores iniciales.

Veamos varios ejemplos de declaraciones de vectores. En primer lugar, la inicialización de dos vectores unidimensionales conteniendo 3 elementos:

```
integer, dimension(3) :: ia = (/1, 2, 3/), ib = (/(i, i=1, 3)/)
```

Declaración de un vector automático lb:

```
logical, dimension(size(la)) :: lb
```

la es un vector de argumentos ficticios y size() una función intrínseca que devuelve el tamaño del vector la. Declaración de dos vectores bidimensionales dinámicos a y b:

```
real, dimension (:, :), allocatable :: a, b
```

La forma (número de elementos en cada dimensión) se definiría con una sentencia `allocate` posterior. Declaración de 2 vectores tridimentionales de forma asumida:

```
real, dimension(:,:,:) :: a, b
```

La forma se tomaría de los parámetros actuales de la rutina llamadora.

5.3. Operaciones sobre vectores

En Fortran90 es posible realizar operaciones sobre todos los elementos de un vector sin necesidad de usar bucles. Para ello, los vectores implicados deben ser compatibles. Las operaciones entre dos vectores compatibles se realizan elemento a elemento y todos los operadores intrínsecos están definidos para vectores compatibles. Por ejemplo, podemos hacer el producto:

```
3 4 8       5 2 1       15 8  8
5 6 6   *   3 3 1   =   15 18 6
3 5 7       2 1 3       6  5  21
```

En el que el resultado se calcula multiplicando elemento a elemento.

Si uno de los operandos es un escalar, éste se "expande" para ser compatible con el otro operando. Esta expansión de escalares es útil a la hora de inicializar o escalar vectores.

Un concepto fundamental con respecto a las asignaciones en operaciones sobre vectores es que la evaluación de la expresión del lado derecho de la asignación se realiza antes de que ocurra cualquier asignación. Esto es importante cuando aparecen vectores en ambos lados de la asignación: si no fuera así los elementos de los vectores de la parte derecha se podrían ver afectados antes de que la operación se completara.

Veamos un ejemplo en el que consideremos tres vectores del mismo tamaño. Asignemos 0 a todos los elementos y luego haremos

```
a(i) = a(i) / 3.1 + b(i) * sqrt(c(i))
```

para todo i:

```
real, dimension(20) :: a, b, c
...
a = 0
...
a = a / 3.1 + b * sqrt(c)      ! sqrt para cada elemento
```

Consideremos ahora tres matrices bidimensionales de la misma forma. Multiplicaremos dos de las matrices elemento a elemento asignando el resultado a la tercera matriz:

```
real, dimension (5, 5) :: a, b, c
...
c = a * b
```

Ahora el problema consistirá en hallar el máximo valor menor que 1000 en una matriz tridimensional. El código es:

```
real, dimension(10, 10, 10) :: a
```

mask() es una función que devuelve un vector lógico: sólo los elementos de a que se corresponden con elementos de mask() participan en la llamada a maxval().

Veamos la solución a otro problema: hallar el valor medio de los valores mayores que 3000 en un vector:

```
av = sum(a, mask = (a > 3000)) / count(mask = (a > 3000))
```

Hemos utilizado algunas funciones intrínsecas sobre vectores:

- maxval() retorna el valor máximo de los elementos de un vector
- sum() la suma de los elementos del vector
- count() el número de elementos true

5.3.1. Procedimientos intrínsecos elementales

Fortran90 permite procedimientos intrínsecos sobre vectores. El procedimiento se aplica a cada elemento del vector (matriz). De nuevo, los operandos deben ser compatibles. Por ejemplo, para hallar la raíz cuadrada de todos los elementos de una matriz a:

```
b = sqrt(a)
```

Para hallar la longitud de una cadena dada por un vector de caracteres ch, ignorando los espacios en blanco:

```
longitud = len_trim(ch)
```

5.4. Sentencia `where`

La sentencia **where** se usa para realizar asignaciones sólo si se cumple una condición lógica, y es útil para para realizar operaciones sobre ciertos elementos de un vector. Un ejemplo simple es evitar una división por cero:

```
real, dimension(5, 5) : ra, rb
...
where(rb > 0.0) ra = ra / rb
```

La forma general de la sentencia es:

```
where(<expr. logica sobre un vector}>) <identificador vector> = <expresion vectorial>
```

Se evalúa la expresión lógica y todos los elementos de la expresión vectorial que tienen un valor true se evalúan y asignan. Los elementos que resultan false no se alteran. La expresión lógica ha de tener la misma forma que el vector.

```
real, dimension(5, 5) :: ra, rb
...
where(rb > 0.0)
  ra = ra / rb
elsewhere
  ra = 0.0
end where
```

Figura 1.30: Asignación controlada con un vector lógico

También es posible controlar asignaciones sobre un vector a través de un vector lógico tal como vemos en el ejemplo de la figura 1.30.

5.5. Secciones de vectores

Una parte de un vector (a lo que denominaremos una sección) puede referenciarse especificando un rango de subíndices mediante un subíndice simple, como en

```
ra(2, 2)
```

o bien mediante una tripleta de subíndices de la forma:

```
[<limite inferior>]:[<limite superior>][:<salto>]
```

Por defecto se toman los límites declarados y salto = 1.

Los siguientes ejemplos muestran secciones de vectores utilizando tripletas de subíndices: Un elemento. Forma: (/1/):

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} = ra(2:2, 2:2)$$

Una subcolumna. Forma: (/3/):

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & X & X & X \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} = ra(3, 3:5)$$

Una columna. Forma: (/5/):

$$\begin{pmatrix} 0 & 0 & X & 0 & 0 \\ 0 & 0 & X & 0 & 0 \\ 0 & 0 & X & 0 & 0 \\ 0 & 0 & X & 0 & 0 \\ 0 & 0 & X & 0 & 0 \end{pmatrix} = ra(:, 3)$$

Salto 2 en filas. Forma: (/3, 3/):

$$\begin{pmatrix} 0 & X & X & X & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & X & X & X & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & X & X & X & 0 \end{pmatrix} = ra(1::2, 2:2)$$

Un vector de subíndices es una expresión entera de rango 1 (vector de enteros). Cada elemento de la expresión ha de definirse con valores en el rango de los límites del vector 'padre'. Los elementos del vector de subíndices pueden estar en cualquier orden.

Un ejemplo de expresión entera de rango 1 es:

```
(/3, 2, 12, 2, 1/)
```

```
real, dimension :: ra(6), rb(3)
integer, dimension (3) :: iv
iv = (/ 1, 3, 5 /)                    ! expr. entera de rango 1
ra = (/ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 /)
rb = ra(iv)                           ! iv es el vector de sub\'{\i}ndices
! = (/ ra(1), ra(3), ra(5) /)
! = (/ 1.2, 3.0, 1.0 /)

ra(iv) = (/1.2, 3.4, 5.6/)
!= ra((/1, 3, 5/)) = (/1.2, 3.4, 5.6/)
!= ra(1:5:2) = (/1.2, 3.4, 5.6/)
```

Figura 1.31: Utilización de vectores de subíndices

La figura 1.31 muestra algunos ejemplos de la utilización de vectores de subíndices. Tanto vectores completos como secciones de vectores se pueden usar como operandos en asignaciones con vectores, siempre que sean compatibles.

```
real, dimension(5, 5) :: ra, rb, rc
integer :: i
...
! forma (/5, 5/) y escalar
ra = rb + rc * i

! forma (/3, 2/)
ra(3:5, 3:4) = rb(1::2, 3:5:2) + rc(1:3, :2)

! forma (/5/)
ra(:, 1) = rb(:, 1) + rb(:, 2) + rc(:, 3)
```

Figura 1.32: Vectores como operandos en asignaciones vectoriales

Vemos un ejemplo en la figura 1.32.

Es importante saber cómo obtener recursividad a la hora de operar con vectores en Fortran90. Por ejemplo el código:

```
do i = 2, n
  x(i) = x(i) + x(i - 1)
end do
```

No produce el mismo resultado que:

```
x(2:n)= x(2:n) + x(1:n-1)
```

En el primer caso, la asignación es:

```
x(i) = x(i) + x(i-1) + x(i-2) + ... + x(1)
```

Mientras que en el segundo:

```
x(i) = x(i) + x(i-1)
```

En Fortran90 se puede obtener el efecto recursivo del bucle do utilizando la función intrínseca sum(), que devuelve la suma de todos los elementos del vector argumento:

```
x(2:n) = (/(sum(x(1:i)), i= 2, n)/)
```

5.6. Vectores de tamaño cero

Si el límite inferior de un vector es mayor que el superior, el vector tiene tamaño cero. Estos vectores siguen las reglas normales y resultan útiles para operaciones de contorno (no se necesita código especial para tratar los límites de un modo especial):

```
do i=1, n
  x(i) = b(i) / a(i, i)
  b(i + 1:n) = b(i + 1:n) - a(i + 1:n, i) * x(i)
  ! tamano cero cuando i=n
end do
```

5.7. Constructores de vectores

Un constructor de vectores crea un vector de rango 1 conteniendo valores especificados. Los valores se pueden dar listándolos o bien usando un bucle implícito o una combinación de ambos. Veamos un ejemplo:

```
real, dimension(6) :: a
a=(/<lista de valores>/)
```

Donde <lista de valores> puede tener diferentes formas; por ejemplo:

```
(/(i, i = 1, 6)/)              ! = (/1, 2, 3, 4, 5, 6/)
(/7, (i, i=1, 4), 9/)          ! = (/7, 1, 2, 3, 4, 9/)
(/1.0/real(i), i = 1, 6)/)
!=(/1.0/1.0, 1.0/2.0, 1.0/3.0, 1.0/4.0, 1.0/5.0, 1.0/6.0/)

(/((i+j, i=1, 3), j=1, 2)/)
! = (/((1+j, 2+j, 3+j), j = 1, 2)/)
! = (/2, 3, 4, 3, 4, 5/)

(/a(i, 2:4), a(1:5:2, i + 3)/)
!=(/a(i, 2), a(i, 3), a(i,4), a(1,i+3), a(3,i+3), a(5,i+3)/)
```

En Fortran90:

- Se permite obtener y liberar memoria dinámica a través de los vectores dinámicos
- Se permite a los vectores locales a un procedimiento tener tamaños y formas diferentes en cada llamada a través de los vectores automáticos
- Se reducen los recursos globales necesarios para almacenamiento en memoria
- Se simplifica los argumentos de las subrutinas

5.8. Vectores dinámicos

Un vector dinámico se declara en una sentencia de declaración de tipo con el atributo `allocatable`. El rango del vector debe especificarse en la declaración incluyendo el número adecuado de ′:′ en el atributo dimension. Un vector bidimensional se podría declarar:

```
real, dimension(:,:), allocatable :: a
```

Hemos de tener en cuenta que la declaración no reserva memoria para el vector, sino que hay que realizarla utilizando la sentencia `allocate()` (`deallocate()` se utiliza para liberar la memoria):

```
allocate (a(0:n, m))
allocate (a(0:n+1, m))
deallocate (a)
```

La forma general de estas sentencias es:

```
allocate;(<lista de identificadores> [, stat=;<valor>])
deallocate;(<lista de identificadores> [, stat=;<valor>])
```

Si stat está presente, toma el valor 0 si la operación se realizó correctamente o un valor positivo si hubo un error.

```
integer n
real, dimension(:,:), allocatable :: ra
integer :: estado
...
read(*, *) tam1, tam2
allocate(ra(tam1, tam2), stat = estado)
if (estado > 0) then
  ! ... hubo un error ...
end if
...
deallocate (ra)
```

Figura 1.33: `allocate` y `deallocate`

La figura 1.33 ilustra la utilización de estas sentencias.

Se permite alojar/desalojar varios vectores en la misma sentencia. Y también se dispone de funciones intrínsecas que indican si un vector ha sido alojado o no:

```
if (allocated(a)) deallocate(a)
...
if (.not. allocated(a)) allocate(a(5, 20))
```

Si un vector dinámico se declara con el atributo save:

```
real, dimension(:), allocatable, save :: a
```

El vector estará disponible incluso a la salida del procedimiento en que fue alojado.

Los vectores dinámicos:

- Han de ser creados y liberados dentro de la misma unidad de programa
- El resultado de una función no puede ser un vector dinámico
- No se puede usar vectores dinámicos en una definición de tipo derivado

5.9. Vectores automáticos

Llamaremos vector automático a un vector que tiene forma explícita dentro de un procedimiento y cuyos límites se suministran cuando el procedimiento es invocado

- A través de los parámetros formales
- A través de variables definidas a través de asociación por uso o asociación por hospedaje

Asociación por uso es cuando las variables declaradas en el cuerpo de un módulo se hacen disponibles a un programa a través de una sentencia `use`. La asociación por hospedaje ocurre cuando las variables declaradas en una unidad de programa se hacen disponibles a sus procedimientos internos.

Los vectores automáticos se crean al entrar al procedimiento y se liberan al salir del mismo y no existe un mecanismo para comprobar si hay suficiente memoria para un vector automático. Es frecuente usar la función

```
size(array [,dim])
```

```
subroutine sub(n, a)
  implicit none
  integer :: n
  real, dimension(n, n), intent(inout) :: a
  real, dimension(n, n) :: vec1
  real, dimension(size(a, 1)) :: vec2
    ...
end subroutine sub
```

Figura 1.34: Vectores automáticos

En el código de la figura 1.34, los vectores vec1 y vec2 toman su tamaño de los parámetros formales n y a.

```
module auto_mod
  implicit none
  integer :: n = 1   ! por defecto n=1
contains
  subroutine sub
    implicit none
    real, dimension(n) :: w

    write (*, *) 'Limites y no. de elementos de w: ',  &
                 lbound(w), ubound(w), size(w)
  end subroutine sub
end module auto_mod

program auto_arrays
use auto_mod
  implicit none

  n = 10
  call sub
end program auto_arrays
```

Figura 1.35: Vectores automáticos

El Código de la figura 1.35 muestra los límites de un vector automático dependiendo de una variable global definida en un módulo.

```
program vector
  implicit none
  real, allocatable, dimension(:,:) :: a
  real :: res
  integer :: N1, estado
  ...
  read(*,*) N1
  allocate(a(N1, N1), stat=estado)
  if (estado /= 0) then
    ! ... hubo un error ...
  end if
  call sub(a, N1, res)
  deallocate(a, stat=estado)
  if (estado /= 0) then
    ! ... hubo un error ...
  end if
  ...
contains
  subroutine sub(a, N1, res)
    implicit none
    integer, intent(in) :: N1
    real, intent(inout) :: res
    real, dimension(N1, N1), intent(in) :: a
    real, dimension(N1, N1):: work
    ...
    res=a(...)
    ...
  end subroutine sub
end program vector
```

En el código de la figura 1.36 se supone que el programa principal declara un vector de tamaño NxN pero a una subrutina ha de pasar como argumento una sección de tamaño N1xN1 (N1 < N). Esto se consigue mediante la utilización de vectores automáticos.

5.10. Vectores con forma asumida

Un vector de forma asumida es un vector de una subrutina, que no es local y que tiene un determinado tipo y rango. La extensión del vector se determina a través de una `interface` explícita en la unidad de programa que efectúa la llamada a la subrutina y por una especificación explícita de la forma del vector (que incluye su extensión). Así pues, la forma de un vector de forma asumida no es conocida sino que toma cualquier forma dependiendo de un parámetro actual.

Cuando se declara un vector de forma asumida cada dimensión se especifica como:

[<limite inferior>]:

donde el límite inferior se toma como 1 si se omite. Los vectores de forma asumida hacen posible el paso de vectores entre unidades de programa sin necesidad de pasar las dimensiones como argumentos. Si un procedimiento externo tiene un vector de forma asumida como parámetro formal ha de darse un bloque de interface en la unidad de programa que invoque al procedimiento.

```
subroutine sub(ra, rb, rc)
  implicit none
  real, dimension(:,:), intent(in) :: ra
  ! forma (10, 10)
  real, dimension(:,:), intent(in) :: rb
  ! forma (5, 5)
  !  = real, dimension(1:5, 1:5) :: rb
  real, dimension(0:, 2:), intent(out) :: çrc
  ! forma (5, 5)
  !  = real, dimension (0:4, 2:6) :: rc
  ...
end subroutine sub
```

Figura 1.37: Subrutina con un vector de forma asumida

```
real, dimension (0:9, 10) :: ra    ! forma (10, 10)

interface
  subroutine sub(ra, rb, rc)
  real, dimension(:,:), intent(in):: ra, rb
  real, dimension(0:, 2:), intent(out):: rc
  end subroutine sub
  end interface
  ...
  call sub (ra, ra(0:4, 2:6), ra(0:4, 2:6))
```

Figura 1.38: La interface para el código de la figura 1.37

El programa principal correspondiente al subprograma externo que aparece en la figura 1.37 con vectores de forma asumida ra, rb y rc debería tener una interface como la que se muestra en la figura 1.38.

```
program vector
  implicit none
  real, allocatable, dimension(:,:) :: a
  real :: res
  integer :: n1

  interface
    subroutine sub(a, res)
      real, dimension(:,:), intent(in) :: a
      real, dimension(size(a, 1), size(a, 2)) :: vect
    end subroutine sub
  end interface
  ...
  read (*, *) n1
  allocate (a(n1, n1))! vector ádinmico
  call sub(a, res)
  ...
contains
```

```
    real, intent(out) :: res
    real, dimension(:,:), intent(in) :: a
    ! vector de forma asumida
    real, dimension (size(a, 1), size(a, 2)) :: vect
    ! vector áautomtico
    ...
  res = a(...)
  ...
  end subroutine sub
end program vector
```

Figura 1.39: Un ejemplo de operaciones con vectores de forma asumida

El código de la figura 1.39 utiliza todos los tipos de vectores que hemos estudiado: automáticos, de forma asumida y dinámicos.

6. ADMINISTRACIÓN DE MEMORIA DINÁMICA

6.1. Significado de un puntero

Una variable de tipo puntero (de ahora en adelante diremos simplemente un puntero) tiene el atributo `pointer` y puede apuntar a otra variable de un tipo adecuado, que tendrá que tener el atributo `target` o bien a un área de memoria alojada dinámicamente.

Los punteros en Fortran90 son diferentes a los de C o Pascal: no contienen un dato en sí mismos y no deberían interpretarse como una dirección. Más bien deberían interpretarse como variables que se asocian dinámicamente (en tiempo de ejecución) con otros datos que sí tienen asignado un espacio de memoria física ("target").

Los beneficios más interesantes de la introducción de punteros son:

- Una alternativa más flexible a los arrays dinámicos
- Una herramienta para crear y manipular listas enlazadas y otras estructuras de datos dinámicas

6.2. Declaración de punteros

La forma general de declarar una variable puntero y un destino (`target`) es:

```
<tipo> [[, <atributo}>]...] pointer :: <lista de variables puntero>
<tipo> [[, <atributo}>]...] target :: <lista de variables destino>
```

donde el tipo especifica el tipo de variables que pueden ser apuntadas por el puntero (incluyendo tipos derivados). La lista de atributos da el resto de atributos del tipo de datos (si tiene alguno/s).

Una variable puntero ha de tener el mismo tipo, parámetros y rango que su variable destino. La declaración de un puntero a vectores especifica el tipo y rango de los vectores a los que puede apuntar (sólo el rango, no los límites del vector). El atributo dimensión (`dimension`) de un puntero a vectores no puede especificar una forma explícita o una forma asumida sino que debe tomar la forma de un vector de forma diferida de modo similar a como se hace con los vectores dinámicos. Así por ejemplo la declaración:

```
real, dimension(:), pointer :: p
```

declara un puntero p, que puede apuntar a vectores de reales de rango 1. La declaración:

```
real, dimension(20), pointer :: p
```

Es errónea y no es admitida por el compilador f90.

6.3. Asignaciones de punteros

Un puntero puede convertirse en un alias de una variable destino (`target`) a través de una sentencia de asignación de punteros, que es ejecutable y tiene la forma:

```
<puntero> => <objetivo>
```

donde puntero es una variable con atributo `pointer` y objetivo es una variable con atributo `target` o bien atributo `pointer`. Una vez que un puntero se convierte en alias de un destino, se puede utilizar en cualquier punto en lugar de la variable destino.

```
real, pointer :: p1, p2
real, target :: t1 = 3.4, t2 = 4.5
p1 => t1                ! p1 apunta a t1
print *, t1, p1         ! Ambos valen 3.4
p2 => t2                ! p2 apunta a t2
print *, t2, p2         ! Ambos valen 4.5

p2 => p1                ! p2 apunta al destino de p1
print *, t1, p1, p2     ! Todos valen 3.4

t1 = 5.2
print *, t1, p1, p2     ! Todos valen 5.2
```

Figura 1.40: Asignaciones de punteros

La figura 1.40 muestra algunos ejemplos de asignaciones con punteros. Téngase en cuenta que una asignación como:

```
p2 => p1 + 4.3              ! Error
```

No está permitida porque no se puede asociar un puntero con una expresión aritmética.

```
real, dimension (:), pointer :: pv1
real, dimension (:, :), pointer :: pv2
real, dimension (-3:5), target :: tv1
real, dimension (5, 10), target :: tv2
integer, dimension(3) :: v = (/4, 1, -3/)
pv1 => tv1              ! pv1 es un alias de tv1
pv1 => tv1(:)           ! pv1 apunta a tv1 con
                        ! sub\'{\i}ndices de óseccin
pv1 => tv2(4, :)        ! pv1 apunta a la fila 4
                        ! de tv2
pv2 => tv2(2:4, 4:8)    ! pv2 apunta a una
                        ! óseccin de tv2
pv1 => tv1(1:5:2)       ! pv1 apunta a una
                        ! óseccin de tv1
pv1 => tv1(v)           ! error
```

Figura 1.41: Punteros a vectores

El destino de un puntero puede ser también un vector. La figura 1.41 muestra algunos ejemplos de esta posibilidad. Hay varios aspectos que hemos de tener en cuenta:

El puntero pv1 se asocia en diferentes momentos con vectores (secciones de vector) de diferentes extensiones. Esto se permite porque lo que cuenta es el rango, no la extensión.

Si un puntero a vector se hace alias de un vector, sus extensiones son las mismas que las del vector al que apunta. Así con la asignación

```
pv1 => tv1
```

pv1 tiene los mismos límites que tv1, es decir -3:5. Si un puntero a vector apunta a una sección de vector, su límite inferior en cada dimensión se renumera siempre a 1. Así con `pv1 => tv1(:)`, donde se use el subíndice de la sección de vector los límites de pv1 son 1:9 en lugar de -3:5; de modo que pv1(1) se interpreta como tv1(-3), pv1(2) es tv1(-2), y así sucesivamente. Esta renumeración también ocurre cuando tv2 se interpreta como la sección tv2(2:4, 4:8).

Es lícito asociar un puntero a vector con una sección de vector definida por una tripleta de subíndices, pero no se permite asociarlo con una sección definida por subíndices. Así

```
pv1 => tv1(1:5:2)
```

es correcto con pv1(1) siendo un alias de tv1(1), pv1(2) de tv1(3), y pv1(3) de tv1(5), pero la asignación de punteros:

```
pv1 => tv1(v)    ! error
```

no es admitida por el compilador.

6.4. Estado de una asociación de punteros

Todo puntero tiene uno de los siguientes estados de asociación:

1. Indefinido. Cuando es inicialmente especificado en una sentencia de declaración

3. Asociado. Cuando apunta a un destino

Un puntero puede disociarse explícitamente de su destino (`target`) usando una sentencia `nullify`:

```
nullify(<lista de punteros>)
```

La función intrínseca associated() puede usarse para comprobar el estado de asociación de un puntero usando 1 ó 2 argumentos:

```
associated(p, [, t])
```

Si el segundo argumento no aparece, retorna `.true.` si el puntero p está asociado con algún destino y `.false.` en caso contrario. El segundo argumento puede ser un puntero en cuyo caso retorna `.true.` si ambos punteros están asociados al mismo destino o disociados y `.false.` en caso contario.

Una restricción de associated() es que no se le pueden pasar punteros indefinidos. Por ello se recomienda asociar un puntero después de su declaración o disociarlo explícitamente utilizando `nullify`.

```
real, pointer :: p, q                              ! indefinidos
real, target :: t = 3.4
p => t                                             ! p apunta a t
q => t                                             ! q apunta a t
print *, "associated(p)=", associated(p)           ! .t.
print *, "associated(p, q)=", associated(p, q)     ! .t.
nullify(p)
print *, "associated(p)= ", associated(p)          ! .f.
print *, "associated(p, q)=", associated(p, q)     ! .f.
...
p => t                                             ! p apunta a t
nullify(p, q)
```

Figura 1.42: Utilización de associated() y `nullify`

El código de la figura 1.42 muestra varios ejemplos de utilización de la función associated() y de la sentencia `nullify`. Nótese que la disociación de p no afectó a q a pesar de que ambos apuntaban al mismo objeto. Después de anular un puntero, puede asociársele de nuevo con el mismo u otro objeto.

6.5. Memoria dinámica

Un puntero también puede asociarse con un área de memoria alojada dinámicamente a través de la sentencia `allocate`. Esta sentencia crea (sin nombre) un área de memoria del tamaño, tipo y rango especificados y con un atributo target:

```
real, pointer :: p
real, dimension (:, :), pointer :: pv
integer :: m, n
...
allocate (p, pv(m, n))
```

Aquí el puntero p apunta a un área de memoria dinámica con capacidad para almacenar un real. El puntero pv apunta a un vector (matriz) de tamaño m x n La memoria dinámica puede liberarse utilizando:

```
deallocate(pv)
```

y haciendo que el estado de pv pase a ser `null`.

```
...
real, pointer :: p1, p2
allocate (p1)
p1 = 3.4
p2 => p1
...
deallocate (p1)     ! A partir de este punto, las referencias a p2
...                 ! ahora ásern errores y los resultados son
                    ! impredecibles...
```

Figura 1.43: Referencias suspendidas

La programación con punteros entraña algunos peligros si no se diseña con precaución. El código de la figura 1.43 muestra un ejemplo de referencias suspendidas.

Consideremos el siguiente código:

```
...
real, dimension(:), pointer :: p
allocate (p(1000))
```

Si el puntero p es anulado (`nullify`), se hace apuntar a otra zona de memoria o este código está en un subprograma y el subprograma es abandonado (p no tiene atributo `save`) sin liberar la memoria, no habrá forma de referenciar ese bloque de memoria y por tanto no podrá ser liberado.

Solución: liberar cualquier bloque de memoria antes de modificar un puntero que apunte a ella.

6.6. Argumentos puntero

Los punteros, asociados o no, se pueden pasar como parámetros a los subprogramas pero sólo si se dan las siguientes condiciones:

1. Si un procedimiento tiene un puntero o destino como parámetro formal, la interface al procedimiento ha de ser explícita
2. Si un parámetro formal es un puntero, el parámetro actual ha de ser un puntero con el mismo tipo, parámetros de tipo y rango
3. Un parámetro formal de tipo puntero no puede tener atributo `intent`

Si el parámetro actual es un puntero pero el formal no lo es, el parámetro formal se asocia con el destino del puntero.

```
...                ! unidad de prog. que invoca a sub1 y sub2
interface          ! interface para sub2
  subroutine sub2(b)
    real, dimension(:, :), pointer :: b
  end subroutine sub2
end interface
real, dimension(:, :), pointer :: p
  ...
allocate (p(50, 50))
call sub1(p)
call sub2(p)
...
subroutine sub1(a)             ! a no es un puntero sino
real, dimension(:, :) :: a     ! un vector de forma asumida
...
end subroutine sub1
subroutine sub2(b)             ! b es un puntero
real, dimension(:, :), pointer :: b
...
deallocate(b)
...
end subroutine sub2
```

Figura 1.44: Paso de parámetros de tipo puntero

Hemos de tener en cuenta varios aspectos en relación con el código de la figura 1.44. Tanto sub1() como sub2() son procedimientos externos. Puesto que sub2() tiene un parámetro formal puntero, se necesita un bloque `interface` en la unidad llamadora (no es necesario para sub1()). Una alternativa hubiera sido usar un módulo o un procedimiento interno para suministrar una interface explícita por defecto.

La unidad de programa llamadora hace que el puntero p sea un alias de un vector de 50x50 reales e invoca a sub2(). Esto asocia el parámetro formal puntero b con el parámetro actual puntero p. Cuando sub2() libera b se libera también p en el programa llamador y hace que p pase a ser `null`.

Los vectores dinámicos no pueden usarse como parámetros formales y deben ser alojados y liberados en la misma unidad de programa. Se permite pasar como parámetros actuales vectores dinámicos con memoria asignada, pero no sin que les haya sido asignada memoria.

6.7. Funciones que devuelven punteros

El resultado de una función puede tener el atributo `pointer`, lo cual es útil si el tamaño del resultado depende de cálculos realizados en la función.

```
...
integer, dimension(100) :: x
integer, dimension(:), pointer :: p
...
p => mayor\_cero(x)
...
```

```
  function mayor_cero(a)
    integer, dimension(:), pointer :: mayor_cero
    integer, dimension(:) :: a
    integer :: n
    ...                    ! n = no. de valores > 0
    if (n == 0)
      nullify(mayor_cero)
    else
      allocate (mayor_cero(n))
    endif
    ...                    ! poner los valores en mayor_cero
  end function mayor_cero
...
```

Figura 1.45: Una función que devuelve un puntero

En el código de la figura 1.45 la función mayor_cero() devuelve todos los valores mayores que cero de un vector. La función mayor_cero() se ha codificado como un procedimiento interno porque la interface a una función que devuelve un puntero ha de ser explícita. El resultado de la función puede utilizarse como una expresión en una sentencia de asignación de punteros (pero ha de asociarse antes con un destino -`target`- definido). Como resultado el puntero p apunta a un vector dinámico de enteros del tamaño correcto que contiene todos los valores positivos del vector x.

6.8. Vectores de punteros

Un vector de punteros no se puede declarar directamente:

```
real, dimension(20), pointer :: p       ! error
```

No puede hacerse porque `pointer` es un atributo y no un tipo de datos. Un puntero puede ser una componente de un tipo derivado. Un vector de punteros se declara a través de un tipo derivado:

```
type puntero_real
real, dimension(:), pointer :: p
end type puntero_real
```

Ahora se puede definir un vector de variables de este tipo:

```
type(puntero_real), dimension(100) :: vec_pun
```

y se puede hacer referencia al i-ésimo puntero:

```
vec_pun(i)%p
```

```
integer, parameter :: n = 10
type(puntero_real), dimension(n) :: a
integer :: i

do i = 1, n
  allocate (a(i)%p(i))      ! referencia al
end do                      ! i-esimo puntero
```

Figura 1.46: Una matriz triangular inferior

En el código de la figura1.46 cada columna de una matriz triangular inferior se representa mediante un vector dinámico de tamaño creciente. Nótese que a(i)%p apunta a un vector dinámico de reales de tamaño i y por lo tanto esta representación utiliza sólo la mitad de la memoria que se necesitaría si se utiliza un vector bidimensional convencional.

6.9. Listas enlazadas

Una componente de tipo puntero de un tipo derivado puede apuntar a un objeto del mismo tipo, lo cual permite crear listas enlazadas y otras estructuras de datos dinámicas:

```
type nodo
  integer :: valor                    ! datos
  type (node), pointer :: siguiente   ! puntero
end type nodo
```

En una estructura de este tipo los nodos

- Pueden crearse dinámicamente (en tiempo de ejecución)
- Pueden insertarse en cualquier posición de la lista
- Pueden eliminarse dinámicamente

Por todo ello, el tamaño de estas estructuras puede crecer, casi arbitrariamente, conforme se ejecuta el programa, con la única limitación de la capacidad de memoria del sistema.

BIBLIOGRAFÍA

Adams, J. C. et. al. (1992) Fortran 90 Handbook. McGraw-Hill. ISBN 0-07-000406-4

Brainerd, W. S. et. al., (1994) Programmer's Guide to Fortran 90. 2nd edition , Unicomp. ISBN 0-07-000248-7

Counihan, M. (1991) Fortran 90. Pitman. ISBN 0-273-03073-6

Hahn, B. D. (1994) Fortran 90 for Scientists and Engineers. Edward Arnold. ISBN: 0-340-60034-9

Kerrigan, J. (1993) Migrating to Fortran 90. O'Reilly and Associates. ISBN 1-56592-049-X

Metcalf, M. & Reid, J. (1992) Fortran 90 Explained. Oxford University Press. ISBN: 0-19-850558-2

Morgan, J. S. & Schonfelder, J. L. (1993) Programming in Fortran 90. Alfred Waller Ltd. ISBN 1-872474-06-3

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

Cursos disponibles a través de internet: http://www.man.ac.uk/hpctec/courses/Fortran90/JISCF90page.html

Michel Olagnon's Fortran 90 List http://www.ifremer.fr/ditigo/molagnon/fortran90/engfaq.html

The F Programming Language Homepage http://www.fortran.com/F/

The Fortran Company http://www.fortran.com/

Free Fortran Software http://www.fortran.com/free.html