
5

Punteros



Contenidos del Curso

- 1.- Introducción
- 2.- Código fuente, tipos y estructuras de control
- 3.- Procedimientos y Módulos
- 4.- Proceso de vectores
- 5.- Punteros**
- 6.- Nuevas características de entrada/salida
- 7.- Procedimientos intrínsecos
- 8.- Características redundantes
- 9.- Desarrollos futuros



Punteros

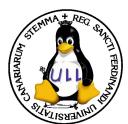
Índice

- 5.1 - Significado de un puntero
- 5.2 - Especificaciones
- 5.3 - Asignaciones de punteros
- 5.4 - Estado de una asociación de punteros
- 5.5 - Memoria dinámica
- 5.6 - Argumentos puntero
- 5.7 - Funciones que devuelven punteros
- 5.8 - Vectores de punteros
- 5.9 - Listas enlazadas
- 5.10 - Ejercicios



Significado de un puntero

- Una variable de tipo puntero (o simplemente un puntero) tiene el atributo pointer y puede apuntar a otra variable de un tipo adecuado, que tendrá el atributo target o bien a un área de memoria alojada dinámicamente
- Los punteros en F90 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 verse 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



Especificaciones

- La forma general de declarar una variable puntero y un destino (target) son:

```
type  [ [,attribute] ... ]  pointer  ::  list  of
      pointer variables
type  [ [,attribute] ... ]  target   ::  list  of
      target variables
```

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



Especificaciones

- 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 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



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:
`pointer => target`
- Donde pointer es una variable con atributo pointer y target 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



Asignaciones de punteros

- Ejemplos:

```
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
```



Asignaciones de punteros



```
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
```

- Téngase en cuenta que la asignación:

p2 => p1 + 4.3 !Error

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



Asignación de punteros / asignaciones normales

- Comparemos lo anterior con el siguiente código, en el que sólo la última línea es diferente:

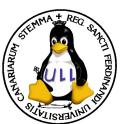
```
real, pointer :: p1, p2
real, target :: t1 = 3.4, t2 = 4.5
p1 => t1                      ! p1 apunta a t1
p2 => t2                      ! p2 apunta a t2
p2 = p1                        ! Asignación normal,
                                ! equivalente a t2 = t1
```



- Después de que la última asignación se ejecuta, la situación es:



- La sentencia tiene el mismo efecto que $t2=t1$ puesto que $p1$ es un alias de $t1$ y $p2$ lo es de $t2$



Punteros a vectores

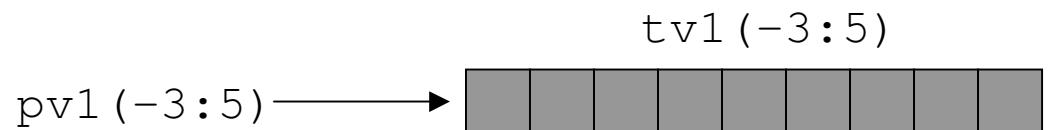
- El destino de un puntero puede ser también un vector. Veamos ejemplos de su uso:

```
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
                                  ! Subíndices de sección
pv1 => tv2(4, :)                 ! pv1 apunta a la 4a fila
                                  ! de tv2
pv2 => tv2(2:4, 4:8)             ! pv2 apunta a una
                                  ! sección de tv2
pv1 => tv1(1:5:2)                ! pv1 apunta a una
                                  ! Sección de tv1
pv1 => tv1(v)                   ! Error
```

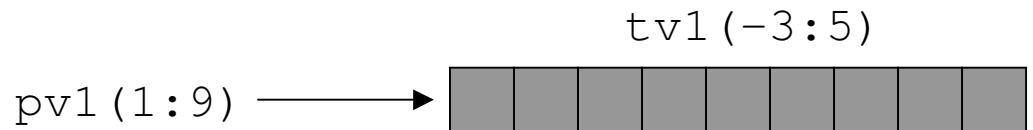


Punteros a vectores

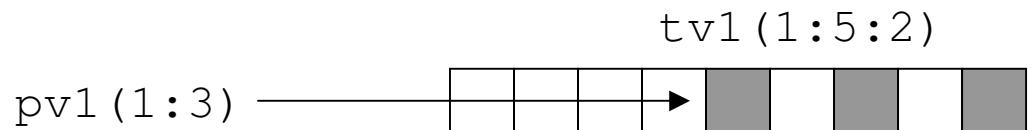
```
real, dimension (:), pointer :: pv1  
real, dimension (-3:5), target :: tv1  
pv1 => tv1           ! pv1 es un alias de tv1
```



```
pv1 => tv1(:)           ! pv1 apunta a tv1 con  
                           ! Subíndices de sección
```

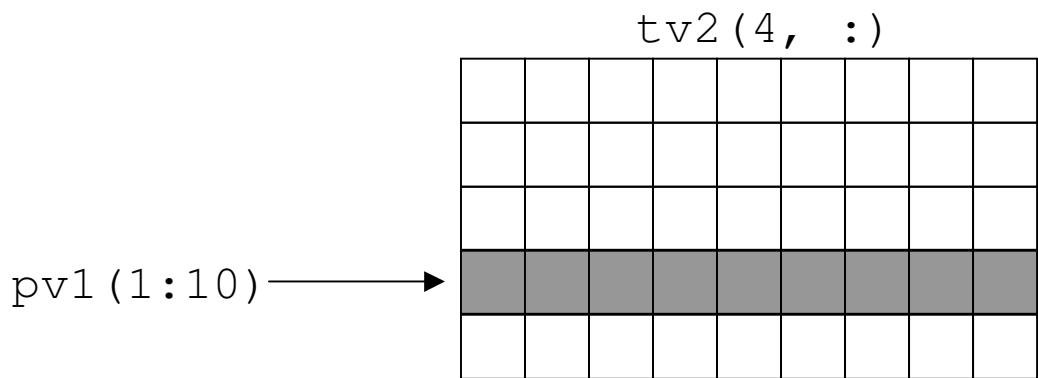


```
pv1 => tv1(1:5:2)       ! pv1 apunta a una  
                           ! Sección de tv1
```

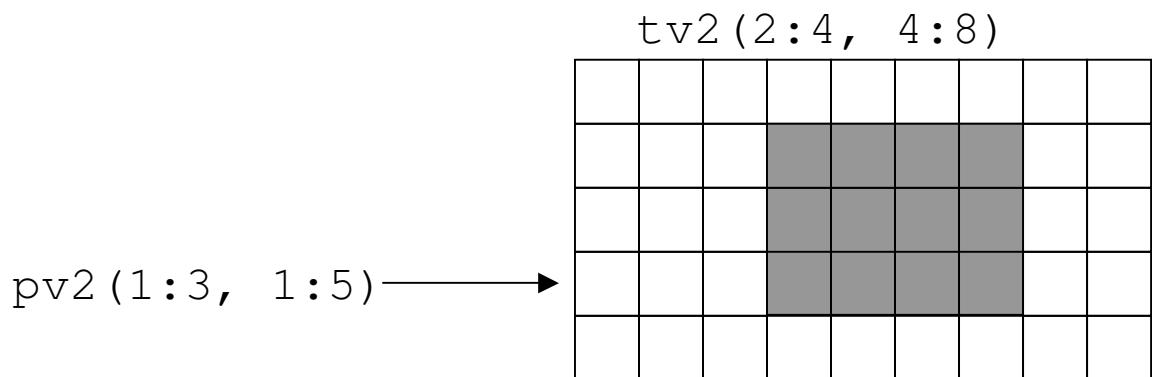


Punteros a vectores

```
real, dimension (:), pointer :: pv1  
real, dimension (:, :), pointer :: pv2  
real, dimension (5, 10), target :: tv2  
pv1 => tv2(4, :)      ! pv1 apunta a la 4a fila  
                      ! de tv2
```



```
pv2 => tv2(2:4, 4:8) ! pv2 apunta a una  
                      ! sección de tv2
```



Punteros a vectores

- Hay varios aspectos a 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 renombra 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)`



Punteros a vectores

- 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



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
 - 2.- Null (disociado) - cuando se hace nulo con una sentencia NULLIFY
 - 3.- Asociado - cuando apunta a un destino
- Un puntero puede disociarse explícitamente de su destino (target) usando una sentencia NULLIFY:
`NULLIFY(list of pointers)`
- La función intrínseca associated puede usarse para comprobar el estado de asociación de un puntero usando 1 ó 2 parámetros:
`associated(p, [,t])`
- Si el 2º parámetro no aparece, retorna .TRUE. si el puntero p está asociado con algún destino y .FALSE. en caso contrario. El 2º parámetro puede ser un puntero en cuyo caso retorna .TRUE. si ambos punteros están asociados al mismo destino o disociados y .FALSE. en caso contrario



Estado de una asociación de punteros

- 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 con NULLIFY

Ejemplos:

```
real, pointer :: p, q      ! Indefinidos
real, target :: t = 3.4
p => t                      ! p apunta a t
q => t                      ! q también 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)
```

- 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

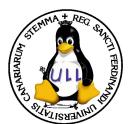


Memoria dinámica

- También un puntero 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 $m \times n$
- La memoria dinámica puede liberarse:
`deallocate (pv)`
haciendo que el estado de `pv` pase a ser `NULL`

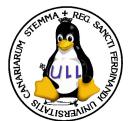


Memoria dinámica

- La forma general de estas sentencias es:

```
allocate(pointer[ (dim. specification) ] ...  
[, STAT = status])  
deallocate(pointer... [, STAT = status])
```

- Donde `pointer` es una variable puntero y la especificación de dimensión especifica la extensión de cada dimensión si el puntero tiene los atributos `pointer` y `dimension` (es un puntero a un vector).
`status` es un entero que tomará el valor 0 si la demanda/liberación de memoria ocurrió correctamente. Ambas sentencias pueden solicitar/liberar memoria para varios punteros
- La programación con punteros entraña algunos peligros si no se diseña con precaución:



Memoria dinámica

- Referencias suspendidas:

```
...
real, pointer :: p1, p2
allocate (p1)
p1 = 3.4
p2 => p1

...
deallocate (p1) ! Las referencias a p2
...               ! ahora serán errores
                 ! y los resultados son
                 ! impredecibles...
```



Memoria dinámica

- Memoria inaccesible:

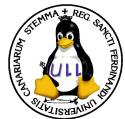
```
...
real, dimension(:), pointer :: p
allocate (p(1000))
...
...
```

- 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



Argumentos puntero

- Los punteros, asociados o no, se pueden pasar como parámetros a los subprogramas pero sólo si se dan las siguientes condiciones:
 - Si un procedimiento tiene un puntero o destino como parámetro formal, la interface al procedimiento ha de ser explícita
 - Si un parámetro formal es un puntero, el parámetro actual ha de ser un puntero con el mismo tipo, parámetros de tipo y rango
 - 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
- Consideremos el siguiente código:



Argumentos puntero

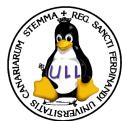
```
...! Unidad de prog. Que invoca a sub1 y sub2
interface      ! interface para sub2
    subroutine sub2(b)
        real, dimension(:, :, ), pointer :: b
    end subroutine sub2
end interface
real, dimension(:, :, ), pointer :: p
...
allocate (p(50, 50))
call sub1(p)
call sub2(p)
...
subroutine sub1(a) !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
```



Argumentos puntero

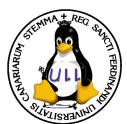
- Los aspectos a tener en cuenta aquí son:
 - Tanto `sub1` como `sub2` son procedimientos externos. Puesto que `sub2` tiene un parámetro formal puntero, se necesita un bloque interface en la unidad llamadora (no es necesario para `sub1`). Una alternativa hubiera sido usar un módulo o un procedimiento 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 50×50 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



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. Veamos un ejemplo: Una función que devuelve todos los valores > 0 de un vector:

```
...
integer, dimension(100) :: x
integer, dimension(:), pointer :: p
...
p => gtzero(x)
...
contains
function gtzero(a)
    integer, dimension(:), pointer :: gtzero
    integer, dimension(:) :: a
    integer :: n
    ...
    ! n = nº de valores > 0
    if (n == 0)
        NULLIFY(gtzero)
    else
        allocate (gtzero(n))
    endif
    ...
    ! Poner los valores en gtzero
end function gtzero
...
.
```

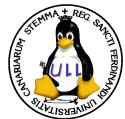


Funciones que devuelven punteros

Consideraciones:

- La función `gtzero` 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`



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

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

```
type(real_pointer), dimension(100) :: a
```

- Y se puede hacer referencia al i-ésimo puntero:
 $a(i)\%p$



Vectores de punteros

- Un ejemplo en el que cada columna de una matriz triangular inferior se representa mediante un vector dinámico de tamaño creciente:

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

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

- 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 con un vector bidimensional convencional



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:

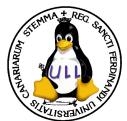
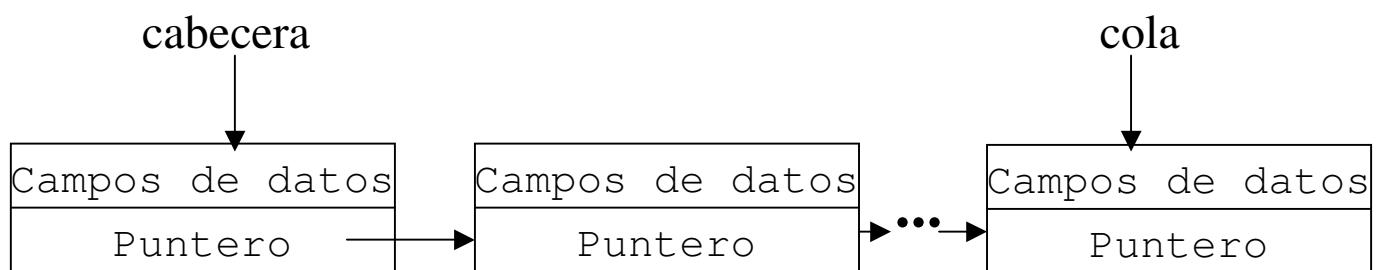
```
type node
    integer :: value          ! Datos
    type (node), pointer :: next ! Puntero
end type node
```

- En una estructura de este tipo los nodos
 - No tienen porqué almacenarse de forma contigua en memoria
 - 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



Listas enlazadas

- Una lista enlazada consiste habitualmente en elementos de un tipo derivado que contiene uno o varios campos de datos más un campo que es un puntero al siguiente elemento del mismo tipo en la lista



Listas enlazadas

```
program simple_linked_list
implicit none
type node
    integer :: value          ! Campo de datos
    type (node), pointer :: next ! Campo puntero
end type node
integer :: num, status
type (node), pointer :: list, current
! build up the list
NULLIFY(list) !Inicialmente, vaciar la lista
do
    read *, num           !Leer num del teclado
    if (num == 0) exit !Hasta que se lea un 0
    allocate(current, STAT = status) !Crear nodo
    if (status > 0) stop 'Fallo al crear nodo'
    current%value = num
    current%next => list      !Apuntar al anterior
    list => current           !Actualizar head
end do
! Recorrer la lista imprimiendo los valores
current => list ! Current es un alias de list
do
    if (.not. associated(current)) exit
    print *, current%value
    current => current%next
end do
end program simple_linked_list
```



Listas enlazadas

!Recorrer la lista eliminando nodos:

```
current => list
do
    if (.not. associated(current)) exit
    list => current%next
    deallocate(current)
    current => list
end do
```

