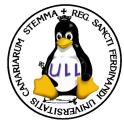

3

Procedimientos

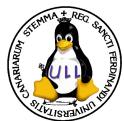
y

Módulos



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



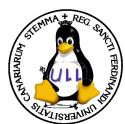
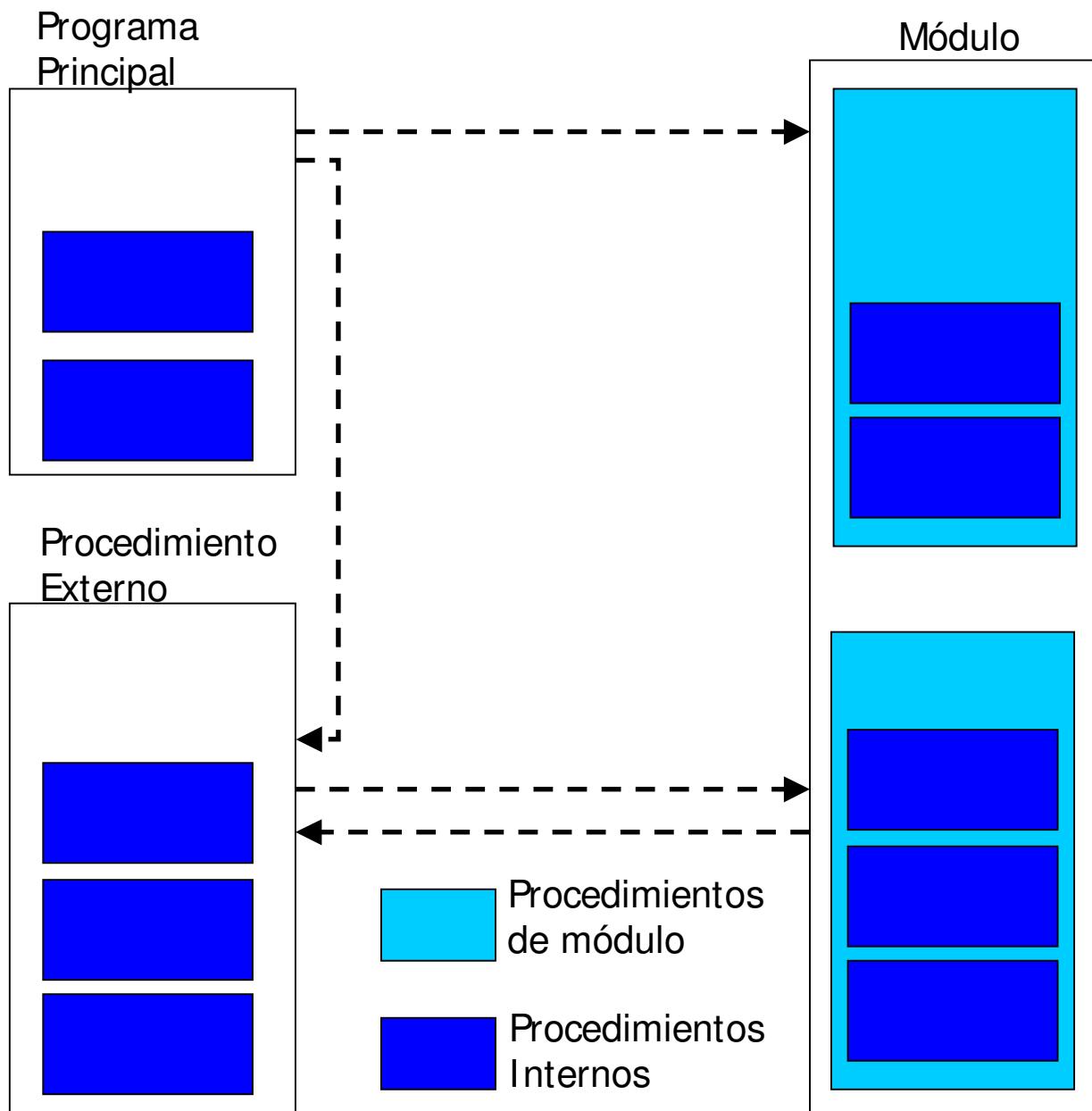
Procedimientos y Módulos

Índice

- 3.1 Unidades de programa
- 3.2 Procedimientos
- 3.3 bloques interface
- 3.4 Procedimientos internos
- 3.5 Argumentos procedurales
- 3.6 Cláusula result para funciones
- 3.7 Funciones que devuelven vectores
- 3.8 Procedimientos recursivos
- 3.9 Procedimientos genéricos
- 3.10 Módulos
- 3.11 Sobrecarga de operadores
- 3.12 Definición de operadores
- 3.13 Sobrecarga del operador de asignación
- 3.14 Ámbito
- 3.15 Estructura de los programas
- 3.16 Ejercicios



Unidades de Programa



Unidades de programa

- Un programa sólo puede tener un único programa principal y todas las unidades de programa que se deseé (módulos o procedimientos externos)
- Un módulo se utiliza para que diferentes programas compartan código (datos y subrutinas)



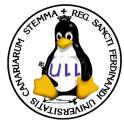
Programa Principal

- Forma:

```
program nombre_de_programa
  [sentencias_de_especificación]
  [sentencias_ejecutables]
  ...
end [program [nombre_de_programa]]
```

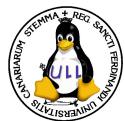
- Ejemplo:

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



Procedimientos

- Llamamos procedimientos de forma genérica a Funciones y Subrutinas
- 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:
 - Externos (autocontenido, no necesariamente Fortran)
 - Internos (dentro de una unidad de programa)
 - de Módulo (miembros de un módulo)
- Fortran 77 sólo tiene procedimientos externos
- Un bloque interface se utiliza para definir los detalles de los parámetros de un procedimiento. Es obligatorio para los procedimientos externos



Procedimientos Externos

- Forma:

```
subroutine nombre (parámetros_formales)
  [sentencias_de_especificación]
  [sentencias_ejecutables]
  ...
end [subroutine [nombre]]
```

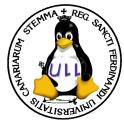
- bien:

```
function nombre (dummy-argument-list)
  [sentencias_de_especificación]
  [sentencias_ejecutables]
  ...
end [function [nombre]]
```



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/function** ha de aparecer en la sentencia **end**



Procedimientos Internos

- 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



Procedimientos Internos

- Ejemplo:

```
program principal
    implicit none
    real :: a, b, c
    real :: suma_ppal
    ...
    suma_ppal = suma()
    ...
contains
    function suma()
        real :: suma      ! a, b y c ya definidas
        suma = a + b + c
    end function suma
    ...
end program principal
```

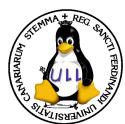
Tiene efecto sobre
todos los proc. internos
que contenga



Procedimientos internos

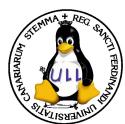
- **implicit none** en una unidad de programa también tiene efecto sobre todos los procedimientos internos que contenga. No obstante, se recomienda volver a usarlo, tanto por claridad como para evitar errores:

```
subroutine aritmetica(n, x, y, z)
    implicit none
    integer :: n
    real, dimension(100) :: x, y, z
    ...
contains
    function suma(a, b, c) result(sum)
        implicit none
        real, intent(in) :: a, b, c
        real :: sum
        sum = a + b + c
    end function suma
end subroutine aritmetica
```



Bloques interface

- Si se suministra explícitamente una interface para un procedimiento, el compilador puede comprobar inconsistencias en los parámetros
- 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



Bloques interface

- Forma general:

```
interface  
    cuerpo_de_la_interface  
    ...  
end interface           !Ojo: no se pone nombre
```

- Donde *cuerpo_de_la_interface* es una copia exacta de la especificación del subprograma, su especificación de argumentos y su sentencia end
- Ejemplo:

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

- El bloque interface se coloca en la unidad de programa que realiza la llamada



Argumentos - Intent

Sirve para especificar si un parámetro de un procedimiento es para:

- Entrada (in) (No puede ser modificado)
- Salida (out)
- o ambos (inout)

○ Ejemplos

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

```
subroutine interc_real(a, b)
implicit none
real, intent(inout) :: a, b
real :: temp
temp = a
a = b
b = temp
end subroutine interc_real
! llamada: call interc_real(x, y)
```



Parámetros con nombre

- Se utiliza para evitar confusión cuando un procedimiento tiene varios parámetros. Evita el tener que recordar el orden de los parámetros.
- Ejemplo: Si se tiene la función

```
real function area(inicio, fin, error)
    implicit none
    real, intent(in) :: inicio, fin, error
    ...
end function area
```

- La llamada podría hacerse:

```
a = area(0.0, 100.0, 0.01)
b = area(inicio=0.0, error=0.01, fin=100.0)
c = area(0.0, error=0.01, fin=100.0)
```

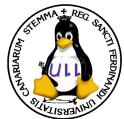


Parámetros con nombre

- Una vez que se usa un nombre para uno de los parámetros, el resto de parámetros han de usarlo también.
Así no es posible:

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

- Aquí no se necesita una interface porque **area** es una función interna. Sí haría falta si se tratara de un procedimiento externo con parámetros



Argumentosopcionales

- En ciertas ocasiones, no todos los parámetros necesitan estar presentes en una llamada. Un parámetro se puede declarar como opcional:

```
real function area(inicio, fin, error)
    implicit none
    real, intent(in), optional :: inicio, &
                                fin, error
    ...
end function area
```

- La llamada podría realizarse:

```
a=area(0.0, 100.0, 0.010)
b=area(inicio=0.0, fin=100.0, error=0.01)
c=area(0.0)
d=area(0.0, error=0.01)
```

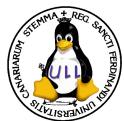


Parámetros opcionales

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

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

- No se podría modificar `error`, porque es `intent(in)`. Por ello se usa una variable local, `ttol`



Parámetros opcionales

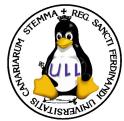
- Si el procedimiento es externo y tiene parámetrosopcionales, se ha de suministrar una interface. Si la función del ejemplo anterior fuera externa sería necesario el siguiente bloque interface:

```
interface
    real function area(inicio, fin, error)
        real, intent(in), optional :: inicio,&
                                    fin, error
    end function area
end interface
```



Tipos derivados como Argumentos

- Los parámetros de un procedimiento pueden ser de un tipo derivado si:
 - 1.- El procedimiento es interno a la unidad de programa en la que se define el tipo derivado
 - 2.- el tipo derivado se define en un módulo que es accesible desde el procedimiento



Parámetros procedurales

- En Fortran77 un parámetro procedural se ha de declarar como external
- En Fortran90 un procedimiento que es pasado como parámetro 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 un bloque interface en la unidad de programa que realiza la llamada
- Un procedimiento de módulo tiene una interface explícita por defecto



Argumentos procedurales

- **Ejemplo:**

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, inicio, fin, error)
```

La función externa:

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



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.

Ejemplo:

```
function suma(a, b, c) result(sum_abc)
    implicit none
    real, intent(in) :: a, b, c
    real :: sum_abc
    sum_abc = a + b + c
end function suma
```

- Las funciones directamente recursivas (la recursividad se estudia más adelante) han de tener una variable result



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 suma_vec (a, b, n)
implicit none
real, dimension (n) :: suma_vec
integer, intent(in) :: n
real,dimension (n), intent(in) :: a, b

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

- Si la función fuera externa, habría que especificar una interface en el programa llamador



Procedimientos Recursivos

- Los procedimientos se pueden invocar recursivamente:
 - P1 invoca a P2 y P2 invoca a P1 o bien
 - P1 invoca directamente a P1 (se requiere result)
- Los procedimientos recursivos han de declararse como tales:

```
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
```

- ¡La recursividad es Ineficiente!



Procedimientos Genéricos

- Cuando se utilizan proc. genéricos, se usa un mismo identificador para el procedimiento y el código que se ejecuta depende del tipo de los parámetros (como 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 generic_name
    cuerpo_especifico_de_la_interface
    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



Procedimientos Genéricos

- **Ejemplo:**

```
subroutine interc_real
    implicit none
    real, intent(inout) :: a, b
    real :: temp
    temp = a
    a = b
    b = temp
end subroutine interc_real
```

```
subroutine interc_int
    implicit none
    integer, intent(inout) :: a, b
    integer :: temp
    temp = a
    a = b
    b = temp
end subroutine interc_int
```



Procedimientos Genéricos

- Interface genérica:

```
interface intercambia
```

```
    subroutine interc_real (a, b)
        real, intent(inout) :: a, b
    end subroutine interc_real
```

```
    subroutine interc_int (a, b)
        integer, intent(inout) :: a, b
    end subroutine interc_int
```

```
end interface
```

- Llamada:

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

- Vea un ejemplo en

```
ej2_modules/ejemplos/
generic_intercambia.f90
```



Módulos

- Se trata de una característica del lenguaje que incide (positivamente) sobre la estructura de los programas
- Constituyen una forma de compartir datos y/o procedimientos entre diferentes unidades de programa
- Los módulos juegan un papel importante en la definición de tipos y operadores asociados
- La forma general es:

```
module nombre
    [sentencias_de_especificación]
    [sentencias_ejecutables]
[contains
    procedimientos_de_módulo]
end [module [nombre]]
```

- Al módulo se accede a través de la sentencia **use**



Módulos: Datos globales

- En Fortran las variables son habitualmente entidades locales. Utilizando módulos es posible hacer accesibles un conjunto de datos a diferentes unidades de programa:

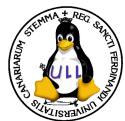
```
module globales  
  
    real, save :: a, b, c  
    integer, save :: i, j, k  
  
end module globales
```

- 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**



Módulos: Datos globales

- 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)
- **Ejemplos de uso de la sentencia use:**

```
use globales
```

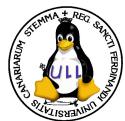
```
! Permite acceder a todas las variables  
! del módulo
```

```
use globales, only : a, c
```

```
! Permite acceder sólo a las variables a  
! y c
```

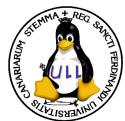
```
use globales, r=>a, s=>b
```

```
! Permite acceder a las variables a y b  
! a través de los identificadores r y s
```



Módulos: Procedimientos de módulo

- Los procedimientos que se especifican dentro de un módulo se llaman procedimientos de módulo
- Son procedimientos que pueden ser accedidos (utilizados) por otras unidades de programa
- Puede haber varios procedimientos de módulo contenidos en el mismo módulo
- Han de estar codificados en Fortran (los externos pueden estar en otro lenguaje)
- 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
- 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



Módulos: Procedimientos de módulo

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

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

contains

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

end module modulo_puntos
```



Módulos: Procedimientos de módulo

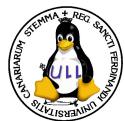
- El programa principal podría declarar:

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



Módulos: Procedimientos genéricos

- La utilización de módulos permite parámetros de tipos derivados y por tanto procedimientos genéricos con tipos derivados
- Veamos como ejemplo, una extensión del procedimiento intercambia genérico que hemos desarrollado, para intercambiar puntos:



Módulos

```
module intercambio_generico
    implicit none

    type puntos
        real :: x, y
    end type puntos

    interface intercambia
        module procedure interc_real, &
                    interc_int, interc_log, interc_puntos
    end interface

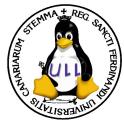
contains
    subroutine interc_puntos (a,b)
        implicit none
        type (puntos), intent(inout) :: a, b
        type (puntos) :: temp
        temp = a
        a = b
        b = temp
    end subroutine interc_puntos
```



Módulos

```
subroutine interc_real
    implicit none
    real, intent(inout) :: a,b
    real :: temp
    temp = a
    a = b
    b = temp
end subroutine interc_real

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



Módulos: Private y Public

- 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 :: fil, col
```

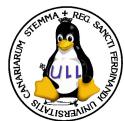


Sobrecarga de operadores

- 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 de operadores. Para conseguirlo se necesita un bloque interface de la forma:

```
interface operator (op_intrinseco)
    cuerpo_de_la_interface
end interface
```

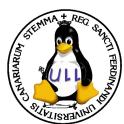
- Por ejemplo, el operador ‘+’ podría extenderse a variables de tipo carácter para concatenar dos cadenas de caracteres ignorando los espacios sobrantes, y el código podría colocarse en un módulo:



Sobrecarga de operadores

Ejemplo:

```
module sobrecarga_op
    implicit none
    ...
    interface operator (+)
        module procedure concat
    end interface
    ...
contains
    function concat(cha, chb)
        implicit none
        character(len=*), intent(in) :: cha, chb
        character(len=(len_trim(cha) +&
                        len_trim(chb))) :: concat
        concat = trim(cha)//trim(chb)
    end function concat
    ...
end module sobrecarga_op
```



Sobrecarga de operadores

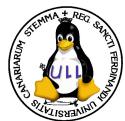
- Ahora la expresión '**cha + chb**' tiene significado en cualquier programa que '**use**' este módulo
- Obsérvese en el ejemplo el bloque interface. La función que define el operador está en un módulo y no es necesario tener interfaces explícitas para procedimientos de módulo que están dentro del mismo módulo
- 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 ...  
    lista_de_procedimientos_de_modulo  
end interface
```



Definición de operadores

- Se pueden definir nuevos operadores
- Resulta especialmente útil para tipos definidos por el usuario
- Los operadores definidos han de tener ‘.’ (punto) al principio y al final. Por ejemplo, en el ejemplo anterior podríamos haber definido `.mas.` en lugar de sobrecargar `+`
- La operación se ha de definir a través de una función que tenga uno o dos parámetros no opcionales con atributo intent(in) (que sean de entrada)
- Veamos un **Ejemplo**: calcular la distancia euclídea entre dos elementos del tipo derivado punto:



Definición de operadores: distanzia_principal.f90

```
program principal
implicit none
use distanza_mod
type(puntos) :: px, py
...
distanzia = px .dist. py
...
end program principal
```



Definición de operadores: distancia_mod.f90

```
module distancia_mod
    implicit none
    ...
    type puntos
        real :: x, y
    end type puntos
    ...
    interface operator (.dist.)
        module procedure calcdist
    end interface
    ...
contains
    ...
    function calcdist (px, py)
        implicit none
        real :: calcdist
        type (puntos), intent(in) :: px, py
        calcdist = &
                    sqrt ((px%x-py%x)**2 + (px%y-py%y)**2 )
    end function calcdist
    ...
end module distancia_mod
```



Módulos mon_mod.f90

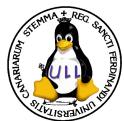
- Veamos otro ejemplo que define en un módulo un tipo derivado y todas las operaciones asociadas:

```
module dineros
    implicit none

    type pasta
        integer :: euros, cents
    end type pasta

    interface operator (+)
        module procedure sumar_pasta
    end interface

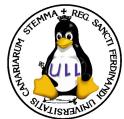
    interface operator (-)
        module procedure negar_pasta,
        resta_pasta
    end interface
```



Módulos mon_mod.f90

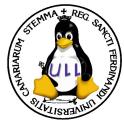
contains

```
function sumar_pasta(a, b)
    implicit none
    type (pasta) :: sumar_pasta
    type (pasta), intent(in) :: a ,b
    integer :: acarreo, tmp_cents
    tmp_cents = a%cents + b%cents
    acarreo = 0
    if (tmp_cents > 100) then
        tmp_cents = tmp_cents - 100
        acarreo = 1
    end if
    sumar_pasta%euros=a%euros + &
                           b%euros+acarreo
    sumar_pasta%cents = tmp_cents
end function sumar_pasta
```



Módulos mon_mod.f90

```
function negar_pasta(a)
    implicit none
    type (pasta) :: negar_pasta
    type (pasta), intent(in) :: a
    negar_pasta%euros = -a%euros
    negar_pasta%cents = -a%cents
end function negar_pasta
```

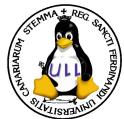


Módulos mon_mod.f90

```
function resta_pasta(a, b)
    implicit none
    type (pasta) :: resta_pasta
    type (pasta), intent(in) :: a, b
    integer :: tmp_euros, tmp_cents, acarreo
    tmp_cents = a%cents - b%cents
    tmp_euros = a%euros - b%euros

! Para incorporar acarreos necesarios
    if ((tmp_cents < 0).and.(tmp_euros > 0)) then
        tmp_cents = 100 + tmp_cents
        tmp_euros = tmp_euros - 1
    else if ((tmp_cents>0).and.(tmp_euros<0)) then
        tmp_cents = tmp_cents - 100
        tmp_euros = tmp_euros + 1
    end if
    resta_pasta%cents = tmp_cents
    resta_pasta%euros = tmp_euros
end function resta_pasta

end module dineros
```



Sobrecarga del operador de asignación

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

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

- Siguiendo con el ejemplo, supongamos que queremos que `ax` tome el valor máximo de las componentes de `px`. Esta asignación se ha de hacer a través de una subrutina con 2 parámetros no opcionales, el primero de ellos `intent(out)` o `intent(inout)` y el segundo `intent(in)`.

Debe crearse un bloque interface de asignación:

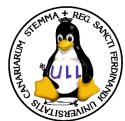
```
interface assignment (=)
  cuerpo_de_la_subrutina_interface
end interface
```



Sobrecarga del operador de asignación

- La definición de la asignación se podría colocar en un módulo como:

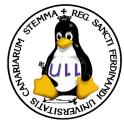
```
module mod_sobrecarga_assign
    implicit none
    type puntos
        real :: x, y
    end type puntos
    ...
    interface assignment (=)
        module procedure assign_puntos
    end interface
    contains
        subroutine assign_puntos (ax, px)
            real, intent(out) :: ax
       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



Sobrecarga del operador de asignación

- El programa principal necesita invocar a este módulo con una sentencia **use**:

```
use mod_sobrecarga_assign
real :: ax
type (puntos) :: px
...
ax = px           ! Correcto
```



Ámbito

- El ámbito de un identificador (de variable, función, subrutina) o etiqueta es el conjunto de entidades no-solapadas donde el identificador o etiqueta puede utilizarse sin ambigüedad
- Estas entidades (unidades de ámbito) son:
 - Una definición de tipo (derivado)
 - Un cuerpo de interface de procedimiento, excluyendo las definiciones de tipos y cuerpos de interface definidos dentro de él
 - Una unidad de programa o procedimiento interno, excluyendo las definiciones de tipos, cuerpos de interfaces y subprogramas contenidos en él



Ámbito

Etiquetas

- Cada subprograma, interno o externo tiene su propio conjunto de etiquetas. La misma etiqueta puede usarse en un programa principal y sus procedimientos internos sin ambigüedad. Por lo tanto el ámbito de una etiqueta es un programa principal o un procedimiento excluyendo cualquier procedimiento interno que pueda contener

Identificadores

- El ámbito de un identificador declarado en una unidad de programa se extiende desde su cabecera hasta su sentencia end
- El ámbito de un identificador declarado en un programa principal o subprograma externo se extiende a todos los subprogramas que contenga a menos que el mismo identificador se redefina en ellos
- El ámbito de un identificador declarado en un subprograma interno es sólo el propio subprograma y no otros subprogramas internos.



Ámbito

Identificadores

- El ámbito del identificador de un subprograma interno, y del número y tipo de sus parámetros se extiende a través de la unidad de programa en que está contenido y por lo tanto a todos los otros subprogramas internos
- El ámbito de un identificador declarado en un módulo se extiende a todas las unidades de programa que use(n) ese módulo, a menos que el identificador tenga atributo private o que sea renombrado en el programa que usa el módulo o que la sentencia use tenga una cláusula use only
- El ámbito de un identificador declarado en un módulo se extiende a todos los subprogramas internos excepto aquellos en los que el identificador sea redefinido



Ámbito

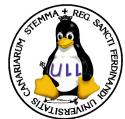
Consideremos las unidades de ámbito señaladas anteriormente

- Entidades declaradas en diferentes unidades de ámbito son siempre diferentes, aunque tengan los mismos identificadores y propiedades
- Dentro de una unidad de ámbito, cada entidad con nombre ha de tener un identificador diferente, con la excepción de los procedimientos genéricos
- Los identificadores de unidades de programa son globales, de modo que han de ser diferentes entre sí y diferentes de cualquier entidad local a las unidades de programa
- El ámbito de un identificador de un procedimiento interno se extiende sólo a toda la unidad de programa que lo contiene
- El ámbito de un identificador declarado en un procedimiento interno es ese procedimiento interno



Ámbito

- Se dice que los identificadores son accesibles por asociación de anfitrión o por asociación de uso:
- **Asociación de anfitrión:** el ámbito de un identificador declarado en una unidad de programa se extiende desde la cabecera de la unidad de programa hasta su sentencia end
- **Asociación por uso:** el ámbito de un identificador declarado en un módulo, cuando no tiene el atributo private, se extiende a cualquier unidad de programa que use el módulo



Ámbito

Ejemplo

```
module ambito1          ! ambito 1
...
contains                ! ambito 1
    subroutine ambito2  ! ambito 2
        type ambito3    ! ambito 3
            ...
        end type        ! ambito 3
        interface       ! ambito 3
            100      ...  ! ambito 4
            end interface ! ambito 3
            real x, y   ! ambito 2
            ...
        contains          ! ambito 2
            function ambito5(...)
                real y     ! ambito 5
                y = x + 1.0 ! ambito 5
            100      ...  ! ambito 5
            end function ambito5 ! ambito 5
        end subroutine ambito2 ! ambito 2
    end module ambito1    ! ambito 1
```



Estructura de los programas

Orden de las sentencias

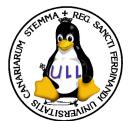
Sentencias PROGRAM , FUNCTION , SUBROUTINE o MODULE		
Sentencias USE		
	Sentencia IMPLICIT NONE	
	Sentencias PARAMETER	Sentencias IMPLICIT
Sentencias FORMAT	Sentencias PARAMETER y DATA	Definiciones de tipos derivados, Bloques interface, Sentencias de declaración de tipos, y Sentencias de especificación
	Sentencias ejecutables	
	Sentencia CONTAINS	
	Subprogramas internos o Subprogramas del módulo	
	Sentencia END	



Estructura de los programas

Cuando utilizar bloques interface

- Cuando un módulo o procedimiento externo es invocado:
 - Si el módulo define o sobrecarga un operador o sobrecarga la asignación
 - Si el módulo utiliza un identificador genérico
- Se utiliza 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 parámetro que es un vector de forma asumida o un puntero
 - Es argumento de otro subprograma (en este caso no es obligatorio, pero se recomienda)



Estructura de los programas

Resumen

- Estilo Fortran77:
 - Programa principal con procedimientos externos, posiblemente en una librería
 - No hay interfaces explícitas de modo que el compilador no comprueba inconsistencias en los parámetros de las llamadas a subprograma
- Fortran90 simple:
 - Programa principal con procedimientos internos
 - Las interfaces son explícitas, de modo que el compilador detecta posibles inconsistencias
- Fortran90 con módulos:
 - Programa principal y módulo(s) conteniendo interfaces y posiblemente especificaciones y la posibilidad de procedimientos externos (y de librerías precompiladas)
 - Una versión F90 del estilo F77 con interfaces para permitir al compilador la comprobación de inconsistencias en los parámetros
 - Programa principal y módulo(s) conteniendo especificaciones, interfaces y procedimientos



Estructura de los programas

Resumen

- Fortran90 con módulos:

- Un programa F90 estructurado consiste en un programa principal y módulos que contienen especificaciones, interfaces y procedimientos. Los procedimientos externos dejan de ser necesarios
- La introducción de nuevas características en el lenguaje tales como tipos derivados, sobrecarga, subprogramas internos y módulos hacen posible el desarrollo de código F90 sofisticado

