Elementos adicionales de Fortran

Mario Merino Martínez

12 de diciembre de 2011

Fundamentos de Fortran

A lo largo de este cuatrimestre, hemos dado a conocer los rudimentos básicos de la programación en Fortran. Sin embargo, son muchos los aspectos que, por falta de tiempo, no han podido enseñarse en las clases. Algunos de estos aspectos son temas avanzados, y por tanto está justificada su ausencia. Sin embargo, algunos otros puntos, son sencillos y de gran importancia, con un uso altamente extendido.

Es por ello que me gustaría describiros someramente algunos de los elementos que no hemos podido ver en clase. Espero con ello que al menos tengáis conocimiento de su existencia, y que incluso con lo que hay aquí escrito podáis utilizarlos sin más—de nuevo, son conceptos muy simples y básicos. Y en todo caso, espero que con lo que aquí os cuento, podáis proseguir vuestro aprendizaje de Fortran por vuestra cuenta, a medida que lo vayáis necesitando. Naturalmente, lo aquí recogido queda fuera del temario de la asignatura.

1. Modules — colecciones de datos y subprogramas

La idea básica detrás de una programación eficiente es la reutilización ordenada de código. Para ello, lo más inteligente es agrupar funciones y subrutinas (subprogramas) que realicen tareas afines (e.g. operaciones matemáticas con matrices), de manera que podamos cargarlos en nuestro programa principal de la forma más cómoda posible.

Los modules permiten crear "librerías" de subprogramas y de variables "universales" que querríamos reutilizar en varios programas. Un `module` se declara con

la estructura **module** nombre_del_module ... **end module** nombre_del_module, y constituye un archivo independiente. El module consta de 2 partes principales: una de declaración de variables globales, como pi, G en el ejemplo siguiente, y otra de declaración de subprogramas internos (después de la palabra clave **contains**). Estos subprogramas "internos" pueden hacer uso de las variables globales declaradas en la primera sección, sin declararlas ellos mismos. Como curiosidad, **contains** también puede utilizarse en otros contextos para declarar subprogramas internos.

```
module herramientas

implicit none ! Bloque de variables globales (estarán disponibles
              ! en los programas principales que carguen el module)
real, parameter :: pi = 3.1416
real, parameter :: G = 6.6738E-11

contains

function f(x)
! ...
end function
! ...
end module herramientas
```

Una vez escrito el module, podemos cargarlo en nuestro programa con la sentencia **use** nombre_del_module. Esta sentencia ha de escribirse antes de **implicit none**. Para que todo funcione, los dos archivos (programa principal y module), hay que crear un proyecto en Plato, y añadir los dos archivos al mismo (para más información sobre proyectos: los apéndices de los apuntes de la asignatura).

```
program main
use herramientas
implicit none
! ...
print*, f(12.3), pi ! podemos usar variables y funciones del module directamente
! ...
end module herramientas
```

2. Arrays con atributo "allocatable" — arrays que pueden cambiar de tamaño

En múltiples ocasiones no podemos dimensionar un array en tiempo de declaración: necesitamos conocer un dato que introducirá el usuario, o bien las dimensiones dependen de cálculos que han de ejecutarse primeramente.

A parte del "truco" aprendido en clase (sobredimensionar el array y luego sólo utilizar la parte que nos haga falta), Fortran proporciona la posibilidad de declarar los arrays con atributo `allocatable`. Esto permite decidir en tiempo de ejecución cómo de grande ha de ser el array (lo que se conoce como *asignación dinámica de memoria*), por ejemplo, después de haberle pedido el dato pertinente al usuario. Lo único que hemos de fijar en tiempo de declaración es el *rango* (o orden) del array: si queremos un vector `v(:)`, una matriz `A(:,:)`, un 3-tensor `T(:,:,:)`, un 4-tensor, `Q(:,:,:,:)`, etc.

Para asignar un tamaño a un array `allocatable`, usamos la función `allocate`. Para borrar de memoria un array "allocatado" (y poder volver a "allocatarlo" de nuevo con otras dimensiones si se desea), primero hay que usar la función `deallocate`.

La función `allocated(A)` devuelve un valor lógico `.true.` si A está allocatada, y `.false.` si no lo está.

```
program main
implicit none
real*8, allocatable :: A(:,:) ! Con esta notación indicamos que A será una ma-
triz
integer, allocatable :: v(:) ! y v un vector.
! ...
print*, "introduce la dimension n"
read(*,*) n
allocate(A(n,2*n)) ! Fija las dimensiones de A como n y 2n
allocate(v(n)) ! Fija la dimensión de v como n
! ...
deallocate(A)
deallocate(v)
! ...
if (allocated(A)) then
! ...
end program main
```

3. Asumir dimensiones en un subprograma — menos argumentos innecesarios

Cuando creamos subprogramas que trabajan con arrays, en ocasiones interesa generalizar nuestro código para que sea capaz de trabajar con arrays de cualquier dimensión. Por ejemplo, una función que calcule la inversa de una matriz: queremos que sea capaz de calcular la inversa de cualquier matriz cuadrada regular, independientemente de sus dimensiones.

Una manera de conseguir esto es pasar explícitamente las dimensiones del array como argumentos adicionales. E.g., `inversa(A,n)`, donde `n` indica las dimensiones de `A`.

Sin embargo, es posible evitar estos argumentos extra usando *dimensiones asumidas*. El subprograma aceptará lo que quiera que le entreguemos como argumento, siempre que coincida en tipo y rango. La notación para ello vuelve a ser "`:`"

```
subroutine operaciones(A)
implicit none
real*8 :: A(:,:) ! Asume las dimensiones de A, que es un array matriz
! ...
end subroutine operaciones
```

Esto sólo funciona dentro de subprogramas, y para las variables que sean argumentos. No puede usarse en el programa principal, ni para variables locales de subprogramas.

Si necesitamos conocer las dimensiones de un argumento que le llega a un subprograma, podemos usar la función **`size`**.

4. Estructuras — crea tus propios tipos de datos

¿Por qué conformarse con **`integer`**, **`real`**, **`character`**, etc.? En ocasiones intentamos describir realidades que tienen más de un dato. Por ejemplo, si estamos simulando el movimiento de varias esferas idénticas sobre un plano, cada esfera tendrá su posición x, y, y su velocidad v_x, v_y. Podemos guardar cada uno de estos datos en una variable independiente, o también podemos agruparlos en una *estructura* con varios *campos*.

Para crear estructuras (o tipos derivados), primero hay que definir cómo es y qué campos queremos que tenga. Para ello, utilizamos bloques **type**, que han de situarse justo antes del bloque de declaraciones. Después de definirla, podemos declarar variables del nuevo tipo con `type(nombre_del_tipo) :: variable`. Las declaraciones **type** también pueden agruparse convenientemente en un **module**.

Los campos son variables en sí mismas, de tipos intrínsecos (**integer**, **real**, **character**) o de otros tipos derivados ya declarados. También es posible definir campos que sean arrays.

Podemos incluso crear arrays de estructuras. Por ejemplo, un vector de 10 bolas podría ser útil para las simulaciones.

```
program main
implicit none

type bola
  real*8 :: x, y
  real*8 :: vx, vy
end type bola

type(bola) :: unabola
type(bola) :: vectorbolas(10)
! ...
end program main
```

Para acceder a un campo de una estructura, se utiliza el símbolo "%":

```
! ...
unabola%x = 4.8
unabola%y = 2.1
unabola%vx = 1.1d0
unabola%vy = 0.5
print*, "La energia cinetica de la bola con masa 1 es: ",
0.5*(unabola%vx**2 + unabola%vy**2)
print*, "Posicion x de la bola 5: ", vectorbolas(5)%x
! ...
```

También podemos asignar valores a una estructura (después de que esté declarada, claro está) en un solo paso con el *constructor de estructuras*: simplemente, se dan los valores en el orden en el que fueron declarados los campos:

```
! ...
unabola = bola(4.8, 2.1, 1.1d0, 0.5)
! ...
```

5. Punteros — variables de variables

Se trata de un tema ya más avanzado, y por tanto, simplemente lo cuento aquí para los más interesados.

Un puntero es un "link" o "alias", un acceso directo, a una variable de verdad. En cualquier momento podemos hacer que el puntero apunte a cualquier variable del mismo tipo y rango. O incluso a rodajas de arrays.

Después de ser asignado a una variable de verdad, el array puede utilizarse en expresiones y en asignaciones como haríamos con una variable "normal". Si guardamos un valor sobre un puntero asignado, lo estamos guardando sobre la variable a la que apunta este; analogamente, si usamos el puntero en una expresión, estamos usando el valor almacenado en la variable destino.

Los punteros pueden además "allocatarse" exactamente igual que las variables **`allocatable`**, y por tanto son más útiles que estas. [de hecho, si quisiéramos allocatar un argumento **`allocatable`** dentro de una subrutina y que después esté disponible en el programa principal, no sería posible: sólo puede hacerse con punteros]. En este caso, el puntero no apunta a otras variables, y simplemente es un array más.

¿Por qué querríamos usar punteros? En ocasiones queremos referirnos de forma genérica a una serie de variables. Por ejemplo, tenemos las matrices `A, B, C` y queremos ejecutar una serie de operaciones sobre cada una de ellas. Una posibilidad consiste en crear un puntero `M`, escribir el código de nuestras operaciones dentro de un bucle, y hacer que apunte cada vez a `A, B` ó `C`.

Cuando nuestro programa crece y los datos que manejan cada vez ocupan más memoria, es necesario ahorrar en este recurso. En este respecto, el uso adecuado de punteros evita hacer copias innecesarias de datos en algunas circunstancias. E.g.: trabajamos con punteros que apuntan a submatrices de un array enorme, en lugar de con los datos directamente.

Todo esto requiere declarar los punteros con el atributo **`pointer`**, y las variables destino con **`target`**. Para apuntar un puntero a una variable destino, usamos el operador "`=>`". Podemos cancelar cualquier vinculación que existiera de un puntero con **`nullify`**:

```
program main
implicit none
real*8, target :: x, y, z
real*8, target :: A(4,5)
real*8, pointer :: p, vp(:)
! ...
x = 2.
y = 32.6
p => x ! hacemos que p apunte a x
print*, p
! ...
p => y ! hacemos que p apunte a y
p = x+13.9; ! guarda el valor en y
print*, y
! ...
vp => A(2,:) ! Esta es una forma de poner nombre a rodajas de arrays genéricas
print*, vp
! ...
allocate(vp(5)) ! Al allocatar, el puntero olvida las asignaciones que tuviera.
! ...
end program main
```

Los nuevos estándares de fortran (2003, 2008...) incluyen un nuevo elemento, llamado *procedure pointers*. Esto permite hacer punteros a funciones y subrutinas, útil si se quiere, por ejemplo, que el usuario elija en tiempo de ejecución qué función/subrutina utilizar de entre todas las definidas para una tarea determinada, etc. Podéis encontrar la sintaxis y el uso en los estandares indicados en la sección 6.

6. Para más información...

Fortran es muy amplio. Saber programar en este idioma implica conocer las estructuras fundamentales y las funciones más utilizadas, pero también saber encontrar información sobre la sintaxis y el uso de ciertos aspectos del lenguaje si surge la necesidad.

Existen varias fuentes de información fundamentales, algunas de ellas son las siguientes:

Manuales de fortran. Suelen incluir muchos ejemplos y explicaciones en mayor o menor detalle de prácticamente todos los aspectos del lenguaje. Básicos

para aprender a hablar Fortran. También encontraréis varios libros y manuales gratuítos en internet:

Adams, Jeanne C. et al. (2009). "The Fortran 2003 Handbook", Springer

Adams, Jeanne C. et al. (1992). "The Fortran 90 Handbook", Mcgraw-Hill

Estándares oficiales. Oscuros y espesos, pero con toda la información, cada detalle de cada aspecto del lenguaje, palabra por palabra. Si los manuales eran una guía para aprender un idioma, los estándares sirven a modo de diccionario.

No os obsesionéis por usar la versión más moderna de Fortran que existe. Aunque ya están definidos los estándares 2008, a fecha de hoy no hay aún ningún compilador que soporte todos los aspectos de Fortran 2003. La versión de Fortran más asentada y robusta es 90/95.

Fortran 2003: http://www.dkuug.dk/jtc1/sc22/open/n3661.pdf

Fortran 2008: ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf

Ayuda del compilador: En el menú "help" encontraréis información muy sucinta sobre la sintaxis de las funciones intrínsecas. Por desgracia, la ayuda que viene incluida en Plato es bastante escasa y a veces poco aclaradora.

Internet: Cómodo y rápido, permite encontrar muchas respuestas a dudas comunes que programadores antes que vosotros se han encontrado. Simplemente buscando en Google suele bastar.