Fortran

Introducción

Fortran es un lenguaje de programación orientado a la resolución de problemas matemáticos, aunque es lo suficientemente versátil como para poder ser usado en otros ámbitos como la creación de interfaces.

Como nota a parte, la forma de comentar en Fortran es mediante el caractér `!`. Al poner el signo delante de la linea que queremos comentar esa parte del código ya no se compilará (también se puede poner en medio de una linea, en ese caso comentará sólo lo que tenga a su derecha).

Tipos de datos

- **integer**: Entero de 4 bits.

```
integer:: i
integer*4 :: i
integer(kind = 4) :: i

i = 11
! Todos dan lo mismo, reservan memoria para un valor entero de 4 bits
```

- **real**: Número real de 4 bits.

```
real:: a
real*4 :: a
real(kind = 4) :: a

a = 4.54
```

- **logical**: variable binaria, sólo toma como valores .true. o .false.

```
logical :: b
b = .true.
```

- **character**: variable que almacena un caractér. Como avance de lo que veremos después, podemos modificar el tamaño de memoria de al variable, en el caso de los caracteres es donde más importancia tiene, al ser útil poder leer líneas de caracteres en vez de uno solo.

```
character :: a
character*80 :: b
character (len = 100) :: c
a = 'f'
b = "Cálculo es la mejor asignatura, y lo sabes"
c = "Todos deseamos que vuelva Majadas, no digais que no"
```

  También podemos concatenar cadenas (unirlas):

```
character*40 :: a, b

a = "hola que tal"
b ="pues muy bien"
print*, a//c      ! devuelve un array con muchos espacios (los no usados)
```

- **complex**: números complejos con parte imaginaria y parte real. En la práctica son vectores con dos posiciones.

```
complex :: z = (1,2)
```

Atributos en declaración

A la hora de declarar podemos incluir una serie de atributos que vienen tras el tipo de dato separados por comas para definir las variables:

- `dimension`: indica las dimensiones que tiene la variable (número de dimensiones, tamaño de cada dimensión, etc...).
- `parameter`: indica que el valor dado a la variable no se puede cambiar en el programa.
- `allocatable`: si es una variable cuyo tamaño será indicado más adelante en el programa.
- `external`: si la función viene dada en otro archivo (importante al incluirla en un subprograma).
- `intent`: en subprogramas, si el dato es de sólo entrada(in), salida(out) o ambos(inout).

Arrays / matrices

Ejemplo de array multidimensional:

```
program test
  implicit none
  integer :: h, j, k
  integer, dimension(2,2,2):: i

  forall(h = 1:2, j = 1:2, k = 1:2) i(h,j,k) = 0

  i(1,1,2) = 1
  do h = 1, 2
     do j = 1, 2
     print*,i(h,j,1:2)
     end do
  end do
end program test
```

Aunque no se aprecie, imprimimos un array de 3 dimensiones. Las array son formas de guardar datos en una misma variable con un orden (los índices).

Declaración:

La forma en la que se declaran es:

```
! data_type, dimension(d1, [d2], ...) :: var_name
```

```
! ejemplos:

  integer, dimension(3,4) :: i    !matriz de 3 filas y 4 columnas

  real, dimension(6,7,2) :: a  ! array de 3 dimensiones: 6 filas,
                               ! 7 columnas y 2 hiperfilas
  logical, dimension(3,3,3,3,3) :: b !array de 5 dimensiones con 3 filas, etc...
```

El límite de dimensiones para un array es 7 (es decir, sólo podemos poner 7D: filas, columnas, etc...). El número máximo de valores que podemos poner en cada dimensión depende del ordenador.

Otra forma alternativa para indicar la dimensión de un array al declararla es:

```
integer :: i(3,5,2,4) !array de 4D
```

También podemos indicar rangos para cada dimensión:

```
integer :: arr(-4:7,5:9, 2:3)
```

De forma que las filas van de la -4 a la 7, las columnas de la 5 a la nueve y la tercera dimensión va de 2 a 3.

Arrays dinámicas

Se entiende por array dinámica una matriz que puede tener diferentes tamaños en cada ejecución del programa (aunque no diferente número de dimensiones). La forma en la que se declaran es:

```
implicit none
integer, allocatable :: a(:)
integer, allocatable, dimension(:,:,:) :: b

! ambas formas son análogas (excepto por el número de dimensiones)
```

Para definir su tamaño empleamos `allocate` cuya sintaxis es:

```
  integer :: n,i,j,k   ! el tamaño de cada dimensión debe ser un entero

  read*, n, i, j, k

allocate(a(n),b(i,j,k))  ! ahora ya podemos trabajar con dichas array
```

Es importante borrar las array que hemos inicializado con allocatable. Para ello empleamos `deallocate`:

```
deallocate(a,b)
```

Formas de navegación

Para seleccionar las filas, columnas etc... que queremos usar en cierto momento se puede emplear:

```
arr(1:4,6:9)     ! tenemos seleccionadas así las filas 1 a 4 y columnas 6 a 9
arr(2:10:2,:)      ! selecciona las filas de 2 en 2 desde la fila 2 a la 10
arr(:)           ! equialente a toda la matriz
arr(4:)          ! selecciona la fila 4 en adelante
arr(9:3:-3)      ! selecciona los elementos de 3 en 3 desde la 9 a la tres hacia atrás
```

Precisión

Cuando hacemos referencia a la precisión entendemos el número de bits reservados para una variable, que en fortran sólo pueden ser 4, 8 o 16 bits.

Para definir la precisión empleamos la notación:

- **Simple precisión**: 4 bits resrvados, es la precisión por defecto, por lo que no hace falta especificar

```
real*4 :: a       ! real de 4 bits, simple precisión

real(kind = 4) :: a !análogo
```

- **Doble precisión**: 8 bits reservados.

  ```
  integer*8 :: i
  integer(Kind = 8) :: i
  ```

- **Cuádruple precisión**: 16 bits reservados.

  ```
  integer*16 :: k
  integer(Kind = 16) :: k
  ```

Implicit

De forma implícita, las variables que empiecen por i,j,k,l,m,n son enteras y el resto, reales. Podemos cambiar eso con la declaración `implicit` que permite o bien definir nuestras propias reglas o deshabilitar dicha función.

- Para deshabilitar la declaración implícita de funciones empleamos:

  ```
  implicit none
  ```

- Para definir nuestras propias reglas implícitas:

```
implicit integer (a,b,c,d,e) real (q,r,s,t) logical (v,w,x,y) complex (z)
```

Así, definimos que las variables cuyo nombre empiecen por *a,b,c,d* o *e* son enteras; las que lo hacen por *q,r,s* o *t* son reales; con *v,w,x* o *y* son lógicos y las que lo hacen por z, complejos. Podemos poner la regla que queramos (el dado es sólo un ejemplo).

Impresion y lectura

Impresión

Para imprimir algo en la terminal se emplea o bien `write` o bien `print`:

Print

la sintáxis más usual de `print` es:

```
  integer :: a = 12
print*, "linea de texto", a
```

Donde el `*` indica que se quiere que la salida sea con el formato estándar. En éste formato los elementos a imprimir se separan con comas.

Se puede dar formato al texto:

```
real :: a = 12.0
print '(" Tu número: ", f5.2)', a
```

Para dar formato con print se substituye el `*` por un string de **comillas simples** (para evitar problemas con las string de comillas dobles que incluyamos luego) y dentro del paréntesis escribimos lo que queremos, con cada elemento separado por comas, como en el ejemplo.

Para incluir variables a los formatos personalizados empleamos los códigos de formato:

Código de formato	Uso
A	Asigna espacio para una cadena de caracteres
An	Asigna espacio para una cadena de caracteres de n longitud
an	Asigna espacio para una array de caracteres de n longitud
dn.d	Punto flotante de doble precisión de n caracteres de los cuales d son decimales
en.d	Punto flotante de n caracteres de los cuales d son decimales
gn.d	Real de n caracteres con d decimales
fn.d	Punto fijo de n caracteres de los cuales d son decimales
in	Entero de n caracteres
L	Asigna 2 caracteres para una variable lógica
X	espacio
Tn	Tabulador de n espacios
\ $ \	(sin las barras oblicuas) Indica que no se efectua un cambio de línea
\ / \	(sin las barras oblicuas) Cambio de línea

Write

Write es más usado para imprimir en otros archivos, aunque tiene una funcionalidad parecida:

```
real :: a = 3.0
write(*,*) "Tu número es ", a
write(*, '(" Tu número: ",f6.3)') a
```

La mayor diferencia con `print` es que tiene dos entradas: en la primera indicamos el punto de salida del texto, si es `*` entonces es la salida estándar, es decir la terminal, y si es un número entero asociado a un archivo entonces la salida será en ese archivo. La segunda salida es la del formato, y es igual que el formato de *print*.

Lectura

Para leer datos se emplea la función `read`. La sintaxis básica es:

```
integer :: a
read*, a
```

El `*` indica que el formato es el estandar, es decir, lectura de datos por terminal con el formato de entrada automático.

El formato puede personalizarse:

```
real :: a
read(*,'(f7.3)') a
read(*,*) a
```

En este caso, el `*` indica que la lectura se da por terminal, puede ser substituído por la etiqueta numérica de algún archivo abierto. En la cadena de caracteres podemos indicar bien un `*` para que la lectura sea estándar o bien un formato específico con los códigos de formato.

Bucles

Los bulces se pueden dividir en dos tipos:

Bucles definidos

```
program example
  integer :: var_it, var_top

  do var_it=1,step,var_top ! también se puede usar un do while (var_it <= var_top) poniendo
                           ! var_it = var_it + 1
 ! Código a ejecutar n veces
  end do

end program example
```

La sintaxis al lado del `do` es:

- **var_{it}**: variabe que itera.
- **var_{top}**: variable que inndica el valor final que tomará var_{it}
- **step**: variable de cuanto es el incremento en cada iteración. Si step es negativo entonces var_{it} irá decreciendo; si es positivo, creciendo.

Bucles indefinidos

Pueden ser declarados de dos formas:

- Empleando un bucle do sin iterador y un condicional:

```
program example

  do
 ! Código a repetir hasta que se cumpla la condición de salida
 if( condicion_de_salida ) exit
  end do

end program example
```

- Empleando un bulce do while:

```
  program example

do while(condicion_de_permanecia_en_bucle)
   ! Código a ejecutar mientras la condición de permanencia se cumpla
end do

  end program example
```

cylce

La función cycle rompe con el bucle cuando es llamada, el iterador se ve aumentado (o disminido, depende de como sea el bucle) y se vuelve a inciar el código a ejecutar.

```
do i = 1, n

   if ( i == 5) cycle
   ! de llamarse a cylce, el código a partir de aquí no se ejecuta
   ! en esta iteración y pasa a la siguiente vuelta.
end do
```

Condicionales

La estrucutra de un condicional es:

```
 program example

   if( condicion1 ) then
      ! Código si Condición1 = .true.
   else if( Condicion2 ) then
      ! Código si Condición1 = .false. y Condición2 = .true.

! else if( ) then           Podemos incluir tantos como queramos

   else
      ! Código si todas las condiciones son falsas
   end if  ! o endif

 end program example
```

También se pueden hacer condicionales compactos:

```
program example

  if( condicion ) !Una línea de Código a evaluar si condicion = .true.
end program example
```

Select case

La estrucutra `select case` sirve como un condicional a gran escala. Consta de una posición donde poner la variable a evaluar y una serie de casos con las distintas opciones y lo que se ejecuta de darse uno de ellos. Al final se incluye el caso `default` en caso de que no se evalúe ningún caso.

```
select case(var)
   case (condicion1)
  !código
   case (condicion2)
  !código
  .
  .
  .
   case default
  !código
   end select
```

Operandos lógicos y realcionales

Existen en Fortran 5 operandos lógicos muy útiles para la escritura de las condiciones en un condicional:

- `.and.`: devuelve `.true.` si ambos términos son `.true.`
- `.or.`: devuelve `.true.` si uno de los términos es `.true.`
- `.eqv.`: devuelve `.true.` si ambas condiciones son equivalentes (es decir, si ambos son `.true.` o `.false.`).
- `.neqv.`: devuelve `.false.` si ambas condiciones son equivalentes.
- `.not.`: devuelve `.false.` si el término es `.true.` y viceversa.

A su vez existen los operandos relacionales:

- `/=` : diferentes
- `==` : iguales
- `>` : mayor que
- `<` : menor que
- `<=` : menor igual que
- `>=` : mayor igual que

Devuelven `.true.` o `.false.` dependiendo de si los números que comparamos cumplen la relación (e.g. 1==1, 2<4, 7/=9 o 10<=10 devuevlven `.true.` al cumplirse la relación entre dichos números, pero 2 >=3 no)

```
( 1 == 1 ).and.( 4 < 9 )
(3 == 5).or.(7 /= 8)
.not.(5 == 5)
(3 < 10).eqv.(10 > 7)
(3 <= 4).neqv.(6 >= 9)
```

Archivos

Para trabajar con archivos se usan dos funciones: `open` y `close`.

```
open(num_tag, file = filename, status = "old"/"new", err = num_tag_err )


close(num_tag)

num_tag_err ! Código en caso de error al abrir el archivo
```

- `open`:
  - **num$_{tag}$**: Es un número entero asignado al archivo que estamos abriendo, de forma que cuando queramos realizar operaciones con el archivo podamos identificarlo.
  - **file**: Es una cadena de caracteres con el nombre del archivo que queremos abrir. Si ponemos sólo el nombre se buscará en la carpeta donde se ejecute el programa.
  - **status**: Indica qué debe esperar el programa sobre el archivo.
    - **new**: El programa creará un nuevo archivo con el nombre dado o sobreescribirá (borrará) el archivo con ese nombre.
    - **old**: El programa esperará que ya exista el archivo, de no serlo dará error y saltará a la línea con la etiqueta dada en **err**.
  - **err**: Funciona de la misma forma que la función `go to`, cuando la apertura del archivo da error el programa salta a la línea donde se sitúe la etiqueta **num$_{tagerr}$**.
- `close`: cierra el archivo con la etiqueta **num$_{tag}$**.

Lectura

Para leer de un archivo que tenga por etiqueta numérica, por ejemplo, 1, empleamos la función `read`:

```
character :: a
read(1,*) a
```

Recordemos, en la función `read`, en la primera posición indicamos la etiqueta numérica del archivo abierto que queramos; en la segunda, el formato de *lectura* como se indica en la sección de lectura más arriba.

```
character(100) :: a
read(1,'(A)') a
```

También hay una variable opcional para la función **read**, **end**, el cual indica que, cuando llega al final del archivo va a la línea que tenga la etiqueta indicada ahí:

```
integer :: a
do
   read(1,*,end=2) a
end do

2 ! Código para cuando se llegue al final del archivo
```

Para leer varios valores en una misma línea del archivo tenemos que usar arrays:

Ejemplo.txt:

```
1 2 3 4
```

```
integer :: a(4)

open(1,file="Ejemplo.txt", status="old", err=2)

read(1,*) a

close(1)
```

Escritura

Para escribir en un archivo debemos usar la función **write**:

```
character :: a
open(1, file="file.txt", status="new", err=2)

write(1,*) a

close(1)
```

El * indica que la salida es estándar. Se puede personalizar la salida como se indica en la sección *write*.

Funciones intrínsecas para Archivos

Rewind(unit)

`Rewind` es una función para mover el cursor de lectura de nuevo al inicio del archivo con especificador `unit`. Su sintáxis es:

```
open (1, file = "ejemplo.txt", status = "old", err = 2)

rewind(1)  ! unit = 1

close(1)
```

Fseek(unit,offset,whence)

Mueve el cursor en el archivo con número `unit` un número de bits especificado (`offset`) en relación a la posición especificada (`whence`):

- **0**: Desde el inicio del archivo.
- **1**: Desde la posición actual del cursor.
- **2**: Desde el final del archivo.

```
open(1, file=filename)
call fseek(1,30,0)  ! mover cursor en archivo 1,
                    ! 30 bits desde le inicio
close(1)
```

Subprogramas

Los subprogramas son código escrito fuera del programa principal pensado para realizar tareas similares con datos variables durante la ejecución. En vez de escribir el mismo código varias veces se define un subprograma y se ejecuta cuando sea necesario. Dichos subprogramas pueden estar escirtos dentro o fuera del archivo donde se situe el programa principal, en caso de que estén escritos fuera debemos enlazar todos los archivos .o a usar al mismo tiempo (o compilar todos los archivos .f95 al mismo tiempo si creamos el ejecutable con los archivos .f95).

Hay que tener en cuenta que las variables declaradas en los subprogramas sólo funcionan en cada subprograma. Por ejemplo:

```
program example
real :: a = 3
call ejemplo()
print*, a                  ! devuelve 3 en vez de 2.

end program example

subroutine ejemplo()
real :: a
a = 2
end subroutine ejemplo
```

Existen dos tipos de subprogramas: subrutinas y funciones.

Subrutinas

Las subrutinas no tienen un tipo de dato en específico y no devuelven valor alguno. Son el tipo más versátil de los dos al carecer de dichas limitaciones.

Definición

Para definir una subrutina se emplea la siguiente sintaxis después del programa principal.

```
program example
  ! Código
end program example

subroutine ejemplo( variables de entrada )
  ! Código
end subroutine ejemplo
```

Variables de entrada

Las variables de entrada son los valores que se introducen desde el programa principal al subprograma. Tienen un orden, tipo de dato y permisos específicos que nosotros indicamos. La sintaxis para declararlas son:

```
subroutine example(a,b,c)
     ! (1)                    (2)
  ! data_type, intent(in / out / inout) :: var_name
  real, dimension(4), intent(in) :: a
  character, intent(out) :: b
  logical, intent(inout) :: c

  ! Código

end subroutine example
```

1. **Data$_{\text{type}}$**: El tipo de dato del cual debe ser la variable a introducir.
2. **intent**: el permiso que tiene el suprograma para operar con dicha variable:
   - **in**: Sólo lectura.
   - **out**: Sólo definición, es decir el valor de entrada debe estar vacío.
   - **inout**: La variable puede ser tanto leída como modificada.

También existe otra etiqueta en la declaración de las variables de entrada: `optional`. Con esta etiqueta es posible introducir o no la variable indicada, dejando la posibilidad a diferentes comportamientos de la subrutina. Su sintaxis sería:

```
subroutine example(a, b, c)
  real, intent(in) :: a, b
  real, intent(out), optional :: c
  if(present(c)) then

     ! Código

  else

     ! Código

     end if
end subroutine example
```

Es necesario el condicional para evitar comportamientos anómalos o fallos en la compilación. En dicho condicional se emplea la función `present` () que devuelve `verdadero` si la variable está presente y `falso` en caso contrario.

Funciones

Las funciones son subprogramas que devuelven un valor al ser llamados. El tipo de dato que devuelve está especificado en la declaración de la función.

Definición

```
program example
  ! data_type :: func_name
  real(3) :: func

  stop
end program example

! Ejemplo:
real(3) function func(a, b, c) result(d)
  ! Código
end function func
```

La función cuenta con un tipo de dato que es el mismo tipo que del valor que devuelve. En el caso del ejemplo `d` es una variable `real` tipo array de 3 espacios. La variable del resultado no hace falta especificar después su tipo, dará un error al compilar.

En el caso de que tengamos la función en un archivo a parte y queramos inclírla en un subprograma, debe ser usada, tras la declaración del tipo de variable que es la función, el atributo `external`.

```
real(3), external :: func
```

Variables de entrada

El método es el mismo que con las *subrutinas*.

Recursivas

Las funciones recursivas tienen la ventaja de poder ser llamadas dentro de la propia función. Tienen como sintaxis:

```
  recursive function recurs(variables) result(res)
real :: res  ! es un ejemplo, el tipo de variable del resultado
         ! no tiene que ver con la recursividad
! Código
recurs(variables que queramos) ! como se dijo, podemos volver a
                   ! llamar a la función dentro de su
                                ! propia definición
  end function recurs
```

Interface

En el caso de que una o varias de las variables de entrada sean arrays sin tamaño definido se debe incluír la declaración de variables en una sección denominada `interface` antes de la declaración de variables en el programa principal:

```
program example
  interface
     subroutine ejemplo(a)
   integer, intent(in) :: a(:)

     end subroutine ejemplo
  end interface


end program example


subroutine ejemplo(a)
  integer, intent(in) :: a(:)

end subroutine ejemplo
```

Compilar, Enlazar y Ejecutar

Para compilar programas de fortran se emplea el comando f95/gfortran en al terminal (de no funcionar uno probar el otro, de no funcionar ninguno mirar en la página del compliardor gfortran). La sintaxis es:

```
f95 -c files.f95
```

Donde pone **files.f95** se ponen todos los archivos .f95 que queremos compilar. Tras la compilación se formarán los archivos .o, uno por cada archivo .f95.

Para enlazar y formar el ejecutable se emplea también f95/gfortran:

```
f95 -o name files.o
```

Donde pone **files.o** escribimos todos los archivos .o que queremos enlazar. Por **name** designamos el nombre que queremos ponerle al ejecutable.

Para ejecutar empleamos, en el caso de sistemas Linux y en la terminal, la sintaxis:

```
./name
```

Donde **name** designa el nombre que le pusimos a nuestro ejecutable.

Avanzado

Al compilar y/o enlazar

Para ahorrar tiempo y esfuerzo, podemos emplear archivos planos (sin formato ni extensión) que convertimos en ejecutables para almacenar las instrucciones de compilación y enlace (e inclo ejecución si queremos, todo lo que escribamos se entenderá como lineas de comando para la terminal).

Por convenio le ponemos **princ.Complink** (por princ se entiende el nombre del archivo donde esté el programa principal o incluso en nombre del programa, aunque podemos llamar a este archivo como queramos). En él escribimos en cada linea los comandos a ejecutar y, después, cambiamos su permiso a ejecutable, ya sea click-derecho>propiedades>permisos>click-en-es-ejecutable o con el comando:[1]

```
chmod 700 princ.complink
```

Al ejecutar

A la hora de leer o introducir datos en el programa se puede hacer de forma más fácil (en caso de ser grandes salidas o entradas de datos) usando archivos.

Para la entrada de datos escribimos en un archivo **.dat** los datos que queremos que se lean en el programa, uno por cada línea y en el orden en el que serán

[1] El 700 indicaría que el propietario del archivo puede leer, escribir y ejecutar el programa, mientras que el resto no.

pedidos. Cuando ejecutemos el programa, junto al comando inclímos un $<$ y el nombre dado al fichero .dat (e.g. datos):

```
./name<datos.dat
```

Para la salida de datos podemos escribir en general cualquier tipo de archivos, por conveniencia escribimos con la extensión **.sal** (no indica ningún tipo de archivo, por lo tanto es un documento plano). Para volcar todo lo que se imprima en el programa en el archivo usamos $<$ junto al ejecutable:

```
./name>salida.sal
```

También se puede usar todo a la vez:

```
./name<datos.dat>salida.sal
```

Funciones intrínsecas

Las funciones de base son aquellas que ya vienen por defecto en fortran sin necesidad de librerías externas. En orden alfabético tenemos:

Aimag(z)

Devuelve el valor de la parte imaginaria de z (`complex`).

```
complex :: z = (4,1)
print*, aimag(z)
```

All(x)

Es una función que devuelve `.true.` si todos los valores de x (`array`) son `.true.`

```
integer :: i(20)
i(1:20) = 2
print*, all(i == 2)      ! true porque todos los valores de i son igual a 2
i(4) = 5

print*, all(i == 2)      ! false porque uno de ellos es igual a 5
```

Any(x)

Devuelve `.true.` si hay al menos un `.true.` en x (`array`).

```
real :: a(10,10)
a(1:5,1:4) = 7           ! definimos una matriz de 10 por 10,
 a(1:5,6:10) = 19        ! para dejar claro que la función
a(6:10,1:10) = 9         ! funciona para cualquier dimansión
a(1:5,6:10) = 2
a(5,5) = 200
print*, any(a == 200)  ! devuelve true porque hay al menos un
                       ! valor en a que es 200, aunque sea sólo uno
```

Ceiling(x)

Redondea x (`real`) al entero más pequeño que lo mayore.

```
real :: a = 34.5

print*, ceiling(a)   ! da 35
```

Conjug(z)

Devuelve el conjugado de z (`complex`).

```
complex :: z = (7,3)
print*, conjg(z)
```

Count(x)

cuenta cuántos valores `.true.` tiene x (`array`).

```
logical :: a(10)
a(1:5) = .true.
a(6:10) = .false.
print*, count(a)
```

También funciona para contar cuantos valores cumplen cierta realación. E.g. si a es un array con los enteros del 1 al 10 y aplicamos count(a $<$ 6) devolvería 5, pues al hacer a $<$ 6 se genera un array de valores lógicos `.true.` o `.false.`

Maxval(x[,dim][,mask])

Devuelve el máximo valor en x (`array`).

- Si `dim` está presente:
  - **1** para hacerlo por filas (devolverá entonces una matriz).
  - **2** para hacerlo por columnas.
  - etc... con el resto de dimensiones

Minval(x[,dim][,mask])

Análogo que maxval pero con el valor mínimo.

Size(array, dim, kind)

Devuelve el tamaño de una array.

```
program example
  real :: a(3)

  print*, size(a)

end program example
```

Los dos valores opcionales son dim y kind:

- `Dim` sirve en caso de ser un array de más de una dimensión. Si es `1` entonces devuelve el número de filas; si es `2` devuelve el número de columnas; etc...

Valor de DIM	Efecto
1	devuelve el número de filas
2	devuelve el número de columnas
etc..	lo mismo para el resto de dimensiones

```
program example
  real :: a(3,2)

  print '("Número de filas: ",i0)', size(a,1)
  print '("Número de columnas: ",i0)', size(a,2)
end program example
```

Sum(x[, dim][,mask])

Sum realiza la suma de los valores de x (`array`).

- Si `dim` está presente:
  - **1** para sumar por filas.
  - **2** para sumar por columnas.
  - etc... para el resto de dimensiones

```
integer:: a(5,5)
a(1:3,:) = 4
a(4:5,:) = 7
print*, sum(a,dim = 2) ! suma por columnas
```

Trim(s)

Devuelve la cadena de caracteres introducida sin los espacios en balnco al final de s (`character`).

```
character*30 :: a, b
a = "hola"
b  = "que tal"
print*, a//b
print*, trim(a)//b
```

Footnotes