Introducción a Fortran 90

6 de marzo de 2007

Esquema que seguiremos

1. Generalidades
2. Variables en Fortran 90
3. Expresiones
4. Arrays
5. Bloques de control
6. Programación estructurada
7. Modules
8. Instrucciones de entrada/salida

Tipos de programas en FORTRAN

- Programa fuente (nombre.f90): Es el programa que editamos y que contiene las instrucciones del lenguaje FORTRAN. Para su edición, utilizaremos cualquier editor de texto (edit, emacs, vi, pico,...).

⇓ COMPILACIÓN ⇓

- Programa objeto (nombre.o): El programa fuente se hace comprensible al ordenador.

⇓ LINKADO ⇓

- Programa ejecutable (nombre.out).

Órdenes de compilación y linkado

- Compilación:

  > gfortran -c dicotomia.f90 ! se crea dicotomia.o

  Pueden compilarse al mismo tiempo varios programas fuente:

  > gfortran -c dicotomia.f90 auxiliar.f90
  ! se crean dicotomia.o y auxiliar.o

- Linkado:

  > gfortran -o dicotomia.out dicotomia.o

  Se puede crear un único ejecutable con varios programas objeto:

  > gfortran -o dicotomia.out dicotomia.o auxiliar.o

- Todo junto **(no recomendado)**:

  > gfortran -o dicotomia.out dicotomia.f90

  > gfortran -o dicotomia.out dicotomia.f90 auxiliar.f90

- Ejecución:

  > ./dicotomia.out

Empezando...

- No hay distinción entre mayúsculas y minúsculas.
- Los nombres de programas y variables deben tener un máximo de 32 caracteres. El primer carácter tiene que ser una letra y los siguientes pueden ser números, letras o subrayado.
- **Comentarios:** Se ponen después del signo "!".
- La longitud máxima de línea es de 132 caracteres.
- Para indicar que una instrucción continua en la línea siguiente se pone "&" al final de la línea:

```
print*, &
        'Esto es una prueba'
```

  Además, la línea siguiente tiene que empezar por "&" si queremos continuar textos entrecomillados:

```
print*, &
        'Esta linea es muy larga &
        & y por eso la dividimos'
```

- Se pueden poner varias instrucciones en la misma línea si las separamos por ";"

```
A=0 ; B=0
```

Variables en FORTRAN 90

Hay cinco tipos básicos de variables:

- Enteras.
- Reales.
- Complejas.
- Lógicas.
- Carácter.

De forma implícita, FORTRAN guarda los nombres de variables que empiecen por {i, j, k, l, m, n} para variables enteras y el resto para variables reales de precisión simple.

Es recomendable empezar un programa anulando este criterio predefinido con la instrucción IMPLICIT NONE. Así tenemos que declarar todas las variables que utilicemos y evitaremos muchos errores.

Variables enteras

Almacenamos en ellas números enteros: ...,-3,-2,-1,0,1,2,3,...

> *integer :: i, j1, j2* !Declara i,j1,j2 como variables enteras

Dentro de las variables enteras se distinguen distintos tipos dependiendo de la cantidad de memoria que se use para almacenarlas. Estos tipos pueden depender de la máquina en que estemos trabajando. En general:

- *integer (kind=1) :: i ! Números enteros entre -127 y 127*
- *integer (kind=2) :: j ! Números enteros entre -32767 y 32767*
- *integer (kind=4) :: k ! Números enteros entre -2147483647 y 2147483647*

En nuestro sistema, el tipo por defecto para variables enteras es kind=4. De todas formas, conviene ponerlo explícitamente.

Variables reales

Almacenamos en ellas números reales.

Al igual que para las variables enteras, hay distintos tipos de variables reales que dependen de la máquina utilizada. Habitualmente son:

- *real (kind=4) :: x1 ! 7 cifras significativas (precisión simple).*
- *real (kind=8) :: x2 ! 14 cifras significativas (precisión doble).*
- *real (kind=16) :: x3 ! 28 cifras significativas (prec. cuádruple).*

¡¡Atención!! El tipo real por defecto es kind=4. Nosotros usaremos, salvo que se diga lo contrario, kind=8.

Sobre las constantes reales


```
real (kind=8) :: x1
x1=0.3
print*, x1      !En pantalla: 0.300000011920929
```

```
real (kind=8) :: x1
x1=0.3_8
print*, x1      !En pantalla: 0.300000000000000
```

Variables complejas

Los números complejos se almacenan como pares de números reales.

Los tipos son los mismos que para las variables reales:

> *complex (kind=4) :: c1*

> *complex (kind=8) :: c2*

> *c1=(1._4 , 2.05_4) ; c2=(0.1_8,0)*

Variables lógicas

Admiten dos valores posibles : .FALSE. y .TRUE.

```
> logical :: log1, log2
> log1= .true.
> log2= .false.
```

Variables carácter

Nos permiten guardar cadenas alfanuméricas. Debemos especificar el número máximo de caracteres.

> *character (len=10) :: fichero*
> *fichero= 'datos'*

Parameter

Sirve para almacenar un valor constante en una variable de forma que no cambie a lo largo de todo el programa:

> *integer, parameter :: dim=100*

> *real, parameter :: pi_simple=3.14159265*

> *real(kind=8), parameter :: pi= 3.14159265358979*

Expresiones

Expresiones aritméticas:

- operandos aritméticos
- operadores (ordenados por prioridad):
  **
  *, /
  +, -
  El orden de prioridad puede alterarse usando paréntesis.
- resultado: será del tipo del operando más potente
  (por ej., real(kind=8)*entero(kind=4) $\Rightarrow$ real(kind=8))

Expresiones relacionales:

- operandos lógicos o aritméticos
- operadores:
  > o .GT. ; >= o .GE.
  < o .LT. ; <= o .LE.
  == o .EQ. ; /= o .NE.
- resultado: .TRUE. o .FALSE.

Expresiones

Expresiones lógicas:

- operandos: expresiones relacionales
- operadores: .NOT. , .AND. , .OR.
- resultado: .TRUE. o .FALSE.

Ejemplo:

> *if ((i<10).AND.(i/=5)) ...*

Expresiones de caracteres:

- operandos: cadenas de caracteres
- operador: concatenación (//)
- resultado: otra cadena de caracteres

Ejemplo:

```
> ch1='fich_'
> ch2='out'
> ch3=char1//char2     ! ch3='fich_out'
```

Expresiones

Orden de prioridad:

1. aritméticas
2. concatenación
3. relacionales
4. lógicas

Arrays

Los arrays son variables dimensionadas.
Identifican un conjunto de posiciones de memoria, todas ellas de un mismo tipo (real, entero, etc.).

La forma de definirlas es:

```
real, dimension(3) :: a    ! tres variables reales: a(1), a(2) y a(3)

real (kind=8), dimension(0:2) :: b       ! b(0), b(1) y b(2)

real, dimension(3,0:2) :: c       ! c(1,0), c(2,0),c(3,0),c(1,1), ...

integer (kind=4), dimension(-1:2,2) :: d ! d(-1,1), d(0,1),...
```

Asignación de valores en un array

- Asignación directa de un vector:

```
real (kind=8), dimension(0:3) :: a
a=(/1.0_8, 2.1_8, 3.9_8, 4.24_8/)
```

- Asignación mediante un bucle implícito:

```
integer, dimension(5) :: i1,i2,i3
i1=0
i2(1:4)=1; i2(5)=0
i3(1:5)=(/(2.0*j, j=1,5)/)
```

- Para la definición de una matriz hay dos opciones:

```
integer, dimension(3,2) :: i
```

(1) i(1,:)= (/1,2/) ; i(2,:)=(/3,4/) ; i(3,:)=(/5,6/)

(2) i=reshape((/1,3,5,2,4,6/), shape=(/3,2/)).

Nota: El comando RESHAPE suele llevar a errores.

Operaciones con arrays

- Se puede operar de manera directa:

```
integer, dimension(3) :: a, b, c, d
real (kind=8), dimension(20) :: x1, x2, x3
a=(/1, 2, 3/) ; b=(/4, -5, 6/)
c=a+b        !c = (/5, −3, 9/)
d=a*b        !d = (/4, −10, 18/)
:
x1=x1/3.1_8 + x2*dsqrt(x3)
```

- Otra posibilidad (**no recomendada**) es operar elemento a elemento en un bucle.

Funciones predefinidas con arrays

- `where`: Efectúa cierta instrucción en las posiciones del array que verifican una condición:
  where (a /= 0) b=b/a
  where (a == 0) b=1/epsilon(b)
- `count` ((expresión lógica) [,dim]): Número de elementos que verifican la expresión lógica. Si dim=1, da el resultado por columnas; si dim=2, por filas.
  count (a == 0) $\Longleftrightarrow$ count (mask=(a == 0))
- `maxval`(array [,dim] [,mask]): Calcula el valor máximo.
  maxval(a, mask=(a < 0)) ; maxval(b)
- `minval`(array [,dim] [,mask]): Calcula el valor mínimo.
- `maxloc`(array [,mask]): Localización del mayor elemento. En este caso la posición que resulta es absoluta.
- `minloc`(array [,mask]): Localización del menor elemento.

Funciones predefinidas con arrays


```
program max_val
implicit none
integer, dimension(3,3) :: M
M(1,:)=(/1,2,6/) ; M(2,:)=(/4,11,3/) ; M(3,:)=(/7,8,9/)
print*,maxval(M,dim=1)     ! Imprime: 7,11,9
print*,maxval(M,dim=2)     ! Imprime: 6,11,9
print*,maxval(M,dim=1,mask=(M<5))     ! Imprime: 4,2,3
end program max_val
```

Funciones predefinidas con arrays

- `product`(array [,dim] [,mask]): Producto de los elementos del array.
- `sum`(array [,dim] [,mask]): Suma de los elementos del array.
- `dot_product`(vec_1, vec_2): Producto escalar.
- `matmul`(matriz_a, matriz_b): Multiplicación matricial.
- `transpose`(matriz): Matriz transpuesta.

El producto escalar permite simplificar muchos sumatorios:

```
xsum=0.0_8
do i=1,n
  xsum=xsum+a(i)*b(i)
end do
```

$\Leftrightarrow$ `xsum=dot_product(a(1:n),b(1:n))`

Norma euclídea de un vector:

```
norm=sum(vec*vec)
norm=dsqrt(norm)
```

$\Leftrightarrow$ `norm=dot_product(vec,vec)`

Dimensionamiento dinámico

Fortran 90 permite reservar memoria en tiempo de ejecución. Para ello tenemos las instrucciones ALLOCATE y derivadas.

```
real, dimension(:), allocatable :: a,b
read*, n
allocate(a(n)) ; allocate(b(0:n))
    [...]
deallocate(a) ; deallocate(b)
```

Para comprobar si hubo algún problema en la reserva o liberación de memoria, se usa el comando STAT. El resultado es 0 si no hay problemas.

```
[...]
allocate (a(n), stat=testeo)
if (testeo /= 0) stop
```

Para comprobar si una variable "allocatable" tiene reservada memoria se usa la instrucción ALLOCATED:

```
if (allocated(a)) deallocate(a)
if (.not. allocated(a)) allocate(a(n))
```

Bloque de control IF

Estructura tipo:

[nombre:] IF (expresión lógica) THEN

⋮

ELSE IF (expresión lógica) THEN [nombre]

⋮

ELSE [nombre]

⋮

END IF [nombre]

Otra opción, en ocasiones más cómoda, es:

IF (expresión lógica) {una sola orden (sin THEN)}

Ejemplo de uso de IF


```
program aprobar_CN
implicit none
real(kind=8) :: nota_teor, nota_prac

print*,'Nota de teoria?'
read*, nota_teor
print*,'Nota de practicas?'
read*, nota_prac
if ((nota_teor>=4.5_8).and.(nota_prac>=0.5_8)) then
     print*,'ENHORABUENA! Has aprobado &
          & las dos partes de la asignatura'
else if ((nota_teor+nota_prac=5.0_8).and.(nota_prac>=0.5_8)) &
               then
     print*,'Por los pelos... pero pasas'
else
     print*,'Nos vemos en septiembre'
end if
end program aprobar_CN
```

Bloque de control CASE

Ejecuta un bloque de sentencias u otro según el valor que pueda tomar una expresión o variable.

```
[nombre:] SELECT CASE (expresión o variable)
 CASE (...) [nombre]
    ⋮
 CASE (...) [nombre]
    ⋮
 CASE DEFAULT [nombre]
    ⋮
 END SELECT [nombre]
```

Nota: La opción CASE DEFAULT no tiene que ir necesariamente en último lugar.

Ejemplo de uso de CASE


```
program que_hacemos_hoy
implicit none
integer(kind=4) :: dia
do
    print*,'Introduzca el dia de la semana (lunes=1, etc.)'
    read*, dia
    if ((dia>=1).and.(dia<=7)) exit
end do
print*,'Hoy toca...'
select case (dia)
    case (1,2,3,4)
        print*,'chapar'
    case (5)
        print*,'chapar'
    case default
        print*,'chapar en casa'
end select
end program que_hacemos_hoy
```

Bucles: DO

Estructura tipo:

```
[nombre:] DO (variable de control = vinicial,vfinal,incremento)
    ⋮
END DO [nombre]
```

Un ejemplo:

```
integer (kind=4):: it
do it = 0, 10, 2
    ⋮
end do
```

En este caso el bucle va de 0 a 10 incrementando en 2 la variable de control cada vez que llega al "end do".
Si el incremento es 1 se puede omitir.
La variable de control no puede modificarse dentro del bucle.

Bucles: DO

Hay dos comandos importantes que pueden emplearse en la estructura DO:

- EXIT: abandona el bucle (pasa a la primera línea después de END DO).
- CYCLE: aumenta el contador (pasa a la línea END DO).

Una posibilidad interesante es prescindir del bloque de control y poner un EXIT combinado con un IF:

```
do
   :
    if (...) exit
end do
```

Bucles: DO


```
program acertar_numero_entero
implicit none
integer (kind=4) :: i,num,pru

num=68

do i=1,10
    print*,'Intento ',i
    print*,'Teclea un numero del 1 al 100 '
    read*,pru
    if (pru==num) then
        print*,'Acertaste!'
        exit
    end if
end do

if (i==11) print*,'No ha habido suerte'
end program acertar_numero_entero
```

Bucles: DO


```
program acertar_numero_entero
implicit none
integer (kind=4) :: i,num,pru

num=68

do i=10,1,-1
    print*,'Te quedan ',i,' intentos'
    print*,'Teclea un numero del 1 al 100 '
    read*,pru
    if (pru==num) then
        print*,'Acertaste!'
        exit
    end if
end do

end program acertar_numero_entero
```

Bucles: DO


```
program acertar_numero_entero
implicit none
integer (kind=4) :: i,num,pru

num=68

i=0 ; print*,'De esta cae...'
do
    print*,'Llevas ',i,' intentos'
    print*,'Teclea un numero del 1 al 100 '
    read*,pru
    if (pru==num) then
        print*,'Acertaste!'
        exit
    end if
    i=i+1
end do

end program acertar_numero_entero
```

Procedimientos

En Fortran 90 hay dos tipos de procedimientos:

- Funciones.
- Subrutinas.

Los procedimientos pueden ser internos o externos:

- Internos:
  Antes de la línea de END PROGRAM, se escribe CONTAINS y se añaden los procedimientos.
- Externos:
  El procedimiento se sitúa fuera del cuerpo del programa principal, ya sea en el mismo fichero o en otro distinto.

Funciones

Las funciones permiten calcular un valor a partir de otros dados.
Es la traducción informática de la idea de función matemática.

```
program posit_negat
implicit none
real(kind=8) :: x,y,z
print*,'Introduce la abscisa' ; read*,x
print*,'Introduce la ordenada' ; read*,y
z=f(x,y)
if (z>= 0) then
     print*,'Resultado positivo'
else
     print*,'Resultado negativo'
end if
contains
     function f(x,y)
     real(kind=8) :: f,x,y
     f=dsin(x*y)*dcos(x*y)
     end function f
end program posit_negat
```

Funciones

Ejemplo de función usando RESULT (recomendado):

```
program posit_negat
implicit none
real(kind=8) :: x,y,z
print*,'Introduce la abscisa'; read*,x
print*,'Introduce la ordenada'; read*,y
z=f(x,y)
if (z>= 0) then
     print*,'Resultado positivo'
else
     print*,'Resultado negativo'
end if
contains
     function f(x,y) result (res)
     real(kind=8) :: x,y,res
     res = dsin(x*y)*dcos(x*y)
     end function f
end program posit_negat
```

Subrutinas

Permiten aislar una parte del programa, facilitando la programación estructurada.

```
program suma
implicit none
real (kind=8):: a , b , res
call lee_datos
call calculos(a,b,res)
print*,'res: ',res
contains
  subroutine lee_datos
  read*,a
  read*,b
  end subroutine lee_datos

  subroutine calculos(x1,x2,x3)
  real(kind=8) :: x1,x2,x3
  x3=x1+x2
  end subroutine calculos
end program suma
```

Recursividad

Tanto las funciones como las subrutinas pueden llamarse a sí mismas:

```
recursive function factorial(n) result(res)
integer (kind=4):: n
real(kind=8) :: res
if (n==1) then
  res=1.0_8
else
  res=n*factorial(n-1)
end if
end function factorial
```

Procedimientos internos

Los procedimientos internos heredan las variables del programa principal, lo que puede dar lugar a errores. Las variables que sean locales a un procedimiento deben definirse en éste.

¡MAL!

```
program prueba
implicit none
integer (kind=4) :: i
do i=1,10
  print*,'i: ',i
  call sub_int
end do
contains
  subroutine sub_int
  do i=1,5
    print*,'i en sub: ',i
  end do
  end subroutine sub_int
end program prueba
```

BIEN

```
program prueba
implicit none
integer (kind=4) :: i
do i=1,10
  print*,'i: ',i
  call sub_int
end do
contains
  subroutine sub_int
  integer (kind=4) :: i
  do i=1,5
    print*,'i en sub: ',i
  end do
  end subroutine sub_int
end program prueba
```

Procedimientos externos

En este caso, **no hay transferencia implícita de variables.** Se debe incluir en el programa que realiza la llamada un bloque INTERFACE, para detectar incompatibilidades de tipos.

```
program suma
implicit none
real (kind=8):: a, b, res
interface
  subroutine calculos (x1,x2,x3)
  real (kind=8) :: x1,x2,x3
  end subroutine calculos
end interface
call int_datos
call calculos(a,b,res)
print*,'res: ',res
contains
  subroutine int_datos
  read*,a ; read*,b
  end subroutine int_datos
end program suma
```

```
subroutine calculos(x1,x2,x3)
implicit none
real(kind=8) :: x1,x2,x3
x3=x1+x2
end subroutine calculos
```

Apuntes importantes

1. En cualquier momento en un procedimiento (interno o externo) se puede utilizar el comando RETURN para devolver el control al programa principal.
2. Las variables que se pasan como argumentos se pueden (y se deben) clasificar en función de si son de entrada (IN, no se pueden modificar), de salida (OUT) o de entrada/salida (INOUT). Hacerlo así facilita la comprensión de los códigos. Para ello, se usa el comando INTENT:

```
subroutine calculos(x1,x2,x3)
implicit none
real(kind=8), intent(in) :: x1
real(kind=8), intent(inout) :: x2
real(kind=8), intent(out) :: x3
x3=x1+x2
x2=x1*x2
end subroutine calculos
```

Argumentos desordenados

Al llamar a un procedimiento, los argumentos se pueden colocar en cualquier orden si se indica en la llamada el nombre de los argumentos virtuales. En este caso es obligatorio el uso del bloque interface.

```
program desordena
implicit none; real(kind=8) :: x1,x2,dif,x
interface
    subroutine escritura_resultados(resultado,error)
    real(kind=8), intent(in) :: resultado, error
    end subroutine escritura_resultados
end interface
[...]
call escritura_resultados(error=dif,resultado=x)
end program desordena
subroutine escritura_resultados(resultado,error)
implicit none
real(kind=8), intent(in) :: resultado, error
print*,'error ',error ; print*,'resultado ',resultado
end subroutine escritura_resultados
```

Dimensiones asumidas

Al pasar un array a un procedimiento se pueden **asumir las dimensiones superiores** (por defecto, la inferior es 1).

```
real(kind=8), dimension(10) :: a,b
real(kind=8), dimension(0:10,-1:2) :: c
call opera(a,b,c)
  ...
contains
  subroutine opera(a,b,c)
  real(kind=8), dimension(:) :: a,b
  real(kind=8), dimension(0:,-1:) :: c
  :
```

Si el procedimiento es externo y queremos asumir dimensiones, es **obligatorio** usar un bloque INTERFACE en el programa que llama a la subrutina donde se asumen dimensiones, excepto si usamos un módulo, como veremos más adelante.

Transferencia de variables *allocatables*

Cuando a un procedimiento se le pasa como argumento un array *allocatable*, debe recibirse como un array de dimensión asumida y es **obligatorio** el uso de un bloque interface. En este caso, **hay que alocatar el array en el programa principal**.

```
real(kind=8), dimension(:), allocatable :: a
interface
    subroutine asignar(v)
    real(kind=8), dimension(:) :: v
    end subroutine
end interface
read*, n
allocate(a(n))
call asignar (a)
:
subroutine asignar(v)
real(kind=8), dimension(:) :: v
v=(/ (i*2.0_8, i=1,n) /)
end subroutine asignar
```

Arrays automáticos

Si en una subrutina necesitamos un array auxiliar, podemos definirlo en la misma subrutina con argumentos conocidos en tiempo de ejecución (por ejemplo, relativos al tamaño de otro array):

```
subroutine calculos (x,y)
implicit none
real(kind=8), dimension(:), intent(in) :: x,y
real(kind=8), dimension(size(x)) :: aux
    [...]
end subroutine calculos
```

Arrays como resultado de funciones

El resultado de una función puede ser un array (allocatable o no). En este caso no es posible recibirlo como de dimensión asumida; sí es posible dimensionarlo en función de una variable.

```
function potencia(a,b) result(c)
implicit none
integer, dimension(:), intent(in) :: a,b
integer, dimension(size(a)) :: c
c = a**b
end function potencia
```

Módulos

Los módulos son una estructura muy importante en Fortran 90 porque permite compartir variables y procedimientos, facilitando tanto una transferencia sencilla de variables como la creación de librerías propias.

```
MODULE nombre
  ⋮
CONTAINS
  ⋮
END MODULE nombre
```

Módulos para compartir variables

La estructura MODULE permite pasar variables comunes a distintos procedimientos de una forma sencilla. Se facilita así la transferencia de parámetros y de variables dimensionadas.

```
module variables
implicit none
real(kind=8), dimension(:), allocatable :: a
end module variables
```

```
program alocatar
use variables
implicit none
integer(kind=4) :: n
read*, n
allocate (a(n))
call asignar(n)
end program alocatar
```

```
subroutine asignar(n)
use variables
implicit none
integer(kind=4) :: n
 a=(/ (i*2.0_8, i=1,n) /)
end subroutine asignar
```

Módulos para compartir variables

Podemos restringir el acceso a las variables de un módulo con el comando ONLY:

```
module variables
implicit none
real(kind=8), dimension(:), allocatable :: a,b
end module variables

subroutine asignar(n)
use variables, only : a
implicit none
integer(kind=4) :: n
  a=(/ (i*2.0_8,  i=1,n) /)
end subroutine asignar
```

Módulos para compartir variables

Y también con ONLY podemos cambiarles localmente el nombre:

```
module variables
implicit none
real(kind=8), dimension(:), allocatable :: a,b
end module variables

subroutine asignar(n)
use variables, only : vector => a
implicit none
integer(kind=4) :: n
  vector=(/ (i*2.0_8,  i=1,n) /)
end subroutine asignar
```

Módulos para compartir procedimientos

Además, en un módulo se pueden incluir procedimientos que luego se compartirán de forma sencilla con otras partes del programa. Podemos evitar así el uso de muchos bloques interface.

```
module asignacion
contains
    subroutine asignar(v,n)
    implicit none
    integer::i,n
    real(kind=8), dimension(:) :: v
    vector=(/ (i*2.0_8,  i=1,n) /)
    end subroutine asignar
end module asignacion

program probamos
use asignacion
implicit none;
real(kind=8), dimension(:), allocatable :: a
read*, n;    allocate(a(n))
call asignar (a,n)
end program probamos
```

Otras posibilidades...

Los módulos también permiten redefinir operaciones o trabajar con nuevos tipos que nosotros podemos crear.

No detallaremos más sus posibilidades. Para los que estén interesados recomendamos leer las referencias en la página web:

http://www.cs.rpi.edu/ szymansk/OOF90/main.html

o el libro (que está en la bibliografía):

W.S. Brainerd, Ch.H. Goldberg, J.C. Adams, *Programmer's guide to Fortran 90*, Unicomp, 1994.

Instrucciones de Entrada / Salida

Transfieren información entre la memoria principal y la externa o auxiliar. Son de 3 tipos:

- Apertura y cierre de ficheros: OPEN, CLOSE
- Transferencia de datos: READ, WRITE, PRINT
- Posicionado en el fichero: BACKSPACE, REWIND, ENDFILE

Un fichero se compone de registros, los cuales tienen varios campos.

- Tipos de registros:
  - Formateados
  - No formateados
  - EOF (fin de fichero)
- Tipos de ficheros:
  - **Formateados**
  - No formateados
- Formas de acceso a un fichero:
  - **Secuencial**
  - Directa

Apertura, cierre y posicionado

- OPEN: Establece una conexión entre una unidad lógica y un fichero, definiendo las propiedades de la misma.

  OPEN (UNIT, FILE, STATUS, ACCESS, FORM)

  - UNIT: Expresión entera ($1 \leq n \leq 255$) que define la unidad lógica
  - FILE: Nombre del fichero (cadena de caracteres)
  - STATUS: 'new', 'old', '**unknown**'
  - ACCESS: 'direct', '**sequential**'
  - FORM: **'formatted'**, 'unformatted'

```
open (unit=1, file='alumnos.dat', status='old')
open (2, file='notas.out', status='unknown')
```

- CLOSE: Cierra la conexión entre fichero y unidad lógica.

```
close(1)
close(unit=2)
```

- REWIND: "Rebobina" y se coloca en el primer registro de la unidad.

```
rewind(1)
```

Transferencias de datos

Componentes de una sentencia de transferencia de datos:

1. Orden: READ, WRITE, PRINT.
2. Lista de control: especificaciones.
   2.1. Unidad de fichero.
   2.2. Formato para la lectura o escritura.
3. Lista de entrada/salida: nombre de variables que leemos o escribimos.

Transferencias de datos: Lectura

READ (secuencial): Lee datos de registros secuenciales.

- Formateados: lee, convierte a binario y asigna.
- No formateados: lee y asigna.

```
read (1,100) a
100 format (F5.3)

read (5,*) a
read*, a

read 100, a
```

La unidad 5 es la predefinida para la lectura de datos por teclado.

Transferencia de datos: Escritura

- WRITE: Transfiere datos de la memoria a unidades auxiliares (pantalla /ficheros). Puede ser secuencial o de acceso directo.

```
write (1,100) a
100 format (F5.3)

write (6,*) a
write*, a
write 100, a
```

La unidad 6 es la predefinida para la escritura de datos por pantalla.

- PRINT: Transferencia de datos desde la memoria principal a la pantalla. Pueden escribirse formateados o sin formatear:

```
print*, 'hola'
print 200, a
200 format (...)
```

Sentencias de especificación de formato

Indican cómo se encuentran formateados los datos que leemos o cómo queremos escribirlos en el fichero o en pantalla.
Existen tres maneras de especificar el formato:

- Usando una expresión carácter:
  write(2, '(1x,F8.2)') a
- Mediante una etiqueta con la sentencia que contiene el formato:
  write (2,100) a
  100 format (F5.3)
- En formato libre:
  write (2,*) a

Veremos a continuación los tipos de códigos para datos.

Formato i (valores enteros)

i: Formato para valores enteros.
Sintaxis: in[.m]

- n : constante entera que indica el número de posiciones del campo.
- m : escribe al menos 'm' caracteres, rellenando con ceros si fuera necesario (es opcional).

Ejemplo (**entrada**):

Valor externo	Formato	Valor interno
8235	i4	8235
␣-67	i4	-67
-346	i4	-346
4530	i3	453

Ejemplo (**entrada**):

Cadena externa: -23␣4783+15
Sentencia: *read(1,100) k1,k2,k3*
100 format (i3, i3, i2)
Resultado: k1=-23, k2=47, k3=83

Formato i (valores enteros)

i: Formato para valores enteros.
Sintaxis: in[.m]

- n : constante entera que indica el número de posiciones del campo.
- m : escribe al menos 'm' caracteres, rellenando con ceros si fuera necesario (es opcional).

Ejemplo (**salida**):

Valor interno	Formato	Valor externo
824	i4	␣824
0	i3	␣␣0
1273	i3	***

Ejemplo (**salida**):

```
k=128
write(6,100) k
100 format(i7.5)
```

→ ␣␣00128

Formato f (forma decimal)

f: Formato para números reales en forma decimal.

- Sintaxis: fn.d
  - n : número total de dígitos incluyendo el signo y el punto decimal (si existen).
  - d : número de dígitos de la parte decimal.
- Observaciones (referentes a la salida):
  - Los caracteres se ajustan a la derecha.
  - Si el número tiene más de 'd' decimales se redondea.
  - Si tiene más de 'n' caracteres, escribe asteriscos.
- Ejemplo (entrada):

Valor externo	Formato	Valor interno
43305E-2	f6.2	433.05
43305E-2	f8.0	433.05

- Ejemplo (salida):

Valor interno	Formato	Valor externo
25.338	f9.3	␣␣␣25.338
25.338	f6.1	␣␣ 25.3
-12.345	f5.3	*****

Formato e

e: Números reales en formato exponencial.

- Sintaxis: en.d
  - n : número total de posiciones del campo (ha de incluirse el signo, el punto decimal y el exponente, además de la mantisa).
  - d : número de cifras decimales de la mantisa.

- Ejemplo (salida):

Valor interno	Formato	Valor externo
83.974	e10.2	␣␣0.84E+02
	e10.4	0.8397E+02
	e9.4	*********

Formatos a, x y especiales

- **a**: Manejo de cadenas de caracteres.
  character (len=6) :: variable
  variable= 'musica'
  write (6,100) musica
  100 format (a6) $\longrightarrow$ musica
  100 format (a7) $\longrightarrow$ musica$\not\emptyset$
  100 format (a4) $\longrightarrow$ musi
  100 format (a) $\longrightarrow$ musica
- **x** : Espacios en blanco.
  nx: $\begin{cases} \text{(entrada) n= número de caracteres que se salta} \\ \text{(salida) n= número de blancos que escribe} \end{cases}$
- **t**: Tn lee o escribe en la n-ésima columna.
- **/** : Salto de línea.
  $: Elimina el 'retorno de carro' (sólo se usa en salida).

Ejemplo


```
3 2
15.245 728.346
2.5 7.8654
14.324 1025.22
```

matriz.dat

```
program trasponer
implicit none
integer(kind=4) :: nf,nc,iter
real(kind=8),dimension(:,:),allocatable::m
open(1,file='matriz.dat',form='formatted')
rewind(1)
read(1,*)nf,nc
allocate(m(nf,nc))
do iter=1,nf
  read(1,*) m(iter,1:nc)
end do
close(1)
```

Ejemplo


```
open(2,file='matriz_tras.dat',form='formatted')
rewind(2)
write(2,'(i2,1x,i2)')nc,nf
do iter=1,nc
  write(2,100) m(1:nf,iter)
end do
close(2)
100 format(99(F6.2,1x))
end program
```

```
 2   3
 15.24      2.50     14.32
728.35      7.87 ******
```

matriz_tras.dat

Otro ejemplo


```
3 2
15.245 728.346
2.5 7.8654
14.324 1025.22
```

matriz.dat

```
program trasponer
implicit none
integer(kind=4) :: nf,nc,iter
real(kind=8),dimension(:,:),allocatable::m
open(1,file='matriz.dat',form='formatted')
rewind(1)
read(1,*)nf,nc
allocate(m(nf,nc))
do iter=1,nf
  read(1,*) m(iter,1:nc)
end do
close(1)
```

Otro ejemplo


```
open(2,file='matriz_tras.dat',form='formatted')
rewind(2)
write(2,'2(i3,1x)')nc,nf
do iter=1,nc
  write(2,100) m(1:nf,iter)
end do
close(2)
100 format(99(E12.6,1x))
end program
```

matriz_tras.dat:

```
 2   3
0.152450E+02 0.250000E+01 0.143240E+02
0.728346E+03 0.786540E+01 0.102522E+04
```

"Un programa de ordenador presenta muchas similitudes con una obra musical, un libro, una película o un plato de cocina. Su estructura, su ritmo, su estilo y sus ingredientes permiten identificar o, al menos, acercarse mucho, al autor que está detrás. Un *hacker* no es sino una persona que practica la programación informática con un cierto tipo de pasión estética y que se identifica con un cierto tipo de cultura que no se reduce simplemente a una forma de ser, vestir o vivir, sino también a una manera especial de ver el código, de comprender la belleza de su composición y la perfección de su funcionamiento. Para que el código de un programa presente esa belleza sólo tiene que cumplir dos normas elementales: simplicidad y claridad. Si puedes hacer que el ordenador ejecute una orden con una sola instrucción, ¿para qué utilizar cien, o mil? Si puedes encontrar una solución brillante y limpia a un problema concreto, ¿para qué copiar pedazos de código de otras aplicaciones y malcasarlos con remiendos? Si puedes clasificar las funciones una detrás de otra de forma clara, ¿para qué complicar el código con saltos y vueltas que sólo sirven para ralentizar su funcionamiento?"
Matilde Asensi. *El origen perdido* (2003)