F

- Empieza con program nombre_del_programa → No tiene porqué coincidir con el nombre del archivo (esto es indiferente, puede llamarse como [illegible] luego gfortran)

  y termina con end program nombre_del_programa. Un stop acaba el programa
- Para evitar problemas, es mejor no escribir más de 80 caracteres por línea. (tamaño de terminal → 80x24)

  (más bien columna 80)
- NO se distingue entre mayúsculas y minúsculas
- Hay que definir el tipo de todas las variables p.ej.: real :: a, b ; real, dimension(1:3) :: vector → $\mathbb{R}^3$
- Los comentarios, de una línea, empiezan por !
- La asignación de variables se hace con = (Reales → 32 bits, Enteros → 4 bits)
- Funciones simples:
  - Potenciación: **
  - Concatenación de caracteres: //
  - Relaciones matemáticas: ==, <=, >=, /= → distinto
  - Relaciones lógicas:
    - .and.
    - .not.
    - .or.
    - .eqv. → ambas ciertas o ambas falsas
    - .neqv. → lo contrario de .eqv.
- Estructura general:

```
program nombre_del_programa
   [ Declaración de variables
   [ Instrucciones
end program nombre_del_programa
```

- Los print y read son sencillos:

```
read *, var1, var2
print *, var
```

· En lo que respecta a funciones y procedimientos:

```
function nombre(V_entrada_1, ..., V_entrada_n) result (V_salida)
    [declaración de variables → al declarar las de entrada, hay que añadir un intent(in)
    [instrucciones

end function nombre
```

es decir, misma estructura que un programa

```
subroutine nombre(V_entrada_1, ..., V_salida_n)
    [declaración de variables → Como en funciones
                                 intent(in) → sólo de entrada, NO se puede modificar
                                 intent(inout) → de entrada y salida SÍ se puede modificar
                                 intent(out) → sólo de salida !! ¡lo único que se puede hacer es darles
                                 dentro de la subrutina es asignarle un valor, no se puede operar con ellas

    [instrucciones

end subroutine nombre
```

las funciones y subrutinas van al final, tras un contains

las subrutinas se llaman con un call

COMPILAR

· Usaremos un compilador libre, g95. Se llama con F en el servidor y con g95 en casa. (en ocasiones hay que poner -std=F)

La sintaxis es sencillísima: F nombre_del_fichero.f95

esto llama el programa a a.out

Para cambiar el nombre, F -o nombre nombre_del_fichero.f95

llama el programa a nombre.out

RECOMENDACIONES

· Comentar, explicar, aclarar

· Conviene usar espacios en vez de tabuladores

· No conviene usar muchas operaciones complicadas en una misma línea

· No es bueno comprimir el código tanto que sea ilegible; ya se encarga el compilador de optimizar

MATRICES

· En muchas ocasiones nos interesa definir matrices/vectores de tamaño variable. Eso se hace con allocatable: { datos de funciones/subrutinas, se puede asignar tamaño de variables de entrada para matrices locales
{ siempre se puede usar punteros para dimensiones estáticas

↳ Dimensión no definida

```
Real, dimension (:,:), allocatable :: matriz
(...)
allocate (matriz (m, n)) ! A partir de ahora la matriz es m x n
(...)
deallocate (matriz) ! Opcional : elimina la memoria que ocupaba esa matriz
```

↳ Forma con índices: allocate (M(1:3, 2:5))

· Para asignar valores a una matriz de golpe sin ningún do:

```
forall (i=1:m, j=1:n)
      m1(i,j) = i+j
end forall
```

Tb. se puede en una línea: forall (i=1:m, j=1:n) m1(i,j) = i+j

es más eficiente que do. El libro explica sutilezas

· Imprimir matrices es costoso, pues Fortran las almacena por columnas:

```
do i = 1, m        ↳ fila
   print *, (m1(i,j), j=1, n)
end do
```

⟹ Interesante: es un do implícito que sólo se puede usar en prints o reads

KIND

- selected_int_kind(n): indica el kind que puede representar enteros entre -10^n y 10^n ($n \leq 18$)
  selected_real_kind(p): " " " " " " reales con p decimales ($p \leq 18$)
- La utilidad es sobre todo para números grandes: hoy en día es olvidar ahorrar memoria.
- Devuelve -1 si se le pide más que lo que el ordenador puede representar. Hay máquinas ordenadas con mucha precisión.
  Los 32 cifras (el segundo latín)
- El módulo de Moodle usa:

ENTEROS
- int 2 ↔ byte : $\pm 10^2$
- int 4 ↔ short : $\pm 10^4$
- int 8 ↔ int : $\pm 10^8$
- int 10 ↔ long : $\pm 10^{10}$

REALES
- single ↔ sencillo ↔ sp : 7 cifras
- double ↔ doble ↔ dp : 15 cifras

Añadido por mí
- quad ↔ cuad ↔ qp : 33 cifras

derecha portabilidad

- Para asignar kind a parámetros internos, se hace con _ (e.g. 2.0_dp)
- Al convertir tipos, hay un segundo argumento opcional que determina el kind.

Programacion modular: F -c compila módulos (útil para compilar)
La estructura contains no es la forma mas eficiente de programar, ya que requiere que cada programa tenga definidas las funciones y subrutinas. Es mucho mas eficiente definirlas juntas en un modulo, al que haremos referencia en cada programa.
La estrucutra de un modulo es similar a la de un programa, exceptuando las instrucciones:

```
module nombre_modulo

declaracion de variables

contains

funciones y subrutinas

end module nombre_modulo
```

Este module y contains solo puede haber declaración de variables, funcs o subrutinas

Para usar las funciones de un modulo, usaremos el comando use antes de la declaracion de variables:

```
program nombre_programa

use nombre_modulo     !solo puede haber un modulo en cada linea use, pero podemos usar todos los modulos que queramos

declaracion

instrucciones

contains

funciones y variables adicionales

end program
```

Las variables de un modulo pueden ser publicas o privadas. Las variables declaradas como privadas no pueden ser utilizadas en el programa, solo dentro del modulo. Tambien pueden ser publicas o privadas las funciones y subrutinas. Ejemplo:

```
module modulo_1

integer, public :: l,m
real, private :: q
public :: v,w   !nombres de las funciones y subrutinas definidas en el contains

contains

funciones y subrutinas

end module modulo_1
```

Para usar este modulo, lo guardamos en un fichero x.f95. En el propio programa nos referiremos a el por su nombre, pero a la hora de compilar indicaremos el nombre del fichero que lo contiene. Ejemplo de compilacion:

```
F -o Z x.f95 y.f95 !x.f95 contiene el modulo e y.f95 el programa. En caso de tener modulos que necesiten otros modulos, el orden de compilacion sera modulo 1 modulo 2 que contiene a modulo 1 modulo 3 que contiene a ambos programa
```

Importante: no podemos usar el mismo nombre para variables, funciones o subrutinas del programa y el modulo. Al escribir use modulo hemos introducido ya dichos nombres en el programa.

Ejemplo de programa:

```
program nombre_programa
use modulo_1
integer :: i,j,k
real, dimension (:,:) :: matriz
character (len=80) :: line
character (len=*), parameter :: unknown="nothing"    !las cadenas constantes tienen longitud indefinida. Si ponemos len=7 da error
read *, i,j
```

FICHEROS → CAPÍTULO 9

Antes de leer o escribir hay que abrir el archivo externo. Eso se hace con:

open (unit = numero, file = "Nombre", status = "[old, new, replace, scratch]",
action = "[read, write, readwrite]", position = "[rewind, append]", iostat = enum2)

- **Unit** asigna un número a nuestro fichero. **No** se puede usar $<^{5}_{6}$, pues son los reservados a teclado y pantalla.

- **file** es la ruta del archivo

- **status**:
  - **old**: archivo ya existente
  - **new**: crea un nuevo archivo
  - **replace**: abre el archivo si existe y si no lo crea
  - **scratch**: crea un fichero temporal, que desaparece al finalizar el programa. Si se usa esto NO SE USA FILE

- **action**:
  - **read**: sólo lee
  - **write**: sólo escribe
  - **readwrite**: lee y escribe

- **position**:
  - **rewind**: se sitúa en la primera línea del archivo, así que reescribe la información anterior; pero se puede leer
  - **append**: se sitúa al final del archivo: no se puede leer pero no reescribe sobre lo anterior

- **iostat** es una variable de salida que vale **0** si el proceso ha sido satisfactorio

. Una vez hemos abierto, se lee con:

read(unit= numero , fmt=* iostat= numero2)(,) var1, var2, ..., varn

(OPCIONAL: "(,)")
(fmt=*: se puede personalizar pero no lo hemos)

Va como read*, en el sentido de que lee primero a var1, tras un espacio/salto de línea pasa a var2, luego a var3...

de hecho, read* equivale a read(unit=*, fmt=*)

. Y se escribe con:

write(unit=numero, fmt="(Formato1,Formato2,...)", iostat=numero2)(,) var1,var2,...,varn

(OPCIONAL: "(,)")

el comportamiento de las var1, var2... va como print*,

de hecho, print* equivale a write(unit=*, fmt=*)

(print "(fmt)" permite usar formatos)

. Formatos: van separados por comas, aplicándose (salvo los saltos de línea y antes) a las variables por orden. Son:

in: escribe enteros con n caracteres (incluyendo -). Si es más largo, escribe n asteriscos.
n=0 permite tamaños arbitrarios

fn.m: escribe reales con n caracteres (incluyendo el . y el -) y m decimales.
n=0 significa que coja los caracteres necesarios

a: Texto

n(fmt): aplica el formato a n variables seguidas. Útil para vectores, que si no va iterando.
fmt pueden ser una sucesión de formatos diferentes

/ : un salto de línea

// : dos saltos de línea

...

tr n: mueve el cursor n caracteres a la derecha.

```
allocate (matriz(j,-1:i))
call operaciones (i,j,k,line, matriz)
deallocate matriz      !libera la memoria ocupada por esta matriz

contains

subroutine operaciones(i,l,m,xlin,array)
use modulo_3
integer, intent(in) :: i,l,m
real, parameter :: pi=3.1416
real, dimension (:,0:), intent(out) :: array     !obviamente, array debe tener la misma dimension que
matriz, pero podemos cambiar las etiquetas. Escribiendo (:,0:) obetenemos una matriz (j, 0:i+1)
character (len=*), intent(out) :: xli
end subroutine opearaciones

end program nombre_programa

¿Como declaramos una funcion dentro de una subrutina?
subroutine integra (func,a,b)
integer, intent(in) :: a,b
interface
  function func(y) result (resultado)
     real, intent(in) :: y
     real :: resultado
  end function func
end interface

Dentro de la estructura interface solo hay declaracion de variables de entrada y salida de la funcion,
no la propia funcion.
Asi, la subrutina integra admite como input cualquier funcion real de una variable.
```

Sólo se pone:
*Variables de entrada, con tipo y kind correcto
*Variables de salida, con tipo y kind correcto

NO se ponen ni variables locales ni instrucciones

La interface sólo refleja la declaración de lo que estoy pasando a la subrutina

```
interface

function f(x) result (res)
Real (kind=8), intent(in) :: x
Real (kind=8) :: res

end function f

end interface
```

Nótese que en la función original las variables de entrada y salida no tienen por qué llamarse x y res.