Introducción a Fortran95

Diego Rosales

Instituto de Física de Líquidos y Sistemas Biológicos.

IFLYSIB
CONICET
UNLP

Departamento de Ciencias Básicas, Fac. de Ingeniería, UNLP

FACULTAD DE INGENIERÍA | UNIVERSIDAD NACIONAL DE LA PLATA

Herramientas **C**omputacionales para **C**ientíficas/os
Curso 2024

PARTE 2

PROCEDIMIENTOS EXTERNOS

Procedimientos externos

Características

- Unidades-programa “pequeñas” y reutilizables, independientes de programa principal, cuya función es realizar subtareas individuales.
- Cada unidad de programa se puede depurar y probar individualmente.

FORTRAN permite dos tipos de procedimientos externos

- Subrutinas
- Funciones

Procedimientos externos

Características

- Unidades-programa “pequeñas” y reutilizables, independientes de programa principal, cuya función es realizar subtareas individuales.
- Cada unidad de programa se puede depurar y probar individualmente.

FORTRAN permite dos tipos de procedimientos externos

- Subrutinas
- Funciones

Subrutinas

Una subrutina es un bloque de código que, toma uno o más valores de entrada, para realizar un conjunto de acciones y, potencialmente, modificar los valores de los argumentos. La estructura de una subrutina es:

Subrutina: Forma general

```
SUBROUTINE NombreSubrutina(argumentos)
    ! Declaraciones variables locales

    ! Acciones y procesamiento

END SUBROUTINE NombreSubrutina
```

¿Cómo usarla?

Programa

```
program main_program
integer :: a,b,c
interface
  !declaraciones
endinterface
c=a+b
call nombre_subroutina(lista_argumentos)
end program
```

Subrutina

```
subroutine nombre_subrutina(lista_argumentos)
  !sección de declaración de variables
  !sección de ejecución
  ...
  return
end subroutine nombre_subrutina
```

Subrutinas

Programa sin subrutina

```
program abc
implicit none
real :: a, b, c, temp
read(*,*) a, b
temp = a**2 + b**2
c = sqrt(temp)
write(*,*) c
end program
```

Programa con subrutina

```
program abc
implicit none
real :: a, b, c
interface
endinterface
read(*,*) a, b
call hypotenuse(a,b,c)
write(*,*) c
end program
```

```
subroutine hypotenuse(a,b,c)
real, intent(in) :: a, b
real, intent(out) :: c
real :: temp
temp = a**2 + b**2
c = sqrt(temp)
return
end subroutine
```

Código: 1-subrutina.f95

Atributo INTENT

Programa

```
subroutine hypotenuse(a,b,c,d)
real,intent(in) :: a, b
real,intent(out):: c
real,intent(inout) :: d
.
.
```

Los argumentos de las subrutinas (y funciones) deben tener el atributo INTENT declarado

- INTENT(in): el argumento es utilizado únicamente para pasar datos a la subrutina y no puede ser modificado en la subrutina
- INTENT(out): el argumento es utilizado únicamente para retornar un resultado
- INTENT(inout): el argumento es utilizado tanto para pasar datos a la subrutina como para retornar un resultado.

Funciones

Una función es un bloque de código que toma uno o más valores de entrada, realiza algún procesamiento y devuelve un resultado. En Fortran, las funciones se definen de la siguiente manera:

Función

```
FUNCTION NombreFuncion(argumentos)
    ! Declaraciones locales

    ! Cálculos y procesamiento

    NombreFuncion = resultado
END FUNCTION NombreFuncion
```

Funciones

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

Ejemplo definición de una función

```
REAL FUNCTION Panqueque(X,Y)
IMPLICIT NONE
REAL, INTENT(IN) :: X,Y
REAL :: Z !variable interna del la función
Z = X**2 + Y**2
Panqueque = X + Y + Z
END FUNCTION Panqueque
```

X e Y son las variables de entrada, Z es una variable interna auxiliar, y **Panqueque** es el valor que tomará la función y no es necesario declarar que es de salida.

Funciones

Sin función

```
program abc
implicit none
real :: a, b, c, temp
read(*,*) a, b
temp = a**2 + b**2
c = sqrt(temp)
write(*,*) c
end program
```

Con función

```
program abc
implicit none
real :: a, b, c
interface
endinterface
read(*,*) a, b
c=hypotenuse(a,b)
write(*,*) c
end program
```

```
real hypotenuse(a,b)
real, intent(in) :: a, b
real :: temp
temp = a**2 + b**2
hypotenuse = sqrt(temp)
return
end function
```

Código: 2-funcion.f95

Interfaces

El bloque INTERFACE en Fortran se utiliza para declarar interfaces explícitas para procedimientos, permitiendo que el compilador tenga conocimiento sobre cómo deben ser llamados, los tipos de datos de los argumentos, y otras características importantes.

- **Declarar Procedimientos Externos:** Si tienes una subrutina o función en otro archivo o módulo, puedes usar un bloque INTERFACE para informar al compilador sobre su existencia y sus parámetros.
- **Sobrecarga de Procedimientos:** Permite definir múltiples procedimientos con el mismo nombre pero diferentes argumentos (sobrecarga).
- **Procedimientos Genéricos:** Puedes usar INTERFACE para crear procedimientos genéricos que pueden trabajar con diferentes tipos de datos.

Interfaces

Al trabajar con procedimientos externos se debe incluir en el programa que realiza la llamada un bloque INTERFACE, para detectar incompatibilidades de tipos.

Programa principal

```
program trazando
 implicit none
 real,dimension(3,3):: w
 real::res1
 interface
  subroutine traza(a,suma)
   real,intent(in) :: a(3,3)
   real,intent(out):: suma
  endsubroutine traza
 endinterface

 call traza(w,res1)
 write (*,*) res1
endprogram trazando
```

subrutina

```
subroutine traza(a,suma)
 implicit none
 real,intent(in):: a(3,3)
 real,intent(out):: suma
 integer:: i

 suma = 0.0
 do i = 1, 3
    suma = suma + a(i,i)
 enddo
endsubroutine traza
```

Código: 3-interface.f95

Interfaces con funciones

Programa principal

```
program trazando
 implicit none
 real :: tata
 real,dimension(3,3):: w
interface
 function traza(a,n)
 real:: traza
 integer,intent(in):: n
 real,intent(in):: a(n,n)
 endfunction traza
endinterface
 .
 w=1.0
 !uso la función
 tata=traza(w,3)/2.0
 write (*,*) tata
endprogram trazando
```

función

```
function traza(a,n)
 implicit none
 real:: traza
 integer,intent(in):: n
 real,intent(in):: a(n,n)
 integer:: i
 real:: suma

 suma = 0.0
 do i = 1, n
    suma = suma + a(i,i)
 enddo
 traza = suma
endfunction traza
```

Recomendaciones sobre procedimientos

- Siempre usar `interface` en los programas que llaman
- Para que un argumento de tipo arreglo pueda ser de cualquier tamaño usar `dimension(:)`. Se puede usar `size` para determinar el tamaño.
- Para que un carácter pueda ser de cualquier tamaño usar `character(len=*)`. Se puede usar `len` para determinar el tamaño.

función

```
real function sum_elements(arr)
real, dimension(:), intent(in) :: arr
do i = 1, size(arr)
    sum_elements = sum_elements + arr(i)
end do
end function sum_element
```

Ámbito de variables y tipos de argumentos

Fortran maneja los objetos (valiables) por referencia y no por valor, como lo hacen otros lenguajes de programación. El manejo de objetos por referencia significa que el programa, cuando se ejecuta, trabaja únicamente con el argumento (de uso).

- Las variables internas no interfieren y se reinician en cada llamada
- Los argumentos pasados por referencia sí interfieren

REFERENCIA: son aquellos argumentos que al ser modificados en un procedimiento se ven modificados en el programa que lo llamó.

VALOR: son aquellos argumentos que al ser modificados en un procedimiento no se ven alterados en el programa que lo llamó.

Ejemplos

```
call fulano(a)
b=factorialde(n)
```

Ejemplos

```
call fulano(1.2)
a=factorialde(n+3)
```

- Declarar siempre el `intent` (in,out,inout) de los argumentos
- Para las funciones sólo usar `intent(in)`

Arreglos como argumentos

El tamaño de los arreglos pasados a una función o subrutina puede o no especificarse. Pero siempre se especifica el número de índices.

- Forma fija

```
subroutine traza(a)
real,dimension(3,3):: a
```

- Forma ajustable (se pasan las dimensiones como argumentos)

```
subroutine traza(a,n)
integer:: n
real,dimension(n,n):: a
```

- Forma asumida (las dimensiones no se especifican)

```
subroutine traza(a)
real,dimension(:,:):: a
```

Pero hay que usar `interface`

```
interface
subroutine traza(a)
 real,dimension(:,:):: a
end subroutine sub
end interface
```

Módulos

Los módulos son una estructura muy importante a partir de Fortran 90 porque permite compartir variables y procedimientos, facilitando tanto una transferencia sencilla de variables como la creación de librerías propias.
Un módulo, como unidad programática independiente, puede contener la información o los elementos siguientes:

- Precisión utilizada, si es simple o doble precisión.
- Definición de constantes y variables compartidas.
- Interfaces de procedimientos.
- Procedimientos de módulos.

Estructra sintáxica

```
MODULE <NOMBRE>
: !instrucciones de especificación o declaración
contains
: !procedimientos de modulo
END MODULE <NOMBRE>
```

Variables globales y módulos

MODULO:

```
module globales
real,parameter :: pi =3.14159,&
grados= pi/180.0
real , dimension (2) :: x
endmodule globales
```

SUBRUTINA:

```
subroutine f(r,a)
use globales
implicit none
real,intent(in) :: r,a
x(1)=r*cos(a*grados)
x(2)=r*sin(a*grados)
end subroutine f
```

PROGRAMA:

```
program test
use globales
implicit none

interface
 subroutine f(r,a)
 use globales
 real,intent(in) :: r,a
 endsubroutine f
endinterface

x=4.0
write(*,*) x,pi,grados
call f(2.0,45.0)
write(*,*) x
endprogram test
```

Código: 4-modulo-subrutina.f95

Variables globales y módulos

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

MODULO:

```
module variables
 implicit none
 real, allocatable :: a(:),b(:)
end module variables
```

SUBRUTINA:

```
subroutine asignar(n)
 use variables, only : a
 implicit none
 integer :: n
 a=(/ (i*2.0 8, i=1,n) /)
end subroutine asignar
```

Procedimientos y módulos

Además de definir variables, en un módulo se pueden incluir procedimientos (funciones y/o subrutinas) que luego se pueden de forma sencilla con otras partes del programa, funciones o subrutinas.

MODULO

```
module globales
 real,parameter,private::&
      pi=3.141592,&
      grados=pi/180.0
 real,dimension(2):: x

contains
 subroutine f(r,a)
  implicit none
  real,intent(in):: r,a

  x(1)=r*cos(a*grados)
  x(2)=r*sin(a*grados)
 endsubroutine f
endmodule globales
```

PROGRAMA:

```
program test
 use globales
 implicit none
 real:: pi

 pi=5.0
 x=4.0
 write (*,*) x,pi
 call f(2.0,45.0)
 write (*,*) x
endprogram test
```

Código 4-modulo-función.f95

Comentarios sobre procedimientos y módulos

- Definición de funciones internas
- Funciones externas pueden llamar a otras
- Funciones de módulo no necesitan `interface`
- `save,private` (Código 5-save_public_private.f95)
- `COMMON` (Código 6-common.f95)
- `use globales, only: x`
- Procedimientos como argumentos
- Sobrecarga (Código 7-concatena.f95)
- Recursión

Ejercicio reloaded

Ejemplo

Modificar el programa hecho en el Ejercicio Propuesto de modo que las operaciones de lectura de matrices, de multiplicación y de determinación de positivo sean hechas por procedimientos apropiados.

- Tipos de procedimientos: subrutinas y funciones
- Paso de información desde y hacia procedimientos
- Ámbito de las variables

Recapitulación

- Funciones
- Subrutinas
- Argumentos (referencia, valor)
- Ámbito de variables (locales, globales)
- Módulos (variables, procedimientos)

Ejercitación

- Probar los ejemplos dados
- Inventar variantes: funciones que llaman a funciones, usar globales en lugar de argumentos, etc.
- Resolver el Ejercicio 2
- Intentar crear un módulo con funciones que determinen (a) el máximo, (b) el mínimo y (c) la suma de los elemento de un vector de cualquier tamaño.

Variables derivadas:TYPE

TYPE se utiliza para definir un nuevo tipo de datos, llamado “tipo de datos derivado” o “tipo de datos compuesto”. Este tipo de datos permite agrupar diferentes tipos de datos bajo un solo nombre, facilitando la creación de estructuras más complejas.

Uso de TYPE

```
TYPE vehiculo
  REAL :: masa
  REAL :: ruedas
  REAL :: potencia
END TYPE vehiculo
```

Código: 08-type_variables.f95

Bibliografía

- ADAMS, J.C., BRAINERD, W.S., MARTIN, J.T. SMITH, B.T. y WAGENER, J.L. (1997) “Fortran 95 Handbook. Complete ISO/ANSI Reference” MIT Press
- BRAINERD, W.S., GOLDBERG, C.H. y ADAMS, J.C. (1996) “Programmer’s Guide to Fortran 90” McGraw-Hill
- HAHN, B.D. (1997) "Fortran 90 for Scientists and Engineers.Arnold
- METCALF, M. y REID, J. (1996) “FORTRAN 90/95 Explained” Oxford University Press
- METCALF, M. y REID, J. (1999) “FORTRAN 90/95 Explained” Second Edition, Oxford University Press
- METCALF, M., REID, J. y COHEN, M. (2004) “FORTRAN 95/2003 Explained” Oxford University Press
- REDWINE, C. (1995) “Upgrading to Fortran 90” Springer Verlag
- Página web principal: http://www.fortran.com
- PRESS, W.H., TEUKOLSKY, S.A., VETTERLING, W.T. y FLANNERY, B.P. (1996) “Numerical Recipes in Fortran 90. The Art of Parallel Scientific Computing” Second Edition, Cambridge University Press

Si queda tiempo...

Más sobre asignación dinámica: Punteros

- Un puntero es una variable que apunta a un objetivo de su mismo tipo.
- Hay que considerarlos como un “alias”.
- Los usos más importantes para un puntero son:
  - Dar una alternativa mejor a la reserva dinámica de memoria.
  - Crear y manipular listas y otras estructuras de datos dinámicas.
- La forma general de declarar un puntero es:
  `TIPO, POINTER :: nombre_puntero`
- Si el puntero apunta a una variable, esta debe ser declarada como TARGET, y tiene que ser del mismo tipo que el puntero. La sentencia de asignación para los punteros es la siguiente: `POINTER=>TARGET`

Ejemplo: podemos crear un puntero que apunte a un array de una dimensión (por ejemplo, a una columna de un array de dos dimensiones)

Ejemplo

```
REAL, DIMENSION ( : ) , POINTER :: P
REAL, DIMENSION (5, 0:6) , TARGET :: A
P => A(3, :)
```

Ventajas del Uso de Punteros

- Modificación Dinámica: los punteros permiten cambiar a qué variable apunta en tiempo de ejecución. Útil para alterar la referencia de datos en un programa sin tener que modificar otras partes del código.
- Acceso a Diferentes Variables: Un puntero puede apuntar a diferentes variables en diferentes momentos.
- Manipulación de Datos: Permite manipular datos de forma indirecta a través del puntero, sin necesidad de pasar múltiples variables entre procedimientos.

Ejemplo

```
program punteros1
REAL,TARGET :: B,C
REAL, POINTER :: pB
pB => B
B = 10.0
WRITE(*,*)pB,B,C
pB=>C
pB = 32.0
WRITE(*,*)pB,B,C
end program punteros1
```

Salida

```
A la salida tenemos
primero: 10.0,10.0,0.0
luego:   32.0,10.0,32.0
```

Código 09-punteros1.f95

Más sobre asignación dinámica: Punteros

Sentencias para punteros: ALLOCATE y DEALLOCATE

- Son la sentencia ALLOCATE, creamos espacio de memoria para un valor y hacemos que el puntero se refiera a ese espacio.
- La sentencia DEALLOCATE, desecha el espacio de memoria al que apunta su argumento y hace que su objetivo sea nulo.

NULLIFY

- Para crear un puntero nulo, que se llama, utilizamos la sentencia NULLIFY, que consta de la palabra NULLIFY seguida por el nombre de la variable puntero entre paréntesis (desasociar un puntero de su objetivo)

ASSOCIATED

- La función ASSOCIATED(P1), me indica si el puntero P1, que tiene que estar definido, apunta a algún objeto. Esta función tiene otras dos aplicaciones:
- ASSOCIATED(P1, R), me indica si el puntero P1 es el alias del objetivo R.
- ASSOCIATED(P1, P2), me indica si los punteros P1 y P2 tienen el mismo objetivo, o si ambos son nulos

Polimorfismo en Tipos Derivados

- **Polimorfismo:** La capacidad de una función o procedimiento para operar de manera diferente dependiendo del tipo de datos con los que se invoca.
- **Tipos Derivados:** Cómo los tipos derivados extienden los tipos base y pueden tener procedimientos específicos.
- **Procedimientos Asociados:** Cómo los procedimientos asociados en tipos derivados pueden ser implementaciones concretas de una interfaz de procedimiento definida en el tipo base.

`Código: 10-polimorfismo.f95`

función System()

La función “system()” pasa una cadena de caracteres a su shell como entrada, como si la cadena se hubiera escrito por liena de comandos.

Ejemplo

```
program usandosystem
implicit none
INTEGER :: i,j
REAL :: A,B,Y
A=0.0
B=0.5
OPEN(unit=40,file='test.dat')
DO I=1,20
      Y=A+B*I**2
      WRITE(40,*)I,Y
ENDDO
CLOSE(40)
! call system("gnuplot gnuplot_template.gp")
end program usandosystem
```

Ver código 11-system-a.f95

Creación y Uso de Objetos .o en Fortran

¿Qué es un archivo .o?

- **Código Compilado:** Un archivo .o contiene el código fuente de un módulo Fortran compilado a un formato intermedio, comprensible por el *linkeador*.
- **Unidad de Compilación:** Cada archivo .o representa una unidad de compilación independiente.
- **Enlace:** Los archivos .o se enlazan posteriormente para formar un ejecutable.

¿Por qué usar archivos .o?

- **Modularidad:** Permite organizar el código en módulos más pequeños.
- **Reutilización:** Los archivos .o pueden reutilizarse en diferentes proyectos.
- **Optimización:** Permite compilar y enlazar solo los módulos que han cambiado.
- **Depuración:** Facilita la identificación y corrección de errores en módulos.

`Ver ejemplos`

Proceso de Creación y Enlace

Compilación Individual

- Se compila cada módulo Fortran con la opción -c para generar el archivo .o correspondiente.
- Ejemplo: `gfortran -c ejemplo3_a.f95`

Enlace

- Los archivos .o se enlazan utilizando el compilador, especificando todos los archivos .o necesarios y el nombre del ejecutable final.
- Ejemplo: `gfortran ejemplo3_a.o ejemplo3_b.o -o mi_programa`

Ejemplo

```
# Compilar los módulos por separado
gfortran -c modulo1.f90
gfortran -c modulo2.f90
gfortran -c programa_principal.f90
# Enlazar para crear el ejecutable
gfortran modulo1.o modulo2.o programa_principal.o -o salida
```