EBook Gratis

APRENDIZAJE Fortran

Free unaffiliated eBook created from **Stack Overflow contributors.**

#fortran

Tabla de contenido

Acerca de 1

Capítulo 1: Empezando con Fortran 2

Observaciones 2

Versiones 2

Examples 2

Instalación o configuración 2

Hola Mundo 3

Ecuación cuadrática 4

Insensibilidad a la caja 5

Capítulo 2: Alternativas modernas a las características históricas. 6

Examples 6

Tipos de variables implícitas 6

Aritmética si declaración 7

Construcciones de OD no en bloque 8

Retorno alternativo 8

Forma de fuente fija 10

Bloques comunes 12

Asignado GOTO 13

GOTO computado 14

Especificadores de formato asignado 14

Funciones de sentencias 15

Capítulo 3: Arrays 17

Examples 17

Notación básica 17

Arreglos asignables 18

Constructores de matrices 18

Especificación de la naturaleza de la matriz: rango y forma 20

Forma explícita 21

Forma asumida 21

Tamaño asumido 22

Forma diferida....22

Forma implícita....22

Arreglos enteros, elementos de arreglo y secciones de arreglo....23

Arreglos enteros....23

Elementos de matriz....23

Secciones de matriz....23

Componentes de matrices de matrices....24

Operaciones de matriz....25

Adición y sustracción....25

Función....25

Multiplicación y división....26

Operaciones matriciales....26

Secciones de matrices avanzadas: tripletes de subíndices y subíndices vectoriales....26

Tripletes de subíndices....27

Subíndices vectoriales....27

Secciones de matriz de rango más alto....28

Capítulo 4: Control de ejecución....29

Examples....29

Si construir....29

SELECCIONAR CASO de construcción....30

Construcción de bloques DO....31

Donde construir....33

Capítulo 5: Extensiones de archivo fuente (.f, .f90, .f95, ...) y cómo se relacionan con e....35

Introducción....35

Examples....35

Extensiones y significados....35

Capítulo 6: I / O....37

Sintaxis....37

Examples....37

E / S simple....37

Leer con un poco de comprobación de errores....37

Pasando argumentos de línea de comando....38

Capítulo 7: Interfaces explícitas e implícitas 41
Examples 41
Subprogramas internos / módulos e interfaces explícitas 41
Subprogramas externos e interfaces implícitas 42
Capítulo 8: Interoperabilidad C 44
Examples 44
Llamando C desde Fortran 44
C structs en fortran 45
Capítulo 9: Procedimientos - Funciones y subrutinas. 46
Observaciones 46
Examples 46
Sintaxis funcional 46
Declaración de devolución 47
Procedimientos recursivos 47
La intención de los argumentos ficticios 48
Haciendo referencia a un procedimiento 49
Capítulo 10: Procedimientos intrínsecos 52
Observaciones 52
Examples 52
Uso de PACK para seleccionar elementos que cumplan una condición 52
Capítulo 11: Programación orientada a objetos 54
Examples 54
Definición de tipo derivado 54
Tipo de Procedimientos 54
Tipos derivados abstractos 55
Extensión de tipo 56
Tipo constructor 57
Capítulo 12: Tipos de datos 59
Examples 59
Tipos intrinsecos 59
Tipos de datos derivados 60
Precisión de los números en coma flotante 61

Parámetros de tipo de longitud asumida y diferida 63
Constantes literales 64
Acceso a subcadenas de caracteres 66
Accediendo a componentes complejos. 67
Declaración y atributos. 67
Capítulo 13: Unidades de programa y diseño de archivos. 69
Examples 69
Programas de fortran 69
Módulos y submódulos. 70
Procedimientos externos 70
Bloquear unidades de programa de datos. 71
Subprogramas internos 71
Archivos de código fuente 72
Capítulo 14: Uso de módulos 74
Examples 74
Sintaxis del modulo 74
Usando módulos de otras unidades de programa 74
Módulos intrínsecos 76
Control de acceso 76
Entidades de módulo protegido 78
Creditos 80

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: fortran

It is an unofficial and free Fortran ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Fortran.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Fortran

Observaciones

Fortran es un lenguaje utilizado ampliamente en la comunidad científica debido a su idoneidad para el cálculo numérico. Particularmente atractiva es su intuitiva notación matricial, que facilita la escritura de cómputos vectorizados rápidos.

A pesar de su edad, Fortran todavía está activamente desarrollado, con numerosas implementaciones, incluyendo GNU, Intel, PGI y Cray.

Versiones

Versión	Nota	Lanzamiento
FORTRAN 66	Primera estandarización por ASA (ahora ANSI)	1966-03-07
FORTRAN 77	Forma fija, histórica	1978-04-15
Fortran 90	Forma libre, norma ISO, operaciones matriciales.	1991-06-15
Fortran 95	Procedimientos puros y elementales	1997-06-15
Fortran 2003	Programación orientada a objetos	2004-04-04
Fortran 2008	Co-matrices	2010-09-10

Examples

Instalación o configuración

Fortran es un lenguaje que puede compilarse utilizando compiladores suministrados por muchos proveedores. Diferentes compiladores están disponibles para diferentes plataformas de hardware y sistemas operativos. Algunos compiladores son software libre, algunos se pueden utilizar de forma gratuita y otros requieren la compra de una licencia.

El compilador gratuito más común de Fortran es GNU Fortran o gfortran. El código fuente está disponible en GNU como parte de GCC, la colección del compilador GNU. Los binarios para muchos sistemas operativos están disponibles en https://gcc.gnu.org/wiki/GFortranBinaries . Las distribuciones de Linux a menudo contienen gfortran en su gestor de paquetes.

Otros compiladores están disponibles por ejemplo:

- EKOPath por PathScale
- LLVM (backend vía DragonEgg)

- Oracle Developer Studio

- Absoft Fortran Compiler
- Compilador Intel Fortran
- Compilador NAG Fortran
- Compiladores de IGP

En los sistemas HPC, a menudo hay compiladores especializados disponibles por el proveedor del sistema como, por ejemplo, los compiladores IBM o Cray .

Todos estos compiladores soportan el estándar Fortran 95. ACM Fortran Forum ofrece una descripción general sobre el estado de Fortran 2003 y el estado de Fortran 2008 de varios compiladores y está disponible en la Wiki de Fortran.

Hola Mundo

Cualquier programa de Fortran tiene que incluir el `end` como última declaración. Por lo tanto, el programa Fortran más simple se ve así:

```
end
```

Aquí hay algunos ejemplos de programas "hola mundo":

```
print *, "Hello, world"
end
```

Con declaración de `write` :

```
write(*,*) "Hello, world"
end
```

Para mayor claridad, ahora es común usar la declaración del `program` para iniciar un programa y darle un nombre. La declaración `end` puede referirse a este nombre para que sea obvio a qué se refiere, y permitir que el compilador verifique que el código sea correcto. Además, todos los programas de Fortran deben incluir una declaración `implicit none` . Por lo tanto, un programa mínimo de Fortran debería tener el siguiente aspecto:

```
program hello
  implicit none
  write(*,*) 'Hello world!'
end program hello
```

El siguiente paso lógico a partir de este punto es cómo ver el resultado del programa hello world. Esta sección muestra cómo lograr eso en un entorno similar a Linux. Suponemos que tiene algunas nociones básicas de los comandos de shell , principalmente sabe cómo llegar al terminal de shell. También asumimos que ya ha configurado su entorno `fortran` . Utilizando su editor de texto preferido (notepad, notepad ++, vi, vim, emacs, gedit, kate, etc.), guarde el programa de saludo de arriba (copie y pegue) en un archivo llamado `hello.f90` en su directorio de inicio.
`hello.f90`

es su archivo fuente. Luego vaya a la línea de comandos y navegue hasta el directorio (¿directorio de inicio?) Donde guardó su archivo fuente, luego escriba el siguiente comando:

```
>gfortran -o hello hello.f90
```

Acabas de crear tu programa ejecutable hello world. En términos técnicos, acaba de compilar su programa. Para ejecutarlo, escriba el siguiente comando:

```
>./hello
```

Debería ver la siguiente línea impresa en su terminal shell.

```
> Hello world!
```

Enhorabuena, acaba de escribir, compilar y ejecutar el programa "Hello World".

Ecuación cuadrática

Hoy en día Fortran se utiliza principalmente para el cálculo numérico. Este ejemplo muy simple ilustra la estructura básica del programa para resolver ecuaciones cuadráticas:

```
program quadratic
  !a comment

  !should be present in every separate program unit
  implicit none

  real :: a, b, c
  real :: discriminant
  real :: x1, x2

  print *, "Enter the quadratic equation coefficients a, b and c:"
  read *, a, b, c

  discriminant = b**2 - 4*a*c

  if ( discriminant>0 ) then

    x1 = ( -b + sqrt(discriminant)) / (2 * a)
    x2 = ( -b - sqrt(discriminant)) / (2 * a)
    print *, "Real roots:"
    print *, x1, x2

    ! Comparison of floating point numbers for equality is often not recommended.
    ! Here, it serves the purpose of illustrating the "else if" construct.
  else if ( discriminant==0 ) then

    x1 = - b / (2 * a)
    print *, "Real root:"
    print *, x1
  else

    print *, "No real roots."
  end if
```

```
end program quadratic
```

Insensibilidad a la caja

Las letras mayúsculas y minúsculas del alfabeto son equivalentes en el conjunto de caracteres de Fortran. En otras palabras, Fortran no *distingue entre mayúsculas y minúsculas* . Este comportamiento está en contraste con los lenguajes que distinguen entre mayúsculas y minúsculas, como C ++ y muchos otros.

Como consecuencia, las variables `a` y `A` son la misma variable. En principio se podría escribir un programa de la siguiente manera

```
pROgrAm MYproGRaM
..
enD mYPrOgrAM
```

Es bueno para el programador evitar tales elecciones feas.

Lea Empezando con Fortran en línea: https://riptutorial.com/es/fortran/topic/904/empezando-con-fortran

Capítulo 2: Alternativas modernas a las características históricas.

Examples

Tipos de variables implícitas

Cuando Fortran se desarrolló originalmente, la memoria era muy importante. Las variables y los nombres de los procedimientos podrían tener un máximo de 6 caracteres, y las variables a menudo se *escribían de forma implícita* . Esto significa que la primera letra del nombre de la variable determina su tipo.

- variables que comienzan con i, j, ..., n son `integer`
- todo lo demás (a, b, ..., h, yo, p, ..., z) son `real`

Programas como los siguientes son aceptables Fortran:

```
program badbadnotgood
  j = 4
  key = 5 ! only the first letter determines the type
  x = 3.142
  print*, "j = ", j, "key = ", key, "x = ", x
end program badbadnotgood
```

Incluso puede definir sus propias reglas implícitas con la declaración `implicit` :

```
! all variables are real by default
implicit real (a-z)
```

o

```
! variables starting with x, y, z are complex
! variables starting with c, s are character with length of 4 bytes
! and all other letters have their default implicit type
implicit complex (x,y,z), character*4 (c,s)
```

La tipificación implícita ya no se considera la mejor práctica. Es muy fácil cometer un error al utilizar la escritura implícita, ya que los errores tipográficos pueden pasar desapercibidos, por ejemplo,

```
program oops
  real :: somelongandcomplicatedname

  ...

  call expensive_subroutine(somelongandcomplEcatedname)
end program oops
```

Este programa estará felizmente ejecutado y hará lo incorrecto.

Para desactivar la escritura implícita, se puede usar la instrucción `implicit none` .

```
program much_better
  implicit none
  integer :: j = 4
  real :: x = 3.142
  print*, "j = ", j, "x = ", x
end program much_better
```

Si no hubiéramos usado `implicit none` en el programa `oops` arriba, el compilador se habría dado cuenta de inmediato y produjo un error.

Aritmética si declaración

La instrucción aritmética `if` permite usar tres ramas dependiendo del resultado de una expresión aritmética

```
if (arith_expr) label1, label2, label3
```

Esta instrucción `if` transfiere el flujo de control a una de las etiquetas de un código. Si el resultado de `arith_expr` es negativo, `label1` está involucrado, si el resultado es cero, se usa `label2` y si el resultado es positivo, se aplica la última `label3` . Aritmética `if` requiere las tres etiquetas, pero permite la reutilización de las etiquetas, por lo tanto, esta declaración se puede simplificar a una rama dos `if` .

Ejemplos:

```
if (N * N - N / 2) 130, 140, 130

if (X) 100, 110, 120
```

Ahora esta función es obsoleta con la misma funcionalidad ofrecida por el `if` declaración y `if-else` construcción. Por ejemplo, el fragmento.

```
    if (X) 100, 110, 120
100 print*, "Negative"
    goto 200
110 print*, "Zero"
    goto 200
120 print*, "Positive"
200 continue
```

puede ser escrito como la construcción `if-else`

```
if (X<0) then
  print*, "Negative"
else if (X==0) then
  print*, "Zero"
```

```
else
  print*, "Positive"
end if
```

Un sustituto `if` de

```
    if (X) 100, 100, 200
100 print *, "Negative or zero"
200 continue
```

tal vez

```
if (X<=0) print*, "Negative or zero"
```

Construcciones de OD no en bloque

La no-bloque `do` construir parece

```
    integer i
    do 100, i=1, 5
100 print *, i
```

Es decir, donde la instrucción de terminación etiquetada no es una instrucción de `continue` . Hay varias restricciones en la declaración que pueden usarse como la declaración de terminación y todo es generalmente muy confuso.

Dicha construcción no de bloque se puede reescribir en forma de bloque como

```
    integer i
    do 100 i=1,5
      print *, i
100 continue
```

o mejor, usando una sentencia de `end do` terminación,

```
integer i
do i=1,5
  print *, i
end do
```

Retorno alternativo

El rendimiento alternativo es una facilidad para controlar el flujo de ejecución en el retorno desde una subrutina. A menudo se utiliza como una forma de manejo de errores:

```
real x

call sub(x, 1, *100, *200)
print*, "Success:", x
stop
```

```
100 print*, "Negative input value"
stop

200 print*, "Input value too large"
stop

end

subroutine sub(x, i, *, *)
  real, intent(out) :: x
  integer, intent(in) :: i
  if (i<0) return 1
  if (i>10) return 2
  x = i
end subroutine
```

El retorno alternativo está marcado por los argumentos `*` en la lista de argumentos ficticios de subrutina.

En la declaración de `call` anterior `*100` y `*200` refieren a las declaraciones etiquetadas `100` y `200` respectivamente.

En la subrutina, las declaraciones de `return` correspondientes a la devolución alternativa tienen un número. Este número no es un valor de retorno, pero denota la etiqueta proporcionada a la cual se pasa la ejecución en el retorno. En este caso, el `return 1` pasa la ejecución a la declaración etiquetada `100` y el `return 2` ejecución pasa a la declaración etiquetada `200` . Una declaración de `return` sin adornos, o la finalización de la ejecución de subrutina sin una declaración de `return` , la ejecución de Passess inmediatamente después de la instrucción de llamada.

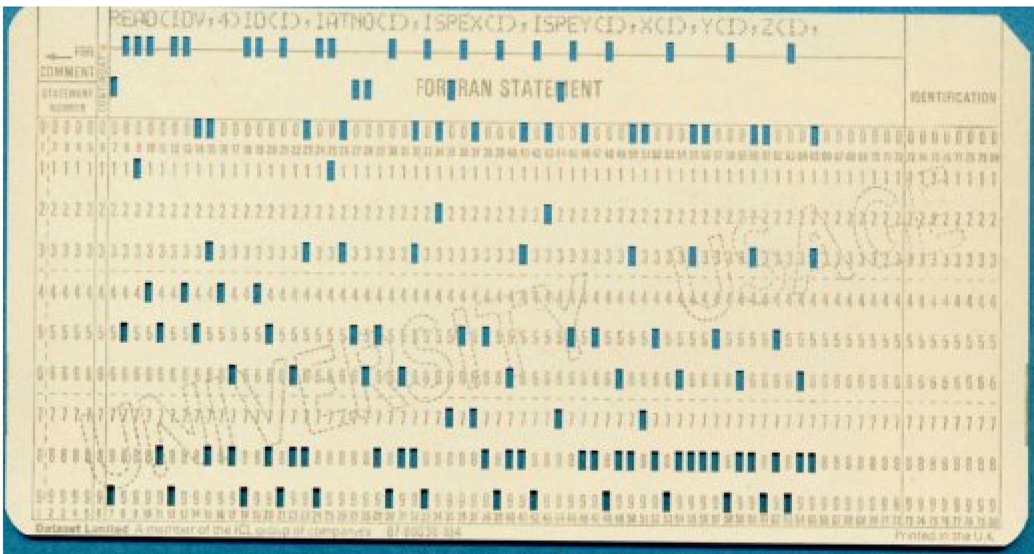
La sintaxis alternativa de retorno es muy diferente de otras formas de asociación de argumentos y la instalación introduce un control de flujo contrario a los gustos modernos. El control de flujo más agradable se puede administrar con la devolución de un código de "estado" entero.

```
real x
integer status

call sub(x, 1, status)
select case (status)
case (0)
  print*, "Success:", x
case (1)
  print*, "Negative input value"
case (2)
  print*, "Input value too large"
end select

end

subroutine sub(x, i, status)
  real, intent(out) :: x
  integer, intent(in) :: i
  integer, intent(out) :: status

  status = 0
```

```
  if (i<0) then
    status = 1
  else if (i>10)
    status = 2
  else
    x = i
  end if

end subroutine
```

Forma de fuente fija

Fortran originalmente fue diseñado para un formato de formato fijo basado en una tarjeta perforada de 80 columnas:



Sí: esta es una línea del propio código del autor.

Estos fueron creados en una máquina perforadora de tarjetas, muy similar a esto:

Las imágenes son fotografías originales del autor.

El formato, como se muestra en la tarjeta de muestra ilustrada, tenía las primeras cinco columnas reservadas para las etiquetas de estados de cuenta. La primera columna se usó para denotar los comentarios mediante una letra **C.** La sexta columna se usó para denotar una continuación de la

declaración (insertando cualquier carácter que no sea un cero '0'). Las últimas 8 columnas se usaron para la identificación y secuenciación de las tarjetas, ¡lo cual fue muy valioso si dejaste caer el mazo de cartas en el suelo! La codificación de caracteres para tarjetas perforadas tenía solo un conjunto limitado de caracteres y solo estaba en mayúsculas. Como resultado, los programas de Fortran se parecían a esto:

```
      DIMENSION A(10)                                                    00000001
C THIS IS A COMMENT STATEMENT TO EXPLAIN THIS EXAMPLE PROGRAM            00000002
      WRITE (6,100)                                                      00000003
 100  FORMAT(169HTHIS IS A RATHER LONG STRING BEING OUTPUT WHICH GOES OVE00000004
     1R MORE THAN ONE LINE, AND USES THE STATEMENT CONTINUATION MARKER IN00000005
     2COLUMN 6, AND ALSO USES HOLLERITH STRING FORMAT)                   00000006
      STOP                                                               00000007
      END                                                                00000008
```

El carácter de espacio también se ignoró en todas partes, excepto dentro de una constante de carácter *Hollerith* (como se muestra arriba). Esto significaba que los espacios podían aparecer dentro de palabras y constantes reservadas, o se podían perder por completo. Esto tuvo el efecto secundario de algunas afirmaciones bastante engañosas como:

```
      DO 1 I = 1.0
```

es una asignación a la variable `DO1I` mientras que:

```
      DO1I = 1,0
```

es en realidad un bucle de `DO` en la variable `I`

Fortran moderno ahora no requiere esta forma fija de entrada y permite la forma libre usando cualquier columna. Los comentarios ahora están indicados por un `!` que también se puede añadir a una línea de declaración. Los espacios ahora no están permitidos en ningún lugar y deben usarse como separadores, como en la mayoría de los otros idiomas. El programa anterior se podría escribir en Fortran moderno como:

```
! This is a comment statement to explain this example program
Print *,"THIS IS A RATHER LONG STRING BEING OUTPUT WHICH no longer GOES OVER MORE THAN ONE
LINE, AND does not need to USE THE STATEMENT CONTINUATION MARKER IN COLUMN 6, or the HOLLERITH
STRING FORMAT"
```

Aunque la continuación de estilo antiguo ya no se usa, el ejemplo anterior ilustra que todavía se producirán declaraciones muy largas. Modern Fortran usa un símbolo `&` al final y al comienzo de la continuación. Por ejemplo, podríamos escribir lo anterior en una forma más legible:

```
! This is a comment statement to explain this example program
Print *,"THIS IS A RATHER LONG STRING BEING OUTPUT WHICH still &
         &GOES OVER MORE THAN ONE LINE, AND does need to USE THE STATEMENT &
         &CONTINUATION notation"
```

Bloques comunes

En las primeras formas de Fortran, el único mecanismo para crear un almacén de variables global visible desde subrutinas y funciones es usar el mecanismo de bloque `COMMON` . Esto permitió que las secuencias de variables fueran nombres y se compartieran en común.

Además de los bloques comunes con nombre, también puede haber un bloque común en blanco (sin nombre).

Un bloque común en blanco podría ser declarado como

```
common i, j
```

mientras que las `variables` bloque nombradas podrían ser declaradas como

```
common /variables/ i, j
```

Como ejemplo completo, podríamos imaginar una tienda de almacenamiento dinámico utilizada por rutinas que pueden agregar y eliminar valores:

```
        PROGRAM STACKING
        COMMON /HEAP/ ICOUNT, ISTACK(1023)
        ICOUNT = 0
        READ *, IVAL
        CALL PUSH(IVAL)
        CALL POP(IVAL)
        END

        SUBROUTINE PUSH(IVAL)
        COMMON /HEAP/ ICOUNT, ISTACK(1023)
        ICOUNT = ICOUNT + 1
        ISTACK(ICOUNT) = IVAL
        RETURN
        END

        SUBROUTINE POP(IVAL)
        COMMON /HEAP/ ICOUNT, ISTACK(1023)
        IVAL = ISTACK(ICOUNT)
        ICOUNT = ICOUNT - 1
        RETURN
        END
```

Las declaraciones comunes se pueden usar para declarar implícitamente el tipo de una variable y para especificar el atributo de `dimension` . Este comportamiento solo es a menudo una fuente suficiente de confusión. Además, la asociación de almacenamiento implícita y los requisitos para definiciones repetidas en las unidades del programa hacen que el uso de bloques comunes sea propenso a errores.

Finalmente, los bloques comunes están muy restringidos en los objetos que contienen. Por ejemplo, una matriz en un bloque común debe ser de tamaño explícito; objetos asignables no pueden ocurrir; Los tipos derivados no deben tener inicialización por defecto.

En Fortran moderno, este uso compartido de variables se puede manejar mediante el uso de módulos . El ejemplo anterior se puede escribir como:

```
module heap
  implicit none
  ! In Fortran 2008 all module variables are implicitly saved
  integer, save :: count = 0
  integer, save :: stack(1023)
end module heap

program stacking
  implicit none
  integer val
  read *, val
  call push(val)
  call pop(val)

contains
  subroutine push(val)
    use heap, only : count, stack
    integer val
    count = count + 1
    stack(count) = val
  end subroutine push

  subroutine pop(val)
    use heap, only : count, stack
    integer val
    val = stack(count)
    count = count - 1
  end subroutine pop
end program stacking
```

Los bloques comunes nombrados y en blanco tienen comportamientos ligeramente diferentes. De nota:

- Los objetos en bloques comunes nombrados pueden definirse inicialmente; Los objetos en blanco no serán comunes.
- los objetos en bloques comunes en blanco se comportan como si el bloque común tuviera el atributo de `save` ; los objetos en bloques comunes nombrados sin el atributo de `save` pueden volverse indefinidos cuando el bloque no está en el alcance de una unidad de programa activa

Este último punto puede contrastarse con el comportamiento de las variables del módulo en el código moderno. Todas las variables de módulo en Fortran 2008 se guardan implícitamente y no se vuelven indefinidas cuando el módulo queda fuera del alcance. Antes de que las variables del módulo Fortran 2008, como las variables en bloques comunes con nombre, también se volvieran indefinidas cuando el módulo quedaba fuera del alcance.

Asignado GOTO

El GOTO asignado utiliza una variable entera a la que se asigna una etiqueta de declaración utilizando la instrucción ASSIGN.

```
100 CONTINUE

...

ASSIGN 100 TO ILABEL

...


GOTO ILABEL
```

El GOTO asignado es obsolescente en Fortran 90 y se elimina en Fortran 95 y posteriores. Se puede evitar en el código moderno utilizando procedimientos, procedimientos internos, indicadores de procedimientos y otras características.

GOTO computado

El GOTO computado permite la bifurcación del programa según el valor de una expresión entera.

```
GOTO (label_1, label_2,... label_n) scalar-integer-expression
```

Si `scalar-integer-expression` es igual a 1, el programa continúa en la etiqueta `label_1` , si es igual a 2, va a `label_2` y así sucesivamente. Si es menos de `1` o mayor que `n` programa continúa en la siguiente línea.

Ejemplo:

```
ivar = 2

...

GOTO (10, 20, 30, 40) ivar
```

saltará a la etiqueta de declaración 20.

Esta forma de `goto` es obsolescente en Fortran 95 y versiones posteriores, siendo reemplazada por el constructo de `select case` .

Especificadores de formato asignado

Antes de Fortran 95 era posible usar formatos asignados para entrada o salida. Considerar

```
integer i, fmt
read *, i

assign 100 to fmt
if (i<100000) assign 200 to fmt

print fmt, i

100 format ("This is a big number", I10)
200 format ("This is a small number", I6)
```

```
end
```

La declaración de `assign` asigna una etiqueta de declaración a una variable entera. Esta variable entera se usa más tarde como el especificador de formato en la declaración de `print` .

Dicha asignación de especificador de formato se eliminó en Fortran 95. En su lugar, un código más moderno puede usar alguna otra forma de control de flujo de ejecución

```
integer i
read *, i

if (i<100000) then
  print 100, i
else
  print 200, i
end if

100 format ("This is a big number", I10)
200 format ("This is a small number", I6)

end
```

o se puede usar una variable de carácter como el especificador de formato

```
character(29), target :: big_fmt='("This is a big number", I10)'
character(30), target :: small_fmt='("This is a small number", I6)'
character(:), pointer :: fmt

integer i
read *, i

fmt=>big_fmt
if (i<100000) fmt=>small_fmt

print fmt, i

end
```

Funciones de sentencias

Considerar el programa

```
implicit none
integer f, i
f(i)=i

print *, f(1)
end
```

Aquí `f` es una función de declaración. Tiene un tipo de resultado de entero, tomando un argumento ficticio de entero. [1]

Dicha función de declaración existe dentro del alcance en el que se define. En particular, tiene acceso a variables y constantes con nombre accesibles en ese ámbito.

Sin embargo, las funciones de declaración están sujetas a muchas restricciones y son potencialmente confusas (mirando a simple vista como una instrucción de asignación de elementos de matriz). Las restricciones importantes son:

- El resultado de la función y los argumentos ficticios deben ser escalares.
- Los argumentos ficticios están en el mismo ámbito que la función.
- funciones de declaración no tienen variables locales
- Las funciones de sentencias no se pueden pasar como argumentos reales.

Los beneficios principales de las funciones de sentencias son repetidos por las funciones internas.

```
implicit none

print *, f(1)

contains

  integer function f(i)
    integer i
    f = i
  end function

end
```

Las funciones internas no están sujetas a las restricciones mencionadas anteriormente, aunque quizás vale la pena señalar que un subprograma interno puede no contener un subprograma interno adicional (pero puede contener una función de declaración).

Las funciones internas tienen su propio alcance, pero también tienen una asociación de host disponible.

---

[1] En los ejemplos de código antiguo anterior, no sería extraño ver que los argumentos ficticios de una función de declaración se escriben de forma implícita, incluso si el resultado tiene un tipo explícito.

Lea Alternativas modernas a las características históricas. en línea: https://riptutorial.com/es/fortran/topic/2103/alternativas-modernas-a-las-caracteristicas-historicas-

Capítulo 3: Arrays

Examples

Notación básica

Cualquier tipo se puede declarar como una matriz utilizando el atributo de *dimensión* o simplemente indicando directamente la `dimension` o `dimension` de la matriz:

```
! One dimensional array with 4 elements
integer, dimension(4) :: foo

! Two dimensional array with 4 rows and 2 columns
real, dimension(4, 2) :: bar

! Three dimensional array
type(mytype), dimension(6, 7, 8) :: myarray

! Same as above without using the dimension keyword
integer :: foo2(4)
real :: bar2(4, 2)
type(mytype) :: myarray2(6, 7, 8)
```

La última forma de declarar una matriz multidimensional, permite la declaración de matrices de diferente rango / dimensiones del mismo tipo en una línea, como sigue

```
real :: pencil(5), plate(3,-2:4), cuboid(0:3,-10:5,6)
```

El rango máximo permitido (número de dimensiones) es 15 en el estándar Fortran 2008 y fue 7 antes.

Fortran almacena matrices en orden de *columnas mayores* . Es decir, los elementos de la `bar` se almacenan en la memoria de la siguiente manera:

```
bar(1, 1), bar(2, 1), bar(3, 1), bar(4, 1), bar(1, 2), bar(2, 2), ...
```

En Fortran, la numeración de matrices comienza en **1** por defecto, a diferencia de C que comienza en **0** . De hecho, en Fortran, puede especificar explícitamente los límites superior e inferior de cada dimensión:

```
integer, dimension(7:12, -3:-1) :: geese
```

Esto declara una matriz de forma `(6, 3)` , cuyo primer elemento son los `geese(7, -3)` .

Los límites inferior y superior a lo largo de las 2 (o más) dimensiones pueden accederse mediante las funciones intrínsecas `ubound` y `lbound` . De hecho, `lbound(geese,2)` devolvería `-3` , mientras que `ubound(geese,1)` devolvería `12` .

Se puede acceder al tamaño de una matriz mediante el `size` función intrínseca. Por ejemplo, `size(geese, dim = 1)` devuelve el tamaño de la primera dimensión que es 6.

Arreglos asignables

Las matrices pueden tener el atributo *asignable* :

```
! One dimensional allocatable array
integer, dimension(:), allocatable :: foo
! Two dimensional allocatable array
real, dimension(:,:), allocatable :: bar
```

Esto declara la variable pero no le asigna ningún espacio.

```
! We can specify the bounds as usual
allocate(foo(3:5))

! It is an error to allocate an array twice
! so check it has not been allocated first
if (.not. allocated(foo)) then
  allocate(bar(10, 2))
end if
```

Una vez que una variable ya no es necesaria, se puede *desasignar* :

```
deallocate(foo)
```

Si por alguna razón falla una instrucción de `allocate` , el programa se detendrá. Esto se puede evitar si el estado se comprueba mediante la palabra clave `stat` :

```
real, dimension(:), allocatable :: geese
integer :: status

allocate(geese(17), stat=status)
if (stat /= 0) then
  print*, "Something went wrong trying to allocate 'geese'"
  stop 1
end if
```

El `deallocate` declaración tiene `stat` de palabras clave también:

```
deallocate (geese, stat=status)
```

`status` es una variable entera cuyo valor es 0 si la asignación o desasignación fue exitosa.

Constructores de matrices

Se puede crear un valor de matriz de rango 1 utilizando un *constructor de matriz* , con la sintaxis

```
(/ ... /)
[ ... ]
```

La forma `[...]` se introdujo en Fortran 2003 y generalmente es considerado como más fáciles de interpretar, sobre todo en expresiones complejas. Este formulario se utiliza exclusivamente en este ejemplo.

Los valores que aparecen en un constructor de matriz pueden ser valores escalares, valores de matriz o bucles implícitos.

Los parámetros de tipo y tipo de la matriz construida coinciden con los de los valores en el constructor de matriz

```
[1, 2, 3]      ! A rank-1 length-3 array of default integer type
[1., 2., 3.]   ! A rank-1 length-3 array of default real type
["A", "B"]     ! A rank-1 length-2 array of default character type

integer, parameter :: A = [2, 4]
[1, A, 3]      ! A rank-1 length-4 array of default integer type, with A's elements

integer i
[1, (i, i=2, 5), 6]  ! A rank-1 length-6 array of default integer type with an implied-do
```

En los formularios anteriores, todos los valores dados deben ser del mismo tipo y tipo de parámetro. Los tipos de mezcla, o parámetros de tipo, no están permitidos. Los siguientes ejemplos **no son válidos.**

```
[1, 2.]      ! INVALID: Mixing integer and default real
[1e0, 2d0]   ! INVALID: Mixing default real and double precision
[1., 2._dp]  ! INVALID: Allowed only if kind `dp` corresponds to default real
["Hello", "Frederick"]  ! INVALID: Different length parameters
```

Para construir una matriz utilizando diferentes tipos, se dará una especificación de tipo para la matriz

```
[integer :: 1, 2., 3d0]    ! A default integer array
[real(dp) :: 1, 2, 3._sp]  ! A real(dp) array
[character(len=9) :: "Hello", "Frederick"]  ! A length-2 array of length-9 characters
```

Esta última forma para matrices de caracteres es especialmente conveniente para evitar el espacio de relleno, como la alternativa

```
["Hello    ", "Frederick"]  ! A length-2 array of length-9 characters
```

El tamaño de una matriz denominada constante puede estar implícito en el constructor de la matriz que se utiliza para establecer su valor

```
integer, parameter :: ids(*) = [1, 2, 3, 4]
```

y para tipos parametrizados por longitud, el parámetro de longitud puede ser asumido

```
character(len=*), parameter :: names(*) = [character(3) :: "Me", "You", "Her"]
```

La especificación de tipo también se requiere en la construcción de matrices de longitud cero. Desde

```
[ ] ! Not a valid array constructor
```

los parámetros de tipo y tipo no se pueden determinar a partir del conjunto de valores no existentes. Para crear una matriz de enteros por defecto de longitud cero:

```
[integer :: ]
```

Los constructores de matrices construyen solo matrices de rango 1. A veces, como al establecer el valor de una constante nombrada, también se requieren arreglos de rango más alto en una expresión. Las matrices de rango más alto se pueden tomar del resultado de la `reshape` con una matriz de rango 1 construida

```
integer, parameter :: multi_rank_ids(2,2) = RESHAPE([1,2,3,4], shape=[2,2])
```

En un constructor de matriz, los valores de la matriz en orden de elementos con cualquier matriz en la lista de valores es como si los elementos individuales se dieran en orden de elementos de matriz. Así, el ejemplo anterior

```
integer, parameter :: A = [2, 4]
[1, A, 3]      ! A rank-1 length-4 array of default integer type, with A's elements
```

es equivalente a

```
[1, 2, 4, 3]   ! With the array written out in array element order
```

En general, los valores en el constructor pueden ser expresiones arbitrarias, incluidos los constructores de matrices anidadas. Para que un constructor de matriz de este tipo cumpla ciertas condiciones, como ser una expresión constante o de especificación, las restricciones se aplican a los valores constituyentes.

Aunque no es un constructor de matriz, ciertos valores de matriz también pueden crearse convenientemente usando la función intrínseca de `spread` . Por ejemplo

```
[(0, i=1,10)]  ! An array with 10 default integers each of value 0
```

También es el resultado de la referencia de la función.

```
SPREAD(0, 1, 10)
```

Especificación de la naturaleza de la matriz: rango y forma

El atributo de `dimension` en un objeto especifica que ese objeto es una matriz. Hay, en Fortran 2008, cinco naturalezas de matrices: [1]

- forma explícita
- forma asumida
- tamaño asumido
- forma diferida
- forma implícita

Toma las tres matrices de rango 1 [2]

```
integer a, b, c
dimension(5) a    ! Explicit shape (default lower bound 1), extent 5
dimension(:) b    ! Assumed or deferred shape
dimension(*) c    ! Assumed size or implied shape array
```

Con estos se puede ver que se requiere un contexto adicional para determinar completamente la naturaleza de una matriz.

Forma explícita

Una matriz de forma explícita es siempre la forma de su declaración. A menos que la matriz se declare como local a un subprograma o construcción de `block` , los límites que definen la forma deben ser expresiones constantes. En otros casos, una matriz de forma explícita puede ser un objeto automático, utilizando extensiones que pueden variar en cada invocación de un subprograma o `block` .

```
subroutine sub(n)
  integer, intent(in) :: n
  integer a(5)   ! A local explicit shape array with constant bound
  integer b(n)   ! A local explicit shape array, automatic object
end subroutine
```

Forma asumida

Una matriz de forma asumida es un argumento ficticio sin el atributo `allocatable` o `pointer` . Dicha matriz toma su forma del argumento real con el que está asociada.

```
integer a(5), b(10)
call sub(a)   ! In this call the dummy argument is like x(5)
call sub(b)   ! In this call the dummy argument is like x(10)

contains

  subroutine sub(x)
    integer x(:)    ! Assumed shape dummy argument
  end subroutine sub

end
```

Cuando un argumento ficticio ha tomado forma, el alcance que hace referencia al procedimiento debe tener una interfaz explícita disponible para ese procedimiento.

Tamaño asumido

Una matriz de tamaño asumido es un argumento ficticio que tiene su tamaño asumido a partir de su argumento real.

```
subroutine sub(x)
  integer x(*)    ! Assumed size array
end subroutine
```

Las matrices de tamaños asumidos se comportan de manera muy diferente a las matrices de formas asumidas y estas diferencias se documentan en otra parte.

Forma diferida

Una matriz de forma diferida es una matriz que tiene el atributo `allocatable` o `pointer` . La forma de dicha matriz está determinada por su asignación o asignación de puntero.

```
integer, allocatable :: a(:)
integer, pointer :: b(:)
```

Forma implícita

Una matriz de formas implícita es una constante con nombre que toma su forma de la expresión utilizada para establecer su valor

```
integer, parameter :: a(*) = [1,2,3,4]
```

Las implicaciones de estas declaraciones matriciales en argumentos ficticios deben documentarse en otra parte.

---

[1] Una Especificación Técnica que extiende Fortran 2008 agrega una sexta naturaleza de matriz: rango asumido. Esto no está cubierto aquí.

[2] Se pueden escribir de manera equivalente como

```
integer, dimension(5) :: a
integer, dimension(:) :: b
integer, dimension(*) :: c
```

o

```
integer a(5)
```

```
integer b(:)
integer c(*)
```

Arreglos enteros, elementos de arreglo y secciones de arreglo.

Considere la matriz declarada como

```
real x(10)
```

Entonces tenemos tres aspectos de interés:

1. Toda la matriz `x` ;
2. Elementos de matriz, como `x(1)` ;
3. Arreglo de secciones, como `x(2:6)` .

Arreglos enteros

En la mayoría de los casos, toda la matriz `x` refiere a todos los elementos de la matriz como una sola entidad. Puede aparecer en sentencias ejecutables, como `print *, SUM(x)` , `print *, SIZE(x)` o `x=1` .

Una matriz completa puede hacer referencia a matrices que no tienen forma explícita (como `x` arriba):

```
function f(y)
  real, intent(out) :: y(:)
  real, allocatable :: z(:)

  y = 1.         ! Intrinsic assignment for the whole array
  z = [1., 2.,]  ! Intrinsic assignment for the whole array, invoking allocation
end function
```

Una matriz de tamaño supuesto también puede aparecer como una matriz completa, pero solo en circunstancias limitadas (para ser documentada en otra parte).

Elementos de matriz

Se hace referencia a un elemento de la matriz para dar índices enteros, uno para cada rango de la matriz, que denota la ubicación en la matriz completa:

```
real x(5,2)
x(1,1) = 0.2
x(2,4) = 0.3
```

Un elemento de matriz es un escalar.

Secciones de matriz

Una sección de matriz es una referencia a una serie de elementos (quizás solo uno) de una matriz completa, utilizando una sintaxis que involucra dos puntos:

```
real x(5,2)
x(:,1) = 0.          ! Referring to x(1,1), x(2,1), x(3,1), x(4,1) and x(5,1)
x(2,:) = 0.          ! Referring to x(2,1), x(2,2)
x(2:4,1) = 0.        ! Referring to x(2,1), x(3,1) and x(4,1)
x(2:3,1:2) = 0.      ! Referring to x(2,1), x(3,1), x(2,2) and x(3,2)
x(1:1,1) = 0.        ! Referring to x(1,1)
x([1,3,5],2) = 0.    ! Referring to x(1,2), x(3,2) and x(5,2)
```

La forma final anterior utiliza un *subíndice vectorial* . Esto está sujeto a una serie de restricciones más allá de otras secciones de la matriz.

Cada sección de matriz es en sí misma una matriz, incluso cuando solo se hace referencia a un elemento. Eso es `x(1:1,1)` es una matriz de rango 1 y `x(1:1,1:1)` es una matriz de rango 2.

Las secciones de matriz en general no tienen un atributo de toda la matriz. En particular, donde

```
real, allocatable :: x(:)
x = [1,2,3]     ! x is allocated as part of the assignment
x = [1,2,3,4]   ! x is dealloacted then allocated to a new shape in the assignment
```

la asignación

```
x(:) = [1,2,3,4,5]   ! This is bad when x isn't the same shape as the right-hand side
```

no está permitido: `x(:)` , aunque una sección de matriz con todos los elementos de `x` , no es una matriz asignable.

```
x(:) = [5,6,7,8]
```

está bien cuando `x` es de la forma del lado derecho.

Componentes de matrices de matrices

```
type t
   real y(5)
end type t

type(t) x(2)
```

También podemos referirnos a matrices completas, elementos de matriz y secciones de matriz en configuraciones más complicadas.

De lo anterior, `x` es una matriz completa. También tenemos

```
x(1)%y        ! A whole array
x(1)%y(1)     ! An array element
```

```
x%y(1)         ! An array section
x(1)%y(:)      ! An array section
x([1,2]%y(1)   ! An array section
x(1)%y(1:1)    ! An array section
```

En tales casos, no se nos permite tener más de una parte de la referencia que consiste en una matriz de rango 1. Lo siguiente, por ejemplo, no está permitido

```
x%y              ! Both the x and y parts are arrays
x(1:1)%y(1:1)    ! Recall that each part is still an array section
```

Operaciones de matriz

Debido a sus objetivos computacionales, las operaciones matemáticas en matrices son sencillas en Fortran.

Adición y sustracción

Las operaciones en matrices de la misma forma y tamaño son muy similares al álgebra matricial. En lugar de recorrer todos los índices con bucles, se puede escribir suma (y resta):

```
real, dimension(2,3) :: A, B, C
real, dimension(5,6,3) :: D
A    = 3.    ! Assigning single value to the whole array
B    = 5.    ! Equivalent writing for assignment
C    = A + B ! All elements of C now have value 8.
D    = A + B ! Compiler will raise an error. The shapes and dimensions are not the same
```

Las matrices de corte son también válidas:

```
integer :: i, j
real, dimension(3,2) :: Mat = 0.
real, dimension(3)   :: Vec1 = 0., Vec2 = 0., Vec3 = 0.
i = 0
j = 0
do i = 1,3
  do j = 1,2
    Mat(i,j) = i+j
  enddo
enddo
Vec1 = Mat(:,1)
Vec2 = Mat(:,2)
Vec3 = Mat(1:2,1) + Mat(2:3,2)
```

Función

De la misma manera, la mayoría de las funciones intrínsecas se pueden usar de forma implícita asumiendo una operación de componentes (aunque esto no es sistemático):

```
real, dimension(2) :: A, B
```

```
A(1) = 6
A(2) = 44 ! Random values
B    = sin(A) ! Identical to B(1) = sin(6), B(2) = sin(44).
```

Multiplicación y división

Se debe tener cuidado con el producto y la división: las operaciones intrínsecas que usan `*` y `/` son símbolos en forma elemental:

```
real, dimension(2) :: A, B, C
A(1) = 2
A(2) = 4
B(1) = 1
B(2) = 3
C = A*B ! Returns C(1) = 2*1 and C(2) = 4*3
```

Esto no debe confundirse con operaciones matriciales (ver más abajo).

Operaciones matriciales

Las operaciones matriciales son procedimientos intrínsecos. Por ejemplo, el producto matricial de las matrices de la sección anterior se escribe de la siguiente manera:

```
real, dimension(2,1) :: A, B
real, dimension(1,1) :: C
A(1) = 2
A(2) = 4
B(1) = 1
B(2) = 3
C = matmul(transpose(A),B) ! Returns the scalar product of vectors A and B
```

Las operaciones complejas permiten encapsular funciones mediante la creación de arreglos temporales. Mientras que algunos compiladores y opciones de compilación lo permiten, esto no se recomienda. Por ejemplo, un producto que incluye una transposición matricial puede escribirse:

```
real, dimension(3,3) :: A, B, C
A(:) = 4
B(:) = 5
C = matmul(transpose(A),matmul(B,matmul(A,transpose(B)))) ! Equivalent to A^t.B.A.B^T
```

Secciones de matrices avanzadas: tripletes de subíndices y subíndices vectoriales

Como se mencionó en otro ejemplo , se puede hacer referencia a un subconjunto de los elementos de una matriz, llamada sección de matriz. De ese ejemplo podemos tener

```
real x(10)
x(:)   = 0.
```

```
x(2:6) = 1.
x(3:4) = [3., 5.]
```

Sin embargo, las secciones de matriz pueden ser más generales que esto. Pueden tomar la forma de tripletes de subíndice o subíndices vectoriales.

Tripletes de subíndices

Un subíndice triple toma la forma `[bound1]:[bound2][:stride]` . Por ejemplo

```
real x(10)
x(1:10) = ...   ! Elements x(1), x(2), ..., x(10)
x(1:) = ...     ! The omitted second bound is equivalent to the upper, same as above
x(:10) = ...    ! The omitted first bound is equivalent to the lower, same as above
x(1:6:2) = ...  ! Elements x(1), x(3), x(5)
x(5:1) = ...    ! No elements: the lower bound is greater than the upper
x(5:1:-1) = ... ! Elements x(5), x(4), x(3), x(2), x(1)
x(::3) = ...    ! Elements x(1), x(4), x(7), x(10), assuming omitted bounds
x(::-3) = ...   ! No elements: the bounds are assumed with the first the lower, negative
stride
```

Cuando se especifica un paso (que no debe ser cero), la secuencia de elementos comienza con el primer límite especificado. Si la zancada es positiva (resp. Negativa), los elementos seleccionados siguiendo una secuencia incrementada (resp. Decrementada) por la zancada hasta que el último elemento no sea mayor (resp. Más pequeño) que el segundo límite se toma. Si se omite el paso, se trata como si fuera uno.

Si el primer límite es mayor que el segundo límite, y la zancada es positiva, no se especifican elementos. Si el primer límite es más pequeño que el segundo límite y la zancada es negativa, no se especifican elementos.

Cabe señalar que `x(10:1:-1)` no es lo mismo que `x(1:10:1)` aunque cada elemento de `x` aparezca en ambos casos.

Subíndices vectoriales

Un subíndice vectorial es una matriz de enteros de rango 1. Esto designa una secuencia de elementos correspondientes a los valores de la matriz.

```
real x(10)
integer i
x([1,6,4]) = ...     ! Elements x(1), x(6), x(4)
x([(i,i=2,4)]) = ... ! Elements x(2), x(3) and x(4)
print*, x([2,5,2])   ! Elements x(2), x(5) and x(2)
```

Una sección de matriz con un subíndice vectorial está restringida en la forma en que se puede usar:

- puede que no sea un argumento asociado con un argumento ficticio que se define en el procedimiento;

- puede que no sea el objetivo en una instrucción de asignación de puntero;
- puede que no sea un archivo interno en una declaración de transferencia de datos.

Además, tal sección de matriz puede no aparecer en una declaración que implique su definición cuando el mismo elemento se selecciona dos veces. Desde arriba:

```
print*, x([2,5,2])   ! Elements x(2), x(5) and x(2) are printed
x([2,5,2]) = 1.      ! Not permitted: x(2) appears twice in this definition
```

Secciones de matriz de rango más alto

```
real x(5,2)
print*, x(::2,2:1:-1)  ! Elements x(1,2), x(3,2), x(5,2), x(1,1), x(3,1), x(5,1)
```

Lea Arrays en línea: https://riptutorial.com/es/fortran/topic/996/arrays

Capítulo 4: Control de ejecución

Examples

Si construir

El constructo `if` (llamado una instrucción IF de bloque en FORTRAN 77) es común en muchos lenguajes de programación. Se ejecuta condicionalmente un bloque de código cuando una expresión lógica se evalúa como verdadera.

```
[name:] IF (expr) THEN
    block
[ELSE IF (expr) THEN [name]
    block]
[ELSE [name]
   block]
END IF [name]
```

dónde,

- **nombre** - el nombre de la construcción if (opcional)
- **expr** - una expresión lógica escalar entre paréntesis
- **bloque** - una secuencia de cero o más declaraciones o construcciones

Un nombre de construcción al comienzo de una instrucción `if then` debe tener el mismo valor que el nombre de construcción al `end if` instrucción `end if` , y debe ser único para la unidad de alcance actual.

En las sentencias `if` , las ecuaciones (in) y las expresiones lógicas que evalúan una sentencia se pueden utilizar con los siguientes operadores:

```
.LT.  which is <   ! less than
.LE.           <=  ! less than or equal
.GT.           >   ! greater than
.GE.           >=  ! greater than or equal
.EQ.           =   ! equal
.NE.           /=  ! not equal
.AND.              ! logical and
.OR.               ! logical or
.NOT.              ! negation
```

Ejemplos:

```
! simplest form of if construct
if (a > b) then
    c =  b / 2
end if
!equivalent example with alternate syntax
if(a.gt.b)then
   c=b/2
```

```
endif

! named if construct
circle: if (r >= 0) then
    l = 2 * pi * r
end if circle

! complex example with nested if construct
block: if (a < e) then
    if (abs(c - e) <= d) then
        a = a * c
    else
        a = a * d
    end if
else
    a = a * e
end if block
```

Un uso histórico de la construcción `if` encuentra en lo que se denomina una declaración "aritmética if". Sin embargo, dado que esto puede ser reemplazado por construcciones más modernas, no se trata aquí. Más detalles se pueden encontrar aquí .

SELECCIONAR CASO de construcción

Una construcción de `select case` condicionalmente ejecuta un bloque de construcciones o declaraciones según el valor de una expresión escalar en una declaración de `select case` . Este constructo de control puede considerarse como un reemplazo para el `goto` computado.

```
[name:] SELECT CASE (expr)
[CASE (case-value [, case-value] ...) [name]
   block]...
[CASE DEFAULT [name]
   block]
END SELECT [name]
```

dónde,

- **nombre** : el nombre de la construcción de `select case` (opcional)
- **expr** - una expresión escalar de tipo entero, lógico o carácter (entre paréntesis)
- **valor-caso** : una o más expresiones de inicialización escalar de enteros, lógicas o caracteres encerradas entre paréntesis
- **bloque** - una secuencia de cero o más declaraciones o construcciones

Ejemplos:

```
! simplest form of select case construct
select case(i)
case(:-1)
    s = -1
case(0)
    s = 0
case(1:)
    s = 1
```

```
case default
    print "Something strange is happened"
end select
```

En este ejemplo, `(:-1)` valor de un caso es un rango de valores que coincide con todos los valores menores que cero, `(0)` coincide con los ceros, y `(1:)` coincide con todos los valores por encima de cero, la sección `default` incluye si otras secciones lo hicieron sin ejecutar.

Construcción de bloques DO

Una construcción `do` es una construcción de bucle que tiene una serie de iteraciones gobernadas por un control de bucle

```
integer i
do i=1, 5
  print *, i
end do
print *, i
```

En la forma anterior, la variable de bucle `i` pasa a través del bucle 5 veces, tomando los valores de 1 a 5 por turno. Una vez que la construcción ha completado, la variable de bucle tiene el valor 6, es decir, **la variable de bucle se incrementa una vez más después de completar el bucle** .

Más generalmente, la construcción de bucle `do` se puede entender de la siguiente manera

```
integer i, first, last, step
do i=first, last, step
end do
```

El bucle comienza con `i` con el valor `first` , incrementando cada iteración por `step` hasta que `i` sea mayor que la `last` (o menor que la `last` si el tamaño del paso es negativo).

Es importante tener en cuenta que desde Fortran 95, la variable de bucle y las expresiones de control de bucle deben ser enteras.

Una iteración se puede finalizar de manera prematura con la instrucción de `cycle`

```
do i=1, 5
  if (i==4) cycle
end do
```

y toda la construcción puede cesar la ejecución con la declaración de `exit`

```
do i=1, 5
  if (i==4) exit
end do
print *, i
```

`do` construcciones pueden ser nombradas:

```
do_name: do i=1, 5
end do do_name
```

Lo cual es particularmente útil cuando hay construcciones anidadas `do`

```
do1: do i=1, 5
  do j=1,6
    if (j==3) cycle        ! This cycles the j construct
    if (j==4) cycle        ! This cycles the j construct
    if (i+j==7) cycle do1  ! This cycles the i construct
    if (i*j==15) exit do1  ! This exits the i construct
  end do
end do1
```

`do` construcciones `do` también pueden tener un control de bucle indeterminado, ya sea "para siempre" o hasta que se cumpla una condición determinada

```
integer :: i=0
do
  i=i+1
  if (i==5) exit
end do
```

o

```
integer :: i=0
do while (i<6)
  i=i+1
end do
```

Esto también permite un infinito `do` bucle a través de un `.true.` declaración

```
print *,'forever'
do while(.true.)
  print *,'and ever'
end do
```

Un constructo `do` también puede dejar el orden de iteraciones indeterminado.

```
do concurrent (i=1:5)
end do
```

teniendo en cuenta que la forma de control de bucle es la misma que en un control `forall` .

Hay varias restricciones en las declaraciones que pueden ejecutarse dentro del rango de una construcción `do concurrent` que están diseñadas para garantizar que no haya dependencias de datos entre las iteraciones de la construcción. Esta indicación explícita por parte del programador puede permitir una mayor optimización (incluida la paralelización) por parte del compilador, lo que puede ser difícil de determinar de otro modo.

Las variables "privadas" dentro de una interacción se pueden realizar mediante el uso de una construcción de `block` dentro del `do concurrent` :

```
do concurrent (i=1:5, j=2:7)
  block
    real tempval  ! This is independent across iterations
  end block
end do
```

Otra forma del bloque `do` constructivo utiliza una instrucción `continue` etiquetada en lugar de un `end do` :

```
    do 100, i=1, 5
100 continue
```

Incluso es posible anidar tales construcciones con una declaración de terminación compartida

```
    do 100, i=1,5
    do 100, j=1,5
100 continue
```

Ambas formas, y especialmente la segunda (que es obsolescente), generalmente deben evitarse en aras de la claridad.

Finalmente, también es un no-bloque `do` construir. Esto también se considera que es obsoleto y se describe en otra parte , junto con los métodos para reestructurar como un bloque `do` construir.

Donde construir

El `where` constructo, disponible en Fortran90 representa en adelante un enmascarado `do` construir. La declaración de enmascaramiento sigue las mismas reglas de la instrucción `if` , pero se aplica a todos los elementos de la matriz dada. Usando `where` permite que las operaciones se lleven a cabo en una matriz (o múltiples matrices del mismo tamaño), cuyos elementos satisfacen una regla determinada. Esto se puede usar para simplificar operaciones simultáneas en varias variables.

Sintaxis:

```
[name]: where (mask)
    block
[elsewhere (mask)
    block]
[elsewhere
    block]
end where [name]
```

Aquí,

- **nombre** - es el nombre dado al bloque (si es nombrado)

- **máscara** - es una expresión lógica aplicada a todos los elementos
- **bloque** - serie de comandos a ejecutar

Ejemplos:

```
! Example variables
real:: A(5),B(5),C(5)
A = 0.0
B = 1.0
C = [0.0, 4.0, 5.0, 10.0, 0.0]

! Simple where construct use
where (C/=0)
    A=B/C
elsewhere
    A=0.0
end

! Named where construct
Block: where (C/=0)
    A=B/C
elsewhere
    A=0.0
end where Block
```

Lea Control de ejecución en línea: https://riptutorial.com/es/fortran/topic/1657/control-de-ejecucion

Capítulo 5: Extensiones de archivo fuente (.f, .f90, .f95, ...) y cómo se relacionan con el compilador.

Introducción

Los archivos de Fortran están bajo una variedad de extensiones y cada uno de ellos tiene un significado diferente. Especifican la versión de lanzamiento de Fortran, el estilo de formato de código y el uso de directivas de preprocesador similares al lenguaje de programación C.

Examples

Extensiones y significados

Las siguientes son algunas de las extensiones comunes que se utilizan en los archivos fuente de Fortran y las funcionalidades en las que pueden trabajar.

F minúscula en la extensión

Estos archivos no tienen las características de las directivas de preprocesador similares al lenguaje de programación C. Se pueden compilar directamente para crear archivos de objetos. Por ejemplo: .f, .for, .f95

F mayúscula en la extensión

Estos archivos tienen las características de las directivas de preprocesador similares al lenguaje de programación C. Los preprocesadores se definen dentro de los archivos o utilizan archivos de encabezado como C / C ++ o ambos. Estos archivos deben preprocesarse para obtener los archivos de extensión en minúsculas que se pueden usar para compilar. Por ejemplo: .F, .FOR, .F95

.f, .for, .f77, .ftn

Se usan para archivos Fortran que usan el **formato de estilo Fijo** y, por lo tanto, usan la versión de lanzamiento de **Fortran 77** . Como son extensiones en minúsculas, no pueden tener directivas de preprocesador.

.F, .FOR, .F77, .FTN

Se usan para archivos Fortran que usan el **formato de estilo Fijo** y, por lo tanto, usan la versión de lanzamiento de **Fortran 77** . Como son extensiones en mayúsculas, pueden tener directivas de preprocesador y, por lo tanto, deben procesarse previamente para obtener los archivos de extensión en minúsculas.

.f90, .f95, .f03, .f08 Se usan para archivos Fortran que usan el **formato de estilo libre** y, por lo tanto, usan versiones de Fortran más recientes. Las versiones de lanzamiento están en el nombre.

- f90 - Fortran 90
- f95 - Fortran 95
- f03 - Fortran 2003
- f08 - Fortran 2008

Como son extensiones en minúsculas, no pueden tener directivas de preprocesador.

.F90, .F95, .F03, .F08 Se usan para archivos Fortran que usan el **formato de estilo libre** y, por lo tanto, usan versiones de Fortran más recientes. Las versiones de lanzamiento están en el nombre.

- F90 - Fortran 90
- F95 - Fortran 95
- F03 - Fortran 2003
- F08 - Fortran 2008

Como son extensiones en mayúsculas, tienen directivas de preprocesador y, por lo tanto, deben procesarse previamente para obtener los archivos de extensión en minúsculas.

Lea Extensiones de archivo fuente (.f, .f90, .f95, ...) y cómo se relacionan con el compilador. en línea: https://riptutorial.com/es/fortran/topic/10265/extensiones-de-archivo-fuente---f---f90---f95-------y-como-se-relacionan-con-el-compilador-

Capítulo 6: I / O

Sintaxis

- `WRITE(unit num, format num)` genera los datos después de los corchetes en una nueva línea.
- `READ(unit num, format num)` entradas a la variable después de los corchetes.
- `OPEN(unit num, FILE=file)` abre un archivo. (Hay más opciones para abrir archivos, pero no son importantes para E / S.
- `CLOSE(unit num)` cierra un archivo.

Examples

E / S simple

Como ejemplo de escritura de entrada y salida, tomaremos un valor real y devolveremos el valor y su cuadrado hasta que el usuario ingrese un número negativo.

Como se especifica a continuación, el comando de `read` toma dos argumentos: el número de unidad y el especificador de formato. En el siguiente ejemplo, usamos `*` para el número de unidad (que indica stdin) y `*` para el formato (que indica el valor predeterminado para reales, en este caso). También especificamos el formato para la declaración de `print` . Alternativamente, se puede usar `write(*,"The value....")` o simplemente ignorar el formato y tenerlo como

```
print *,"The entered value was ", x," and its square is ",x*x
```

lo que probablemente resultará en algunas cadenas y valores espaciados de manera extraña.

```
program SimpleIO
   implicit none
   integer, parameter :: wp = selected_real_kind(15,307)
   real(kind=wp) :: x

   ! we'll loop over until user enters a negative number
   print '("Enter a number >= 0 to see its square. Enter a number < 0 to exit.")'
   do
      ! this reads the input as a double-pricision value
      read(*,*) x
      if (x < 0d0) exit
      ! print the entered value and it's square
      print '("The entered value was ",f12.6,", its square is ",f12.6,".")',x,x*x
   end do
   print '("Thank you!")'

end program SimpleIO
```

Leer con un poco de comprobación de errores

Un ejemplo moderno de Fortran que incluye la verificación de errores y una función para obtener

un nuevo número de unidad para el archivo.

```
module functions

contains

    function get_new_fileunit() result (f)
        implicit none

        logical    :: op
        integer    :: f

        f = 1
        do
            inquire(f,opened=op)
            if (op .eqv. .false.) exit
            f = f + 1
        enddo

    end function

end module

program file_read
    use functions, only : get_new_fileunit
    implicit none

    integer            :: unitno, ierr, readerr
    logical            :: exists
    real(kind(0.d0))   :: somevalue
    character(len=128) :: filename

    filename = "somefile.txt"

    inquire(file=trim(filename), exist=exists)
    if (exists) then
        unitno = get_new_fileunit()
        open(unitno, file=trim(filename), action="read", iostat=ierr)
        if (ierr .eq. 0) then
            read(unitno, *, iostat=readerr) somevalue
            if (readerr .eq. 0) then
                print*, "Value in file ", trim(filename), " is ", somevalue
            else
                print*, "Error ", readerr, &
                        " attempting to read file ", &
                        trim(filename)
            endif
        else
            print*, "Error ", ierr ," attempting to open file ", trim(filename)
            stop
        endif
    else
        print*, "Error -- cannot find file: ", trim(filename)
        stop
    endif

end program file_read
```

Pasando argumentos de línea de comando

Donde se admiten los argumentos de la línea de comando, se pueden leer a través del intrínseco `get_command_argument` (introducido en el estándar Fortran 2003). El intrínseco `command_argument_count` proporciona una manera de conocer el número de argumentos proporcionados en la línea de comandos.

Todos los argumentos de la línea de comando se leen como cadenas, por lo que se debe realizar una conversión de tipo interna para los datos numéricos. Como ejemplo, este código simple suma los dos números proporcionados en la línea de comando:

```
PROGRAM cmdlnsum
IMPLICIT NONE
CHARACTER(100) :: num1char
CHARACTER(100) :: num2char
REAL :: num1
REAL :: num2
REAL :: numsum

!First, make sure the right number of inputs have been provided
IF(COMMAND_ARGUMENT_COUNT().NE.2)THEN
  WRITE(*,*)'ERROR, TWO COMMAND-LINE ARGUMENTS REQUIRED, STOPPING'
  STOP
ENDIF

CALL GET_COMMAND_ARGUMENT(1,num1char)   !first, read in the two values
CALL GET_COMMAND_ARGUMENT(2,num2char)

READ(num1char,*)num1                    !then, convert them to REALs
READ(num2char,*)num2

numsum=num1+num2                        !sum numbers
WRITE(*,*)numsum                        !write out value

END PROGRAM
```

El argumento del número en `get_command_argument` útilmente entre `0` y el resultado de `command_argument_count` . Si el valor es `0` se proporciona el nombre del comando (si se admite).

Muchos compiladores también ofrecen intrínsecos no estándar (como `getarg`) para acceder a los argumentos de la línea de comandos. Como estos no son estándar, se debe consultar la documentación del compilador correspondiente.

El uso de `get_command_argument` puede extenderse más allá del ejemplo anterior con los argumentos de `length` y `status` . Por ejemplo, con

```
character(5) arg
integer stat
call get_command_argument(number=1, value=arg, status=stat)
```

el valor de `stat` será `-1` si el primer argumento existe y tiene una longitud mayor que 5. Si hay alguna otra dificultad para recuperar el argumento, el valor de `stat` será un número positivo (y `arg` constará completamente de espacios en blanco). De lo contrario su valor será `0` .

El argumento de `length` se puede combinar con una variable de carácter de longitud diferida,

como en el siguiente ejemplo.

```
character(:), allocatable :: arg
integer arglen, stat
call get_command_argument(number=1, length=arglen)  ! Assume for simplicity success
allocate (character(arglen) :: arg)
call get_command_argument(number=1, value=arg, status=stat)
```

Lea I / O en línea: https://riptutorial.com/es/fortran/topic/6778/i---o

Capítulo 7: Interfaces explícitas e implícitas

Examples

Subprogramas internos / módulos e interfaces explícitas

Un *subprograma* (que define un *procedimiento*), puede ser una `subroutine` o una `function` ; se dice que es un *subprograma interno* si se llama o se invoca desde el mismo `program` o *subprograma* que lo `contains` , como sigue

```
program my_program

  ! declarations
  ! executable statements,
  ! among which an invocation to
  ! internal procedure(s),
  call my_sub(arg1,arg2,...)
  fx = my_fun(xx1,xx2,...)

contains

  subroutine my_sub(a1,a2,...)
    ! declarations
    ! executable statements
  end subroutine my_sub

  function my_fun(x1,x2,...) result(f)
    ! declarations
    ! executable statements
  end function my_fun

end program my_program
```

En este caso, el compilador sabrá todo sobre cualquier procedimiento interno, ya que trata la unidad del programa como un todo. En particular, "verá" la `interface` del procedimiento, es decir,

- si es una `function` o `subroutine` ,
- cuáles son los nombres y propiedades de los argumentos `a1` , `a2` , `x1` , `x2` , ...,
- cuáles son las propiedades del *resultado* `f` (en el caso de una `function`).

Al conocerse la interfaz, el compilador puede verificar si los argumentos reales (`arg1` , `arg2` , `xx1` , `xx2` , `fx` , ...) pasaron al procedimiento con los argumentos ficticios (`a1` , `a2` , `x1` , `x2` , `f` , .. .).

En este caso decimos que la interfaz es *explícita* .

Se dice que un subprograma es un subprograma de *módulo* cuando es invocado por una declaración en el módulo que lo contiene,

```
module my_mod

  ! declarations
```

```
contains

  subroutine my_mod_sub(b1,b2,...)
    ! declarations
    ! executable statements
    r = my_mod_fun(b1,b2,...)
  end subroutine my_sub

  function my_mod_fun(y1,y2,...) result(g)
    ! declarations
    ! executable statements
  end function my_fun

end module my_mod
```

o por una declaración en otra unidad de programa que `use` s ese módulo,

```
program my_prog

  use my_mod

  call my_mod_sub(...)

end program my_prog
```

Como en la situación anterior, el compilador sabrá todo sobre el subprograma y, por lo tanto, decimos que la interfaz es *explícita* .

Subprogramas externos e interfaces implícitas

Se dice que un subprograma es *externo* cuando no está contenido en el programa principal, ni en un módulo o subprograma adicional. En particular, se puede definir por medio de un lenguaje de programación que no sea Fortran.

Cuando se invoca un subprograma externo, el compilador no puede acceder a su código, por lo que toda la información permitida al compilador está implícitamente contenida en la declaración del programa que llama y en el tipo una propiedad de los argumentos reales, no los argumentos ficticios (cuya declaración es desconocida para el compilador). En este caso decimos que la interfaz es *implícita* .

Se puede usar una declaración `external` para especificar que el nombre de un procedimiento es relativo a un procedimiento externo,

```
external external_name_list
```

Pero aun así, la interfaz permanece implícita.

Se puede utilizar un bloque de `interface` para especificar la interfaz de un procedimiento externo,

```
interface
  interface_body
```

```
end interface
```

donde `interface_body` es normalmente una copia exacta del encabezado del procedimiento seguido de la declaración de todos sus argumentos y, si es una función, del resultado.

Por ejemplo, para la función `WindSpeed`

```
real function WindSpeed(u, v)
  real, intent(in) :: u, v
  WindSpeed = sqrt(u*u + v*v)
end function WindSpeed
```

Puedes escribir la siguiente interfaz

```
interface
  real function WindSpeed(u, v)
    real, intent(in) :: u, v
  end function WindSpeed
end interface
```

Lea Interfaces explícitas e implícitas en línea:
https://riptutorial.com/es/fortran/topic/2882/interfaces-explicitas-e-implicitas

Capítulo 8: Interoperabilidad C

Examples

Llamando C desde Fortran

Fortran 2003 introdujo características de lenguaje que pueden garantizar la interoperabilidad entre C y Fortran (y para más idiomas utilizando C como intermediario). A estas funciones se accede principalmente a través del módulo intrínseco `iso_c_binding` :

```
use, intrinsic :: iso_c_binding
```

La palabra clave `intrinsic` aquí garantiza que se use el módulo correcto, y no un módulo creado por el usuario con el mismo nombre.

`iso_c_binding` da acceso a parámetros de tipo de tipo *interoperables* :

```
integer(c_int) :: foo    ! equivalent of 'int foo' in C
real(c_float) :: bar     ! equivalent of 'float bar' in C
```

El uso de parámetros de tipo C garantiza que los datos pueden transferirse entre los programas C y Fortran.

La interoperabilidad de los caracteres C char y Fortran es probablemente un tema por sí mismo y, por lo tanto, no se trata aquí.

Para llamar realmente a una función C desde Fortran, primero se debe declarar la interfaz. Esto es esencialmente equivalente al prototipo de la función C, y le permite al compilador conocer el número y el tipo de los argumentos, etc. El atributo de `bind` se usa para decirle al compilador el nombre de la función en C, que puede ser diferente al de Fortran. nombre.

gansos.h

```
// Count how many geese are in a given flock
int howManyGeese(int flock);
```

gansos f90

```
! Interface to C routine
interface
  integer(c_int) function how_many_geese(flock_num) bind(C, 'howManyGeese')
    ! Interface blocks don't know about their context,
    ! so we need to use iso_c_binding to get c_int definition
    use, intrinsic :: iso_c_binding, only : c_int
    integer(c_int) :: flock_num
  end function how_many_geese
end interface
```

El programa de Fortran debe estar vinculada con la librería C *(compilador dependiente, se incluyen aquí?),* Que incluye la implementación de `howManyGeese()` , y luego `how_many_geese()` puede ser llamado desde Fortran.

C structs en fortran

El `bind` atributo también se puede aplicar a tipos derivados:

gansos.h

```
struct Goose {
   int flock;
   float buoyancy;
}

struct Goose goose_c;
```

gansos f90

```
use, intrinsic :: iso_c_binding, only : c_int, c_float

type, bind(C) :: goose_t
  integer(c_int) :: flock
  real(c_float) :: buoyancy
end type goose_t

type(goose_t) :: goose_f
```

Los datos ahora se pueden transferir entre `goose_c` y `goose_f` . Las rutinas C que toman argumentos de tipo `Goose` se pueden llamar desde Fortran con `type(goose_t)` .

Lea Interoperabilidad C en línea: https://riptutorial.com/es/fortran/topic/2184/interoperabilidad-c

Capítulo 9: Procedimientos - Funciones y subrutinas.

Observaciones

Las funciones y las *subrutinas* , junto con los *módulos* , son las herramientas para dividir un *programa* en unidades. Esto hace que el programa sea más legible y manejable. Cada una de estas unidades puede considerarse como parte del código que, idealmente, podría compilarse y probarse de forma aislada. Los programas principales pueden llamar (o invocar) a dichos subprogramas (funciones o subrutinas) para realizar una tarea.

Las funciones y las subrutinas son diferentes en el siguiente sentido:

- **Las funciones** devuelven un solo objeto y, generalmente, no alteran los valores de sus argumentos (es decir, ¡actúan como una función matemática);
- **Las subrutinas** generalmente realizan una tarea más complicada y generalmente alteran sus argumentos (si hay alguno presente), así como otras variables (por ejemplo, aquellas declaradas en el módulo que contiene la subrutina).

Las funciones y subrutinas van colectivamente bajo el nombre de *procedimientos* . (En lo siguiente, usaremos el verbo "call" como sinónimo de "invocar" incluso si técnicamente los procedimientos a ser `call` son `subroutine` , mientras que las `function` s aparecen como parte derecha de la asignación o en expresiones).

Examples

Sintaxis funcional

Las funciones se pueden escribir utilizando varios tipos de sintaxis.

```
function name()
  integer name
  name = 42
end function
```

```
integer function name()
  name = 42
end function
```

```
function name() result(res)
  integer res
  res = 42
end function
```

Las funciones devuelven valores a través de un *resultado de función* . A menos que la declaración de función tenga una cláusula de `result` el `result` la función tendrá el mismo nombre que la

función. Con `result` el `result` la función es el dado por el `result` . En cada uno de los dos primeros ejemplos anteriores, el resultado de la función viene dado por `name` ; en el tercero por `res` .

El resultado de la función debe definirse durante la ejecución de la función.

Las funciones permiten utilizar algunos prefijos especiales.

Función *pura* significa que esta función no tiene ningún efecto secundario:

```
pure real function square(x)
  real, intent(in) :: x
  square = x * x
end function
```

La función *elemental* se define como operador escalar, pero se puede invocar con array como argumento real, en cuyo caso la función se aplicará de forma elemental. A menos que se especifique el prefijo `impure` (introducido en Fortran 2008), una función *elemental* también es una función *pura* .

```
elemental real function square(x)
  real, intent(in) :: x
  square = x * x
end function
```

Declaración de devolución

La declaración de `return` se puede utilizar para salir de la función y la subrutina. A diferencia de muchos otros lenguajes de programación, no se utiliza para establecer el valor de retorno.

```
real function f(x)
  real, intent(in) :: x
  integer :: i

  f = x

  do i = 1, 10

    f = sqrt(f) - 1.0

    if (f < 0) then
      f = -1000.
      return
    end if

  end do
end function
```

Esta función realiza un cálculo iterativo. Si el valor de `f` vuelve negativo, la función devuelve el valor -1000.

Procedimientos recursivos

En Fortran, las funciones y las subrutinas deben declararse explícitamente como *recursivas* , si han de llamarse a sí mismas de nuevo, directa o indirectamente. Por lo tanto, una implementación recursiva de la serie Fibonacci podría tener este aspecto:

```
recursive function fibonacci(term) result(fibo)
  integer, intent(in) :: term
  integer :: fibo

  if (term <= 1) then
    fibo = 1
  else
    fibo = fibonacci(term-1) + fibonacci(term-2)
  end if

end function fibonacci
```

Otro ejemplo es permitido calcular factorial:

```
recursive function factorial(n)  result(f)
  integer :: f
  integer, intent(in) :: n

  if(n == 0) then
    f = 1
  else
    f = n * f(n-1)
  end if
end function factorial
```

Para que una función se refiera directamente a sí misma, su definición debe usar el sufijo de `result` . No es posible que una función sea tanto `recursive` como `elemental` .

La intención de los argumentos ficticios

El atributo de `intent` de un argumento ficticio en una subrutina o función declara su uso previsto. La sintaxis es una de

```
intent(IN)
intent(OUT)
intent(INOUT)
```

Por ejemplo, considere esta función:

```
real function f(x)
  real, intent(IN) :: x

  f = x*x
end function
```

La `intent(IN)` especifica que el argumento ficticio (sin puntero) `x` nunca se puede definir o perder su definición a lo largo de la función o su inicialización. Si un argumento ficticio de puntero tiene el atributo `intent(IN)` , esto se aplica a su asociación.

`intent(OUT)` para un argumento ficticio no puntero significa que el argumento ficticio se convierte en indefinido en la invocación del subprograma (a excepción de cualquier componente de un tipo derivado con inicialización predeterminada) y se debe establecer durante la ejecución. El argumento real pasado como argumento ficticio debe ser definible: no se permite pasar una constante con nombre o literal, o una expresión.

De forma similar a antes, si un argumento ficticio de puntero es `intent(OUT)` el estado de asociación del puntero se vuelve indefinido. El argumento real aquí debe ser una variable de puntero.

`intent(INOUT)` especifica que el argumento real es definible y es adecuado para pasar y devolver datos del procedimiento.

Finalmente, un argumento ficticio puede estar sin el atributo de `intent` . Tal argumento ficticio tiene su uso limitado por el argumento real pasado.

Por ejemplo, considere

```
integer :: i = 0
call sub(i, .TRUE.)
call sub(1, .FALSE.)

end

subroutine sub(i, update)
  integer i
  logical, intent(in) :: update
  if (update) i = i+1
end subroutine
```

El argumento `i` puede tener ningún atributo de `intent` que permita las dos llamadas de subrutina del programa principal.

Haciendo referencia a un procedimiento

Para que una función o subrutina sea útil debe ser referenciada. Se hace referencia a una subrutina en una instrucción de `call`

```
call sub(...)
```

y una función dentro de una expresión. A diferencia de muchos otros idiomas, una expresión no forma una declaración completa, por lo que una referencia de función se ve a menudo en una instrucción de asignación o se usa de alguna otra manera:

```
x = func(...)
y = 1 + 2*func(...)
```

Hay tres formas de designar un procedimiento al que se hace referencia:

- como el nombre de un procedimiento o puntero de procedimiento

- un componente de procedimiento de un objeto de tipo derivado
- un nombre de enlace de procedimiento vinculado

El primero puede verse como

```
procedure(), pointer :: sub_ptr=>sub
call sub()   ! With no argument list the parentheses are optional
call sub_ptr()
end

subroutine sub()
end subroutine
```

y los dos últimos como

```
module mod
  type t
    procedure(sub), pointer, nopass :: sub_ptr=>sub
  contains
    procedure, nopass :: sub
  end type

contains

  subroutine sub()
  end subroutine

end module

use mod
type(t) x
call x%sub_ptr()   ! Procedure component
call x%sub()       ! Binding name

end
```

Para un procedimiento con argumentos ficticios, la referencia requiere argumentos *reales* correspondientes, aunque es posible que no se proporcionen argumentos ficticios opcionales.

Considere la subrutina

```
subroutine sub(a, b, c)
  integer a, b
  integer, optional :: c
end subroutine
```

Esto puede ser referenciado de las siguientes dos maneras

```
call sub(1, 2, 3)    ! Passing to the optional dummy c
call sub(1, 2)       ! Not passing to the optional dummy c
```

Esto se denomina referencia de *posición* : los argumentos reales se asocian según la posición en las listas de argumentos. Aquí, el dummy `a` está asociado con `1` , `b` con `2` y `c` (cuando se

especifica) con `3` .

Alternativamente, la referencia de *palabras clave* se puede usar cuando el procedimiento tiene una interfaz explícita disponible

```
call sub(a=1, b=2, c=3)
call sub(a=1, b=2)
```

que es lo mismo que el anterior.

Sin embargo, con palabras clave los argumentos reales se pueden ofrecer en cualquier orden

```
call sub(b=2, c=3, a=1)
call sub(b=2, a=1)
```

Se pueden usar referencias posicionales y de palabras clave.

```
call sub(1, c=3, b=2)
```

siempre que se proporcione una palabra clave para cada argumento después de la primera aparición de una palabra clave

```
call sub(b=2, 1, 3)  ! Not valid: all keywords must be specified
```

El valor de la referencia de palabras clave es particularmente pronunciado cuando hay múltiples argumentos ficticios opcionales, como se ve a continuación si en la definición de subrutina anterior `b` también eran opcionales

```
call sub(1, c=3)  ! Optional b is not passed
```

Las listas de argumentos para procedimientos vinculados a tipo o punteros de procedimiento de componente con un argumento pasado se consideran por separado.

Lea Procedimientos - Funciones y subrutinas. en línea: https://riptutorial.com/es/fortran/topic/1106/procedimientos---funciones-y-subrutinas-

Capítulo 10: Procedimientos intrínsecos

Observaciones

Muchos de los procedimientos intrínsecos disponibles tienen tipos de argumentos en común. Por ejemplo:

- un argumento lógico `MASK` que selecciona elementos de las matrices de entrada para ser procesados
- un argumento escalar entero `KIND` que determina el tipo de resultado de la función
- un argumento entero `DIM` para una función de reducción que controla la dimensión sobre la que se realiza la reducción

Examples

Uso de PACK para seleccionar elementos que cumplan una condición

La función de `pack` intrínseco `pack` una matriz en un vector, seleccionando elementos basados en una máscara dada. La función tiene dos formas.

```
PACK(array, mask)
PACK(array, mask, vector)
```

(Es decir, el argumento del `vector` es opcional).

En ambos casos, la `array` es una matriz, y una `mask` de tipo lógico y compatible con la `array` (ya sea un escalar o una matriz de la misma forma).

En el primer caso, el resultado es una matriz de rango 1 de tipo y los parámetros de tipo de `array` con el número de elementos siendo el número de elementos verdaderos en la máscara.

```
integer, allocatable :: positive_values(:)
integer :: values(5) = [2, -1, 3, -2, 5]
positive_values = PACK(values, values>0)
```

resulta en `positive_values` siendo la matriz `[2, 3, 5]` .

Con el argumento `vector` rango-1 presente, el resultado ahora es el tamaño del `vector` (que debe tener al menos tantos elementos como valores reales en la `mask` .

El efecto con `vector` es devolver esa matriz con los elementos iniciales de esa matriz sobrescritos por los elementos enmascarados de la `array` . Por ejemplo

```
integer, allocatable :: positive_values(:)
integer :: values(5) = [2, -1, 3, -2, 5]
positive_values = PACK(values, values>0, [10,20,30,40,50])
```

resulta en `[2,3,5,40,50]` `positive_values` siendo la matriz `[2,3,5,40,50]` .

Se debe tener en cuenta que, independientemente de la forma de la `array` argumentos, el resultado es siempre una matriz de rango 1.

Además de seleccionar los elementos de una matriz que cumplen una condición de enmascaramiento, a menudo es útil determinar los índices para los cuales se cumple la condición de enmascaramiento. Este lenguaje común se puede expresar como

```
integer, allocatable :: indices(:)
integer i
indices = PACK([(i, i=1,5)], [2, -1, 3, -2, 5]>0)
```

dando como resultado `indices` son la matriz `[1,3,5]` .

Lea Procedimientos intrínsecos en línea:
https://riptutorial.com/es/fortran/topic/2643/procedimientos-intrinsecos

Capítulo 11: Programación orientada a objetos

Examples

Definición de tipo derivado

Fortran 2003 introdujo el soporte para la programación orientada a objetos. Esta característica permite aprovechar las modernas técnicas de programación. Los tipos derivados se definen con la siguiente forma:

```
TYPE [[, attr-list] :: ] name [(name-list)]
   [def-stmts]
   [PRIVATE statement or SEQUENCE statement]. . .
   [component-definition]. . .
   [procedure-part]
END TYPE [name]
```

dónde,

- **attr-list** - una lista de especificadores de atributos
- **nombre** - el nombre del tipo de datos derivado
- **nombre-lista** - una lista de nombres de parámetros de tipo separados por comas
- **def-stmts** : una o más declaraciones INTEGER de los parámetros de tipo nombrados en la lista de nombres
- **definición de componente** : una o más declaraciones de declaración de tipo o instrucciones de puntero de procedimiento que definen el componente de tipo derivado
- **procedure-part** : una declaración CONTAINS, opcionalmente seguida de una declaración PRIVATE y una o más instrucciones vinculantes de procedimiento

Ejemplo:

```
type shape
    integer :: color
end type shape
```

Tipo de Procedimientos

Para obtener un comportamiento similar a una clase, el tipo y los procedimientos relacionados (subrutina y funciones) se colocarán en un módulo:

Ejemplo:

```
module MShape
    implicit none
    private
```

```
    type, public :: Shape
    private
        integer :: radius
    contains
        procedure :: set   => shape_set_radius
        procedure :: print => shape_print
    end type Shape

contains
    subroutine shape_set_radius(this, value)
        class(Shape), intent(in out) :: self
        integer, intent(in)          :: value

        self%radius = value
    end subroutine shape_set_radius

    subroutine shape_print(this)
        class(Shape), intent(in) :: self

        print *, 'Shape: r = ', self%radius
    end subroutine shape_print
end module MShape
```

Más adelante, en un código, podemos usar esta clase de Forma de la siguiente manera:

```
! declare a variable of type Shape
type(Shape) :: shape

! call the type-bound subroutine
call shape%set(10)
call shape%print
```

Tipos derivados abstractos

Un tipo derivado extensible puede ser *abstracto*

```
type, abstract :: base_type
end type
```

Tal tipo derivado nunca puede ser instanciado, tal como por

```
type(base_type) t1
allocate(type(base_type) :: t2)
```

pero un objeto polimórfico puede tener esto como su tipo declarado

```
class(base_type), allocatable :: t1
```

o

```
function f(t1)
  class(base_type) t1
end function
```

Los tipos abstractos pueden tener componentes y procedimientos de tipo unido

```
type, abstract :: base_type
  integer i
contains
  procedure func
  procedure(func_iface), deferred :: def_func
end type
```

El procedimiento `def_func` es un procedimiento de tipo *diferido* con interfaz `func_iface` . Este tipo de procedimiento diferido debe ser implementado por cada tipo de extensión.

Extensión de tipo

Un tipo derivado es *extensible* si no tiene el atributo de `bind` ni el atributo de `sequence` . Tal tipo puede ser extendido por otro tipo.

```
module mod

  type base_type
    integer i
  end type base_type

  type, extends(base_type) :: higher_type
    integer j
  end type higher_type

end module mod
```

Una variable polimórfica con el tipo `base_type` declarado es compatible con el tipo `higher_type` y puede tenerlo como tipo dinámico

```
class(base_type), allocatable :: obj
allocate(obj, source=higher_type(1,2))
```

La compatibilidad de tipos desciende a través de una cadena de hijos, pero un tipo puede extenderse solo a otro tipo.

Un tipo derivado extendido hereda los procedimientos de tipo enlazado del padre, pero esto puede ser anulado

```
module mod

  type base_type
  contains
    procedure :: sub => sub_base
  end type base_type

  type, extends(base_type) :: higher_type
  contains
    procedure :: sub => sub_higher
  end type higher_type
```

```
contains

  subroutine sub_base(this)
    class(base_type) this
  end subroutine sub_base

  subroutine sub_higher(this)
    class(higher_type) this
  end subroutine sub_higher

end module mod

program prog
  use mod

  class(base_type), allocatable :: obj

  obj = base_type()
  call obj%sub

  obj = higher_type()
  call obj%sub

end program
```

Tipo constructor

Se pueden crear constructores personalizados para tipos derivados utilizando una interfaz para sobrecargar el nombre del tipo. De esta manera, los argumentos de palabras clave que no corresponden a componentes pueden usarse al construir un objeto de ese tipo.

```
module ball_mod
  implicit none

  ! only export the derived type, and not any of the
  ! constructors themselves
  private
  public :: ball

  type :: ball_t
     real :: mass
  end type ball_t

  ! Writing an interface overloading 'ball_t' allows us to
  ! overload the type constructor
  interface ball_t
     procedure :: new_ball
  end interface ball_t

contains

  type(ball_t) function new_ball(heavy)
    logical, intent(in) :: heavy

    if (heavy) then
       new_ball%mass = 100
    else
       new_ball%mass = 1
```

```
    end if

  end function new_ball

end module ball_mod

program test
  use ball_mod
  implicit none

  type(ball_t) :: football
  type(ball_t) :: boulder

  ! sets football%mass to 4.5
  football = ball_t(4.5)
  ! calls 'ball_mod::new_ball'
  boulder = ball_t(heavy=.true.)
end program test
```

Esto se puede usar para hacer una API más ordenada que usar rutinas de inicialización separadas:

```
subroutine make_heavy_ball(ball)
  type(ball_t), intent(inout) :: ball
  ball%mass = 100
end subroutine make_heavy_ball

...

call make_heavy_ball(boulder)
```

Lea Programación orientada a objetos en línea: https://riptutorial.com/es/fortran/topic/2374/programacion-orientada-a-objetos

Capítulo 12: Tipos de datos

Examples

Tipos intrinsecos

Los siguientes son tipos de datos *intrínsecos* a Fortran:

```
integer
real
character
complex
logical
```

`integer` , `real` y `complex` son tipos numéricos.

`character` es un tipo utilizado para almacenar cadenas de caracteres.

`logical` se utiliza para almacenar valores binarios `.true.` o `.false.` .

Todos los tipos intrínsecos numéricos y lógicos se parametrizan utilizando tipos.

```
integer(kind=specific_kind)
```

o solo

```
integer(specific_kind)
```

donde `specific_kind` es un entero llamado constante.

Las variables de caracteres, además de tener un parámetro de clase, también tienen un parámetro de longitud:

```
character char
```

declara que `char` es una variable de longitud-1 carácter de tipo predeterminado, mientras que

```
character(len=len) name
```

declara que el `name` es una variable de carácter del tipo predeterminado y la longitud `len` . El tipo también puede ser especificado

```
character(len=len, kind=specific_kind) name
character(kind=specific_kind) char
```

declara que el `name` es un carácter de tipo `kind` y longitud `len` . `char` es una longitud de 1 carácter de tipo `kind` .

Alternativamente, la forma obsoleta para la declaración de caracteres.

```
character*len  name
```

puede verse en el código anterior, declarando que el `name` es de longitud `len` y tipo de carácter predeterminado.

La declaración de una variable de tipo intrínseco puede ser de la forma anterior, pero también puede usar la forma `type(...)` :

```
integer i
real x
double precision y
```

es equivalente a (pero se prefiere mucho más que)

```
type(integer) i
type(real) x
type(double precision) y
```

Tipos de datos derivados

Definir un nuevo tipo, `mytype` :

```
type :: mytype
  integer :: int
  real    :: float
end type mytype
```

Declara una variable de tipo *mytype* :

```
type(mytype) :: foo
```

Se puede acceder a los componentes de un tipo derivado con el operador `%` [1] :

```
foo%int = 4
foo%float = 3.142
```

Una característica de Fortran 2003 (aún no implementada por todos los compiladores) permite definir tipos de datos parametrizados:

```
type, public :: matrix(rows, cols, k)
  integer, len :: rows, cols
  integer, kind :: k = kind(0.0)
  real(kind = k), dimension(rows, cols) :: values
end type matrix
```

La `matrix` tipos derivada tiene tres parámetros de tipo que se enumeran entre paréntesis después

del nombre de tipo (son `rows` , `cols` `k`). En la declaración de cada parámetro de tipo debe indicarse si son parámetros de tipo kind (`kind`) o length (`len`).

Los parámetros de tipo de clase, como los de los tipos intrínsecos, deben ser expresiones constantes, mientras que los parámetros de tipo de longitud, como la longitud de una variable de carácter intrínseco, pueden variar durante la ejecución.

Tenga en cuenta que el parámetro `k` tiene un valor predeterminado, por lo que se puede proporcionar u omitir cuando se declara una variable de tipo `matrix` , de la siguiente manera

```
type (matrix (55, 65, kind=double)) :: b, c ! default parameter provided
type (matrix (rows=40, cols=50)     :: m    ! default parameter omitted
```

El nombre de un tipo derivado no puede ser de `doubleprecision` o el mismo que cualquiera de los tipos intrínsecos.

---

1. Muchas personas se preguntan por qué Fortran usa `%` como operador de acceso a componentes, en lugar de los más comunes `.` . Esto se debe a `.` ya está en la sintaxis del operador, es decir, `.not.` , `.and.` , `.my_own_operator.` .

Precisión de los números en coma flotante.

Los números de punto flotante de tipo `real` no pueden tener ningún valor real. Pueden representar números reales hasta cierta cantidad de dígitos decimales.

FORTRAN 77 garantizó dos tipos de punto flotante y las normas más recientes garantizan al menos dos tipos reales. Las variables reales pueden ser declaradas como

```
real x
double precision y
```

`x` aquí es un tipo real por defecto e `y` es un tipo real con mayor precisión decimal que `x` . En Fortran 2008, la precisión decimal de `y` es al menos 10 y su rango de exponente decimal al menos 37.

```
 real, parameter             :: single = 1.12345678901234567890
 double precision, parameter :: double = 1.12345678901234567890d0

 print *, single
 print *, double
```

huellas dactilares

```
   1.12345684
   1.1234567890123457
```

en compiladores comunes utilizando la configuración por defecto.

Observe la `d0` en la constante de doble precisión. Un literal real que contiene `d` lugar de `e` para denotar el exponente se usa para indicar doble precisión.

```
! Default single precision constant
1.23e45
! Double precision constant
1.23d45
```

Fortran 90 introdujo tipos `real` parametrizados utilizando tipos. El tipo de un tipo real es un entero llamado constante o constante literal:

```
real(kind=real_kind) :: x
```

o solo

```
real(real_kind) :: x
```

Esta declaración declara que `x` es de tipo `real` con una cierta precisión dependiendo del valor de `real_kind` .

Los literales de punto flotante se pueden declarar con un tipo específico usando un sufijo

```
1.23456e78_real_kind
```

El valor exacto de `real_kind` no está estandarizado y difiere de un compilador a otro. Para investigar el tipo de cualquier variable real o constante, la función `kind()` se puede usar:

```
print *, kind(1.0), kind(1.d0)
```

normalmente se imprimirá

```
4 8
```

o

```
1 2
```

Dependiendo del compilador.

Los números de clase se pueden establecer de varias maneras:

1. Precisión simple (por defecto) y doble:

   ```
   integer, parameter :: single_kind = kind(1.)
   integer, parameter :: double_kind = kind(1.d0)
   ```

2. Uso de la función intrínseca `selected_real_kind([p, r])` para especificar la precisión decimal requerida. El tipo devuelto tiene una precisión de al menos `p` dígitos y permite un exponente de al menos `r` .

```
integer, parameter :: single_kind = selected_real_kind( p=6, r=37 )
integer, parameter :: double_kind = selected_real_kind( p=15, r=200 )
```

3. A partir de Fortran 2003, las constantes predefinidas están disponibles a través del módulo intrínseco `ISO_C_Binding` para garantizar que las clases reales sean interoperables con los tipos `float` , `double` o `long_double` del compilador C que lo acompaña:

```
use ISO_C_Binding

integer, parameter :: single_kind = c_float
integer, parameter :: double_kind = c_double
integer, parameter :: long_kind = c_long_double
```

4. A partir de Fortran 2008, las constantes predefinidas están disponibles a través del módulo intrínseco `ISO_Fortran_env` . Estas constantes proporcionan tipos reales con cierto tamaño de almacenamiento en bits

```
use ISO_Fortran_env

integer, parameter :: single_kind = real32
integer, parameter :: double_kind = real64
integer, parameter :: quadruple_kind = real128
```

Si cierto tipo no está disponible en el compilador, el valor devuelto por `selected_real_kind()` o el valor de la constante entera es `-1` .

Parámetros de tipo de longitud asumida y diferida

Las variables de tipo de carácter o de un tipo derivado con parámetro de longitud pueden tener el parámetro de longitud *asumido* o *diferido* . El `name` variable de caracteres.

```
character(len=len) name
```

Es de longitud `len` largo de la ejecución. Por el contrario, el especificador de longitud puede ser

```
character(len=*) ...  ! Assumed length
```

o

```
character(len=:) ...  ! Deferred length
```

Las variables de longitud asumidas asumen su longitud de otra entidad.

En la funcion

```
function f(dummy_name)
  character(len=*) dummy_name
end function f
```

el argumento ficticio `dummy_name` tiene la longitud del argumento real.

La constante nombrada `const_name` en

```
character(len=*), parameter :: const_name = 'Name from which length is assumed'
```

tiene la longitud dada por la expresión constante en el lado derecho.

Los parámetros de tipo de longitud diferida pueden variar durante la ejecución. Una variable con longitud diferida debe tener el atributo `allocatable` o `pointer`

```
character(len=:), allocatable :: alloc_name
character(len=:), pointer :: ptr_name
```

La longitud de dicha variable se puede establecer de cualquiera de las siguientes maneras

```
allocate(character(len=5) :: alloc_name, ptr_name)
alloc_name = 'Name'         ! Using allocation on intrinsic assignment
ptr_name => another_name    ! For given target
```

Para los tipos derivados con parametrización de longitud, la sintaxis es similar.

```
  type t(len)
    integer, len :: len
    integer i(len)
  end type t

  type(t(:)), allocatable :: t1
  type(t(5)) t2

  call sub(t2)
  allocate(type(t(5)) :: t1)

contains

  subroutine sub(t2)
    type(t(*)), intent(out) :: t2
  end subroutine sub

end
```

Constantes literales

Las unidades de programa a menudo hacen uso de constantes literales. Estos cubren los casos obvios como

```
print *, "Hello", 1, 1.0
```

Excepto en un caso, cada constante literal es un escalar que tiene tipo, parámetros de tipo y valor dado por la sintaxis.

Las constantes literales enteras son de la forma

```
1
-1
-1_1   ! For valid kind parameter 1
1_ik   ! For the named constant ik being a valid kind paramter
```

Las constantes literales reales son de la forma

```
1.0    ! Default real
1e0    ! Default real using exponent format
1._1   ! Real with kind parameter 1 (if valid)
1.0_sp ! Real with kind paramter named constant sp
1d0    ! Double precision real using exponent format
1e0_dp ! Real with kind named constant dp using exponent format
```

Las constantes literales complejas son de la forma

```
(1, 1.)       ! Complex with integer and real components, literal constants
(real, imag)  ! Complex with named constants as components
```

Si los componentes real e imaginario son ambos enteros, la constante literal compleja es compleja por defecto, y los componentes enteros se convierten en real por defecto. Si un componente es real, el parámetro kind de la constante literal compleja es el de lo real (y el componente entero se convierte a ese tipo real). Si ambos componentes son reales, la constante literal compleja es de la clase de lo real con la mayor precisión.

Constantes lógicas literales son

```
.TRUE.     ! Default kind, with true value
.FALSE.    ! Default kind, with false value
.TRUE._1   ! Of kind 1 (if valid), with true value
.TRUE._lk  ! Of kind named constant lk (if valid), with true value
```

Los valores literales de caracteres difieren ligeramente en concepto, ya que el especificador de clase precede al valor

```
"Hello"       ! Character value of default kind
'Hello'       ! Character value of default kind
ck_"Hello"    ! Character value of kind ck
"'Bye"        ! Default kind character with a '
'''Bye'       ! Default kind character with a '
""            ! A zero-length character of default kind
```

Como se sugirió anteriormente, las constantes literales de los caracteres deben estar delimitadas por apóstrofes o comillas, y los marcadores de inicio y final deben coincidir. Los apóstrofes literales se pueden incluir estando dentro de los delimitadores de comillas o apareciendo duplicados. Lo mismo para las comillas.

Las constantes de BOZ son distintas de las anteriores, ya que solo especifican un valor: no tienen tipo ni parámetro de tipo. Una constante BOZ es un patrón de bits y se especifica como

```
B'00000'    ! A binary bit pattern
B"01010001" ! A binary bit pattern
O'012517'   ! An octal bit pattern
O"1267671"  ! An octal bit pattern
Z'0A4F'     ! A hexadecimal bit pattern
Z"FFFFFF"   ! A hexadecimal bit pattern
```

Las constantes literales de BOZ están limitadas en donde pueden aparecer: como constantes en declaraciones de `data` y una selección de procedimientos intrínsecos.

Acceso a subcadenas de caracteres

Para la entidad del personaje.

```
character(len=5), parameter :: greeting = "Hello"
```

Una subcadena puede ser referenciada con la sintaxis.

```
greeting(2:4)  ! "ell"
```

Para acceder a una sola letra no basta con escribir.

```
greeting(1)    ! This isn't the letter "H"
```

pero

```
greeting(1:1)  ! This is "H"
```

Para una matriz de caracteres

```
character(len=5), parameter :: greeting(2) = ["Hello", "Yo!  "]
```

tenemos subcadena de acceso como

```
greeting(1)(2:4)  ! "ell"
```

pero no podemos hacer referencia a los caracteres no contiguos

```
greeting(:)(2:4)  ! The parent string here is an array
```

Incluso podemos acceder a subcadenas de constantes literales.

```
"Hello"(2:4)
```

Una parte de una variable de carácter también se puede definir utilizando una subcadena como variable. Por ejemplo

```
integer :: i=1
character :: filename = 'file000.txt'

filename(9:11) = 'dat'
write(filename(5:7), '(I3.3)') i
```

Accediendo a componentes complejos.

La entidad compleja

```
complex, parameter :: x = (1., 4.)
```

Tiene parte real `1.` y parte compleja `4.` .. Podemos acceder a estos componentes individuales como

```
real(x)  ! The real component
aimag(x) ! The complex component
x%re     ! The real component
y%im     ! The complex component
```

La forma `x%..` es nueva para Fortran 2008 y no es ampliamente compatible con compiladores. Sin embargo, esta forma se puede usar para establecer directamente los componentes individuales de una variable compleja

```
complex y
y%re = 0.
y%im = 1.
```

Declaración y atributos.

A lo largo de los temas y ejemplos aquí veremos muchas declaraciones de variables, funciones, etc.

Además de su nombre, los objetos de datos pueden tener *atributos* . Cubierto en este tema son declaraciones de declaración como

```
integer, parameter :: single_kind = kind(1.)
```

que le da al objeto `single_kind` el atributo de `parameter` (por lo que es una constante con nombre).

Hay muchos otros atributos, como

- `target`
- `pointer`
- `optional`
- `save`

Los atributos pueden especificarse con las denominadas *declaraciones de especificación de atributos*

```
integer i     ! i is an integer (of default kind)...
pointer i     ! ... with the POINTER attribute...
optional i    ! ... and the OPTIONAL attribute
```

Sin embargo, generalmente se considera que es mejor evitar el uso de estas declaraciones de especificación de atributos. Para mayor claridad, los atributos pueden especificarse como parte de una sola declaración.

```
integer, pointer, optional :: i
```

Esto también reduce la tentación de usar la escritura implícita.

En la mayoría de los casos, en esta documentación de Fortran, se prefiere esta declaración única.

Lea Tipos de datos en línea: https://riptutorial.com/es/fortran/topic/939/tipos-de-datos

Capítulo 13: Unidades de programa y diseño de archivos.

Examples

Programas de fortran

Un programa completo de Fortran se compone de varias unidades de programa distintas. Las unidades del programa son:

- programa principal
- función o subprograma de subrutina
- módulo o submódulo
- unidad de programa de datos de bloque

El programa principal y algunos subprogramas de procedimiento (función o subrutina) pueden ser proporcionados por un idioma que no sea Fortran. Por ejemplo, un programa principal de C puede llamar a una función definida por un subprograma de funciones de Fortran, o un programa principal de Fortran puede llamar a un procedimiento definido por C.

Estas unidades del programa Fortran pueden ser archivos distintos o dentro de un solo archivo.

Por ejemplo, podemos ver los dos archivos:

prog.f90

```
program main
  use mod
end program main
```

mod.f90

```
module mod
end module mod
```

Y el compilador (invocado correctamente) podrá asociar el programa principal con el módulo.

El único archivo puede contener muchas unidades de programa

todo.f90

```
module mod
end module mod

program prog
  use mod
end program prog
```

```
function f()
end function f()
```

En este caso, sin embargo, debe tenerse en cuenta que la función `f` sigue siendo una *función externa* en lo que respecta al programa principal y el módulo. Sin embargo, el programa principal estará accesible para el módulo.

Las reglas de alcance de escritura se aplican a cada unidad de programa individual y no al archivo en el que están contenidas. Por ejemplo, si queremos que cada unidad de alcance no tenga una escritura implícita, el archivo anterior debe escribirse como

```
module mod
  implicit none
end module mod

program prog
  use mod
  implicit none
end program prog

function f()
  implicit none
  <type> f
end function f
```

Módulos y submódulos.

Los módulos están documentados en otra parte .

Los compiladores a menudo generan los llamados *archivos de módulo* : generalmente el archivo que contiene

```
module my_module
end module
```

resultará en un archivo llamado algo así como `my_module.mod` por el compilador. En tales casos, para que un módulo sea accesible por una unidad de programa, ese archivo de módulo debe estar visible antes de que se procese esta última unidad de programa.

Procedimientos externos

Un procedimiento externo es uno que se define fuera de otra unidad de programa, o por un medio que no sea Fortran.

La función contenida en un archivo como

```
integer function f()
  implicit none
end function f
```

Es una función externa.

Para procedimientos externos, su existencia se puede declarar utilizando un bloque de interfaz (para una interfaz explícita)

```
program prog
  implicit none
  interface
    integer function f()
  end interface
end program prog
```

o por una declaración de declaración para dar una interfaz implícita

```
program prog
  implicit none
  integer, external :: f
end program prog
```

o incluso

```
program prog
  implicit none
  integer f
  external f
end program prog
```

El atributo `external` no es necesario:

```
program prog
  implicit none
  integer i
  integer f
  i = f()   ! f is now an external function
end program prog
```

Bloquear unidades de programa de datos.

Las unidades de programa de datos de bloque son unidades de programa que proporcionan valores iniciales para objetos en bloques comunes. Estos se dejan deliberadamente sin documentar aquí, y aparecerán en la documentación de las características históricas de Fortran.

Subprogramas internos

Una unidad de programa que no es un subprograma interno puede contener otras unidades de programa, llamadas *subprogramas internos* .

```
program prog
  implicit none
contains
  function f()
  end function f
```

```
  subroutine g()
  end subroutine g
end program
```

Dicho subprograma interno tiene una serie de características:

- existe una asociación de host entre las entidades en el subprograma y el programa externo
- las reglas de escritura implícitas se heredan (`implicit none` está vigente en `f` arriba)
- Los subprogramas internos tienen una interfaz explícita disponible en el servidor.

Los subprogramas de módulo y los subprogramas externos pueden tener subprogramas internos, como

```
module mod
  implicit none
contains
  function f()
  contains
    subroutine s()
    end subroutine s
  end function f
end module mod
```

Archivos de código fuente

Un archivo de código fuente es un (generalmente) archivo de texto plano que debe ser procesado por el compilador. Un archivo de código fuente puede contener hasta un programa principal y cualquier número de módulos y subprogramas externos. Por ejemplo, un archivo de código fuente puede contener lo siguiente

```
module mod1
end module mod1

module mod2
end module mod2

function func1()    ! An external function
end function func1

subroutine sub1()   ! An external subroutine
end subroutine sub1

program prog        ! The main program starts here...
end program prog    ! ... and ends here

function func2()    ! An external function
end function func2
```

Debemos recordar aquí que, aunque los subprogramas externos se encuentran en el mismo archivo que los módulos y el programa principal, los subprogramas externos no son conocidos explícitamente por ningún otro componente.

Alternativamente, los componentes individuales pueden estar distribuidos en múltiples archivos, e

incluso compilados en diferentes momentos. La documentación del compilador debe leerse sobre cómo combinar varios archivos en un solo programa.

Un solo archivo de código fuente puede contener un código fuente de forma fija o de forma libre: no se pueden mezclar, aunque varios archivos que se combinan en tiempo de compilación pueden tener diferentes estilos.

Para indicar al compilador el formulario de origen, generalmente hay dos opciones:

- elección del nombre de archivo sufijo
- uso de banderas del compilador

El indicador de tiempo de compilación para indicar el origen de forma libre o fija se puede encontrar en la documentación del compilador.

Los sufijos de nombre de archivo significativos también se encuentran en la documentación del compilador, pero como regla general, se considera que un archivo llamado `file.f90` contiene una fuente de forma libre, mientras que el archivo `file.f` contiene una fuente de forma fija.

El uso del sufijo `.f90` para indicar la fuente de forma libre (que se introdujo en el estándar Fortran 90) a menudo tienta al programador a usar el sufijo para indicar el estándar de idioma con el que se ajusta el código fuente. Por ejemplo, podemos ver archivos con sufijos `.f03` o `.f08` . Esto generalmente debe evitarse: la mayoría de las fuentes de Fortran 2003 también son compatibles con Fortran 77, Fortran 90/5 y Fortran 2008. Además, muchos comilers no consideran automáticamente tales sufijos.

Los compiladores también suelen ofrecer un preprocesador de código incorporado (generalmente basado en cpp). Una vez más, se puede usar un indicador de tiempo de compilación para indicar que el preprocesador debe ejecutarse antes de la compilación, pero el sufijo del archivo de código fuente también puede indicar dicho requisito de preprocesamiento.

Para los sistemas de archivos que `file.F` mayúsculas y minúsculas, el archivo `file.F` menudo se toma como un archivo fuente de forma fija que debe `file.F90` previamente y `file.F90` es un archivo fuente de forma libre que debe `file.F90` previamente. Como antes, se debe consultar la documentación del compilador para dichos indicadores y sufijos de archivos.

Lea Unidades de programa y diseño de archivos. en línea:
https://riptutorial.com/es/fortran/topic/2203/unidades-de-programa-y-diseno-de-archivos-

Capítulo 14: Uso de módulos

Examples

Sintaxis del modulo

El módulo es una colección de declaraciones de tipo, declaraciones de datos y procedimientos. La sintaxis básica es:

```
module module_name
  use other_module_being_used

  ! The use of implicit none here will set it for the scope of the module.
  ! Therefore, it is not required (although considered good practice) to repeat
  ! it in the contained subprograms.
  implicit none

  ! Parameters declaration
  real, parameter, public :: pi = 3.14159
  ! The keyword private limits access to e parameter only for this module
  real, parameter, private :: e = 2.71828

  ! Type declaration
  type my_type
    integer :: my_int_var
  end type

  ! Variable declaration
  integer :: my_integer_variable

! Subroutines and functions belong to the contains section
contains

  subroutine my_subroutine
    !module variables are accessible
    print *, my_integer_variable
  end subroutine

  real function my_func(x)
    real, intent(in) :: x
    my_func = x * x
  end function my_func
end module
```

Usando módulos de otras unidades de programa

Para acceder a las entidades declaradas en un módulo desde otra unidad de programa (módulo, procedimiento o programa), el módulo debe *usarse* con la declaración de `use` .

```
module shared_data
  implicit none

  integer :: iarray(4) = [1, 2, 3, 4]
```

```
  real :: rarray(4) = [1., 2., 3., 4.]
end module


program test

  !use statements most come before implicit none
  use shared_data

  implicit none

  print *, iarray
  print *, rarray
end program
```

La declaración de `use` permite importar solo los nombres seleccionados

```
program test

  !only iarray is accessible
  use shared_data, only: iarray

  implicit none

  print *, iarray

end program
```

También se puede acceder a las entidades con un nombre diferente utilizando una *lista de cambio de nombre* :

```
program test

  !only iarray is locally renamed to local_name, rarray is still acessible
  use shared_data, local_name => iarray

  implicit none

  print *, local_name

  print *, rarray

end program
```

Además, el cambio de nombre se puede combinar con la `only` opción

```
program test
  use shared_data, only : local_name => iarray
end program
```

de modo que solo se accede a la entidad del módulo `iarray` , pero tiene el nombre local `local_name` .

Si se selecciona para importar nombres de marca como *privados,* no puede importarlos a su programa.

Módulos intrínsecos

Fortran 2003 introdujo módulos intrínsecos que brindan acceso a constantes con nombre especiales, tipos derivados y procedimientos de módulos. Ahora hay cinco módulos intrínsecos estándar:

- `ISO_C_Binding` ; apoyando la interoperabilidad de C;
- `ISO_Fortran_env` ; detallando el entorno de Fortran;
- `IEEE_Exceptions` , `IEEE_Arithmetic` y `IEEE_Features` ; soportando la llamada facilidad aritmética IEEE.

Estos módulos intrínsecos son parte de la biblioteca Fortran y se accede a ellos como otros módulos, excepto que la declaración de `use` puede tener la naturaleza intrínseca explícitamente establecida:

```
use, intrinsic :: ISO_C_Binding
```

Esto garantiza que el módulo intrínseco se use cuando esté disponible un módulo proporcionado por el usuario con el mismo nombre. A la inversa

```
use, non_intrinsic :: ISO_C_Binding
```

asegura que se acceda al mismo módulo proporcionado por el usuario (que debe ser accesible) en lugar del módulo intrínseco. Sin la naturaleza del módulo especificado como en

```
use ISO_C_Binding
```

se preferirá un módulo no intrínseco disponible sobre el módulo intrínseco.

Los módulos IEEE intrínsecos son diferentes de otros módulos en que su accesibilidad en una unidad de alcance puede cambiar el comportamiento del código allí, incluso sin hacer referencia a ninguna de las entidades definidas en ellos.

Control de acceso

La accesibilidad de los símbolos declarados en un módulo puede controlarse utilizando atributos y declaraciones `private` y `public` .

Sintaxis del formulario de declaración:

```
!all symbols declared in the module are private by default
private

!all symbols declared in the module are public by default
public

!symbols in the list will be private
private :: name1, name2
```

```
!symbols in the list will be public
public :: name3, name4
```

Sintaxis de la forma de atributo:

```
integer, parameter, public :: maxn = 1000

real, parameter, private :: local_constant = 42.24
```

Se puede acceder a los símbolos públicos desde las unidades de programa que usan el módulo, pero los símbolos privados no pueden.

Cuando no se utiliza ninguna especificación, el valor predeterminado es `public` .

La especificación de acceso por defecto usando

```
private
```

o

```
public
```

se puede cambiar especificando un acceso diferente con *la lista de declaración de entidad*

```
public :: name1, name2
```

o utilizando atributos.

Este control de acceso también afecta a los símbolos importados de otro módulo:

```
module mod1
  integer :: var1
end module

module mod2
  use mod1, only: var1

  public
end module

program test
  use mod2, only: var1
end program
```

es posible, pero

```
module mod1
  integer :: var1
end module

module mod2
```

```
  use mod1, only: var1

  public
  private :: var1
end module

program test
  use mod2, only: var1
end program
```

no es posible porque `var` es privado en `mod2` .

Entidades de módulo protegido

Además de permitir que las entidades de módulo tengan control de acceso (ya sea `public` o `private`), las entidades de módulo también pueden tener el atributo de `protect` . Una entidad protegida pública puede estar asociada al uso, pero la entidad utilizada está sujeta a restricciones sobre su uso.

```
module mod
  integer, public, protected :: i=1
end module

program test
  use mod, only : i
  print *, i   ! We are allowed to get the value of i
  i = 2        ! But we can't change the value
end program test
```

Un objetivo protegido público no puede ser apuntado fuera de su módulo

```
module mod
  integer, public, target, protected :: i
end module mod

program test
  use mod, only : i
  integer, pointer :: j
  j => i   ! Not allowed, even though we aren't changing the value of i
end program test
```

Para un puntero público protegido en un módulo, las restricciones son diferentes. Lo que está protegido es el estado de asociación del puntero.

```
module mod
  integer, public, target :: j
  integer, public, protected, pointer :: i => j
end module mod

program test
  use mod, only : i
  i = 2   ! We may change the value of the target, just not the association status
end program test
```

Al igual que con los punteros variables, los punteros de procedimiento también pueden protegerse, impidiendo nuevamente el cambio de asociación objetivo.

Lea Uso de módulos en línea: https://riptutorial.com/es/fortran/topic/1139/uso-de-modulos

Creditos

S. No	Capítulos	Contributors
1	Empezando con Fortran	Alexander Vogt, Community, Enrico Maria De Angelis, Gilles, haraldkl, High Performance Mark, Ingve, innoSPG, milancurcic, packet0, RamenChef, Serenity, Vladimir F, Yossarian
2	Alternativas modernas a las características históricas.	Brian Tompsett - , d_1999, Enrico Maria De Angelis, francescalus, Serenity, TTT, Vladimir F, Yossarian
3	Arrays	Enrico Maria De Angelis, francescalus, G.Clavier, Gilles, Serenity, TTT, Vladimir F, Yossarian
4	Control de ejecución	Enrico Maria De Angelis, francescalus, haraldkl, ptev, Serenity, syscreat, TTT, Vladimir F
5	Extensiones de archivo fuente (.f, .f90, .f95, ...) y cómo se relacionan con el compilador.	Arun
6	I / O	AL-P, Ed Smith, francescalus, Kyle Kanos, TTT
7	Interfaces explícitas e implícitas	Enrico Maria De Angelis, Serenity, Vladimir F
8	Interoperabilidad C	Serenity, Yossarian
9	Procedimientos - Funciones y subrutinas.	Alexander Vogt, Enrico Maria De Angelis, francescalus, haraldkl , Serenity, Vladimir F, Yossarian
10	Procedimientos intrínsecos	francescalus
11	Programación orientada a objetos	Enrico Maria De Angelis, francescalus, syscreat, Yossarian
12	Tipos de datos	Alexander Vogt, Enrico Maria De Angelis, francescalus, Vladimir F, Yossarian
13	Unidades de	agentp, francescalus, haraldkl, trblnc

	programa y diseño de archivos.	
14	Uso de módulos	Alexander Vogt, Enrico Maria De Angelis, francescalus, Serenity, Vladimir F