

Tutorial Rápido de Fortran *(Incluye versiones Fortran 77 y 90)*

Julio R. Baeza Pereyra, Dr.

1. Prefacio

La meta de este documento acerca de Fortran es dar una rápida introducción a las características más comunes del lenguaje de programación F77/90/95. No se pretende que sea una referencia completa, muchos detalles han sido omitidos, la presentación del material se enfoca al cómputo científico, principalmente álgebra lineal.

2. ¿Qué es Fortran?

Fortran es lenguaje de propósito general, principalmente orientado a la computación matemática, por ejemplo en ingeniería. Fortran es un acrónimo de FORmula TRANslator, y originalmente fue escrito con mayúsculas como FORTRAN. Sin embargo la tendencia es poner sólo la primera letra con mayúscula, por lo que se escribe actualmente como Fortran. Fortran fue el primer lenguaje de programación de alto nivel. El desarrollo de Fortran inicio en la década de 1950 en IBM y han habido muchas versiones desde entonces. Por convención, una versión de Fortran es acompañada con los últimos dos dígitos del año en que se propuso la estandarización. Por lo que se tiene:

- Fortran 66
- Fortran 77
- Fortran 90 (95)
- Fortran 2000

La versión más común de Fortran actualmente es todavía Fortran 77 (*F77*), sin embargo Fortran 90 (*F90*) esta creciendo en popularidad. Fortran 95 es una versión revisada de F90 la cual fue aprobada por ANSI en 1996. Hay también varias versiones de Fortran para computadoras paralelas. La más importante de ellas es High Performance Fortran (*HPF*), la cual es de hecho el estándar.

Los usuarios deben ser cuidadosos con la versión 77 del lenguaje Fortran. Aunque los compiladores más recientes pueden trabajar con código desarrollado en dicho lenguaje, un programa F77 puede contener extensiones no estandarizadas. Así mismo, muchas de las características encontradas en compiladores mas recientes pueden no reconocer o en su defecto hacer caso omiso a las características anteriores. Sin embargo, los compiladores recientes generalmente soportan la sintaxis y los comandos de la versión F77, tales como la numeración de las líneas, la instrucción GOTO computada, así como el estilo de programación no estructurada.

Razones para aprender Fortran

Fortran es un lenguaje de programación dominante usado en muchas aplicaciones de ingeniería y matemáticas, por lo que es importante que se tengan bases para poder leer y modificar un código de Fortran. Algunas opiniones de expertos han dicho que Fortran será un lenguaje que pronto decaerá en popularidad y se extinguirá, lo que no ha sucedido todavía. Una de las razones para esta supervivencia es la inercia del software, ya que una vez que una compañía ha gastado muchos millones de dólares y de años en el desarrollo de software, no le es conveniente traducir el software a un lenguaje diferente, por el costo que implica y por ser una tarea difícil y laboriosa.

Portabilidad

Una ventaja principal de Fortran es que ha sido estandarizado por ANSI e ISO, por lo que un programa escrito en ANSI F77 se podrá ejecutar todavía en cualquier computadora que tenga un compilador de Fortran. Para continuar programando en compiladores Fortran mas recientes (tales como Fortran 2000/2005) es necesario crear código estructurado, hacer uso de comandos como CASE (que sustituye al GOTO computado) así como programar en módulos, entre otros detalles. En contrapartida, ya es posible crear programas basados en objetos y en algunos casos, tener a disposición de un entorno para programación basada en eventos y formularios, más acorde con las tendencias actuales.

Lo más aconsejable es aprender a construir código estructurado desde el principio. En este trabajo se hace énfasis en el uso de código que puede estar o no estructurado. Si el lector utiliza código antiguo o heredado, se le recomienda que los programas se desarrollen en el futuro se apeguen a los estándares de las versiones 90/95 y subsiguientes.

3. Introducción a Fortran

Un programa de Fortran es una secuencia de líneas de texto. El texto debe de seguir una determinada *sintaxis* para ser un programa válido de Fortran. Se analiza el siguiente ejemplo:

```
c F77
      program circulo
      Real r, area
c      Este programa lee un número real r
c      y muestra el área del círculo con radio r.
      write (*,*) 'Escribe el radio r:'
      read  (*,*) r
      area = 3.14159*r*r
      write (*,*) 'Area = ', area
      stop
      end

! F90 y siguientes
program circulo
  Real :: r, area
  ! Este programa lee un número real r y muestra
  ! el área del círculo con radio r.
  write (*,*) 'Escribe el radio r:'
  read  (*,*) r
  area = 3.14159*r*r
  write (*,*) 'Area = ', area
stop
end
```

Las líneas que comienzan con el caracter "!" son *comentarios* y no tienen otro propósito más que hacer los programas más legibles. Se puede utilizar también el caracter "c", puesto que este es el que se utiliza en la versión 77, sin embargo dicha versión no admite el símbolo de asignación ":", por lo que se omite.

Originalmente todos los programas de Fortran tenían que ser escritos solamente con letras mayúsculas, actualmente se pueden escribir con minúsculas con lo cual se mejora la legibilidad, por lo que se hará de esa forma.

4. Organización del Programa

Un programa de Fortran por lo general consiste de un programa principal o *main* (o manejador) y posiblemente varios subprogramas (o procedimientos o subrutinas). Por el momento se considerara que todas las sentencias están en el programa principal; los subprogramas se revisarán más adelante. La estructura del programa principal es:

```

! F90/95
program NOMBRE
    Inclusión de librerías y módulos externas
    Declaraciones de variables y tipos
    Instrucciones

stop
end

```

En este trabajo las palabras que estén en *itálicas* no deberán ser tomadas en forma literal, sino como una descripción general. La sentencia `stop` es opcional y podría ser vista como redundante ya que el programa terminará cuando alcance el fin, pero se recomienda que el programa termine con la sentencia `stop` para resaltar que la ejecución del programa ahí termina.

F77 *no* es un lenguaje de formato libre, ya que tiene un conjunto de reglas estrictas de cómo el código fuente debe ser ordenado en el editor. Las reglas más importantes son las reglas para la posición en columnas:

```

Col. 1      : Blanco (espacio), o un caracter "c" o "*" para comentarios
Col. 2-5    : Etiqueta de sentencia (opcional)
Col. 6      : Continuación de una línea previa (opcional)
Col. 7-72   : Sentencias
Col. 73-80 : Número de secuencia (opcional, raramente usado actualmente)

```

Muchas líneas de un programa de F77 inician con 6 espacios y terminan antes de la columna 72, solamente el campo de sentencia es usado. Para F90 se permite el libre formato.

Continuación

Ocasionalmente una sentencia no cabe en una sola línea. Se puede dividir la sentencia en dos o más líneas, y usar la marca de continuación en la posición 6. Ejemplo:

```

c 2 3 4 5 6 (;Esto muestra la posición de la columna!)
c          La siguiente sentencia esta en dos líneas físicas
          area = 3.14159265358979
          +      * r * r

```

Cualquier caracter puede ser usado en vez del signo "+" como caracter de continuación. Se considera un buen estilo de programación usar el signo más, ampersand (&) o números (2 para la segunda línea, 3 para la tercera y así sucesivamente).

En F90, para continuar una línea, el caracter de continuación es el ampersand "&", el cual se debe colocar al final de la línea que se pretende continuar en el renglón de abajo. El siguiente ejemplo muestra a la sentencia anterior, pero escrita en formato F90

```

!      La siguiente sentencia esta en dos líneas físicas
      area = 3.14159265358979 &
          * r * r

```

Espacios en Blanco

Los espacios en blanco son ignorados en F77. Por lo tanto si se remueven todos los espacios en blanco en un programa de F77, el programa sintácticamente es correcto, pero no es legible para los humanos.

5. Declaración y tipos de Variables

Nombre de Variables

Los nombres de variables en Fortran consisten de 1 a 6 caracteres escogidos de la a a la z y de los dígitos del 0 al 9. El primer caracter debe ser una letra. (Nota: en F90 se permiten nombres de longitud arbitraria). F77 no diferencia entre mayúsculas y minúsculas, de hecho, asume que toda la entrada es con minúsculas. Sin embargo hace poco los compiladores de F77 empezaron a aceptar letras minúsculas. Si por alguna razón se llegará a dar la situación que el compilador de F77 insiste en usar letras mayúsculas, se puede usar el comando `tr` de Unix para hacer la conversión.

Tipos y declaraciones

Cada variable debería ser definida con una declaración. Esto indica el tipo de la variable. Las declaraciones más comunes son:

<i>c F 77</i>		<i>! F90/95/2000</i>	
Integer	lista de variables	Integer	:: lista de variables
Real	lista de variables	Real	:: lista de variables
Double precision	lista de variables	Double precision	:: lista de variables
Complex	lista de variables	Complex	:: lista de variables
Logical	lista de variables	Logical	:: lista de variables
Character	lista de variables	Character	:: lista de variables

La lista de variables consiste de nombres de variables separadas por comas. Cada variable deberá ser declarada exactamente una vez. Si una variable no esta declarada, tanto F77 como 95/2000 usan un conjunto implícito de reglas para establecer el tipo. Con lo anterior todas las variables que comiencen con el conjunto de letras i-n son enteros y el resto tipo real. Varios programas viejos de Fortran usan estas reglas implícitas, pero no se recomienda su uso. La probabilidad de errores en el programa crece exponencialmente si no se declaran las variables explícitamente.

Variables Enteras y de punto flotante

F77 sólo tiene un tipo para variables enteras. Los enteros son usualmente guardados en 32 bits (4 bytes). Por lo que el rango de valores que pueden tomar los enteros es de (-231,231-1).

F77 tiene dos tipos diferentes para punto flotantes conocidos como real y doble precisión. Mientras el tipo real es por lo general adecuado, algunos cálculos numéricos requieren de una mayor precisión por lo que `double precision` deberá ser usado. El tamaño por lo general es para el tipo real de 4 bytes y el de doble precisión es de 8 bytes, pero lo anterior depende de la máquina y el compilador. Algunas versiones no estandarizadas de Fortran usan la sintaxis `real*8` para indicar una variable de punto flotante de 8 bytes.

La sentencia `parameter`

Algunas constantes aparecen varias veces en un programa, por lo que es deseable que se definan una sola vez al principio del `program`. Esto se puede hacer con la sentencia `parameter`, a la vez que hace los programas más legibles. Por ejemplo el programa visto anteriormente podría haberse escrito de la siguiente forma:

```

program circulo
real r, area, pi
parameter (pi=3.14159)

```

*c Este programa lee un número real r y muestra
c el área del círculo con radio r.*

```

write (*,*) 'Escribe el radio r:'
read (*,*) r
area = pi*r*r
write (*,*) 'Area = ', area
stop
end

```

O

```

program circulo
real :: r, area
real, parameter :: pi = 3.14159

```

*! Este programa lee un número real r y muestra
! el área del círculo con radio r.*

```

write (*,*) 'Escribe el radio r:'
read (*,*) r
area = pi*r*r
write (*,*) 'Area = ', area
stop
end

```

La sintaxis de la sentencia **parameter** en F77 es

```
parameter (nombre = constante, ... , nombre = constante)
```

para F90 existe la siguiente sintaxis:

```
tipo, parameter :: nombre = constante, ... , nombre = constante
```

Las reglas para la sentencia **parameter** son:

- La "constante" definida en una sentencia **parameter** no es una variable, es una constante por lo que su valor nunca cambia.
- Una "constante" puede aparecer a lo más una vez en la sentencia **parameter**.
- La(s) sentencia(s) **parameter** deberán estar antes que cualquier sentencia de ejecución.

Algunas de las razones para usar **parameter** son:

- ayuda a recordar más fácilmente el uso de constantes.
- es fácil cambiar una constante si aparece muchas veces en el programa.

Asignación y Expresiones

Constantes

La forma más simple de una expresión es una constante. Hay seis tipos de constantes, que corresponden con los tipos de datos que maneja Fortran. Se muestran a continuación ejemplos de constantes enteras:

```
1
0
-100
32767
+15
```

Ejemplos de constantes de tipo real:

```
1.0
-0.25
2.0E6
3.333E-1
```

La notación E se usa para números que se encuentran escritos en notación científica. Por lo que 2.0E6 es dos millones, mientras que 3.333E-1 es aproximadamente la tercera parte de uno. Para constantes que rebasen la capacidad de almacenamiento de un real se puede usar doble precisión. En este caso, la notación usada es el símbolo "D" en vez de "E". Por ejemplo:

```
2.0D-1
1D99
```

Por lo que 2.0D-1 es la quinta parte de uno almacenada en un tipo doble precisión y 1D99 es un uno seguido por 99 ceros.

El siguiente tipo son constantes complejas. Los cuales son indicados por un par de constantes (enteras o reales), separadas por una coma y encerrados entre paréntesis. Por ejemplo:

```
(2, -3)
(1., 9.9E-1)
```

El primer número denota la parte real y el segundo la parte imaginaria.

El quinto tipo, son las constantes lógicas. Estas sólo pueden tener uno de dos posibles valores, que son:

```
.TRUE.
.FALSE.
```

Obsérvese que se requiere que se encierren con punto las palabras.

El último tipo son las constantes de caracteres. Estas son por lo general usadas como un arreglo de caracteres, llamado cadena. Estas consisten de una secuencia arbitraria de caracteres encerradas con apóstrofes (comillas sencillas), por ejemplo:

```
'ABC'
';Cualquier cosa!'
'Es un magnífico día'
```

Las constantes de cadenas son sensibles a mayúsculas y minúsculas. Se presenta un problema cuando se quiere poner un apóstrofe dentro de la cadena. En este caso la cadena debe ser encerrada entre comillas dobles, por ejemplo:

"Hay muchos CD's piratas"

6. Expresiones

Las expresiones más sencillas son de la forma

operando operador operando

y un ejemplo es

$x + y$

El resultado de una expresión es por si misma otro operando, por lo que se puede hacer anidamiento como lo siguiente:

$x + 2 * y$

Con lo anterior se presenta la pregunta de la precedencia, ¿la última expresión significa $x + (2*y)$ o $(x+2)*y$? La precedencia de los operadores aritméticos en Fortran es (de la más alta a la más baja):

```
**   {exponenciación}
*   /   {multiplicación, división}
+   -   {suma, resta}
```

Todos los operadores son evaluados de izquierda a derecha, excepto el operador de exponenciación, el cual tiene precedencia de derecha a izquierda. Si se desea cambiar el orden de evaluación, se pueden usar paréntesis.

Todos los operadores anteriores son binarios. Se tienen también operadores unarios, uno de ellos es el de negación - y que tiene una precedencia mayor que los anteriores. Por lo que la expresión $-x+y$ significa lo que se esperaría.

Se debe tener cuidado al usar el operador de división, el cual devolverá distintos valores dependiendo del tipo de datos que se estén usando. Si ambos operandos son enteros, se hace una división entera, de otra forma se hace una división que devuelve un tipo real. Por ejemplo, $3/2$ es igual a 1 y $3./2.$ es igual a 1.5.

Asignación

Una asignación tiene la forma

nombre_de_variable = expresión

La interpretación es como sigue: se evalúa el lado derecho y se asigna el valor del resultado a la variable de la izquierda. La expresión de la derecha puede contener otras variables, pero estas nunca cambiarán de valor. Por ejemplo:

$area = pi * r**2$

no cambia el valor de pi , ni de r , solamente de $area$.

Conversión de Tipos

Cuando diferentes tipos de datos ocurren en una expresión se lleva a cabo una conversión de tipos ya sea explícita o implícitamente. Fortran realiza algo de conversión implícita. Por ejemplo

```
real x
x = x + 1
```

el uno será convertido al tipo real, y se incrementa la variable x en uno. Sin embargo, en expresiones más complicadas, es una buena práctica de programación forzar explícitamente la conversión de tipos. Para números se tienen las siguientes funciones disponibles:

```
int
real
dble
ichar
char
```

Las primeras tres no requieren explicación. Con `ichar` se toma un carácter y se convierte a un entero y con `char` se hace lo contrario.

Ejemplo: ¿Cómo multiplicar dos variables tipo real x e y usando doble precisión y guardando el resultado en una variable de doble precisión w?

```
w = dble(x)*dble(y)
```

Obsérvese que es diferente de:

```
w = dble(x*y)
```

7. Expresiones Lógicas

Una expresión lógica puede tener solamente el valor de `.TRUE.` o de `.FALSE.`. Una valor lógico puede ser obtenido al comparar expresiones aritméticas usando los siguientes operadores relacionales:

F77	F90
<code>.LT.</code>	<code><</code>
<code>.LE.</code>	<code><=</code>
<code>.GT.</code>	<code>></code>
<code>.GE.</code>	<code>>=</code>
<code>.EQ.</code>	<code>=</code>
<code>.NE.</code>	<code>/=</code>

Por lo que no se pueden usar símbolos como `<` o `=` para comparación en F77 estandar, por lo que se tienen que usar abreviaturas de dos letras encerradas con puntos. Sin embargo en F90 (y en general en todos los compiladores comerciales de Fortran) ya pueden ser usados.

Las expresiones lógicas pueden ser combinadas con los operadores lógicos `.AND.` `.OR.` `.NOT.` que corresponden a los operadores lógicos conocidos Y, O y negación respectivamente.

Asignación de Variables Lógicas

Los valores booleanos pueden ser guardados en variables lógicas. La asignación es de forma análoga a la asignación aritmética. Ejemplo:

```
logical:: a, b
a = .TRUE.
b = a .AND. 3 .LT. 5/2
```

El orden de precedencia es importante, como se muestra en el último ejemplo. La regla es que las expresiones aritméticas son evaluadas primero, después las que contienen operadores relacionales, y finalmente las de operadores lógicos. Por lo que a b se le asigna `.FALSE.` en el ejemplo anterior.

Las expresiones lógicas son usadas frecuentemente en sentencias condicionales como `if`.

La sentencia `if`

Una parte importante de cualquier lenguaje de programación son las sentencias condicionales. La sentencia más común en Fortran es `if`, la cual tiene varias formas de uso. La forma más simple de la sentencia `if` es:

```
if (expresión lógica) sentencia
```

Lo anterior tiene que ser escrito en una sola línea. El siguiente ejemplo obtiene el valor absoluto de `x`:

```
if (x .LT. 0) x = -x
```

Si más de una sentencia necesita ser ejecutada dentro de la sentencia `if`, entonces la siguiente sintaxis deberá ser usada:

```
if (expresión lógica) then
    sentencias
endif
```

La forma más general más general de la sentencia `if` tiene la siguiente forma:

```
if (expresión lógica) then
    sentencias
elseif (expresión lógica) then
    sentencias
:
:
else
    sentencias
endif
```

El flujo de ejecución es de arriba hacia abajo. Las expresiones condicionales son evaluadas en secuencia hasta que se encuentra una que es verdadera. Entonces el código asociado es ejecutado y el control salta a la siguiente sentencia después de la sentencia `endif`.

Sentencias `if` anidadas

La sentencia `if` puede ser anidada varios niveles. Para asegurar la legibilidad es importante sangrar las sentencias. Se muestra un ejemplo:

```
if (x > 0) then
    if (x >= y) then
        write(*,*) 'x es positivo y x >= y'
    else
        write(*,*) 'x es positivo pero, x < y'
    endif
elseif (x < 0) then
    write(*,*) 'x es negativo'
else
    write(*,*) 'x es cero'
endif
```

Se debe evitar anidar muchos niveles de sentencias `if` ya que es difícil de seguir.

8 Ciclos

Para la repetir la ejecución de sentencias se usan los ciclos. Si se esta familiarizado con otros lenguajes de programación se habrá escuchado de los ciclos-for y de los ciclos-until, F77 tiene solamente una construcción de ciclo, conocida como el ciclo-do. El ciclo-do corresponde al ciclo-for que existe en otros lenguajes de programación. Otros ciclos pueden ser simulados usando las sentencias if y goto.

Ciclos-do

El ciclo-do es usado para repetir un conjunto de sentencias una determinada cantidad de veces. Se muestra el siguiente ejemplo donde se calcula la suma de los enteros desde el 1 hasta n (suponiendo que a n se le ha asignado un valor previamente):

```
integer i, n, suma
:
:
:
suma = 0
do 10 i = 1, n
    suma = suma + i
    write(*,*) 'i =', i
    write(*,*) 'suma =', suma
10 continue
```

El número 10 es una sentencia de etiqueta. Típicamente, podría haber varios ciclos y otras sentencias en un programa que requieran una sentencia de etiqueta. El programador es responsable de asignar un número único a cada etiqueta en cada programa (o subprograma). Recordar que las posiciones de las columnas 2-5 son reservadas para sentencias de etiquetas. El valor numérico de las sentencias de etiqueta no tienen ningún significado, por lo que cualquier valor entero puede ser usado. Por lo general, los programadores incrementan las etiquetas de 10 en 10 cada vez.

La variable en la sentencia do es incrementada en 1 por default. Sin embargo, se puede usar cualquier otro entero para el paso o incremento. El siguiente segmento de programa muestra los números pares en forma decreciente entre el 1 y 10:

```
integer :: i
:
:
:
do 20 i = 10, 1, -2
    write(*,*) 'i =', i
20 continue
```

La forma general del ciclo do es la siguiente en F77:

```
do etiq var = expr1, expr2, expr3
    sentencias
etiq continue
```

o de forma alterna en F90:

```
do (expr lógica)
    sentencias
end do
```

donde:

var es la variable del ciclo (conocida con frecuencia como el índice del ciclo) el cual deberá ser del tipo **integer**.
expr1 indica el valor inicial de **var**,
expr2 es el valor hasta el que llegará el índice, y
expr3 es el incremento (**step**).
etiq número de línea

Nota: La variable del ciclo **do** nunca deberá ser modificada por otras sentencias dentro del ciclo, ya que puede generar errores de lógica.

Muchos compiladores de F77 permiten que los ciclos **do** sean cerrados por la sentencia **end do**. La ventaja es que la sentencia etiqueta puede ser omitida, ya que en este caso la sentencia **end do** cierra la sentencia **do** más cercana. La construcción **end do** es ampliamente usada, pero no es parte del ANSI F77.

Ciclos **while**

La forma más intuitiva para escribir un ciclo **while** es

```
while (expr lógica) do  
    sentencias  
end do
```

o de forma alterna

```
do while (expr lógica)  
    sentencias  
end do
```

Las sentencias en el cuerpo serán repetidas mientras la condición en el ciclo **while** sea verdadera. A pesar de que esta sintaxis es aceptada por muchos compiladores (incluyendo el de Linux), no forma parte del ANSI F77.

A continuación se tiene un ejemplo que calcula y muestra el doble de todos los número anterior comenzando con el 2 y que son menores a 100:

```
integer n  
n = 1  
10 if (n .lt. 100) then  
    n = 2*n  
    write (*,*) n  
    goto 10  
endif
```

Ciclos-**until**

Es un ciclo el cual el criterio de terminación esta al final en vez del inicio. En pseudocódigo tendríamos el siguiente formato:

```
haz  
    sentencias  
hasta (expr lógica)
```

lo cual nuevamente, puede ser implementado en F77 usando las sentencias `if` y `goto`:

```
      etiq      continue
              sentencias
              if (expr lógica) goto etiq
```

Obsérvese que la expresión lógica en la última versión deberá ser la negación de la expresión dada en pseudocódigo.

Ciclos en F90

F90 ha adoptado la construcción `do-end do` como su ciclo (el f77 de linux la reconoce como válida). Por lo que el ejemplo de decrementar de dos en dos queda como:

```
      do i = 10, 1, -2
          write(*,*) 'i =', i
      end do
```

para simular los ciclos `while` y `until` se puede usar la construcción `do-end do`, pero se tiene que agregar una sentencia condicional de salida `exit` (salida). El caso general es:

```
      do
          sentencias
          if (expr lógica) exit
          sentencias
      end do
```

Si se tienen la condición de salida al principio es un ciclo `while`, y si esta al final se tiene un ciclo `until`.

9 Arreglos

Muchos cálculos científicos usan vectores y matrices. El tipo de dato usado en Fortran para representar tales objetos es el array. Un arreglo unidimensional corresponde a un vector, mientras que un arreglo bidimensional corresponde a una matriz. Para entender como son usados en F77, no solamente se requiere conocer la sintaxis para su uso, sino también como son guardados estos objetos en la memoria.

Arreglos Unidimensionales

El arreglo más sencillo es el de una dimensión, el cual es sólo un conjunto de elementos almacenados secuencialmente en memoria. Por ejemplo, la declaración

```
      real d(20)
```

o de manera alternativa:

```
      real, dimension (20) :: d
```

declara a `d` como un arreglo del tipo `real` con 20 elementos. Esto es, `d` consiste de 20 números del tipo `real` almacenados en forma contigua en memoria. Las dos instrucciones anteriores son validas para compiladores F77 como F90 modernos. La segunda instrucción sin embargo, permite definir un conjunto de arreglos de una misma dimensión e índice de manera más cómoda y hasta cierto punto mas legible.

Como ejemplo, tómesese la siguiente instrucción:

```
real, dimension (20) :: d, Lista, NumerosReales, A, B, C, H
```

y compárese con la siguiente:

```
real d(20), Lista(20), NumerosReales(20), A(20), B(20), C(20), H(20)
```

Por convención, los arreglos en Fortran están indexados a partir del valor 1. Por lo tanto el primer elemento en el arreglo es $d(1)$ y el último es $d(20)$. Sin embargo, se puede definir un rango de índice arbitrario para los arreglos como se observa en los siguientes ejemplos:

```
real b(0:19), c(-162:237)
```

En el caso de b es similar con el arreglo d del ejemplo previo, excepto que el índice corre desde el 0 hasta el 19. El arreglo c es un arreglo de longitud $237 - (-162) + 1 = 400$. El tipo de los elementos de un arreglo puede ser cualquiera de los tipos básicos de datos ya vistos. Ejemplos:

```
integer :: i(10)  
logical :: aa(0:1)  
double precision :: x(100)
```

Cada elemento de un arreglo puede ser visto como una variable separada. Se referencia al i -ésimo elemento de un arreglo a por $a(i)$. A continuación se muestra un segmento de código que guarda los primeros 10 cuadrados en un arreglo `cuadrado`.

```
integer i, cuadrado(10)  
do i=1, 10, 1  
    cuadrado(i) = i**2;  
    write(*,*) cuadrado(i)  
end do
```

Un error común en Fortran es hacer que el programa intente acceder elementos del arreglo que están fuera de los límites. Lo anterior es responsabilidad del programador, ya que tales errores no son detectados por el compilador.

Arreglos Bidimensionales

Las matrices son muy importantes en álgebra lineal. Las matrices son usualmente representadas por arreglos bidimensionales. Por ejemplo, la declaración

```
real Arreglo(3,5)
```

O

```
real, dimension (3, 5) :: Arreglo
```

define un arreglo bidimensional de $3 \times 5 = 15$ números del tipo real. Es útil pensar que el primer índice es el índice del renglón, y el segundo índice corresponde a la columna. Por lo tanto se vería como:

	1	2	3	4	5
1					
2					
3					

Un arreglo bidimensional podría también tener índices de rango arbitrario. La sintaxis general para declarar el arreglo es:

```
nombre (índice1_inf : índice1_sup, índice2_inf : índice2_sup)
```

El tamaño total del arreglo es de

```
tamaño = (índice1_sup - índice1_inf + 1) x (índice2_sup - índice2_inf + 1)
```

Es muy común en Fortran declarar arreglos que son más grandes que la matriz que se va a guardar. Lo anterior es porque Fortran no tiene almacenamiento dinámico de memoria como el lenguaje C. Por ejemplo:

```
real A(3,5)
integer i,j
! Solamente se usará
! una submatriz de 3 x 3 del arreglo
do i=1, 3
    do j=1, 3
        a(i,j) = real(i)/real(j)
    end do
end do
```

Los elementos en la submatriz A(1:3,4:5) no están definidas. No se debe considerar que estos elementos están inicializados a cero por el compilador (algunos compiladores lo hacen, pero otros no).

Forma de Almacenamiento para un arreglo bidimensional . Fortran almacena los arreglos de más de una dimensión como una secuencia contigua lineal de elementos. Es importante saber que los arreglos bidimensionales son guardados por columnas. Por lo tanto en el ejemplo anterior, el elemento del arreglo (1,2) esta después del elemento (3,1), luego sigue el resto de la segunda columna, la tercera columna y así sucesivamente.

Considerando otra vez el ejemplo donde solamente se usa la submatriz de 3 x 3 del arreglo de 3 x 5. Los primeros 9 elementos que interesan se encuentran en las primeras nueve localidades de memoria, mientras que las últimas seis celdas no son usadas. Lo anterior funciona en forma transparente porque la dimensión principal es la misma para ambos, el arreglo y la matriz que se guarda en el arreglo. Sin embargo, frecuentemente la dimensión principal del arreglo será más grande que la primera dimensión de la matriz. Entonces la matriz no será guardada en forma contigua en memoria, aún si la arreglo es contiguo. Por ejemplo, supongamos que la declaración hubiera sido A(5,3) entonces hubiera habido dos celdas "sin usar" entre el fin de la primera columna y el principio de la siguiente columna (suponiendo que asumimos que la submatriz es de 3 x 3).

Esto podría parecer complicado, pero actualmente es muy simple cuando se empieza a usar. Si se tiene en duda, puede ser útil hallar la dirección de un elemento del arreglo. Cada arreglo tendrá una dirección en la memoria asignada a partir del arreglo, que es el elemento (1,1). La dirección del elemento (i,j) esta dada por la siguiente expresión:

$$\text{dirección}[A(i,j)] = \text{dirección}[A(1,1)] + (j-1)*\text{princ} + (i-1)$$

donde `princ` es la dimensión principal (la columna) de A. Obsérvese que `princ` es en general diferente de la dimensión actual de la matriz. Muchos errores de lógica en Fortran son causados por lo anterior, por lo tanto es importante entender la diferencia.

Arreglos Multi-dimensionales

F77 permite arreglos de hasta 7 dimensiones. La sintaxis y forma de almacenamiento es análoga al caso de dos dimensiones.

La sentencia `dimension`

Hay una forma alterna para declarar un arreglo en Fortran. Las siguientes sentencias

```
real A, x
dimension      x(50)
dimension      A(10,20)
```

son equivalentes a

```
real A(10,20), x(50)
```

La sentencia `dimension` es considerada actualmente como una forma en desuso.

10 Subprogramas

Cuando un programa tiene más de cien líneas, es difícil de seguir. Los códigos de Fortran que resuelven problemas reales de ingeniería por lo general tienen decenas de miles de líneas. La única forma para manejar códigos tan grandes, es usar una aproximación modular y dividir el programa en muchas unidades independientes pequeñas llamadas subprogramas.

Un subprograma es una pequeña pieza de código que resuelve un subproblema bien definido. En un programa grande, se tiene con frecuencia que resolver el mismo subproblema con diferentes tipos de datos. En vez de replicar el código, estas tareas pueden resolverse con subprogramas. El mismo subprograma puede ser llamado varias veces con distintas entradas de datos.

En Fortran se tienen dos tipos diferentes de subprogramas, conocidas como funciones y subrutinas.

Funciones

Las funciones en Fortran son bastante similares a las funciones matemáticas: ambas toman un conjunto de variables de entrada (parámetros) y regresan un valor de algún tipo. Al inicio de la sección se comentó de los subprogramas definidos por el usuario, pero F77 tiene también funciones incorporadas.

Un ejemplo simple muestra como usar una función:

```
x = cos(pi/3.0)
```

En este caso la función coseno `cos` de 60° , asignará a la variable `x` el valor de 0.5 (si `pi` ha sido definido correctamente; F77 no tiene constantes incorporadas). Hay varias funciones incorporadas en F77. Algunas de las más comunes son:

<code>abs</code>	valor absoluto
<code>min</code>	valor mínimo
<code>max</code>	valor máximo
<code>sqrt</code>	raíz cuadrada
<code>sin</code>	seno
<code>cos</code>	coseno
<code>tan</code>	tangente
<code>atan</code>	arco tangente
<code>exp</code>	exponencial (natural)
<code>log</code>	logaritmo (natural)

En general, una función siempre tiene un tipo. Varias de las funciones incorporadas mencionadas anteriormente son sin embargo genéricas. Por lo tanto en el ejemplo anterior `pi` y `x` podrían ser del tipo `real` o del tipo `double precision`. El compilador revisará los tipos y usará la versión correcta de la

función `cos` (`real` o `double precision`). Desafortunadamente, Fortran no es un lenguaje polimórfico, por lo que en general, el programador debe hacer coincidir los tipos de las variables y las funciones.

Se revisa a continuación como implementar las funciones escritas por el usuario. Supongamos el siguiente problema: un meteorólogo ha estudiado los niveles de precipitación en el área de una bahía y ha obtenido un modelo (función) $ll(m,t)$ donde ll es la cantidad de lluvia, m es el mes, y t es un parámetro escalar que depende de la localidad. Dada la fórmula para ll y el valor de t , calcular la precipitación anual

La forma obvia de resolver el problema es escribir un ciclo que corra sobre todos los meses y sume los valores de ll . Como el cálculo del valor de ll es un subproblema independiente, es conveniente implementarlo como una función. El siguiente programa principal puede ser usado:

```
program lluvia
real :: r, t, suma
integer :: m

    write(*,*) 'Ingrese el tiempo (t) '
    read (*,*) t
    suma = 0.0
    do m = 1, 12
        suma = suma + ll(m, t)
    end do
    write (*,*) 'La precipitación Anual es ', suma, 'pulgadas'
stop
end
```

Además, la función ll tiene que ser definida como una función de Fortran. La fórmula del meteorólogo es:

$$ll(m,t) = t/10 * (m**2 + 14*m + 46) \text{ si la expresión es positiva}$$
$$ll(m,t) = 0 \text{ otro caso}$$

La correspondiente función en Fortran es

```
real function ll(m,t)
Integer :: m
Real :: t

    ll = 0.1*t * (m**2 + 14*m + 46)
    if (ll .LT. 0) ll = 0.0
    return
end
```

Se puede observar que la estructura de una función es parecida a la del programa principal. Las diferencias son:

Las funciones tienen un tipo. El tipo debe coincidir con el tipo de la variable que recibirá el valor. El valor que devolverá la función, deberá ser asignado en una variable que tenga el mismo nombre que la función.

Las funciones son terminadas con la sentencia `return` en vez de la sentencia `stop`. Para resumir, la sintaxis general de una función en F77 es:

```
tipo function nombre (lista_de parámetros)
declaraciones
:
sentencias
:
return
end
```

La función es llamada simplemente usando el nombre de la función y haciendo una lista de argumentos entre paréntesis.

Subrutinas

Una función de Fortran puede devolver únicamente un valor. En ocasiones se desean regresar dos o más valores y en ocasiones ninguno. Para este propósito se usa la construcción subrutina. La sintaxis es la siguiente:

```
subroutine nombre (lista_de parámetros)
declaraciones
:
sentencias
:
return
end
```

Obsérvese que las subrutinas no tienen tipo y por consecuencia no pueden hacerse asignación al momento de llamar al procedimiento. Se da un ejemplo de una subrutina muy sencilla. El propósito de la subrutina es intercambiar dos valores enteros.

```
subroutine iswap (a, b)
integer :: a, b
! Variables locales
integer tmp
! Sentencias
tmp = a
a = b
b = tmp
return
end
```

Se debe observar que hay dos bloques de declaración de variables en el código. Primero, se declaran los parámetros de entrada/salida, es decir, las variables que son comunes al que llama y al que recibe la llamada. Después, se declaran las variables locales, esto es, las variables que serán sólo conocidas dentro del subprograma. Se pueden usar los mismos nombres de variables en diferentes subprogramas.

Llamada por referencia

F77 usa el paradigma de llamada por referencia. Esto significa que en vez de pasar los valores de los argumentos a la función o la subrutina (llamada por valor), se pasa la dirección (apuntadores) de los argumentos.

```

        program llamaint
        integer m, n
c
        m = 1
        n = 2

        call iswap(m, n)
        write(*,*) m, n

        stop
        end

```

La salida de este programa es "2 1", justo como se habría esperado. Sin embargo, si F77 hubiera hecho una llamada por valor entonces la salida hubiera sido "1 2", es decir, las variables m y n hubieran permanecido sin cambio. La razón de esto último, es que solamente los valores de m y n habrían sido copiados a la subrutina `iswap`, a pesar de que a y b hubieran sido cambiados dentro de la subrutina, por lo que los nuevos valores no sería regresados al programa que hizo la llamada.

En el ejemplo anterior, la llamada por referencia era lo que se quería hacer. Se debe tener cuidado al escribir código en Fortran, porque es fácil introducir efectos laterales no deseados. Por ejemplo, en ocasiones es tentador usar un parámetro de entrada en un subprograma como una variable local y cambiar su valor. No se deberá hacer nunca, ya que el nuevo valor se propagará con un valor no esperado.

Se revisará más conceptos cuando se vea la sección de Arreglos en subprogramas para pasar arreglos como argumentos.

11. E/S Básica

Una parte importante del cualquier programa de cómputo es manejar la entrada y la salida. En los ejemplos revisados previamente, se han usado las dos construcciones más comunes de Fortran que son: `read` and `write`. La E/S con Fortran puede ser un poco complicada, por lo que nos limitaremos a los casos más sencillos en el tutorial.

Lectura y Escritura

La sentencia `read` es usada para la entrada y la sentencia `write` para la salida. El formato es:

```

read (núm_unidad, núm_formato) lista_de_variables
write(núm_unidad, núm_formato) lista_de_variables

```

El número de unidad se puede referir a la salida estándar, entrada estándar o a un archivo. Se describirá más adelante. El número de formato se refiere a una etiqueta para la sentencia `format`, la cual será descrita brevemente. Es posible simplificar estas sentencias usando asteriscos (*) para algunos argumentos, como lo que se ha hecho en los ejemplos anteriores. A lo anterior se le conoce como una lectura/escritura de lista dirigida.

```

read (*,*) lista_de_variables
write(*,*) lista_de_variables

```

La primera sentencia leerá valores de la entrada estándar y asignará los valores a las variables que aparecen en la lista, y la segunda escribe la lista en la salida estándar.

Ejemplos

Se muestra un segmento de código de un programa de Fortran:

```
integer :: m, n
real    :: x, y

read(*,*) m, n
read(*,*) x, y
```

Se ingresan los datos a través de la entrada estándar (teclado), o bien, redireccionando la entrada a partir de un archivo. Un archivo de datos consiste de registros de acuerdo a los formatos válidos de Fortran. En el ejemplo, cada registro contiene un número (entero o real). Los registros están separados por espacios en blanco o comas. Por lo que una entrada válida al programa anterior puede ser:

```
-1 100
-1.0 1e+2
```

O se pueden agregar comas como separadores:

```
-1, 100
-1.0, 1e+2
```

Obsérvese que la entrada en Fortran es sensible a la línea, por lo que es importante contar con el número apropiado de elementos de entrada (registros) en cada línea. Por ejemplo, si se da la siguiente entrada en una sola línea

```
-1, 100, -1.0, 1e+2
```

entonces a *m* y a *n* se asignarán los valores de -1 y 100 respectivamente, pero los dos últimos valores serán descartados, dejando a *x* e *y* sin definir.

12. Sentencia `Format`

En las secciones anteriores se ha mostrado el formato libre de entrada/salida. Éste caso sólo usa una reglas predefinidas acerca de como los diferentes tipos (integers, reals, characters, etc.) serán mostrados. Por lo general un programador desea indicar algún formato de entrada o salida, por ejemplo, el número de decimales que tendrá un número real. Para este propósito Fortran tiene la sentencia `format`. La misma sentencia `format` puede ser usada para la entrada o salida.

Sintaxis

```
write(*, etiqueta) lista_de_variables
etiqueta format códigos_de_formato
```

Un ejemplo simple muestra como trabaja. Supongamos que se tiene una variable entera que se quiere mostrar con un ancho de 4 caracteres y un número real que se quiere mostrar en notación de punto fijo con 3 decimales.

```
900 write(*, 900) lista_de_variables
format (I4, F8.3)
```

La etiqueta 900 de la sentencia `format` es escogida en forma arbitraria, pero es una práctica común numerar las sentencias `format` con números más grandes que las etiquetas de control de flujo. Después de la palabra `format` se ponen los códigos de formato encerrados entre paréntesis. El código I4 indica

que un entero tendrá un ancho de 4 y F8.3 significa que el número deberá mostrarse en notación de punto fijo con un ancho de 8 y 3 decimales.

La sentencia `format` puede estar en cualquier lugar dentro del programa. Hay dos estilos de programación: agrupar por parejas las sentencias (como en el ejemplo), o poner el grupo de sentencias `format` al final del (sub)programa.

Códigos comunes de formato

Las letras para códigos de formato más comunes son:

- A - cadena de texto
- D - números de doble precisión, notación científica
- E - números reales, notación científica
- F - números reales, formato de punto fijo
- I - entero
- X - salto horizontal (espacio)
- / - salto vertical (nueva línea)

El código de formato F (y similarmente D y E) tiene la forma general Fa.d donde a es una constante entera indicando el ancho del campo y d es un entero constante que indica el número de dígitos significativos.

Para los enteros solamente el campo de ancho es indicado, por lo que la sintaxis es la. En forma parecida las cadenas de caracteres pueden ser especificadas como A a pero el campo de ancho por lo general no es usado.

Si un número o cadena no llena todo el ancho del campo, espacios son agregados. Usualmente el texto será ajustado a la derecha, pero las reglas exactas varían de acuerdo a los códigos de formato.

Para un espaciado horizontal, el código nX es usado. Donde n indica el número de espacios horizontales. Si n es omitido se asume n=1. Para espaciado vertical (nuevas líneas) se usa el código /. Cada diagonal corresponde a una nueva línea. Obsérvese que cada sentencia read o write por defecto termina con un salto de línea (a diferencia de C).

Algunos Ejemplos

El siguiente código de Fortran

```
x = 0.025
      write(*,100) 'x=', x
100   format (A,F)
      write(*,110) 'x=', x
110   format (A,F5.3)
      write(*,120) 'x=', x
120   format (A,E)
      write(*,130) 'x=', x
130   format (A,E8.1)
```

genera la siguiente salida una vez que es ejecutado:

```
x=          0.0250000
x=0.025
x=  0.2500000E-01
x=  0.3E-01
```

Obsérvese que espacios en blanco son automáticamente puestos del lado izquierdo y que el ancho del campo por default para números tipo real es de usualmente de 14. Se puede ver también que F77 sigue la regla de redondeo donde los dígitos del 0-4 son redondeados hacia abajo y los dígitos del 5-9 son redondeados hacia arriba.

En este ejemplo cada sentencia `write` usa una sentencia `format` diferente. Pero es correcto usar la misma sentencia `format` varias veces con distintas sentencias `write`. De hecho, esta es una de las principales ventajas de usar sentencias `format`. Esta característica es buena cuando se muestra el contenido de una tabla por ejemplo, y se desea que cada renglón tenga el mismo formato. `format`.

Cadenas de formato en las sentencias `read/write`

En vez de indicar el código de formato en una sentencia `format` por separado, se puede dar el código de formato en la sentencia `read/write` directamente. Por ejemplo, la sentencia

```
write (*, '(A, F8.3)') 'La respuesta es x = ', x
```

que es equivalente a

```
          write (*, 990) 'La respuesta es x = ', x
990 format (A, F8.3)
```

Algunas veces cadenas de texto son dadas en las sentencias de formato, por ejemplo la siguiente versión es también equivalente:

```
          write (*, 999) x
999 format ('La respuesta es x = ', F8.3)
```

Ciclos Implícitos y Repetición de Formatos

Ahora se mostrará un ejemplo más complejo. Supongamos que se tiene un arreglo bidimensional de enteros y que se desea mostrar la submatriz izquierda 5 por 10, con 10 valores cada 5 renglones.

```
          do 10 i = 1, 5
              write(* ,1000) (a(i,j), j=1,10)
10 continue
1000 format (I6)
```

Se tiene un ciclo explícito `do` sobre los renglones y un ciclo implícito sobre el índice `j` para la columna.

Con frecuencia una sentencia `format` involucra repetición, por ejemplo:

```
950 format (2X, I3, 2X, I3, 2X, I3, 2X, I3)
```

Hay una notación abreviada para lo anterior, que es:

```
950 format (4(2X, I3))
```

Es también posible permitir la repetición sin hacerlo explícitamente indicando las veces que el formato deberá repetirse. Supongamos que tenemos un vector, del cual se desea mostrar los primeros 50 elementos, con 10 elementos en cada línea. Se muestra una forma de hacerlo:

```

        write(*,1010) (x(i), i=1,50)
1010   format (10I6)

```

La sentencia `format` dice que 10 números deberán ser mostrados. Pero en la sentencia `write`, se hace con los primeros 50 números. Después de que los primeros 10 números han sido mostrados, la misma sentencia `format` es automáticamente usada para los siguientes 10 números y así sucesivamente.

12. E/S de Archivos

Se han estado haciendo ejemplos donde la salida/entrada se ha realizado a los dispositivos estándares de entrada/salida. También es posible leer o escribir de archivos los cuales son guardados en algún dispositivo externo de almacenamiento, por lo general un disco (disco duro, floppy) o una cinta. En Fortran cada archivo está asociado con un número de unidad, un entero entre 1 y 99. Algunos números están reservados: 5 es la entrada estándar, 6 es la salida estándar.

Abriendo y cerrando un archivo

Antes de que pueda usarse un archivo se requiere que sea abierto (`open`). El comando es

```
open (lista_de_especificadores)
```

donde los especificadores más comunes son:

```

[UNIT=]  u
IOSTAT=  ios
ERR=     err
FILE=    nomb_arch
STATUS=  sta
ACCESS=  acc
FORM=    frm
RECL=    rl

```

u	El número de unidad es un número en el rango de 9-99 para algún archivo, el programador lo escoge debiendo ser un número único.
ios	es el identificador del estado de la E/S y debe ser una variable entera. El valor que regresa <code>ios</code> es cero si la sentencia fue exitosa y sino, regresa un valor diferente de cero.
err	es una etiqueta a la cual el programa saltará si hay un error.
nomb_arch	es una cadena de caracteres que contiene el nombre del archivo.
sta	es una cadena de caracteres que tiene que ser NEW, OLD o SCRATCH. Esta muestra el estatus del archivo. Un archivo scratch es aquel que es creado y borrado cuando el archivo es cerrado (o el programa termina).
acc	deberá ser SEQUENTIAL o DIRECT. El valor predefinido es SEQUENTIAL.
frm.	deberá ser FORMATTED o UNFORMATTED. El valor predefinido es UNFORMATTED
rl	indica la longitud de cada registro en un archivo de acceso directo.

Para más detalles en los especificadores, se recomienda que se revise un buen libro de Fortran.

Una vez que un archivo ha sido abierto, se puede acceder con sentencias de lectura y escritura. Cuando se manipula un archivo y se termina de usar, deberá ser cerrado usando la sentencia.

```
close ([UNIT=]u[, IOSTAT=ios, ERR=err, STATUS=sta])
```

donde, los parámetros en bracket [] son opcionales

Complemento de read and write

El único cambio necesario de los ejemplos previos de las sentencias read/write, es que el número de unidad debe ser indicado. Pero se pueden incluir especificadores adicionales. Se muestra como:

```
read ([UNIT=]u, [FMT=]fmt, IOSTAT=ios, ERR=err, END=s)  
write([UNIT=]u, [FMT=]fmt, IOSTAT=ios, ERR=err, END=s)
```

donde la mayoría de los especificadores han sido descritos anteriormente. El especificador END=s define a que sentencia saltará el programa si se alcanza el fin del archivo (eof).

Ejemplo

Se da un archivo de datos con las coordenadas xyz de un grupo de puntos. El número de puntos es dado en la primera línea. El nombre del archivo de datos es puntos.dat. El formato para cada coordenada es de la forma F10.4 Se muestra un programa que lee los datos y los pone en tres arreglos x, y, z. Se presenta la versión en la que se ha conservado el formato original del programa en F77, el cual contiene sentencias GOTO. Así mismo se presenta una versión estructurada en formato F90.

```
program entdat  
c  
c Este programa lee n puntos desde un archivo de datos y los guarda en  
c 3 arreglos x, y, z.  
c Versión F77  
c  
    integer nmax, u  
    parameter (nmax=1000, u=20)  
    real x(nmax), y(nmax), z(nmax)  
  
c Abrir el archivo de datos  
    open (u, FILE='puntos.dat', STATUS='OLD')  
  
c Leer el número de puntos  
    read(u,*) n  
    if (n.GT.nmax) then  
        write(*,*) 'Error: n = ', n, 'es más largo que nmax = ', nmax  
        goto 9999  
    endif  
  
c Ciclo para recuperar los datos  
    do 10 i= 1, n  
        read(u,100) x(i), y(i), z(i)  
10 end do  
100 format (3(F10.4))  
  
c Cerrar el archivo  
    close (u)  
  
c Ahora se procesarán los datos  
    .....  
9999 stop  
    end
```

```

program entdat
!
! Este programa lee n puntos desde un archivo de datos y los guarda en
! 3 arreglos x, y, z.
! Versión F90 por Julio R. Baeza Pereyra
!
integer, parameter :: nmax = 1000, u = 20
real dimension (nmax) :: x, y, z

! Abrir el archivo de datos:
  open (u, FILE='puntos.dat', STATUS='OLD')
! Leer el número de puntos:
  read(u,*) n
  if (n > nmax) then
    write(*,*) 'Error: n = ', n, 'es más largo que nmax = ', nmax
  else
    ! Ciclo para recuperar los datos
    do i= 1, n
      read(u,100) x(i), y(i), z(i)
    end do
    format (3(F10.4))
100
    ! Cerrar el archivo
    close (u)
    ! Ahora se procesarán los datos
    .....
  endif
stop
end

```

13. Arreglos en subprogramas

Las llamadas a subprogramas en Fortran están basadas en llamadas por referencia. Lo que significa que los parámetros con los que son llamadas los subprogramas no son copiados al subprograma, sino que son pasadas las direcciones de los parámetros. Con lo anterior se ahorra una gran cantidad de espacio cuando se manipulan arreglos. No se requiere almacenamiento adicional ya que la subrutina manipula las mismas localidades de memoria como lo hace el código que hizo la llamada. Como programador se debe conocer y tener en cuenta lo anterior.

Es posible declarar un arreglo local en un subprograma en Fortran, pero es muy poco usado. Por lo general, todos los arreglos son declarados (y dimensionados) en el programa principal y entonces son pasados a los subprogramas conforme se van necesitando.

Arreglos de Longitud Variable

Una operación básica con un vector es la operación saxpy, la cual calcula la expresión

$$y = \alpha * x + y$$

donde alpha es un escalar y x e y son vectores. Se muestra una subrutina simple para esto:

```

subroutine saxpy (n, alpha, x, y)
integer n
real alpha, x(*), y(*)
! Versión F90 por Julio R. Baeza Pereyra
! Saxpy: Calcula  $y := \alpha * x + y$ ,
! donde x e y son vectores de longitud n (al menos).
!
! Variables Locales
      integer i
! Sentencias :
      do i = 1, n
          y(i) = alpha*x(i) + y(i)
      end do
return
end

```

La única característica nueva es el uso del asterisco en las declaraciones de x(*) y y(*). Con la notación anterior se indica que x e y son arreglos de longitud arbitraria. La ventaja es que se puede usar la misma subrutina para vectores de cualquier longitud. Se debe recordar que como Fortran esta basado en llamadas por referencia, no se requiere espacio adicional, ya que la subrutina trabaja directamente en el arreglo de elementos de la rutina o programa que la llamo. Es la responsabilidad del programador asegurarse que los vectores x e y han sido realmente declarados para tener longitud n o mayor en algún lugar del programa. Un error común en Fortran sucede cuando se intenta acceder arreglos del elemento fuera de los límites.

Se pudieron también haber declarado los arreglos como:

```

real, dimension (n) :: x, y

```

Muchos programadores prefieren usar la notación asterisco para recalcar que la "longitud verdadera del arreglo" es desconocida. Algunos programas viejos de F77 podrían declarar arreglos de longitud variable de la siguiente forma:

```

real x(1), y(1)

```

La sintaxis anterior es válida, aunque los arreglos sean mayores que uno, pero es un estilo pobre de programación y no se recomienda hacerlo.

Pasando subsecciones de un arreglos

Ahora se quiere escribir una subrutina para la multiplicación de matrices. Hay dos formas básicas para hacerlo, ya sea haciendo productos internos u operaciones saxpy. Se intentará ser modular y reusar el código saxpy de la sección previa. Un código simple es dado a continuación.

```

subroutine matvec (m, n, A, lda, x, y)
integer :: m, n, lda
real :: x(*), y(*), A(lda,*)
!
! Calcular  $y = A * x$ , donde A es m por n y guardado en un arreglo
! con dimensión principal lda.
!
! Variables locales
integer :: i, j
! Inicializar y

```

```

        do i = 1, m
            y(i) = 0.0
        end do
        ! Producto Matriz-vector por saxpy en las columnas de A.
        ! Obsérvese que la longitud de cada columna de A es m, y no n
        do j = 1, n
            call saxpy (m, x(j), A(1,j), y)
        end do
    return
end

```

Hay varias cosas importantes de comentar. Primero, se debe observar que a pesar de que se pretendió escribir el código tan general como fuera posible para permitir dimensiones arbitrarias de m y n , se necesita todavía especificar la dimensión principal de la matriz A . La declaración de longitud variable (*) puede ser solamente usado para la última dimensión de un arreglo. La razón de lo anterior es la forma como F77 guarda un arreglo multidimensional.

Cuando se calcula $y=A*x$, se necesita acceder las columnas de A . La j -ésima columna de A es $A(1:m,j)$. Sin embargo, en F77 no se puede usar una sintaxis de subarreglo (pero se puede hacer en F90). En vez de eso se da un apuntador al primer elemento en la columna, el cual es $A(1,j)$ (no es realmente un apuntador, pero puede ser útil pensarlo como si fuera). Se sabe que las siguientes localidades de memoria contendrán los siguientes elementos del arreglo en esta columna. La subrutina saxpy tratará a $A(1,j)$ como el primer elemento de un vector, y no sabe nada de que este vector es una columna de una matriz.

Finalmente, se debe observar que se ha tomado por convención que las matrices tienen m renglones y n columnas. El índice i es usado como un índice de renglones (de 1 a m), mientras el índice j es usado como un índice de columnas (de 1 a n). Muchos programas de Fortran que manejan álgebra lineal usan esta notación, lo cual facilita mucho la lectura del código.

Dimensiones Distintas

Algunas veces puede ser benéfico tratar un arreglo de 1-dimensión como un arreglo de 2 dimensiones y viceversa. Es muy fácil hacerlo en Fortran. Se muestra un ejemplo muy simple. Otra operación básica con vectores es el escalamiento, por ejemplo, multiplicar cada elemento de un vector por la misma constante. La subrutina es la siguiente:

```

subroutine escala(n, alpha, x)
integer :: n
real    :: alpha, x(*)
!
! Variables Locales
integer :: i
do i = 1, n
    x(i) = alpha * x(i)
end do
return
end

```

Supongamos que ahora se tiene una matriz de m por n que se quiere escalar. En vez de escribir una nueva subrutina para lo anterior, se puede hacer tratando la matriz como un vector y usar la subrutina escala. Una versión sencilla se da a continuación:

```

        integer, parameter :: m=10, n=20
        real                :: alpha, A(m,n)

!      Algunas sentencias definen A ...

!      Ahora se quiere escalar A
        call escala(m*n, alpha, A)

```

Obsérvese que este ejemplo trabaja porque se asume que la dimensión declarada de x es igual a la dimensión actual de la matriz guardada en A. Esto no sucede siempre. Por lo general la dimensión principal lda es diferente de la dimensión actual de m, y se debe tener mucho cuidado para hacerlo correctamente. Se muestra un subrutina más robusta para escalar una matriz que usa la subrutina escala

```

subroutine mescala(m, n, alpha, A, lda)
integer :: m, n, lda
real    :: alpha, A(lda,*)
!
! Variables Locales
!
integer :: j

        do j = 1, n
                call escala(m, alpha, A(1,j) )
        end do
return
end

```

Esta versión trabaja aún cuando m no sea igual a lda ya que se escala una columna cada vez y solamente se procesan los m primeros elementos de cada columna (dejando los otros sin modificar).

14. Bloques Comunes

F77 no tiene variables tipo global, es decir, variables que se compartan en varias unidades del programa (subrutinas). La única forma para pasar información entre subrutinas ya se ha visto previamente, y es usando una lista de parámetros en la subrutina. Algunas veces es inconveniente, por ejemplo cuando muchas subrutinas comparten un conjunto grandes de parámetros. En tales casos se puede usar un bloque común (*common block*). Esta es una forma para indicar que ciertas variables podrían compartirse en ciertas subrutinas. Se recomienda en lo general, minimizar el uso de bloques comunes en los programas.

Ejemplo

Supongamos que se tienen dos parámetros alpha y beta, los cuales son usados por varias subrutinas. El siguiente ejemplo muestra como puede hacerse usando bloques comunes.

```

        program principal
c      algunas declaraciones
        real alpha, beta
        common /coeff/ alpha, beta
                sentencias

        stop
        end

        subroutine sub1 (algunos argumentos)
c      declaraciones de argumentos

```

```

real alpha, beta
common /coeff/ alpha, beta
    ... sentencias ...
return
end

c subroutine sub2 (algunos argumentos)
    declaraciones de argumentos
real alpha, beta
common /coeff/ alpha, beta

    ... sentencias ...

return
end

```

En el código anterior se ha definido un bloque común con el nombre `coeff`. El contenido del bloque común son las dos variables `alpha` y `beta`. Un bloque común puede contener todas las variables que se deseen, no se necesita que todas sean del mismo tipo. Cada subrutina que quiere usar algunas de las variables del bloque común, tiene que declarar todo el bloque.

Se observa que en este ejemplo se pudo haber evitado fácilmente los bloques comunes, pasando `alpha` y `beta` como argumentos. Una buena regla es tratar de evitar los bloques comunes si es posible. Hay pocos casos donde no es posible hacerlo.

Sintaxis

```
common / nombre / lista_de_variables
```

Se debe tener presente que la sentencia `common` deberá aparecer junto a las declaraciones de variables, es decir antes de las sentencias de ejecución. A bloques comunes diferentes deberán corresponderles nombres diferentes (como se hace con las variables). Las variables pueden pertenecer a más de un bloque común. Por último, las variables en un bloque común no necesitan tener los mismos nombres que tienen en el lugar que ocurren (sin embargo no es mala idea hacerlo), pero si deberán aparecer en el mismo orden y con el mismo tipo.

Para mostrar lo anterior, se puede ver el siguiente ejemplo:

```

c subroutine sub3 (algunos argumentos)
    declaraciones de argumentos
real a, b
common /coeff/ a, b

    sentencias
return
end

```

Esta declaración es equivalente a la de la versión previa donde fue usada `alpha` y `beta`. Se recomienda que siempre se use el mismo nombre de la variable que aparece en el bloque común, para evitar confusión. Se muestra un ejemplo de algo que no debe hacerse:

```

c subroutine sub4 (algunos argumentos)
    declaraciones de argumentos
real alpha, beta
common /coeff/ beta, alpha

```

```

sentencias
return
end

```

Ahora alpha es la beta del programa principal y viceversa. Si se ve algo como lo anterior, es probable que se tenga un error de lógica. Tales errores son muy difíciles de encontrar.

Arreglos en Bloques Comunes

Los bloques comunes pueden incluir arreglos también, pero nuevamente no es recomendado, la razón principal es por flexibilidad. Un ejemplo que muestra porque es una mala idea. Suponiendo que se tienen las siguientes declaraciones en el programa main:

```

c      program main
        integer nmax
        parameter (nmax=20)
        integer n
        real A(nmax, nmax)
        common /matriz/ A, n, nmax
        ... etc ...
        stop
        end

```

Este bloque común contiene todos los elementos de A, y los enteros n y nmax. Supongamos que se quiere usar la matriz A en algunas subrutinas. Entonces se tiene que incluir la misma declaración en todas las subrutinas, por ejemplo:

```

c      subroutine subl (...)
        integer nmax
        parameter (nmax=20)
        integer n
        real A(nmax, nmax)
        common /matriz/ A, n, nmax
        ... etc ...
        return

```

Los arreglos con dimensiones variables no pueden aparecer en bloques comunes, por lo tanto el valor de nmax tiene que ser exactamente el mismo que el del programa principal. Recordar que el tamaño de la matriz tiene que ser conocido en tiempo de compilación, por eso nmax pertenece al bloque común, pero esto será ilegal.

El ejemplo muestra que no hay ganancia poniendo arreglos en bloques comunes. Por lo que el método preferido en F77 es pasar arreglos como argumentos a las subrutinas (junto con las dimensiones principales).

15. Datos y bloques de datos

La sentencia `data`

La sentencia `data` es otra forma de introducir datos que son conocidos cuando se esta escribiendo el programa. Es similar a la sentencia de asignación. La sintaxis es:

```

data lista_de_variables/ lista_de valores/, ...

```

donde los puntos suspensivos significan que el patrón puede ser repetido. Se muestra a continuación un ejemplo:

```
data m/10/, n/20/, x/2.5/, y/2.5/
```

Se podría haber escrito también como:

```
data m, n/10,20/, x, y/2*2.5/
```

Se podría haber hecho también lo mismo usando asignaciones

```
m = 10  
n = 20  
x = 2.5  
y = 2.5
```

La sentencia `data` es más compacta y por lo tanto más conveniente. Obsérvese la forma compacta para la asignación de valores iguales varias veces. La sentencia `data` es hecha sólo una vez, exactamente antes de que la ejecución del programa inicie. Por esta razón, la sentencia `data` es usada principalmente en el programa principal y no en las subrutinas.

La sentencia `data` puede ser usada también para inicializar arreglos (vectores, matrices). El siguiente ejemplo muestra como se puede asegurar que una matriz este llena de ceros antes de iniciar la ejecución del programa:

```
real A(10,20)  
data A/200*0.0/
```

Algunos compiladores inicializarán automáticamente los arreglos como en el ejemplo, pero no todos, por lo que si se quiere asegurar de los elementos del arreglo son cero, se deberá hacer algo como lo anterior. Por supuesto que los arreglos pueden ser inicializados con otro valor diferente de cero, o aún, inicializar los elementos individualmente, como se muestra a continuación:

```
data A(1,1)/ 12.5/, A(2,1)/ -33.3/, A(2,2)/ 1.0/
```

O se pueden listar todos los elementos para arreglos pequeños como se muestra enseguida:

```
integer v(5)  
real B(2,2)  
data v/10,20,30,40,50/, B/1.0,-3.7,4.3,0.0/
```

Los valores para arreglos bidimensionales serán asignados en el orden de primero columna como se acostumbra.

La sentencia `block data`

La sentencia `data` no puede ser usada dentro de un `common block`. Hay una "subrutina" especial para este caso, llamada `block data`. Esta no es realmente una subrutina, pero es parecida porque se da en una unidad de programa separada. Se muestra el ejemplo:

```

block data
  integer nmax
  parameter (nmax=20)
  real v(nmax), alpha, beta
  common /vector/ v, alpha, beta
  data v/20*100.0/, alpha/3.14/, beta/2.71/
end

```

Tal como en la sentencia `data`, el `block data` es ejecutado antes de que la ejecución del programa inicie. La posición de la "subrutina" `block data` en el código fuente es irrelevante (siempre y cuando no este anidada en el programa principal o un subprograma).

16. Estilo de programación con Fortran

Hay muchos estilos diferentes de programación, pero se intentará dar algunas guías generales que son de aceptación general.

Estructura del Programa

La estructura total del programa deberá ser modular. Cada subprograma deberá resolver una tarea bien definida. Mucha gente prefiere escribir cada subprograma en un archivo por separado.

Comentarios

Se repite lo que se había indicado previamente: Escriba código legible, pero también agregue comentarios al código fuente para explicar lo que se está haciendo. Es especialmente importante tener una buena cabecera para cada subprograma que explique cada argumento de entrada/salida y que hace el subprograma.

Sangrado

Se debe siempre usar el sangrado apropiado para bloques de ciclos y sentencias `if` como se mostró en el tutorial.

Variables

Declarar siempre todas las variables. No se recomienda la declaración implícita. Asegurarse que los primeros 6 caracteres son únicos.

Subprogramas

Nunca se debe permitir que las funciones tengan "efectos laterales", por ejemplo no se deben cambiar los valores de los parámetros de entrada. Usar subrutinas en tales casos.

En las declaraciones separar los parámetros, bloques comunes y variables locales.

Minimizar el uso de bloques comunes.

Goto

Minimizar el uso de la sentencia `goto`. Desafortunadamente si se requiere usar `goto` en algunos ciclos, ya que el ciclo `while` no es estándar en Fortran y esto puede variar de una a otra versión .

Arreglos

En muchos casos es mejor declarar todos los arreglos grandes en el programa principal y entonces pasarlos como argumentos a las distintas subrutinas. De esta forma toda la asignación de espacio es hecha en un sólo lugar. Recordar que se deben pasar también las dimensiones principales. Evitar el innecesario "redimensionamiento de matrices".

Asuntos de Eficiencia

Cuando se tenga un ciclo doble que esta accediendo a un arreglo bidimensional, es usualmente mejor tener el primer índice (renglón) dentro del arreglo más interno. Lo anterior es debido al esquema de almacenamiento en Fortran.

Cuando se tengan sentencias `if-then-elseif` con condiciones múltiples, intentar colocar primero aquellas condiciones que vayan a ser las más frecuentes que ocurran.

17. Sugerencias de depuración

Se ha estimado que cerca del 90% del tiempo que toma desarrollar un software comercial se usa en depurar y probar. Lo anterior no dice lo importante que es escribir buen código desde el primer momento. Todavía, se deben descubrir los errores ("bugs"). A continuación algunas sugerencias de como descubrirlos.

Opciones útiles del compilador

Muchos compiladores de Fortran tienen un conjunto de opciones que pueden ser activadas si así se desea.

Algunos errores comunes

Se muestran algunos errores comunes que se deben vigilar:

Si se trabaja con un compilador F77, hay que asegurarse que las líneas terminen antes de la columna 72. El resto será ignorado.

¿Se corresponde la lista de parámetros con la lista de argumentos que se están pasando?

¿Se corresponden los bloques comunes?

¿Se esta haciendo división entera cuando se quiere división real?

Herramientas de depuración:

Si se tiene un error, se debe intentar localizarlo. Los errores de sintaxis son fáciles de hallar. El problema es cuando se tienen errores en tiempo de ejecución. La forma vieja para encontrar errores es agregar sentencias `write` en el código y tratar de llevarles la pista a las variables. Esto es un poco tedioso ya que se debe recompilar el código fuente cada vez que se haga algún cambio. Actualmente se pueden usar los depuradores (debuggers) que son una herramienta muy conveniente. Se puede avanzar en pasos a través del programa, ya sea línea por línea o se pueden colocar puntos de interrupción (breakpoints).

18. Características Principales de Fortran 90

Formato libre en el código fuente.

En F90 se puede usar el formato de entrada de F77 o el formato libre. Si se usa el formato libre, la extensión *.f90 deberá ser usada para el nombre del archivo.

Apuntadores y asignación dinámica.

Es posible usar almacenamiento dinámico, con lo que se puede hacer que todos los arreglos "trabajen" no importando su tamaño.

Tipos de datos definidos por el usuario.

Se pueden definir tipos compuestos de datos a partir de los tipos básicos que se proporcionan con las diferentes versiones, de forma parecida a como se hace en C++ con `struct` o en Pascal con `record`.

Módulos.

Los módulos permiten hacer una programación en un estilo orientado a objetos parecido a como se hace en C++. Los módulos pueden también ser usados para ocultar variables globales, por lo que hace que la construcción `common` caiga en desuso.

Funciones recursivas.

Ahora como una parte del lenguaje.

Operaciones con arreglos construidas internamente

Las sentencias como $A = 0$ y $C = A + B$ son ahora válidas cuando A, B y C son arreglos. También hay una función para la multiplicación de matrices (`matmul`).

Sobrecarga de operadores.

Se pueda definir un significado propio para operadores como $+ y =$ para los propios tipos de datos (objetos).

Hay otras muchas características, bastante numerosas para ser mencionadas. F90 es muy diferente a las primeras versiones de Fortran. Pero guarda compatibilidad con las anteriores, F77 ha sido incluido como un subconjunto de F90.

Bibliografía

1. Chapman, Stephen J. (1997). "*Fortran 90/95 for scientists and engineers*". McGraw Hill, ISBN 0-07-011938-4
2. Lawrence, Norman (2002), "*Compaq Visual Fortran: A guide to creating windows applications*", Elsevier Digital Press, ISBN 1-55558-249-4
3. Universidad de Michoacán (2000), "Tutorial de Fortran", http://fismat.umich.mx/mn1/tutor_fort/index.html.