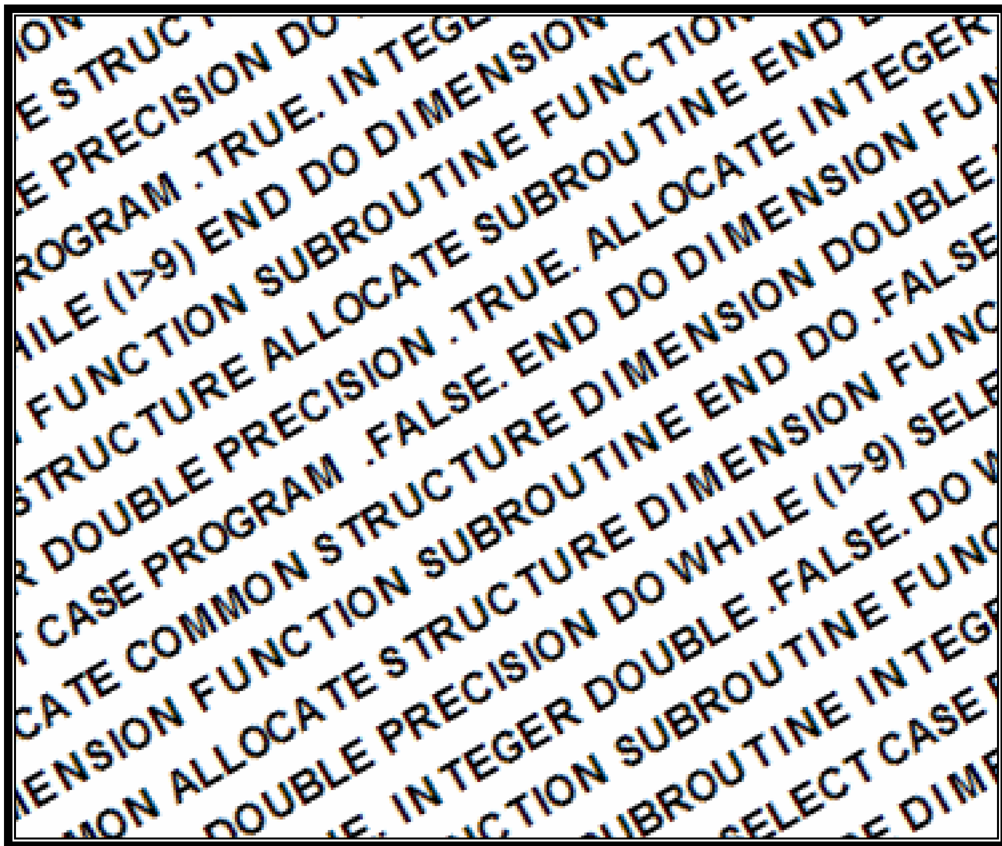
FUNDAMENTOS DE INFORMÁTICA

PROGRAMACIÓN EN FORTRAN



Pilar Bernardos Llorente

Departamento de Matemática Aplicada y Ciencias de la Computación

Escuela Técnica Superior de Ingenieros Industriales y de Telecomunicación

Universidad de Cantabria, 2008

PRÓLOGO

El propósito de este libro es recopilar los aspectos básicos del lenguaje de programación Fortran 90/95.

Está organizado en siete capítulos de nivel de complejidad creciente. Los capítulos no son independientes entre sí, de modo que su lectura no puede realizarse en cualquier orden, sino en el que se propone en este libro.

Cada capítulo contiene una primera parte teórica y otra parte práctica que consta a su vez de una serie de ejercicios resueltos y propuestos. Se pretende que, a la vista de la teoría y los programas que se presentan, el alumno sea capaz de construir programas de dificultad similar, tales como los que se plantean en los ejercicios propuestos.

Este libro está pensado para cualquier persona que quiera iniciarse en el lenguaje de programación Fortran. Lógicamente, es imprescindible que el aprendizaje se lleve a cabo con un computador.

Todos los programas presentados funcionan en los entornos de programación Compaq Visual Fortran versión 6.5 y Plato 3 versión 3.20 de la Salford Software, Ltd 2005. Este último entorno es de distribución gratuita en Internet para uso personal. Ambos permiten compilar el lenguaje de programación Fortran con formatos libre y fijo y Plato 3 a su vez también compila otros lenguajes como Java, C++, C#.

El libro sirve de guía básica a los alumnos de primer curso de Ingeniería Industrial y Química de la Universidad de Cantabria en la asignatura de Fundamentos de Informática para la parte de programación en el lenguaje Fortran. No obstante, en ningún caso, pretende sustituir a un manual de referencia del lenguaje.

TABLA DE CONTENIDOS

0 INTRODUCCIÓN 11

0.1 ¿QUÉ ES FORTRAN? 11
0.2 CAMBIOS EN LOS DIFERENTES ESTÁNDARES FORTRAN 12
0.3 ¿POR QUÉ FORTRAN? 12
0.4 ELEMENTOS DEL LENGUAJE 13

1 TIPOS DE DATOS Y LOS PRIMEROS PASOS: LEER, CALCULAR, ESCRIBIR 15

1.1 JUEGO DE CARACTERES FORTRAN 15
1.2 ESTRUCTURA DE UN PROGRAMA FORTRAN 15
1.3 PROGRAM 16
1.4 STOP 16
1.5 EJEMPLO DE USO DE STOP 17
1.6 END PROGRAM 17
1.7 FORMATO DE LAS LÍNEAS EN FORTRAN 90/95 17
1.8 TIPOS DE DATOS 18
1.9 CONSTANTES EN FORTRAN 19
1.9.1 CONSTANTES ENTERAS 20
1.9.2 CONSTANTES REALES 20
1.9.3 CONSTANTES LÓGICAS 20
1.9.4 CONSTANTES COMPLEJAS 21
1.9.5 CONSTANTES CARÁCTER 21
1.10 IDENTIFICADORES 21
1.11 VARIABLES 21
1.11.1 DECLARACIÓN EXPLÍCITA 22
1.11.2 DECLARACIÓN IMPLÍCITA 22
1.12 INICIALIZACIÓN DE VARIABLES 23
1.13 CONSTANTES CON NOMBRE: PARAMETER 23
1.14 EXPRESIONES ARITMÉTICAS 24
1.14.1 REGLAS DE PRECEDENCIA DE OPERADORES ARITMÉTICOS ... 24
1.14.2 EJEMPLO DE REGLAS DE PRECEDENCIA 25
1.15 ARITMÉTICA CON TIPOS MEZCLADOS 25
1.16 ASIGNACIÓN ARITMÉTICA 27

1.17 FUNCIONES INTRÍNSECAS FORTRAN 27

EJERCICIOS RESUELTOS 29

EJERCICIOS PROPUESTOS 34

2 ESTRUCTURAS DE CONTROL CONDICIONALES 35

2.1 EXPRESIONES LÓGICAS RELACIONALES 35
2.2 EJEMPLOS DE EXPRESIONES LÓGICAS RELACIONALES 36
2.3 EXPRESIONES LÓGICAS COMBINACIONALES 36
2.4 PRECEDENCIAS LÓGICAS-ARITMÉTICAS 37
2.5 SENTENCIA DE ASIGNACIÓN LÓGICA 38
2.6 BLOQUE IF 39
2.7 BLOQUE IF CON NOMBRE 40
2.8 EJEMPLOS DE BLOQUES IF.. 41
2.9 IF LÓGICO 41
2.10 BLOQUE SELECT CASE 42
2.11 EJEMPLOS DE BLOQUE SELECT CASE 42

EJERCICIOS RESUELTOS 45

EJERCICIOS PROPUESTOS 53

3 ESTRUCTURAS DE CONTROL REPETITIVAS. BUCLES..... 55

3.1 ESTRUCTURAS DE REPETICIÓN 55
3.2 REPETICIÓN CONTROLADA POR CONTADOR O BUCLE DO ITERATIVO . 55
3.3 REPETICIÓN CONTROLADA POR EXPRESIÓN LÓGICA O BUCLE WHILE 57
3.4 BUCLE DO WHILE... 59
3.5 SENTENCIAS EXIT Y CYCLE 61
3.6 BUCLES CON NOMBRE.. 62
3.7 BUCLES ANIDADOS 63
3.8 BUCLES ANIDADOS DENTRO DE ESTRUCTURAS IF Y VICEVERSA........ 64

EJERCICIOS RESUELTOS 65

EJERCICIOS PROPUESTOS 73

4 ARRAYS 75

4.1 INTRODUCCIÓN 75
4.2 DECLARACIÓN DE ARRAYS 75
4.3 REFERENCIA A LOS ELEMENTOS DE UN ARRAY 76
4.4 INICIALIZACIÓN DE ARRAYS 77
4.4.1 INICIALIZACIÓN DE ARRAYS EN SENTENCIAS DE DECLARACIÓN DE TIPO 77
4.4.2 INICIALIZACIÓN DE ARRAYS EN SENTENCIAS DE ASIGNACIÓN 78
4.4.3 INICIALIZACIÓN DE ARRAYS EN SENTENCIAS DE LECTURA 79
4.5 OPERACIONES SOBRE ARRAYS COMPLETOS 81
4.6 OPERACIONES SOBRE SUBCONJUNTOS DE ARRAYS 82
4.6.1 TRIPLETES DE ÍNDICES 82
4.6.2 VECTORES DE ÍNDICES 83
4.7 CONSTRUCCIÓN WHERE 83
4.8 SENTENCIA WHERE 85
4.9 CONSTRUCCIÓN FORALL 85
4.10 SENTENCIA FORALL 86
4.11 ARRAYS DINÁMICOS 87

EJERCICIOS RESUELTOS 89

EJERCICIOS PROPUESTOS 105

5 PROCEDIMIENTOS 107

5.1 DISEÑO DESCENDENTE 107
5.2 FUNCIONES 108
5.3 SUBRUTINAS 111
5.4 TRANSFERENCIA DE ARRAYS A PROCEDIMIENTOS 113
5.5 COMPARTIR DATOS CON MÓDULOS 114
5.6 PROCEDIMIENTOS MÓDULO 116
5.7 PROCEDIMIENTOS COMO ARGUMENTOS 117
5.8 ATRIBUTO Y SENTENCIA SAVE 118
5.9 PROCEDIMIENTOS INTERNOS 119
5.10 PROCEDIMIENTOS RECURSIVOS 119

5.11 ARGUMENTOS OPCIONALES Y CAMBIOS DE ORDEN 120

EJERCICIOS RESUELTOS 123

EJERCICIOS PROPUESTOS 137

6 CARACTERES Y CADENAS..139

6.1 CARACTERES Y CADENAS..139
6.2 EXPRESIÓN CARÁCTER..140
6.3 ASIGNACIÓN CARÁCTER 140
6.4 FUNCIONES INTRÍNSECAS CARÁCTER 141

EJERCICIOS RESUELTOS 145

EJERCICIOS PROPUESTOS 154

7 FORMATOS Y ARCHIVOS..155

7.1 ENTRADA/SALIDA EN FORTRAN 155
7.2 SALIDA POR PANTALLA..155
7.3 ENTRADA POR TECLADO 156
7.4 DESCRIPTORES DE FORMATO 157
7.4.1 DESCRIPTOR I DE FORMATO ENTERO...............................158
7.4.2 DESCRIPTOR F DE FORMATO REAL..................................158
7.4.3 DESCRIPTOR E DE FORMATO EXPONENCIAL 159
7.4.4 DESCRIPTOR ES DE FORMATO CIENTÍFICO 160
7.4.5 DESCRIPTOR L DE FORMATO LÓGICO 161
7.4.6 DESCRIPTOR A DE FORMATO CARÁCTER..........................162
7.4.7 DESCRIPTORES X, T DE POSICIÓN HORIZONTAL Y / DE POSICIÓN VERTICAL 163
7.4.8 REPETICIÓN DE GRUPOS DE DESCRIPTORES DE FORMATO 164
7.5 PROCESAMIENTO DE ARCHIVOS..165
7.6 POSICIÓN EN UN ARCHIVO...166
7.7 SALIDA POR ARCHIVO...167
7.8 ENTRADA POR ARCHIVO..168

EJERCICIOS RESUELTOS 171

EJERCICIOS PROPUESTOS 178

BIBLIOGRAFÍA 181

ÍNDICE DE TABLAS

Tabla 1.1: Ejemplo de uso de STOP 17
Tabla 1.2: Tipos de datos intrínsecos en Fortran 19
Tabla 1.3: Operadores aritméticos Fortran 24
Tabla 1.4: Reglas de precedencia de operadores aritméticos 25
Tabla 1.5: Ejemplo de reglas de precedencia de operadores aritméticos 25
Tabla 1.6: Orden de precedencia de los tipos Fortran.......................... 26
Tabla 1.7: Aritmética con tipos mezclados.. 26
Tabla 2.1: Operadores lógicos relacionales Fortran 35
Tabla 2.2: Ejemplos de expresiones lógicas relacionales 36
Tabla 2.3: Operadores lógicos combinacionales Fortran 90/95 37
Tabla 2.4: Tabla de verdad de los operadores lógicos combinacionales. 37
Tabla 2.5: Orden de precedencia de operadores lógicos combinacionales Fortran 37
Tabla 2.6: Orden de precedencia de operadores Fortran 38
Tabla 7.1: Símbolos usados en los descriptores de formatos 157
Tabla 7.2: Formatos de escritura de enteros 158
Tabla 7.3: Formatos de lectura de enteros..158
Tabla 7.4: Formatos de escritura de reales 159
Tabla 7.5: Formatos de lectura de reales..159
Tabla 7.6: Formatos de escritura de reales en formato exponencial 160
Tabla 7.7: Formatos de lectura de reales en formato exponencial........160
Tabla 7.8: Formatos de escritura de reales en formato científico.........161
Tabla 7.9: Formatos de lectura de reales en formato científico 161
Tabla 7.10: Formatos de escritura de datos lógicos...........................162
Tabla 7.11: Formatos de lectura de datos lógicos 162
Tabla 7.12: Formatos de escritura de caracteres 163
Tabla 7.13: Formatos de lectura de caracteres.................................163

0 INTRODUCCIÓN

0.1 ¿Qué es Fortran?

- Fortran es el primer lenguaje de programación de alto nivel creado en el año 1957 por obra de un equipo de científicos de IBM dirigido por John Backus.
- Por aquel entonces, sólo los científicos e ingenieros utilizaban los computadores para resolver problemas numéricos. Por tanto, la facilidad de aprendizaje del lenguaje equivalía a que la notación fuese un reflejo de la notación matemática. No en vano, Fortran deriva de las palabras inglesas FORmula TRANslation.
- Desde su creación en la década de los años 50 en IBM, ha sido y es ampliamente utilizado, habiendo pasado por un proceso de evolución que ha dado lugar a distintas versiones que, por convención, se identifican por los dos últimos dígitos del año en que se propuso el estándar correspondiente. Las distintas versiones son:
  - Fortran 66 publicada por ANSI[1] X3.9-1966.
  - Fortran 77 publicada por ANSI X3.9-1978 y ISO[2]/IEC 1539:1980.
  - Fortran 90 titulado *Programming Language "Fortran" Extended* (ANSI X3.198-1992 and ISO/IEC 1539:1991).
  - Fortran 95 titulado I*nformation technology - Programming languages - Fortran - Part 1: Base language* (ISO/IEC 1539: 1997).
  - Fortran 2003 titulado I*nformation technology - Programming languages - Fortran - Part 1: Base language*. (ISO/IEC 1539: 2004).
- Por supuesto, todas las versiones incluyen a las anteriores. Así, cualquier programa escrito en Fortran 66, Fortran 77 o Fortran 90, compila, sin problemas, en un compilador Fortran 95.
- Debemos remarcar, no obstante, que en general los compiladores de Fortran que proporcionan las distintas casas de hardware y/o

---

[1] ANSI (American National Standards Institute).

[2] ISO (International Standards Organization).

software son versiones ampliadas, que permiten la utilización de extensiones del lenguaje no normalizadas.

- La ventaja de la normalización del Fortran, supone que sea fácilmente transportable a cualquier entorno informático que disponga de un compilador compatible con el estándar.
- Si la transportabilidad es un requisito, hay que procurar evitar las extensiones no normalizadas que incorporan los distintos fabricantes.

0.2 Cambios en los diferentes estándares Fortran

- Fortran 90 incluye todas las características de las versiones anteriores, de forma que las inversiones en software están protegidas. Pero añade un conjunto de características para hacerlo competitivo con lenguajes más modernos:
    - Las sentencias se pueden escribir en un formato libre.
    - Se permiten nombres más largos.
    - Se crean nuevas construcciones de control para ejecución selectiva y repetitiva.
    - Aparecen nuevos tipos de subprogramas para facilitar la programación modular.
    - Nuevos mecanismos de procesamiento de matrices.
    - Tipos de datos definidos por el usuario.
    - Punteros y estructuras de datos dinámicas.
- Fortran 95 es una revisión menor del estándar anterior, que añade características para programación paralela del dialecto *High Performance* Fortran, tales como funciones puras y elementales definidas por el usuario, y la construcción FORALL.
- Fortran 2003 presenta como características nuevas más importantes: soporte para el manejo de excepciones, programación orientada a objeto e interoperatividad mejorada con el lenguaje C.

0.3 ¿Por qué Fortran?

Hoy en día el aprendizaje del Fortran es interesante porque:

- Es el lenguaje predominante en aplicaciones matemáticas, científicas y de ingeniería.
- Es un lenguaje fácil de aprender y utilizar.
- Es el único lenguaje que perdura desde los años 50 hasta el momento actual.
- Existen miles de programas de cálculo, y librerías de uso absolutamente generalizado: IMSL (International Mathematics and Statistical Library), NAG (Numerical Algorithms Group), etc.

0.4 Elementos del lenguaje

- Un programa Fortran es simplemente una secuencia de líneas de texto o instrucciones.
- Como en cualquier lenguaje de programación, el texto que aparece en las distintas líneas del programa debe seguir una sintaxis determinada, de forma que dé lugar a un programa Fortran construido correctamente.
- Cuando aparece una instrucción Fortran nueva en el texto, se explica su sintaxis general, utilizando para ello los siguientes criterios:
    - La instrucción aparece resaltada en gris.
    - Las palabras reservadas del lenguaje se escriben en mayúsculas.
    - Si aparecen corchetes, éstos no forman parte de la sintaxis e indican que lo que está dentro es opcional.
    - Si aparecen puntos suspensivos, éstos tampoco forman parte de la sintaxis e indican listas de elementos como el inmediatamente anterior.
    - Si aparecen llaves en las que hay varias opciones separadas con barras verticales { Op1 | Op2 | Op3 }, se trata de elegir una de esas opciones (en este caso, Op1, Op2 o Op3). Tanto las llaves como las barras verticales no forman parte de la sintaxis.
- Los ejercicios resueltos que aparecen al final de cada capítulo están codificados usando el formato libre de líneas, específico de Fortran 90, para mayor comodidad del usuario. Las reglas para escribir en este formato se explican en el capítulo 1, sección 7.

1 TIPOS DE DATOS Y LOS PRIMEROS PASOS: LEER, CALCULAR, ESCRIBIR

1.1 Juego de caracteres Fortran

- Fortran 90/95 tiene su propio alfabeto especial llamado *juego de caracteres Fortran*. Sólo los caracteres de su alfabeto pueden usarse en este lenguaje. Consta de los siguientes:
- Caracteres alfanuméricos:
    - Caracteres alfabéticos en mayúscula (26): A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.
    - Caracteres alfabéticos en minúscula (26): a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z.
    - Caracteres numéricos (10): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
    - Carácter de subrayado (1): _
- Caracteres especiales (22): = + - * ** / ( ) . , $ ': " % ; ! & < > ? <blanco>.
- Fortran no distingue entre caracteres alfabéticos en mayúsculas o en minúsculas, excepto cuando forman parte de cadenas de caracteres, como veremos en el capítulo 6.

1.2 Estructura de un programa Fortran

- La estructura general de un programa Fortran, como todas las unidades de programa[3] Fortran, consta de cuatro partes:
- Cabecera:
    - PROGRAM *nombre_programa*
    - Pequeña descripción del programa.
- Parte de especificaciones:
    - Define las variables e identificadores empleados en el programa.
- Parte ejecutable:

---

[3] Una unidad de programa es un trozo de código Fortran compilado separadamente. Se estudiarán en el capítulo 5.

    - Describe las acciones que llevará a cabo el programa.
- Parte de terminación:
    - END PROGRAM *nombre_programa*
- Pueden insertarse comentarios libremente en cualquier punto del programa: dentro, antes o después del mismo.

1.3 PROGRAM

- La sentencia `PROGRAM` define el nombre del programa Fortran que comienza la ejecución.
- Sintaxis:

[PROGRAM nombre_programa]

- *Nombre_programa* es el nombre del programa (y su punto de entrada).
- El nombre en una sentencia `PROGRAM` se usa única y exclusivamente con propósitos de documentación del programa.
- Si se utiliza la sentencia `PROGRAM`, deberá ser la primera sentencia no comentada del programa fuente.

1.4 STOP

- La sentencia `STOP` detiene la ejecución de un programa, y opcionalmente, imprime un mensaje en la salida estándar de errores.

STOP [n]

- *n* es una constante carácter o un entero de hasta 5 dígitos.
- `STOP` termina la ejecución de un programa Fortran, antes de que éste llegue al final de dicho programa. `STOP` también manda un mensaje a la salida estándar de errores si se ha especificado algo después de él (dígitos o cadena de caracteres).
- Un programa Fortran puede tener varias sentencias `STOP` (es decir, varios puntos de parada), por ello aunque el uso de *n* no es obligatorio, es conveniente, ya que nos dará una idea clara del punto en que ha parado el programa.
- Cuando la sentencia `STOP` precede inmediatamente a la sentencia `END PROGRAM` es opcional. El compilador genera automáticamente un comando `STOP` cuando alcanza la sentencia `END PROGRAM`.
- Entendiendo que un buen programa Fortran debe tener un único punto de entrada y otro de salida, sin ningún otro punto de parada intermedio, el uso de esta sentencia está desaconsejado.

1.5 Ejemplo de uso de STOP

CÓDIGO	DESCRIPCIÓN
STOP 7373	SE ESCRIBE "STOP 7373" EN LA SALIDA DE ERRORES
STOP 'ACABO'	SE ESCRIBE "STOP ACABO" EN LA SALIDA DE ERRORES
STOP	NO SE ESCRIBE NADA EN LA SALIDA DE ERRORES

Tabla 1.1: Ejemplo de uso de STOP

1.6 END PROGRAM

- Debe ser la última sentencia del programa.
- Esta sentencia indica al compilador que no hay más sentencias que compilar en el programa.
- Sintaxis:

END PROGRAM [nombre_programa]

- Ej:

PROGRAM ejemplo

……..

……..

END PROGRAM ejemplo

O bien:

!Programa sin nombre

……..

……..

END PROGRAM

1.7 Formato de las líneas en Fortran 90/95

- Al contrario que en el 77, Fortran 90/95 es mucho más flexible en el formato de líneas:
    - Una línea puede tener hasta 132 caracteres.
    - Una línea puede tener más de una sentencia, separadas por “;”.

- Si una sentencia continúa en la línea siguiente, se debe terminar la línea con el carácter ampersand "&" y, opcionalmente, también se puede empezar la línea siguiente. Se permiten hasta 39 líneas de continuación.
- Si una cadena de caracteres debe continuar en una línea adicional, se debe poner el carácter "&" al final de cada línea y al principio de las líneas de continuación.
- Los caracteres que siguen al "!" hasta el final de la línea son comentarios.
- Si una sentencia requiere una etiqueta, ésta debe ser numérica (1...99999), debe preceder a la sentencia y estar separada por, al menos, un blanco.

- Los espacios en blanco son ignorados en Fortran 90/95, excepto dentro de las palabras clave, nombres de las funciones intrínsecas y cadenas de caracteres. Así, se pueden usar tantos espacios en blanco e intercalar tantas líneas en blanco en un programa como se quiera para mejorar el aspecto del mismo.

1.8 Tipos de datos

- Hay 5 tipos de datos predefinidos llamados *intrínsecos* para constantes y variables Fortran.
- Cada uno de estos tipos de datos intrínsecos admiten varias longitudes de memoria, dependiendo de la anchura de la palabra del computador, conocidas como *clases*. Una de esas clases, la más habitual, se le llama clase por defecto.

TIPO	DENOMINACIÓN EN FORTRAN	BYTES OCUPADOS[4]	INTERVALO DE VALORES
ENTERO	INTEGER	4	-2^{32-1}=-2147483648 2^{32-1}-1=2147483647

[4] Depende del computador.

REAL	REAL	4	$-3.402823\ 10^{+39}$ $-1.175495\ 10^{-39}$ Y $1.175495\ 10^{-39}$ $3.402823\ 10^{+39}$
COMPLEJO	COMPLEX	8	IGUAL QUE REAL
LÓGICO	LOGICAL	4	.TRUE. O .FALSE.
CARÁCTER	CHARACTER [{(len=nº caract de variab)| (nº caract de variab)}]	len	CONJUNTO DE CARACTERES ASCII DE 8-BITS

Tabla 1.2: Tipos de datos intrínsecos en Fortran

- Suponiendo enteros de 4 Bytes, cualquier intento de usar un valor fuera del rango de representación dado en la Tabla 1.2, es decir, mayor o menor que sus valores límite, resulta en un llamado error de *overflow*.
- Para solucionar esta limitación, en parte, se incluye el tipo de dato real. Los números reales se caracterizan por dos cantidades: el *rango* que marca la diferencia entre los números mayor y menor que pueden representarse y la *precisión* o número de dígitos significativos que puede conservar un número. Para reales de 4 Bytes, el rango es aproximadamente de 10^{-38} a 10^{38} y la precisión de 7 dígitos. Así, 1000000.0 y 1000000.1 no pueden distinguirse en un programa Fortran en el que ambos son definidos de tipo real de 4 Bytes. Más adelante se aprenderá a usar reales de alta precisión.
- En cuanto a las variables declaradas de tipo carácter, su longitud viene dada por el número máximo de caracteres que van a almacenar. Si no aparece el paréntesis en la declaración de tipo, las variables de la lista sólo podrán almacenar un carácter.
- Además, Fortran 90/95 permite al programador definir tipos de datos *derivados*, que son tipos de datos especiales para resolver problemas particulares.

1.9 Constantes en Fortran

- Definición de constante:
  - Valor específico y determinado que se define al hacer un programa y que no cambia a lo largo del mismo.

- Cuando un compilador Fortran encuentra una constante, coloca el valor de la constante en una localización conocida de la memoria y hace referencia a esa localización cada vez que se usa la constante en el programa.

1.9.1 Constantes enteras

- Puede tomar únicamente un valor entero (positivo, negativo o cero). Se representa como un signo (opcional) seguido de una cadena no vacía de números.
- Ej:

39, -6743, +8899, 0, -89, 0012, 27, 3.

1.9.2 Constantes reales

- Básica.

Se compone de:

  - signo (opcional).
  - parte entera (secuencia de dígitos).
  - punto decimal.
  - parte fraccional (secuencia de dígitos).

- Ej: 12.343 -0.0032 86. .3475.
- Básica con exponente.

El exponente real se compone de:

  - carácter alfabético E.
  - signo (opcional).
  - constante entera (2 dígitos como máximo).

- Ej:

+34.453E4

12.323E+03

-0.0034E-03

.89E-2

5.434E0

-4344.49E-5

1.9.3 Constantes lógicas

- Una constante lógica puede tener únicamente los valores verdadero y falso.
  - .TRUE. Verdadero.
  - .FALSE. Falso.

1.9.4 Constantes complejas

- Una constante compleja representa un número complejo como un par ordenado de números reales. El primer número del par representa la parte real y el segundo la parte imaginaria. Cada parte vendrá codificada como un número real.
- Ej: (3.E34,0.332) (-3.329,-.3E9).

1.9.5 Constantes carácter

- Consiste en una cadena de caracteres.
- El carácter blanco es válido y significativo dentro de la cadena.
- La longitud de la constante carácter coincide con el número de caracteres que componen la cadena.
- Cada carácter de la cadena tiene una posición concreta que es numerada consecutivamente (comenzando en 1). El número indica la posición del carácter dentro de la cadena comenzando por la izquierda.
- Para codificar una constante carácter dentro de un programa deberá encerrarse entre dos caracteres delimitadores comilla simple o dobles comillas. La longitud de la cadena deberá ser mayor que 0.
    - Los delimitadores no forman parte de la constante en: ‘Hola que tal’ “hasta luego Lucas”.
    - Para que la constante carácter incluya comillas simples (dobles) rodearla de dobles (simples) comillas: “’Hola que tal’” ‘“hasta luego Lucas”’.

1.10 Identificadores

- Un identificador es un nombre que se usa para denotar programas, algunas constantes, variables y otras entidades.
- Los identificadores deben empezar con una letra y pueden tener hasta 31 letras, dígitos o caracteres de subrayado _.
    - Por ejemplo, identificadores válidos son:
    - Masa, MASA, Velocidad_de_la_luz, Sueldo_del_ultimo_mes, X007.
    - Y no válidos:
    - R2-D2, 34JUlio, pepe$.

1.11 Variables

- Una variable es una entidad que tiene un nombre y un tipo. Un nombre de variable es un nombre simbólico de un dato. Por tanto, ese dato podrá ser identificado, definido y referenciado a través de dicha variable.

- Cuando un compilador Fortran encuentra una variable, reserva espacio en memoria para esa variable de modo que, cada vez que se use la variable en el programa, se hace referencia a su ubicación en memoria.
- El nombre de las variables debe ser un identificador válido.
- El tipo de una variable puede venir determinado de forma explícita o implícita.

1.11.1 Declaración explícita

TIPO:: lista de variables

- TIPO es cualquiera de los tipos de datos Fortran válidos de la Tabla 1.2.
- *lista de variables* es un conjunto de variables separadas por comas cuyos nombres son identificadores válidos.
- Ej:

REAL:: radio, area

INTEGER:: mesas,sillas

COMPLEX:: z1,z2

LOGICAL:: testea

CHARACTER (len=20):: alumno1,alumno2

1.11.2 Declaración implícita

- Si una variable no ha sido definida explícitamente por una sentencia de definición de tipo, éste será determinado por la primera letra de su nombre:
  - I,J,K,L,M,N: entera.
  - resto de letras: real.
- Existe la posibilidad de gobernar los tipos implícitos. La sintaxis general es:

IMPLICIT {NONE | TIPO (lista de letras)}

- NONE significa que no hay nada implícito. Por lo tanto, todas las variables del programa deben declararse explícitamente.
- TIPO es cualquiera de los tipos de datos Fortran válidos de la Tabla 1.2.
- *lista de letras* es un conjunto de letras que corresponden a las iniciales de los nombres de las variables.
  - Si no son consecutivas en el alfabeto, van separadas por comas.
  - Si son consecutivas en el alfabeto, *letra_inicial-letra_final*.

- La declaración implícita puede ser una fuente de problemas. En su lugar, se recomienda usar la sentencia IMPLICIT NONE y declarar de forma explícita todas las variables y constantes que se vayan a emplear.
  - Ej:

IMPLICIT NONE

!No hay nada implícito.

IMPLICIT LOGICAL (H-J)

!Variables que comiencen por H,I,J son lógicas.

1.12 Inicialización de variables

- A partir de Fortran 90, las variables pueden inicializarse al ser declaradas.

TIPO:: var1=valor1[,var2=valor2]…

- Ej: REAL:: velocidad=3.25,aceleracion=0.75,espacio=10.9
- Ej: CHARACTER (len=10):: apell1,apell2,nombre='Santiago'
- El valor de una variable no inicializada no está definido por el estándar de Fortran 90/95.
  - Algunos compiladores automáticamente cargan un cero en esa variable,
  - otros cargan cualquier valor existente en la posición de memoria de esa variable y
  - otros incluso causan un error de ejecución al usar una variable que no ha sido previamente inicializada.

1.13 Constantes con nombre: PARAMETER

- Cuando en un programa se utilizan constantes conocidas, ya sean: físicas como la aceleración de la gravedad o la constante de Planck, matemáticas como el número *pi* o el número *e*, químicas como el número de Avogadro etc., es deseable escribirlas siempre igual y con la máxima precisión que acepte el computador.
- La mejor manera de conseguir consistencia y precisión en un programa en cuanto al uso de constantes conocidas es asignarles un nombre y usar ese nombre siempre para referirse a la tal constante a lo largo del programa.
- Las constantes con nombre se crean usando el atributo PARAMETER de una sentencia de declaración de tipo. La forma general de una sentencia de este estilo es:

TIPO, PARAMETER:: nombre1 = valor1[, nombre2 = valor2]…

- Permite dar un nombre simbólico a una expresión constante que será invariable en todo el programa
- En tiempo de compilación, se hace una sustitución del *nombre* por el *valor*.
- Cualquier intento de cambiar su valor con una sentencia de asignación o de lectura provoca un error de compilación.
- Ej:

REAL, PARAMETER:: PI = 3.14159, E = 2.71828

1.14 Expresiones aritméticas

- Los operadores aritméticos son:

OPERADOR	DESCRIPCIÓN
OPERADORES BINARIOS	
+	SUMA
-	RESTA
*	MULTIPLICACIÓN
/	DIVISIÓN
**	POTENCIA
OPERADORES UNARIOS	
+	SIGNO POSITIVO
-	SIGNO NEGATIVO

Tabla 1.3: Operadores aritméticos Fortran

- No se pueden escribir dos operadores adyacentes.
- Ej:

La expresión a**-b es ilegal en Fortran y debe ser escrita como a**(-b).

1.14.1 Reglas de precedencia de operadores aritméticos

OPERADOR(ES)	PRECEDENCIA
( )	MAYOR
**	

+ , - (UNARIOS: SIGNOS)	
* , /	
+ , -	MENOR

Tabla 1.4: Reglas de precedencia de operadores aritméticos

- Si una expresión contiene dos o más operadores de la misma precedencia se siguen las siguientes reglas:
    - Cuando existen paréntesis anidados se evalúan desde el más interno hasta el más externo. Es conveniente usar tantos paréntesis como sean necesarios en una expresión para hacerla más clara y fácil de comprender.
    - Las operaciones de potencia se evalúan de derecha a izquierda.
    - Multiplicación/división y suma/resta se evalúan de izquierda a derecha.

1.14.2 Ejemplo de reglas de precedencia

- Sea la sentencia: A*B+C+D**(E*(F-6.25+G))**SIN(H):

PARTE EVALUADA	EXPRESIÓN RESULTANTE
OP1=F-6.25	A*B+C+D**(E*(OP1+G))**SIN(H)
OP2=(OP1+G)	A*B+C+D**(E*OP2)**SIN(H)
OP3=(E*OP2)	A*B+C+D**OP3**SIN(H)
OP4=OP3**SIN(H)	A*B+C+D**OP4
OP5=D**OP4	A*B+C+OP5
OP6=A*B	OP6+C+OP5
OP7=OP6+C	OP7+OP5
RESULTADO=OP7+OP5	VALOR FINAL DE LA EXPRESIÓN

Tabla 1.5: Ejemplo de reglas de precedencia de operadores aritméticos

1.15 Aritmética con tipos mezclados

- Si se mezclan en una expresión operandos de distintos tipos, el resultado se eleva a la categoría del mayor, según el siguiente orden:

TIPO	PRECEDENCIA
COMPLEX	MAYOR
REAL	
INTEGER	
LOGICAL	MENOR

Tabla 1.6: Orden de precedencia de los tipos Fortran

+,-,*,/,**	INTEGER	REAL
INTEGER	INTEGER	REAL
REAL	REAL	REAL

Tabla 1.7: Aritmética con tipos mezclados

+,-,*,/,** INTEGER REAL DOUBLE P.

- Ej:

Sean:

a=8.0 b=4.0 c=3.0 (Reales)

i=8 j=4 k=3 (Enteros)

El resultado de:

- a/b 2.0 (real)
- j/a 0.5 (real)
- i/j 2 (entero)
- b/c 1.33333... (real)
- a/c 2.66667 (real)
- j/k 1 (entero)
- j/i 0 (entero)
- j/c 1.33333... (real)

- Dado que la división entera puede producir resultados inesperados, los enteros deberían ser usados únicamente para cosas que son enteras intrínsecamente por naturaleza, como los contadores y los índices.
- Debido a la longitud de palabra finita de un computador, algunos números reales no pueden representarse exactamente. Por ejemplo, la representación de 1./3. puede ser 0.333333 y, como resultado, algunas cantidades que son teóricamente iguales no lo son al ser evaluadas en un computador: 3.*(1./3.)≠ 1.

1.16 Asignación aritmética

- Una sentencia de asignación aritmética asigna el valor de una expresión aritmética a una variable o un elemento de una matriz.
- El operador de asignación en Fortran es el símbolo “=“.

variable = expresión_aritmética

- El funcionamiento es:
    - Se evalúa la expresión_aritmética.
    - Se asigna el valor obtenido a la *variable*.
    - Si el tipo de la *variable* es diferente de la *expresión_aritmética*, se produce una conversión de tipo: *expresión_aritmética* es convertida al tipo de *variable* antes de ser asignada a *variable*.
- Pueden producirse problemas de truncamiento.
- Ej:

Si i es una variable entera:

i = 3./2. ! i se le asigna el valor 1

i=i+1 ! Incrementa en una unidad el valor de i

1.17 Funciones intrínsecas Fortran

- Hasta ahora hemos definido los operadores aritméticos Fortran. Pero, ¿cómo se codifican las funciones trigonométricas o el logaritmo de un número, su exponencial, raíz cuadrada, valor absoluto, etc.? y funciones más complicadas como las funciones hiperbólicas, funciones de Bessel etc.?
- Fortran incorpora todas las funciones matemáticas (y de otros tipos) en unas librerías a disposición del programador. Las funciones de esas librerías se denominan funciones *intrínsecas* del lenguaje. Otro tipo de funciones llamadas *funciones externas e internas* pueden crearse por el propio programador para resolver problemas específicos.
- Cada compilador Fortran viene provisto de las funciones intrínsecas y, en general, pueden verse también en los apéndices de cualquier libro de texto Fortran.
- En matemáticas, una *función* es una expresión que acepta uno o más valores de entrada y calcula un resultado único a partir de ellos. De la misma forma ocurre en Fortran para cualquier función.
- La sintaxis general de una función intrínseca Fortran es:

NOMBRE (lista de argumentos)

- NOMBRE es el nombre reservado para la función intrínseca Fortran.

- *Lista de argumentos* es una lista de variables, constantes, expresiones, o incluso los resultados de otras funciones, separadas por comas, en número y tipo fijado para cada función intrínseca.
- El resultado de la evaluación de la función en su lista de argumentos es de un tipo también fijado para cada función intrínseca.
- Hay dos tipos de funciones intrínsecas: las *genéricas* que pueden usar más de un tipo de datos de entrada y las *específicas* que sólo admiten un tipo de datos de entrada.
- Las funciones Fortran se usan escribiendo su nombre en una expresión y en cualquier caso, estarán a la derecha de una sentencia de asignación.
- Cuando aparece una función en una sentencia Fortran, los argumentos de la misma son pasados a una rutina separada que computa el resultado de la función y lo coloca en lugar de su nombre en la sentencia dada.

EJERCICIOS RESUELTOS

Objetivos:

En este capítulo se construyen los primeros programas Fortran. Para ayudar al usuario a comprender los diversos conceptos, se introducen aquí las instrucciones de entrada/salida (READ/WRITE), si bien la teoría de las mismas se explica en el Capítulo 7.

Básicamente, se aprende a codificar fórmulas en este lenguaje de programación. Para ello, se usan algunas *funciones intrínsecas* matemáticas.

De cada programa mostramos su código Fortran, qué conseguimos al ejecutar el programa; alguna aclaración sobre su contenido y propuestas para conseguir ligeras modificaciones en su ejecución, teniendo en cuenta la teoría vista en el capítulo.

1. Escribir por pantalla el mensaje: HOLA A TODOS.

```
PROGRAM cap1_1

WRITE (*,*) 'HOLA A TODOS'
STOP 'FIN DE PROGRAMA'
END PROGRAM cap1_1
```

- ¿Es imprescindible la primera línea de código del programa y la penúltima?
- Comenta ambas líneas del código y analiza sus efectos.
- ¿Cómo modificarías el programa para escribir cada palabra del mensaje en una línea distinta?
- ¿Y para dejar una línea en blanco?

2. Escribir por pantalla ejemplos de los diferentes tipos de variables Fortran 90/95.

```
PROGRAM cap1_2

INTEGER :: a
REAL :: b1,b2,sueldo_del_ultimo_mes
LOGICAL :: d1,d2
COMPLEX :: e
CHARACTER (LEN=18) :: pal
CHARACTER (LEN=180) :: frase_larga
a=-123
b1=-2.98
b2=0.9E-8
sueldo_del_ultimo_mes=2850.75
d1=.true.
d2=.false.
e=(2.3,3.7)
pal='CONSTANTE CARACTER'
frase_larga='"CONSTANTE CARACTER dividida en dos lineas usando &
&el caracter & al final de la primera y al principio de la siguiente"'
WRITE (*,*) 'CONSTANTE ENTERA',a
WRITE (*,*) 'CONSTANTES REALES (NOTAC NORMAL Y EXPON)',b1,b2
WRITE (*,*) 'IDENTIFICADOR DE VARIABLE REAL (MAX. 31 letras)',&
   sueldo_del_ultimo_mes,'EUROS'
```

```
WRITE (*,*) 'CONSTANTES LOGICAS',d1,d2
WRITE (*,*) 'CONSTANTE COMPLEJA',e
WRITE (*,*) pal !OBSERVAR QUE NO SALEN LOS APOSTROFES
WRITE (*,*) frase_larga !AQUI SI SALEN LAS COMILLAS DOBLES
END PROGRAM cap1_2
```

– El programa presenta declaraciones de todos los tipos de variables Fortran (ver Tabla 1.2). Se usan sentencias de asignación para asignar valores a esas variables, mostrándolos a continuación por monitor.

– Las sentencias demasiado largas se han dividido en dos líneas usando el operador & como marca de continuación de la línea anterior (ver sección 1.7).

3. Calcular la potencia de un número entero, leídos ambos previamente por teclado, y escribir el resultado por pantalla.

```
PROGRAM cap1_3

IMPLICIT NONE
INTEGER :: l,m,resul
PRINT*,'TECLEA 2 NUMEROS ENTEROS SEPARADOS CON INTRO'
READ*,l,m
resul=l**m
PRINT*,'EL RESULTADO ES:'
PRINT*,resul
STOP
END PROGRAM cap1_3
```

– La tercera línea del programa incluye la declaración de variables que usamos en este programa. Notar que para escribir el contenido de la variable RESUL por pantalla se eliminan los apóstrofes en la sentencia PRINT*.

– ¿Cómo cambia el programa si queremos operar con dos números reales?

– Ejecuta el programa para dos números grandes. ¿Qué tipos de variables debes usar para que funcione el programa?

4. Calcular el valor de la expresión $3x^2 + 2y - \frac{z}{4}$ para tres números reales x, y, z leídos por teclado y escribir el resultado por pantalla.

```
PROGRAM cap1_4

REAL :: x,y,z,resul
WRITE(*,*) 'DAME 3 NUMEROS REALES'
READ (*,*) x,y,z
resul=3*x**2+2*y-z/4
WRITE(*,*) 'EL RESULTADO ES:',resul
END PROGRAM cap1_4
```

– Repite el ejercicio eliminando la declaración de variables, es decir, la segunda línea de código. Revisa el apartado de la declaración implícita de variables.

– ¿Qué ocurre si defines las variables de tipo INTEGER? Revisa los apartados de la aritmética con tipos mezclados y la asignación.

5. Resuelve la ecuación de segundo grado $Ax^2 + Bx + C = 0$ únicamente para soluciones reales. Lee los coeficientes A, B y C por teclado y escribe las dos soluciones por pantalla.

```
PROGRAM cap1_5
IMPLICIT NONE
REAL :: a,b,c,x1,x2
WRITE(*,*) 'DAME A B Y C'
READ(*,*) a,b,c
x1=(-b+SQRT(b**2-4*a*c))/(2*a)
x2=(-b-SQRT(b**2-4*a*c))/(2*a)
WRITE(*,*) 'EL RESULTADO ES:',x1,x2
END PROGRAM cap1_5
```

– SQRT(*arg*) es una función intrínseca Fortran que permite calcular la raíz cuadrada del argumento escrito entre paréntesis. El argumento no puede ser de tipo INTEGER pero si REAL. El resultado es de tipo REAL.

– Comprueba que se obtiene el mismo resultado calculando la raíz cuadrada como el radicando elevado a un medio. ¿Qué operaciones se realizan cuando se quita el paréntesis en (2*A)? ¿Y si se quita en (-B-SQRT(B**2-4*A*C))?

– Ejecuta el programa para A=1, B=1 y C=1. ¿Qué tipo de error se obtiene? Reflexiona sobre la importancia de testear el funcionamiento de un programa para todos los posibles juegos de valores de prueba.

6. Codifica la expresión $\frac{sen^2(x)+\cos^2(x)}{\pi}$ para un ángulo x en radianes leído por teclado. Escribe el resultado por monitor.

```
PROGRAM cap1_6

IMPLICIT NONE
REAL :: x,pepe
REAL, PARAMETER :: pi=3.14159
WRITE(*,*) 'DAME EL VALOR DEL ANGULO EN RADIANES'
READ(*,*) x
pepe=(SIN(x)**2+COS(x)**2)/pi
WRITE(*,*) 'EL RESULTADO ES',pepe
END PROGRAM cap1_6
```

– Aquí se usan las funciones intrínsecas seno SIN(*arg*) y coseno COS(*arg*). Para ambas, el argumento (ángulo) debe ser dado en radianes y puede ser REAL. El resultado es del mismo tipo que su argumento.

– Cambia el programa para que funcione con ángulos dados en grados sexagesimales.

– ¿Qué ocurre si escribimos la sentencia de asignación siguiente debajo de la de lectura de *x*: PI=3.14?

EJERCICIOS PROPUESTOS

1) Programa que muestre 'FORTRAN' y 'FORmula TRANslation' en líneas diferentes.

2) Programa que pida por teclado una longitud expresada en pulgadas y la muestre convertida en centímetros (1" = 2.54 cm).

3) Programa que pida una cantidad en pesetas y la muestre en € euros.

4) Programa que acepte por teclado el diámetro de una circunferencia, muestre la longitud de la circunferencia ($L=2\pi R$) y el área del círculo encerrado ($A=\pi R^2$).

5) Programa que pida por teclado el radio de la base y la altura de un cono. Calcula y escribe el valor de su volumen ($V=\frac{1}{3}\pi R^2 H$).

6) Programa que pida por teclado el lado de un cuadrado. Calcula y escribe los valores de su área ($A=L^2$) y diagonal (Teor. Pitágoras).

7) Programa que pida el radio de una esfera y calcule y muestre su superficie ($S=4\pi R^2$) y volumen ($V=\frac{4}{3}\pi R^3$).

8) Programa que pida los dos catetos de un triángulo rectángulo y muestre su hipotenusa, el área del rectángulo cuyos lados son los dos catetos y los dos ángulos del triángulo expresados en grados sexagesimales.

9) Programa que pida coordenadas polares de un punto (r,alfa) y muestre sus coordenadas cartesianas o rectangulares (x,y).

10) Programa que pida coordenadas cartesianas de los extremos de dos vectores origen y calcule el vector suma, mostrando sus componentes, módulo y ángulo en grados.

11) Programa que pida una cantidad en euros y muestre la cantidad de monedas de cada tipo (2 euros, 1 euro, 50 céntimos, 20 céntimos, 10 céntimos, 5 céntimos, 2 céntimos y 1 céntimos).

2 ESTRUCTURAS DE CONTROL CONDICIONALES

- Hasta ahora, todas las sentencias que forman los programas se ejecutan. Sin embargo, hay ocasiones en que un determinado conjunto de sentencias se deben ejecutar sólo si una determinada condición es cierta y sino no.
- Los valores lógicos: constantes, variables y expresiones lógicas, permiten *controlar* la ejecución de las sentencias de un programa.
- Hay dos tipos de expresiones lógicas: las expresiones lógicas relacionales y las expresiones lógicas combinacionales.

2.1 Expresiones lógicas relacionales

- Las expresiones lógicas relacionales comparan los valores de dos expresiones aritméticas o dos expresiones de tipo carácter.
- La evaluación de una expresión lógica relacional produce un resultado de tipo lógico: .TRUE. o .FALSE.
- La sintaxis de una expresión lógica de tipo relacional es:

operando1 OPERADOR_LÓGICO_RELACIONAL operando2

  - *operando* es una expresión, variable o constante aritmética o de tipo carácter.
  - OPERADOR_LÓGICO_RELACIONAL puede ser:

OPERADOR LÓGICO RELACIONAL		SIGNIFICADO
F77	F90/95	
.EQ.	==	IGUAL
.NE.	/=	DISTINTO
.LT.	<	MENOR QUE
.LE.	<=	MENOR O IGUAL QUE
.GT.	>	MAYOR QUE
.GE.	>=	MAYOR O IGUAL QUE

Tabla 2.1: Operadores lógicos relacionales Fortran

- Los operadores lógicos relacionales de Fortran 77 han sobrevivido y funcionan en los compiladores actuales de Fortran 90/95. Por lo tanto, es interesante que el programador sepa reconocerlos en los programas, sin embargo, es preferible usar la forma de Fortran 90 en sus programas nuevos, que además, es mucho más intuitiva.

2.2 Ejemplos de expresiones lógicas relacionales

- Sea la sentencia de declaración de tipos:

INTEGER :: i=3,j=5

OPERACIÓN	RESULTADO
3<=i	.TRUE.
j**2-1>=0	.TRUE.
i==j	.FALSE.
i/=10	.TRUE.
'ANA'<'PEPE'	.TRUE.

Tabla 2.2: Ejemplos de expresiones lógicas relacionales

- La última expresión lógica relacional de la Tabla 2.2 es cierta porque los caracteres se evalúan en orden alfabético. Más detalles en el capítulo 6.
- Son inválidas las siguientes expresiones lógicas relacionales:

<=5 Falta operando1

8.44/='XYZ' No se pueden comparar reales con caracteres.

i=3 No es una expresión lógica relacional, sino una sentencia de asignación! No confundir el operador lógico relacional de igualdad con el operador de asignación.

2.3 Expresiones lógicas combinacionales

- La evaluación de una expresión lógica combinacional produce un resultado de tipo lógico: .TRUE. o .FALSE.
- La sintaxis de una expresión lógica de tipo combinacional es:

operando1 OPERADOR_LÓGICO_COMBINACIONAL operando2

  - *operando* es una expresión relacional, variable lógica o constante lógica. *Operando1* no existe cuando el operador lógico usado es unario.
  - OPERADOR_LÓGICO_COMBINACIONAL puede ser:

OPERADOR	TIPO	SIGNIFICADO
.NOT.	UNARIO	ES LO OPUESTO A *OPERANDO2*
.AND.	BINARIO	ES .TRUE. SI Y SÓLO SI *OPERANDO1* Y *OPERANDO2* SON .TRUE.

.OR.	BINARIO	ES .TRUE. SI UNO DE LOS DOS OPERANDOS O LOS DOS ES .TRUE.
.EQV.	BINARIO	ES .TRUE. SI Y SÓLO SI LOS DOS OPERANDOS SON .TRUE. O LOS DOS OPERANDOS SON .FALSE.
.NEQV.	BINARIO	ES .TRUE. SI Y SÓLO SI LOS DOS OPERANDOS TIENEN VALORES DISTINTOS

Tabla 2.3: Operadores lógicos combinacionales Fortran 90/95

Sean *operando1* y *operando2* A y B, respectivamente, la tabla de verdad de los operadores lógicos combinacionales es:

A	B	.NOT.B	A.AND.B	A.OR.B	A.EQV.B	A.NEQV.B
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.

Tabla 2.4: Tabla de verdad de los operadores lógicos combinacionales

2.4 Precedencias lógicas-aritméticas

- Precedencia de los operadores lógicos combinacionales:

OPERADOR(ES)	PRECEDENCIA
( )	MAYOR
.NOT.	
.AND.	
.OR.	
.EQV. , .NEQV.	MENOR

Tabla 2.5: Orden de precedencia de operadores lógicos combinacionales Fortran

- Si una expresión lógica contiene dos o más operadores de la misma precedencia se siguen las siguientes reglas:
    - Cuando existen paréntesis anidados se evalúan desde el más interno hasta el más externo.

- Si existen varios operadores .EQV. y/o .NEQV. se evalúan de izquierda a derecha.

- Precedencia de los operadores aritméticos, lógicos relacionales y lógicos combinacionales:

OPERACIÓN	PRIORIDAD
OPERADORES ARITMÉTICOS	
**	1
*, /	2
+ , -	3
OPERADORES LÓGICOS RELACIONALES	
> , >= , < <= , == ,.NE.	4
OPERADORES LÓGICOS COMBINACIONALES	
.NOT.	5
.AND.	6
.OR.	7
.EQV.,.NEQV.	8

Tabla 2.6: Orden de precedencia de operadores Fortran

2.5 Sentencia de asignación lógica

- Asigna un valor lógico (.TRUE. o .FALSE.) a una variable lógica.

variable_lógica = expresión_lógica

  - *variable_lógica* es el nombre de una variable, o elemento de matriz, declarada en el programa del tipo LOGICAL.
  - *expresión_lógica* es una expresión lógica Fortran válida.

- El funcionamiento es:
  - Se evalúa la expresión_lógica.
  - Se asigna el valor obtenido a la *variable_lógica*.

- Ej:

```
LOGICAL :: log1
INTEGER :: i = 10
log1 = (i==10)
```

2.6 Bloque IF

- Permite que un bloque de sentencias (puede ser sólo una) sea ejecutado si y sólo si el valor de una expresión lógica es cierta. Si la expresión lógica es falsa se salta ese bloque de sentencias y se ejecuta la primera sentencia ejecutable por debajo de END IF.

```
IF (expresión lógica) THEN
   bloque de sentencias
END IF
```

- ENDIF marca la terminación de la sentencia bloque IF.
- El *bloque de sentencias* suele dentarse por dos o más espacios para facilitar la lectura del bloque IF, aunque no es obligatorio hacerlo.
- La estructura del bloque IF puede ser más complicada. A veces, se quiere ejecutar un bloque de sentencias si una expresión lógica es cierta y diferentes bloques de sentencias si otras condiciones son ciertas. Por ejemplo:

```
IF (expresión lógica 1) THEN
   bloque de sentencias 1
ELSE IF (expresión lógica 2) THEN
   bloque de sentencias 2]
ELSE
   bloque de sentencias 3
END IF
```

- Si la *expresión lógica* 1 es cierta se ejecuta el *bloque de sentencias* 1 y se salta el resto de bloques hasta la primera sentencia ejecutable por debajo de END IF.
- Si la *expresión lógica 1* es falsa se salta el *bloque de sentencias* 1, se evalúa la *expresión lógica* 2 y si es cierta se ejecuta el *bloque de sentencias* 2 y se salta hasta la primera sentencia ejecutable por debajo de END IF.
- Si ambas expresiones lógicas son falsas, el programa ejecuta el *bloque de sentencias* 3.
- En general, un bloque IF puede contener opcionalmente cualquier número de subbloques ELSE IF (0, 1, 2, 3,...), pero sólo puede contener un subbloque ELSE.
- La sintaxis general de un bloque IF es:

```
IF (expresión lógica 1) THEN
   bloque de sentencias 1
[ELSE IF (expresión lógica 2) THEN
   bloque de sentencias 2]
```

```
...
[ELSE
   bloque de sentencias n]
END IF
```

- La expresión lógica de una cláusula es evaluada sólo si las expresiones lógicas de todas las cláusulas precedentes han sido falsas. Cuando la prueba de una expresión lógica es cierta se ejecuta el bloque de sentencias correspondiente y se salta a la primera sentencia ejecutable por debajo de END IF.
- Cuando el bloque IF no incluye la cláusula ELSE, puede ocurrir que ninguna de las expresiones lógicas sean ciertas y que, por lo tanto, no se ejecute ninguno de los bloques de sentencias dados.

2.7 Bloque IF con nombre

- Es posible asignar un nombre a un bloque IF.
- La sintaxis general de un bloque IF con nombre es:

```
[nombre:] IF (expresión lógica 1) THEN
   bloque de sentencias 1
[ELSE IF (expresión lógica 2) THEN [nombre]
   bloque de sentencias 2]
...
[ELSE [nombre]
   bloque de sentencias n]
END IF [nombre]
```

- *nombre* es cualquier identificador válido.
- Es recomendable usar nombres en los bloque IF largos y complicados. Por un lado, el programador estructura mejor los programas y, por otro, el compilador encuentra errores en su código de forma más precisa.
- Además, los bloques IF pueden estar *anidados*. Dos bloques IF se dice que están anidados cuando uno de ellos se encuentra dentro de otro. En este caso, cada uno de los bloques IF requerirá su propia sentencia ENDIF.

```
[nombre_externo:] IF (expresión lógica 1) THEN
   bloque de sentencias 1
   [nombre_interno:]IF (expresión lógica 2) THEN
   bloque de sentencias 2
   END IF [nombre_interno]
END IF [nombre_externo]
```

- La expresión lógica 2 sólo se evalúa cuando la expresión lógica 1 ha sido cierta.

2.8 Ejemplos de bloques IF

```
IF (num == 0) THEN
   cont0=cont0+1
   ELSE IF (num>0) THEN
   contp=contp+1
ELSE
   contn=cotn+1
END IF


IF (x<0) THEN
   y=SQRT(ABS(x))
   z=x+1-y
ELSE
   y=SQRT(x)
   z=x+1-y
END IF
```

2.9 IF lógico

- La sentencia IF lógica es el caso particular más simple del bloque IF. La sentencia IF lógica evalúa una expresión lógica y ejecuta una sentencia si dicha expresión es cierta.

```
IF (expresión lógica) sentencia
```

- *sentencia* es cualquier sentencia ejecutable excepto DO, END, bloque IF u otra sentencia IF lógica.
- Si el valor de la expresión lógica es .TRUE., se ejecuta la sentencia. Si es .FALSE. se salta esa sentencia y se pasa el control a la sentencia siguiente.
- Ej:

```
res=0
IF (a==b) res=a+b
res=res+10
```

2.10 Bloque SELECT CASE

- El bloque SELECT CASE aparece en Fortran 90/95 como otra forma de controlar la ejecución de determinados bloques de sentencias junto con el bloque IF.
- Su sintaxis general es:

```
[nombre:] SELECT CASE (expresión caso)
CASE (selector de caso 1) [nombre]
  bloque de sentencias 1
[CASE (selector de caso 2) [nombre]
  bloque de sentencias 2]
…
[CASE DEFAULT [nombre]
  bloque de sentencias n]
END SELECT [nombre]
```

- El bloque SELECT CASE ejecuta un bloque determinado de sentencias cuando el valor de la *expresión caso* coincide o pertenece al rango dado de su correspondiente *selector de caso*.
- Opcionalmente puede existir un CASE DEFAULT en un bloque SELECT CASE. El bloque de sentencias de este *caso por defecto* se ejecuta cuando el valor de la *expresión caso* no coincide con ningún *selector de caso*.
- *expresión caso* es una expresión entera, lógica o de caracteres.
- *Selector de caso* es una lista de uno o más valores posibles del mismo tipo que la *expresión caso*. Cada selector de caso debe ser mutuamente excluyente. Los valores pueden escribirse como:
  - valor
  - valormin: valormax
  - : valormax
  - valormin:
  - o una combinación de las formas anteriores separadas por comas.
- Es recomendable poner nombre a un bloque SELECT CASE largo y complicado. Por un lado, el programador estructura mejor los programas y, por otro, el compilador encuentra errores en su código de forma más precisa.

2.11 Ejemplos de bloque SELECT CASE

- Determinar si un número entero entre 1 y 10 es par o impar y visualizar un mensaje adecuado en consecuencia.

```
INTEGER :: valor
parimpar: SELECT CASE (valor)
CASE (1, 3, 5, 7, 9)
  WRITE (*,*) 'el valor es impar'
CASE (2, 4, 6, 8, 10)
  WRITE (*,*) 'el valor es par'
CASE (11:)
  WRITE (*,*) 'el valor es demasiado grande'
CASE DEFAULT
  WRITE (*,*) 'el valor es negativo o cero'
END SELECT parimpar
```

- Visualizar un mensaje de acuerdo con el valor de la temperatura dada.

```
INTEGER :: temp
friocalor: SELECT CASE (temp)
CASE (:-1)
  WRITE (*,*) 'por debajo de 0 Celsius'
CASE (0)
  WRITE (*,*) 'Está helando'
CASE (1:10)
  WRITE (*,*) 'Hace frío'
CASE (11:20)
  WRITE (*,*) 'templado'
CASE (21:)
  WRITE (*,*) 'Hace calor'
END SELECT friocalor
```

EJERCICIOS RESUELTOS

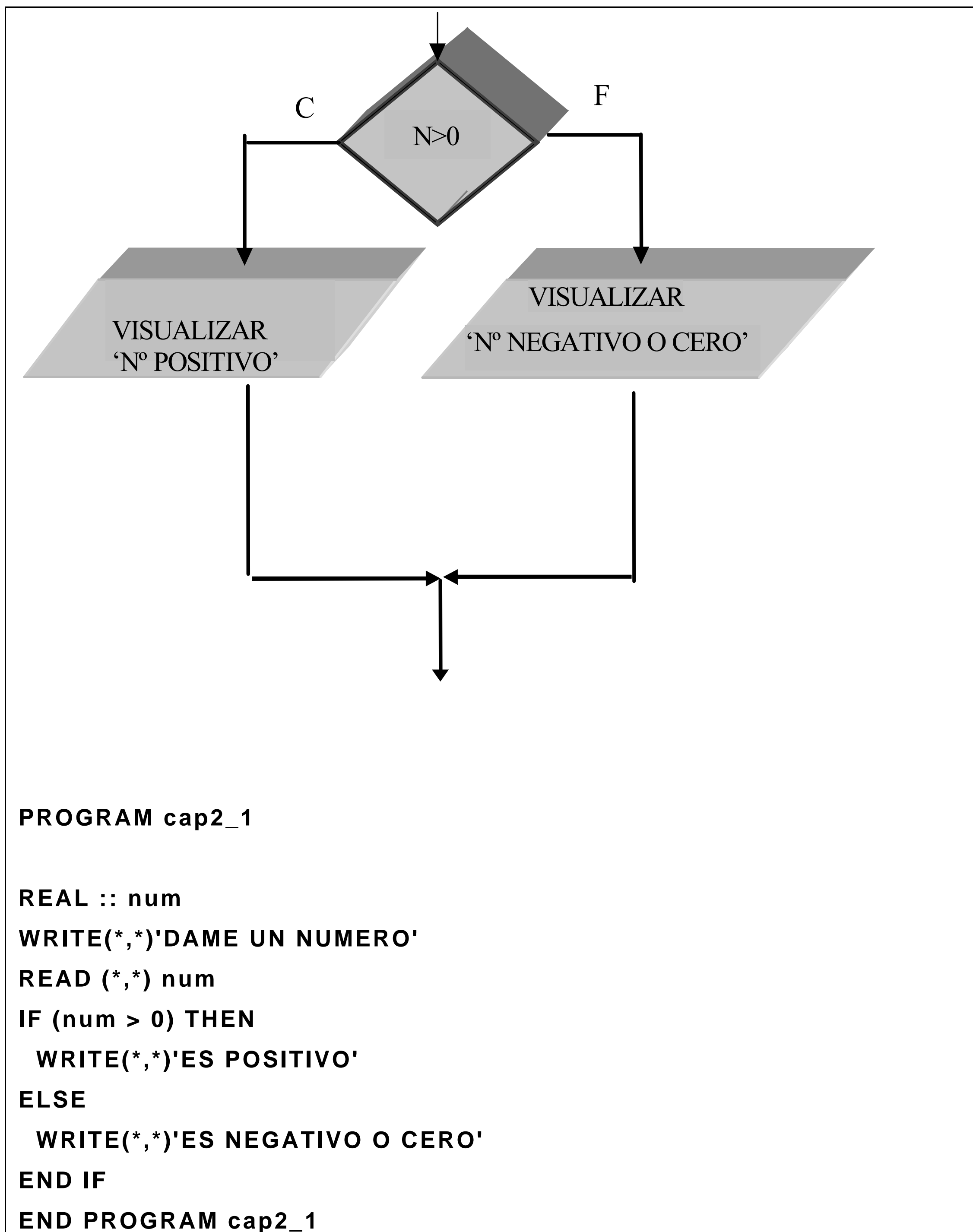
Objetivos:

Aprender a ejecutar unas instrucciones u otras dependiendo de que se cumplan o no una serie de condiciones. Es decir, manejar los bloques IF simples y con anidamiento y los bloques SELECT CASE.

Antes de abordar los primeros programas se muestran los organigramas que representan los algoritmos de los problemas planteados.

Los problemas planteados usan expresiones lógicas en las que intervienen operadores lógicos relacionales y/o combinacionales.

1. Pedir un número real por teclado y escribir si es positivo o no.

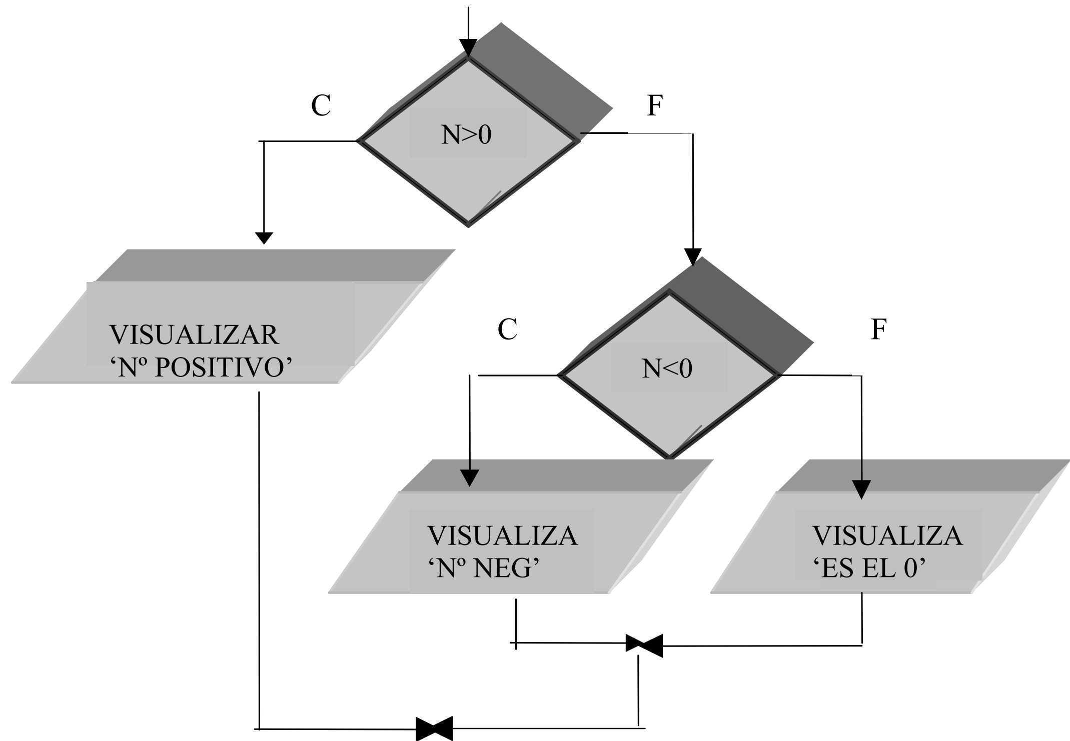



```
PROGRAM cap2_1

REAL :: num
WRITE(*,*)'DAME UN NUMERO'
READ (*,*) num
IF (num > 0) THEN
  WRITE(*,*)'ES POSITIVO'
ELSE
  WRITE(*,*)'ES NEGATIVO O CERO'
END IF
END PROGRAM cap2_1
```

– Este ejercicio requiere un bloque IF simple.

2. Pedir un número real por teclado y escribir si es positivo, negativo o cero.




```
PROGRAM cap2_2


REAL :: num
WRITE (*,*) 'DAME UN NUMERO'
READ (*,*) num
IF (num > 0) THEN
  WRITE (*,*) 'ES POSITIVO'
ELSE IF (num < 0) THEN
    WRITE (*,*) 'ES NEGATIVO'
ELSE
    WRITE (*,*) 'ES EL 0'
END IF
END PROGRAM cap2_2
```

3. Resolver una ecuación de 2º grado. Suponer que es efectivamente de 2º grado (A≠0).

```
PROGRAM cap2_3
IMPLICIT NONE
REAL :: a,b,c,d,r1,r2,pr,pi
WRITE (*,*) 'DAME A, B, C, CON A DISTINTO DE CERO'
READ (*,*) a,b,c
d=b**2-4*a*c
discriminante: IF (d == 0) THEN
  r1=-b/(2*a)
  r2=r1
  WRITE (*,*)'SOLUC. REALES DOBLES, R1=R2=',r1,r2
ELSE IF (d > 0) THEN
  r1=(-b+SQRT(d))/(2*a)
  r2=(-b-SQRT(d))/(2*a)
  WRITE (*,*)'LAS RAICES DEL POLINOMIO SON:'
  WRITE (*,*)'R1=',r1,' R2=',r2
ELSE
  pr=-b/(2*a)
  pi=SQRT(ABS(d))/(2*a)
  WRITE (*,*)'SOLUCIONES COMPLEJAS'
  WRITE (*,*)'PARTE REAL:',pr
  WRITE (*,*)'PARTE IMAGINARIA:',pi
END IF discriminante
END PROGRAM cap2_3
```

- ABS(argumento) es una función intrínseca que devuelve el valor absoluto del argumento escrito entre paréntesis. Su tipo puede ser entero o real.
- Repite el ejercicio cambiando el orden de evaluación de las expresiones lógicas relacionales.

4. Resolver una ecuación de 2º grado con A, B y C cualquier valor.

```
PROGRAM cap2_4
IMPLICIT NONE
REAL :: a,b,c,d,r1,r2,pr,pi,r
```

```
WRITE (*,*) 'INTRODUCE A,B,C, CON A, B, C CUALQUIER VALOR'
READ (*,*) a,b,c
d=b**2-4*a*c

ec2: IF (a /= 0) THEN

  discriminante: IF (d == 0) THEN
    r1=-b/(2*a)
    r2=r1
    WRITE (*,*) 'SOLUC. REALES DOBLES, R1=R2=',r1,r2
  ELSE IF(d > 0) THEN discriminante
      r1=(-b+SQRT(d))/(2*a)
      r2=(-b-SQRT(d))/(2*a)
      WRITE (*,*) 'LAS RAICES DEL POLINOMIO SON:'
      WRITE (*,*) 'R1=',r1,'R2=',r2
  ELSE discriminante
      pr=-b/(2*a)
      pi=SQRT(ABS(d))/(2*a)
      WRITE (*,*) 'SOLUCIONES COMPLEJAS'
      WRITE (*,*) 'PARTE REAL:',pr
      WRITE (*,*) 'PARTE IMAGINARIA:',pi
  END IF discriminante

ELSE IF (b /= 0) THEN ec2
    r=-c/b
    WRITE (*,*) 'SOLUCION UNICA, R=',r
ELSE IF (c /= 0) THEN ec2
    WRITE (*,*) 'ECUACION IMPOSIBLE'
    ELSE
    WRITE (*,*) 'ECUACION INDETERMINADA'
END IF ec2
END PROGRAM cap2_4
```

5. Dada una calificación numérica obtener la correspondiente alfabética según la siguiente clasificación:

$0 \leq$ Nota < 5	Suspenso
$5 \leq$ Nota < 7	Aprobado

7≤ Nota < 9	Notable
9≤ Nota < 10	Sobresaliente
Nota = 10	Matricula de Honor

```
PROGRAM cap2_5

REAL :: nota
WRITE (*,*) 'DAME UNA NOTA'
READ (*,*) nota
IF (nota < 0 .OR. nota > 10) THEN
  WRITE (*,*) 'NOTA INCORRECTA'
ELSE IF (nota < 5) THEN
  WRITE (*,*) 'SUSPENSO'
ELSE IF (nota < 7) THEN
  WRITE (*,*) 'APROBADO'
ELSE IF (nota < 9) THEN
  WRITE (*,*) 'NOTABLE'
ELSE IF (nota < 10) THEN
  WRITE (*,*) 'SOBRESALIENTEE'
ELSE
  WRITE (*,*) 'MATRICULA DE HONOR'
END IF
END PROGRAM cap2_5
```

6. Realizar una operación entre cuatro posibles según se desee. El programa pide dos números y una operación visualizando el resultado de la misma.

```
PROGRAM cap2_6

REAL :: a,b,c
CHARACTER (LEN=1) :: oper
WRITE (*,*) 'DAME 2 NUMEROS '
READ (*,*)  a, b
WRITE (*,*) 'DAME UNA OPERACION (+, -, *, /)'
READ (*,*) oper
IF (oper == '+') THEN
```

```
  c=a+b
  WRITE (*,*) 'C=',c
ELSE IF (oper == '-') THEN
  c=a-b
  WRITE (*,*) 'C=',c
ELSE IF (oper == '*') THEN
  c=a*b
  WRITE (*,*) 'C=',c
ELSE IF (oper == '/' .AND. b /= 0) THEN
  c=a/b
  WRITE (*,*) 'C=',c
ELSE IF (oper == '/' .AND. b == 0) THEN
  WRITE (*,*) 'NO SE PUEDE DIVIDIR POR CERO'
ELSE
  WRITE (*,*) oper,' NO ES UNA OPERACION VALIDA'
END IF
END PROGRAM cap2_6
```

– En este ejercicio, OPER es el nombre que damos a la variable donde almacenamos el símbolo aritmético de la operación. La última expresión lógica evaluada requiere del operador lógico combinacional Y para evitar que el programa pueda dividir por cero y provoque un error en tiempo de ejecución.

7. Comprobar si un número leído por teclado pertenece al intervalo cerrado [0-10] o no. Escribir un mensaje por pantalla que informe de ello.

```
PROGRAM cap2_7

REAL :: num
WRITE (*,*) 'DAME UN NUMERO'
READ (*,*) num
IF (num >= 0 .AND. num <= 10) THEN
  WRITE (*,*) num,' PERTENECE AL INTERVALO [0-10]'
ELSE
  WRITE (*,*) num,' NO PERTENECE AL INTERVALO [0-10]'
END IF
END PROGRAM cap2_7
```

8. Presentar un menú en pantalla con diferentes opciones.

```
PROGRAM cap2_8

INTEGER :: num
WRITE (*,*) 'DAME UNA OPCION'
WRITE (*,*) '1 PARA EJECUTAR BLOQUE 1'
WRITE (*,*) '2 PARA EJECUTAR BLOQUE 2'
WRITE (*,*) '3 PARA SALIR'
READ (*,*) num
SELECT CASE (num)
CASE (1)
WRITE (*,*) 'SE HACE BLOQUE 1'
CASE (2)
WRITE (*,*) 'SE HACE BLOQUE 2'
CASE (3)
WRITE (*,*) 'ADIOS'
CASE DEFAULT
WRITE (*,*) 'OPCION INCORRECTA'
END SELECT
END PROGRAM cap2_8
```

– Cuando el valor de la variable entera num coincide con algún valor de CASE, el control del programa pasa a ejecutar el bloque de sentencias correspondiente. Si el valor de num no coincide con ningún valor de CASE, se ejecuta el CASE DEFAULT visualizando el mensaje “OPCION INCORRECTA”.

– Repite el ejercicio usando la estructura condicional IF y compara con la estructura SELECT CASE.

EJERCICIOS PROPUESTOS

1) Programa que acepte el número del año, y muestre "PRESENTE" si el número es el año actual, "PASADO" si es menor o "FUTURO" si es mayor.

2) Programa que pida un número natural y muestre si es par o impar.

3) Programa que pida una temperatura y su escala (Celsius/Fahrenheit) y muestre su valor en la otra escala (0 C=32 F y 100 C=212 F). Ecuación de conversión: C= $\frac{5}{9}(F-32)$

4) Programa que acepte el número de un año e indique si es bisiesto o no. Un año es bisiesto si es múltiplo de cuatro, pero no de cien. Excepción a la regla anterior son los múltiplos de cuatrocientos que siempre son bisiestos. Son bisiestos: 1600,1996, 2000, 2004. No son bisiestos: 1700, 1800, 1900, 1998, 2002.

5) Programa que pida la longitud de los lados de un triángulo, compruebe si los datos son correctos, muestre si es equilátero, isósceles o escaleno, y el valor de sus ángulos en grados. A saber: los lados de un triángulo son correctos si cada uno de ellos es menor que la suma de los otros dos. Un triángulo es equilátero si sus tres lados son iguales, isósceles si dos lados son iguales y escaleno si sus 3 lados son distintos. Teorema del coseno $a^2=b^2+c^2-2bc\cos(b,c)$.

6) Programa que pida coordenadas cartesianas de un punto e indique en qué cuadrante se encuentra dicho punto, en qué eje o si se trata del origen de coordenadas.

3 ESTRUCTURAS DE CONTROL REPETITIVAS. BUCLES

3.1 Estructuras de repetición

- Una estructura de repetición, también llamada lazo o bucle, hace posible la ejecución repetida de secciones específicas de código.
- Hay dos tipos básicos de estructuras de repetición, cuya diferencia principal radica en cómo se controlan las mismas:
    - Repetición controlada por contador o bucle DO iterativo. Con esta estructura, un bloque de sentencias se ejecuta una vez para cada uno de los valores que va tomando un contador. Se ejecuta un número específico de veces, siendo el número de repeticiones conocido antes de que empiece la ejecución de tal bucle.
    - Repetición controlada por expresión lógica o bucle WHILE. En este caso, un bloque de sentencias se ejecuta un número indefinido de veces, hasta que se satisface alguna condición establecida por el usuario, lo cual desde el punto de vista de la programación, equivale a que una cierta expresión lógica tome el valor .TRUE..

3.2 Repetición controlada por contador o bucle DO iterativo

- Esta estructura ejecuta un bloque de sentencias un número específico de veces. Se construye de la siguiente manera:

```
DO indice = inicio, fin [, paso]
  sentencia_1   ┐
  [sentencia_2] ├ Cuerpo
  …             ┘
END DO
```

- *Índice* es una variable entera que se usa como contador del bucle.
- *Inicio, fin y paso* son cantidades enteras y constituyen los parámetros del *índice* del bucle. Controlan los valores de la variable *índice* durante la ejecución. Pueden ser constantes, variables o expresiones enteras.
- Las sentencias entre la sentencia DO y la sentencia END DO se llaman *cuerpo* del bucle. Para mejorar la lectura del código es recomendable endentar el cuerpo del bucle con dos o más espacios.
- *Inicio* marca el valor inicial que toma *índice* cuando comienza la ejecución del bucle DO.
- *Fin* marca el valor final que puede tomar *índice*.

- *Paso* indica el incremento con que *índice* es modificado después de cada ejecución del bucle DO. Es un parámetro opcional, cuando no aparece, su valor por defecto es 1. Puede ser positivo o negativo.
- Cuando se ejecuta una sentencia DO, tienen lugar la secuencia de operaciones siguiente:
    - Si los parámetros del índice del bucle son expresiones enteras se calculan sus valores antes del comienzo de la ejecución del bucle.
    - Se asigna el valor *inicio* a la variable de control *índice*. Si la condición *índice* * *paso* ≤ *fin* * *paso*, el programa ejecuta las sentencias del cuerpo del bucle.
    - Después, se recalcula el valor de *índice* como*: índice = índice + paso*. Si aún se cumple la condición anterior *índice* * *paso* ≤ *fin* * *paso*, el programa ejecuta otra vez las sentencias del cuerpo del bucle.
    - Se repite el punto anterior hasta que deja de cumplirse la condición de desigualdad dada*,* en cuyo caso la ejecución salta a la primera sentencia siguiente a la sentencia END DO.
    - El número de iteraciones que se llevan a cabo en el bucle DO se puede calcular a partir de la siguiente ecuación:

$$n^{o}\, iteraciones = \frac{fin - inicio + paso}{paso}$$ **Ecuación 3-1**

- Precauciones:
    - No está permitido modificar el valor de *índice* en el cuerpo del bucle. La mayoría de los compiladores Fortran suelen reconocer este problema y generan un error en tiempo de compilación si se intenta modificar su valor. Caso de que el compilador no detectara este problema, se podría dar el caso de un bucle infinito.
    - La única manera de parar la ejecución de un programa cuando éste contiene un bucle infinito es *matar* el programa pulsando control +C.
    - En muchos computadores, el valor de *índice* después de la finalización normal del bucle DO, está definido por el siguiente valor asignado como resultado del incremento. Esto es lo que ocurre en el compilador de la Salford FTN95. Sin embargo, su valor es indefinido oficialmente en el Fortran estándar, de modo que algunos compiladores pueden producir resultados diferentes. Por lo tanto, no conviene nunca usar el valor del índice de un bucle en las siguientes sentencias del programa Fortran.
    - Si *paso* toma el valor 0 se puede dar un bucle infinito.
    - Puede ocurrir que el cuerpo de un bucle no se ejecute nunca, si *inicio* * *paso* > *fin* * *paso*.

- Ejemplos de bucles DO iterativos

```
INTEGER :: i
DO i=1,5
  sentencia
END DO
WRITE (*,*) i
```

Aplicando la ecuación 3-1, se obtiene que sentencia se ejecuta 5 veces. La variable índice i toma los valores 1, 2, 3, 4 y 5. A continuación, el control vuelve a la sentencia DO, i toma el valor 6, pero 6*1 > 5*1 y se salta a la sentencia WRITE, escribiéndose 6 por monitor.

```
INTEGER :: i
DO i=5,0,-2
  sentencia
END DO
```

Aplicando la ecuación 3-1, se obtiene que sentencia se ejecuta 3 veces, para los valores de i: 5, 3, y 1. A continuación, el control vuelve a la sentencia DO, i toma el valor -1, pero (-1)*(-2) > 0*(-2) y se salta a la sentencia siguiente a END DO.

- Escribir un mensaje por monitor 10 veces:

```
DO i=1, 10
  WRITE (*,*) ‘HOLA’
END DO
```

En este caso, el índice del bucle se usa únicamente para indicarnos cuántas veces se ejecuta el cuerpo.

- Sumar los pares de 2 a 10 sin leer datos.

```
acum=0
DO k = 2, 10,2
  acum=acum+k
END DO
WRITE (*,*) acum
```

En este caso, el índice del bucle interviene en el cuerpo del mismo.

3.3 Repetición controlada por expresión lógica o bucle WHILE

- En un bucle WHILE, un bloque de sentencias se ejecuta un número indefinido de veces, hasta que se satisface alguna condición

establecida por el usuario, o dicho de otro modo, hasta que una cierta expresión lógica toma el valor .TRUE..

- La sintaxis general de esta estructura es:

```
DO
  sentencia_1
  [sentencia_2]
  …
  IF (expresión_lógica) EXIT
  sentencia_3
  [sentencia_4]
  …
END DO
```

- El bloque de sentencias entre la sentencia DO y la sentencia END DO se ejecuta indefinidamente mientras *expresión_lógica* es falsa. Cuando *expresión_lógica* se hace cierta; entonces, se ejecuta la palabra reservada EXIT cuyo efecto es transferir el control fuera del bucle DO, a la primera sentencia siguiente a END DO.
- Un bucle WHILE puede contener varias sentencias EXIT, generalmente, formando parte una sentencia IF o bloque IF. Sin embargo, no conviene usar más de uno por bucle WHILE, en aras de construir programas bien estructurados. De esta manera, cada bucle WHILE tendrá un único punto de entrada, la sentencia DO, y un único punto de salida, la sentencia EXIT.
- Precauciones: si *expresión lógica* nunca se hace cierta, estaremos en un bucle infinito.
- Ejemplos de bucles WHILE:
    - Se presentan los dos ejemplos vistos en la sección anterior para comparar mejor las estructuras.
    - Escribir un mensaje por monitor 10 veces:

```
i=1
DO
  IF (i>10) EXIT
  WRITE (*,*) ‘HOLA’
  i=i+1
END DO
```

    - Sumar los pares de 2 a 10 sin leer datos.

```
i=2
acum=0
DO
```

```
  IF (i>10) EXIT
  acum=acum+i
  i=i+2
END DO
WRITE (*,*) acum
```

- Leer por teclado un número entre 0 y 10, ambos límites incluidos. El usuario puede equivocarse, en cuyo caso el programa le dará tantas posibilidades como necesite (indefinidas veces) hasta conseguir que el número introducido esté en el rango dado. Entonces, el programa muestra cuantos intentos ha usado.

```
INTEGER :: num,cont=0
DO
  WRITE (*,*)
  WRITE (*,*) "Dame un numero de 0 a 10"
  READ (*,*) num
  cont=cont+1
  IF (num>=0.AND.num<=10) EXIT
  WRITE (*,*) "Has tecleado",num
  WRITE (*,*) "El numero debe estar entre 0 y 10"
  WRITE (*,*) "Vuelve a intentarlo"
END DO
WRITE (*,*) "*****ENHORABUENA*****"
WRITE (*,*) "lo conseguistes en",cont,"veces"
```

3.4 Bucle DO WHILE

- En Fortran 90/95 hay otra forma alternativa de bucle WHILE, el llamado bucle DO WHILE.
- Su sintaxis general es:

```
DO WHILE (expresión_lógica)
  sentencia_1
  [sentencia_2]        Cuerpo
  …
END DO
```

- Esta estructura funciona de la siguiente manera:
  - si *expresión_lógica* es cierta se ejecuta el cuerpo del bucle y el control vuelve a la sentencia DO WHILE.

- si *expresión_lógica* es cierta aún, se ejecuta otra vez el cuerpo del bucle y el control vuelve a la sentencia DO WHILE.
- El proceso anterior se repite hasta que *expresión*_lógica se hace falsa, en cuyo caso el control pasa a ejecutar la primera sentencia siguiente a END DO.
- El bucle DO WHILE es un caso especial del bucle WHILE, en el que el testeo de salida ocurre siempre en el cabecero del bucle.

- Ejemplos de bucles DO WHILE:
    - Nuevamente, se presentan los ejemplos vistos en la sección anterior para comparar mejor las diferentes estructuras.
    - Escribir un mensaje por monitor 10 veces:

```
i=1
DO WHILE (i<=10)
  WRITE (*,*) ‘HOLA’
  i=i+1
END DO
```

- Sumar los pares de 2 a 10 sin leer datos.

```
i=2
acum=0
DO WHILE (i<=10)
  acum=acum+i
  i=i+2
END DO
WRITE (*,*) acum
```

- Leer por teclado un número entre 0 y 10, ambos límites incluidos. Cerciorarse de que el número está en el rango dado sin limitar el número de intentos. Mostrar cuantos intentos ha necesitado el usuario.

```
INTEGER :: num,cont=1
WRITE (*,*) “Dame un numero de 0 a 10”
READ (*,*) num
DO WHILE (num<0.OR.num>10)
  WRITE (*,*) "Has tecleado",num
  WRITE (*,*) "El numero debe estar entre 0 y 10"
  WRITE (*,*) “Vuelve a intentarlo”
  WRITE (*,*) “Dame un numero de 0 a 10”
```

```
  READ (*,*) num
  cont=cont+1
END DO
WRITE (*,*) “*****ENHORABUENA*****”
WRITE (*,*) “lo conseguistes en”,cont,”veces”
```

- En la sección siguiente, se verá que es posible saltar algunas sentencias del cuerpo de un bucle en una iteración y pasar a ejecutar la siguiente iteración, en cualquier momento mientras el bucle se está ejecutando.

3.5 Sentencias EXIT y CYCLE

- En las secciones anteriores se ha visto que la sentencia EXIT produce la salida inmediata de un bucle.
- Otra sentencia que permite controlar la ejecución de un bucle es la sentencia CYCLE.
- CYCLE detiene la ejecución de la iteración actual y devuelve el control a la cabecera del bucle continuando su ejecución en la iteración siguiente.
- Ambas sentencias pueden usarse tanto en bucles WHILE como en bucles iterativos.
- Ejemplo de la sentencia CYCLE.
    - Si en el código Fortran anterior construido con bucle DO queremos mostrar los tres últimos mensajes de error sólo a partir del tercer intento:

```
INTEGER :: num,cont=0
DO
  WRITE (*,*)
  WRITE (*,*) "Dame un numero de 0 a 10"
  READ (*,*) num
  cont=cont+1
  IF (num>=0.AND.num<=10) EXIT
  IF (cont<3) CYCLE
  WRITE (*,*) "Has tecleado",num
  WRITE (*,*) "El numero debe estar entre 0 y 10"
  WRITE (*,*) "Vuelve a intentarlo"
END DO
WRITE (*,*) "*****ENHORABUENA*****"
WRITE (*,*) "lo conseguistes en",cont,"veces"
```

3.6 Bucles con nombre

- Es posible asignar un nombre a un bucle.
- La sintaxis general de un bucle WHILE con un nombre añadido es:

```
[nombre:] DO
  sentencia_1
  [sentencia_2]
  …
[  IF (expresión_lógica) CYCLE [nombre]]
  [sentencia_3]
  …
  IF (expresión_lógica) EXIT [nombre]
  [sentencia_4]
  …
END DO [nombre]
```

- La sintaxis general de un bucle iterativo con un nombre añadido es:

```
[nombre:] DO indice = inicio, fin [, paso]
  sentencia_1
  [sentencia_2]   ← Cuerpo
  …
 [  IF (expresión_lógica) CYCLE [nombre]]
  [sentencia_4]
  …
END DO [nombre]
```

- En ambas estructuras, el nombre del bucle debe ser un identificador válido.
- Es opcional poner nombre a un bucle, pero si se pone, éste debe repetirse en la sentencia END DO.
- Es opcional poner nombre a las sentencias CYCLE y EXIT, pero si se pone, éste debe ser el mismo que el de la sentencia DO.
- La utilidad de los nombres en los bucles aumenta a medida que lo hace el tamaño de los mismos. Por un lado, ayuda al usuario a identificar rápidamente las sentencias que pertenecen un bucle complicado y extenso y, por otro, ayuda al compilador a afinar la localización de las sentencias de error.

3.7 Bucles anidados

- Un bucle puede estar contenido completamente dentro de otro bucle. En este caso, se dice que ambos bucles están *anidados*.
- Cuando dos bucles están anidados, el bucle interno se ejecuta completamente para cada iteración del bucle externo. Es decir, el índice del bucle interno toma todos los valores posibles permitidos por su condición de desigualdad para cada uno de los valores posibles del índice del bucle externo permitidos por su condición de desigualdad.
- Los bucles anidados se cierran en orden inverso a cómo se abren. Así, cuando el compilador encuentra una sentencia END DO, asocia esa sentencia con el bucle más interno abierto.
- Es conveniente asignar nombres a todos los bucles anidados para que sean más fáciles de comprender y de localizar los errores de compilación.
- Los índices de bucles anidados deben tener identificadores distintos.
- Si aparecen sentencias CYCLE o EXIT sin nombre en bucles anidados, éstas se asocian por defecto con el bucle más interno abierto. Para evitar asociaciones automáticas no deseadas, es conveniente escribir las sentencias CYCLE o EXIT con nombre, que será el mismo que el dedicado al cabecero y fin del bucle involucrado.
- La sintaxis general de dos bucles anidados WHILE con nombre añadido es:

```
[externo:] DO
 bloque_de_sentencias1
 [IF (expresión_lógica) CYCLE [externo]]
 bloque_de_sentencias2
 IF (expresión_lógica) EXIT [externo]
 bloque_de_sentencias3
[interno:] DO
          bloque_de_sentencias4
 [IF (expresión_lógica) CYCLE [interno]]
 bloque_de_sentencias5
 IF (expresión_lógica) EXIT [interno]
 bloque_de_sentencias6
END DO [interno]
 bloque_de_sentencias7
END DO [externo]
```

- No es necesario tener bloques de sentencias en todos los puntos marcados del bucle. Dependiendo del caso, como mínimo debe haber alguna sentencia o bloque de ellas a elegir entre los bloques llamados 1, 2 y 3, y, lo mismo, con los bloques 4, 5 y 6.
- Ejemplo de dos bucles DO anidados:
    - Calcular el factorial de los números 3 al 6 y mostrar los resultados por monitor

```
numero: DO i=3,6
 fact=1
 factorial_de_numero: DO j=1,i
   fact=fact*j
 END DO factorial_de_numero
 WRITE (*,*) 'FACTORIAL DE ',i,'=',fact
END DO numero
```

- Para cada valor de i, j toma todos los valores desde 1 hasta esa i, multiplicándolos entre sí para calcular el factorial de ese número i.

3.8 Bucles anidados dentro de estructuras IF y viceversa

- Es posible anidar bucles dentro de estructuras IF o tener estructuras IF dentro de bucles.
- Si un bucle se anida dentro de una estructura IF, el bucle debe permanecer completamente dentro de un único bloque de código de la estructura IF.
- Ejemplo:

```
externo: IF (x<y) THEN
…..
 interno: DO i=1,5


..END DO interno
…..
ELSE
 …..
END IF externo
```

EJERCICIOS RESUELTOS

Objetivos:

Aprender a usar estructuras de control repetitivas o bucles para abreviar el código de los programas Fortran.

Se muestran ejemplos de bucles DO controlados con contador y bucles controlados con expresión lógica. En primer lugar, aparecen estas estructuras simples en los programas y, a continuación, anidadas.

1. Mostrar un mensaje por pantalla, por ejemplo, HOLA A TODOS, cien veces.

```
PROGRAM cap3_1
IMPLICIT NONE
INTEGER :: i

DO  i=1,100
  WRITE (*,*) 'HOLA A TODOS'
END DO
END PROGRAM cap3_1
```

– El usuario que desconoce la existencia de los bucles en programación, podría pensar, en principio, en construir el programa copiando cien veces la instrucción: WRITE (*,*) ’HOLA A TODOS’. Sin embargo, intuitivamente el usuario comprende que debe existir algún procedimiento en Fortran que abrevie significativamente esta tarea, es decir, los bucles.

– En este ejercicio, el índice del bucle i se usa únicamente para indicarnos cuántas veces se ejecuta el cuerpo del bucle.

– Cambia los valores de los parámetros del índice del bucle sin modificar el resultado de la ejecución del programa. ¿Cuántas posibilidades hay?

2. Sumar todos los números naturales desde el 1 hasta el 100, ambos incluidos.

```
PROGRAM cap3_2
IMPLICIT NONE
INTEGER :: suma=0, i

DO i=1,100
  suma=suma+i
!  WRITE(*,*) 'i,suma',i,suma
END DO
WRITE(*,*) 'EL RESULTADO ES',suma
END PROGRAM cap3_2
```

– En este caso, el índice del bucle interviene en el cuerpo del mismo.

– Activa la sentencia comentada del programa y estudia su significado.

– ¿Qué resultado se obtiene si cambiamos el cabecero del bucle por la instrucción DO i=100,1,-1

– Utiliza el depurador del entorno Fortran para comprender mejor la ejecución del bucle.

3. Sumar todos los números naturales pares desde el 10 hasta el 100, ambos incluidos.

```
PROGRAM cap3_3
IMPLICIT NONE
INTEGER :: sumapar=0, i
DO i=100,10,-2
  sumapar=sumapar+i
  WRITE(*,*) 'i,sumapar',i,sumapar
END DO
WRITE(*,*) 'EL RESULTADO ES',sumapar
END PROGRAM cap3_3
```

4. Calcular el valor de π aplicando la fórmula: π = 4* (1 – 1 / 3 +1 / 5 – 1/7) incluyendo hasta el término1 / 99.

```
PROGRAM cap3_4
IMPLICIT NONE
INTEGER :: signo,i
REAL :: pic
pic=0
signo=1
DO  i=1,99,2
  pic=pic+1.0/i*signo
  signo=-signo
END DO
WRITE(*,*) 'EL RESULTADO ES',4*pic
END PROGRAM cap3_4
```

– El índice del bucle se usa para obtener los denominadores de la serie alternada. En cada pasada se cambia el signo del sumando en la variable signo.

– Aunque el ejercicio puede hacerse, como vemos, con un único bucle, repite el ejercicio usando dos, uno para sumar los positivos de la serie y otro para sumar los negativos de la misma. Resta ambos resultados y multiplica por cuatro; debes obtener el mismo resultado. De esta manera, el código del programa aumenta en relación con el anterior, sin embargo es más legible, y, por lo tanto, más sencillo de comprender.

5. Calcular la media de un conjunto de números. El programa recoge tantos datos como el usuario quiera.

```
PROGRAM cap3_5

INTEGER :: cont=0
REAL :: num,acum=0
CHARACTER (LEN=2) :: seguir

DO
  WRITE(*,*) 'DAME UN NUMERO'
  READ (*,*) num
  cont=cont+1
  acum=acum+num
  WRITE(*,*) 'OTRO NUMERO?'
  WRITE(*,*) 'TECLEA SI O NO EN MAYUSCULAS Y ENTRE
APOSTROFES'
  READ (*,*) seguir
  IF (seguir /= 'SI') EXIT
END DO
WRITE(*,*) 'LA MEDIA ES:',acum/cont
END PROGRAM cap3_5
```

– Mientras el valor de la variable seguir sea ‘SI’, la expresión lógica dada es FALSA y se ejecuta el cuerpo del bucle.

– En el momento en que el valor de la variable seguir es distinto de ‘SI’, la expresión lógica se hace cierta, se ejecuta la instrucción EXIT y, por lo tanto, el control pasa a ejecutar la instrucción WRITE(*,*) ‘La media es’,acum./cont

– Recuerda que Fortran no distingue las letras minúsculas de las mayúsculas en todos los contextos excepto en constantes de tipo carácter.

– Si el número de elementos de la lista es conocido a priori, ¿Cómo cambia el código del programa? Sustituye el bucle WHILE por uno iterativo.

6. Calcular la media de un conjunto de números. El programa recoge tantos datos como el usuario quiera calculando tantas medias como él quiera.

```
PROGRAM cap3_6
IMPLICIT NONE
INTEGER :: cont
REAL :: num,acum
CHARACTER (LEN=2) :: seguir,mas_medias

externo: DO
  cont=0
  acum=0
  interno: DO
    WRITE(*,*) 'DAME UN NUMERO'
    READ (*,*) num
    cont=cont+1
    acum=acum+num
    WRITE(*,*) 'OTRO NUMERO?'
    WRITE(*,*) 'TECLEA SI O NO EN MAYUSCULAS Y ENTRE
APOSTROFES'
    READ (*,*) seguir
    IF (seguir /= 'SI') EXIT interno
  END DO interno
  WRITE(*,*) 'LA MEDIA ES:',acum/cont
  WRITE(*,*) 'OTRA MEDIA?'
  WRITE(*,*) 'TECLEA SI O NO EN MAYUSCULAS Y ENTRE
APOSTROFES'
  READ (*,*) mas_medias
  IF (mas_medias /= 'SI') EXIT externo
END DO externo
END PROGRAM cap3_6
```

– Necesitamos dos bucles anidados para realizar el ejercicio. Destacar que la inicialización de las variables cont y acum a cero debe

hacerse antes de entrar al bucle interno WHILE, para que cada vez que se calcule otra media se vuelvan a poner a cero sus valores.

– Si el número de elementos de la lista y el número de listas es conocido a priori, ¿Cómo cambia el código del programa? Sustituye los bucles WHILE por dos iterativos.

7. Calcula y escribe por pantalla la cantidad de números positivos que hay en una lista dada de N elementos. El proceso se repite todas las veces que el usuario quiera.

```
PROGRAM cap3_7

CHARACTER (LEN=1) :: seguir
INTEGER :: pos
REAL :: num

externo: DO
  WRITE(*,*) '"DAME EL NUMERO DE ELEMENTOS DE LA LISTA"'
  READ (*,*) n
  pos=0 !IMPORTANTE, INICIALIZAR PARA CADA LISTA
  interno: DO  i=1,n
    WRITE(*,*) 'INTRODUCE NUMERO',i
    READ (*,*) num
    IF (num > 0) THEN
      pos=pos+1
    END IF
  END DO interno
  WRITE(*,*) 'CANTIDAD DE POSITIVOS DE LOS',n,' LEIDOS ES:',pos
  WRITE(*,*) 'QUIERES SEGUIR CON OTRA LISTA (S/N)?'
  READ (*,*) seguir
  IF (seguir /= 'S') EXIT externo
END DO externo
END PROGRAM cap3_7
```

– Se utilizan dos bucles anidados: el bucle externo controla si seguimos con otra lista o no, mientras el interno opera con una lista determinada, controlando que se lea cada número de la misma, se evalúe si es positivo y, si es cierto, se incremente el valor de un contador una unidad.

– Notar que declarando una única variable num se consigue realizar el ejercicio lo que supone un aprovechamiento eficiente de la memoria.

– Completa el ejercicio contando la cantidad de positivos, negativos y ceros que hay en cada lista.

– Repite el ejercicio sustituyendo el bucle externo WHILE por un bucle DO WHILE.

8. Leer números por teclado hasta introducir uno negativo.

```
PROGRAM cap3_8
IMPLICIT NONE
LOGICAL :: positiv
REAL :: num
DO
  WRITE(*,*) 'DAME UN NUMERO POSITIVO'
  READ (*,*) num
  IF (num > 0) THEN
    positiv=.TRUE.
  ELSE
    positiv=.FALSE.
  END IF
    IF (.NOT.positiv) EXIT
END DO
WRITE(*,*) num,' NO ES POSITIVO'
END PROGRAM cap3_8
```

– Ejemplo de estructura IF dentro de un bucle WHILE.

– Este ejercicio utiliza una variable lógica, llamada positiv, para evaluar la expresión lógica del bucle WHILE. Mientras se lean números positivos, su valor será CIERTO y se ejecuta el cuerpo del bucle. En el momento en que se lee un número negativo, positiv toma el valor FALSO, la expresión lógica se hace cierta, se ejecuta la instrucción EXIT y, por tanto, el control del bucle pasa a ejecutar la sentencia WRITE, escribiendo por monitor que el número dado no es positivo.

9. Calcular el factorial de un número natural (que se pedirá por teclado) usando un bucle. Escribir los resultados parciales.

```
PROGRAM cap3_9
IMPLICIT NONE
```

```
INTEGER :: num,fact,i
WRITE(*,*) 'DAME UN NUMERO'
READ (*,*) num
IF (num<0) THEN
  WRITE(*,*) 'NO EXISTE EL FACTORIAL DE UN NEGATIVO'
ELSE IF (num==0) THEN
  WRITE(*,*) 'EL FACTORIAL DE CERO ES UNO'
ELSE
  fact=1
  interno: DO i=num,1,-1
      fact=fact*i
      WRITE(*,*) 'RESULTADO PARCIAL',fact
  END DO interno
  WRITE(*,*) 'EL FACTORIAL DE',num,' ES ',fact
END IF
END PROGRAM cap3_9
```

- Ejemplo de bucle DO iterativo dentro de una estructura IF.
- Si el número introducido es negativo se escribe un mensaje avisando de que no existe su factorial, sino, si el número es el cero, su factorial es uno y sino se calcula el factorial del número usando un bucle.

EJERCICIOS PROPUESTOS

1) Programa que sume y muestre por pantalla todos los números naturales del 1 hasta el 5, ambos incluidos. Lo mismo pero de 1 a 50; lo mismo pero de 1 a 500.

2) Programa que sume el número 5 y sus múltiplos hasta el 100 inclusive y muestre el resultado por pantalla.

3) Realizar un programa que calcule y muestre la suma de los múltiplos de 5 comprendidos entre dos valores A y B. El programa no permitirá introducir valores negativos para A y B y verificará que A es menor que B. Si A es mayor que B, intercambiará sus valores.

4) Programa que calcule y muestre e^x utilizando los cuatro primeros términos de la serie: $e^x = 1 + x/1! + x^2/2! + x^3/3!$. El valor "exacto" se puede obtener con la función intrínseca EXP(argumento).

5) Programa que lea un número natural y diga si es o no es *triangular*. A saber: un número N es triangular si, y solamente si, es la suma de los primeros M números naturales, para algún valor de M. Ejemplo: 6 es triangular pues 6 = 1 + 2 + 3. Una forma de obtener los números triangulares es aplicando la fórmula: $\frac{n(n+1)}{2} \; \forall n \in N$.

6) Programa que encuentre un número natural n y otro m tal que se cumpla: $1^2 + 2^2 + 3^2 + 4^2 + \ldots + m^2 = n^2$. Solución: $1^2 + 2^2 + 3^2 + 4^2 + \ldots + 24^2 = 70^2$.

7) Programa que muestre la serie de Fibonacci 0, 1, 1, 2, 3, 5, 8, 13, 21,…Los primeros términos son 0 y 1, los siguientes suma de los dos anteriores.

8) Programa que verifique si un número es reproductor de Fibonacci. A saber: un número n se dice reproductor de Fibonacci si es capaz de reproducirse a sí mismo en una secuencia generada con los m dígitos del número en cuestión y continuando en la serie con un número que es la suma de los m términos precedentes. Ejemplo: 47 es un número reproductor de Fibonacci pues la serie: 4, 7, 11, 18, 29, 47,… contiene el 47.

9) Retoma el ejercicio anterior y explora todos los números reproductores de Fibonacci de 2 y de 3 dígitos. Soluciones para dos dígitos: 14, 19, 28, 47, 61, 75. Soluciones para tres dígitos: 197, 742.

10) Programa que pida por teclado la nota de examen, mientras sea suspenso.

11) Programa que pida la estatura (en metros) y sexo (V/M) de un número indeterminado de personas (mientras el operador quiera). Posteriormente escribirá la estatura media de los varones y la estatura media de las mujeres.

12) Escribir un programa que calcule los centros numéricos entre 1 y n. Un centro numérico es un número que separa una lista de números enteros (comenzando en 1) en dos grupos de números, cuyas sumas son iguales. El primer centro numérico es el 6, el cual separa la lista (1 a 8) en los grupos: (1, 2, 3, 4, 5) y (7, 8) cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, el cual separa la lista (1 a 49) en los grupos: (1 a 34) y (36 a 49) cuyas sumas son ambas iguales a 595.

13) Programa que calcule el producto n! (n-1)!...3!*2!*1!

14) Programa que pida un número por teclado y diga si es primo o no, mostrando todos sus divisores.

4 ARRAYS

4.1 Introducción

- En computación es frecuente trabajar con conjuntos ordenados de valores: listas o vectores y tablas o matrices. Para ello, existe una estructura de datos denominada *array*.
- Un array está constituido por un grupo de variables o constantes, todas del mismo tipo. Cada variable dentro del array se denomina *elemento del array*.
- Un array está definido por los siguientes parámetros:
  - Rango: es su número de dimensiones. Así, un escalar posee rango cero, un vector rango uno, una matriz rango dos, etc.
  - Extensión: es el total de componentes que posee en cada una de sus dimensiones. Por ejemplo, en una matriz de 5 filas y 3 columnas, la extensión de su dimensión primera (filas) es 5 y la de su dimensión segunda (columnas) es 3.
  - Tamaño: es el total de elementos que tiene, es decir, el producto de sus extensiones. En el ejemplo anterior, el tamaño de una matriz de 5 filas x 3 columnas es 15.
  - Perfil: es la combinación del rango y la extensión del array en cada dimensión. Por tanto, dos arrays tienen el mismo perfil si tienen el mismo rango y la misma extensión en cada una de sus dimensiones.

4.2 Declaración de arrays

- Antes de usar un array, es necesario declararlo en una sentencia de declaración de tipo. La sintaxis general es:

TIPO, DIMENSION (d1[,d2]...) :: lista de arrays

- TIPO es cualquiera de los tipos de datos Fortran válidos de la Tabla 1.2.
- El atributo DIMENSION permite especificar los valores de los índices máximo y, opcionalmente, mínimo, de cada una de las dimensiones del mismo. Así, la dimensión 1 o d1 de la sintaxis general anterior se sustituye por [límite_inferior_d1:]límite_superior_d1 y análogamente para las demás dimensiones, si las hay.
- La extensión de un array en una dimensión n dn viene dada por la ecuación: Extensión_dn=límite_superior_dn-límite_inferior_dn+1
- Los valores de los límites inferior y superior pueden ser positivos, negativos o 0.
- Si sólo se identifica el límite superior, el límite inferior es por defecto 1.

- El rango máximo de un array es 7.
- *lista de arrays* es un conjunto de arrays separados por comas cuyos nombres son identificadores válidos.
- Ejemplos de sentencias de declaración de arrays:

INTEGER, DIMENSION (4) :: vector1,vector2

INTEGER, DIMENSION (-3:0) :: vector3

CHARACTER (len=20), DIMENSION (50) :: nombres_alumnos

REAL, DIMENSION (2,3) :: matriz1,matriz2

REAL, DIMENSION (0:2,-2:3) :: matriz3,matriz4

- El compilador usa las sentencias de declaración de arrays para reservar el espacio adecuado en la memoria del computador Los elementos de un array ocupan posiciones *consecutivas* en la memoria. En el caso de arrays de rango 2, se almacenan por columnas, es decir, el primer índice toma todos los valores permitidos por su extensión para cada uno de los valores permitidos para el segundo índice.
- En programas largos y complicados, es conveniente usar el atributo PARAMETER para dar nombres a los tamaños de los arrays declarados. De esta manera, sus valores se cambiarán fácilmente.
- Ejemplos:

INTEGER, PARAMETER :: TM=50,TMF=2,TMC=3

CHARACTER (len=20), DIMENSION (TM) :: nombres_alumnos

REAL, DIMENSION (TMF,TMC) :: matriz1,matriz2

4.3 Referencia a los elementos de un array

- La sintaxis general de un elemento de un array es:

nombre_array (i1 [,i2] ...)

- índice i1 es un entero que verifica la siguiente condición límite_inferior_d1 ≤ i1 ≤ límite_superior_d1 y análogamente para los demás índices. Es decir, los valores de los índices permitidos vienen determinados por la extensión de la dimensión correspondiente, definida en la sentencia de declaración del array en cuestión. Por lo tanto, no se puede usar como índice ningún entero fuera de ese intervalo.
- Ejemplo. Sea la declaración siguiente:

REAL, DIMENSION (-1:1,3) :: X

- Elementos válidos de X son:

X(-1,1) X(-1,2) X(-1,3)

X(0,1) X(0,2) X(0,3)

X(1,1) X(1,2) X(1,3)

- No todos los compiladores comprueban que los índices de un array están dentro de sus límites.

4.4 Inicialización de arrays

- Al igual que las variables, los arrays deben inicializarse antes de usarse. Hay tres formas de inicializar un array:
    - En la propia sentencia de declaración de tipo del array, en tiempo de compilación.
    - En sentencias de asignación.
    - En sentencias de lectura (READ).

4.4.1 Inicialización de arrays en sentencias de declaración de tipo

- Para inicializar vectores pequeños, se usa el constructor de arrays, con los delimitadores (/ y /).
- Por ejemplo, para inicializar un vector v1 de 5 elementos con los valores 1, 2, 3, 4 y 5:

```
INTEGER, DIMENSION (5) :: v1 = (/ 1, 2, 3, 4, 5/)
```

- Para inicializar arrays de rango>1 se usa una función intrínseca especial llamada RESHAPE que cambia el perfil de un array sin cambiar su tamaño. Esta función tiene dos argumentos principalmente:

```
RESHAPE (fuente, perfil)
```

- *fuente* es un vector de constantes con los valores de los elementos del array.
- *perfil* es un vector de constantes con el perfil deseado.
- El array construido tendrá sus elementos dispuestos en su orden natural, es decir, si se trata de un array de rango dos, por columnas.
- Ejemplo:

```
INTEGER, DIMENSION (2,3) :: mat1 = &
RESHAPE ( (/ 1, 2, 3, 4, 5, 6/) , (/2,3/) )
```

Matemáticamente, la matriz construida es: $mat1 = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

- Para inicializar arrays de tamaño grande, se usa un bucle DO implícito. La sintaxis general de un bucle implícito es:

```
(arg1[, arg2] …, índice = inicio, fin [, paso])
```

- *arg1* y los demás argumentos, si los hay, se evalúan cada iteración del bucle.

- El índice del bucle y los parámetros del mismo funcionan exactamente igual que en el caso de los bucles DO iterativos estudiados en la sección 3.2.
- Ejemplos:

```
INTEGER, DIMENSION (5) :: v1 = (/ (i, i= 1, 5) /)
INTEGER, DIMENSION (100) :: v2 = (/ (i, i= 100, 1, -1) /)
```

- Los bucles DO implícitos pueden anidarse. La sentencia siguiente inicializa los elementos del vector v3 a 0 si no son divisibles por 5 y al número de elemento si son divisibles por cinco.

```
INTEGER, DIMENSION (20) :: v3 = (/ ((0, i= 1,4),5*j, j=1,4) /)
```

- El bucle interno (0, i=1,4) se ejecuta completamente para cada valor del bucle externo j. Se obtiene:

0, 0, 0, 0, 5, 0, 0, 0, 0, 10, 0, 0, 0, 0, 15, 0, 0, 0, 0, 20

- También es posible inicializar todos los elementos de un array a un valor único constante.
- Ejemplo:

```
INTEGER, DIMENSION (100) :: v = 0
INTEGER, DIMENSION (2,3) :: mat=0
```

4.4.2 Inicialización de arrays en sentencias de asignación

- Los procedimientos vistos en el apartado anterior para inicializar arrays, son también válidos cuando se realizan en el cuerpo de un programa. Por ejemplo, sean las declaraciones de arrays:

```
INTEGER, DIMENSION (5) :: v1,v2
INTEGER, DIMENSION (2,3) :: mat1
…
v1 = (/ 1, 2, 3, 4, 5/)
v2 = (/ (i, i= 10, 6, -1) /)
mat1 = RESHAPE ( (/ 1, 2, 3, 4, 5, 6/) , (/2,3/) )
```

- También se pueden inicializar arrays usando bucles DO iterativos. Por ejemplo, sean las declaraciones de arrays:

```
INTEGER :: i,j,k=0
INTEGER, DIMENSION (5) :: v1,v2
INTEGER, DIMENSION (2,3) :: mat1
…
DO i=1,5
  v1(i)=i
END DO
```

```
DO i=10,6,-1
  v2(11-i)=i
END DO

externo : DO j=1,3
  interno :DO i=1,2
    k=k+1
    mat1(i,j)=k
  END DO interno
END DO externo
```

- También es posible inicializar todos los elementos de un array a un valor único con una simple sentencia de asignación.
- Ejemplo:

```
INTEGER, DIMENSION (100) :: v1
INTEGER, DIMENSION (2,3) :: mat
v1=0
mat=0
```

4.4.3 Inicialización de arrays en sentencias de lectura

- Se pueden leer arrays usando bucles DO iterativos. Por ejemplo, sean las declaraciones:

```
INTEGER :: i,j
INTEGER, PARAMETER :: TM=5,TMF=2,TMC=3
INTEGER, DIMENSION (TM) :: v1,v2
INTEGER, DIMENSION (TMF,TMC) :: mat1
…
DO i=1,TM
  WRITE (*,*) 'DAME ELEMENTO',i,'DE v1 Y v2'
  READ (*,*) v1(i),v2(i)
END DO

DO i=1,TMF
  DO j=1,TMC
    WRITE (*,*) 'DAME ELEMENTO',i,j,'DE mat1'
    READ (*,*) mat1(i,j)
```

```
!lectura por filas
  END DO
END DO

DO j=1,TMC
  DO i=1,TMF
    WRITE (*,*) 'DAME ELEMENTO',i,j,'DE mat1'
    READ (*,*) mat1(i,j)
!lectura por columnas
  END DO
END DO
```

- En los ejemplos anteriores, la lectura de los arrays se realiza elemento a elemento en el orden establecido en los bucles.
- También se puede leer un array usando bucles DO implícitos. La sintaxis general de lectura con un bucle implícito es:

READ (*,*) (arg1[, arg2] …, índice = inicio, fin [, paso])

- *arg1* y los demás argumentos, si los hay, son los argumentos que se van a leer.
- El índice del bucle y los parámetros del mismo funcionan exactamente igual que en el caso de los bucles DO iterativos estudiados en la sección 3.2.
- Ejemplos:

```
INTEGER, DIMENSION (5) :: w
INTEGER, DIMENSION (0:3,2) :: m
…
READ (*,*) ( w(i), i=5,1,-1)
READ (*,*) ( (m(i,j), i=0,3) , j=1,2) !lectura por columnas
READ (*,*) ( (m(i,j), j=1,2) , i=0,3) !lectura por filas
```

- En los ejemplos anteriores, la lectura de los arrays se realiza elemento a elemento en el orden establecido en los bucles.
- También es posible leer un array escribiendo únicamente su nombre, sin especificar su(s) índice(s). En ese caso, el compilador asume que se va a leer todo el array, es decir, todos sus elementos en el orden natural Fortran.
- Ejemplo:

```
INTEGER, DIMENSION (10) :: v
INTEGER, DIMENSION (2,3) :: mat
READ (*,*) v
```

READ (*,*) mat

- Todas las formas anteriores usadas para leer arrays son también válidas para escribir arrays, sin más que sustituir la palabra reservada READ por WRITE, en las instrucciones correspondientes.

4.5 Operaciones sobre arrays completos

- Cualquier elemento de un array se puede usar como cualquier otra variable en un programa Fortran. Por tanto, un elemento de un array puede aparecer en una sentencia de asignación tanto a la izquierda como a la derecha de la misma.
- Los arrays completos también se pueden usar en sentencias de asignación y cálculos, siempre que los arrays involucrados tengan el mismo perfil, aunque no tengan el mismo intervalo de índices en cada dimensión. Además, un escalar puede operarse con cualquier array.
- Una operación aritmética ordinaria entre dos arrays con el mismo perfil se realiza *elemento a elemento*.
- Si los arrays no tienen el mismo perfil, cualquier intento de realizar una operación entre ellos provoca un error de compilación.
- La mayoría de las funciones intrínsecas comunes (ABS, SIN, COS, EXP, LOG, etc.) admiten arrays como argumentos de entrada y de salida, en cuyo caso la operación se realiza elemento a elemento. A este tipo de funciones intrínsecas se las llama funciones intrínsecas *elementales*.
- Existen otros dos tipos de funciones intrínsecas, las llamadas de *consulta* y las *transformacionales*. Las primeras operan con arrays completos y las segundas proporcionan información sobre las propiedades de los objetos estudiados.
- Ejemplo de operaciones sobre arrays completos:

```
INTEGER, DIMENSION (5) :: v1 =(/ 9,8,7,6,5 /)
INTEGER, DIMENSION (20:24) :: v2 =(/ -9,-8,-7,-6,-5 /)
INTEGER, DIMENSION (50:54) :: v3,v4

v3=v1+v2
v4=v1*v2
WRITE (*,*) ‘v3, dos veces exponencial de v3’, v3, 2*EXP(REAL(v3))
WRITE (*,*) ‘v4, valor absoluto de v4’, v4, ABS(v4)
```

Se escriben v3, dos veces exponencial de v3: 0 0 0 0 0, 2, 2, 2, 2, 2 y en otra línea v4, valor absoluto de v4: -81, -64, -49, -36, -25, 81, 64, 49, 36, 25.

- Otra forma de codificar lo anterior, operando con los índices de los arrays en bucles DO iterativos es (usar el mismo rango de índices en todos los arrays, para facilitar la tarea, por ejemplo de 1 a 5):

```
INTEGER :: i
INTEGER, DIMENSION (5) :: v1 =(/ 9,8,7,6,5 /),&
v2 =(/ -9,-8,-7,-6,-5 /),v3,v4

DO i=1,5
  v3(i)=v1(i)+v2(i)
  v4(i)=v1(i)*v2(i)
  WRITE (*,*) v3(i), 2*EXP(REAL(v3(i)))
  WRITE (*,*) v4(i), ABS(v4(i))
END DO
```

4.6 Operaciones sobre subconjuntos de arrays

- Hasta ahora hemos visto que es posible usar elementos de arrays o array completos en sentencias de asignación y cálculos. Asimismo, es posible usar cualquier subconjunto de un array, llamado *sección de array*.
- Para especificar una sección de un array, basta sustituir el índice del array por:
  - un *triplete de índices* o
  - un *vector de índices*.

4.6.1 Tripletes de índices

- Sea la declaración:

```
TIPO, DIMENSION (d1[,d2]...):: nombre_array
```

- La sintaxis general de una sección del array anterior con tripletes de índices es:

```
nombre_array (trip1, [tripl2]...)
```

- El triplete de índices se puede usar para especificar secciones de un array en cualquiera de sus dimensiones o en todas, según interese. La sintaxis general de un triplete *tripl* es:

```
[indice_inicial_tripl] : [indice_final_tripl] [: paso_tripl]
```

- El triplete en una dimensión dada especifica un subconjunto *ordenado* de índices del array *nombre_array* para esa dimensión, empezando con *indice_inicial* y acabando con *indice_final* con incrementos dados por el valor de *paso*.
- Cualquier valor del triplete o los tres son opcionales:

  - Si *indice_inicial* no aparece, su valor por defecto es el índice del primer elemento del array.
  - Si *indice_final* no aparece, su valor por defecto es el índice del último elemento del array.
  - Si *paso* no aparece, su valor por defecto es 1.

- Ejemplo de secciones de array con tripletes de índices:

Sea la matriz= $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$, la sección del array formada por los elementos de la primera fila se puede especificar con el triplete: matriz(1,:)= $\begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$. La sección del array formada por los elementos de la primera columna se puede especificar con el triplete: matriz(:,1)= $\begin{pmatrix} 1 \\ 5 \\ 9 \\ 13 \end{pmatrix}$. La sección del array formada por los elementos de las filas impares y columnas pares es: matriz(1:3:2,2:4:2)= $\begin{pmatrix} 2 & 4 \\ 10 & 12 \end{pmatrix}$.

4.6.2 Vectores de índices

- Un vector de índices es un array entero *unidimensional* que especifica los elementos de un array a usar en un cálculo. Los elementos del array se pueden especificar en *cualquier orden* y pueden aparecer *más de una vez*. En este último caso, la sección del array no puede aparecer a la izquierda de una sentencia de asignación pues ello significaría que dos o más valores diferentes tendrían que ser asignados al mismo elemento del array.
- Ejemplo de sección de un vector con un vector de índices:

INTEGER, DIMENSION (5) :: v_indices =(/ 1,6,3,4,6 /)

REAL, DIMENSION (8) :: v =(/ -1.,2.,3.3,4.0,-5.5,0,-7.0,8.8 /)

Según las sentencias de declaración anteriores, v(v_indices) especifica los valores de los elementos v(1), v(6), v(3), v(4), v(6), es decir, -1. 0 3.3 4.0 0.

4.7 Construcción WHERE

- La construcción WHERE permite llevar a cabo asignaciones y cálculos sólo sobre determinados elementos de uno o más arrays. Para ello, esta construcción trabaja con la ayuda de un filtro o máscara formada por un *array lógico* en la propia sentencia WHERE.

- La sintaxis general de una construcción WHERE es similar a la de un bloque IF:

```
[nombre:] WHERE (expresión máscara 1)
   bloque 1 de sentencia(s) de asignación de arrays
[ELSEWHERE (expresión máscara 2) [nombre]
   bloque 2 de sentencia(s) de asignación de arrays]
…
[ELSEWHERE [nombre]
   bloque n de sentencia(s) de asignación de arrays]
END WHERE [nombre]
```

- Expresión máscara es un array lógico del mismo perfil que los arrays manipulados en las sentencias de asignación de arrays de los bloques.
- Puede haber cualquier número de cláusulas ELSEWHERE con máscara y, a lo sumo, una cláusula ELSEWHERE.
- Su funcionamiento es el siguiente:
    - Se aplica la operación o conjunto de operaciones del bloque 1 a todos los elementos del array para los cuales la expresión máscara 1 es cierta.
    - Se aplica la operación o conjunto de operaciones del bloque 2 a todos los elementos del array para los cuales la expresión máscara 1 es falsa y la expresión máscara 2 es cierta.
    - Se repite el razonamiento anterior para todas las cláusulas ELSEWHERE con máscara que haya.
    - Finalmente, se aplica la operación o conjunto de operaciones del bloque n a todos los elementos del array para los cuales todas las expresiones máscara anteriores han sido falsas.
- Ejemplo. Calcular la raíz cuadrada de los elementos positivos de un array. Poner un cero en el resto de los elementos.

```
REAL, DIMENSION (5,4) :: mati,matf
…
WHERE (mati>0)
  matf=SQRT(mati)
ELSEWHERE
  matf=0
END WHERE
```

La expresión máscara mati>0 produce un array lógico cuyos elementos son cierto cuando los elementos correspondientes de mati son positivos y falso cuando los elementos correspondientes de mati son negativos o cero.

Otro modo, combinando dos bucles anidados con un bloque IF:

```
recorre_filas: DO i=1,5
  recorre_columnas: DO j=1,4
    elemento_positivo: IF (mati(i,j)>0) THEN
      matf(i,j)=SQRT(mati(i,j))
    ELSE
      matf(i,j)=0
    ENDIF elemento_positivo
  END DO recorre_columnas
END DO recorre_filas
```

- La construcción WHERE es generalmente más elegante que las operaciones efectuadas elemento a elemento, especialmente con arrays multidimensionales.

4.8 Sentencia WHERE

- Es el caso particular más simple de una construcción WHERE.
- Su sintaxis general es:

```
WHERE (expresión máscara) sentencia de asignación de arrays
```

- La sentencia de asignación se aplica sólo sobre aquellos elementos del array para los cuales la expresión máscara es cierta.

4.9 Construcción FORALL

- Aparece en Fortran 95 para realizar un conjunto de operaciones efectuadas elemento a elemento sobre un subconjunto de elementos de un array.
- Su sintaxis general es:

```
[nombre:] FORALL (ind1=tripl1 [,ind2=tripl2]…[,expresión lógica])
   sentencia1
   [sentencia2]
   …
END FORALL [nombre]
```

- Cada índice ind se especifica por un triplete de la forma vista en el apartado 4.6.1: [*ind_inicial_tripl*] : [*ind_final_tripl*] [: *paso_tripl*].
- La sentencia o sentencias que forman el cuerpo de la estructura se ejecutan sólo sobre aquellos elementos del array que cumplen las restricciones de índices y la expresión lógica dada en la cláusula FORALL.

- Ejemplo: calcular el inverso de cada elemento perteneciente a la diagonal principal de una matriz de 7x7 evitando las divisiones por cero.

```
REAL, DIMENSION (7,7) :: mat
…
FORALL (i=1:7, mat(i,i)/=0)
  mat(i,i)=1./mat(i,i)
END FORALL
```

Otro modo, combinando dos bucles anidados y un bloque IF:

```
recorre_diagonal: DO i=1,7
  IF (mat(i,i)/=0) mat(i,i)=1./mat(i,i)
END DO recorre_diagonal
```

- Todo lo que se programa con una construcción FORALL se puede programar con un conjunto de bucles anidados combinado con un bloque IF.
- La diferencia entre ambas formas es que mientras en los bucles las sentencias deben ser ejecutadas en un orden estricto, la construcción FORALL puede ejecutarlas en cualquier orden, seleccionadas por el procesador. Así, un computador con varios procesadores en paralelo puede repartirse los elementos de una sentencia optimizando la velocidad de cálculo. Cuando el cuerpo de la estructura FORALL contiene más de una sentencia, el computador procesa completamente todos los elementos involucrados en la misma antes de pasar a la siguiente. Por tanto, no pueden usarse funciones cuyo resultado dependa de los valores de un array completo (funciones transformacionales).
- Ejemplo:

```
REAL, DIMENSION (7,7) :: mat
…
FORALL (i=1:7, j=1:7,mat(i,j)/=0)
  mat(i,j)=SQRT(ABS(mat(i,j)))
  mat(i,j)=1./mat(i,j)
END FORALL
```

4.10 Sentencia FORALL

- Es el caso particular más simple de una construcción FORALL.
- Su sintaxis general es:

```
FORALL (ind1=tripl1 [,ind2=tripl2]…[,expresión lógica]) sentencia
```

- La sentencia de asignación se aplica sólo sobre aquellos elementos del array cuyos índices cumplen las restricciones y la expresión lógica dadas.

4.11 Arrays dinámicos

- Hasta ahora el tamaño de los arrays se ha especificado en las propias sentencias de declaración de tipo. Se dice que son arrays *estáticos* en cuanto que tamaño se fija en tiempo de compilación y a partir de entonces no se puede modificar.
- Sin embargo, lo ideal sería que el programador pudiera dar el tamaño de los arrays al ejecutar sus programas según sus necesidades, para aprovechar eficazmente la memoria del computador.
- A partir de Fortran 90, es posible usar arrays cuyo tamaño se fija en tiempo de ejecución, los llamados arrays *dinámicos*.
- La sintaxis general de una sentencia de declaración de arrays dinámicos es:

```
TIPO, ALLOCATABLE, DIMENSION (:[,:]...):: lista_de_arrays
```

- ALLOCATABLE es el atributo que declara que los arrays de la *lista* serán dimensionados dinámicamente. El número de sus dimensiones se declara con dos puntos :, sin especificar los límites.
- La memoria se asigna en tiempo de ejecución para la lista de arrays dinámicos, usando una sentencia ALLOCATE, que especifica los tamaños:

```
ALLOCATE ( arr1( d1 [ ,d2] ... ) [, arr2( d1 [ ,d2]] ...) [,STAT=estado] )
```

- Esta instrucción asigna memoria en tiempo de ejecución para la lista de arrays previamente declarada con el atributo ALLOCATABLE. Establece para cada array su tamaño, con los límites inferior y superior de cada dimensión.
- STAT= *estado*. Cualquier fallo en la localización de memoria causa un error en tiempo de ejecución, a menos que se use este parámetro. La variable entera *estado* devuelve un cero si se ha conseguido reservar espacio suficiente en memoria, en otro caso devuelve un número de error que depende del compilador.
- Se debe liberar la memoria reservada dinámicamente cuando ya no hace falta escribiendo la instrucción:

```
DEALLOCATE (lista_de_arrays [, STAT=estado])
```

- Esta sentencia es útil en programas grandes con necesidades grandes de memoria. En ellos, se usará la sentencia DEALLOCATE en la línea de código a partir de la cual determinados arrays ya no juegan ningún papel, liberando ese espacio de memoria que queda por lo tanto disponible para realizar nuevas reservas de espacio.

- Cualquier fallo al intentar liberar la memoria causa un error en tiempo de ejecución, a menos que el parámetro STAT= *estado* esté presente. La variable *estado* devuelve un cero si la liberación de memoria se hizo con éxito, en otro caso devuelve un número de error que depende del compilador.
- Ejemplo:

```
INTEGER ::estado_reserva, estado_libera
REAL, ALLOCATABLE, DIMENSION (:,:) :: alfa,beta
……
ALLOCATE (alfa(1:100, -1:200), STAT=estado_reserva)
IF (estado_reserva/=0) STOP ‘array alfa NO guardado en memoria’
…
DEALLOCATE (alfa, STAT= estado_libera)
IF (estado_libera/=0) STOP ‘NO liberada memoria para array alfa’
ALLOCATE (beta(1:1000, 200), STAT=estado_reserva)
IF (estado_reserva/=0) STOP ‘array beta NO guardado en memoria’
…
```

EJERCICIOS RESUELTOS

Objetivos:

Aprender a usar arrays en Fortran, comenzando por los vectores y ampliando los conceptos a matrices bidimensionales y tridimensionales.

Por un lado, se usan bucles DO iterativos cuyos índices serán los índices de los arrays, para recorrer los elementos de los mismos. Por otro lado, se manejan las estructuras vistas en este capítulo para trabajar con secciones de arrays o arrays completos.

1. Pedir cinco números por teclado y calcular su media usando un vector para almacenar los números.

```
PROGRAM cap4_1
IMPLICIT NONE
INTEGER :: i
REAL :: media=0
REAL,DIMENSION (5) :: v
DO i=1,5
  WRITE(*,*) 'DAME COMPONENTE',i
  READ(*,*) v(i)
END DO
DO i=1,5
  media=media+v(i)
END DO
media=media/5
WRITE(*,*) 'LA MEDIA ES:',media
END PROGRAM cap4_1
```

– Este programa puede resolverse declarando únicamente una variable (como se hizo en el capítulo anterior). La motivación de almacenar en memoria los cinco números estaría, en el caso de que el programa fuera más grande, y necesitara de sus valores para seguir operando con ellos más adelante.

– Los cinco números pueden guardarse en cinco variables o en un vector de 5 componentes. Lógicamente, la utilidad de un vector aumenta cuantos más números tengamos que almacenar.

2. Calcular y presentar en pantalla los valores 3 a N (100 como máximo) de la sucesión:$X_i=X_{i-1}+ 3X_{i-2} + 4 / 3$. El programa leerá por teclado los valores de N, X_1 y X_2.

```
PROGRAM cap4_2
REAL, DIMENSION(100) :: x
DO
  WRITE(*,*) 'DAME EL VALOR DE N (100 COMO MAXIMO)'
  READ(*,*) n
  IF (n<=100) EXIT
  WRITE(*,*) 'COMO MUCHO 100!'
END DO
```

```
WRITE(*,*) 'DAME EL VALOR DE X1 Y X2'
READ(*,*) x(1),x(2)
DO i=3, n
  x(i)=x(i-1)+3*x(i-2)+4.0/3
  WRITE(*,*) 'X(',i,'):',x(i)
END DO
END PROGRAM cap4_2
```

– Se dimensiona el vector X a 100 componentes, valor máximo que pide el ejercicio. De este modo, se reserva el máximo espacio en memoria que se puede ocupar, aunque generalmente sobre.

– Un bucle WHILE controla que el usuario escribe un valor para N menor o igual que el máximo, 100.

3. Buscar un número en un vector de seis componentes desordenado. Introduce el vector en la propia sentencia de declaración de tipo.

```
PROGRAM cap4_3
INTEGER :: x,switch=0,i
INTEGER, DIMENSION(6):: v=(/17,3,22,11,33,19/)
WRITE(*,*) 'DAME UN NUMERO'
READ(*,*) x
DO i=1,6
  IF (x == v(i)) THEN
    WRITE(*,*) 'ENCONTRADO EN POSICION',i
    switch=1
  END IF
END DO
IF (switch == 0) THEN
  WRITE(*,*) 'NO ENCONTRADO'
END IF
END PROGRAM cap4_3
```

– El bucle DO iterativo permite recorrer todas las componentes del vector y mostrar todas las posiciones del mismo en que encuentra el valor buscado X, si las hay.

– Cuando el valor buscado no coincide con ninguna componente del vector, la variable interruptor o switch tendrá el valor cero y sirve de “chivato” para conocer este hecho.

4. Realizar la búsqueda de un número en un vector. Reducir la búsqueda a encontrar la primera posición del array en que se encuentra coincidencia. Si no existe el nº en el array, mostrar un mensaje en consecuencia.

```
PROGRAM cap4_4
INTEGER :: num,i,chivato=0
INTEGER,PARAMETER :: NC=5
INTEGER, DIMENSION(NC) ::vec=(/1,1,6,6,7/)
WRITE(*,*) 'DAME UN NUMERO'
READ(*,*) num
DO i=1,NC
  IF (vec(i) == num) THEN
    WRITE(*,*) 'PRIMERA COINCIDENCIA EN POSICION',i,'DEL VECTOR'
    chivato=1
    EXIT
  END IF
END DO
IF (chivato == 0) THEN
  WRITE(*,*) 'NO EXISTE EL NUMERO EN EL VECTOR'
END IF
END PROGRAM cap4_4
```

– La sentencia EXIT del bucle WHILE permite reducir la búsqueda a la primera coincidencia en el vector.

5. Buscar un número en un vector de siete componentes ordenado ascendentemente. Introduce el vector en la propia sentencia de declaración de tipo.

```
PROGRAM cap4_5
INTEGER :: x,i=0
INTEGER,DIMENSION(7):: v=(/0,3,5,7,11,11,23/)
WRITE(*,*) 'DAME UN NUMERO'
READ(*,*) x
DO
  i=i+1
  IF (i==7 .OR. v(i)>=x) EXIT
END DO
IF (x == v(i)) THEN
```

```
  WRITE(*,*) 'ENCONTRADO EN POSICION',i
ELSE
  WRITE(*,*) 'NO ENCONTRADO'
END IF
END PROGRAM cap4_5
```

– Este algoritmo recorre todas las componentes del vector de izquierda a derecha hasta que, o bien llegamos a la última componente, o bien el valor buscado es inferior a algún elemento del vector (pues está ordenado).

– El programa encuentra únicamente la primera posición del vector en que se halla el valor buscado, aunque éste aparezca repetido en varias posiciones del mismo.

6. Buscar un número en un vector de siete componentes ordenado ascendentemente. Introduce el vector en la propia sentencia de declaración de tipo. Usa el algoritmo de la búsqueda binaria o dicotómica.

```
PROGRAM cap4_6
INTEGER :: x,izq=1,der=7,cen
INTEGER, DIMENSION(7)::v=(/-2,0,4,6,8,19,23/)
WRITE(*,*) 'DAME UN NUMERO'
READ(*,*) x
cen=(izq+der)/2
WRITE(*,*) 'CENTRO',cen
DO WHILE (x/=v(cen).AND.izq<der)
  IF (x>v(cen)) THEN
    izq=cen+1
  ELSE
    der=cen-1
  END IF
  cen=(izq+der)/2
  WRITE(*,*) 'CENTRO',cen
END DO
IF (x==v(cen)) THEN
  WRITE(*,*) 'ENCONTRADO EN POSICION',cen
ELSE
  WRITE(*,*) 'NO EXISTE EL VALOR BUSCADO'
END IF
```

```
END PROGRAM cap4_6
```

– La búsqueda dicotómica es un algoritmo más eficiente que el anterior. Aquí la búsqueda se realiza siempre en la componente central de un vector cuya longitud se va reduciendo a la mitad izquierda o derecha del centro (según sea el número) sucesivamente hasta que, o bien encontramos el valor buscado, o bien el intervalo de búsqueda se ha reducido a una componente.

7. Ordenar ascendentemente las seis componentes de un vector. Usar el método de la burbuja.

```
PROGRAM cap4_7
INTEGER::aux,i,j
INTEGER, PARAMETER ::NC=6
INTEGER, DIMENSION(NC) :: vec
vec=(/8,5,15,2,-19,0/)
cantidad_parejas:DO i=NC-1,1,-1
  pareja:DO j=1,i
    ordenar_pareja:IF (vec(j) > vec(j+1)) THEN
      aux=vec(j)
      vec(j)=vec(j+1)
      vec(j+1)=aux
    END IF ordenar_pareja
  END DO pareja
END DO cantidad_parejas
WRITE(*,*) 'EL VECTOR ORDENADO ES'
!$$$$$$ DO i=1,NC
!$$$$$$   WRITE(*,*) vec(i)
!$$$$$$ END DO
WRITE(*,*) (vec(i),i=1,NC)
END PROGRAM cap4_7
```

– Este algoritmo considera dos bucles DO iterativos anidados para realizar el ordenamiento; el bucle externo controla el número de parejas consideradas en el tratamiento y el bucle interno controla la pareja evaluada. A su vez, el bloque IF controla que tal pareja esté ordenada ascendentemente, intercambiando sus valores en caso contrario.

– Para ordenar el vector descendentemente (de mayor a menor) basta cambiar el operador relacional "mayor que" (>) por "menor que" (<) en el código del programa.

8. Leer una matriz de 2 filas y 3 columnas por teclado y mostrarla por pantalla.

```
PROGRAM cap4_8
IMPLICIT NONE
INTEGER::i,j
INTEGER, DIMENSION(2,3):: mat
WRITE(*,*) 'DAME LAS COMPONENTES DE LA MATRIZ'
READ(*,*) mat
filas: DO  i=1,2
  columnas: DO  j=1,3
    WRITE(*,*) 'COMPONENTE',i,j,':',mat(i,j),LOC(mat(i,j))
  END DO columnas
END DO filas
END PROGRAM cap4_8
```

– El interés del ejercicio radica en conocer cómo se almacenan las matrices en memoria. Por defecto, las matrices almacenan sus componentes en posiciones de memoria consecutivas *por columnas*.

– El nombre de la matriz es un puntero a la primera componente, es decir, es una variable que almacena la dirección de memoria de la primera componente de la matriz.

– LOC(*arg*) es una función intrínseca que recibe como argumento una variable de cualquier tipo o elemento de array y devuelve la dirección de memoria en la que está almacenado.

MAT					
MAT(1,1)	MAT(2,1)	MAT(1,2)	MAT(2,2)	MAT(1,3)	MAT(2,3)

9. Inicializar las componentes de una matriz de 2X3 en su sentencia de declaración de tipo a ar= $\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$.

```
PROGRAM cap4_9
```

```
INTEGER,DIMENSION(2,3)::ar=&
RESHAPE((/1,1,2,2,3,3/),(/2,3/))
columnas: DO  j=1,3
  filas: DO  i=1,2
    WRITE(*,*)
    WRITE(*,*) 'COMPONENTE',i,j,'=',ar(i,j)
    WRITE(*,*)
    WRITE(*,*) 'DIRECCION DE MEMORIA',LOC(ar(i,j))
  END DO filas
END DO columnas
END PROGRAM cap4_9
```

– La función intrínseca RESHAPE se utiliza para inicializar el array bidimensional *ar*. Se verifica que la carga de los valores de la matriz se realiza por columnas.

10. Buscar un número en una matriz desordenada de tres filas y cinco columnas (3X5). Leer la matriz con un bucle implícito anidado.

```
PROGRAM cap4_10
INTEGER::x,sw,i,j
INTEGER, PARAMETER:: NF=3,NC=5
INTEGER, DIMENSION(NF,NC):: mat
WRITE(*,*) 'dame la matriz por columnas'
READ(*,*) ((mat(i,j),i=1,NF),j=1,NC)
sw=0
WRITE(*,*) 'DAME UN NUMERO'
READ(*,*) x
WRITE(*,*) 'LA MATRIZ ES'
!..............................
DO i=1,NF
  WRITE(*,*) (mat(i,j),j=1,NC)
END DO
!.............................
!WRITE(*,*) mat
!...............................
!filas: DO i=1,NF
!  columnas: DO j=1,NC
!    WRITE(*,*) mat(i,j)
```

```
!  END DO columnas
!END DO filas
!............................
filas: DO i=1,NF
  columnas: DO j=1,NC
    coincidir: IF (x==mat(i,j)) THEN
      sw=1
      WRITE(*,*) 'ENCONTRADO EN FILA:',i,'COLUMNA:',j
    END IF coincidir
  END DO columnas
END DO filas
IF (sw==0) WRITE(*,*) 'NO EXISTE EL VALOR BUSCADO'
END PROGRAM cap4_10
```

– Para visualizar por pantalla el contenido de la matriz en la forma habitual hemos usado un bucle iterativo para el índice de las filas y un bucle implícito para el índice de las columnas.

– El algoritmo utilizado es el mismo que el del ejercicio CAP4_4 para un vector.

11. Calcular el vector suma de 3 vectores de 4 componentes conocidas y el módulo del vector suma, escribiendo los resultados por pantalla.

```
PROGRAM cap4_11
INTEGER:: j,k
INTEGER, PARAMETER:: NC=4
REAL :: modu
REAL,DIMENSION(NC):: v1=-1.,v2=2.,v3=-3.,s
DO j=1,NC
  s(j)=v1(j)+v2(j)+v3(j)
  WRITE(*,*) 'COMPONENTE',j,'DEL VECTOR SUMA:',s(j)
END DO
!$$$$$$ s=v1+v2+v3
!$$$$$$ WRITE(*,*) 'VECTOR SUMA:',s
modu=0
DO k=1,NC
  modu=modu+s(k)**2
END DO
modu=SQRT(modu)
```

```
WRITE(*,*) 'EL MODULO DEL VECTOR SUMA ES:',modu
END PROGRAM cap4_11
```

```
PROGRAM cap4_11
INTEGER, PARAMETER:: NC=4
REAL :: modu
REAL,DIMENSION(NC):: v1=-1.,v2=2.,v3=-3.,s
s=v1+v2+v3
WRITE(*,*) 'VECTOR SUMA:',s
modu=SQRT(SUM(s**2))
WRITE(*,*) 'EL MODULO DEL VECTOR SUMA ES:',modu
END PROGRAM cap4_11
```

- ¿Cómo cambia el programa si el número de componentes de los vectores es 5? Notar que, gracias al atributo PARAMETER usado para especificar la dimensión de los vectores en el programa, los cambios que hay que hacer en el código del mismo se reducen a sustituir 4 por 5 en esa sentencia.
- Para calcular el vector suma de un número grande de vectores, podemos seguir el mismo procedimiento que en este ejercicio. Sin embargo, es más práctico usar una matriz bidimensional en la que cada fila o cada columna de la misma sea un vector.

12. Calcular el vector suma de 3 vectores de 4 componentes conocidas y el módulo del vector suma, escribiendo los resultados por pantalla. Utilizar una matriz para almacenar los 3 vectores.

```
PROGRAM cap4_12
INTEGER:: i,j
INTEGER,PARAMETER::N=3,M=4
REAL :: modu
REAL,DIMENSION(N,M):: a
REAL,DIMENSION(M)::s
!NUMERO DE FILAS DE A ES Nº DE VECTORES A SUMAR
!NUMERO DE COLUMNAS DE A ES Nº DE ELEM. DE CADA VECTOR
filas: DO i=1,N
  columnas: DO j=1,M
    WRITE(*,*) 'INTRODUCE LA COMPONENTE',i,j,'DE LA MATRIZ A'
    READ(*,*)a(i,j)
  END DO columnas
```

```
END DO filas
DO i=1,M
  s(i)=0
END DO
columnas: DO  j=1,M
  filas: DO  i=1,N
    s(j)=s(j)+a(i,j)
  END DO filas
  WRITE(*,*) 'COMPONENTE',j,'DEL VECTOR SUMA:',s(j)
END DO columnas
sumcuad=0
DO  k=1,M
  sumcuad=sumcuad+s(k)**2
  WRITE(*,*) 'SUMA DE CUADRADOS DE COMP. DE S ES:',sumcuad
END DO
modu=SQRT(sumcuad)
WRITE(*,*) 'EL MODULO DEL VECTOR SUMA ES:',modu
END PROGRAM cap4_12
```

```
PROGRAM cap4_12
INTEGER:: j
INTEGER,PARAMETER::N=3,M=4
REAL :: modu
REAL,DIMENSION(N,M):: a
REAL,DIMENSION(M)::s
!NUMERO DE FILAS DE A ES Nº DE VECTORES A SUMAR
!NUMERO DE COLUMNAS DE A ES Nº ELEM. DE CADA VECTOR
WRITE(*,*) 'DAME LA MATRIZ POR FILAS'
READ(*,*) ((a(i,j),j=1,M),i=1,N)
columnas: DO i=1,M
  s(i)=SUM(a(1:N,i))
  WRITE(*,*) 'elemento',i,'del vector suma:',s(i)
END DO columnas
modu=SQRT(SUM(s**2))
WRITE(*,*) 'EL MODULO DEL VECTOR SUMA ES:',modu
END PROGRAM cap4_12
```

– Notar que el vector suma juega el papel de un acumulador, de ahí que sea necesario inicializarlo antes de que aparezca en la sentencia de asignación en el siguiente bucle.

– Recordar que en Fortran el orden de cierre de los bucles es inverso al orden de apertura de los mismos.

13. Calcular la suma de dos matrices de 3 filas y 2 columnas (3X2) y escribir el resultado por pantalla.

```
PROGRAM cap4_13
INTEGER:: i,j
INTEGER,PARAMETER::N=3,M=2
REAL,DIMENSION(N,M):: mat1,mat2,suma
!***LECTURA DE LAS MATRICES, POR FILAS
filas: DO  i=1,N
  columnas: DO  j=1,M
    WRITE(*,*) 'ELEMENTO',i,j,' DE MAT1 Y MAT2'
    READ(*,*) mat1(i,j),mat2(i,j)
  END DO columnas
END DO filas
!***CALCULO DE LA MATRIZ SUMA, POR FILAS
DO  i=1,N
  DO  j=1,M
    suma(i,j)=mat1(i,j)+mat2(i,j)
  END DO
END DO
!$$$$$$ suma=mat1+mat2 !operando con las matrices completas
!***VISUALIZACION DE LA MATRIZ SUMA, POR FILAS
PRINT*
WRITE(*,*) 'MATRIZ SUMA'
PRINT*
DO  i=1,N
  WRITE(*,*) (suma(i,j),j=1,M)
END DO
END PROGRAM cap4_13
```

```
PROGRAM cap4_13
IMPLICIT NONE
INTEGER, PARAMETER:: NF=3,NC=2,NM=2
```

```
REAL, DIMENSION(NF,NC,NM) :: x
REAL, DIMENSION(NF,NC):: suma=0
INTEGER:: i,j,k
!***LECTURA DE LAS MATRICES, POR FILAS
matriz: DO  k=1,NM
  filas: DO  i=1,NF
    columnas: DO  j=1,NC
      WRITE(*,*) 'ELEMENTO',i,j,' DE MATRIZ',k
      READ(*,*) x(i,j,k)
    END DO columnas
  END DO filas
END DO matriz
!***CALCULO DE LA MATRIZ SUMA
filas: DO  i=1,NF
  columnas: DO  j=1,NC
    matriz: DO  k=1,NM
      suma(i,j)=suma(i,j)+x(i,j,k)
    END DO matriz
  END DO columnas
END DO filas
!***VISUALIZACION DE ELEMENTOS DE LA MATRIZ SUMA, POR FILAS
PRINT*
PRINT*,'MATRIZ SUMA'
PRINT*
DO  i=1,NF
  WRITE(*,*) (suma(i,j),j=1,NC)
END DO
END PROGRAM cap4_13
```

- Para evitar corregir el atributo PARAMETER cada vez que cambien las dimensiones de las matrices, teniendo que compilar el nuevo programa, un truco puede ser reservar más espacio en memoria del necesario habitualmente.
- En este ejercicio, tanto la lectura de las matrices, como el cálculo de la matriz suma se ha realizado por filas. Repite el ejercicio trabajando por columnas.
- Para sumar un número grande de matrices bidimensionales podríamos usar el mismo procedimiento que en este ejercicio. Sin embargo, es más práctico usar un array tridimensional en el que la tercera dimensión se corresponda con el número de matrices bidimensionales a sumar. Por ejemplo, para sumar 15 matrices de

3X2, declarar un array de dimensiones (3,2,15). Gráficamente, equivale a imaginar 15 planos en cada uno de los cuales tenemos una matriz de 3X2. Repite la segunda versión del ejercicio trabajando con secciones de arrays.

14. Calcular el producto de dos matrices de N x M y M x L y escribir la matriz producto por pantalla. Las matrices son: $MAT1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$ y

$$MAT2 = \begin{pmatrix} -1 & -4 \\ -2 & -5 \\ -3 & -6 \end{pmatrix}$$

```
PROGRAM cap4_14
INTEGER:: i,j,k
INTEGER,PARAMETER::N=2,M=3,L=2
REAL,DIMENSION(N,M)::mat1
REAL,DIMENSION(M,L)::mat2
REAL,DIMENSION(N,L)::prod
mat1=RESHAPE((/1,1,2,2,3,3/),(/N,M/))
mat2=RESHAPE((/-1,-2,-3,-4,-5,-6/),(/M,L/))
!***CALCULO DE LA MATRIZ PRODUCTO ***
DO  i=1,N
  DO  j=1,L
    prod(i,j)=0
    DO  k=1,M
      prod(i,j)=prod(i,j)+mat1(i,k)*mat2(k,j)
    END DO
  END DO
END DO
!$$$$$$ prod=matmul(mat1,mat2) !funcion intrinseca transformacional
!***VISUAL. DE MATRIZ PRODUCTO,POR FILAS
WRITE(*,*)
WRITE(*,*) 'MATRIZ PRODUCTO'
WRITE(*,*)
DO  i=1,N
  WRITE(*,*) (prod(i,j),j=1,L)
END DO
END PROGRAM cap4_14
```

– Recuerda que la inicialización de matrices se hace por columnas por defecto.

– Notar que la matriz PROD juega el papel de un acumulador, puesto que cada elemento se calcula como una suma de productos. Por esta razón, es imprescindible inicializar esta matriz en el bucle con índice j antes de entrar al bucle con índice k.

15. Calcular la media de NUM números. Usar un vector para almacenarlos, el cual se dimensiona dinámicamente en tiempo de ejecución.

```
PROGRAM cap4_15
INTEGER, ALLOCATABLE, DIMENSION(:) :: vector
INTEGER :: error
REAL:: media
WRITE(*,*) 'DAME NUMERO DE ELEMENTOS DEL VECTOR'
READ(*,*) num
ALLOCATE (vector(1:num),STAT=error)
IF (error /= 0) THEN
  WRITE(*,*) 'NO SUFICIENTE ESPACIO MEMORIA'
  STOP
END IF
WRITE(*,*) 'DAME EL VECTOR'
READ(*,*) vector
media=SUM(vector)/REAL(num)
WRITE(*,*) 'LA MEDIA ES',media
DEALLOCATE (vector,STAT=error)
IF (error /= 0) THEN
  WRITE(*,*) 'ERROR AL LIBERAR LA MEMORIA'
  STOP
END IF
END PROGRAM cap4_15
```

– ALLOCATABLE es el atributo que declara que el array VECTOR será dimensionado dinámicamente. Se trata de un array unidimensional puesto que sólo aparece una vez el carácter “:”.

– ALLOCATE dimensiona dinámicamente el array VECTOR, previamente declarado con el atributo ALLOCATABLE. Establece los límites inferior y superior de su dimensión.

– DEALLOCATE libera la memoria para el array VECTOR, reservada previamente en la sentencia ALLOCATE.

EJERCICIOS PROPUESTOS

1) Programa que pida el grado de un polinomio (inferior a diez), sus coeficientes y un valor de x para evaluarlo.

2) Programa que pida los grados de dos polinomios (inferior a diez y no tienen porqué ser iguales), sus coeficientes y muestre por pantalla el polinomio suma.

3) Programa que pida los grados de dos polinomios (inferior a diez y no tienen porqué ser iguales), sus coeficientes y muestre el polinomio producto.

4) Programa que pida coordenadas cartesianas de los extremos de dos vectores, escriba el ángulo que forman (en grados) y su producto escalar utilizando el coseno del ángulo comprendido entre ambos.

5) Programa que pida coordenadas cartesianas de los extremos de dos vectores, escriba por pantalla el ángulo que forman (en grados) y el módulo de su producto vectorial utilizando el seno del ángulo comprendido entre ambos.

6) Programa que pida las temperaturas de los días de la semana. Posteriormente escribirá los números de los días que superen la temperatura media.

7) Programa que calcule el máximo y el mínimo de un vector de números de dimensión variable, como mucho 10.

8) Programa que lea un número natural por teclado, almacenando cada dígito del mismo en un elemento de un vector y compruebe si es *narcisista*. A saber: un número de n dígitos es narcisista si coincide con la suma de las potencias de orden n de sus dígitos. Ejemplo: 153, pues $1^3+5^3+3^3= 1+125+27=153$. Los números 370 y 371 son también narcisistas. ¿Cómo buscarías todos los números narcisistas de dos y de 4 dígitos?

9) Programa que lea por teclado una matriz 3x3 y muestre su matriz traspuesta.

10) Programa que obtenga la recta de regresión $y=mx+b$ de n parejas de valores (x,y) y el coeficiente de correlación lineal r.

$$m = \frac{n\sum xy - \sum x \sum y}{n\sum x^2 - (\sum x)^2} \quad b = \frac{\sum y \sum x^2 - \sum x \sum xy}{n\sum x^2 - (\sum x)^2}$$

$$r = \frac{n\sum xy - \sum x \sum y}{\sqrt{(n\sum x^2 - (\sum x)^2)(n\sum y^2 - (\sum y)^2)}}$$

11) Programa que lea por teclado una matriz M(3x3) y muestre las matrices M+M y MxM.

12) Programa que calcule el máximo de cada columna par y el mínimo de cada columna impar de una matriz de 5 filas por 6 columnas.

13) Programa que compruebe si una matriz A es idempotente de grado 2, es decir, si $A=A^2$.

14) Programa que calcule la traza de una matriz cuadrada.

15) Programa que compruebe si la columna de una matriz está llena de ceros.

16) Programa que lea la matriz siguiente y compruebe si es un *cuadrado mágico especular*. A saber: una matriz es un cuadrado mágico si las sumas de sus filas, columnas y diagonales son el mismo número. Además, el cuadrado mágico es especular si al invertir los dígitos de cada número (elemento de la matriz) resulta otro cuadrado mágico.

$$\begin{pmatrix} 96 & 64 & 37 & 45 \\ 39 & 43 & 98 & 62 \\ 84 & 76 & 25 & 57 \\ 23 & 59 & 82 & 78 \end{pmatrix}$$

5 PROCEDIMIENTOS

5.1 Diseño descendente

- En programación es muy importante elegir el diseño adecuado a cada problema. Para esta tarea, se utiliza un diseño descendente, *top-down*, que consiste en dividir el problema en subproblemas más pequeños, que se pueden tratar de forma separada.
- Hasta ahora, no había manera de compilar, testear y depurar cada uno de esos subproblemas separadamente hasta construir el programa final combinando adecuadamente las diferentes secciones de código Fortran generadas.
- Sin embargo, existe la posibilidad de *tratar* cada subproblema de un problema más grande de forma independiente. Consiste en codificar cada subproblema en una *unidad de programa*[5] separada llamada *procedimiento externo*. Cada procedimiento externo puede ser compilado, testado y depurado independientemente de los otros procedimientos del programa antes de combinarlos entre sí para dar lugar al programa final.
- En Fortran, hay dos tipos de procedimientos externos: son los *subprogramas funciones* o simplemente *funciones* y las *subrutinas*. Su ejecución se controla desde alguna otra unidad de programa (que puede ser el programa principal u otro procedimiento externo). Ambos tipos de procedimientos externos se estudian en este capítulo.
- Los beneficios del diseño descendente en los programas son:
    - Es mucho más fácil encontrar errores en el código, sobre todo en programas largos.
    - Permite usar procedimientos construidos por otros programadores.
    - Evita cambios indeseables en las variables del programa. Sólo algunas de ellas se transfieren entre las diferentes unidades de programa, aquellas variables que son necesarias para realizar los cálculos previstos. Las demás variables sólo son accesibles en la unidad de programa donde se declaran,

[5] Unidad de programa es una porción de un programa Fortran compilada separadamente. Son unidades de programa los programas principales, las subrutinas y los subprogramas función.

quedando por lo tanto a salvo de cambios imprevistos para el resto de las unidades de programa.

5.2 Funciones

- Hay dos tipos de funciones:
    - *Intrínsecas*: todas aquellas funciones que suministra el propio lenguaje (ya comentadas en el capítulo 1).
    - *definidas por el propio programador* o *subprogramas función*: procedimientos que permiten responder a necesidades particulares del usuario, no proporcionadas por las funciones intrínsecas.
- Las funciones definidas por el programador se usan igual que las funciones intrínsecas, pueden formar parte de expresiones, y por lo tanto, pueden aparecer en todos aquellos lugares donde se puede usar una expresión. Su resultado es un valor numérico, lógico, cadena de caracteres o array.
- La estructura general de un procedimiento función es:

Cabecera de función

Sección de especificaciones

Sección de ejecución

Terminación de la función

A continuación, se explica en detalle la estructura anterior.

- La sintaxis general de la cabecera de función es la siguiente:

[TIPO] FUNCTION nombre_funcion ([*Lista de argumentos formales*])

- Es la primera sentencia no comentada del procedimiento, e identifica esa unidad de programa como procedimiento función.
- TIPO es cualquier tipo Fortran válido relativo a nombre_funcion. Si no aparece TIPO en la cabecera de la función, se debe especificar en la sección de especificaciones.
- nombre_funcion es cualquier identificador Fortran válido.
- *Lista de argumentos formales* es una lista (puede ser vacía) de constantes, variables, arrays o expresiones, separados por comas. Se emplean para pasar información al cuerpo de la

función. Se les llama formales porque no conllevan una reserva de espacio en memoria.

- La sección de especificaciones debe declarar el TIPO de nombre_funcion, si no se ha declarado en la cabecera. Además, debe declarar el tipo de los argumentos formales y las variables *locales*[6] a la función, si las hay.
- La sección de ejecución debe incluir al menos una sentencia de asignación en la que se cargue al nombre de la función el resultado de una expresión del mismo tipo (de ahí la necesidad de declarar el tipo del nombre de la función):

nombre_funcion = expresion

- Finalmente, la terminación de la función tiene la sintaxis general:

END FUNCTION [nombre_funcion]

Una función se invoca escribiendo:

nombre_función ([lista de argumentos actuales])

- formando parte de una *expresión* o en cualquier lugar donde puede aparecer una expresión. Como resultado de la evaluación de la función en sus argumentos actuales se devuelve un valor que es usado para evaluar, a su vez, la expresión de la que forme parte.
- Debe haber una concordancia en el número, tipo y orden de los argumentos actuales que aparecen en la llamada a la función y los argumentos formales que aparecen en la cabecera de la misma. Asimismo, el tipo del nombre de la función debe ser el mismo en la(s) unidad(es) de programa que invoca(n) a la función y el declarado en el propio procedimiento función.
- La ejecución de la llamada ocurre de la siguiente manera:
    - Se evalúan los argumentos actuales que son expresiones.
    - Se *asocian* los argumentos actuales con sus correspondientes argumentos formales.
    - Se ejecuta el cuerpo de la función especificada.
    - Se devuelve el control a la unidad de programa que hizo la llamada, en concreto, a la sentencia donde se invocó a la función, sustituyendo su nombre por el resultado de la ejecución de la función.

---

[6] Variables necesarias para llevar a cabo los cálculos previstos dentro de la función. Son inaccesibles fuera de la misma.

- Una función bien diseñada debe producir un resultado (transferido por el nombre de la misma) a partir de uno o más valores de entrada (transferidos por la lista de argumentos).
- En Fortran, la transferencia de argumentos entre dos unidades de programa cualesquiera se realiza *por dirección*. Así, la asociación entre argumentos actuales y formales significa pasar las direcciones de memoria que ocupan los argumentos actuales al procedimiento llamado, de modo que éste puede leer y escribir en esas direcciones de memoria. Por ejemplo, para escribir en un argumento formal basta ponerle a la izquierda de una sentencia de asignación en el interior de un procedimiento llamado.
- La sintaxis general de los argumentos formales es:

TIPO, [INTENT(*intencion_paso*)] :: arg_formal1[,arg_formal2]...

- donde *intencion_paso* se sustituye por:
    - IN: si el argumento formal es un valor de entrada.
    - OUT: si el argumento formal es un valor de salida.
    - IN OUT: si el argumento formal es un valor de entrada y salida.
- El atributo INTENT ayuda al compilador a encontrar errores por el uso indebido de los argumentos formales de un procedimiento.
- En el caso de un procedimiento función bien diseñado, el atributo de todos los argumentos formales debe ser IN.
- Ejemplo. Sea la función que convierte una temperatura en grados Fahrenheit a grados Celsius:

```
FUNCTION cent (temperatura)
REAL:: cent !declaracion del nombre de la funcion
REAL, INTENT(IN):: temperatura !declaracion del argumento formal
REAL, PARAMETER:: CTE=5./9. !parametro local a la funcion
cent = CTE*(temperatura – 32.)
END FUNCTION cent
```

- Llamadas válidas desde un programa principal son:

```
PROGRAM conversion
REAL:: cent, x,var1,var2
…
var1 = cent (23.)
x=35.75
var2 = -2.*cent (x)+4
WRITE(*,*) var1,var2
…
```

```
END PROGRAM conversion
```

- Se observa que la declaración de CENT es REAL en la unidad de programa principal en concordancia con su tipo en la función y que el argumento actual en ambas llamadas es REAL, en concordancia con su tipo como argumento formal en la función.
- Los nombres de los argumentos actuales y sus correspondientes formales no tienen porqué ser iguales.
- Ejemplo. Sea la función factorial:

```
INTEGER FUNCTION factorial (n)
INTEGER, INTENT(IN) :: n ! declaración del argumento formal
INTEGER :: i ! variable local a la funcion
factorial = 1
DO i = 2, n
factorial = factorial*i
END DO
END FUNCTION factorial
```

- Un programa principal puede ser:

```
PROGRAM ejemplo
INTEGER:: i, n, factorial
! argumento actual y variable local a la unidad de programa principal
WRITE (*,*) 'Teclee un numero entero:'
READ (*,*) i
n = factorial (i)
..
END PROGRAM ejemplo
```

- Las variables *n, i* del programa principal no tienen nada que ver con las variables *n, i* de la función. Puesto que las variables locales a una unidad de programa son visibles únicamente en el interior de la misma, se pueden usar los mismos nombres para variables locales en unidades de programa diferentes, sin problemas de conflictos.

5.3 Subrutinas

- Son procedimientos más generales que las funciones, aunque comparten casi todas sus características. Pueden retornar más de un valor, o no retornar nada en absoluto. Reciben los valores de entrada y devuelven los valores de salida a través de su lista de argumentos.
- La estructura general de una subrutina es idéntica a la de una función:

Cabecera de subrutina

Sección de especificaciones

Sección ejecutable

Terminación de subrutina

- donde la sintaxis general de la cabecera de subrutina es:

SUBROUTINE nombre_subrutina ([Lista de argumentos formales])

- y la sintaxis general de la terminación de subrutina es:

END SUBROUTINE [nombre_subrutina]

- La sección de especificaciones incluye la declaración de los tipos de los argumentos formales con su atributo INTENT correspondiente a su intención de uso y la de las variables locales a la subrutina, si las hay.
- La sintaxis general de llamada a una subrutina desde cualquier unidad de programa es:

CALL nombre_subrutina ([lista de argumentos actuales])

- La ejecución de la llamada ocurre de la siguiente manera:
    - Se evalúan los argumentos actuales que son expresiones.
    - Se *asocian* los argumentos actuales con sus correspondientes argumentos formales. El paso de los argumentos se realiza por dirección.
    - Se ejecuta el cuerpo de la subrutina especificada.
    - Se devuelve el control a la unidad de programa que hizo la llamada, en concreto, a la sentencia siguiente a la sentencia CALL.
    - La subrutina NO devuelve ningún valor a través de su nombre, sino que son los argumentos los encargados de realizar las transferencias de resultados.
- Debe haber concordancia en el número, tipo y orden de los argumentos actuales y sus correspondientes argumentos formales, al igual que en las funciones.
- Ejemplo. Sea la subrutina para convertir grados, minutos y segundos a grados decimales:

```
SUBROUTINE convierte (grados, minutos, segundos, grads)
INTEGER, INTENT(IN) :: grados, minutos, segundos
REAL, INTENT(OUT)  :: grads
grads= REAL (grados) + REAL (minutos)/60. + REAL (segundos)
/3600
END SUBROUTINE convierte
```

- Llamadas válidas desde un programa principal son:

```
PROGRAM principal
INTEGER :: g,m,s
REAL:: n,gd
…
CALL convierte (30, 45, 2, n) ! llamada 1
WRITE (*,*) '30 grados, 45 minutos, 2 segundos equivalen a'
WRITE (*,*) n,'grados decimales'
CALL convierte (g, m, s, gd) ! llamada 2
WRITE (*,*) g,'grados', m,'minutos', s,'segundos equivalen a'
WRITE (*,*) gd,'grados decimales'
…
END PROGRAM principal
```

5.4 Transferencia de arrays a procedimientos

- Cuando la llamada a un procedimiento incluye el nombre de un array en un argumento actual, se transfiere la dirección de memoria del primer elemento del mismo. De esta manera, el procedimiento es capaz de acceder al array, pues todos sus elementos ocupan direcciones de memoria consecutivas.
- Además, el procedimiento debe conocer el tamaño del array, en concreto, los límites de los índices de cada dimensión para que las operaciones efectuadas en el cuerpo del procedimiento se realicen sobre elementos permitidos. Hay dos formas de hacerlo:
    - array formal con perfil explícito. Consiste en pasar la extensión de cada dimensión del array en la lista de argumentos y usarlas en la declaración del array formal en el procedimiento. Esta forma permite operaciones con arrays completos y subconjuntos de arrays dentro del procedimiento.
    - Ejemplo:

```
…
CALL proced (matriz, d1, d2, resul) !llamada a subrutina
…
SUBROUTINE proced (matriz, d1, d2, resul)
INTEGER, INTENT(IN):: d1,d2
INTEGER, INTENT(IN), DIMENSION(d1,d2)::matriz ! perfil explícito
INTEGER, INTENT(OUT):: resul
```

- array formal con perfil asumido. La declaración de un array formal de este tipo usa dos puntos : para cada índice del

mismo. Permite operaciones con arrays completos y subconjuntos de arrays dentro del procedimiento. El procedimiento debe tener interfaz explícita, concepto que se estudiará más adelante.

- Ejemplo:

```
MODULE mod1
CONTAINS
SUBROUTINE sub1 (matriz)
INTEGER, INTENT(INOUT), DIMENSION(:,:)::matriz ! perfil asumido
…
END SUBROUTINE sub1
END MODULE mod1
```

5.5 Compartir datos con módulos

- Hasta ahora, un programa Fortran intercambia datos entre sus distintas unidades de programa (principal, función, subrutina) a través de las listas de argumentos.
- Además de las listas de argumentos, un programa Fortran puede intercambiar datos a través de *módulos*.
- Un módulo es una unidad de programa compilada por separado del resto que contiene, al menos, las declaraciones e inicializaciones necesarias de los datos que se quieren compartir entre las unidades de programa.
- La sintaxis general de un módulo es:

```
MODULE nombre_modulo
[SAVE]
Declaración e inicialización datos compartidos   Cuerpo del módulo
END MODULE nombre_modulo
```

- La sentencia SAVE es útil para preservar los valores de los datos del módulo cuando éste se comparte entre varias unidades de programa.
- Para poder usar los datos de módulos en una unidad de programa, escribir la sentencia:

```
USE nombre_modulo1[,nombre_modulo2],…
```

- como primera sentencia no comentada inmediatamente después del cabecero de la unidad de programa que quiere usarlos.
- Los módulos de datos son útiles cuando se necesita compartir grandes cantidades de datos entre muchas unidades de programa, pero manteniéndolos invisibles para las demás.

- Ejemplo. Escribir un módulo de datos que comparta dos vectores con valores iniciales v1(1 1 1 1 1) y v2(10 11 12 13 14) y una matriz *m* entre el programa principal y una subrutina *sub*. El programa principal debe calcular el vector suma de v1 y v2 y la subrutina debe volcar el vector suma en la primera columna de la matriz y el vector v2 en la segunda columna.

```
MODULE comparte_datos
IMPLICIT NONE
SAVE
INTEGER, PARAMETER:: TM=5
INTEGER:: i
INTEGER, DIMENSION(TM) :: v1=1,v2=(/ (i, i=10,14) /)
INTEGER, DIMENSION(TM,2) :: m
END MODULE comparte_datos

PROGRAM principal
USE comparte_datos
IMPLICIT NONE
INTEGER :: j
WRITE(*,*) 'v1',v1
WRITE(*,*) 'v2',v2
v1=v1+v2
WRITE(*,*) 'v1',v1
CALL sub
WRITE(*,*) 'm'
DO i=1,TM
WRITE(*,*) (m(i,j),j=1,2)
END DO
END PROGRAM principal

SUBROUTINE sub
USE comparte_datos
IMPLICIT NONE
m( : , 1 ) = v1
m( : , 2 ) = v2
END SUBROUTINE sub
```

5.6 Procedimientos módulo

- Además de datos, un modulo puede contener procedimientos (subrutinas y/o funciones), que se denominan entonces *procedimientos módulo*.
- La sintaxis general de un procedimiento módulo es:

```
MODULE nombre_modulo
[SAVE]
Declaración e inicialización datos compartidos
CONTAINS
Estructura general procedimiento1
[Estructura general procedimiento2]
…
END MODULE nombre_modulo
```

- Como ocurre con los módulos de datos, para hacer accesibles procedimientos módulos a una unidad de programa, escribir:

```
USE nombre_modulo1[,nombre_modulo2]…
```

- como la primera sentencia no comentada, inmediatamente después del cabecero de la unidad de programa que quiere usarlo.
- Un procedimiento contenido en un módulo se dice que tiene una *interfaz explícita*, pues el compilador conoce todos los detalles de su lista de argumentos. Como consecuencia, cuando se usa el módulo en cualquier unidad de programa, el compilador chequea la concordancia de número, tipo y orden entre las listas de argumentos actuales y sus correspondientes formales, así como usos indebidos de los últimos según el valor del atributo INTENT.
- Por contraposición, un procedimiento externo fuera de un módulo se dice que tiene una *interfaz implícita*. El compilador desconoce los detalles de las listas de argumentos y, por tanto, no puede chequear errores de concordancias en las mismas. Es responsabilidad del programador encargarse de chequearlo.
- Ejemplo. Sumar dos números enteros usando un procedimiento módulo.

```
MODULE mod1
CONTAINS
SUBROUTINE sub1 (a, b, sumar)
IMPLICIT NONE
INTEGER, INTENT(IN):: a,b
INTEGER, INTENT(OUT):: sumar
sumar=a+b
```

```
END SUBROUTINE sub1
END MODULE mod1

PROGRAM principal
USE mod1
IMPLICIT NONE
INTEGER::x,y,resul
WRITE(*,*) 'dame dos numeros'
READ(*,*) x,y
CALL sub1(x,y,resul)
WRITE(*,*) 'la suma es',resul
END PROGRAM principal
```

- Comprobar que si se declaran los argumentos actuales del tipo REAL, el compilador encuentra el error de concordancia de tipos. Sin embargo, si se repite el ejercicio con la subrutina sub1 como procedimiento externo, el compilador no encuentra errores.

5.7 Procedimientos como argumentos

- Los argumentos actuales de un procedimiento pueden ser nombres de subrutinas o funciones definidas por el programador. Como el paso de argumentos se realiza por dirección, en este caso, se pasa la dirección de memoria de comienzo del procedimiento.
    - Si el argumento actual es una función, necesita el atributo EXTERNAL en su sentencia de declaración de tipo, tanto en el procedimiento de llamada como en el procedimiento llamado. La sintaxis general es:

```
TIPO, EXTERNAL:: nombre_funcion
```

- Ejemplo:

```
PROGRAM principal
IMPLICIT NONE
INTEGER:: a=5, b=7
REAL, EXTERNAL:: fun
REAL:: x
CALL sub (fun, a, b, x)
WRITE(*,*) 'resultado',x
END PROGRAM principal

SUBROUTINE sub (f, a,b,res)
```

```
IMPLICIT NONE
REAL, EXTERNAL:: f
INTEGER, INTENT(IN):: a,b
REAL, INTENT(OUT):: res
res=f(a)+b**2
END SUBROUTINE sub

REAL FUNCTION fun(a)
INTEGER, INTENT(IN):: a
fun=(2.*a-5.)/7.
END FUNCTION fun
```

- Si el argumento actual es una subrutina, es necesario escribir una sentencia EXTERNAL, tanto en el procedimiento de llamada como en el procedimiento llamado. La sintaxis general es:

```
EXTERNAL:: nombre_subrutina
```

5.8 Atributo y sentencia SAVE

- Cada vez que se sale de un procedimiento, los valores de sus variables locales se pierden, a menos que se guarden poniendo el atributo SAVE en las sentencias de declaración de tipo de aquellas variables que se quieren guardar. La sintaxis general es:

```
TIPO, SAVE:: variable_local1[, variable_local2]...
```

- Para guardar todas las variables locales a un procedimiento escribir simplemente SAVE en una sentencia ubicada en la sección de especificaciones del procedimiento.
- Automáticamente, toda variable local inicializada en su sentencia de declaración se guarda.
- Ejemplo:

```
INTEGER FUNCTION fun(N)
INTEGER INTENT(IN):: N
INTEGER, SAVE:: cuenta
cuenta=0
cuenta = cuenta + 1! Cuenta las veces que se llama la función
…
END FUNCTION fun
```

5.9 Procedimientos internos

- Hasta ahora, se han estudiado dos tipos de procedimientos: los procedimientos externos y los procedimientos módulo. Además, existe un tercer tipo de procedimientos, los llamados procedimientos internos.
- Un procedimiento interno es un procedimiento completamente contenido dentro de otra unidad de programa, llamada *anfitrión* o *host*. El procedimiento interno se compila junto con su anfitrión y sólo es accesible desde él. Debe escribirse a continuación de la última sentencia ejecutable del anfitrión, precedido por una sentencia CONTAINS.
- La estructura general de una unidad de programa que contiene un procedimiento interno es:

Cabecero de unidad de programa

Sección de especificaciones

Sección ejecutable

CONTAINS

Procedimiento interno

Fin de unidad de programa

- Un procedimiento interno tiene acceso a todos los datos definidos por su anfitrión, salvo aquellos datos que tengan el mismo nombre en ambos. Los procedimientos internos se usan para realizar manipulaciones de bajo nivel repetidamente como parte de una solución.

5.10 Procedimientos recursivos

- Un procedimiento es recursivo cuando puede llamarse a sí mismo directa o indirectamente las veces que se desee.
- Para declarar un procedimiento como recursivo, añadir la palabra clave RECURSIVE a la sentencia cabecero del procedimiento.
  - La sintaxis general para el caso de un procedimiento subrutina es:

RECURSIVE SUBROUTINE nombre_subrutina [(Lista arg formales)]

  - Además de esto, en el caso de una función recursiva, Fortran permite especificar dos nombres distintos para invocar a la función y para devolver su resultado, con el fin de evitar confusión entre los dos usos del nombre de la función. En particular, el nombre de la función se usa para invocar a la función, mientras que el resultado de la misma se devuelve a través de un argumento formal especial especificado entre paréntesis a la derecha de una cláusula RESULT en la propia

sentencia cabecero de la función. La sintaxis general de una función recursiva es:

RECURSIVE FUNCTION nom_fun ([List arg form]) RESULT (resultado)

- Con esta forma, la sección de especificaciones no incluirá declaración de tipo del nombre de la función; en su lugar, debe declararse el tipo de *resultado*.

- Ejemplo. Calcular el factorial de un número usando una subrutina recursiva.

```
RECURSIVE SUBROUTINE factorial (n, resultado)
INTEGER, INTENT(IN):: n
INTEGER, INTENT(OUT):: resultado
INTEGER :: temp
IF (n>=1) THEN
CALL factorial (n-1, temp)
resultado=n*temp
ELSE
resultado=1
ENDIF
END SUBROUTINE factorial
```

- Ejemplo. Lo mismo pero usando una función recursiva.

```
RECURSIVE FUNCTION factorial (n) RESULT (resultado)
INTEGER, INTENT(IN):: n
INTEGER:: resultado
IF (n>=1) THEN
resultado=n*factorial(n-1)
ELSE
resultado=1
ENDIF
END FUNCTION factorial
```

5.11 Argumentos opcionales y cambios de orden

- Hasta ahora se ha dicho que la lista de argumentos actuales debe coincidir en orden, tipo y número con su lista de argumentos formales.
- Sin embargo, es posible cambiar el orden de los argumentos actuales y/o especificar sólo algunos de ellos pero no todos según

interese en cada llamada al procedimiento, siempre que tal procedimiento tenga interfaz explícita.

- Para cambiar el orden de los argumentos actuales en la llamada a un procedimiento, cada argumento actual se debe especificar en la llamada con la sintaxis general:

nombre_argumento_formal= argumento_actual

- Ejemplo. Sea la función con interfaz explícita:

```
MODULE mod
CONTAINS
FUNCTION calcula (primero, segundo, tercero)
…
END FUNCTION calcula
END MODULE mod
```

Llamadas idénticas que producen los mismos resultados son:

```
WRITE(*,*) calcula (5, 3, 7)
WRITE(*,*) calcula (primero=5, segundo=3, tercero=7)
WRITE(*,*) calcula (segundo=3, primero=5, tercero=7)
WRITE(*,*) calcula (5, tercero=7, segundo=3)
```

- El cambio de orden de los argumentos actuales constituye una complicación por sí solo sin interés. Su utilidad radica en combinar el cambio de orden con el hecho de que algunos argumentos sean opcionales.
- Un argumento formal es opcional cuando no necesita siempre estar presente cuando se llama al procedimiento que lo incluye. Para definir un argumento formal como opcional hay que añadir en su declaración de tipo el atributo OPTIONAL.
- Ejemplo:

```
MODULE mod
CONTAINS
SUBROUTINE sub (arg1, arg2, arg3)
INTEGER, INTENT(IN), OPTIONAL:: arg1
INTEGER, INTENT(IN):: arg2
INTEGER, INTENT(OUT):: arg3
…
END SUBROUTINE sub
END MODULE mod
```

- Los argumentos formales opcionales sólo pueden declararse en procedimientos con interfaz explícita. Cuando están presentes en la

llamada al procedimiento, éste los usa, sino están presentes, el procedimiento funciona sin ellos. La forma de testar si el argumento opcional debe usarse en el procedimiento o no, es con la función intrínseca lógica PRESENT. Para ello, en el ejemplo anterior, el cuerpo de la subrutina *sub* debe incluir la estructura condicional:

IF (PRESENT(arg1)) THEN

Acciones a realizar cuando arg1 está presente

ELSE

Acciones a realizar cuando arg1 está ausente

ENDIF

- Al llamar a un procedimiento con argumentos opcionales, pueden ocurrir varias situaciones:
    - Si están presentes los argumentos opcionales en el orden adecuado, la llamada tiene la forma habitual: CALL sub (2, 9, 0)
    - Si están ausentes los argumentos opcionales y se respeta el orden de los mismos, la llamada es: CALL sub (9, 0)
    - Si están ausentes los argumentos opcionales y hay cambios de orden en los mismos, la llamada es: CALL sub (arg3=0,arg2=9)

EJERCICIOS RESUELTOS

Objetivos:

Aprender a dividir un programa Fortran en subprogramas FUNCTION o SUBROUTINE.

Se muestran los aspectos básicos de los procedimientos, relativos a las formas de llamada a un procedimiento y de transferencias de datos entre distintas unidades de programa. También se ven algunos aspectos avanzados que aportan mayor flexibilidad y control sobre los procedimientos.

En la práctica, todos los ejercicios que se muestran en este capítulo se pueden realizar de dos formas en un entorno de programación FORTRAN: escribiendo todas las unidades de programa en el mismo archivo Fortran, unas a continuación de otras, o bien, creando un proyecto y añadiendo un archivo por cada unidad de programa, que se compilará por separado. En este caso, es recomendable usar procedimientos módulo, según se ha estudiado en la teoría.

1. Sumar dos números enteros usando una función.

```
PROGRAM cap5_1
IMPLICIT NONE
INTEGER :: a,b,suma
WRITE(*,*) 'DAME 2 NUMEROS'
READ(*,*) a,b
WRITE(*,*) 'LA SUMA ES',suma(a,b)
END PROGRAM cap5_1

INTEGER FUNCTION suma(x,y)
IMPLICIT NONE
INTEGER, INTENT(IN) :: x,y
suma=x+y
END FUNCTION suma
```

– La llamada a la función SUMA se realiza en una sentencia WRITE en el programa principal.

– La ejecución de la llamada ocurre de la siguiente manera:

– 1. Se asocian los argumentos actuales con sus correspondientes argumentos formales. Es decir, a (variable entera) se asocia con x (del mismo tipo que a) y b (variable entera) con y (del mismo tipo que b).

– 2. Se ejecuta el cuerpo de la función SUMA, lo cual requiere cargar en SUMA el resultado de la adición de x e y.

– 3. Se devuelve el control a la sentencia WRITE del programa principal.

2. Calcular el número combinatorio $\binom{m}{n}$ sabiendo que m debe ser mayor o igual que n.

```
PROGRAM cap5_2
IMPLICIT NONE
INTEGER :: n,m
INTEGER :: fact
REAL :: resul
WRITE(*,*) 'DAME 2 NUMEROS'
READ(*,*) m,n
```

```
IF (m<n) THEN
  WRITE(*,*) 'NO SE PUEDE'
ELSE
  resul=fact(m)/(fact(n)*fact(m-n))
  WRITE(*,*) 'RESULTADO',resul
END IF
END PROGRAM cap5_2

FUNCTION fact(x)
IMPLICIT NONE
INTEGER, INTENT(IN) :: x
INTEGER :: i,fact
fact=1
DO i=1,x
  fact=fact*i
END DO
END FUNCTION fact
```

– En este ejercicio se realizan tres llamadas a la función FACT, que calcula el factorial de un número. Estas llamadas forman parte de una expresión aritmética. En el caso de FACT(m-n), en primer lugar se evalúa la resta en el argumento actual, a continuación se asocia con el argumento formal x definido en la función y se ejecuta la función.

– Obviamente, cuanto mayor es el número de llamadas a una función, mayor es la motivación de codificarla por separado del programa principal.

3. Calcular $\sum_{i=1}^{n}\frac{1}{i!}$ siendo n leído por teclado. Usar una función para calcular el factorial (i!). El programa se ejecuta tantas veces como el usuario quiera, por ejemplo, mientras se teclee la letra 'S'.

```
PROGRAM cap5_3
IMPLICIT NONE
CHARACTER (LEN=1) :: seguir
INTEGER :: fact,i,n
REAL :: sumator
DO
  WRITE(*,*) 'NUMERO DE ELEMENTOS DEL SUMATORIO?'
```

```
  READ(*,*) n
  sumator=0
  DO i=1,n
    sumator=sumator+1./fact(i)
  END DO
  WRITE(*,*) 'EL RESULTADO ES:',sumator
  WRITE(*,*) 'DESEA CONTINUAR (S/N)?'
  READ(*,*) seguir
  IF (seguir /= 'S') EXIT
END DO
END PROGRAM cap5_3

FUNCTION fact(x)
IMPLICIT NONE
INTEGER, INTENT(IN) :: x
INTEGER :: fact, i
fact=1
DO i=1,x
  fact=fact*i
END DO
END FUNCTION fact
```

– La llamada a la función FACT se realiza, en este caso, desde el interior de un bucle, y forma parte de una expresión aritmética. La ejecución de la función se realiza tantas veces como valores toma el índice del bucle.

4. Calcular la media de cinco números (leídos por teclado) utilizando para ello una función.

```
PROGRAM cap5_4
IMPLICIT NONE
REAL :: media
REAL, DIMENSION(5):: vector
INTEGER :: i
DO i=1,5
  WRITE(*,*) 'DAME COMPONENTE',i,'DEL VECTOR'
  READ(*,*) vector(i)
END DO
WRITE(*,*) 'LA MEDIA ES:',media(vector,5)
```

```
END PROGRAM cap5_4

REAL FUNCTION media(xx,d1)
IMPLICIT NONE
INTEGER, INTENT(IN) :: d1
REAL,DIMENSION(d1),INTENT(IN):: xx!perfil explicito
REAL :: suma
INTEGER :: i
suma=0
DO i=1,d1
  suma=suma+xx(i)
END DO
media=suma/d1
END FUNCTION media
```

- Notar que para transferir un array (vector o matriz) a un procedimiento como argumento basta escribir el nombre del mismo. El array formal XX se ha declarado usando la dimensión transferida en el argumento d1. ¿Qué cambios hay que hacer en el programa para usar perfil asumido en el array formal xx?
- El resultado de la media se devuelve al programa principal a través del nombre de la función.

5. Lo mismo que en el ejercicio CAP5_1 pero usando una subrutina.

```
PROGRAM cap5_5
IMPLICIT NONE
INTEGER :: a,b,s
WRITE(*,*) 'DAME 2 NUMEROS'
READ(*,*) a,b
CALL suma(a,b,s)
WRITE(*,*) 'LA SUMA ES',s
END PROGRAM cap5_5

SUBROUTINE suma(x,y,z)
IMPLICIT NONE
INTEGER, INTENT(IN) :: x
INTEGER, INTENT(IN) :: y
INTEGER, INTENT(OUT) :: z
```

```
z=x+y
END SUBROUTINE suma
```

– Notar la forma de llamar a la subrutina usando la sentencia CALL.

– La ejecución de la llamada ocurre de la misma manera que en el caso de una función.

– El resultado de la suma de las dos variables se transfiere al programa principal a través del argumento Z.

– ¿En qué tipo de problemas usarías subprogramas función y en cuáles subprogramas subrutina? ¿Es indiferente?

6. Intercambiar los valores de dos variables enteras. Usar una subrutina para realizar el intercambio.

```
PROGRAM cap5_6
IMPLICIT NONE
INTEGER :: a=5,b=10
WRITE(*,*) 'ANTES DEL CAMBIO'
WRITE(*,*)' A= ',a,' B= ',b
CALL cambia(a,b)
WRITE(*,*) 'DESPUES DEL CAMBIO EN PRINCIPAL'
WRITE(*,*) ' A= ',a,' B= ',b
END PROGRAM cap5_6

SUBROUTINE cambia(x,y)
IMPLICIT NONE
INTEGER, INTENT(IN OUT) :: x
INTEGER, INTENT(IN OUT) :: y
INTEGER :: aux
aux=x
x=y
y=aux
WRITE(*,*) 'DESPUES DEL CAMBIO EN SUBRUTINA'
WRITE(*,*) ' X= ',x,' Y= ',y
END SUBROUTINE cambia
```

– En este programa vemos que el cambio de valores de los argumentos formales x e y se refleja también en los argumentos actuales a y b.

7. Lo mismo que en el ejercicio CAP5_4 pero usando una subrutina.

```
PROGRAM cap5_7
IMPLICIT NONE
REAL :: resul
REAL, DIMENSION(5)::vector
INTEGER :: i
WRITE(*,*) 'DAME VECTOR'
READ(*,*) (vector(i),i=1,5)
CALL media(vector,resul,5)
WRITE(*,*) 'LA MEDIA ES:',resul
END PROGRAM cap5_7

SUBROUTINE media(num,solu,d1)
IMPLICIT NONE
INTEGER, INTENT(IN):: d1
REAL, DIMENSION(d1),INTENT(IN) :: num !perfil explicito
REAL, INTENT(OUT) :: solu
REAL :: suma
INTEGER :: i
suma=0
DO i=1,d1
  suma=suma+num(i)
END DO
solu=suma/d1
END SUBROUTINE media
```

– En la llamada a la subrutina, la dirección de memoria del primer elemento de vector se pasa al argumento formal num y la solución calculada, media de los cinco números de ese array, se pasa a su vez al programa principal a través del argumento formal solu.

8. Calcular la cantidad de números positivos, negativos y ceros que hay en una matriz, sabiendo que el nº de filas y columnas es como máximo 10. Usar una subrutina para leer el número de filas y columnas de la matriz, así como sus elementos y otra subrutina para calcular el número de positivos, negativos y ceros que tiene la matriz.

```
PROGRAM cap5_8
```

```
IMPLICIT NONE
INTEGER, PARAMETER:: TM=10
INTEGER, DIMENSION(TM,TM) :: mat
INTEGER:: fila,columna,pos,neg,ceros
CALL leer(mat,fila,columna,TM)
CALL cuenta(mat,fila,columna,pos,neg,ceros)
WRITE(*,*) 'EL NUMERO DE POSITIVOS ES:',pos
WRITE(*,*) 'EL NUMERO DE NEGATIVOS ES:',neg
WRITE(*,*) 'EL NUMERO DE CEROS ES:',ceros
END PROGRAM cap5_8

SUBROUTINE leer(mat,fil,col,TM)
IMPLICIT NONE
INTEGER, INTENT(IN):: TM
INTEGER, DIMENSION(TM,TM),INTENT(OUT):: mat
INTEGER, INTENT(OUT)   :: fil,col
INTEGER :: k,j

DO
  WRITE(*,*) '¿Nº DE FILAS?'
  READ(*,*) fil
  WRITE(*,*)'¿Nº DE COLUMNAS?'
  READ(*,*) col
  IF (fil<=10 .AND. col<=10) EXIT
END DO
DO  k=1,fil
  DO  j=1,col
    WRITE(*,*) 'ELEMENTO',k,j,'DE LA MATRIZ'
    READ(*,*) mat(k,j)
  END DO
END DO
END SUBROUTINE leer

SUBROUTINE cuenta(mat,fil,col,pos,neg,cer)
IMPLICIT NONE
INTEGER, INTENT(IN):: fil,col
INTEGER, DIMENSION(10,10), INTENT(IN):: mat
INTEGER, INTENT(OUT):: pos,neg,cer
INTEGER :: k,j
pos=0;neg=0;cer=0;
```

```
DO  k=1,fil
  DO  j=1,col
    IF (mat(k,j) < 0) THEN
      neg=neg+1
    ELSE IF (mat(k,j) == 0) THEN
        cer=cer+1
    ELSE
        pos=pos+1
    END IF
  END DO
END DO
END SUBROUTINE cuenta
```

– Un bucle controla que se introduce el tamaño permitido para la matriz, es decir, como máximo 10X10. Notar que el tamaño de la matriz debe ser en todas las unidades de programa igual 10X10, puesto que la transferencia del mismo se realiza por columnas.

– Se usan tres contadores para calcular las cantidades pedidas.

9. Generar aleatoriamente un número de 1 a 100. Se trata de adivinar qué número es, con sucesivos intentos.

```
PROGRAM cap5_9
IMPLICIT NONE
INTEGER :: suerte,n2,n,intento
n=suerte()
WRITE(*,*) "SE HA GENERADO UN NUMERO ENTRE 1 Y 100"
WRITE(*,*) "INTENTA ADIVINARLO"
  intento=0
  DO
    WRITE(*,*) 'DAME UN NUMERO'
    READ(*,*) n2
    intento=intento+1
    IF (n2==n) THEN
      WRITE(*,*)'ACERTASTES!'
      WRITE(*,*) 'HAS NECESITADO',intento,'     INTENTOS!'
      EXIT
    END IF
    CALL pista(n,n2)
```

```
  END DO
END PROGRAM cap5_9

INTEGER FUNCTION suerte()
IMPLICIT NONE
REAL :: n
CALL random_seed()
CALL random_number(n)
n=n*100
suerte =n+1
END FUNCTION suerte

SUBROUTINE pista(n,n2)
INTEGER, INTENT(IN) :: n,n2
IF (n<n2) THEN
  WRITE(*,*) "LO QUE BUSCAS ES MENOR"
ELSE
  WRITE(*,*) "LO QUE BUSCAS ES MAYOR"
END IF
END SUBROUTINE pista
```

– RANDOM_SEED() y RANDOM_NUMBER(arg) son dos subrutinas intrínsecas FORTRAN. La primera inicializa el procedimiento aleatorio y la segunda genera un número aleatorio real arg tal que 0<=arg<1.

10. Leer por teclado el coeficiente de convección (h), la diferencia de temperatura (dT), el radio y la altura de un cilindro y calcular la pérdida de calor, según la fórmula q=hAdT, donde A es el área del cilindro.

```
PROGRAM cap5_10
IMPLICIT NONE
REAL :: h,dt,r,al,area
WRITE(*,*) 'COEFICIENTE DE CONVECCION'
READ(*,*) h
WRITE(*,*) 'DIFERENCIA DE TEMPERATURA'
READ(*,*) dt
WRITE(*,*) 'RADIO Y ALTURA'
READ(*,*) r,al
```

```
WRITE(*,*) 'PERDIDA DE CALOR: ',h*area(r,al)*dt
END PROGRAM cap5_10

REAL FUNCTION area(r,al)
IMPLICIT NONE
REAL, INTENT(IN) :: r,al
REAL :: pi,circulo
pi=2*ASIN(1.)
area=(2*pi*r*al)+2*circulo(r,pi)
END FUNCTION area

REAL FUNCTION circulo(rad,pi)
IMPLICIT NONE
REAL, INTENT(IN) :: rad,pi
circulo=pi*rad**2
END FUNCTION circulo
```

– En este ejercicio se ve cómo se realiza la llamada a una función dentro de otra función. En concreto, la función *area* realiza la llamada a la función *circulo* para completar el cálculo del área del cilindro que se requiere.

11. Cargar por teclado la temperatura T de 25 puntos del espacio con coordenadas (X, Y) en un instante de tiempo dado. Se pide:

– Visualizar la temperatura T de un punto del espacio (X, Y) solicitado por el usuario por teclado.

– Visualizar los puntos del espacio (X, Y), si los hay, que tienen el mismo valor de temperatura T, dada por el usuario por teclado.

– Calcular la mayor (en valor absoluto) de las diferencias de las temperaturas respecto de la temperatura media.

– Usar programación modular en la elaboración del programa. Unidades: (X, Y, T) = (m, m, C).

```
PROGRAM cap5_11
IMPLICIT NONE
INTEGER, PARAMETER:: d1=25,d2=3
REAL :: x,y,t,buscat,difm
REAL, DIMENSION(d1,d2):: mat
REAL, EXTERNAL:: media
```

```
CALL lectura(mat,d1,d2)
WRITE(*,*) 'DAME UN PUNTO'
READ(*,*) x,y
WRITE(*,*) 'TEMPERATURA EN ESE PUNTO ES',buscat(x,y,mat,d1,d2)
WRITE(*,*) 'DAME UNA TEMPERATURA'
READ(*,*) t
CALL puntos(mat,t,d1,d2)
WRITE(*,*)'DESV MAX RESPECTO TEMP. MEDIA', difm(mat,media,d1,d2)
! LA FUNCIÓN MEDIA ES UN ARGUMENTO ACTUAL
END PROGRAM cap5_11

SUBROUTINE lectura(mat,d1,d2)
IMPLICIT NONE
INTEGER, INTENT(IN)::d1,d2
REAL, DIMENSION(d1,d2),INTENT(OUT):: mat
INTEGER:: i,j
DO  i=1,d1
  DO  j=1,d2
    WRITE(*,*) 'DAME X,Y Y T DEL PUNTO',i
    READ(*,*) mat(i,j)
  END DO
END DO
END SUBROUTINE lectura

REAL FUNCTION buscat(x,y,mat,d1,d2)
IMPLICIT NONE
INTEGER, INTENT(IN)::d1,d2
REAL, INTENT(IN) :: x,y
REAL, DIMENSION(d1,d2),INTENT(IN) :: mat
INTEGER:: i
DO i=1,d1
  IF (mat(i,1)==x .AND. mat(i,2)==y) THEN
    buscat=mat(i,d2)
  END IF
END DO
END FUNCTION buscat

SUBROUTINE puntos(mat,t,d1,d2)
IMPLICIT NONE
INTEGER, INTENT(IN)::d1,d2
```

```
REAL, DIMENSION(d1,d2),INTENT(IN):: mat
REAL, INTENT(IN) :: t
INTEGER :: sw,i
sw=0
DO i=1,d1
  IF (mat(i,d2) == t) THEN
    sw=1
    WRITE(*,*) 'COORDENADAS DE ESA TEMPER.',mat(i,1),mat(i,2)
  END IF
END DO
IF(sw == 0) WRITE(*,*) 'NO REGISTRADA ESA TEMPER.EN EL ARRAY'
END SUBROUTINE puntos

REAL FUNCTION difm(x,f,d1,d2)
IMPLICIT NONE
INTEGER, INTENT(IN)::d1,d2
REAL, DIMENSION(d1,d2), INTENT(INOUT):: x!ENTRA AQUI PERO SALE
A F
REAL, EXTERNAL:: f
REAL :: y,dif
INTEGER:: i
y=f(x,d1,d2)
difm=ABS(x(1,d2)-y)
DO i=2,d1
  dif=ABS(x(i,d2)-y)
  IF (dif > difm) THEN
    difm=dif
  END IF
END DO
END FUNCTION difm

REAL FUNCTION media(mat,d1,d2)
IMPLICIT NONE
INTEGER, INTENT(IN)::d1,d2
REAL, DIMENSION(d1,d2),INTENT(IN) :: mat
INTEGER:: i
media=0
DO i=1,d1
  media=media+mat(i,d2)
```

```
END DO
media=media/d1
END FUNCTION media
```

- La función *media* es un argumento actual de la función *difm*. Para especificar este hecho en Fortran se usa el atributo EXTERNAL. Repasa la sección 5.7.
- La utilidad de la función *media* aumentaría si la función *difm* calculara las medias de varios conjuntos de temperaturas.

EJERCICIOS PROPUESTOS

1) Programa que pida dos números naturales y use una función lógica para saber si ambos son *cuadrones pares* o no. A saber: dos números son cuadrones pares si al sumarlos y restarlos se obtienen cuadrados perfectos. Ejemplo: 10 y 26 son cuadrones pares pues: 10+26 =36 (cuadrado perfecto) y 26-10 = 16 (cuadrado perfecto).

2) Modifica el ejercicio anterior para obtener todos los números cuadrones pares hasta 1000.

3) Programa que lee por teclado una matriz 3x3 y calcula su determinante. Utilizar la función siguiente para calcular adjuntos:

```
INTEGER FUNCTION adjto (a,b,c,d)
INTEGER, INTENT(IN):: a,b,c,d
adjto=a*d-b*c
END FUNCTION adjto
```

4) Programa que lee por teclado una matriz 3x3 y calcula su determinante. Utilizar la subrutina siguiente para calcular adjuntos:

```
SUBROUTINE adjto (a,b,c,d,det2)
INTEGER, INTENT(IN):: a,b,c,d
INTEGER, INTENT(OUT):: det2
det2=a*d-b*c
END SUBROUTINE adjto
```

5) Programa que pida 5 números por teclado y averigüe si son primos o no utilizando el algoritmo de Wilson. A saber: un número K es primo si (K-1)!+1 es divisible entre K. Utilizar una función que devuelva a través de su nombre los valores .TRUE: si el número dado es primo y .FALSE. si no lo es.

6) Programa que pida por teclado una matriz cuadrada de 4X4 y calcule su traza y la suma de los elementos por encima y por debajo de la diagonal principal. Usar una subrutina para la lectura de la matriz, una función para calcular la traza y una subrutina para las dos sumas pedidas.

7) Programa que desplace los valores de las componentes del vector A(5,10,15,20,25,30,35) una posición hacia la derecha de modo que el valor de la última componente pase a la primera, es decir, después del desplazamiento, el vector resultante es A(35,5,10,15,20,25,30). Usar una subrutina para realizar el desplazamiento a la derecha.

8) Lo mismo que en el ejercicio anterior pero desplazando los valores de las componentes del vector A(5,10,15,20,25,30,35) una posición hacia la izquierda de modo que el valor de la primera componente pase a la última, es decir, después del desplazamiento, el vector resultante es A(10,15,20,25,30,35,5). Usar una subrutina para realizar el desplazamiento a la izquierda.

9) Programa que calcule el producto de 2 matrices de 3X2 y 2X3, respectivamente. Usar una subrutina para la lectura de las dos matrices a multiplicar y otra subrutina para calcular la matriz producto. (Guíate por el programa CAP4_14).

10) Programa que pida al usuario por teclado el número de filas y columnas de dos Matrices A y B (iguales para ambas) y sus componentes. Dimensiona dinámicamente las matrices. A continuación, el programa presentará estas opciones:

– 1. Mostrar por monitor la Matriz A.

– 2. Mostrar por monitor la Matriz B.

– 3. Mostrar por monitor la traspuesta de la Matriz A.

– 4. Mostrar por monitor la traspuesta de la Matriz B.

– 5. Mostrar por monitor Matriz A + Matriz B.

– 6. Mostrar por monitor Matriz A - Matriz B.

– 7. Salir.

– Usa el mismo subprograma para responder a las opciones 1 y 2 del menú anterior, otro subprograma para responder a las opciones 3 y 4 y otro para responder a las opciones 5 y 6.

– Antes de acabar el programa libera el espacio reservado en memoria previamente para las matrices A y B.

11) Repite cualquier ejercicio resuelto de este capítulo que use arrays pero dimensionándolos dinámicamente, en tiempo de ejecución.

6 CARACTERES Y CADENAS

6.1 Caracteres y cadenas

- En el capítulo 1 se indicó cómo tratar con variables y constantes carácter en Fortran 90/95. Recordar que la declaración de este tipo de variables, como ocurre con cualquier otro, puede incluir una inicialización de las mismas. Por otro lado, las constantes carácter pueden tener nombres simbólicos si se añade el atributo PARAMETER en su sentencia de declaración y deben encerrarse entre comillas dobles o simples.
- Posteriormente, en el capítulo 4 se estudió que la forma de declarar un array es idéntica para cualquier tipo de datos.
- Ejemplo. Escribir sentencias de declaración de diferentes datos carácter.

CHARACTER (len=15):: apellido ! declara variable carácter

CHARACTER:: seguir='S' ! declara e inicializa var carácter

CHARACTER (len=10), PARAMETER:: archivo='entrada' ! declara nombre simbólico para constante carácter

CHARACTER (len=25), DIMENSION(50)::alumnos ! declara array de 50 ! elementos cada uno de los cuales puede ser un conjunto de 25 !caracteres como máximo

- Una cadena de caracteres o simplemente *cadena* es una sucesión explícita de caracteres.
- Una subcadena de caracteres o simplemente *subcadena* es una porción contigua de caracteres de una cadena. Para referenciar una subcadena de una cadena, la sintaxis general es.

nombre ([pos_inicial]: [pos_final])

- *nombre* es el nombre de una variable o elemento de array carácter.
- *pos_inicial* es una expresión numérica que especifica la posición inicial (más a la izquierda) del primer carácter de la subcadena. Si no se especifica, se toma como valor por defecto la posición del primer carácter.
- *pos_final* es una expresión numérica que especifica la posición final (más a la derecha) del último carácter de la subcadena. Si no se especifica, se toma como valor por defecto la longitud de *nombre*.
- Los valores de *pos_inicial* y *pos_final* deben cumplir la condición:

1 ≤ *pos_inicial* ≤ *pos_final* ≤ longitud_nombre

- Ejemplo. Sea la declaración:

CHARACTER (len=10):: nombre='Susana'

nombre(2:4)   hace referencia a ‘usa’

nombre(:)   hace referencia a ‘Susana____’[7]

6.2 Expresión carácter

- Los operadores disponibles en Fortran 90/95 para operar con cadenas son:
  - el operador de concatenación // para concatenar cadenas. La sintaxis general de una expresión carácter que emplea este operador para concatenar variables carácter es:

var1_caracter // var2_caracter

  - Ejemplo. Sea la declaración:

```
CHARACTER (len=10):: c1,c2
c1 = ‘aero’
c2 = ‘plano’
WRITE(*,*) c1//c2 !se escribe por pantalla aeroplano_
```

  - los operadores relacionales (ver Tabla 2.1) para comparar cadenas. Sin embargo, su uso está desaconsejado pues el resultado de la comparación puede variar de computador a computador. En su lugar, se aconseja comparar cadenas utilizando las funciones intrínsecas léxicas cuyos resultados son independientes del procesador. Estas funciones se estudian en la sección 6.4.
  - Ejemplo. Sean las declaraciones:

```
CHARACTER (len=15):: apellido1,apellido2
apellido1 < apellido2 !Expresión carácter que compara las 2 variables
```

6.3 Asignación carácter

- Una sentencia de asignación carácter asigna el valor de una expresión carácter a una variable o elemento de array del mismo tipo. La sintaxis general es:

variable_carácter = expresión_carácter

- El funcionamiento es:
  - Se evalúa la expresión carácter.

---

[7] Cada guión bajo representa un blanco.

- Se asigna el valor obtenido a la variable carácter.
- Si la longitud de la variable es mayor que la de la expresión, el valor de la expresión se ajusta a la izquierda de la variable y se añaden blancos hasta completar la longitud total de la variable.
- Si la longitud de la variable es menor que la de la expresión, el valor de la expresión es truncado.

- Ejemplo. Sea la declaración:

CHARACTER (len=10):: c1,c2,c3

c1 = 'aero'

c2 = 'plano'

c3=c1//c2

WRITE(*,*) c3 !se escribe por pantalla *aeroplano_*

c3='demasiado largo'

WRITE(*,*) c3 !se escribe por pantalla *demasiado_*

6.4 Funciones intrínsecas carácter

- A continuación, se definen algunas funciones intrínsecas útiles para manipular caracteres: IACHAR, ACHAR, LEN, LEN_TRIM TRIM, INDEX.
    - IACHAR(caracter) convierte el carácter de entrada en un número que corresponde a su posición en el código ASCII.
    - ACHAR(numero) es la función inversa de IACHAR, pues convierte el número de entrada en un carácter según su posición en el código ASCII.
    - LEN(cadena) devuelve la longitud declarada para la variable carácter.
    - LEN_TRIM(cadena) devuelve un número entero que corresponde a la longitud de la cadena eliminando los blancos.
    - TRIM(cadena) devuelve la cadena eliminado los blancos.
    - INDEX(cadena1,subcadena2[,.TRUE.]) devuelve la primera coincidencia del patrón subcadena2 en la cadena1. Si el tercer argumento no está, la búsqueda se realiza de izquierda a derecha. Si el tercer argumento está presente, la búsqueda se realiza de derecha a izquierda. Si no se encuentra coincidencia devuelve un 0.
- Ejemplo.

CHARACTER (len=10)::pal1='raton'

WRITE(*,*) IACHAR('A'),IACHAR('Z'),IACHAR('a'),IACHAR('z')

!se escribe por pantalla 65,90,97,122

```
WRITE(*,*) ACHAR(65),ACHAR(90),ACHAR(97),ACHAR(122)
!se escribe por pantalla A,Z,a,z
WRITE(*,*) LEN(pal1),LEN_TRIM(pal1)
!se escribe por pantalla 10   5
WRITE(*,*) pal1,TRIM(pal1)
!se escribe por pantalla raton_____raton
WRITE(*,*) INDEX(pal1,'a')
!se escribe por pantalla 2
```

- Las funciones intrínsecas *léxicas* permiten comparar cadenas y son las siguientes: LLT (Lexically Less Than), LLE (Lexically Less or Equal than), LGT (Lexically Great Than) y LGE (Lexically Great or Equal than). Estas funciones son equivalentes a los operadores relaciones <, <=, > y >=, respectivamente. Ahora bien, mientras las funciones léxicas utilizan siempre el código ASCII como base para realizar las comparaciones, los operadores relacionales pueden utilizar este código o cualquier otro, según el computador.
- La comparación entre cadenas se realiza de la siguiente manera:
  - Se comparan las dos cadenas carácter a carácter, comenzando por el primer carácter (el que se encuentra más a la izquierda) y continuando hasta que se encuentra un carácter distinto o hasta que finaliza la cadena.
  - Si se encuentran caracteres distintos, el operando que contiene el carácter menor[8], será considerado el operando menor. Por tanto, el orden de los operandos viene marcado por el primer carácter que difiere entre ambos operandos.
  - Si se alcanza el final de uno de los operandos y no hay caracteres distintos, la ordenación de las cadenas se hará en función de sus longitudes. Así, si ambas cadenas tienen la misma longitud, las cadenas son iguales, mientras que si las cadenas tienen longitudes diferentes, la comparación continúa como si la cadena más corta estuviera rellena de blancos hasta la longitud de la cadena más larga.
- Ejemplo. Sean las declaraciones:

```
CHARACTER (len=15):: pal1,pal2
```

[8] Un carácter es menor que otro si la posición que ocupa el primero en el código ASCII es menor que la que ocupa el segundo.

```
LOGICAL:: result1,result2
pal1=’Begoña’
pal2=’Paula’
result1=pal1<pal2
result2=LLT(pal1,pal2)
```

El valor de result1 puede variar de procesador a procesador, pero el valor de result2 es siempre .TRUE. en cualquier procesador.

EJERCICIOS RESUELTOS

Objetivos:

Aprender a usar variables y arrays carácter en Fortran y a transferirlos a procedimientos externos. Manejar las funciones intrínsecas más importantes relacionadas con este tipo de variables.

1. Pedir el nombre y apellido de una persona. Hallar la longitud de la cadena nombre. Guardar el nombre completo en una única variable y decir cuantas ‘aes’ tiene.

```
PROGRAM cap6_1
IMPLICIT NONE
CHARACTER (LEN=10) :: nom, apel
CHARACTER (LEN=20) :: nomc
INTEGER :: long1,long2=0,i,conta=0

WRITE(*,*) 'DAME TU NOMBRE'
READ(*,*) nom

long1=len_trim(nom)
WRITE(*,*) 'nom,' tiene ',long1,’caracteres’
!otro modo de calcular la longitud de una cadena
DO i=1,LEN(nom)
  IF (nom(i:i) /= ' ') THEN
    long2=long2+1
  END IF
END DO
WRITE(*,*)'nom,' tiene ',long2,’caracteres’
!..........................................
WRITE(*,*) 'DAME TU APELLIDO'
READ(*,*) apel

nomc=TRIM(nom)//' '//apel
WRITE(*,*) 'TU NOMBRE COMPLETO ES ',nomc

DO i=1,LEN_TRIM(nomc)
  IF (nomc(i:i)=='A'.OR. nomc(i:i) == 'a') THEN
    conta=conta+1
  END IF
END DO
WRITE(*,*) 'LA CANTIDAD DE A EN ',nomc,' ES',conta

END PROGRAM cap6_1
```

– El operador de concatenación // permite concatenar el nombre y apellido en una expresión carácter y asignar el resultado a la variable nomc.

– Repasar en la sección 6.1 la forma de referenciar una subcadena de una cadena. En este ejercicio, para extraer de una en una las letras de la variable nomc, se escribe nomc(i:i), con la posición inicial igual a la posición final, dentro de un bucle, en el que el índice i toma los valores desde 1 hasta la longitud de la cadena nombre.

2. Escribir por pantalla la tabla de caracteres ASCII usando la función intrínseca IACHAR.

```
PROGRAM cap6_2
IMPLICIT NONE
CHARACTER (LEN=27) :: abc='ABCDEFGHIJKLMNÑOPQRSTUVWXYZ',&
abcm='abcdefghijklmnñopqrstuvwxyz'
INTEGER:: i
WRITE(*,*) 'N ASCII    LETRA   N ASCII    LETRA'
DO i=1,27
  WRITE(*,100) IACHAR(abc(i:i)),abc(i:i),IACHAR(abcm(i:i)),abcm(i:i)
PAUSE
END DO
100 FORMAT(2X,I4,8X,A2,8X,I4,6X,A2)
END PROGRAM cap6_2
```

– La sentencia PAUSE suspende temporalmente la ejecución del programa.

– La sentencia FORMAT permite mostrar con formatos específicos la lista de variables dada de forma que éstas quedan en columnas bien alineadas. Esta sentencia se explica en detalle en el capítulo 7.

– ¿Qué números corresponden a las letras ñ y Ñ?

– ¿Qué relación existe entre el número asociado a una letra minúscula y su correspondiente mayúscula?

3. Pasar a minúsculas un nombre que se lee por teclado usando las funciones intrínsecas IACHAR y ACHAR. Suponer que el nombre leído puede tener mezcladas letras minúsculas y mayúsculas.

```
PROGRAM cap6_3
IMPLICIT NONE
```

```
CHARACTER (LEN=20) :: nom=' ',nom_mayus
INTEGER :: num,num_mayus,i
WRITE(*,*) 'DAME UN NOMBRE EN MAYUSCULAS'
READ(*,*) nom_mayus
WRITE(*,*) 'EL NOMBRE TECLEADO ES ',nom_mayus
DO i=1,LEN_TRIM(nom_mayus)
   num_mayus=IACHAR(nom_mayus(i:i))
   IF (num_mayus >= 65.AND.num_mayus <= 90) THEN
     num=num_mayus+32
     nom(i:i)=ACHAR(num)
   ELSE
     nom(i:i)=ACHAR(num_mayus)
   END IF
END DO
WRITE(*,*) 'EL NOMBRE EN MINUSCULAS ES ',nom
END PROGRAM cap6_3
```

– Para saber si cada letra del nombre es mayúscula, se obtiene su número asociado y se testea si pertenece al intervalo [65-90] (se prescinde de la ñ). En caso afirmativo, se suma 32 al número, se reconvierte a letra y se coloca en la posición adecuada de la variable declarada para almacenar el nombre en minúsculas.

4. Pasar a mayúsculas un nombre que se lee por teclado usando las funciones intrínsecas IACHAR y ACHAR. Suponer que el nombre leído puede tener mezcladas letras minúsculas y mayúsculas.

```
PROGRAM cap6_4
IMPLICIT NONE
CHARACTER (LEN=20) :: nom,nom_mayus=' '
INTEGER :: num,num_mayus,i
WRITE(*,*) 'DAME UN NOMBRE EN MINUSCULAS'
READ(*,*) nom
WRITE(*,*) 'EL NOMBRE TECLEADO ES ',nom
DO i=1,LEN_TRIM(nom)
  num=IACHAR(nom(i:i))
  IF (num >= 97 .AND. num <= 122) THEN
     num_mayus=num-32
     nom_mayus(i:i)=ACHAR(num_mayus)
  ELSE
```

```
    nom_mayus(i:i)=ACHAR(num)
  END IF
END DO
WRITE(*,*) 'EL NOMBRE EN MAYUSCULAS ES ',nom_mayus
END PROGRAM cap6_4
```

– Se usa el mismo procedimiento que en el ejercicio anterior. Para pasar a mayúsculas una letra minúscula basta restar 32 al número correspondiente según la tabla ASCII.

– ¿Qué ocurre si restamos un valor constante distinto de 32 a cada número? ¿puede servir este método para encriptar mensajes?

5. Invertir una palabra usando una subrutina. La palabra invertida debe almacenarse en la misma variable usada para la palabra original.

```
PROGRAM cap6_5
IMPLICIT NONE
CHARACTER (LEN=50) :: nombre
INTEGER :: long,i=0
WRITE(*,*) 'DAME UN NOMBRE'
READ(*,*) nombre

DO
  i=i+1
  IF (nombre(i:i) == ' ') EXIT
END DO
WRITE(*,*) 'LA PALABRA TIENE',i-1,' CARACTERES'
long=i-1
CALL invertir(nombre,long)
WRITE(*,*) 'LA PALABRA INVERTIDA ES ',nombre
END PROGRAM cap6_5

SUBROUTINE invertir(nombre,long)
IMPLICIT NONE
INTEGER, INTENT(IN) :: long
CHARACTER (LEN=long), INTENT(IN OUT) :: nombre
CHARACTER (LEN=1) :: aux
INTEGER :: cen,i,j
```

```
j=long
cen=long/2
DO i=1,cen
  aux=nombre(i:i)
  nombre(i:i)=nombre(j:j)
  nombre(j:j)=aux
  j=j-1
  WRITE(*,*) 'NOMBRE ',nombre
END DO
END SUBROUTINE invertir
```

– El algoritmo usado para la inversión consiste en localizar la posición central de la palabra e intercambiar las posiciones de las letras última y primera, penúltima y segunda y así sucesivamente hasta llegar a la posición central de la palabra.
– El bucle DO del programa principal permite calcular la longitud de la palabra a invertir. La función intrínseca LEN_TRIM (cadena) realiza la misma tarea.
– Al llamar a la subrutina, se transfiere la dirección de memoria del primer carácter de la variable nombre.

6. Leer una sílaba y una palabra y escribir en qué posición de la palabra está la sílaba si es que está, empezando por la izquierda.

```
PROGRAM cap6_6
IMPLICIT NONE
CHARACTER (LEN=6) :: sil=' '
CHARACTER (LEN=30) :: pal=' '
WRITE(*,*) 'PALABRA'
READ(*,*) pal
WRITE(*,*) 'SILABA'
READ(*,*) sil
WRITE(*,*) INDEX(pal,TRIM(sil))
END PROGRAM cap6_6
```

7. Buscar un nombre en un array de cuatro nombres.

```
PROGRAM cap6_7
IMPLICIT NONE
```

```
CHARACTER (LEN=15), DIMENSION(4) :: &
nom=(/'PEPE GOTERA','ROMPETECHOS','MORTADELO  ','FILEMON    '/)
CHARACTER (LEN=15):: busca
INTEGER :: switch,i=0,ipos

WRITE(*,*) 'DAME UN NOMBRE'
READ(*,*) busca
switch=0
DO
  i=i+1
  IF (nom(i) == busca) THEN
    switch=1
    ipos=i
    WRITE(*,*) 'EL NOMBRE SE ENCUENTRA EN LA POSICION',ipos
  END IF
  IF (i == 4 .OR. switch == 1) EXIT
END DO
IF (switch == 0) THEN
  WRITE(*,*) 'EL NOMBRE NO ESTA EN LA LISTA'
END IF
END PROGRAM cap6_7
```

- En este programa nom es el identificador de un array de 4 componentes, cada una de las cuales es un nombre de 15 caracteres como máximo. Todos los elementos deben tener la misma longitud.
- El algoritmo usado en este ejercicio es el mismo que el del programa cap4_3 para buscar un número en un vector de números. Al igual que allí, la variable switch funciona de interruptor, de modo que el programa actúa según su contenido.

8. Inicializar los códigos y los nombres de diez provincias en dos vectores carácter. A continuación, se pide al usuario un código por teclado y el programa debe mostrar el nombre de la provincia correspondiente. Si no existe el código leído, mostrar un mensaje que avise de ello. El programa se ejecuta mientras el usuario lo desee.

```
PROGRAM cap6_8
IMPLICIT NONE
INTEGER:: i
CHARACTER (LEN=1) :: resp
```

```
CHARACTER(LEN=2):: cod
CHARACTER (LEN=2), DIMENSION(10) :: &
tcod=(/'A ','AL','AV','B ','BA','C ','CA','CC','CO','CS'/)
CHARACTER (LEN=9),DIMENSION(10) :: &
tnom=(/'ALICANTE ','ALMERIA   ','AVILA     ','BARCELONA', 'BADAJOZ '&
,'CORUÑA   ','CADIZ    ','CACERES  ','CORDOBA  ', 'CASTELLON'/)

DO
  WRITE(*,*) 'DAME UN CODIGO DE PROVINCIA'
  READ(*,*)cod
! ***BUSQUEDA EN EL ARRAY
  i=0
  DO
    i=i+1
         IF (cod == tcod(i) .OR. i == 10) EXIT
  END DO
  IF (cod /= tcod(i)) THEN
    WRITE(*,*) 'ERROR. NO EXISTE ESE CODIGO'
  ELSE
    WRITE(*,*) 'LA PROVINCIA ES ',tnom(i)
  END IF
  WRITE(*,*) 'CONTINUAR(S/N)?'
  READ(*,*) resp
  IF (resp /= 'S' .AND. resp /= 's') EXIT
END DO
END PROGRAM cap6_8
```

9. Ordenar ascendentemente los nombres de tres personas que se introducen por teclado. Utilizar el método de la burbuja.

```
PROGRAM cap6_9
IMPLICIT NONE
INTEGER, PARAMETER::N=3
CHARACTER (LEN=20), DIMENSION(N) :: nom
INTEGER:: i
CALL leer(nom,N)
CALL ordenar(nom,N)
WRITE(*,*) 'LA LISTA ORDENADA ASCENDENTEMENTE ES'
WRITE(*,*) (nom(i),i=1,N)
```

```
END PROGRAM cap6_9

SUBROUTINE leer(x,tam)
IMPLICIT NONE
INTEGER, INTENT(IN):: tam
CHARACTER (LEN=20), DIMENSION(tam), INTENT(OUT) :: x
INTEGER:: i
DO i=1,tam
  WRITE(*,*) 'DAME NOMBRE ENTRE APOSTROFES',i
  READ(*,*) x(i)
END DO
END SUBROUTINE leer

SUBROUTINE ordenar(x,n)
IMPLICIT NONE
INTEGER, INTENT(IN):: n
CHARACTER (LEN=*), DIMENSION(n), INTENT(IN OUT) :: x
CHARACTER (LEN=20)::aux
INTEGER:: i,j
  DO  i=n-1,1,-1
  DO  j=1,i
   IF (LGT(x(j),x(j+1))) THEN
     aux=x(j)
     x(j)=x(j+1)
     x(j+1)=aux
   END IF
  END DO
END DO
END SUBROUTINE ordenar
```

- El método de la burbuja ya se implementó para ordenar números en el capítulo 4 (ver cap4_7).
- ¿Cómo cambia el programa si se quiere realizar un ordenamiento descendente y el número de personas es siete?

EJERCICIOS PROPUESTOS

1) Programa que codifica la frase 'EXAMEN DE INFORMATICA'.

2) Programa que descodifica la frase codificada en el ejercicio anterior. Utiliza una subrutina para codificar y descodificar la frase.

3) Programa que lea dos palabras y las muestre ordenadas alfabéticamente. (No usar ningún método de ordenamiento).

4) Programa que pida una frase y cuente el número de palabras.

5) Programa que lea una palabra y verifique si es un palíndromo o no. Utilizar el programa cap6_5 para invertir la palabra. Usar una función lógica para determinar si la palabra dada es un palíndromo o no.

6) Programa que lea una frase y cuente el número de vocales de cada palabra mostrando la de mayor número por pantalla. Usa una subrutina para leer la frase y otra para determinar la palabra que contiene más vocales.

7) Programa que lea una palabra aguda y diga si requiere tilde o no según las reglas de acentuación. El programa se ejecuta hasta que el usuario introduzca la palabra FIN. Usa programación modular para la construcción del programa.

8) Programa que lea una palabra y el idioma en que está escrita (inglés/castellano) y muestre su traducción (castellano/inglés). Suponer que el diccionario está formado por las palabras siguientes: Computador, raton, pantalla, teclado, programa, ejecutar. Computer, mouse, screen, keyboard, program, execute. Usa programación modular para la construcción del programa.

7 FORMATOS Y ARCHIVOS

7.1 Entrada/salida en Fortran

- Al igual que en el resto de los lenguajes, se llama **entrada o lectura** de datos al proceso de pasar esos datos desde un dispositivo de entrada (teclado, archivo, etc) a un computador.
- El proceso simétrico se llama **salida o escritura** de datos.
- En Fortran, hay dos maneras de realizar entradas/salidas:
    - entrada/salida *dirigida por lista*. En este caso, el formato de los datos depende de sus tipos (enteros, reales, etc.) y del computador. Se dice que es una entrada/salida con *formato libre*.
    - entrada/salida *con formatos*. El programador define la manera exacta en que quiere leer/escribir los datos.
- Hay una sentencia de lectura: READ y dos de escritura: WRITE y PRINT. Se comienza estudiando las sentencias de escritura por ser más útiles que la de lectura.

7.2 Salida por pantalla

- Hasta ahora, la salida de datos se ha realizado siempre por pantalla con formato libre. Para ello, se ha empleado la sentencia:

WRITE (*,*) lista de variables

- en la cual el primer asterisco se refiere al dispositivo de salida estándar (generalmente, la pantalla) y el segundo asterisco se refiere al formato libre con que se mostrarán las variables de la lista,

- o la sentencia:

PRINT *, lista de variables

- en la cual PRINT significa escribir en el dispositivo de salida estándar y el asterisco indica con formato libre.

- Sin embargo, estas salidas no tienen generalmente el aspecto deseado, ya que aparecen demasiados blancos extra. En realidad, constituyen un caso particular de unas sentencias más generales en las que se puede indicar con exactitud cómo se quieren escribir los datos. La sintaxis general de una salida por pantalla con formatos es:

WRITE (*, formato) lista de variables ó

PRINT formato, lista de variables

- donde *formato* puede ser un * (salida dirigida por lista), pero también puede ser una expresión carácter, variable o constante, que contiene los *descriptores de formato* de la lista, o la etiqueta de una

sentencia FORMAT, es decir, un entero entre 1 y 99999. En este último caso, debe existir además una sentencia de la forma:

etiqueta FORMAT (lista de descriptores de formato)

- En las secciones siguientes, se estudian en detalle los descriptores de formato disponibles en Fortran 90/95.
- La sentencia PRINT sólo funciona con el dispositivo de salida estándar y, por lo tanto, es mucho menos flexible que la sentencia WRITE estándar, como se verá en una sección siguiente.
- PRINT sobrevive en Fortran 90/95 por su extraordinario uso en versiones anteriores. Es útil reconocer esta sentencia, sin embargo, no debería usarse en los programas creados por el usuario.
- Ejemplos de salidas por pantalla.

WRITE (*, *) N,M ! formato libre

WRITE (*,'2I3') N,M !constante carácter especifica formatos de N y M

string='2I3'

WRITE (*,string) N,M !variable carácter especifica formatos de N y M

WRITE (*, 200) N,M !etiqueta de sentencia FORMAT

200 FORMAT (2I3) !aqui se especifican formatos de N y M

7.3 Entrada por teclado

- Hasta ahora, la entrada de datos se ha realizado siempre por teclado con formato libre. Para ello, se ha empleado la sentencia:

READ (*,*) lista de variables

  - en la cual el primer asterisco se refiere al dispositivo de entrada estándar (generalmente, el teclado) y el segundo asterisco se refiere al formato libre con que se leerán las variables de la lista,

- o la sentencia:

READ *, lista de variables

  - en la cual READ significa leer del dispositivo de entrada estándar y el asterisco indica con formato libre.

- Sin embargo, estas entradas pueden no tener el resultado deseado y constituyen un caso particular de unas sentencias más generales en las que se puede indicar con exactitud cómo se quieren leer los datos de teclado. La sintaxis general de una entrada por teclado con formatos es:

READ (*, formato) lista de variables ó

READ formato, lista de variables

- donde *formato* puede ser un asterisco *, entrada dirigida por lista, pero también una expresión carácter, variable o constante, que contiene los descriptores de formato de la lista, o la etiqueta de una sentencia FORMAT, es decir, un entero entre 1 y 99999. En este último caso, debe existir además una sentencia de la forma:

etiqueta FORMAT (lista de descriptores de formato).

- La segunda forma de READ sólo funciona con el dispositivo de entrada estándar y, por lo tanto, es mucho menos flexible que la primera, cómo se verá en una sección siguiente. Sobrevive en Fortran 90/95 por su extraordinario uso en versiones anteriores. Es útil reconocer esta sentencia, sin embargo, no debería usarse en los programas creados por el usuario.

7.4 Descriptores de formato

- Hay 4 categorías básicas de descriptores de formato:
    - Los que describen la posición vertical de la línea de texto.
    - Los que describen la posición horizontal de los datos en una línea.
    - Los que describen el formato de entrada/salida de un valor particular.
    - Los que controlan la repetición de descriptores o grupos de descriptores de formato.
- La siguiente tabla contiene una lista de símbolos usados con los descriptores de formatos más comunes:

SÍMBOLO	SIGNIFICADO
c	número de columna
d	nº de dígitos a la derecha del punto decimal para entrada/salida de datos reales
m	mínimo nº de dígitos
n	nº de espacios saltados
r	factor de repetición: nº de veces que se usa un descriptor o grupo de descriptores
w	anchura del campo: nº de caracteres de entrada/salida

Tabla 7.1: Símbolos usados en los descriptores de formatos

7.4.1 Descriptor I de formato entero

- Sintaxis general para salida de datos enteros:

[r]Iw[.m]

- Los símbolos usados tienen el significado que se muestra en la Tabla 7.1. El valor se ajusta a la derecha del campo. Si el valor es demasiado grande para mostrarse con w caracteres, se muestran w asteriscos.
- Sintaxis general para entrada de datos enteros:

[r]Iw

- El valor puede estar en cualquier posición dentro del campo especificado.
- Ejemplos de salida:

DESCRIPTOR	VALOR INTERNO	SALIDA
I4	452	•452
I2	6234	**
I5	-52	••-52
I4.3	3	•003
I2.0	0	••

Tabla 7.2: Formatos de escritura de enteros

- Ejemplos de entrada:

DESCRIPTOR	CAMPO ENTRADA	VALOR LEÍDO
I4	•1••	1
I2	-1	-1
I4	-123	-123
I3	•12	12
I2	123	12

Tabla 7.3: Formatos de lectura de enteros

7.4.2 Descriptor F de formato real

- Sintaxis general para entrada/salida de datos reales:

[r]Fw.d

- Los símbolos usados tienen el significado que se muestra en la Tabla 7.1.
- Para salida, el valor se ajusta a la derecha del campo.
- Si d es menor que el número de dígitos decimales del número, el valor se redondea.
- Si d es mayor que el número de dígitos decimales del número, se añaden ceros hasta completarlo.
- Si el valor es demasiado grande para leerse/escribirse con w caracteres, el campo w se llena de asteriscos.
- Para evitar mal interpretaciones, es conveniente incluir siempre un punto decimal en cualquier valor real usado en una sentencia de lectura con formato.
- Ejemplos de salida:

DESCRIPTOR	VALOR INTERNO	SALIDA
F9.3	25.338	•••25.338
F5.1	0.35247	••0.4
F6.2	0.089235	••0.09
F8.3	732.56	•732.560
F4.3	-12.345	****

Tabla 7.4: Formatos de escritura de reales

- Ejemplos de entrada:

DESCRIPTOR	CAMPO ENTRADA	VALOR LEÍDO
F6.3	49.225	49.225
F6.2	49.225	49.23
F7.1	-1525.1	-1525.1

Tabla 7.5: Formatos de lectura de reales

7.4.3 Descriptor E de formato exponencial

- La notación científica es muy usada por científicos e ingenieros para representar números muy grandes o muy pequeños. En esta notación, los números se normalizan al rango de valores entre 1.0 y 10.0 y se representan, por un número entre 1.0 y 10.0 multiplicado por una potencia de diez. Por ejemplo: el número de Avogadro en notación científica es $6.023x10^{23}$.

- El formato exponencial no se corresponde exactamente con la notación científica, pues los números se normalizan al rango de valores entre 0.1 y 1.0. En el ejemplo anterior, el número de Avogadro se representa en formato exponencial por 0.6023E+24.
- La sintaxis general para entrada/salida de datos reales con formato exponencial es:

[r]Ew.d

- Los símbolos usados tienen el significado que se muestra en la Tabla 7.1.
- Los mismos criterios que operan sobre un dato real con formato F se aplican a ese dato real con formato exponencial.
- Si un número se quiere leer/escribir en formato exponencial con d cifras decimales, la anchura de campo mínima w ≥ d+7, pues tal número se representa ±0.ddddE±ee y requiere como mínimo un carácter para representar el signo (sólo si es negativo), otro para la parte entera del número, el punto decimal, el carácter E, el signo del exponente y los dos dígitos del mismo.
- Ejemplos de salida:

DESCRIPTOR	VALOR INTERNO	SALIDA
E8.2	83.456	0.83E+02
E7.2	83.456	*******
E10.3	8.3974	•0.840E+01
E10.4	0.83E2	0.8300E+02

Tabla 7.6: Formatos de escritura de reales en formato exponencial

- Ejemplos de entrada:

DESCRIPTOR	CAMPO ENTRADA	VALOR LEÍDO
E11.2	•43.258E+03	0.43E+05
E11.5	•43.258E+03	0.43258E+05
E11.3	•••0.43E-02	0.430E-02

Tabla 7.7: Formatos de lectura de reales en formato exponencial

7.4.4 Descriptor ES de formato científico

- El formato científico coincide exactamente con la definición de notación científica dada más arriba.
- La sintaxis general para entrada/salida de datos reales con formato científico es:

[r]ESw.d

- Los símbolos usados tienen el significado que se muestra en la Tabla 7.1.
- Los mismos criterios que operan sobre un dato real con formato F se aplican a ese dato real con formato científico.
- Si un número se quiere leer/escribir en formato científico con d cifras decimales, la anchura de campo mínima w ≥ d+8, pues tal número se representa ±1.ddddES±ee y requiere como mínimo un carácter para representar el signo (sólo si es negativo), otro para la parte entera del número, el punto decimal, los dos caracteres ES, el signo del exponente y los dos dígitos del mismo.
- La diferencia entre el formato exponencial y el científico para un valor real dado con una anchura de campo dada es que en el formato científico se representa una cifra significativa más que en el formato exponencial.
- Ejemplos de salida:

DESCRIPTOR	VALOR INTERNO	SALIDA
ES8.2	83.456	8.35E+01
ES7.2	83.456	*******
ES10.3	8.3974	•8.397E+00
E11.4	0.83ES2	8.3000ES+01

Tabla 7.8: Formatos de escritura de reales en formato científico

- Ejemplos de entrada:

DESCRIPTOR	CAMPO ENTRADA	VALOR LEÍDO
ES11.2	•43.258E+03	4.33ES+04
ES11.5	•43.258E+03	4.32580ES+04
ES11.3	•••0.43E-02	4.300ES-01

Tabla 7.9: Formatos de lectura de reales en formato científico

7.4.5 Descriptor L de formato lógico

- Sintaxis general para entrada/salida de datos lógicos:

[r]Lw

- La salida de un dato lógico es T o F y su valor se ajusta a la derecha del campo.

- La entrada de un dato lógico se usa muy raramente pero puede ser T o F como primer carácter no blanco situado en cualquier posición dentro de la anchura de campo dada.
- Ejemplos de salida:

DESCRIPTOR	VALOR INTERNO	SALIDA
L5	.FALSE.	••••F
L4	.TRUE.	•••T
L1	.TRUE.	T
L2	.FALSE.	•F

Tabla 7.10: Formatos de escritura de datos lógicos

- Ejemplos de entrada:

DESCRIPTOR	CAMPO ENTRADA	VALOR LEÍDO
L5	•••T•	.TRUE.
L2	F1	.FALSE.
L4	•X•T	ERROR

Tabla 7.11: Formatos de lectura de datos lógicos

7.4.6 Descriptor A de formato carácter

- Sintaxis general para entrada/salida de datos carácter:

[r]A[w]

- Si w no aparece, el descriptor A lee/escribe el dato carácter en una anchura igual a la longitud de la variable carácter.
- Si w aparece, el descriptor A lee/escribe el dato carácter en una anchura fija a w.
    - Si w>longitud de la variable carácter:
        - para salida, la cadena se ajusta a la derecha del campo y,
        - para entrada, el dato del fragmento derecho del campo se lee en la variable carácter.
    - Si w<longitud de la variable carácter:
        - para salida, sólo se escriben los primeros w caracteres de la cadena y,

- para entrada, sólo los primeros w caracteres de la cadena se ajustan a la izquierda de la variable carácter y el resto se llena con blancos.

- Ejemplos de salida:

DESCRIPTOR	VALOR INTERNO	LONGITUD DE LA VARIABLE CARACTER	SALIDA
A	ABCDEF	6	ABCDEF
A8	ABCDEF	6	••ABCDEF
A4	ABCDEF	6	ABCD

Tabla 7.12: Formatos de escritura de caracteres

- Ejemplos de entrada:

DESCRIPTOR	CAMPO ENTRADA	LONGITUD DE LA VARIABL CARACTER	VALOR LEÍDO
A	ABCDEFGH	6	ABCDEF
A	ABCDEFGH	8	ABCDEFGH
A8	ABCDEFGH	4	EFGH
A4	ABCDEFGH	6	ABCD••

Tabla 7.13: Formatos de lectura de caracteres

7.4.7 Descriptores X, T de posición horizontal y / de posición vertical

- Los descriptores X y T se usan para controlar el espacio horizontal y el descriptor slash / para controlar el espacio vertical. La sintaxis general de cada uno de ellos es:

nX

- Para salida: suele emplearse para espaciar los datos. El descriptor nX salta n espacios en la línea actual.
- Para entrada: puede emplearse para saltar por encima de campos de entrada que no se quieren leer en la línea actual.

Tc

- Salta directamente a la columna número c de la línea actual. Funciona como un tabulador más general, pues puede saltar hacia derecha o izquierda.
    - Para salida: suele emplearse para espaciar los datos.

- Para entrada: puede emplearse para saltar por encima de campos de entrada que no se quieren leer o para leer varias veces unos datos.

/[/][…]

- Este es un descriptor especial que no es necesario separarlo de los demás descriptores por comas, si bien pueden usarse.
    - Para salida: un slash envía la línea actual a salida y empieza una nueva. Así, una sentencia WRITE puede escribir los valores de salida separados en dos líneas. Si aparecen varios slashes juntos, se saltarán varias líneas.
    - Para entrada, un slash ignora la línea actual y comienza a procesar la siguiente línea.
- Ejemplo. Sean las declaraciones:

INTEGER:: numero1=345, numero2=678

REAL:: a=7.5, b=0.182

PRINT ‘(1X,T30,A)’,’RESULTADOS’

PRINT ‘(1X,I3,2X,I3)’, numero1, numero2

PRINT ‘(1X, 2I4, F6.3/,1X,F6.3)’, numero1, numero2, a, b

Las salidas generadas son:

•••••••••••••••••••••••••••••RESULTADOS

•345••678

••345•678•7.500

••0.182

- Ejemplo. Sean las declaraciones:

INTEGER:: a,b,c,d

READ (*,30) a,b,c,d

30 FORMAT (2I2,//, 2I2)

Si los datos de entrada son:

•1•2•3

•4•5•6

•7•8•9

Se leen a, b, c, d con los valores 1, 2, 7 y 8, respectivamente.

7.4.8 Repetición de grupos de descriptores de formato

- Para repetir un grupo de descriptores de formato hay que encerrar tal grupo entre paréntesis y colocar un factor de repetición a la izquierda del mismo.
- Ejemplo:

30 FORMAT (1X, I4, F9.2, I4, F9.2, I4)

30 FORMAT (1X, I4, 2(F9.2, I4))

7.5 Procesamiento de archivos

- Las aplicaciones que manejan conjuntos de datos muy grandes, es conveniente que almacenen los datos en algún archivo en disco o algún otro dispositivo de memoria auxiliar.
- Antes de que Fortran pueda usar un archivo, debe abrirlo, asignándole una unidad. La sintaxis general para abrir un archivo es:

OPEN (lista_open)

- donde *lista_open* puede incluir varias cláusulas separadas por comas, colocadas en cualquier orden, de las cuales se estudian a continuación las más importantes:
    - [UNIT=]unidad es un número entero comprendido entre 1 y 99 que identifica al archivo.
    - FILE = archivo es una expresión carácter que indica el nombre del archivo que se quiere abrir.
    - [STATUS = estado_del_archivo] es una de las siguientes constantes carácter: 'OLD', 'NEW', 'REPLACE', 'SCRATCH' o 'UNKNOWN' (opción por defecto).
        - La opción 'SCRATCH' crea un archivo temporal que se destruye automáticamente cuando se cierra el archivo o cuando acaba la ejecución del programa. Se suele usar para guardar resultados intermedios durante la ejecución de un programa. Estos archivos no pueden tener nombre.
        - La opción 'UNKNOWN' implica, si existe el archivo antes de ejecutar el programa, que lo reemplaza, y si no existe, lo crea en tiempo de ejecución y lo abre.
    - [ACTION = accion] es una de las siguientes constantes carácter: 'READ', 'WRITE' o 'READWRITE' (opción por defecto).
    - [IOSTAT = error_de_apertura] es una variable entera que almacena el estado de la operación de apertura de archivo. Aunque es una cláusula opcional, se aconseja usarla para evitar abortar un programa cuando se produce un error de apertura. Si el valor de la variable es
        - cero, significa éxito en la apertura del archivo,
        - positivo, significa que se ha producido un error al abrir el archivo.
    - [ACCESS = acceso] es una de las siguientes constantes carácter: 'SEQUENTIAL'(opción por defecto) o 'DIRECT'.

Los archivos permiten acceso directo, es decir, saltar de una línea (también llamada registro) a cualquier otra, independientemente de su situación en el archivo. Sin embargo, por razones históricas, la técnica de acceso por defecto en Fortran es secuencial, es decir, que el acceso a los registros se realiza en orden consecutivo, desde el primer registro hasta el último.

- [POSITION = posicion] es una de las siguientes constantes carácter: 'REWIND', 'ASIS' (opción por defecto) o 'APPEND'. Si la posición es:
    - 'REWIND', el puntero de archivo se coloca en el primer registro.
    - 'ASIS', el puntero de archivo se coloca en un registro dependiente del procesador.
    - 'APPEND', el puntero de archivo se coloca después del último registro justo antes de la marca de fin de archivo.

- Ejemplo. Apertura de un archivo para lectura:

```
INTEGER:: error_de_apertura
OPEN(UNIT=12,FILE='datos',STATUS='OLD',ACTION='READ',&
IOSTAT=error_de_apertura)
```

- Ejemplo. Apertura de un archivo para escritura:

```
INTEGER:: error_de_apertura
OPEN(UNIT=9,FILE='resultado',STATUS='NEW',ACTION='WRITE
',&IOSTAT=error_de_apertura)
```

- La asociación unidad-archivo que estableció OPEN se termina con la sentencia:

CLOSE (lista_close)

- donde *lista_close* puede incluir varias cláusulas separadas por comas, de las cuales sólo el número de unidad es obligatoria:
    - [UNIT=]unidad

- Ejemplo. Cerrar archivo identificado por la unidad 12.

```
CLOSE (UNIT = 12)
```

7.6 Posición en un archivo

- Fortran proporciona dos sentencias para ayudar a moverse en un archivo secuencial. La sintaxis general es:

REWIND unidad

- para reposicionar un archivo al principio y

BACKSPACE unidad

- para reposicionar un archivo al principio de la línea o registro anterior.
- Si el archivo está en su posición inicial, las dos sentencias anteriores no tienen efecto.
- Ejemplos:

REWIND 10

BACKSPACE 30

7.7 Salida por archivo

- La sentencia WRITE permite escribir datos en cualquier dispositivo de salida, como los archivos. La sintaxis general de salida por archivo es:

WRITE ([UNIT =] unidad, formato[,IOSTAT = error_de_escritura][,...]) lista_de_variables

- Aunque esta sentencia puede incluir varias cláusulas, a continuación se estudian las tres más importantes, de las cuales sólo las dos primeras son obligatorias.
    - *unidad* es un número que identifica el dispositivo en el que se va a efectuar la salida de datos.
    - *formato* indica los formatos con que se van a escribir las variables de la lista en el dispositivo de salida. Puede ser, como ya se ha explicado en una sección anterior:
        - *, salida dirigida por lista.
        - Una expresión carácter, variable o constante, que contiene los descriptores de formatos de la lista.
        - La etiqueta de una sentencia FORMAT, es decir, un entero entre 1 y 99999. En este caso, debe existir una sentencia de especificación:

etiqueta FORMAT (lista de descriptores de formato).

- - *error_de_escritura*, indica el éxito o no de la operación de escritura. Aunque esta cláusula es opcional, se aconseja usarla para evitar abortar un programa cuando se produce un error de escritura. Si el valor de esa variable entera es:
        - cero, significa éxito en la operación de escritura,
        - positivo, significa que se ha producido un error en la escritura.
- Por lo tanto, la sentencia WRITE estándar toma los datos de la lista de variables, los convierte de acuerdo con los descriptores de formato especificados y, si no hay errores, los escribe en el archivo asociado a la unidad especificada.
- Ejemplos de escritura de un archivo:

```
INTEGER:: error_de_escritura
…
WRITE (UNIT=16,*, IOSTAT= error_de_escritura) X,Y
WRITE (20,'3F8.3') X,Y,Z
```

7.8 Entrada por archivo

- La sentencia READ estándar permite leer datos de cualquier dispositivo de entrada, como los archivos. La sintaxis general de entrada por archivo es:

READ ([UNIT =] unidad, formato [,IOSTAT = error_de_lectura][,...]) lista_de_variables

- Aunque esta sentencia puede incluir varias cláusulas, a continuación se estudian las tres más importantes, de las cuales sólo las dos primeras son obligatorias.
    - *unidad* es un número que identifica el dispositivo desde donde se va a efectuar la entrada de datos.
    - *formato* indica los formatos con que se van a leer las variables de la lista en el dispositivo de entrada. Puede ser, como ya se ha explicado en una sección anterior:
        - *, entrada dirigida por lista.
        - Una expresión carácter, variable o constante, que contiene los descriptores de formatos de la lista.
        - La etiqueta de una sentencia FORMAT, es decir, un entero entre 1 y 99999. En este caso, debe existir una sentencia de especificación:

etiqueta FORMAT (lista de descriptores de formato).

    - *error_de_lectura*, indica el éxito o no de la operación de lectura. Aunque esta cláusula es opcional, se aconseja usarla para evitar abortar un programa cuando ocurre un error de lectura o cuando se intenta leer más allá del fin de archivo. Si el valor de esa variable entera es:
        - cero, significa éxito en la operación de lectura,
        - positivo, significa que se ha producido un error en la lectura,
        - negativo, significa fin de archivo.
- Por lo tanto, la sentencia READ estándar lee los datos de un archivo asociado a una unidad, convierte sus formatos de acuerdo con los descriptores de formato y, si no hay errores, almacena esos datos fomateados en la lista de variables dada.
- Ejemplo de lectura de un archivo:

```
INTEGER:: error_de_lectura
```

```
…
READ (15, *, IOSTAT = error_de_lectura) a,b,c
```

- Ejemplo de procesamiento de archivos.

```
PROGRAM temperaturas
! Lee una serie de temperaturas de un archivo y
! calcula el valor medio
IMPLICIT NONE
CHARACTER (LEN=20):: archivo
INTEGER:: cuenta, error_de_apertura, error_de_lectura
REAL:: temperatura, suma, media
! Abrir archivo como unidad 15
WRITE( *, *) '    Dame nombre del archivo:'
READ (*, *) archivo
OPEN(15, FILE= archivo, STATUS='OLD', ACTION='READ',&
IOSTAT= error_de_apertura)
IF (error_de_apertura > 0) STOP ' error al abrir el archivo '
100 FORMAT (1X, F4.1)
cuenta=0
suma=0
DO
  READ (15, 100, IOSTAT = error_de_lectura) temperatura
  IF (error_de_lectura >0 ) THEN
    WRITE(*,*) 'error de lectura'
    EXIT
  ELSE IF (error_de_lectura <0 ) THEN
!    WRITE(*,*) 'fin del archivo'
    EXIT
  ELSE
  suma = suma + temperatura
  cuenta = cuenta + 1
  ENDIF
END DO
! Calcular temperatura media
media = suma/ REAL(cuenta)
WRITE (*, *) ' temperatura media: '
```

```
WRITE (*, *) media
CLOSE (15)
END PROGRAM temperaturas
```

EJERCICIOS RESUELTOS

Objetivos:

Aprender a leer y escribir con formatos específicos los diferentes tipos de variables Fortran estudiados.

Además, se aprende a manejar archivos para leer información de entrada requerida para la ejecución de un programa y/o escribir información de salida, generada en la ejecución del mismo.

1. Escribe una tabla de raíces cuadradas y cúbicas de todos los números naturales desde 1 hasta 100.

```
PROGRAM cap7_1
IMPLICIT NONE
INTEGER:: n
WRITE(*,8) 'NUMERO, RAIZ CUADRADA, RAIZ CUBICA'
8 FORMAT (1X, A)
WRITE (*,10) (n,SQRT(REAL(n)),REAL(n)**(1.0/3.0),n=1,100)
10  FORMAT (1X, I5, 4X, F8.3, 8X, F8.3)
END PROGRAM cap7_1
```

2. Escribir algunas variables enteras, reales y carácter con distintos formatos.

```
PROGRAM cap7_2

IMPLICIT NONE
INTEGER :: m=-123,n=9877,l=889,i=77
REAL:: r=89.126, s=14.532
CHARACTER (LEN=6):: pal='abcdef'

WRITE(*,100) 'Enteros',m,n,l,i
100  FORMAT(1X/1X,T10,A/,1X,I4,1X,I4,1X,I3,1X,I1)

WRITE(*,150) 'Reales','se redondea',r,'se rellena con ceros',r,'no cabe',r
150 FORMAT(1X/1X,T10,A/,1X,A,1X,F5.2/,1X,A,1X,F7.4/,1X,A,1X,F4.2)

WRITE(*,200) 'formato real',s,'formato exponencial',s,'formato cientifico',s
200  FORMAT(1X/1X,A,1X,F5.2/,1X,A,1X,E9.2/,1X,A,1X,ES9.2)

WRITE(*,250) 'Caracteres','w=6=LEN(pal)',pal,&
'w=8. La variable se ajusta a la derecha del campo',pal,&
'w=4. Se muestran los 4 primeros caracteres de la variable',pal
250  FORMAT(1X/,T10,A/,1X,A,1X,A/,1X,A,1X,A8/,1X,A,1X,A4)

WRITE(*,300)'repeticion de descriptores',m,n,l,i,r,s,pal
```

```
300 FORMAT(1X/,1X,A/,1X,4I5,2F6.2,A7)
END PROGRAM cap7_2
```

3. Leer de un archivo con nombre “cap7_3in.txt” los nombres y las notas de todos los alumnos de una clase. Crear en la misma carpeta un archivo con nombre “cap7_3out.txt” y escribir en él los nombres de los alumnos y su clase según sus notas, de acuerdo con la siguiente clasificación:

Clase	Nota
1	[8.5,10]
2	[5,8.5)
3	[0,5)

```
PROGRAM cap7_3
IMPLICIT NONE
CHARACTER (LEN=18) :: nombre
REAL :: nota
INTEGER                                                                 ::
clase,error_de_apertura,error_de_lectura,error_de_escritura

OPEN (60,FILE='cap7_3in.txt',STATUS='OLD',ACTION='READ',&
IOSTAT=error_de_apertura)
IF (error_de_apertura>0) STOP 'cap7_3in.txt NO ABIERTO'
OPEN (70,FILE='cap7_3out.txt',STATUS='NEW',ACTION='WRITE',&
IOSTAT=error_de_apertura)
IF (error_de_apertura>0) STOP 'cap7_3out.txt NO ABIERTO'
WRITE(70,9,IOSTAT=error_de_escritura) 'ALUMNO','CLASE'
WRITE(70,'(1X,A6,T20,A5)')'======','====='
IF (error_de_escritura>0) STOP 'error de escritura'
9 FORMAT (1X,A6,T20,A5)
DO
  READ(60,*,IOSTAT=error_de_lectura) nombre,nota
  externo:IF (error_de_lectura >0 ) THEN
    WRITE(*,*) 'error de lectura'
    EXIT
  ELSE IF (error_de_lectura <0 ) THEN
```

```
    WRITE(*,*) 'fin del archivo'
    EXIT
  ELSE
!    WRITE(*,*) nombre,nota
    interno:IF (nota < 5.0) THEN
      clase=3
    ELSE IF (nota < 8.5) THEN
      clase=2
    ELSE
      clase=1
    END IF interno
    WRITE (70,10,IOSTAT=error_de_escritura) nombre,clase
10 FORMAT(1X,A,I2)
    IF (error_de_escritura>0) STOP 'error de escritura'
  END IF externo
END DO
END PROGRAM cap7_3
```

– El archivo *cap7_3in.txt* debe existir en el mismo directorio donde se guarda este programa, antes de ejecutarlo.

– El archivo *cap7_3out.txt* se crea como resultado de la ejecución del programa anterior en la misma ubicación.

– Un ejemplo de archivo de entrada es:

'Raul Cagigal' 7.3

'Sara Mayoral' 2.6

'Guillermo Fuentes' 8.9

'Laura Cayon' 5.5

'Alvaro Torres' 9.7

'Gregorio Lanza' 4.9

'Ana Cuadra' 3.4

'Maria Aguirre' 4.7

'Lorenzo San Miguel' 6.5

'Jose Luis Casado' 6.2

4. Obtener el nombre y la nota del mejor y peor alumno de clase en Informática. Dar la media de la clase y la cantidad de suficientes.

– El nombre de los alumnos y sus notas están en un archivo de datos con nombre “cap7_4in.txt”.

– La media de la clase y la cantidad de suficientes han de calcularse a través de sendas funciones.

– La escritura de los resultados ha de volcarse a un archivo de salida, con nombre “cap7_4out.txt”.

– Usar formatos para escribir en el archivo de salida. Dedicar una cifra decimal para los números reales involucrados.

```
PROGRAM cap7_4
IMPLICIT NONE
CHARACTER (LEN=15), DIMENSION (5) :: nomb
CHARACTER (LEN=15)::nombm,nombp
REAL, DIMENSION(5) :: nota
REAL:: mejor,peor,mediaclase,media
INTEGER:: i,n,error_de_apertura,error_de_lectura,suficientes,num

OPEN (10,FILE='cap7_4in.txt',STATUS='OLD',ACTION='READ',&
IOSTAT=error_de_apertura)
IF (error_de_apertura>0) STOP 'cap7_4in.txt NO ABIERTO'

!..LECTURA DE DATOS.
i=1
DO
  READ(10,*,IOSTAT=error_de_lectura) nomb(i),nota(i)
  externo:IF (error_de_lectura >0 ) THEN
    WRITE(*,*) 'error de lectura'
    EXIT
  ELSE IF (error_de_lectura <0 ) THEN
!    WRITE(*,*) 'fin del archivo'
    n=i-1
    EXIT
  ENDIF
  i=i+1
END DO
!..CALCULO DEL PEOR DE LA CLASE..
peor=nota(1)
nombp=nomb(1)
DO i=2,n
```

```
  IF (nota(i) < peor) THEN
    peor=nota(i)
    nombp=nomb(i)
  END IF
END DO
!..CALCULO DEL MEJOR DE LA CLASE..
mejor=nota(1)
nombm=nomb(1)
DO i=2,n
  IF (nota(i) > mejor) THEN
    mejor=nota(i)
    nombm=nomb(i)
  END IF
END DO
num=suficientes(nota,n)
media=mediaclase(nota,n)

!ESCRITURA DE RESULTADOS EN UN ARCHIVO

OPEN (20,FILE='cap7_4out.txt',STATUS='NEW',ACTION='WRITE',&
IOSTAT=error_de_apertura)
IF (error_de_apertura>0) STOP 'cap7_4out.txt NO ABIERTO'
WRITE(20,'(1X,A,I3,A)') 'HAY',num,' ALUMNOS CON SUFICIENTE'
WRITE(20,'(1X,A,F4.1)') 'LA NOTA MEDIA DE LA CLASE ES:',media
WRITE(20,'(1X,3A,F4.1)') 'El peor alumno es: ',nombp,'y su nota
es:',peor
WRITE(20,'(1X,3A,F4.1)') 'El mejor alumno es: ',nombm, &
'y su nota es:',mejor
CLOSE (10)
CLOSE (20)
END PROGRAM cap7_4

!..ALUMNOS QUE HAN SACADO SUFICIENTE..

INTEGER FUNCTION suficientes(nota,n)
IMPLICIT NONE
INTEGER, INTENT(IN) :: n
REAL, DIMENSION(n),INTENT(IN) :: nota
INTEGER :: contador,i
```

```
contador=0
DO i=1,n
  IF (nota(i) >= 5.AND.nota(i) < 6) THEN
    contador=contador+1
  END IF
END DO
suficientes=contador
END FUNCTION suficientes

!..CALCULO DE LA MEDIA DE LA CLASE..

REAL FUNCTION mediaclase(nota,n)
IMPLICIT NONE
INTEGER, INTENT(IN) :: n
REAL, DIMENSION(n), INTENT(IN) :: nota
INTEGER:: i
mediaclase=0
DO  i=1,n
  mediaclase=mediaclase+nota(i)
END DO
mediaclase=mediaclase/n
END FUNCTION mediaclase
```

– Un ejemplo de archivo de entrada es:

```
'Santiago Lopez'   8.75
'Elisa Fernandez'  2.5
'Jose Garcia'      5.25
'Maria Rodriguez' 6.8
'Luis Martin'      5.9
```

EJERCICIOS PROPUESTOS

1) Programa que escriba en un archivo con nombre cap7_1p_out.txt los 100 primeros números naturales, generados por el propio programa.

2) Programa que abra el archivo del ejercicio anterior, calcule el cuadrado de cada número y lo escriba en un archivo con nombre cap7_2p_out.txt. ¿Cómo se escribiría todo en el mismo archivo? Para comodidad del usuario, copiar cap7_1p_out.txt en cap7_2p_inout.txt antes de ejecutar el programa.

3) Programa que escriba las temperaturas de los días de una semana en un archivo con nombre cap7_3p_out.txt.

4) Programa que lea las temperaturas de los días de la semana del ejercicio anterior y escriba en otro archivo con nombre cap7_4p_out.txt:

– Temperatura media = valor con dos cifras decimales.

– Temperatura mínima = valor y nombre del día de la semana correspondiente.

– Temperatura máxima = valor y nombre del día de la semana correspondiente.

¿Cómo se añadiría la información anterior en el archivo de entrada?

5) Programa que lea las temperaturas de los días de la semana del ejercicio anterior y escriba en otro archivo con nombre cap7_5p_out.txt las temperaturas ordenadas de menor a mayor, con su valor y nombre del día de la semana correspondiente.

6) Una empresa dispone de una base de datos de sus trabajadores. Se dispone de la siguiente información para este mes:

Nombre completo, Sueldo (Euros), Pagado (SI/NO), Antigüed (años).

El archivo que recoge esta información se llama cap7_6p_in.txt.

El aspecto del archivo es el siguiente:

```
‘Ana García’          1200.95        ‘SI’   5
‘Roberto Iglesias’    1800.59        ‘NO’  10
…
```

Se desea saber:

– ¿Cuál es la antigüedad de cualquier empleado en la empresa?

– ¿A cuánto asciende la cantidad pagada a todos los trabajadores este mes?

- ¿Cuántos trabajadores no han sido pagados aún este mes? Escribe sus nombres en un archivo que se llame cap7_6p_out.txt junto con sus sueldos. (La lista debe de estar ordenada alfabéticamente).

Escribe un programa Fortran que permita responder a todo esto, mostrando los resultados de los dos primeros apartados por monitor. Usa técnicas de programación modular en la elaboración del programa.

BIBLIOGRAFÍA

1) Stephen J. Chapman. *Fortran 90/95 for Scientists and Enginners*. 2nd Edition, International Edition, McGraw-Hill, 2004.

2) Michael Metcalf, John Reid, Malcolm Cohen. *Fortran 95/2003 explained*. Oxford University Press, 2005.

3) L.R. Nyhoff, S.C. Leestma. *Introduction to Fortran 90 for Engineers and Scientists*. Prentice Hall, 1997.

4) Elliot B. Koffman, Frank L. Friedman. *FORTRAN with engineering applications*. 5th Edition, Addison-Wesley, 1997.

5) F. García Merayo, V. Martín Ayuso, S. Boceta Martínez y E. Salete Casino. *Problemas resueltos de programación en Fortran 95*. Thomson, 2005.

6) F. García Merayo. *Lenguaje de programación Fortran 90*. Paraninfo, 1999.

7) Sebastián Ventura Soto, José Luis Cruz Soto, Cristóbal Romero Morales. *Curso básico de Fortran 90*. Universidad de Córdoba, 2000.

8) L.A.M. Quintales, L. Alonso Romero. *Programación en Fortran 90*. Curso en Internet. Dpto. Informática y Automática, Universidad de Salamanca, 2000.

9) E. Alcalde, M. García. *Metodología de la programación. Aplicaciones en Basic, Pascal, Cobol*. Serie: Informática de gestión. McGraw-Hill, 1989.

10) Clifford A. Pickover. *El prodigio de los números. Desafíos, paradojas y curiosidades matemáticas*. Ciencia Ma Non Troppo, 2002.