Índice general

1 Algoritmos y programas

Índice

1.1. Arquitectura de un ordenador

Un ordenador está formado por los siguientes elementos básicos

- *hardware* o conjunto de componentes físicos: memoria principal, Unidad Central de Proceso (CPU, *Central Processing Unit*) y dispositivos de Entrada/Salida (I/O *Input/Output*).
- *software* o conjunto de programas disponibles.

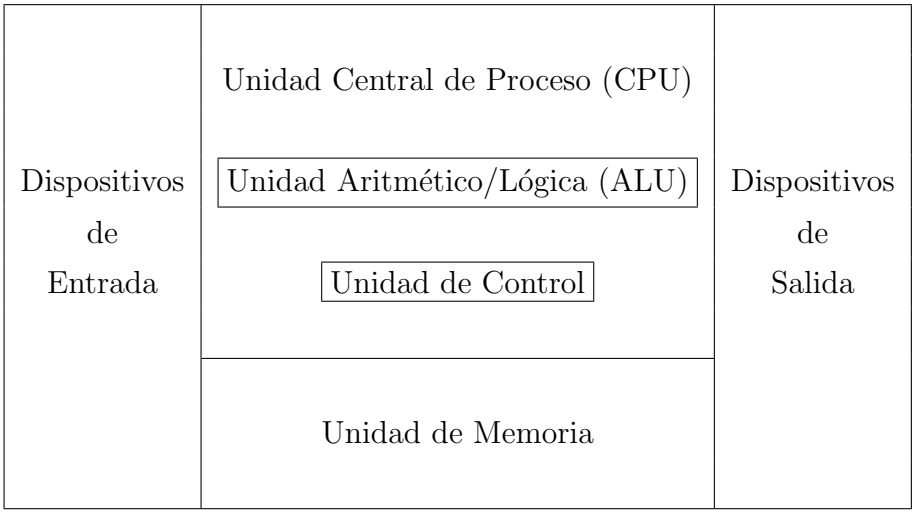
La memoria principal consiste en millones de conjuntos elementales capaces de almacenar un cero (0) o un uno (1). De esta forma, la memoria queda organizada en elementos de información que son capaces de almacenar datos e instrucciones de un programa. La representación interna que utiliza un ordenador es binaria. La unidad mínima de información es el *bit*, que posee dos estados 0 ó 1. Grupos de 8 *bits* constituyen un *byte*, que posee 2^8 estados diferentes. Las memorias principales pueden ser

- Memoria de Acceso Aleatorio (RAM, *Random Access Memory*), en la que se puede leer y escribir, pero la información se pierde cuando no se recibe energía eléctrica. En cualquier localización de la memoria RAM se puede almacenar temporalmente datos e instrucciones de programas, y el ordenador puede recuperar esta información si se le indica hacerlo.
- Memoria Solo de Lectura (ROM, *Read Only Memory*), de la que solo se puede leer. La información reside en ella de forma permanente, aunque no exista suministro de energía eléctrica. Esta memoria está reservada para instrucciones de arranque del ordenador y otra información crítica.

La CPU realiza secuencialmente las instrucciones, almacenadas en la memoria principal, que constituyen un programa escrito en lenguaje máquina. Consta de dos unidades

- Unidad Aritmético/Lógica (ALU, *Arithmetic/Logical Unit*). Es la componente del ordenador que realiza las operaciones aritméticas y lógicas.
- Unidad de Control. Es la componente del ordenador que gestiona el orden secuencial en que se realizan las instrucciones.

Los dispositivos I/O son aquellos que aceptan datos para ser procesados (Entrada) o presentan los resultados del procesamiento (Salida).



Los programas más importantes que residen en un ordenador son

- El *sistema operativo*. Es un programa que gestiona el uso de sistemas compartidos: procesador, memoria, dispositivos de entrada y salida, utilidades *software* comunes.
- El *compilador*. Es un programa que realiza el análisis sintáctico y la traducción del lenguaje de programación de alto nivel (Fortran, C, ...) a lenguaje máquina. Otra función del *compilador* es la de optimizar el programa de alto nivel para generar un programa en lenguaje máquina óptimo para una arquitectura dada.

1.2. Introducción a la programación

Es una creencia errónea, aunque generalizada entre personas ajenas al mundo de la informática, que un ordenador tiene una inteligencia superior a la del hombre. Esta idea debe ser descartada, ya que un ordenador es una máquina que solo es capaz de ejecutar un pequeño conjunto de instrucciones muy simples, en el momento que se le ordene y en la forma en que se le indique. Las ventajas de utilizar un ordenador son la rapidez de cálculo y la exactitud en la resolución de problemas, bajo una programación adecuada.

El objetivo de los lenguajes de programación es utilizar el ordenador como una herramienta para la resolución de problemas, en cuyo proceso se pueden identificar dos fases:

1.- Resolución del problema.
2.- Implementación en el ordenador.

El resultado de la primera fase es el diseño de un *algoritmo* o conjunto de instrucciones que conducen a la solución del problema. El algoritmo se expresa en un lenguaje de programación que el ordenador pueda comprender. Dicho algoritmo así expresado se denomina *programa*. La ejecución y verificación del programa es el objetivo final de la fase de implementación.

En definitiva, el proceso de diseño de un programa viene reflejado en la figura 1.1.

1.2.1. Análisis del problema

El análisis consiste en estudiar el problema planteado para tener una idea clara de los datos de partida, la forma en que éstos van a entrar en el programa, su tratamiento para obtener el resultado final, los datos de salida y la forma en que éstos van a salir del programa.

1.2.2. Diseño y verificación del algoritmo

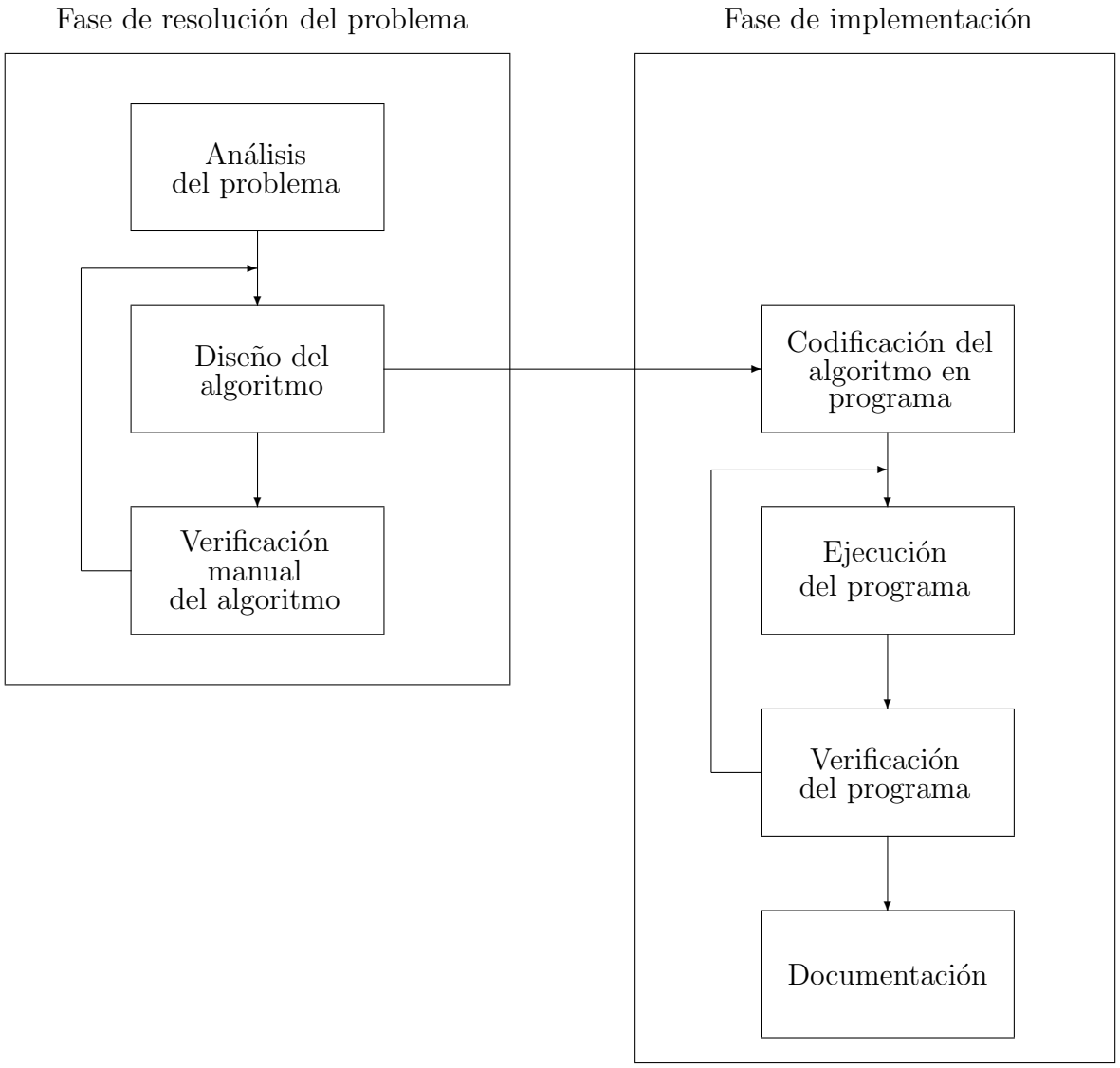
Una vez analizado el problema, es preciso diseñar un algoritmo que indique claramente los pasos a seguir para resolverlo. Las propiedades básicas de un buen algoritmo son

1.- Debe ser *preciso* e indicar el orden de realización de cada paso.

2.- Debe estar *definido*, si se sigue un mismo algoritmo dos veces se debe obtener idéntico resultado.

3.- Debe ser *finito*, debe terminar en algún momento.

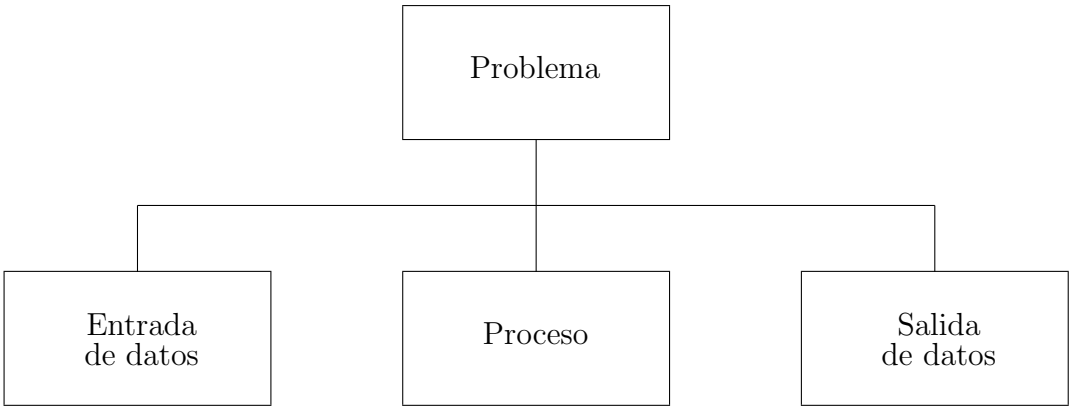
Figura 1.1 Proceso de diseño de un programa



Para facilitar el seguimiento de un algoritmo así como su implementación en el ordenador, es necesario dividir cualquier problema planteado en varios subproblemas más fáciles de resolver. Un algoritmo consta, como mínimo, de tres partes:

entrada	(información dada al algoritmo)
proceso	(cálculos necesarios para encontrar la solución del problema)
salida	(resultados finales de los cálculos)

o bien

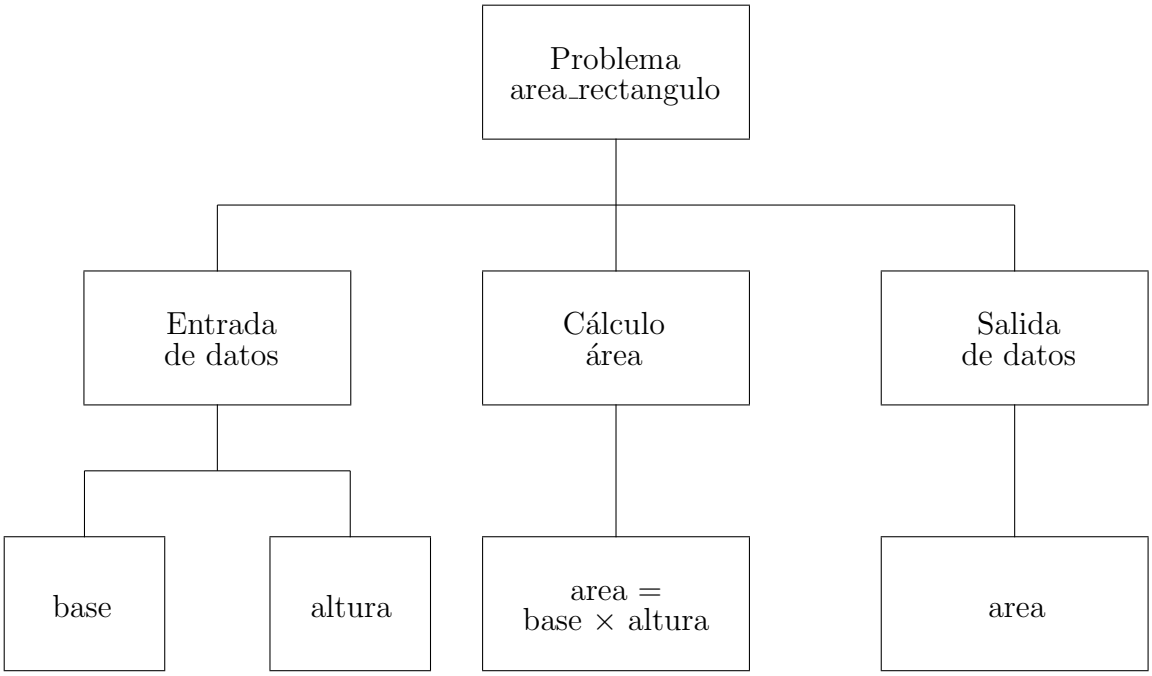


Por ejemplo, cálculo del área de un rectángulo.

Paso 1: `Entrada de datos (base y altura)`

Paso 2: `Cálculo de area = base` $\times$ `altura`

Paso 3: `Salida de datos (area)`



Una vez que se ha descrito el algoritmo, es necesario verificarlo, es decir, comprobar, con lápiz y papel, que produce el resultado correcto y esperado. Para ello, es conveniente utilizar datos significativos que abarquen todo el rango posible de los valores de entrada.

1.2.3. Codificación del algoritmo en el programa

Para que el algoritmo diseñado pueda ser introducido en el ordenador debe ser codificado o escrito en un lenguaje de programación: PASCAL, Fortran, C ..., siguiendo las

reglas de sintaxis propias de cada lenguaje. El algoritmo, una vez codificado, se denomina *programa*.

Existen dos fases previas que debe realizar el ordenador antes de la ejecución del programa

- *Compilar* el código. Durante la compilación, se realiza un estudio sintáctico del programa. Si no hay problemas de compilación, el compilador crea un fichero *objeto*, que contiene la traducción de nuestro programa a código máquina.
- *Linkar* el código. Durante el *linkado*, se realiza un estudio de las sentencias contenidas en el programa. Si no hay problemas de *linkado*, el compilador crea un fichero *ejecutable*, que contiene las instrucciones necesarias para que se ejecute el programa.

1.2.4. Ejecución y verificación del programa

El éxito en la *compilación* y *linkado* de un programa, no significa que el programa resuelva correctamente el problema. Por tanto, es indispensable verificar su correcto funcionamiento, probando con un conjunto de datos de entrada lo más amplio posible.

1.2.5. Documentación de un programa

El objetivo de esta fase es que cualquier programador sea capaz de entender e incluso modificar, si es necesario, el programa.

Una correcta documentación debe contener:

- Autor y fecha.
- Breve descripción del problema.
- Descripción de los datos de entrada.
- Descripción de los datos de salida.
- Comentarios sobre las sentencias de ejecución más relevantes.

2 Bases de Fortran

Índice

2.1. Introducción

El objetivo principal de la programación tal y como debe entenderse en un ámbito científico e ingenieril, es la resolución numérica a través de un ordenador del problema planteado. En este sentido Fortran constituye un lenguaje de programación estructurado y de alto nivel orientado desde sus orígenes, hace más de 40 años, a la resolución de problemas científicos. De hecho su nombre no es más que la contracción de *FORmula TRANslation*.

La primera versión de Fortran surgió a mediados de los sesenta como un intento de unificar diferentes dialectos creados con el fin de conseguir un lenguaje intermedio entre el lenguaje matemático y el lenguaje máquina (único conocido hasta esa fecha). El resultado final, conocido como Fortran77, constituyó el paradigma de programación durante muchos años, de ahí que muchas librerías estándar matemáticas como IMSL (*International Mathematics and Statistics Library*) o NAG (*Numerical Algorithmis Group*) usadas por científicos e ingenieros estuviesen escritas en esa versión. Durante los siguientes veinte años, aproximadamente, fueron surgiendo otros lenguajes de alto nivel (Pascal, Ada, C, etc), que abordaban de modo más eficiente ciertos aspectos particulares del cálculo científico. Ante la necesidad de que Fortran contemplase estos avances se creó en 1991 una nueva versión, objeto de estudio en el presente curso, denominada Fortran90. Esta versión se diseñó con el objetivo de incorporar utilidades avanzadas (manipulación de *arrays*, tipos de datos definidos por el usuario, asignación dinámica de memoria y manejo de punteros, etc), manteniendo la compatibilidad con la versión anterior debido a la gran cantidad de programas que ya estaban escritos en Fortran77.

Posteriormente han surgido dos nuevas versiones Fortran95 y, más recientemente Fortran 2003. El objetivo de la primera fue eliminar los aspectos considerados obsoletos de la versión anterior y que, por tanto, no entrarán a formar parte del temario aquí expuesto. La última versión, Fortran 2003, aborda aspectos que superan los objetivos del presente curso.

En este capítulo se expondrán las normas generales de escritura de un programa Fortran, así como la estructura general de dicho programa.

2.2. Notación

A lo largo de todo el texto vamos a utilizar las siguientes convenciones, aunque en ocasiones, por cuestiones de claridad, no seremos muy rigurosos en su cumplimiento.

- Tanto las palabras clave como el texto específico de Fortran se escribirá en tipográfico:

  `integer :: i, j`
- Para especificar de forma simplificada nombres o instrucciones en lenguaje natural se utilizará el modo itálico:

  `if (` *expr-lógica* `)` *sentencia*
- Cuando parte de una sentencia sea opcional, irá metida entre corchetes

  `end if` [*nombre*]
- Para especificar la existencia de espacios en blanco se utilizará el carácter ␣

2.3. Normas de escritura

En la figura 2.1 se muestra un ejemplo de algunas de las normas básicas para escribir un código en Fortran.

- Cada línea contiene 132 columnas.
- Los espacios en blanco tienen significación en algunos casos.
- Si una sentencia es demasiado larga, se puede continuar en la línea siguiente, escribiendo el operador `&` al final de la primera línea.
- Una misma línea puede contener varias sentencias, aunque separadas por el operador `;`
- Un comentario comienza con el operador **!** y se puede poner al final de una sentencia de ejecución, o en una línea aparte.
- En Fortran, las letras mayúsculas y minúsculas se usan indistintamente.

2.4. Estructura general de un programa

El esquema básico de un programa principal en Fortran es el siguiente

```
program nombre
    Declaración e inicialización de parámetros
    Declaración [ e inicialización ] de variables
    sentencias de ejecución
end program nombre
```

Figura 2.1 Normas de escritura

```
x = 23.0*(a - 5.0) + &
    sqrt(a**3)
    .
    .
i = 0; j = 0; k = 5
    .
    .
z = 0.0         ! Inicializo la variable
    .
    .
z = 3.0 + x
```

- El nombre del programa no tiene porqué coincidir con el nombre del fichero que lo contiene. En realidad, `nombre` es una variable más del programa, por lo que no puede coincidir con el nombre de ninguna de las variables, constantes o subprogramas referenciados en el propio programa.
- En Fortran la última sentencia de **todo** programa, ya sea el principal o cualquier subprograma, debe ser `end`.

2.5. Salidas de un programa

Un programa finaliza su ejecución cuando llega a la sentencia `end`. Sin embargo, es posible pararlo definitivamente (abortarlo) en cualquier punto de la ejecución utilizando la sentencia `stop`, cuya estructura general es

`stop [ "`*frase_a_imprimir_por_pantalla*`" ]`

- La *frase_a_imprimir_por_pantalla* es opcional pero, si se utiliza, debe ir entre comillas (dobles o simples).
- La sentencia `stop` puede ir en cualquier punto de un programa o subprograma.

Cuando se produce un error grave en la ejecución de un programa, el propio sistema provoca una parada forzada en ese punto. La mayoría de estos errores se pueden prever

durante la implementación del código (división entre cero, fichero de lectura incorrecto, memoria disponible insuficiente, ...). La sentencia `stop` constituye una manera de controlar estas eventualidades por el propio programador. Aborta la ejecución del programa y, por medio de una frase suficientemente clara, advierte del problema o error que se ha producido.

Hay que tener en cuenta ciertas normas de uso extendido

- No hay que abusar del uso de la sentencia `stop`. Cuando se elabora el algoritmo hay que intentar dar una solución a los posibles problemas que se pueden producir, sin necesidad de abortar el programa.
- No es conveniente utilizar la sentencia `stop` dentro de un subprograma. El programa principal es el que tiene el control sobre la ejecución y debe ser él el que provoque su parada.
- Antes de abortar la ejecución de un programa es necesario finalizar de manera adecuada los procesos que permanecen activos: liberar memoria, cerrar ficheros, etc.

3 Tipos de datos

Índice

3.1. Introducción

Los programas, con independencia del lenguaje en que se escriban, están diseñados para manipular información o *datos*. En Fortran se pueden definir y manipular varios tipos de datos. Por ejemplo, se puede asignar el valor `10` a una variable llamada `i` definida como entera. En este caso tanto `10` como `i` son de tipo entero. `10` es un valor *constante* mientras que `i` es una *variable* que puede modificar su valor a lo largo del programa. Además de las *constantes* y *variables* existe la posibilidad de definir un *parámetro*, un dato cuyo valor no cambia durante la ejecución del programa.

Un *tipo de dato* consiste en

- Un *identificador de tipo*
- Un conjunto válido de valores
- Una forma de denotar sus valores
- Un conjunto de operaciones válidas

Por ejemplo, para un dato de tipo entero, el identificador de tipo es `integer`, el conjunto válido de valores es {`..., -3, -2, -1, 0, 1, 2, 3, ...`} entre ciertos límites que dependen de la especie (en inglés, *kind*) de entero definido y del procesador. Tales símbolos se denominan *constantes literales* y cada tipo de dato tiene su propia forma de expresarlos. Las operaciones válidas entre datos *enteros* son las típicas de la aritmética, por ejemplo si a la variable `i` le hemos asignado el valor `10`, el resultado de la operación `i + 3` es `13`, como era de esperar.

Las propiedades arriba mencionadas están asociadas a todos los tipos de datos de Fortran y será lo que desarrollaremos a lo largo de este capítulo. El propio lenguaje de Fortran contiene 5 tipos de datos cuya existencia siempre se puede asumir. Estos tipos se conocen como *tipos de datos intrínsecos*. Tres de ellos son numéricos: `integer`, `real` y `complex` y dos no numéricos: `logical` y `character`. De cada tipo intrínseco existe una especie por defecto (en inglés, *default kind*) y un número variable de especies dependientes del ordenador (en adelante, utilizaremos preferentemente el término inglés *kind* para referirnos a la especie de un tipo de dato). Cada *kind* está asociado a un entero no negativo conocido como *parámetro identificador de especie* (en inglés, *kind type parameter*).

Además de los datos de tipo intrínseco, Fortran permite definir nuevos tipos de datos, denominados *tipos derivados*, generalmente utilizados para materializar objetos matemáticos o geométricos (un intervalo, una circunferencia, etc) en un tipo de dato.

En adición a los *datos simples*, ya sean de tipo intrínseco o derivado, es posible generar una colección de datos del mismo tipo e igual *kind* denominada *array*, usada normalmente para materializar vectores, matrices y tablas de más dimensiones.

Cuando nos planteamos resolver un problema científico con un ordenador, generalmente nos centramos en el conjunto de operaciones que se tienen que realizar y en qué orden. Sin embargo, es igual de importante tener muy claro, antes de escribir el programa, el tipo de datos que vamos a manejar. En muchas ocasiones, un programa bien escrito y estructurado produce resultados erróneos debido a una elección incorrecta del tipo de dato utilizado. Así pues,

- Para las operaciones numéricas más comunes es aconsejable utilizar datos de tipo `real`.
- Como contadores, es obligatorio utilizar datos de tipo `integer`.
- Para tomar decisiones, es aconsejable utilizar datos de tipo `logical`.
- Para almacenar cadenas de caracteres, es conveniente utilizar datos de tipo `character`.

La especificación en un programa del tipo de dato de un objeto se denomina *declaración*.

Figura 3.1 Declaración de una variable

```
1 program main
2     integer :: n
3     n = 20
4     write(*,*) 5*n
5     n = n + 1
6 end program main
```

En la figura 3.1 se muestra un ejemplo del uso de una variable entera llamada `n`. En la línea 2 se realiza la declaración de la variable. En ese momento, la variable está *indefinida*, es decir no tiene un valor asignado. En la línea 3 se *asigna* a `n` el valor `20` y pasa a estar *definida*. En la línea 4, después de asignarle un valor, ya puede ser *referenciada*, es decir podemos utilizar la variable, y por tanto el dato que contiene, para realizar operaciones.

En el momento de la declaración (línea 2), el procesador habilita una zona de memoria con la etiqueta `n` a la que puede acceder a lo largo de la ejecución del programa. En este paso, la variable está indefinida ya que, aunque la zona de memoria está reservada, en ella no hay ningún dato almacenado. En la línea 3 se produce una asignación y el dato

`20` se almacena en la posición de memoria con la etiqueta `n`. La variable ya está definida. En la línea 5 se accede al dato almacenado en `n` y se opera con él. La sentencia se puede traducir como: imprime por pantalla el resultado de la operación `5*n` o sea `100`. En la línea 6 se produce una nueva asignación: almacena en la posición de memoria con la etiqueta `n` el resultado de la operación `n+1` o sea `21`.

De lo explicado en el párrafo anterior se concluye que:

> Todos los datos utilizados en un programa deben:
>
> - ESTAR EXPLÍCITAMENTE DECLARADOS
> - TENER UN VALOR ASIGNADO ANTES DE SER UTILIZADOS

Respecto al nombre o *identificador* de un dato, ya sea variable o parámetro; simple, derivado o *array*, las reglas para formarlo son:

- El nombre de un dato no puede exceder los 31 caracteres.
- El primer carácter debe ser alfabético.
- Caracteres válidos son:

  `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`

  `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`

  `0,1,2,3,4,5,6,7,8,9`

  _ (subrayado)
- No hay distinción entre mayúsculas y minúsculas.

3.2. Tipos de datos intrínsecos

3.2.1. Datos enteros

La forma más sencilla de declarar un dato numérico de tipo entero es

```
integer :: nombre_variable
```

Las constantes literales asociadas a un dato entero pertenecen al conjunto de los números enteros y, por tanto, son cadenas de dígitos que no contienen ni coma ni punto decimal. Las constantes enteras negativas deben estar precedidas por el signo menos, sin embargo, en las no negativas el signo es opcional.

El rango de valores que puede tomar un dato de tipo entero depende, por una parte, del procesador (32-bit ó 64-bit) y, por otra, del parámetro *kind* especificado en la declaración.

Figura 3.2 Asignación de un dato de tipo entero

```
program main
    integer :: n
    n = 20
    n = -2
    n = 0
    n = 9999
end program main
```

Los identificadores de especie más comunes son:

- `kind = 1`
  El dato ocupa 1 *byte* de memoria y se pueden representar datos enteros comprendidos entre -2^7 y $2^7 - 1$
- `kind = 2`
  El dato ocupa 2 *bytes* de memoria y se pueden representar datos enteros comprendidos entre -2^{15} y $2^{15} - 1$
- `kind = 4` (valor por defecto)
  El dato ocupa 4 *bytes* de memoria y se pueden representar datos enteros comprendidos entre -2^{31} y $2^{31} - 1$
- `kind = 8`
  El dato ocupa 8 *bytes* de memoria y se pueden representar datos enteros comprendidos entre -2^{63} y $2^{63} - 1$. Generalmente los ordenadores personales no tienen capacidad para trabajar en este rango, pero sí las estaciones de trabajo y los ordenadores de gran potencia de cálculo.

Figura 3.3 Declaración de un dato de tipo entero especificando el *kind*

```
program main
    integer(kind=1) :: i
    integer(1)      :: j
    integer(kind=4) :: k
    integer(4)      :: l
    integer(kind=8) :: m
    integer(8)      :: n
end program main
```

3.2.2. Datos reales

La forma más sencilla de declarar un dato numérico de tipo real es

```
real :: nombre_variable
```

Las constantes literales asociadas a un dato real pertenecen al conjunto de los números reales. Se pueden representar en modo decimal o en modo exponencial. Las constantes reales negativas deben estar precedidas por el signo menos, sin embargo, en las no negativas el signo es opcional.

Si un dato real se representa en modo decimal es necesario utilizar el punto decimal (la coma decimal no está permitida). La representación en modo exponencial consiste en un número entero o real seguido por la letra `e` y el número entero como exponente. En la figura 3.4 se muestran distintas formas de representar la constante real `20.508`, todas ellas equivalentes entre sí.

Figura 3.4 Asignación de un dato de tipo real

```
program main
    real :: x
    x = 20.508
    x = 205.08e-1
    x = 2.0508e+1
    x = 0.20508e+2
end program main
```

El rango de valores que puede tomar un dato de tipo real depende, por una parte, del procesador (32-bit ó 64-bit) y, por otra, del parámetro *kind* especificado en la declaración.

Los identificadores de especie más comunes son:

- `kind = 4` (valor por defecto)

  El dato ocupa 4 *bytes* de memoria y se pueden representar datos reales comprendidos entre 1.18×10^{-38} y 3.40×10^{38}. La precisión en coma flotante es 1.19×10^{-7}. Cuando un dato se declara con `kind=4` se dice que es un dato en *simple precisión*.

- `kind = 8`

  El dato ocupa 8 *bytes* de memoria y se pueden representar datos reales comprendidos entre 2.23×10^{-308} y 1.79×10^{308}. La precisión en coma flotante es 2.22×10^{-16}. Cuando un dato se declara con `kind=8` se dice que es un dato en *doble precisión*.

- `kind = 16`

  El dato ocupa 16 *bytes* de memoria. Generalmente los ordenadores personales no tienen capacidad para trabajar con este *kind*, pero sí las estaciones de trabajo y los ordenadores de gran potencia de cálculo.

Figura 3.5 Declaración de un dato de tipo real especificando el *kind*

```
program main
    real(kind=4) :: x
    real(4)      :: y
    real(kind=8) :: z
    real(8)      :: t
end program main
```

Tanto el rango valores como la precisión de los datos reales en simple y doble precisión presentados anteriormente son meramente orientativos.

3.2.3. Datos complejos

La forma más sencilla de declarar un dato numérico de tipo complejo es

```
complex :: nombre_variable
```

Las constantes literales asociadas a un dato complejo pertenecen al conjunto de los números complejos. Constan de parte real y parte imaginaria, ambas definidas por un número real que puede estar representado en modo decimal o en modo exponencial. Las componentes negativas de una constante compleja deben estar precedidas por el signo menos, sin embargo, en las no negativas el signo es opcional.

En la figura 3.6 se muestran distintas formas de representar la constante compleja `20.508 + 3.7i`, todas ellas equivalentes entre sí.

Los datos de tipo `complex` admiten los mismos identificadores de especie ya definidos para datos de tipo `real`, y se aplican tanto a la parte real como a la imaginaria. Por tanto, si el *kind* es `8`, el dato complejo ocupará 16 *bytes* de memoria (8 la parte real + 8 la parte imaginaria). Si no se especifica el *kind*, por defecto se asume que vale `4`.

Figura 3.6 Asignación de un dato de tipo complejo

```
program main
    complex :: z
    z = (20.508, 3.7)
    z = (205.08e-1, 3.7)
    z = (2.0508e+1, 37e-1)
    z = (20.508, 0.37e+1)
end program main
```

Figura 3.7 Declaración de un dato de tipo complejo especificando el *kind*

```
program main
    complex(kind=4) :: x
    complex(4)      :: y
    complex(kind=8) :: z
    complex(8)      :: t
end program main
```

3.2.4. Datos lógicos

La forma más sencilla de declarar un dato lógico es

```
logical :: nombre_variable
```

Las constantes literales asociadas a un dato lógico son, en su forma más básica, `.true.` (verdadero) o `.false.` (falso).

Figura 3.8 Asignación de un dato de tipo lógico

```
program main
    logical :: x
    x = .true.
    x = .false.
end program main
```

Al igual que en el caso de los datos de tipo numérico, los datos de tipo lógico admiten un identificador de especie. El valor por defecto del identificador o *default kind* depende del procesador.

3.2.5. Datos caracteres

La forma más sencilla de declarar un dato de tipo carácter es

```
character :: nombre_variable
```

El conjunto de constantes literales asociadas a un dato de tipo carácter lo forman todos los caracteres válidos del procesador y deben representarse entre comillas simples o dobles. Por ejemplo

```
'Esto es un mensaje'
"Esto tambien"
'D.E. Knuth & M. Metcalf'
```

Las comillas dobles o simples son solo delimitadores, no forman parte del valor del dato. El valor de la constante literal `'hola'` es `hola`. Sí hay que tener en cuenta que los espacios en blanco son caracteres, y que, como carácter, las mayúsculas son distintas de las minúsculas. Así, las constantes `'un saludo'` y `' un saludo'` son distintas y las constantes `'Hola'` y `'hola'` también son distintas.

En caso de que necesitemos utilizar comillas simples o dobles como parte de la constante, basta con usar lo contrario como delimitador. Por ejemplo,

```
'Introduce "Hola"'
"Introduce 'Adios'"
```

Una forma alternativa de introducir una comilla simple en una constante literal es mediante dos comillas simples seguidas, sin ningún carácter entre ellas. El valor de la constante `'don''t work'` es `don't work`.

La longitud de una constante literal de tipo carácter coincide exactamente con el número de caracteres de impresión contenidos entre los delimitadores.

Constante literal	valor	longitud
`'Esto es un mensaje'`	`Esto␣es␣un␣mensaje`	18
`' Esto tambien '`	`␣Esto␣␣␣tambien␣␣`	17
`'Introduce "Hola"'`	`Introduce␣"Hola"`	16
`'don''t work'`	`don't␣work`	10

Sin embargo, a un dato de tipo carácter se le puede especificar una longitud en la propia declaración, utilizando la sentencia

```
character(len=numero) :: nombre_variable
```

Si no se especifica la longitud de un dato, su longitud por defecto es 1. Veamos algunos ejemplos

Figura 3.9 Asignación de un dato de tipo carácter

```
program main
    character          :: c_1
    character(len=4)   :: c_2
    character(len=10)  :: c_3
    character(len=13)  :: c_4
    character(len=13)  :: c_5
    c_1 = 'P'
    c_2 = "Hola"
    c_3 = ' Adios '
    c_4 = "Hola & Adios"
    c_5 = 'Hola & Adios'
end program main
```

De esta forma, la longitud del dato se fija a priori y es invariable. Si al dato se le asigna una constante literal de menor longitud, el dato la rellena con espacios en blanco a la derecha; si al dato se le asigna una constante literal de mayor longitud, el dato la trunca por la derecha. Así, por ejemplo, declarada la variable `ch` de tipo carácter con una longitud `8`

```
character(len=8) :: ch
```

se tiene

Figura 3.10 Longitud de un dato de tipo carácter

Asignación	valor de `ch`	longitud de `ch`
`ch = 'Hola'`	`Hola␣␣␣␣`	8
`ch = ' Hola'`	`␣Hola␣␣␣`	8
`ch = ' Hola '`	`␣Hola␣␣␣`	8
`ch = '  Hola '`	`␣␣Hola␣␣`	8
`ch = 'Informatica'`	`Informat`	8
`ch = ' Informatica'`	`␣Informa`	8

Un dato de tipo `character`, por ejemplo `ch`, declarado con una longitud específica, por ejemplo `10`, se puede entender como una *cadena* de `10` caracteres de longitud `1`.

Podemos acceder a la *subcadena* formada por los caracteres que van desde el carácter i-ésimo hasta el j-ésimo utilizando la expresión `ch(i:j)`, siendo `i` y `j` dos datos simples de tipo `integer` (Fig. 3.11).

Figura 3.11 Cadena y subcadena de caracteres

```
program main
    character(len=10) :: ch
    integer :: i
    integer :: j
    ch = 'carbonilla'
    i = 1
    j = 6
    write(*,*) ch(i:i)                    ! 'c'
    write(*,*) ch(i:j)                    ! 'carbon'
    write(*,*) ch(i+1:j-1)                ! 'arbo'
    write(*,*) ch(i+2:i+2)                ! 'r'
end program main
```

3.3. Tipos de datos derivados

En ocasiones resulta conveniente manipular objetos más sofisticados que los datos intrínsecos descritos en la sección anterior. Supongamos que queremos reunir información relativa a una persona (por ejemplo, nombre, apellido, edad y altura). En ese caso, resulta muy útil definir un nuevo tipo de dato que englobe a todos esos datos, de forma que podamos manipularlos como un único objeto.

Utilizando este ejemplo, la forma de *definir* el nuevo *tipo derivado* llamado `persona` sería

```
type persona
    character(len=10) :: nombre
    character(len=20) :: apellido
    integer           :: edad
    real              :: altura
end type persona
```

Mientras que para *declarar* un dato, llamado `yo`, del tipo derivado `persona`, la sentencia sería

```
type(persona) :: yo
```

Un dato de tipo derivado se denomina *estructura* y los datos que la componen se denominan *componentes*. En resumen, en este ejemplo, hemos definido un tipo derivado (`persona`), cuyas componentes son `nombre`, `apellido`, `edad` y `altura`. Además hemos declarado una estructura llamada `yo` que es de tipo `persona`.

Para referenciar una componente individual de una estructura es necesario utilizar un *selector de componentes*, el carácter tanto por ciento (`%`), en la secuencia:

```
estructura%componente
```

En el ejemplo anterior, la forma de acceder a cada una de las componentes sería

```
persona%nombre
persona%apellido
persona%edad
persona%altura
```

Hay que tener en mente que cuando trabajamos con una componente, estamos trabajando con un dato de tipo intrínseco. Por ejemplo, `persona%edad` es una variable de tipo entero, con las mismas propiedades y características que cualquier otra variable `integer` del programa. Sus constantes literales pertenecen al conjunto de los números enteros, puede estar declarada con un *kind* específico, etc.

Una estructura se dice que está definida si, y solo si, lo están cada una de sus componentes. Para definir una estructura podemos utilizar dos estrategias (Fig. 3.12)

a) Definir las componentes una a una. Para ello, se accede a cada una de las componentes de forma independiente y se le asigna una constante literal apropiada a su tipo intrínseco y en la forma habitual para ese tipo intrínseco. Como estamos trabajando de forma independiente, el orden en el que se definen las componentes no afecta al resultado final.

b) Definir todas las componentes a la vez. Para ello, se utiliza la fórmula: nombre del tipo derivado seguido, entre paréntesis, por las constantes literales asignadas a las componentes, en el mismo orden en el que han sido declaradas.

Figura 3.12 Asignación de un dato de tipo derivado

```
program main
    type persona
       character(len=10) :: nombre
       character(len=20) :: apellido
       integer           :: edad
       real              :: altura
    end type persona

    type(persona) :: ella
    type(persona) :: el

    ! (a) se definen las componentes una a una.
    el%nombre = 'Hugh'
    el%apellido = 'Jackman'
    el%edad = 41
    el%altura = 1.89

    ! (b) se definen todas las componentes a la vez.
    ella = persona('Angelina', 'Jolie', 34, 1.73)

end program main
```

3.4. Arrays

Un *array* es una colección de datos, denominados *elementos*, todos ellos del mismo tipo y *kind*, situados en posiciones contiguas de memoria. Los *arrays* se clasifican en *unidimensionales* (vectores), *bidimensionales* (matrices) y *multidimensionales* (tablas).

Existen varias formas de declarar un *array* pero, por el momento, utilizaremos el modo más sencillo de declarar *arrays* de *forma explícita*, consistente en especificar el tipo, el *kind*, el nombre y la extensión de cada dimensión del *array*, utilizando (Fig. 3.14) o no (Fig. 3.13) el atributo `dimension`.

La nomenclatura específica de un *array* consiste en

- *Rango* del *array*. Es el número de dimensiones (con un máximo de 7). `U` y `V` tienen rango 1, `A`, `Z`, `T` y `Mch` tienen rango 2 y `Tabla` tiene rango 3.

Figura 3.13 Declaración de un *array*

```
program main
    integer           :: U(5)                  ! Vector
    real              :: V(10)                 ! Vector
    real              :: A(2,3)                ! Matriz de 2x3
    real              :: Z(3,2)                ! Matriz de 3x2
    real              :: T(0:2,-1:0)           ! Matriz de 3x2
    character(len=2)  :: Mch(3,2)              ! Matriz de 3x2
    integer           :: Tabla(2,3,4)          ! Tabla de 2x3x4
end program main
```

Figura 3.14 Declaración de un *array* con el atributo `dimension`

```
program main
    integer, dimension(5)             :: U       ! Vector
    real, dimension(10)               :: V       ! Vector
    real, dimension(2,3)              :: A       ! Matriz de 2x3
    real, dimension(3,2)              :: Z       ! Matriz de 3x2
    real, dimension(0:2,-1:0)         :: T       ! Matriz de 3x2
    character(len=2), dimension(3,2)  :: Mch     ! Matriz de 3x2
    integer, dimension(2,3,4)         :: Tabla   ! Tabla de 2x3x4
end program main
```

- *Límites* del *array*. Son los límites inferior y superior de los índices de las dimensiones. Si no se especifica el límite inferior, su valor por defecto es 1. `U` tiene límite superior 5, `V` tiene límite superior 10, `A` tiene límites superiores 2 y 3, `Z` y `Mch` tiene límites superiores 3 y 2, `T` tiene límites superiores 2 y 0 y `Tabla` tiene límites superiores 2, 3 y 4. Todos los *arrays* tienen límite inferior 1 en cada una de sus dimensiones excepto `T` que tiene límites inferiores 0 y -1.

- *Extensión* del *array*. Es el número de elementos (puede ser cero) en cada dimensión. `U` tiene extensión 5, `V` tiene extensión 10, `A` tiene extensiones 2 y 3, `Z`, `T` y `Mch` tienen extensiones 3 y 2, y `Tabla` tiene extensiones 2, 3 y 4.

- *Tamaño* del *array*. Indica el número total de elementos. `U` tiene tamaño 5, `V` tiene tamaño 10, `A`, `Z`, `T` y `Mch` tienen tamaño 6, y `Tabla` tiene tamaño 24.

- *Forma* del *array*. Viene dada por el rango y la extensión. `U` tiene forma 5, `V` tiene forma 10, `A` tiene forma 2×3, `Z`, `T` y `Mch` tiene forma 3×2, y `Tabla` tiene forma $2 \times 3 \times 4$.

- *Conforme*. Dos *arrays* se dice que son conformes, si tienen la misma forma.

Un *array* se identifica por su nombre, que debe seguir las mismas reglas que los identificadores de los datos simples. Por ejemplo: `A, Tabla`, ... Un elemento de un *array* se referencia por el nombre del mismo y su posición. Esta posición viene dada por uno o varios subíndices dados entre paréntesis. Por ejemplo, `U(3)` es el tercer elemento del vector `U`; `A(2,3)` es el elemento de la fila 2 y columna 3 de la matriz `A` y `T(0,-1)` es el elemento de la fila 1 y columna 1 de la matriz `T`. El subíndice debe ser o bien una constante literal entera, o bien un dato simple de tipo `integer`.

Un *array* se dice que está definido si, y solo si, lo están todos sus elementos. Para definir un *array* podemos utilizar dos estrategias dependiendo de si todos los elementos se van a inicializar con la misma constante literal o no (Fig. 3.15).

Figura 3.15 Asignación de un *array*

```
program main
    integer           :: U(5)                  ! Vector
    real              :: V(10)                 ! Vector
    real              :: A(2,3)                ! Matriz de 2x3
    real              :: Z(3,2)                ! Matriz de 3x2
    real              :: T(0:2,-1:0)           ! Matriz de 3x2
    character(len=2)  :: Mch(3,2)              ! Matriz de 3x2
    integer           :: Tabla(2,3,4)          ! Tabla de 2x3x4

    V = 2.0
    Tabla = 8
    U = (/2, -3, 0, 5, 8/)
    A = reshape((/1.1, 1.2, 3.8, 5.5, 4.3, 7.0/), (/2,3/))
    Z = reshape((/1.1, 1.2, 3.8, 5.5, 4.3, 7.0/), (/3,2/))
    T = reshape((/1.1, 1.2, 3.8, 5.5, 4.3, 7.0/), (/3,2/))
    Mch = reshape((/'ab','cd','ef','uv','wx','yz'/), (/3,2/))
end program main
```

a) Para asignar el mismo valor a todos los elementos del *array*, se utiliza el modo habitual de asignación:

```
nombre_array = constante_literal
```

dado que una constante literal se considera conforme a cualquier *array*. Por ejemplo, las sentencias

```
V = 2.0
Tabla = 8
```

dan lugar a las siguientes asignaciones

- En lenguaje matemático:
  $V_1 = 2 \quad V_2 = 2 \quad \ldots \quad V_{10} = 2$
  $Tabla_{111} = 8 \quad Tabla_{211} = 8 \quad \ldots \quad Tabla_{234} = 8$

- En lenguaje de programación:
  `V(1)=2.0  V(2)=2.0  ...  V(10)=2.0`
  `Tabla(1,1,1)=8  Tabla(2,1,1)=8  ...  Tabla(2,3,4)=8`

b) Para asignar distintas constantes literales, tantas como indique el tamaño del *array*, se introduce la lista de las constantes entre los símbolos de apertura y cierre `(/` y `/)`, respectivamente. Además,

b1) Si el *array* es unidimensional, cada elemento del *array* se carga en su orden natural con la constante correspondiente de la lista. Por ejemplo, la sentencia

```
U = (/2, -3, 0, 5, 8/)
```

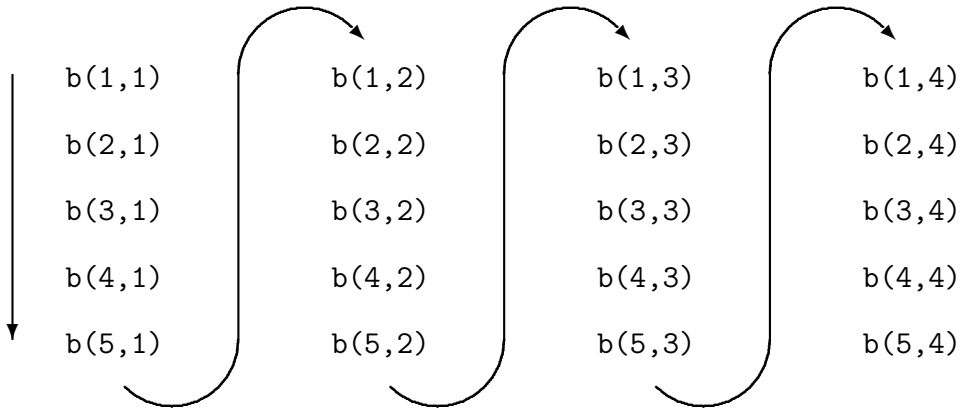
da lugar a las siguientes asignaciones

- En lenguaje matemático:
  $U_1 = 2 \quad U_2 = -3 \quad U_3 = 0 \quad U_4 = 5 \quad$ y $\quad U_5 = 8$
- En lenguaje de programación:
  `U(1)=2  U(2)=-3  U(3)=0  U(4)=5` y `U(5)=8`

b2) Si el *array* es una matriz o una tabla es necesario utilizar, además de la lista de constantes, la función `reshape`. Con esta función se especifica la forma en la que se van a asignar las constantes de la lista a los elementos del *array*,

teniendo siempre en cuenta que, en el *orden natural de los elementos del array*, los índices varían más rápidamente de izquierda a derecha (Fig. 3.16)

Figura 3.16 Ordenación de los elementos de un *array* `b(5,4)`



La función `reshape` tiene dos argumentos. El primero de ellos es un vector con las constantes literales asignadas al *array* en el *orden natural de los elementos del array*. El segundo argumento es un vector que especifica la forma del *array*. Por ejemplo, las sentencias

```
A = reshape((/1.1, 1.2, 3.8, 5.5, 4.3, 7.0/), (/2,3/))
Z = reshape((/1.1, 1.2, 3.8, 5.5, 4.3, 7.0/), (/3,2/))
T = reshape((/1.1, 1.2, 3.8, 5.5, 4.3, 7.0/), (/3,2/))
Mch = reshape((/'ab','cd','ef','uv','wx','yz'/), (/3,2/))
```

da lugar a las siguientes asignaciones

- En lenguaje matemático:

$$A = \begin{pmatrix} 1.1 & 3.8 & 4.3 \\ 1.2 & 5.5 & 7.0 \end{pmatrix} \quad Z = T = \begin{pmatrix} 1.1 & 5.5 \\ 1.2 & 4.3 \\ 3.8 & 7.0 \end{pmatrix} \quad Mch = \begin{pmatrix} ab & uv \\ cd & wx \\ ef & yz \end{pmatrix}$$

- En lenguaje de programación:

```
A(1,1)=1.1 A(1,2)=3.8 A(1,3)=4.3
A(2,1)=1.2 A(2,2)=5.5 A(2,3)=7.0

Z(1,1)=1.1 Z(1,2)=5.5 Z(2,1)=1.2
Z(2,2)=4.3 Z(3,1)=3.8 Z(3,2)=7.0

T(0,-1)=1.1 T(0,0)=5.5 T(1,-1)=1.2
T(1,0)=4.3 T(2,-1)=3.8 T(2,0)=7.0

Mch(1,1)='ab' Mch(1,2)='uv' Mch(2,1)='cd'
Mch(2,2)='wx' Mch(3,1)='ef' Mch(3,2)='yz'
```

3.5. Parámetros dentro de un programa

Un *parámetro*, al contrario que una *variable*, es un dato, simple o compuesto, cuyo valor no puede variar a lo largo de la ejecución del programa. Por este motivo no pueden aparecer en el lado izquierdo de una sentencia de asignación, pero sí pueden formar parte de cualquier expresión del mismo modo que las constantes literales, excepto dentro de una constante de tipo `complex`. Este tipo de objetos está pensado para definir el conjunto de constantes físicas del problema como por ejemplo $g = 9.80665\,\mathrm{ms}^{-2}$, $c = 2.997924562 \times 10^8\,\mathrm{ms}^{-1}$, $R = 8.31451\,\mathrm{Jmol}^{-1}\mathrm{K}^{-1}$, etc.

Para especificar que un dato es un parámetro del programa es necesario:

1. Declararlo con el atributo `parameter`
2. Definirlo en la propia línea de declaración

Como un parámetro es un dato constante, Fortran permite que las dimensiones de un *array* sean declaradas con parámetros de tipo `integer`. En la figura 3.17 se muestran varios ejemplos de uso y declaración de parámetros.

3.6. La sentencia `implicit none`

Es importante recordar que en un programa todos los datos tienen que estar declarados y definidos antes de ser utilizados. La declaración es obligatoria ya que, de lo contrario, el procesador considera implícitamente que todo dato cuyo identificador comienza por alguna de las letras `i, j, k, l, m, n` es de tipo `integer` y el resto de tipo `real`. En las primeras etapas del aprendizaje es habitual cometer el error de no declarar

Figura 3.17 Asignación de un parámetro

```
program main
    real, parameter       :: g = 9.80665
    integer, parameter    :: l = 5
    integer, parameter    :: m = 2
    integer, parameter    :: n = 3
    integer, parameter    :: s = 4
    integer, parameter    :: n1 = 0
    integer, parameter    :: m1 = -1
    integer,dimension(l)  :: U                  ! Vector
    real, dimension(n,m)  :: Z                  ! Matriz de 3x2
    real                  :: A(m,n)             ! Matriz de 2x3
    real                  :: T(0:m,m1:n1)       ! Matriz de 3x2
    character(len=2)      :: Mch(n,m)           ! Matriz de 3x2
    integer               :: Tabla(m,n,s)       ! Tabla de 2x3x4
end program main
```

algún dato y además, olvidar esta regla, con lo que es posible que datos que queremos que sean `integer` el programa los trate como `real` y viceversa.

Para evitar este riesgo, es aconsejable utilizar la sentencia `implicit none`. Esta sentencia, situada antes del cuerpo de las declaraciones, invalida todas las declaraciones implícitas y obliga a declarar todos los objetos que vayan a ser utilizados a lo largo del programa.

Figura 3.18 Sentencia `implicit none`

```
program main
    implicit none
    integer :: m
    integer :: n
    real    :: z
     .
     .
end program main
```

3.7. Definición en la línea de declaración

Una de las pocas operaciones que se pueden realizar en la propia línea de declaración de un dato es la asignación de un valor para ese dato, *inicialización*. En sucesivas secciones se comentarán las limitaciones que tiene el uso de este operador. Por ahora, con el conocimiento que tenemos del manejo del lenguaje, solo podemos aprovechar la inicialización con constantes literales.

```
integer, parameter     :: n = 5
integer, parameter     :: m = 2
integer, dimension(3) :: U = (/1, 5, 8/)
real, dimension(n,m)   :: A = 33.0
character(len=4)       :: Mch = 'hola'
```

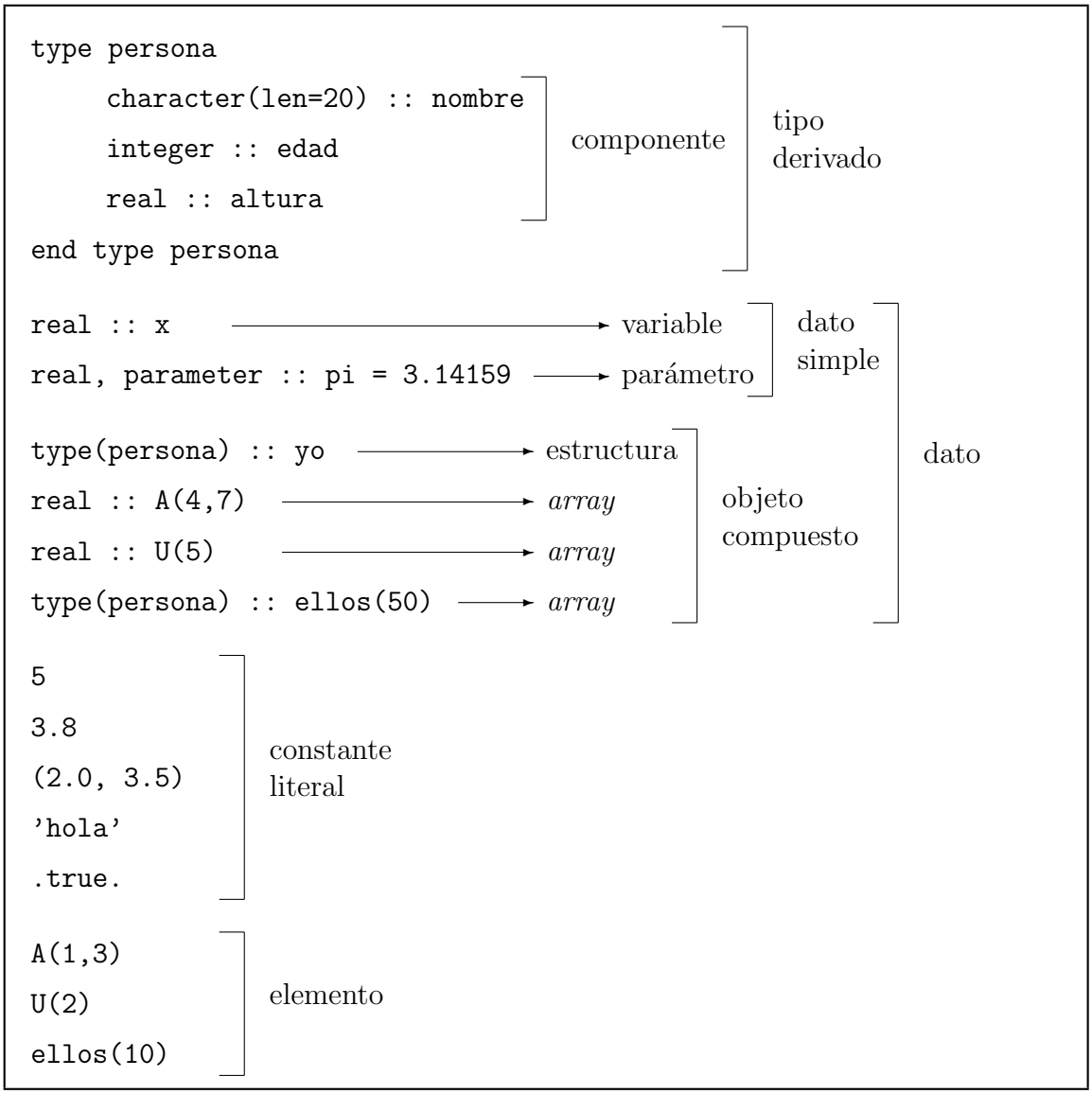
3.8. Resumen

Como resumen de lo visto a lo largo de este capítulo hay que resaltar los siguientes puntos:

- Es muy importante utilizar en todo momento el tipo de dato adecuado al problema que se pretende resolver.
- Además de los 5 tipos intrínsecos, Fortran posee dos herramientas muy poderosas para ampliar los tipos de datos:
    - Los datos derivados, útiles para crear objetos abstractos que agrupan datos de distintos tipos.
    - Los *arrays*, asociados al concepto matemático de vector o matriz.

  En última instancia, cada componente de una estructura y cada elemento de un *array* es un dato simple de tipo intrínseco, por lo que no es necesario estudiarlo ni tratarlo como un ente distinto de una simple variable.
- Declarar un dato significa asignarle un tipo. Cada tipo tiene su propio rango de valores válidos. Si el tipo del dato no coincide con el tipo de la constante literal que se pretende asignar, siempre prevalece el tipo con el que se ha declarado el dato.

Figura 3.19 Resumen de entidades de programación



4 Operaciones básicas

Índice

4.1. Introducción

En el lenguaje natural existen letras que forman palabras que, combinadas siguiendo las reglas gramaticales establecidas, generan las frases lógicas con las que nos comunicamos. Del mismo modo en Fontran existen los datos que, como las letras forman *expresiones* que se combinan siguiendo las reglas propias de este lenguaje de programación, para formar *sentencias*.

A diferencia de lo que ocurre en el lenguaje natural donde la comunicación es posible aun cuando la frase sea gramaticalmente incorrecta, en programación las reglas "gramaticales" se deben cumplir de forma estricta, pues la comunicación se establece con una máquina, es decir un objeto incapaz de interpretar una ambigüedad.

Una *expresión* en Fortran está formada por *operandos* y *operadores*, combinados siguiendo las reglas sintácticas propias del lenguaje. Una *expresión* simple se puede construir utilizando un *operador binario* en la forma

```
operando operador operando
```

por ejemplo

```
x*5
```

o bien, un *operador unario* en la forma

```
operador operando
```

por ejemplo

```
-x
```

Los operandos pueden ser constantes literales, datos o incluso otras expresiones, dando lugar a expresiones más complejas del tipo

```
operando operador operando operador operando
```

Evidentemente, cada operando debe estar definido antes de ser utilizado en una expresión y la propia expresión debe ser conceptualmente correcta. Por ejemplo, matemáticamente es incorrecto establecer una relación de orden entre dos números complejos.

La regla seguida en Fortran para la evaluación de las diferentes partes de una expresión compleja es que son evaluadas sucesivamente de izquierda a derecha, para operadores de la misma precedencia, salvo en el caso de la potenciación. Si, al margen de esta regla, interesa evaluar primero ciertas subexpresiones, éstas deben ir encerradas entre paréntesis.

```
operando operador (operando operador operando)
```

Solo es posible que en una expresión aparezcan dos operadores seguidos si el segundo es unario. Es decir en una expresión del tipo

```
operando operador operador operando
```

el segundo operador nunca puede ser un operador binario. En cualquier caso, sea obligatorio o no, por cuestiones de claridad es recomendable utilizar paréntesis para separar las distintas operaciones de una misma sentencia.

En las siguientes tres secciones de este capítulo expondremos las propiedades de las expresiones intrínsecas de Fortran que involucran como operandos datos simples (expresiones aritméticas, relacionales y lógicas). Las dos últimas secciones se ocupan de dos casos particulares: los datos de tipo `character` y los *arrays*.

4.2. Expresiones aritméticas

Una *expresión aritmética* es una expresión en la que los operandos son de tipo numérico, `integer`, `real` o `complex`, y los operadores son los que dispone de modo intrínseco Fortran para la realización de operaciones aritméticas. Estos operadores vienen dados, por orden de precedencia en la tabla 4.1

Tabla 4.1 Operadores aritméticos por orden de precedencia

Operador	Operación
`**`	Potenciación
`*`	Producto
`/`	Cociente
`+`	Suma
`-`	Resta

Siguiendo las reglas sobre precedencia expuestas en la sección anterior, en ausencia de subexpresiones entre paréntesis, primero se realizan las potencias, posteriormente los productos y cocientes y, por último, las sumas y restas. El orden en que se realizan las operaciones de igual precedencia en una expresión es de izquierda a derecha salvo la potenciación. Así

```
a*b/c
```

es equivalente a

```
(a*b)/c
```

mientras que

```
a**b**c
```

se evalúa como

```
a**(b**c)
```

Los operadores `+` y `-` se pueden utilizar como operadores unarios

```
-a + b + c
```

pero, como ocurre en lenguaje matemático, no deben situarse inmediatamente después de otro operador. Por ejemplo, para expresar x^{-b} se debe encerrar el exponente entre paréntesis

```
x**(-b)
```

Cuando se opera con datos numéricos de distinto tipo y *kind*, es necesario tener en cuenta que el resultado es del tipo con mayor rango de los involucrados en la operación. De hecho, antes de evaluar la expresión se convierten los operandos al tipo más fuerte salvo cuando se eleva un dato de tipo `real` o `complex` a una potencia de tipo `integer`, en el que no se convierte el tipo del exponente (Tablas 4.2 y 4.3)

Se recomienda que en la potenciación el exponente sea, siempre que sea posible, un dato `integer` para asegurar mayor precisión del resultado. Si el exponente es `integer`, el ordenador interpreta la potenciación como: "multiplica la base tantas veces como indique el exponente", mientras que si es `real` lo que hace es la exponencial del producto del exponente por el logaritmo de la base (resultado menos preciso).

Tabla 4.2 Tipo de resultado de *a operador b* donde operador es `*, /, +, -`

Tipo de *a*	**Tipo de** *b*	**Valor usado de** *a*	**Valor usado de** *b*	**Tipo de resultado**
integer	integer	a	b	integer
integer	real	real(a,kind(b))	b	real
integer	complex	cmplx(a,0,kind(b))	b	complex
real	integer	a	real(b,kind(a))	real
real	real	a	b	real
real	complex	cmplx(a,0,kind(b))	b	complex
complex	integer	a	cmplx(b,0,kind(a))	complex
complex	real	a	cmplx(b,0,kind(a))	complex
complex	complex	a	b	complex

Tabla 4.3 Tipo de resultado de `a**b`

Tipo de *a*	Tipo de *b*	Valor usado de *a*	Valor usado de *b*	Tipo de resultado
`integer`	`integer`	`a`	`b`	`integer`
`integer`	`real`	`real(a,kind(b))`	`b`	`real`
`integer`	`complex`	`cmplx(a,0,kind(b))`	`b`	`complex`
`real`	`integer`	`a`	`b`	`real`
`real`	`real`	`a`	`b`	`real`
`real`	`complex`	`cmplx(a,0,kind(b))`	`b`	`complex`
`complex`	`integer`	`a`	`b`	`complex`
`complex`	`real`	`a`	`cmplx(b,0,kind(a))`	`complex`
`complex`	`complex`	`a`	`b`	`complex`

En la tabla 4.4 se muestran algunos ejemplos prácticos, muy útiles para comprender mejor las reglas de conversión.

Tabla 4.4 Ejemplos prácticos de operaciones

Operación	Resultado	Operación	Resultado
`10.0/4.0`	`2.50`	`10.0d0*4.0`	`40.0d0`
`10/4`	`2`	`10.0*4.0d0`	`40.0d0`
`2**(-3)`	`0`	`2.0**(-3)`	`0.125`
`10.0/4`	`2.5`	`2.0d0**(-3)`	`0.125d0`
`10/4.0`	`2.5`	`(-2.0)**3.0`	`-8.0`
`(10.0, 1.0)/4`	`(2.5, 0.25)`	`(-2.0)**3.5`	`NaN`
`(10.0, 1.0)/4.0`	`(2.5, 0.25)`	`(-2.0)**(2/3)`	`1.0`
`(10.0, 1.0)/4.0`	`(2.5, 0.25)`	`(-2.0)**(2.0/3)`	`NaN`

La única excepción a la regla de conversión de tipo y *kind* de los operandos explicada, la constituyen las constantes literales que puedan aparecer en las expresiones aritméticas. Estas mantienen su propia precisión en una operación de modo que, por ejemplo, dada una variable, `a`, declarada en doble precisión, `a/2.5` será menos preciso que `a/2.5d0`.

4.3. Expresiones relacionales

Una *expresión relacional* se utiliza para comparar datos numéricos entre sí o con expresiones aritméticas o incluso datos de tipo `character` con expresiones del mismo

tipo. Los operadores relacionales que Fortran dispone de modo intrínseco vienen dados en la tabla 4.5.

Tabla 4.5 Operadores relacionales

Operador	Significado
`<`	Menor que
`<=`	Menor o igual que
`==`	Igual que
`/=`	Distinto que
`>`	Mayor que
`>=`	Mayor o igual que

El resultado de evaluar una expresión relacional es de tipo `logical` y, por tanto, solo puede tomar uno de los valores `.true.` o `.false.`. Generalmente estas expresiones se utilizan en las estructuras de control que gobiernan el flujo de un programa.

Por coherencia matemática, si al menos uno de los operandos es de tipo `complex` los únicos operadores relacionales disponibles son `==` y `/=`. Veamos algunos ejemplos (Fig. 4.1) de expresiones relacionales en los que intervienen las variables `i` y `j` de tipo `integer`, las variables `a` y `b` de tipo `real` y la variable `ch` de tipo `character`.

Figura 4.1 Ejemplos de expresiones relacionales

```
1        i <= 0
2        a > b
3        (a+b) > (i-j)
4        a+b > i-j
5        ch == 'Z'
```

Las expresiones de las líneas 3 y 4 son de tipo mixto en el sentido de que en ellas aparecen datos de distinto tipo. Además, intervienen relaciones aritméticas. Como regla general, siempre que en una expresión relacional intervengan datos numéricos de distinto tipo o *kind*, antes de evaluar la expresión relacional, los datos numéricos se convierten al tipo y *kind* de mayor rango de los involucrados. Así, en las sentencias de las líneas 3 y 4, después de realizar la operación `(i-j)` el resultado se convierte a tipo `real` y se compara con el resultado de la operación `(a+b)`. Respecto al orden en el que se evalúan las expresiones, primero se realizan las operaciones aritméticas y luego las relacionales, por

lo que las sentencias de las líneas 3 y 4 son equivalentes. Para evitar confusiones conviene utilizar siempre los paréntesis.

4.4. Expresiones lógicas

Las *expresiones lógicas* se utilizan, junto con las relacionales, en las estructuras de control que gobiernan el flujo de un programa. El resultado de evaluar una expresión lógica es de tipo `logical` y, por tanto, solo puede tomar uno de los valores `.true.` o `.false.`. Los operadores lógicos que Fortran dispone de modo intrínseco vienen dados en la tabla 4.6 en orden decreciente de precedencia.

Tabla 4.6 Operadores lógicos en orden decreciente de precedencia

Operador	Significado
Operador unario:	
`.not.`	negación lógica
Operadores binarios:	
`.and.`	intersección lógica
`.or.`	unión lógica
`.eqv.` y `.neqv.`	equivalencia y no equivalencia lógica

Los resultados de las operaciones lógicas habituales se muestran en la figura 4.2 donde se utilizan las variables `a` y `b` de tipo `logical`.

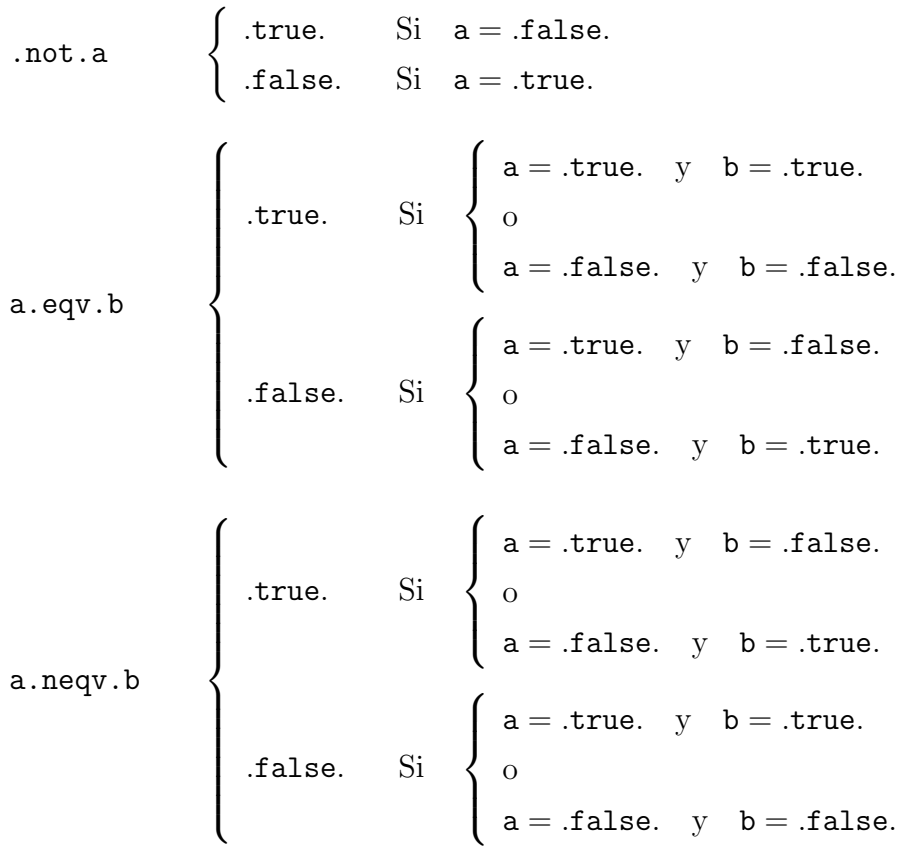
La figura 4.3 recoge algunos ejemplos de expresiones que involucran los 3 tipos de operadores vistos hasta ahora.

Cuando en una expresión lógica interviene alguno de los operadores `.and.` u `.or.` no existe una regla definida sobre el orden en el que se evalúan los operandos. Incluso, dependiendo del procesador, puede ocurrir que alguno de los operandos no llegue a ser evaluado. Por ejemplo, dadas dos variables de tipo `integer` con valores `i=6` y `j=4`, en la expresión

```
(j < 5) .or. (i < 5)
```

algunos procesadores no evaluarían la expresión relacional `(i < 5)` ya que, al ser verdadero el operando de la izquierda, la expresión lógica completa es verdadera, independientemente del valor del operando de la derecha. Sin embargo, habrá procesadores que evalúen primero la subexpresión de la derecha, por lo que, al ser falsa, necesariamente

Figura 4.2 Resultado de algunas expresiones lógicas. `a` y `b` son de tipo `logical`



tendrán que evaluar también la de la izquierda. Por último, existen procesadores que, al margen del valor de los operandos, evalúan todas las subexpresiones involucradas.

4.5. Operaciones con caracteres

Los datos de tipo `character` merecen una consideración aparte en cuanto al tipo de operaciones permitidas y el funcionamiento de los operadores relacionales. La única operación permitida es la *concatenación* de caracteres mediante el *operador de concatenación* definido por el símbolo `//`, que combina dos operandos de tipo `character` en un único resultado de tipo `character`. Por ejemplo, el resultado de concatenar las constantes literales `Agente` y `007` en la forma

```
'Agente'//'007'
```

es la constante `Agente007`. En la concatenación, la longitud del resultado `long_res` es la suma de las longitudes de los operandos. Si dicho resultado se va a asignar a un dato de tipo `character` declarado con una longitud, `long_dato`, mayor que `long_res` se rellena

Figura 4.3 Ejemplo con operadores aritméticos, relacionales y lógicos

```
program main
    type persona
       character(len=20) :: nombre
       integer :: edad
    end type
    integer          :: i
    integer          :: j
    integer          :: k
    integer          :: U(4)
    logical          :: F1
    logical          :: F2
    type(persona)    :: yo
    i = 6
    j = 4
    k = 10
    U = (/1, -2, 8, 0/)
    yo%edad = 20
    write(*,*) i < 0                          ! falso
    write(*,*) i <= k                         ! verdadero
    write(*,*) (i+j) <= k                     ! verdadero
    write(*,*) (i < 20) .and. (j < 20)        ! verdadero
    write(*,*) (i < 5) .and. (j < 5)          ! falso
    write(*,*) (i < 5) .or. (j < 5)           ! verdadero
    write(*,*) U(j) == 1                      ! falso
    write(*,*) U(j)+U(j-1) > 0                ! verdadero
    write(*,*) i+U(1) == 7                    ! verdadero
    write(*,*) yo%edad >= 20                  ! verdadero
    F1 = ((i+j) == k)                         ! verdadero
    F2 = ((i-j) > 0)                          ! falso
    write(*,*) .not. F1                       ! falso
    write(*,*) F1 .eqv. F2                    ! falso
    write(*,*) F1 .neqv. F2                   ! verdadero
end program main
```

el valor del resultado con tantos caracteres blanco a la derecha como sea necesario para

completar la longitud `long_dato`. Si por el contrario es menor, se trunca el valor del resultado a los primeros `long_dato` caracteres (Fig. 4.4).

Figura 4.4 Ejemplo con operador de concatenación

```
program main
    character(len=9)  :: Ag1
    character(len=12) :: Ag2
    character(len=7)  :: Ag3
    character(len=6)  :: nombre
    character(len=3)  :: numero
    nombre = 'Agente'
    numero = '007'
    Ag1 = nombre//numero              ! Ag1 = Agente007
    Ag2 = nombre//numero              ! Ag2 = Agente007␣␣␣
    Ag3 = nombre//numero              ! Ag3 = Agente0
end program main
```

Mención especial merecen las expresiones relacionales en las que los operandos son de tipo `character`. En todos los procesadores los caracteres poseen una propiedad denominada *secuencia de comparación*, que ordena la colección de caracteres asignándoles un *número de orden*. Aunque cada procesador puede tener su propia secuencia, la mayoría utiliza la secuencia definida por el estándar ASCII (del inglés, *American Standard Code for Information Interchange*). Sin embargo, Fortran está diseñado para adaptarse a otras secuencias como, por ejemplo la secuencia EBCDIC (del inglés, *Extended Binary Coded Decimal Interchange Code*) desarrollada por IBM. Esto implica que, en general, por cuestiones de portabilidad, no es conveniente que un programa fundamente parte de su algoritmo en la relación entre datos de tipo `character`. En el apéndice A se muestra una tabla completa del conjunto de caracteres ASCII así como su número de orden.

En cualquier caso, la comparación entre dos operandos de tipo `character` se realiza carácter a carácter de izquierda a derecha siguiendo el orden alfabético para las letras y el numérico para los dígitos. Si los operandos no tienen la misma longitud, el de menor longitud se rellena con espacios en blanco a la derecha hasta completar la longitud máxima antes de realizar la comparación (Fig 4.5).

Figura 4.5 Relaciones entre datos de tipo carácter

```
program main
    character(len=3) :: ch
    ch = 'abc'
    write(*,*) ch == 'Abc'                  ! falso
    write(*,*) ch == ' abc'                 ! falso
    write(*,*) ch == 'abc  '                ! verdadero
    write(*,*) ch > 'abg'                   ! falso
    write(*,*) ch > 'aba'                   ! verdadero
    write(*,*) ch < 'ab '                   ! falso
    write(*,*) ch < ' ab'                   ! falso
    write(*,*) ch > 'abc  '                 ! falso
    write(*,*) ch >= 'abc  '                ! verdadero
end program main
```

4.6. Operaciones con *arrays*

Como ya hemos visto en la sección 3.4, un *array* no es más que una colección de datos del mismo tipo. Las operaciones intrínsecas se realizan elemento a elemento, eso implica que, tanto los operandos como el resultado, deben ser conformes. Recordemos que una constante literal es conforme a cualquier *array*, y puede ser uno de los operandos. En la figura 4.6 se muestran varias operaciones con *arrays*.

4.7. Definición en la línea de declaración

Ahora que hemos visto las operaciones típicas de cada tipo de dato podemos ampliar las utilidades de la definición en la línea de declaración dadas en la sección 3.7.

Todas las operaciones estudiadas en este capítulo se pueden realizar en una inicialización, siempre que los operandos estén inicializados con anterioridad.

```
    integer, parameter     :: n = 3
    integer, parameter     :: m = n-1
    integer, dimension(3)  :: U = n*m
    real, dimension(n,m)   :: A = reshape((/1, 0, 2, 8, 3, 1/), (/n,m/))
    character(len=8)       :: Mch = 'hola'//'pepe'
    logical                :: flag = (m < n)
```

Figura 4.6 Operaciones con *arrays*

```
program main
    integer :: U(3)
    integer :: V(3)
    logical :: Luv(3)
    integer :: a(2,3)
    integer :: b(2,3)
    integer :: c(2,3)
    U = (/1, 2, 3/)
    V = 2
    Luv = (U >= V)                          ! Luv = (/F, T, T/)
    U = U + V                               ! U = (/3, 4, 5/)
    U = U*V                                 ! U = (/6, 8, 10/)
    Luv = .not.Luv                          ! Luv = (/T, F, F/)
    a = reshape((/1, 0, 2, 8, 3, 1/), (/2,3/))
    b = reshape((/0, 4, 1, 2, 5 ,9/), (/2,3/))
    c = a*b                                 ! c = (/0,0,2,16,15,9/)
end program main
```

5 Control de flujo

Índice

5.1. Introducción

Para aumentar la eficiencia de la programación es necesario que los programas tengan una estructura fácil de interpretar, manejar y modificar. Los criterios de programación que cumplen estos objetivos son la *programación estructurada* y la *programación modular*. No son criterios contrapuestos, sino complementarios. El modular tiende a dividir el programa en programas más pequeños, llamados *subprogramas*, mientras que el estructurado se encarga de desarrollar estructuradamente cada uno de los subprogramas.

En este capítulo nos centraremos en la programación estructurada, dejando la modular para capítulos posteriores.

En un programa, las sentencias se ejecutan de forma secuencial o lineal. A menudo, conviene no utilizar esta secuencia sino modificarla a medida que se van haciendo los cálculos. La base de la programación estructurada es que todo programa puede ser escrito utilizando solo tres tipos de estructuras de control de ejecución,

- *secuencial*
- *condicional*
- *iterativa*

que se pueden anidar (pero no solapar) adecuadamente cuando sea necesario.

La estructura secuencial es aquella en la que las acciones se realizan de forma secuencial (una a continuación de la otra), sin posibilidad de omitir ninguna. Es el caso de los programas vistos en los capítulos anteriores. Este paradigma de programación resulta muy sencillo pero de escasa utilidad en la práctica. Es habitual que se requiera que se ejecute determinado grupo de sentencias si se cumple determinada condición (por ejemplo si cierto número es positivo), y otro grupo distinto de sentencias en caso contrario. También suele ser necesario realizar el mismo grupo de operaciones varias veces en determinado punto del programa. Por ejemplo, si se quiere calcular la suma de los mil primeros números naturales resulta ineficiente y tedioso escribir un programa de forma secuencial con mil sentencias en las que se realiza la misma operación. En este caso se requiere la posibilidad de realizar una única adición de forma iterativa. Fortran dispone de dos herramientas para controlar el flujo lógico del programa: la *estructura condicional*, que permite la bifurcación en determinado punto de un programa, y la *estructura iterativa*, indicada para los procesos iterativos.

Como norma general, existe una palabra clave que inicia la estructura (`if, select, do`) y otra que la cierra (`end if, end select, end do`). Entre ambas, se encuentra el grupo de sentencias sometidas a las condiciones impuestas. En todos los casos tenemos

la opción de poner un nombre a la estructura. Nombrar una estructura no es obligatorio pero, si se hace, sí es obligatorio nombrarla tanto al inicio como al final de la misma.

Por cuestiones de claridad, es conveniente sangrar las sentencias contenidas en la estructura, sobre todo en el caso de estructuras anidadas.

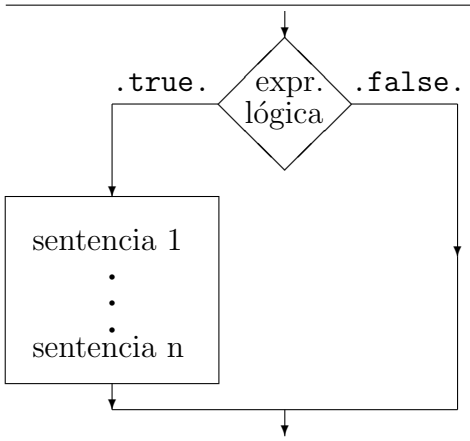
5.2. Estructura condicional

Una estructura condicional es aquella que ejecuta un conjunto u otro de sentencias, dependiendo de si se cumple o no una determinada condición. Existen tres tipos de estructuras condicionales: *simples*, *dobles* y *múltiples*.

5.2.1. Estructura condicional simple

Evalúa una única condición dada por una expresión lógica escalar. Si es cierta se ejecutan las sentencias contenidas en la estructura, y si es falsa se pasa el control a la primera sentencia ejecutable situada fuera de la estructura. El algoritmo correspondiente a esta estructura se muestra en la figura 5.1

Figura 5.1 Estructura condicional simple



[nombre:] if (*expr. lógica*) then
sentencia 1
sentencia n
end if [nombre]

Por ejemplo, queremos intercambiar los valores de `x` e `y` solo en el caso en que `x` sea menor que `y`. El algoritmo sería:

```
if (x < y) then
    temp = x
    x = y
    y = temp
end if
```

Como simplificación, en el caso de que solo haya una sentencia de ejecución dentro de la estructura, el lenguaje permite la siguiente variación

```
if (expr. lógica) sentencia
```

En la figura 5.2 se muestra un ejemplo de utilización de una estructura condicional simple.

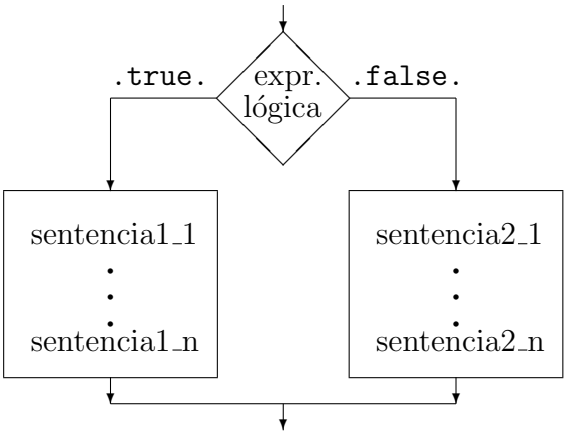
Figura 5.2 Ejemplo con estructura condicional simple. Cálculo del valor absoluto

```
program ValorAbsoluto
    real :: x
    write(*,*) 'Introducir el valor de x'
    read(*,*) x
    if (x < 0.0) x = -x
    write(*,*) 'El valor absoluto de x es ', x
end program ValorAbsoluto
```

5.2.2. Estructura condicional doble

Evalúa una única condición dada por una expresión lógica escalar. Si es cierta se ejecuta un grupo de sentencias de la estructura y, si es falsa, se ejecuta otro grupo de sentencias de la misma. El algoritmo correspondiente a esta estructura se muestra en la figura 5.3 junto con un ejemplo en la figura 5.4.

Figura 5.3 Estructura condicional doble



[nombre:] if (*expr. lógica*) then
sentencia1_1
sentencia1_n
else
sentencia2_1
sentencia2_n
end if [nombre]

Figura 5.4 Ejemplo con estructura condicional doble. Resolver $ax + b = 0$

```
program raiz
    real :: a
    real :: b
    real :: x
    write(*,*) 'Introducir los valores de a y b'
    read(*,*) a, b
    if (a /= 0.0) then
        x = -b/a
        write(*,*) 'La solucion de la ecuacion es x = ', x
    else
        write(*,*) 'La ecuacion planteada es incorrecta'
    end if
end program raiz
```

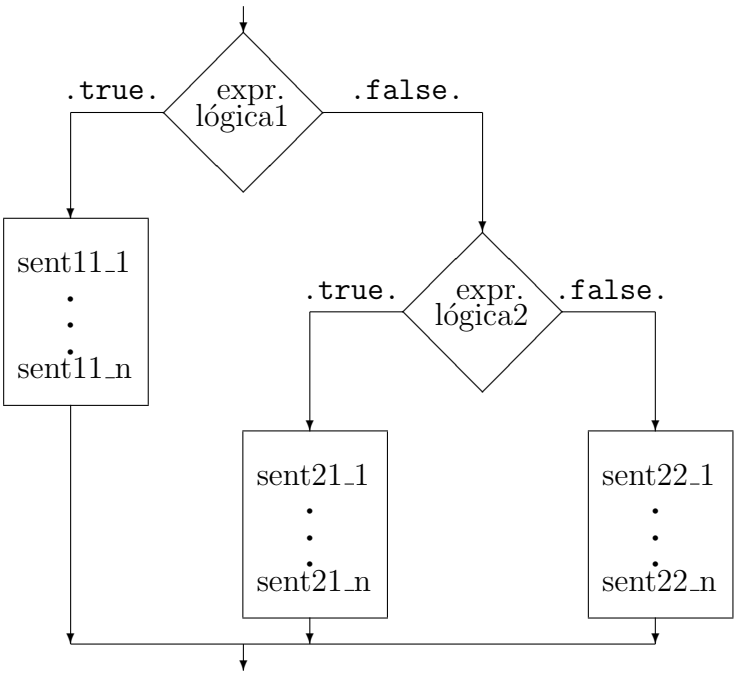
5.2.3. Estructura condicional múltiple

Dentro de las estructuras condicionales múltiples existen dos tipos de construcciones. La construcción `if` que es una mera ampliación de las anteriores y la construcción `select case` basada en el control del valor de una variable o expresión.

5.2.3.1. Construcción `if`

En este caso, la estructura consta de varias condiciones, dadas en forma de expresiones lógicas escalares, que son evaluadas de forma secuencial. El control pasa al grupo de sentencias correspondientes a la primera condición evaluada como verdadera y, tras su ejecución, el control pasa a la primera sentencia ejecutable situada fuera de la estructura. El algoritmo más sencillo para este tipo de construcción se muestra en la figura 5.5 y su forma más general en la figura 5.6. Como en casos anteriores, dar un nombre a la construcción es opcional pero, si se hace, es obligatorio finalizarla con el mismo nombre. Además, el último `else` también es opcional. En la figura 5.7 utilizamos esta estructura condicional múltiple para resolver un ecuación de segundo orden.

Figura 5.5 Estructura condicional múltiple




```
if (expr. lógica1) then
    sentencia11_1
        .
        .
        .
    sentencia11_n

else if (expr. lógica2) then
    sentencia21_1
        .
        .
        .
    sentencia21_n

else
    sentencia22_1
        .
        .
        .
    sentencia22_n

end if
```

Figura 5.6 Estructura condicional múltiple

```
[ nombre: ]  if (expr. lógica1) then
                 sentencias_1
             else if (expr. lógica2) then
                 sentencias_2
             else if (expr. lógica3) then
                 sentencias_3
               ⋮
             [ else
                 sentencias_n]
             end if [ nombre ]
```

5.2.3.2. Construcción `select case`

Un caso particular del esquema condicional múltiple descrito anteriormente es la estructura `select case`, en la cual se evalúa una variable o expresión de tipo `integer`,

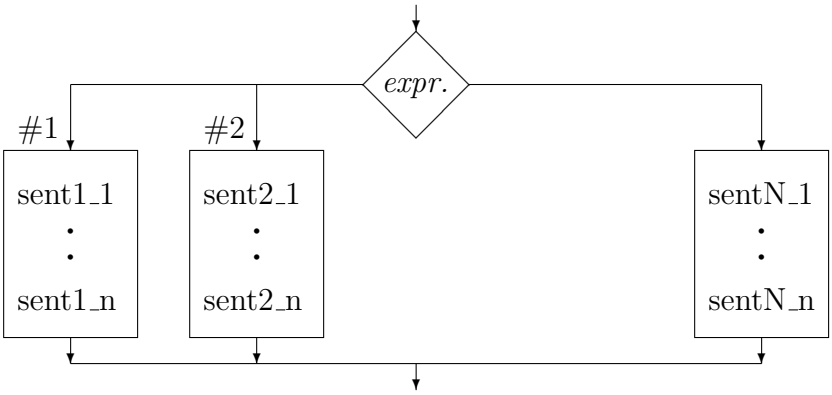
Figura 5.7 Ejemplo con estructura condicional múltiple. Resolver $\mathtt{a}^2\mathtt{x} + \mathtt{b}\mathtt{x} + \mathtt{c} = 0$

```
program raices
    real    :: a
    real    :: b
    real    :: c
    real    :: disc
    real    :: x1
    real    :: x2
    complex :: z1
    complex :: z2

    write(*,*) 'Introducir los valores de a, b y c'
    read(*,*) a, b, c

    disc = b*b - 4.0*a*c
    if (disc < 0.0) then
        z1 = cmplx(-b, sqrt(-disc))
        z1 = z1/(2.0*a)
        z2 = cmplx(-b, -sqrt(-disc))
        z2 = z2/(2.0*a)
        write(*,*) 'Las soluciones de la ecuacion son ', z1, z2
    else if (disc > 0.0) then
        x1 = (-b + sqrt(disc))/(2.0*a)
        x2 = (-b - sqrt(disc))/(2.0*a)
        write(*,*) 'Las soluciones de la ecuacion son ', x1, x2
    else
        x1 = -b/(2.0*a)
        x2 = x1
        write(*,*) 'Las soluciones de la ecuacion son ', x1, x2
    end if

end program raices
```

`character` o `logical` y, según sea su valor, se ejecuta determinado grupo de sentencias dentro de la estructura. Una vez terminada su ejecución, el control pasa a la primera sentencia ejecutable situada fuera de la estructura.

El algoritmo para este tipo de estructuras se muestra en la figura 5.8

Figura 5.8 Construcción `select case`



[`nombre:`] `select case` (*expr. case*)
`case` (*rango de valores #1*)
sentencia1_1
.
.
.
sentencia1_n
`case` (*rango de valores #2*)
sentencia2_1
.
.
.
sentencia2_n
.
.
.
[`case default`
sentenciaN_1
.
.
.
sentenciaN_n]
`end select` [`nombre`]

donde

- *expr. case* es una expresión escalar de tipo `integer`, `character` o `logical`. Por ejemplo, el nombre de un dato simple de uno de estos tipos, `i`, o una expresión aritmética, `i*j+2`

- *rango de valores* puede tomar un único valor, varios valores o un rango de valores, según se escriba de la forma

dato	*expr. case* = dato
dato_1, dato_2, ..., dato_n	*expr. case* = dato_1 ó dato_2 ó ... ó dato_n
dato_1 : dato_2	dato_1 ≤ *expr. case* ≤ dato_2
: dato_2	*expr. case* ≤ dato_2
dato_1 :	dato_1 ≤ *expr. case*

Como en casos anteriores, dar un nombre a la construcción es opcional pero, si se hace, es obligatorio finalizarla con el mismo nombre. Además, la aparición de `case default` también es opcional.

Figura 5.9 Ejemplo con construcción `select case`

```
program ejemplo_case
    integer :: num
    write(*,*) 'Introducir un numero entero'
    read(*,*) num
    select case (num)
    case (:-1)
        write(*,*) ' numero <= -1 '
    case (1, 3, 5, 7, 9)
        write(*,*) ' 1 <= numero <= 9 e impar'
    case (2, 4, 6, 8)
        write(*,*) ' 2 <= numero <= 8 y par'
    case (0, 10:200)
        write(*,*) ' 10 <= numero <= 200 o 0'
    case (205:)
        write(*,*) ' numero >= 205'
    case default
        write(*,*) ' 201 <= numero <= 204'
    end select
end program ejemplo_case
```

5.3. Estructura iterativa

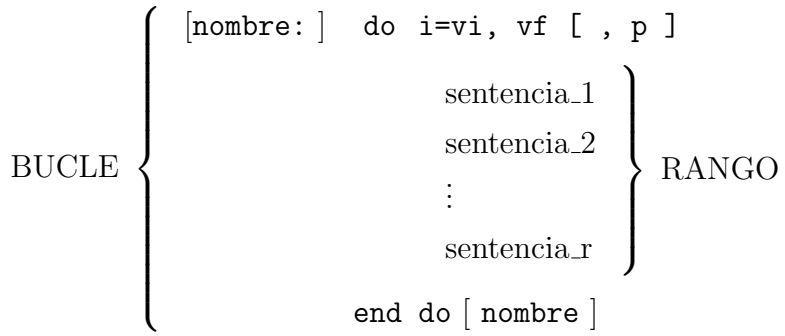
Una técnica fundamental en toda programación es el uso de bucles para implementar acciones (*rango del bucle*), que se deben ejecutar repetidamente mientras se cumpla una condición.

Existen dos tipos de estructura iterativa dependiendo de si el bucle está controlado por un contador o por una expresión lógica.

5.3.1. Bucle controlado por un contador

El número máximo de iteraciones está fijado por el programador a través de una variable de tipo `integer`, llamada *contador*. Su estructura básica es (Fig. 5.10)

Figura 5.10 Bucle controlado por un contador



donde,

- `i` es una variable, llamada *contador*, de tipo `integer`.
- `vi,vf,p` son constantes o variables de tipo `integer`, tales que `vi` es el valor inicial del contador, `vf` es el valor final del contador y `p` es el incremento.
- El incremento `p` no puede ser cero.
- Si `p>0`, debe ser `vi` $\leq$ `vf`; si `p<0`, debe ser `vi` $\geq$ `vf`
- Si `p=1`, entonces no es necesario especificarlo.
- Cuando se produce una salida normal del rango, el contador vale `i=vi+p*(1+(vf-vi)/p)`
- Dentro del rango del bucle se puede redefinir el contador, y este cambio queda reflejado en el siguiente paso, por lo que en esta asignatura no estará permitido hacerlo.

- Dentro del rango se pueden cambiar los valores de las variables `vi`, `vf` o `p`, y aunque estos cambios no afectan al proceso de variación del contador establecido en el inicio del bucle, en esta asignatura no estará permitido hacerlo.

- El número de iteraciones realizadas tras una salida normal del bucle `do` viene dado por la expresión

$$\text{máx}\left\{(vf - vi + p)/p,\ 0\right\}$$

Ejemplos típicos del uso de estructuras iterativas lo constituyen el cálculo del sumatorio y el productorio. En las figuras 5.11 y 5.12 se utiliza la construcción `do` con contador para calcular la suma de los n primeros números naturales y el factorial de un número entero no negativo, respectivamente. Nótese que estamos haciendo uso de una estructura iterativa dentro de una condicional.

Figura 5.11 Programa para calcular un sumatorio

```
program Sumatorio
    integer :: n
    integer :: suma
    integer :: i

    write(*,*) 'Introducir un numero entero no negativo'
    read(*,*) n

    ! calculo de la suma 1 + 2 + ... + n
    if (n >= 0) then
        suma = 0
        do i=1,n
            suma = suma + i
        end do
        write(*,*) 'La suma de los n primeros numeros es ', suma
    else
        write(*,*) 'El numero introducido es incorrecto'
    end if

end program Sumatorio
```

Figura 5.12 Programa para calcular un factorial

```
program Factorial
    integer :: n
    integer :: Fact
    integer :: i

    write(*,*) 'Introducir un numero entero no negativo'
    read(*,*) n

    ! calculo del factorial de n
    if (n >= 0) then
        Fact = 1
        do i=2,n
            Fact = Fact*i
        end do
        write(*,*) 'El factorial de n es ', Fact
    else
        write(*,*) 'El numero introducido es incorrecto'
    end if

end program Factorial
```

5.3.1.1. Bucle implícito

Un caso particular de los bucles controlados por un contador lo constituyen los llamados *bucles implícitos*. Podemos describirlo como una contracción de un bucle normal que se utiliza únicamente en dos situaciones:

1. Para definir un *array* cuyos elementos siguen una regla iterativa sencilla.
2. Para escribir por pantalla o en un fichero un conjunto de datos.

La estructura general de un bucle implícito es

```
(lista_de_variables, i=vi, vf [, p])
```

siendo `i` el contador del bucle y *lista_variables* el rango del mismo, es decir, la variación de los datos deseada en cada iteración.

En el ejemplo de la figura 5.13 se muestran los dos usos del bucle implícito: para definir un vector `U`, de modo que el valor de cada elemento `U(i)` está relacionado con el

índice del mismo, y para imprimir una matriz `A` tal y como lo hacemos matemáticamente, es decir, por filas.

Figura 5.13 Ejemplo de bucle implícito

```
program main
    integer, parameter :: n = 5
    integer, parameter :: m = n+1
    real    :: U(n)
    real    :: A(n,m)
    integer :: i
    integer :: j

    U = (/(i + 1, i=1,n)/)          ! U = (/2,3,4,5,6/)

    ! Imprime la matriz A por filas
    do i=1,n
        do j=1,m
            A(i,j) = i + j
        end do
        write(*,*) (A(i,j), j=1,m)
    end do
end program main
```

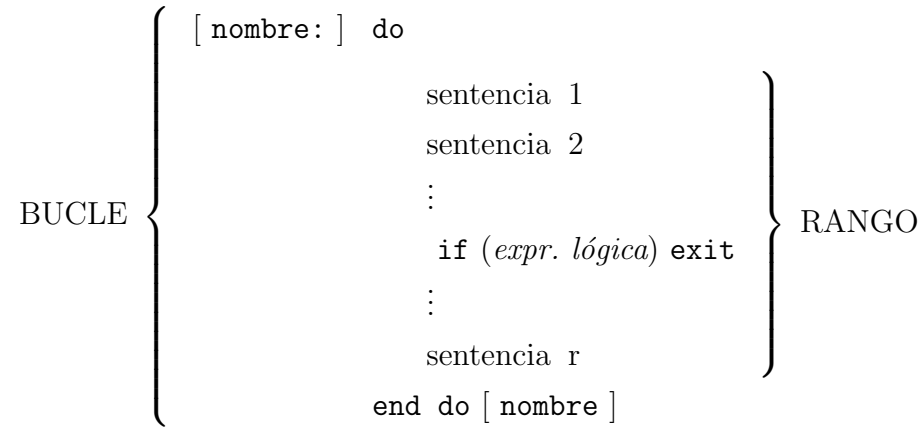
5.3.2. Bucle controlado por una expresión lógica

La estructura general de esta construcción se muestra en la figura 5.14

La forma de trabajar con esta construcción es la siguiente: las sentencias que constituyen el rango del bucle se ejecutan repetidamente hasta que la expresión lógica toma el valor `.true.`, en cuyo caso, se produce una salida del bucle, continuando la ejecución en la sentencia siguiente al `end do`. Por razones de claridad conviene situar la condición del bucle de forma que sea bien la primera sentencia de su rango, o bien la última.

La estructura `if - exit` se puede utilizar dentro de cualquier tipo de bucle, incluido el controlado por contador, con el mismo efecto descrito anteriormente.

Figura 5.14 Bucle controlado por una expresión lógica



En el ejemplo 5.15 se utiliza la construcción `do` sin contador con dos propósitos: (a) comprobación de que el dato de entrada al problema cumple con las especificaciones requeridas y (b) cálculo del número natural n tal que $\sum_{i=1}^{n} i \leq cota$

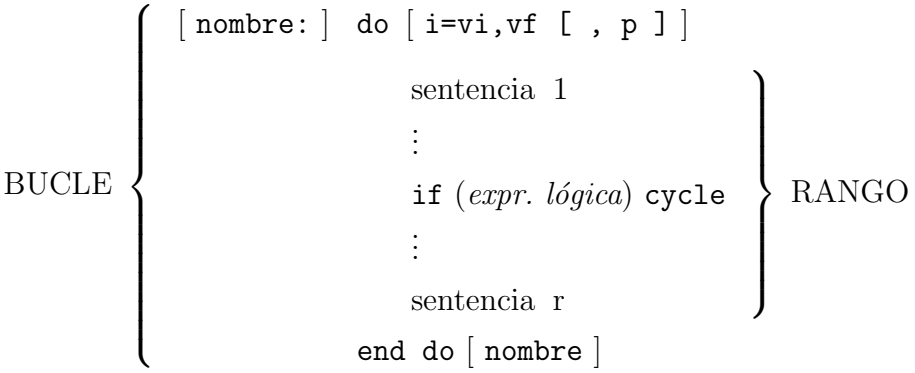
Figura 5.15 Programa para calcular un sumatorio

```
program Sumatorio
    integer :: cota, n
    integer :: i, suma

    ! lectura de la cota
    do
        write(*,*) 'Introducir un numero entero positivo'
        read(*,*) cota
        if (cota > 0) exit
    end do

    ! calculo de n tal que 1 + 2 + ... + n <= cota
    suma = 0
    i = 1
    do
        if (suma > cota) exit
        suma = suma + i
        i = i+1
    end do
    n = i-1
    write(*,*) 'n vale ', n

end program Sumatorio
```

5.3.3. Sentencia `cycle`

En ciertas ocasiones es necesario, sin salir del bucle `do`, saltar un grupo de sentencias del rango del `do` y pasar a la siguiente iteración. Para ello, existe la sentencia `cycle`. Esta sentencia se puede utilizar tanto en bucles controlados por un contador como en los controlados por una expresión lógica. Será una sentencia más del rango del `do`, y su forma general se muestra en la figura 5.16

Figura 5.16 Sentencia `cycle`



6 Funciones intrínsecas

Índice

6.1. Introducción

Dada la orientación de Fortran hacia la resolución de problemas científicos, es lógico que las funciones matemáticas más básicas formen parte del propio lenguaje y no tengan que ser programadas por el usuario. Además estas funciones han sido programadas del modo más eficiente, aprovechando al máximo los recursos del compilador, por lo que resulta muy difícil competir con ellas.

En este capítulo estudiaremos las funciones intrínsecas siguiendo una clasificación basada en el tipo de sus argumentos:

1. Datos numéricos: funciones matemáticas, funciones de conversión de tipo, de truncamiento y redondeo.
2. Datos de tipo `character`: funciones de conversión y de manipulación de cadenas.
3. Datos de tipo *array*.

Dentro del conjunto de funciones intrínsecas definidas en Fortran, merecen especial mención las diseñadas para la manipulación de *arrays*. Estas funciones constituyen una herramienta eficiente y poderosa para la realización de las operaciones básicas entre datos de este tipo.

Todas las funciones intrínsecas se pueden utilizar de forma natural en un programa sin necesidad de declararlas explícitamente. Forman parte del propio lenguaje de programación. La única precaución a tener en cuenta es que, excepto la función `reshape` necesaria para definir un *array*, ninguna otra función intrínseca se puede utilizar en la línea de declaración para inicializar un dato.

6.2. Funciones intrínsecas con datos numéricos

La regla general para la utilización de funciones intrínsecas con datos numéricos es que si interviene más de un argumento, éstos deben ser todos del mismo tipo y *kind*. A lo largo de esta sección utilizaremos la siguiente convención: un argumento con el símbolo † es un argumento opcional. Además, reservamos la letra `k` para denotar el *kind* del resultado de la función.

6.2.1. Funciones de conversión

En la tabla 6.1 se muestran las funciones elementales para convertir datos numéricos de un tipo, en datos numéricos con el mismo valor pero de distinto tipo.

Tabla 6.1 Funciones de conversión de datos numéricos

Nombre función	Valor devuelto	Tipo argumento	Tipo función
`aimag(z)`	Parte imaginaria del número complejo `z`	`complex`	`real`
`cmplx(x,y`†`,k`†`)`	Número complejo de parte real `x` e imaginaria `y`	`real`, `complex`	`complex`
`real(x,k`†`)`	`x` convertido a `real`	`integer`, `real`, `complex`	`real`
`dble(x)`	`x` convertido a `real(8)`	`integer`, `real`, `complex`	`real(8)`
`dprod(x,y)`	`x`×`y` convertido a `real(8)`	`real`	`real(8)`

La función `cmplx(x,y)` requiere una explicación más detallada. Su funcionamiento es como sigue:

- Si `x` es de tipo `complex`, entonces es obligatorio omitir el argumento `y`

$$\texttt{cmplx}(\texttt{x}) = Re(\texttt{x}) + iIm(\texttt{x})$$

- Si `x` es de tipo `real` y se omite el valor de `y`, se asume `y=0.0`

$$\texttt{cmplx}(\texttt{x}) = \texttt{x} + i0.0$$

- Si `x` es de tipo `real` y se especifica el valor de `y`, entonces

$$\texttt{cmplx}(\texttt{x}, \texttt{y}) = \texttt{x} + i\texttt{y}$$

6.2.2. Funciones de truncamiento y redondeo

Se estudian las funciones de truncamiento y redondeo en una sección aparte aunque algunas de ellas, como se observa en la tabla 6.2, realizan también la conversión del tipo del argumento.

Tabla 6.2 Funciones de truncamiento y redondeo

Nombre función	Valor devuelto	Tipo argumento	Tipo función
`aint(x)`	Valor truncado de `x`	`real`	`real`
`anint(x)`	Redondeo al entero más próximo	`real`	`real`
`int(x,k`[†]`)`	Valor truncado de `x`	`integer`, `real`, `complex`	`integer`
`nint(x,k`[†]`)`	Redondeo al entero más próximo	`real`	`integer`
`ceiling(x)`	Menor entero mayor o igual que `x`	`real`	`integer`
`floor(x)`	Mayor entero menor o igual que `x`	`real`	`integer`

En la figura 6.1 se puede observar la diferencia entre truncar y redondear un determinado dato, así como el tipo de resultado devuelto por las funciones `aint`, `anint`, `int`, y `nint`

Figura 6.1 Funciones de truncamiento y redondeo

x	aint(x)	anint(x)	int(x)	nint(x)
±5.0	±5.0	±5.0	±5	±5
±5.1	±5.0	±5.0	±5	±5
±5.2	±5.0	±5.0	±5	±5
±5.3	±5.0	±5.0	±5	±5
±5.4	±5.0	±5.0	±5	±5
±5.5	±5.0	±6.0	±5	±6
±5.6	±5.0	±6.0	±5	±6
±5.7	±5.0	±6.0	±5	±6
±5.8	±5.0	±6.0	±5	±6
±5.9	±5.0	±6.0	±5	±6

6.2.3. Funciones matemáticas

En esta sección se estudian (Tabla 6.3) las funciones típicas para la realización de las operaciones matemáticas.

Tabla 6.3 Funciones matemáticas

Nombre función	Valor devuelto	Tipo argumento	Tipo función
`sqrt(x)`	Raíz de `x`	`real`, `complex`	`real`, `complex`
`conjg(z)`	Conjugado de `z`	`complex`	`complex`
`max(x1,x2,...)`	Máximo de los `x1,x2,...`	`integer`, `real`	`integer`, `real`
`min(x1,x2,...)`	Mínimo de los `x1,x2,...`	`integer`, `real`	`integer`, `real`
`exp(x)`	Exponencial de `x`	`real`, `complex`	`real`, `complex`
`log(x)`	Logaritmo neperiano de `x`	`real`, `complex`	`real`, `complex`
`log10(x)`	Logaritmo decimal de `x`	`real`	`real`
`cos(x)`	Coseno de `x` (rad)	`real`, `complex`	`real`, `complex`
`sin(x)`	Seno de `x` (rad)	`real`, `complex`	`real`, `complex`
`tan(x)`	Tangente de `x` (rad)	`real`	`real`
`acos(x)`	$0 \leq arccos(\mathtt{x}) \leq \pi$	`real`	`real`
`asin(x)`	$-\pi/2 \leq arcsen(\mathtt{x}) \leq \pi/2$	`real`	`real`
`atan(x)`	$-\pi/2 \leq arctg(\mathtt{x}) \leq \pi/2$	`real`	`real`
`atan2(x,y)`	$-\pi < arctg(\mathtt{y}/\mathtt{x}) \leq \pi$	`real`	`real`
`cosh(x)`	Coseno hiperbólico de `x` (rad)	`real`	`real`
`sinh(x)`	Seno hiperbólico de `x` (rad)	`real`	`real`
`tanh(x)`	Tangente hiperbólica de `x` (rad)	`real`	`real`

Para todas aquellas funciones matemáticas que por definición son multivaluadas, el valor retornado es el valor principal. Además de las funciones listadas, suficientemente auto-explicativas, a continuación se detallan otras funciones de uso común.

`abs(x)`

- Si `x` es de tipo `integer` o `real` la función devuelve un dato del mismo tipo y precisión, siendo

$$\texttt{abs(x)} = \begin{cases} x & \text{Si } \texttt{x} >= 0 \\ -x & \text{Si } \texttt{x} < 0 \end{cases}$$

- Si `x` es de tipo `complex` entonces

$$\texttt{abs(x)} = \sqrt{Re^2(\texttt{x}) + Im^2(\texttt{x})}$$

`sign(x,y)`

Transfiere el signo del argumento `y` al argumento `x`. Los argumentos deben ser ambos de tipo `integer` o `real`.

$$\texttt{sign(x,y)} = \begin{cases} |\texttt{x}| & \text{Si } \texttt{y} \geq 0 \\ -|\texttt{x}| & \text{Si } \texttt{y} < 0 \end{cases}$$

`mod(x,y)`

Calcula el resto de la división `x/y`. Los argumentos deben ser ambos de tipo `integer` o `real`.

$$\texttt{mod(x,y)} = \begin{cases} \texttt{x} - \texttt{int(x/y)} \times \texttt{y} & \text{Si } \texttt{y} \neq 0 \\ \text{depende del compilador} & \text{Si } \texttt{y} = 0 \end{cases}$$

6.3. Funciones intrínsecas específicas de caracteres

6.3.1. Funciones de conversión

En la tabla 6.4 aparecen las funciones elementales necesarias para convertir un carácter en su correspondiente número de orden y viceversa.

Tabla 6.4 Funciones de conversión de caracteres

Nombre función	Valor devuelto	Tipo argumento	Tipo función
`achar(i)`	Carácter situado en la posición `i` de la secuencia de comparación ASCII	`integer` $(0 \leq i \leq 127)$	`character`
`iachar(ch)`	Número de orden, dentro de la secuencia de comparación ASCII, del carácter `ch`	`character`	`integer`
`char(i)`	Carácter situado en la posición `i` de la secuencia de comparación del procesador	`integer`	`character`
`ichar(ch)`	Número de orden, dentro de la secuencia de comparación del procesador, del carácter `ch`	`character`	`integer`

6.3.2. Funciones de manipulación

En esta sección aparecen las funciones elementales necesarias para manipular subcadenas de caracteres dentro de una cadena de caracteres. A lo largo de la sección utilizaremos los términos `string`, `substring` y `set` para denotar indistintamente una cadena de caracteres o un dato de tipo `character`. Para especificar que es un dato usaremos el término `ch`. Además aparece la palabra clave `back`.

`len(string)`

Devuelve la longitud de `string`

```
len('␣␣Fortran␣&␣magic␣␣␣') = 20
```

`len_trim(string)`

Devuelve la longitud de `string` sin contar los espacios en blanco de la derecha.

```
len_trim('␣␣Fortran␣&␣magic␣␣␣') = 17
```

`trim(string)`

Devuelve el mismo `string` pero quitando los espacios en blanco de la derecha.

```
trim('␣␣Fortran␣&␣magic␣␣␣') = '␣␣Fortran␣&␣magic'
```

Figura 6.2 Manipulación de datos de tipo carácter

```
program main
    character(len=25) :: ch
    ch = '␣␣Fortran␣&␣magic␣␣␣'
    write(*,*) len(ch)                  ! resultado = 25
    write(*,*) len_trim(ch)             ! resultado = 17
    write(*,*) len(trim(ch))            ! resultado = 17
end program main
```

`adjustl(string)`

Ajusta `string` a la izquierda, manteniendo su longitud

`adjustl('␣␣Fortran␣&␣magic␣␣␣') = 'Fortran␣&␣magic␣␣␣␣␣'`

`adjustr(string)`

Ajusta `string` a la derecha, manteniendo su longitud

`adjustr('␣␣Fortran␣&␣magic␣␣␣') = '␣␣␣␣␣Fortran␣&␣magic'`

`index(string, substring,` [`back`= *valor*]`)`

Devuelve un número entero que especifica la posición que ocupa la cadena de caracteres `substring` dentro de la cadena `string`. El argumento `back` es un dato de tipo lógico opcional, y su valor por defecto es `.false.`

- Si `back=.false.` la función `index` identifica la primera aparición de la subcadena `substring` dentro de `string`. Si la subcadena `substring` es de longitud cero, la función `index` devuelve un 1.
- Si `back=.true.` la función `index` identifica la última aparición de la subcadena `substring` dentro de `string`. Si la subcadena `substring` es de longitud cero, la función `index` devuelve el entero `len(string) + 1`.

```
index('aderezado', 'a') = 1
index('aderezado', 'a', back=.true.) = 7

index('aderezado', 'A') = 0
index('aderezado', 'A', back=.true.) = 0
```

```
index('aderezado', 'ad') = 1
index('aderezado', 'ad', back=.true.) = 7

index('aderezado', 'ae') = 0
index('aderezado', 'ae', back=.true.) = 0

index('aderezado', '') = 1
index('aderezado', '', back=.true.) = 10
```

`scan(string, set,` [`back=` *valor*]`)`

Busca en `string` alguno de los caracteres contenidos en la cadena `set`. El argumento `back` es un dato de tipo lógico opcional, y su valor por defecto es `.false.`

- Si `back=.false.` la función `scan` identifica la primera aparición de un carácter de `set` dentro de `string`.
- Si `back=.true.` la función `scan` identifica la última aparición de un carácter de `set` dentro de `string`.

```
scan('aderezado', 'ae') = 1
scan('aderezado', 'ae', back=.true.) = 7

scan('aderezado', 'ea') = 1
scan('aderezado', 'ea', back=.true.) = 7

scan('aderezado', 'Ae') = 3
scan('aderezado', 'Ae', back=.true.) = 5

scan('aderezado', 'Af') = 0
scan('aderezado', 'Af', back=.true.) = 0

scan('aderezado', '') = 0
scan('aderezado', '', back=.true.) = 0
```

6.4. Funciones intrínsecas específicas de *arrays*

Todas las funciones matemáticas intrínsecas estudiadas hasta ahora admiten que sus argumentos sean *arrays*, en cuyo caso si hay más de un argumento, éstos deben ser *arrays* conformes. En estas aplicaciones el resultado es un *array* conforme con los argumentos y cuyos elementos son el resultado de aplicar la función a los argumentos elemento a elemento. Esto significa que, dada una matriz de `n` × `n` de elementos reales llamada `A`, la matriz `B` de `n` × `n` definida por

```
B = exp(A)
```

tiene como elementos $b_{ij} = e^{a_{ij}}$ con $i, j = 1, \ldots, n$ por lo que, matemáticamente, $B \neq e^A$.

A continuación se describen algunas de las funciones intrínsecas que posee Fortran para realizar las operaciones más habituales del álgebra matricial. En lo que sigue utilizaremos la siguiente convención: `vector` para denominar un *array* unidimensional, `matriz` para denominar un *array* bidimensional y `array` para denominar un *array* genérico.

6.4.1. Funciones de cálculo

`sum`

`sum(array)`

Devuelve un escalar que es la suma todos los elementos de `array`. El tipo y el *kind* del escalar coincide con el de `array`.

`sum(array, mask=`***expr. lógica***`)`

Devuelve un escalar que es la suma de todos los elementos de `array` que cumplen la condición dada por ***`expr. lógica`***. Si ningún elemento cumple la condición lógica, devuelve un cero.

`sum(matriz, dim=`***`num`***`)`

Si ***`num`*** `= 1`, devuelve un vector cuyos elementos son la suma de las columnas de la matriz. Si ***`num`*** `= 2`, devuelve un vector cuyos elementos son la suma de las filas de la matriz.

`sum(matriz, dim=`***`num`***`, mask=`***`expr. lógica`***`)`

Si ***`num`*** `= 1`, devuelve un vector cuyos elementos son la suma de las columnas de la matriz que cumplen la condición dada por ***`expr. lógica`***. Si ***`num`*** `= 2`, devuelve un vector cuyos elementos son la suma de las filas de la matriz que cumplen la condición dada por ***`expr. lógica`***.

Dada la matriz `A` de tipo `integer`,

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

```
sum(A)                          ! devuelve 9
sum(A, mask=(A>0))              ! devuelve 15
sum(A, mask=(A>20))             ! devuelve 0
sum(A, dim=1)                   ! devuelve (/7, 1, 1/)
sum(A, dim=2)                   ! devuelve (/4, 5/)
sum(A, dim=2, mask=(A>0))       ! devuelve (/6, 9/)
sum(A, dim=2, mask=(A>5))       ! devuelve (/0, 6/)
```

`product`

`product(array)`

Devuelve un escalar que es el producto de todos los elementos de `array`. El tipo y el *kind* del escalar coincide con el de `array`.

`product(array, mask=`***expr. lógica***`)`

Devuelve un escalar que es el producto de todos los elementos de `array` que cumplen la condición dada por ***expr. lógica***. Si ningún elemento cumple la condición lógica, devuelve un uno.

`product(matriz, dim=`***num)***

Si ***num*** `= 1`, devuelve un vector cuyos elementos son el producto de las columnas de la matriz. Si ***num*** `= 2`, devuelve un vector cuyos elementos son el producto de las filas de la matriz.

`product(matriz, dim=`***num***`, mask=`***expr. lógica)***

Si ***num*** `= 1`, devuelve un vector cuyos elementos son el producto de las columnas de la matriz que cumplen la condición dada por ***expr. lógica***. Si ***num*** `= 2`, devuelve un vector cuyos elementos son el producto de las filas de la matriz que cumplen la condición dada por ***expr. lógica***.

Dada la matriz `A` de tipo `integer`,

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

```
product(A)                        ! devuelve 720
product(A, mask=(A>0))            ! devuelve 90
product(A, mask=(A>20))           ! devuelve 1
product(A, dim=1)                 ! devuelve (/6, -20, -6/)
product(A, dim=2)                 ! devuelve (/-10, -72/)
product(A, dim=2, mask=(A>0))     ! devuelve (/5, 18/)
product(A, dim=2, mask=(A>5))     ! devuelve (/1, 6/)
```

`dot_product(vector_1,vector_2)`

Esta función devuelve el producto escalar de dos vectores conformes de datos numéricos. El tipo de `vector_1` y `vector_2` no es necesario que sea coincidente. El resultado sigue las reglas usuales de conversión aritmética.

- Si `vector_1` es de tipo `integer` o `real`, el resultado es el mismo que el de

  ```
  sum(vector_1*vector_2)
  ```

- Si `vector_1` es de tipo `complex`, el resultado es el mismo que el de

  ```
  sum(conjg(vector_1)*vector_2)
  ```

`matmul(array_1,array_2)`

Esta función devuelve el producto de dos *arrays* con formas compatibles con las reglas algebraicas. Al menos uno de los dos argumentos debe ser una matriz. El tipo de `array_1` y `array_2` no es necesario que sea coincidente. El resultado sigue las reglas usuales de conversión aritmética.

`transpose(matriz)`

Esta función devuelve la matriz transpuesta de `matriz`, con el mismo tipo y *kind*. La matriz no es necesario que sea una matriz cuadrada.

6.4.2. Funciones de búsqueda

`maxval — minval`

`maxval(array)`

Devuelve un escalar que es el máximo de los elementos de `array`. El tipo y el *kind* del escalar coincide con el de `array`, teniendo en cuenta que `array` no puede ser de tipo `complex`.

`maxval(array, mask=`***expr. lógica***`)`

Devuelve un escalar que es el máximo de los elementos de `array` que cumple la condición dada por ***expr. lógica***. Si ningún elemento cumple la condición lógica, devuelve un número arbitrario, dependiente del procesador.

`maxval(matriz, dim=`***num)***

Si ***num*** `= 1`, devuelve un vector cuyos elementos son el máximo de cada una de las columnas de la matriz. Si ***num*** `= 2`, devuelve un vector cuyos elementos son el máximo de cada una de las filas de la matriz.

`maxval(matriz, dim=`***num***`, mask=`***expr. lógica)***

Si ***num*** `= 1`, devuelve un vector cuyos elementos son el máximo de cada una de las columnas de la matriz que cumplen la condición dada por ***expr. lógica***. Si ***num*** `= 2`, devuelve un vector cuyos elementos son el máximo de cada una de las filas de la matriz que cumplen la condición dada por ***expr. lógica***.

Dados el vector `U` y la matriz `A`, ambos de tipo `integer`,

$$U = (-5, 2, 4, -7) \qquad A = \begin{pmatrix} 1 & 5 & -2 \\ 6 & -4 & 3 \end{pmatrix}$$

```
maxval(U)                          ! devuelve 4
maxval(U, mask=(U<0))              ! devuelve -5
maxval(A)                          ! devuelve 6
maxval(A, mask=(A<5))              ! devuelve 3
maxval(A, dim=1)                   ! devuelve (/6, 5, 3/)
maxval(A, dim=2)                   ! devuelve (/5, 6/)
maxval(A, dim=1, mask=(A<0))       ! devuelve (/-1200, -4, -2/)
```

`maxloc — minloc`

`maxloc(vector)`

Devuelve un vector de un elemento que es la posición (no el índice) que ocupa el mayor elemento del vector. En caso de que haya varios máximos, solo devuelve la posición del primero de ellos.

`maxloc(vector, mask=`***expr. lógica***`)`

Devuelve un vector de un elemento que es la posición (no el índice) que ocupa el mayor elemento del vector que cumple la condición dada en ***expr. lógica***.

En caso de que haya varios máximos, solo devuelve la posición del primero de ellos. Si ningún elemento cumple la condición lógica devuelve el vector cero.

`maxloc(matriz)`

Devuelve un vector de dos elementos que es la posición (no los índices) que ocupa el mayor elemento de la matriz. En caso de que haya varios máximos, solo devuelve la posición del primero de ellos, considerando que la matriz se recorre por columnas.

`maxloc(matriz, mask=`***expr. lógica***`)`

Devuelve un vector de dos elementos que es la posición (no los índices) que ocupa el mayor elemento de la matriz que cumple la condición dada en ***expr. lógica***. En caso de que haya varios máximos, solo devuelve la posición del primero de ellos, considerando que la matriz se recorre por columnas. Si ningun elemento cumple la condición lógica devuelve el vector cero.

Dados el vector `U` y la matriz `A`, ambos de tipo `integer`,

$$U = (-5, 2, 4, -7) \qquad A = \begin{pmatrix} 1 & 5 & -2 \\ 6 & -4 & 3 \end{pmatrix}$$

```
maxloc(U)                          ! devuelve (/3/)
maxloc(U, mask=(U<0))              ! devuelve (/1/)
maxloc(U, mask=(U>10))             ! devuelve (/0/)
maxloc(A)                          ! devuelve (/2, 1/)
maxloc(A, mask=(A<5))              ! devuelve (/2, 3/)
maxloc(A, mask=(A>10))             ! devuelve (/0, 0/)
```

`all(`***expr. lógica***`)`

Devuelve la constante literal `.true.` si todos los elementos del *array* verifican la condición ***expr. lógica***, y la constante `.false.` si alguno los elementos del *array* no la verifica.

Si el *array* es una matriz, se puede utilizar el argumento opcional `dim` para especificar si se evalúan las columnas (`dim=1`) o las filas (`dim=2`). En ese caso el resultado es un vector.

`any(expr. lógica)`

Devuelve la constante literal `.true.` si algún elemento del *array* verifica la condición ***expr. lógica***, y la constante `.false.` si ninguno de los elementos del *array* la verifica.

Si el *array* es una matriz, se puede utilizar el argumento opcional `dim` para especificar si se evalúan las columnas (`dim=1`) o las filas (`dim=2`). En ese caso el resultado es un vector.

`count(expr. lógica)`

Devuelve un dato de tipo `integer` que cuenta el número de elementos del *array* que verifican la condición ***expr. lógica***.

Si el *array* es una matriz, se puede utilizar el argumento opcional `dim` para especificar si se evalúan las columnas (`dim=1`) o las filas (`dim=2`). En ese caso el resultado es un vector.

Dada la matriz `B` de tipo `integer`,

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

```
all(B>0)                ! devuelve F
any(B>0)                ! devuelve T
count(B>0)              ! devuelve 3
all(B>0, dim=1)         ! devuelve (/T, F, F/)
any(B>0, dim=1)         ! devuelve (/T, T, F/)
count(B>0, dim=1)       ! devuelve (/2, 1, 0/)
all(B>0, dim=2)         ! devuelve (/F, F/)
any(B>0, dim=2)         ! devuelve (/T, T/)
count(B>0, dim=2)       ! devuelve (/2, 1/)
```

6.4.3. Funciones de dimensiones

`size`

`size(array)`

Devuelve un escalar de tipo `integer` igual al número total de elementos de `array`.

`size(matriz, dim=`***num)***

Si ***num*** `= 1`, devuelve un escalar igual al número de filas de la matriz. Si ***num*** `= 2`, devuelve un escalar igual al número de columnas de la matriz.

`shape(array)`

Devuelve un vector de tipo `integer` con tantos elementos como dimensiones tiene `array`, que contiene el número de elementos que hay en cada dimensión de `array`.

`lbound`

`lbound(array)`

Devuelve un vector de tipo `integer`, con tantos elementos como dimensiones tiene `array`, que contiene el límite inferior de cada una de las dimensiones de `array`.

`lbound(matriz, dim=`***num)***

Si ***num*** `= 1`, devuelve un escalar igual al límite inferior de las filas de la matriz. Si ***num*** `= 2`, devuelve un escalar igual al límite inferior de las columnas de la matriz.

`ubound`

`ubound(array)`

Devuelve un vector de tipo `integer`, con tantos elementos como dimensiones tiene `array`, que contiene el límite superior de cada una de las dimensiones de `array`.

`ubound(matriz, dim=`***num)***

Si ***num*** `= 1`, devuelve un escalar igual al límite superior de las filas de la matriz. Si ***num*** `= 2`, devuelve un escalar igual al límite superior de las columnas de la matriz.

`reshape(vector_1,vector_2)`

Esta función construye un *array* con la forma dada por el vector `vector_2` y los datos contenidos en `vector_1`.

```
program main
    integer :: n
    integer :: U(2)
    integer :: V(1)
    real    :: T(10:12)
    integer :: A(2,3)
    integer :: B(0:10,4:8)
    real    :: Zt(0:2,3)
    integer :: Vu(2)
    integer :: C(6)
    real    :: Cr(9)

    ! Funcion size
    n = size(T)                 ! n = 3
    n = size(A)                 ! n = 6
    n = size(A, dim=1)          ! n = 2
    n = size(A, dim=2)          ! n = 3
    n = size(B)                 ! n = 55
    n = size(B, dim=1)          ! n = 11
    n = size(B, dim=2)          ! n = 5
    n = size(Zt)                ! n = 9
    n = size(Zt, dim=1)         ! n = 3
    n = size(Zt, dim=2)         ! n = 3

    ! Funcion shape
    V = shape(T)                ! V = (/3/)
    U = shape(A)                ! U = (/2, 3/)
    U = shape(B)                ! U = (/11, 5/)
    U = shape(Zt)               ! U = (/3, 3/)

    ! Funcion lbound
    V = lbound(T)               ! V = (/10/)
    U = lbound(A)               ! U = (/1, 1/)
    n = lbound(A, dim=1)        ! n = 1
    n = lbound(A, dim=2)        ! n = 1
    U = lbound(B)               ! U = (/0, 4/)
    n = lbound(B, dim=1)        ! n = 0
```

```
    n = lbound(B, dim=2)          ! n = 4
    U = lbound(Zt)                ! U = (/0, 1/)
    n = lbound(Zt, dim=1)         ! n = 0
    n = lbound(Zt, dim=2)         ! n = 1

    ! Funcion ubound
    V = ubound(T)                 ! V = (/12/)
    U = ubound(A)                 ! U = (/2, 3/)
    n = ubound(A, dim=1)          ! n = 2
    n = ubound(A, dim=2)          ! n = 3
    U = ubound(B)                 ! U = (/10, 8/)
    n = ubound(B, dim=1)          ! n = 10
    n = ubound(B, dim=2)          ! n = 8
    U = ubound(Zt)                ! U = (/2, 3/)
    n = ubound(Zt, dim=1)         ! n = 2
    n = ubound(Zt, dim=2)         ! n = 3

    ! Funcion reshape
    Vu = (/2, 3/)
    C = (/1, 6, 5, -4, -2, 3/)
    A = reshape(C, Vu)

    Vu = (/3, 3/)
    Cr = 4.0
    Zt = reshape(Cr, Vu)

end program main
```

7 Objetos compuestos

Índice

7.1. Introducción

Hoy en día el uso de objetos compuestos en Fortran se considera una práctica habitual tanto por la sencillez de su manejo como por la compacidad de su notación. Para los *arrays* en particular, se ha visto que Fortran posee un conjunto de funciones intrínsecas que los dotan de una potencia de cálculo superior a la de cualquier otro lenguaje de programación. En el caso de las estructuras, el poder representar de forma compacta un objeto (una figura geométrica, una agenda personal, etc) ha permitido en los últimos años un estilo de programación mucho más claro y estructurado.

Aunque los objetos compuestos ya han sido definidos y parcialmente estudiados, en este capítulo vamos a desarrollar algunos aspectos típicos de su manejo que, por cuestiones de claridad en la exposición, no se han desarrollado anteriormente.

En particular, en el caso de los *arrays* vamos a estudiar dos puntos de gran interés,

1. La construcción de secciones de un *array*. ¿Cómo se puede extraer una matriz o un vector de una matriz dada?

2. La asignación dinámica de memoria. ¿Es posible declarar un *array* del cual se sabe cuántas dimensiones tiene, pero no cuáles son sus límites?

De los datos de tipo derivado nos vamos a ocupar de dos cuestiones que, aunque son sencillas, generalmente dan lugar a confusión,

1. El uso de un *array* como componente de una estructura

2. El uso de una estructura como componente de otra estructura

7.2. Datos *array*

7.2.1. Secciones de un *array*

Una sección de un *array* es un *subarray* definido a partir del mismo sin más que seleccionar determinado rango de variación de los índices en cada dimensión. Por ejemplo a partir de una matriz se puede extraer una submatriz sin más que seleccionar determinadas filas y columnas.

La forma más habitual de referenciar una sección de un *array* consiste en especificar el nombre del *array* indicando, para cada dimensión, los valores inicial y final de la sección, pudiéndose indicar un incremento de forma opcional. Si se omiten los valores

inicial o final, se consideran los valores correspondientes a la declaración del *array*. Si se omite el incremento, se considera que vale 1.

`nombre([ini_1]:[fin_1][:inc_1],[ini_2]:[fin_2][:inc_2],...)`

Así a partir de los *arrays* `V` y `A` siguientes

$$V = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$$

$$A = \begin{pmatrix} 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\ 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 \\ 61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 \end{pmatrix}$$

declarados como

```
integer :: V(0:10)
integer :: A(6,8)
```

se pueden definir los siguientes ejemplos de *subarrays*

`V(0:10:2)`, `V( : :2)` (vector de 6 elementos)

$$V = (\boxed{0}, 1, \boxed{2}, 3, \boxed{4}, 5, \boxed{6}, 7, \boxed{8}, 9, \boxed{10})$$

`V(3:10)`, `V(3: )` (vector de 8 elementos)

$$V = (0, 1, 2, \boxed{3}, \boxed{4}, \boxed{5}, \boxed{6}, \boxed{7}, \boxed{8}, \boxed{9}, \boxed{10})$$

`A(1:3,1:4)`, `A( :3, :4)` (matriz 3×4)

$$A = \begin{pmatrix} \boxed{\begin{matrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{matrix}} & \begin{matrix} 15 & 16 & 17 & 18 \\ 25 & 26 & 27 & 28 \\ 35 & 36 & 37 & 38 \end{matrix} \\ \begin{matrix} 41 & 42 & 43 & 44 \\ 51 & 52 & 53 & 54 \\ 61 & 62 & 63 & 64 \end{matrix} & \begin{matrix} 45 & 46 & 47 & 48 \\ 55 & 56 & 57 & 58 \\ 65 & 66 & 67 & 68 \end{matrix} \end{pmatrix}$$

`A(2:6:2,1:7:3)`, `A(2: :2, :7:3)` (matriz 3×3)

$$A = \begin{pmatrix} 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ \boxed{21} & 22 & 23 & \boxed{24} & 25 & 26 & \boxed{27} & 28 \\ 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 \\ \boxed{41} & 42 & 43 & \boxed{44} & 45 & 46 & \boxed{47} & 48 \\ 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 \\ \boxed{61} & 62 & 63 & \boxed{64} & 65 & 66 & \boxed{67} & 68 \end{pmatrix}$$

`A(2:5,7)` (vector de 4 elementos)

$$A = \begin{pmatrix} 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 21 & 22 & 23 & 24 & 25 & 26 & \boxed{27} & 28 \\ 31 & 32 & 33 & 34 & 35 & 36 & \boxed{37} & 38 \\ 41 & 42 & 43 & 44 & 45 & 46 & \boxed{47} & 48 \\ 51 & 52 & 53 & 54 & 55 & 56 & \boxed{57} & 58 \\ 61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 \end{pmatrix}$$

`A(2:5,7:7)` (matriz de 4×1)

$$A = \begin{pmatrix} 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 21 & 22 & 23 & 24 & 25 & 26 & \boxed{27} & 28 \\ 31 & 32 & 33 & 34 & 35 & 36 & \boxed{37} & 38 \\ 41 & 42 & 43 & 44 & 45 & 46 & \boxed{47} & 48 \\ 51 & 52 & 53 & 54 & 55 & 56 & \boxed{57} & 58 \\ 61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 \end{pmatrix}$$

Otra forma de referenciar una sección de un *array* consiste en especificar el nombre del *array* y, entre paréntesis, un vector para cada dimensión del *array* con los subíndices a los que se hace alusión. Este vector se puede dar en forma de *array* de rango 1 de constantes literales o en forma de dato compuesto. Esta forma permite mayor flexibilidad a la hora de ordenar los índices seleccionados, a costa de sacrificar la elegancia y claridad de la especificación de una sección dada anteriormente.

De nuevo hay que distinguir entre extraer a partir de una matriz un vector o una matriz en la que una de sus dimensiones es 1.

Figura 7.1 Secciones de un *array*

```
program main
    integer :: V(0:10)
    integer :: A(6,8)
    integer :: Uf(4)
    integer :: Uc(5)
    integer :: B(4,5)
    integer :: U(4)
    integer :: vW(4)
    integer :: mW(1,4)
       .
       .
    Uf = (/2, 1, 5, 1/)
    Uc = (/3, 6, 2, 4, 3/)
    U = V(Uf)
    B = A(Uf, Uc)
    vW = A(2, Uf)
    mW = A(2:2, Uf)
       .
       .
end program main
```

En la figura 7.1 se muestra parte de un programa en el que, a partir de los *arrays* `V` y `A` dados anteriormente, se definen las secciones `U`, `B`, `vW`, `mW` cuyos valores son

$$U = (1, 0, 4, 0) \qquad B = \begin{pmatrix} 23 & 26 & 22 & 24 & 23 \\ 13 & 16 & 12 & 14 & 13 \\ 53 & 56 & 52 & 54 & 53 \\ 13 & 16 & 12 & 14 & 13 \end{pmatrix} \qquad mW = \begin{pmatrix} 22 \\ 21 \\ 25 \\ 21 \end{pmatrix}$$

$$vW = (22, 21, 25, 21)$$

7.2.2. Asignación dinámica de memoria

En la sección 3.4 hemos visto cómo se declara un *array* de tamaño fijo. Este tipo de declaración se utiliza para reservar una cantidad fija de espacio en memoria en tiempo de compilación. En ocasiones el tamaño del *array* solo se conoce en tiempo de ejecución, bien porque va a ser leído de una fuente externa o bien porque es el resultado de una operación

intermedia. Es más, es muy habitual en un programa en el que se estudian varios casos de un mismo problema, utilizar un mismo *array* que actualiza su tamaño dependiendo de las necesidades propias del caso en el que se encuentra.

Esto significa que la declaración estática de *arrays* tiene dos inconvenientes,

1.- Si el tamaño prefijado excede el número de valores que se van a almacenar, estamos reservando memoria para elementos que luego no se van a utilizar.

2.- Si, por el contrario, el tamaño prefijado es menor que el número de valores que se van a utilizar, el programa dará un error de ejecución.

Para evitar estos inconvenientes, Fortran permite asignar memoria a ciertos *arrays*, llamados *dinámicos*, en tiempo de ejecución. Es decir, durante la ejecución, por medio de una sentencia, se especifican las dimensiones que va a tener el *array* y se reserva la cantidad de memoria necesaria para el mismo, siempre que en ese momento haya espacio suficiente en la memoria de cálculo del procesador. Si por razones del algoritmo es necesario variar las dimensiones del *array*, primero se tiene que liberar el espacio de memoria que ocupa actualmente, a través de otra sentencia específica, y luego volver a reservar una nueva cantidad de memoria. Una vez que se ha terminado de utilizar el *array* en el programa es necesario liberar el espacio de memoria ocupado.

Todo *array* dinámico debe ser declarado con el atributo `allocatable` especificando su rango pero dejando los límites sin definir, para ello se puede utilizar o no el atributo `dimension`.

Para reservar la cantidad de memoria necesaria para almacenar los valores de un *array* dinámico se utiliza la sentencia `allocate` con la que se definen los límites del *array*. A partir de este momento podemos utilizar el *array* como un objeto compuesto más del programa, recordando que en este punto todavía no está definido.

Para liberar la memoria reservada, se utiliza la sentencia `deallocate`.

Las sentencias `allocate` y `deallocate` tienen un *especificador* opcional denominado `stat` cuyo valor debe cargarse en una variable de tipo `integer`. Si la reserva/liberación de memoria se ha producido de modo correcto, el valor de `stat` es `0`, en caso contrario es positivo. En ausencia de este especificador, si el procesador no es capaz de ejecutar alguna de las dos sentencias, el programa se interrumpe de forma brusca.

En la figura 7.2 se muestra un ejemplo de cómo se reserva (`allocate`) y se libera (`deallocate`) memoria a lo largo de un programa.

Figura 7.2 Asignación dinámica de memoria

```
program main
    real, allocatable :: U(:)
    real, allocatable :: A(:,:)
    integer :: n
    integer :: m
    integer :: Ierr
       .
       .
    write(*,*) 'Introducir la dimension del vector'
    read(*,*) n

    ! Reserva de memoria para el vector U
    allocate(U(0:n), stat=Ierr)
    if (Ierr > 0) stop "*** No hay memoria suficiente &
                          &para alocatar U ***"

    write(*,*) 'Introducir las dimensiones de A'
    read(*,*) n, m

    ! Reserva de memoria para la matriz A
    allocate(A(n,m), stat=Ierr)
    if (Ierr > 0) stop "*** No hay memoria suficiente &
                          &para alocatar A ***"
       .
       .
    ! Liberacion de memoria de U
    deallocate(U, stat=Ierr)
    if (Ierr > 0) stop "*** U no estaba alocatado ***"
       .
       .
    ! Liberacion de memoria de A
    deallocate(A, stat=Ierr)
    if (Ierr > 0) stop "*** A no estaba alocatado ***"
       .
end program main
```

7.2.3. *Arrays* de tamaño cero

Aunque pueda parecer una incongruencia, Fortran permite trabajar con *arrays* de tamaño cero. Un ejemplo típico puede ser: estamos resolviendo varios casos de un mismo problema y *alocatando* un vector `U` según un número, `n`, que varía en cada caso y se calcula durante la ejecución del programa.

```
program main
    real, allocatable :: U(:)
    integer :: n
    integer :: NumCasos
    integer :: i
    integer :: Ierr
       .
    do i=1,NumCasos
       .
       n =
       allocate(U(n), stat=Ierr)
       if (Ierr > 0) stop "*** No se puede alocatar U ***"
       .
       deallocate(U, stat=Ierr)
       if (Ierr > 0) stop "*** U no estaba alocatado ***"
       .
   end do
end program main
```

Si en alguno de los casos resulta que `n` toma un valor no positivo, el programa sigue funcionando con normalidad sin tener que haber añadido líneas de código extra.

Si se definen secciones de un *array* y en algún momento el límite inferior especificado para una dimensión supera al superior, el *subarray* resultante es de tamaño cero. Esto ocurre en el siguiente ejemplo donde el límite inferior de la sección está dado en función del contador del bucle, `i`, de modo que cuando éste adquiere el valor `n-1` la sección definida es un *array* de tamaño cero.

```
program main
    integer, parameter :: n = 20
    real    :: U(n)
    real    :: a
    integer :: i
```

```
        .
    do i=1,n
        .
        U(i+2:n) = a*0.5
        .
   end do
end program main
```

No hay que confundir la situación anterior con el error que se comete cuando en una sección se establece un límite inferior o superior fuera del rango de los límites establecidos en la declaración del *array*.

```
program main
    integer, parameter :: n = 20
    real    :: U(n)
    real    :: a
    integer :: i
        .
    do i=1,n
        .
        U(i:i+2) = a*0.5         ! cuando i=n-1 hay un error
        .
   end do
end program main
```

Se debe resaltar el hecho de que dos *arrays* de tamaño cero pueden tener el mismo rango pero distinta forma, por ejemplo uno puede ser `(0,2)` y el otro `(2,0)`, con lo que no son conformes y, por tanto, no pueden ser operandos de la misma expresión. Al igual que todo *array*, un *array* de tamaño cero es conforme con cualquier escalar.

7.3. Datos de tipo derivado

En esta sección vamos a profundizar en el uso de los objetos compuestos de tipo derivado vistos en el capítulo 3.

La única operación intrínsecamente definida en Fortran entre datos de tipo derivado es la asignación, teniendo en cuenta que en la expresión

```
dato_1 = dato_2
```

ambos datos deben ser del mismo tipo. En Fortran, de forma general, existe la posibilidad de *definir* funciones y operadores. Esta posibilidad se puede extender al caso de operandos de tipo derivado lo que amplía el conjunto de operaciones que se pueden realizar con ellos. Sin embargo, este tipo de operaciones se sale del propósito del presente curso.

7.3.1. Componente de tipo *array*

Un tipo derivado puede tener como componentes datos de tipo *array*. En la definición de un tipo derivado con una componente *array*, ésta debe ser declarada con la forma totalmente especificada. Es decir, una componente de tipo *array* no puede ser declarada con el atributo `allocatable`.

En la figura 7.3 se muestra un ejemplo de *array* como componente de una estructura, cumpliendo las limitaciones anteriormente descritas. Además, para hacer hincapié en el hecho de que son dos conceptos distintos, hemos declarado un vector dinámico de estructuras. Como todo *array* dinámico, antes de definirlo, hay que *alocatarlo* (fijar sus dimensiones y reservar memoria). Una vez utilizado, y antes de finalizar el programa, hay que *dealocatarlo* (liberar el espacio de memoria reservada).

Para acceder a una componente del elemento $i-$ésimo del *array* de estructuras, se utiliza la expresión

```
nombre_array(i)%componente
```

mientras que para acceder al elemento $j-$ésimo de una componente del elemento $i-$ésimo del *array* de estructuras, se utiliza la expresión

```
nombre_array(i)%componente(j)
```

Figura 7.3 *Array* como componente de un tipo derivado

```
program main
    integer, parameter :: numnotas = 3
    type alumno
        character(len=15) :: nombre
        character(len=30) :: apellidos
        real :: nota(numnotas)
        real :: media
    end type alumno
    type(alumno), allocatable :: clase(:)
    integer :: n
    integer :: i
    integer :: Ierr

    write(*,*) 'Introducir numero de alumnos'
    read(*,*) n
    allocate(clase(n), stat=Ierr)
    if (Ierr > 0) stop '*** clase no se puede alocatar ***'

    do i=1,n
        write(*,*) 'Nombre y apellidos del alumno ', i
        read(*,*) clase(i)%nombre, clase(i)%apellidos

        write(*,*) 'Introducir notas del alumno ', i
        read(*,*) clase(i)%nota(:)

        clase(i)%media = sum(clase(i)%nota)/numnotas

        write(*,*) 'La media del alumno ', i, 'es:'
        write(*,*) clase(i)%media
    end do
    deallocate(clase, stat=Ierr)
    if (Ierr > 0) stop '*** clase no estaba alocatado ***'

end program main
```

7.3.2. Componente de tipo estructura

Por último, dando el siguiente paso en complejidad, una componente de un tipo derivado puede ser de otro tipo derivado distinto, y definido con anterioridad.

Para acceder a una componente de una componente de una estructura se utiliza la cadena apropiada de *selectores de componentes* sin paréntesis, tal y como se muestra en el ejemplo (Fig. 7.4), donde se ilustran, por una parte, las diferentes maneras de definir una estructura con una componente de tipo derivado, y por otra, el hecho de que el nombre de la componente de una estructura puede coincidir con el nombre de un dato sin que dé lugar a confusión, ya que se utilizan de forma completamente distinta.

Figura 7.4 Componente de tipo derivado

```
program main
    type punto
        real :: x
        real :: y
        real :: z
    end type punto
    type circulo
        type(punto) :: centro
        real :: radio
    end type circulo
    type(punto) :: P
    type(punto) :: centro
    type(circulo) :: c1
    type(circulo) :: c2
    type(circulo) :: c3

    P%x = 1.0
    P%y = 1.0
    P%z = 1.0

    centro = punto(1.0, 2.0, 0.0)

    c1%centro%x = 1.0
    c1%centro%y = 2.0
    c1%centro%z = 3.0
    c1%radio = 25.0

    c2 = circulo(punto(2.0, 1.0, 0.0), 1.0)

    c3 = circulo(centro, c2%radio)
end program main
```

8 Operaciones de Entrada y Salida

Índice

8.1. Introducción

El objeto de las operaciones de entrada y salida es transferir o recibir datos desde un medio externo (cinta, disco magnético u óptico, pantalla, teclado). La lectura de datos es una operación de entrada y la escritura de datos es una operación de salida. Un *registro* es un conjunto de datos compuesto por *campos* que constituyen las unidades de información más pequeñas e indivisibles. Se denomina *fichero* a un conjunto de registros contenidos en la misma unidad.

registro 1	···	···	···	···
registro 2	···	···	···	···
registro 3	···	···	···	···
registro 4	campo 1	campo 2	campo 3	campo 4

El acceso a los datos de un fichero puede ser *secuencial* o *directo*. En el secuencial, para acceder a un dato situado en un determinado campo, se debe recorrer secuencialmente todo el fichero desde el principio hasta alcanzar la información deseada. En el directo, es posible acceder al dato directamente, sin pasar por los registros anteriores.

Cuando se trabaja con conjuntos muy grandes de datos, es conveniente que sus valores estén contenidos en un fichero y, desde el programa, leer dicho fichero asignando los valores a las variables correspondientes.

Por ejemplo, tenemos un fichero, `datos_in.dat`, con los valores de una matriz de 300 filas y 5 columnas, de la forma

$$\begin{array}{ccccc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{3001} & a_{3002} & a_{3003} & a_{3004} & a_{3005} \end{array}$$

La estructura general sería,

1) Abrir el fichero (`open`)
2) Leer el fichero (`read`)
3) Cerrar el fichero (`close`)

Igualmente, si el programa va a generar una gran cantidad de datos, es conveniente guardarlos en un fichero, por ejemplo `datos_out.dat`, con el que se pueda trabajar. La estructura sería,

1) Abrir el fichero (`open`)
2) Escribir el fichero (`write`)
3) Cerrar el fichero (`close`)

Un fichero puede abrirse y cerrarse varias veces a lo largo del programa. Solo hay que tener la precaución de respetar dos normas lógicas, no se puede cerrar un fichero que no esté abierto y, no se puede abrir un fichero que no esté cerrado.

8.2. Sentencias `read` y `print` básicas

Se utilizan para imprimir (`print`) por pantalla o leer (`read`) de teclado, tanto datos como texto. Cada vez que se ejecuta una de estas sentencias, se salta de línea.

La estructura general, cuando se escribe o lee sin formato, es

```
print*, lista_de_variables
read*, lista_de_variables
```

La estructura general, cuando se escribe o lee con formato, es

```
print '(form1, form2, ...)', lista_de_variables
read '(form1, form2, ...)', lista_de_variables
```

Si se desea dejar una línea en blanco en la salida por pantalla, basta con poner

```
print*
```

8.3. Manejo de ficheros

8.3.1. Sentencia `open`

La estructura general es,

```
open(unit=u, file=nombre, status=estado, action=accion)
```

`unit=u`

`u` es un dato simple o una constante entera positiva. A cada fichero se le asigna una unidad que viene determinada por este número de forma que, a partir de este punto, nos podemos referir al fichero en cuestión solo a través de su unidad sin tener que especificar su nombre. Hay que tener en cuenta que,

- Especificar la unidad es imprescindible.
- Si la unidad va a ser el primer especificador, se puede omitir la palabra clave `unit`.
- No se puede asignar la misma unidad a dos ficheros distintos, si se van a utilizar simultáneamente.

`file=`*`nombre`*

`nombre` es el nombre del fichero que se va a abrir. El nombre se puede dar en forma de constante literal o de dato de tipo `character`

- Especificar la palabra clave `file` y el nombre del fichero es imprescindible.
- Si el fichero no está en el mismo directorio, es necesario indicar la dirección completa.
- El valor de *`nombre`* debe ser una reproducción literal del nombre del fichero (incluidas mayúsculas y minúsculas).
- El nombre de un fichero creado durante la ejecución de un programa debe cumplir las normas típicas del sistema operativo con el que se trabaja.

`status=`*`estado`*

`estado` es una constante literal o un dato de tipo `character` que puede tomar cualquiera de los siguientes valores:

`new` el fichero que vamos a abrir no existe, y lo crea el programa al ejecutarse.

`old` el fichero que vamos a abrir ya existe.

`unknown` no sabemos si existe o no. El programa busca en la dirección especificada, si lo encuentra lo abre y si no lo encuentra lo crea.

`scratch` el fichero que vamos a abrir es temporal, de forma que, o bien cuando termina de ejecutarse el programa o bien si lo cerramos antes de terminar la ejecución, el fichero se borra. Con esta opción no es necesario especificar el nombre del fichero.

En algunos procesadores el especificador `status` es opcional en cuyo caso su valor por defecto es `unknown`.

`action=`*`accion`*

`accion` es una constante literal o un dato de tipo `character` que puede tomar cualquiera de los siguientes valores:

`read` el fichero que vamos a abrir se va a utilizar exclusivamente para leer, por lo que si se intenta utilizar para escribir, dará un error de compilación.

`write` el fichero que vamos a abrir se va a utilizar exclusivamente para escribir, por lo que si se intenta utilizar para leer, dará un error de compilación.

`readwrite` el fichero que vamos a abrir se puede utilizar tanto para leer como para escribir.

El argumento `action` es opcional. Su valor por defecto depende del procesador.

Ejemplos de apertura de un fichero son:

```
open(15, file='Din.dat', status='old', action='read')
open(20, file='Dout.dat', status='unknown', action='write')
open(100, status='scratch', action='readwrite')
```

En los ejemplos anteriores hemos utilizado el orden habitual en el que se suelen poner los especificadores. Si bien este orden se puede alterar, recordando que para cambiar de lugar la unidad es obligatoria la palabra clave, lo recomendable es no hacerlo.

```
open(15, status='old', file='Din.dat', action='read')
open(status='old', unit=15, file='Din.dat', action='read')
open(unit=15, action='read', status='old', file='Din.dat')
```

8.3.2. Sentencias `read` y `write`

Una vez que hemos abierto el fichero, ya estamos en disposición de leer los datos que contiene y cargarlos en una o varias variables de las declaradas en el programa o escribir los valores de las variables que representan una solución parcial o completa del problema que estamos resolviendo.

La estructura general de las sentencias de lectura y escritura es,

```
read(unit=u, fmt='(form1,form2,...)', iostat=v) lista_vbles
write(unit=u, fmt='(form1,form2,...)', iostat=v) lista_vbles
```

`unit=u`

`u` es un dato o una constante entera positiva que especifica el fichero del que se va a leer o escribir

- Especificar la unidad es imprescindible.
- Si la unidad va a ser el primer especificador, se puede omitir la palabra clave `unit`
- Si los datos se van a leer del teclado o escribir en la pantalla, en vez de `unit=u` se pone un asterisco (`*`).

`fmt='(form1,form2,...)'`

`'(form1,form2,...)'` es la lista de formatos con los que se va a leer o escribir los datos. En la mayoría de los casos, si no se conoce el formato o los datos no poseen un formato fijo, se pone un asterisco (`*`). En la sección 8.5 se desarrolla con más detalle el uso de los formatos. Aunque en esta descripción estamos dando la lista de formatos como una constante literal de tipo `character`, también se puede utilizar una variable del mismo tipo.

`iostat=v`

Es posible que mientras leemos o escribimos un fichero se produzca un error. Por ejemplo, nos encontramos con un dato que no tiene el formato especificado, o el fichero no tiene tantas filas como creíamos, etc. También puede ocurrir que queramos leer un fichero de datos del que ignoramos el número total de registros que contiene. El argumento `iostat` sirve para controlar este tipo de incidencias. `v` es una variable entera que puede tomar los siguientes valores: `v>0` si detecta un error de lectura o escritura, `v<0` si se alcanza el final del fichero durante la lectura y `v=0` si la ejecución es correcta. Este argumento es opcional.

`lista_vbles`

Es la lista de variables, separadas por comas, en las que se van a cargar los datos que se van a leer, o cuyos valores se van a escribir.

- Si se desea leer un registro, vacío o no, sin asignar ninguno de sus campos a variables del programa, basta con poner

```
read(u,*)
```

donde `u` es la unidad de la que se va a leer.

- Si se desea escribir un registro en blanco, basta con poner

```
write(u,*)
```

donde `u` es la unidad en la que se va a escribir.

8.3.3. Sentencia `close`

La estructura general es,

```
close(unit=u, status=estado)
```

`unit=u`

`u` es un dato o una constante entera positiva que especifica el fichero que se va a cerrar

- Especificar la unidad es imprescindible.
- Si la unidad va a ser el primer especificador, se puede omitir la palabra clave `unit`

`status=`*`estado`*

`estado` es una constante literal o un dato de tipo `character` que puede tomar cualquiera de los siguientes valores:

`keep` significa que el fichero se va a guardar después de la ejecución del programa. No se pude utilizar con un fichero que se ha abierto con la especificación `scratch`.

`delete` significa que el fichero será borrado después de ejecutarse el programa.

El argumento `status` es opcional. Si no se especifica se considera que toma el valor `keep`, salvo que sea un fichero `scratch`.

8.4. Sentencias `rewind` y `backspace`

La sentencia `rewind` se emplea para rebobinar completamente un fichero, es decir, para situar el cursor al principio del mismo. Su estructura general es muy sencilla,

```
rewind(unit=u)
```

o, en su forma más simple,

```
rewind(u)
```

La sentencia `backspace` se emplea para hacer retroceder un registro completo, situándose el cursor al principio del registro que se acaba de leer o escribir. Su estructura general es,

```
backspace(unit=u)
```

`unit=u`

`u` es un dato o una constante entera positiva que especifica la unidad del fichero actual.

- Especificar la unidad es imprescindible.
- Si la unidad va a ser el primer especificador, se puede omitir la palabra clave `unit`

8.5. Especificaciones de formato

Ya hemos visto que hay distintos tipos de datos (`integer, real, complex, character` y `logical`) y distintas formas de expresar sus respectivas constantes literales. Por ejemplo, si `x=3.0`, podemos escribirlo de las siguientes maneras,

```
x = 3.0          x = 3.000
x = 3.           x = 0.3e+1
x = 3e0          x = 3.0e+0
x = 30e-1        x = 30.0e-1
```

todas ellas equivalentes en el sentido de que, matemáticamente el valor del dato `x` es 3, sin embargo cada una de estas formas posee un *formato* diferente.

En esta sección vamos a describir los modos de representación de las constantes literales de los 5 tipos de datos intrínsecos. Hay que tener en mente en todo momento que una cosa es el valor de un dato y otra cómo representamos ese dato, es decir qué formato se utiliza para expresarlo.

El uso típico de la especificación de formatos se da en las sentencias de escritura (`print` y `write`). La razón por la que es preferible visualizar un fichero con formato queda clara en la figura 8.1

Figura 8.1 Visualización del mismo fichero (a) sin formato y (b) con formato

(a)

```
3.0      -0.145   2.34
5.29 0.  -7.893
-1.333 8.54 9.32
```

(b)

```
 3.000   -0.145    2.340
 5.290    0.000   -7.893
-1.333    8.540    9.320
```

Aunque también se puede leer un conjunto de datos especificando el formato, no es recomendable hacerlo ya que, cualquier pequeña desviación del formato especificado produce un error en la ejecución del programa. Esta es una de las razones por la cual no es habitual ver en un fichero de datos de entrada datos numéricos mezclados con datos no numéricos.

8.5.1. Datos tipo `integer`

Especificación

```
In[.m]
```

- `I` significa que la variable es de tipo `integer`
- `n` es una constante entera positiva que indica el número total de espacios que ocupa la representación del valor de la variable (incluyendo el signo "−", si lo hubiera).
- `m` es una constante entera positiva (menor o igual que `n`) que indica el número total de dígitos que aparecen en la representación del valor de la variable (sin incluir el signo "−", si lo hubiera). Para completar el número de dígitos especificado rellena con ceros a la izquierda. Este descriptor es opcional.

8.5.2. Datos tipo `real` en coma flotante

Especificación

```
Fn.d
```

- `F` significa que la variable es de tipo `real`.

Tabla 8.1 Formato de datos de tipo `integer`

Valor interno	Formato	Valor externo
8234	`I4`	`8234`
203	`I3`	`203`
203	`I6`	`␣␣␣203`
-203	`I4`	`-203`
25	`I2`	`25`
25	`I4.4`	`0025`
25	`I7.3`	`␣␣␣␣025`
-25	`I7.3`	`␣␣␣-025`

- `n` es una constante entera positiva que indica el número total de espacios que ocupa la representación del valor de la variable (incluyendo el signo "−", si lo hubiera, y el punto ".").
- `d` es una constante entera no negativa que indica el número total de decimales que van a aparecer, tras redondear.

Tabla 8.2 Formato de datos de tipo `real`

Valor interno	Formato	Valor externo
3.068	`F5.3`	`3.068`
3.068	`F5.2`	`␣3.07`
0.368	`F3.1`	`3.1`
3.068	`F2.0`	`3.`
3.068	`F3.2`	`***`
3.068	`F10.6`	`␣␣3.068000`
-15.234	`F7.3`	`-15.234`

8.5.3. Datos tipo `real` en forma exponencial

8.5.3.1. Formato estándar

Especificación

```
En.d
```

- `E` significa que la variable es de tipo `real`, y su representación de tipo exponencial normalizada.
- `n` es una constante entera positiva que indica el número total de espacios que ocupa la representación del valor de la variable, incluyendo
  - El signo "−", si lo hubiera
  - El punto "."
  - El signo de la exponencial (no es opcional)
  - Los dígitos de la exponencial (un máximo de tres)
  - La letra `E` de la exponencial, en el caso de que la exponencial tenga menos de tres dígitos significativos
- `d` es una constante entera no negativa que indica el número total de decimales que van a aparecer, tras redondear.

Este formato no es válido para exponentes con valor absoluto mayor que 999.

8.5.3.2. Formato estándar ampliado

Especificación

```
En.dEm
```

- `E` significa que la variable es de tipo `real`, y su representación de tipo exponencial normalizada.
- `n` es una constante entera positiva que indica el número total de espacios que ocupa la representación del valor de la variable, incluyendo
  - El signo "−", si lo hubiera
  - El punto "."
  - El signo de la exponencial (no es opcional)
  - Los dígitos de la exponencial
  - La letra `E` de la exponencial
- `d` es una constante entera no negativa que indica el número total de decimales que van a aparecer, tras redondear.

- `m` es una constante entera no negativa que indica el número total de dígitos que van a aparecer en la exponencial.

Tabla 8.3 Formato estándar y estándar ampliado

Valor interno	Formato	Valor externo	Valor interno	Formato	Valor externo
34.12e5	`E10.4`	`0.3412E+07`	`34.12d3`	`E8.4`	`*********`
34.12e5	`E9.4`	`.3412E+07`	`34.12d300`	`E10.4`	`0.3412+302`
34.12e5	`E8.4`	`*********`	`34.12d300`	`E9.4`	`.3412+302`
34.12e5	`E8.2`	`0.34E+07`	`34.12e5`	`E9.4E1`	`0.3412E+7`
34.12e5	`E12.4`	`␣␣0.3412E+07`	`34.12e5`	`E8.4E1`	`.3412E+7`
34.12d3	`E10.4`	`0.3412E+05`	`34.12e5`	`E12.4E2`	`␣␣0.3412E+07`
34.12d3	`E9.4`	`.3412E+05`	`34.12e5`	`E12.4E4`	`0.3412E+0007`

8.5.3.3. Formato científico

Especificación

```
ESn.d[Em]
```

Su uso es idéntico al del formato estándar o estándar ampliado, excepto que

$$1 \leq |\text{mantisa}| < 10$$

Tabla 8.4 Formato científico

Valor interno	Formato	Valor externo
6.421	`ES9.3`	`6.421E+00`
-0.5	`ES10.3`	`-5.000E-01`
0.00217	`ES9.3`	`2.170E-03`
4721.3	`ES9.3`	`4.721E+03`

8.5.4. Datos tipo `character`

Especificación

```
A[n]
```

- `A` significa que es una variable de tipo `character`.
- `n` es una constante entera positiva que indica el número total de espacios que ocupa la representación del valor de la variable.

Si no se especifica el valor de `n`, se supone que `n=m`, siendo `m` la longitud declarada de la variable.

- Si `n=m`, en la salida aparece el valor completo de la variable.
- Si `n<m`, en la salida aparecen solo los `n` primeros caracteres.
- Si `n>m`, en la salida aparecen `n` caracteres, siendo los `(m-n)` primeros espacios en blanco.

Tabla 8.5 Formato de datos de tipo `character`

Valor interno	**Formato**	**Valor externo**
'HOLA'	A4	HOLA
'HOLA'	A3	HOL
'HOLA'	A6	␣␣HOLA
'␣HOLA␣'	A6	␣HOLA␣
'␣HOLA␣'	A5	␣HOLA
'␣HOLA␣'	A4	␣HOL
'␣HOLA␣'	A8	␣␣␣HOLA␣

8.5.5. Datos tipo `logical`

Especificación

```
L[n]
```

- `L` significa que es una variable tipo `logical`.
- `n` es un entero positivo que indica el número total de espacios que ocupa la representación del valor de la variable. En cualquier caso, la salida será o bien una `T` si el valor de la variable es `.true.` o bien una `F` si el valor de la variable es `.false.`

Tabla 8.6 Formato de datos de tipo `logical`

Valor interno	Formato	Valor externo
`.true.`	`L1`	`T`
`.true.`	`L5`	`␣␣␣␣T`

8.6. Especificación de un formato

En las sentencias `print`, `read` y `write` hemos visto que existe un especificador para dar el formato de las variables que se van a leer o escribir. La forma general es,

Especificación

```
'(form1,form2,...)'
```

siendo `form1,form2,...` los formatos de representación de las variables junto con posibles formatos especiales, separados por comas.

Entre los formatos especiales destacamos los siguientes

- `nX` deja `n` espacios en blanco.
- `/` salta una línea.

Si una cadena de especificaciones de formatos se va a repetir `n` veces en un mismo registro, se puede escribir de la forma

```
'(form1,form2,n(cadena_formatos),...)'
```

Es muy importante tener en cuenta que `n` debe ser una constante literal entera positiva, nunca el identificador de un dato. Sin embargo `n` puede ser mayor que el número de datos que va a leer o escribir.

Otra forma de especificar el formato utiliza una variable de tipo `character` que almacena la lista de formatos.

Vamos a ver un ejemplo. Tenemos un fichero de texto (`d_in.dat`) con los siguientes datos,

```
 3      4
 1.5   0.023   -1.2   24.5
1E-3   4.33    15.     0.
 32.    .24   -100.   10.0
```

Queremos leer este fichero y copiarlo en otro (`d_out.dat`) de forma que todas las variables reales tengan el mismo formato.

```
program fichero
    implicit none
    integer        :: n, m
    integer        :: i
    integer        :: unitID
    real, allocatable :: A(:,:)

    unitID = 15
    open(unit=unitID, file='d_in.dat', status='old', action='read')

    read(unitID,*) n, m

    allocate(A(n,m))
    do i=1,n
        read(unitID,*) A(i,:)
    end do
    close(unitID)

    open(unit=unitID, file='d_out.dat', status='unknown', action='write')

    write(unitID,'(1x,I1,3x,I1)') n, m
    do i=1,n
        write(unitID,'(100(3x,F8.3))') A(i,:)
    end do
    close(unitID)

    deallocate(A)
end program fichero
```

El fichero `d_out.dat` sería

```
␣3␣␣␣4
␣␣␣␣␣␣1.500␣␣␣␣␣␣␣0.023␣␣␣␣␣␣-1.200␣␣␣␣␣␣24.500
␣␣␣␣␣␣␣0.001␣␣␣␣␣␣␣4.330␣␣␣␣␣␣15.000␣␣␣␣␣␣␣0.000
␣␣␣␣␣␣32.000␣␣␣␣␣␣␣0.240␣␣␣␣-100.000␣␣␣␣␣␣10.000
```

8.7. Sentencia `namelist`

Un `namelist` se utiliza para leer o escribir un conjunto de valores de variables de cualquier tipo, refiriéndolas por un único nombre.

- Los `namelist` se escriben en un fichero independiente. No es necesario que se escriban todos en el mismo fichero, puede haber uno para cada `namelist`.
- La estructura de un `namelist` dentro del fichero es la siguiente

```
&nombre_del_namelist
    nombre_variable1 = valor_variable1
    nombre_variable2 = valor_variable2
      .
    nombre_variableN = valor_variableN /
```

  - Los caracteres inicial (`&`) y final (`/`) son obligatorios.
  - No puede haber ningún espacio en blanco entre el carácter inicial y el nombre del `namelist`.
  - Todas las variables contenidas en el `namelist` deben tener un valor asignado dentro del mismo.

- La declaración de un `namelist` dentro de la unidad de programa que lo utiliza se realiza de la forma

```
namelist/nombre_del_namelist/ nombre_variable1 , &
                              nombre_variable2 , &
                                .
                              nombre_variableN
```

  - Es obligatorio declarar el `namelist` dentro de la unidad de programa que lo utiliza.
  - Antes de declarar el `namelist`, es necesario declarar todas la variables del mismo, identificadas con el mismo nombre, dentro de la unidad de programa que lo utiliza.
  - La declaración, tanto de las variables del `namelist` como del propio `namelist`, se realiza dentro del cuerpo de declaraciones de la unidad de programa que lo utiliza.

- Para leer los valores de las variables del `namelist` dentro de la unidad de programa que lo utiliza, la secuencia es la siguiente

```
open(unit=u, file=nombre)
read(u, nml=nombre_del_namelist)
close(u)
```

  - *nombre* es una constante literal o un dato de tipo `character` cuyo valor es el nombre del fichero que contiene el `namelist`.
  - El especificador `nml` es opcional, no así su argumento *nombre_del_namelist*

- A lo largo de la ejecución del programa es posible que alguna de las variables definidas en el `namelist` cambie su valor. Para actualizar los valores de las variables en el propio fichero `namelist` la secuencia es la siguiente

```
open(unit=u, file=nombre)
write(u, nml=nombre_del_namelist)
close(u)
```

  - *nombre* es una constante literal o un dato de tipo `character` cuyo valor es el nombre del fichero que contiene el `namelist`.
  - El especificador `nml` es opcional, no así su argumento *nombre_del_namelist*

  Dado un fichero llamado `datos.dat`

```
&Tiempo
  horas = 12
  minutos = 30
  segundos = 15
  am_pm = 'PM' /
```

  Queremos usar las variables de este fichero en un programa

Figura 8.2 Ejemplo de fichero namelist

```
program main
    integer :: horas
    integer :: minutos
    integer :: segundos
    character(len=2) :: am_pm
    namelist/Tiempo/ horas, minutos, segundos, am_pm
        .
    open(15, file='datos.dat')
    read(15, Tiempo)
        .
    write(15, Tiempo)
        .
    close(15)

end program main
```

9 Programación modular

Índice

9.1. Introducción

La programación modular es una técnica de programación que consiste en dividir un programa en partes bien diferenciadas, llamadas *unidades de programa*, que pueden ser analizadas y programadas por separado. Se distinguen tres tipos de unidades de programa: *programa principal*, *función* y *subrutina*. A estos dos últimos también se les llama de forma genérica *subprogramas*.

Una unidad de programa se puede definir como un conjunto de instrucciones lógicamente enlazadas. A cada unidad de programa se le asigna un nombre, elegido por el programador, para poder identificarlo. Cuando en un punto de una unidad de programa se llama a ejecutar un subprograma, dicha unidad le cede el control para que se ejecuten todas sus instrucciones. Finalizado el mismo, el control se devuelve al punto de la unidad de programa llamadora, y se continúa con la ejecución de la instrucción siguiente a la que realizó la llamada. El orden de ejecución se muestra en las figuras 9.1 y 9.2.

Si un subprograma es lo suficientemente grande se puede subdividir en otros subprogramas, y éstos a su vez en otros subprogramas, y así sucesivamente, para simplificar la visualización del problema completo.

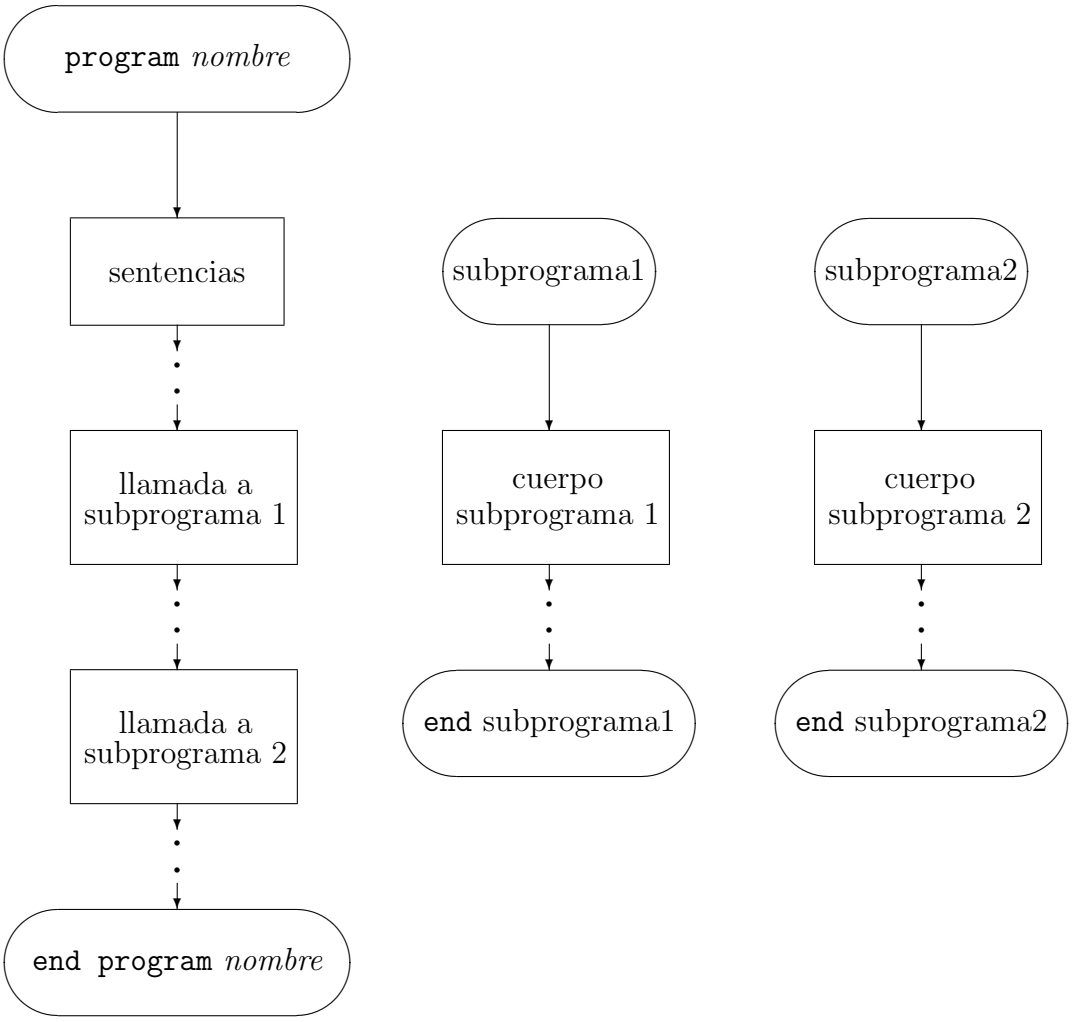
No hay una norma fija para dividir un problema en unidades de programa, sin embargo, se deben seguir unas pautas más o menos generalizadas.

- Solo puede existir una unidad de programa principal, llamado programa principal, que es el encargado de controlar y relacionar a todos los demás. Esta unidad de programa debe indicar la solución completa del problema.
- Cada unidad de programa solo puede tener un punto de entrada y otro de salida.
- Las unidades de programa deben tener la máxima independencia entre ellas.
- Una unidad de programa debe representar por sí misma una estructura lógica coherente y resolver una parte bien definida del problema.

La programación modular es así una técnica que se basa en el desarrollo de programas de lo general a lo particular (diseño descendente). Se comienza considerando qué funciones debe realizar el programa desde el punto de vista general dándolas como resueltas, dejando su diseño para un paso posterior. De esta manera se avanza hasta llegar al máximo nivel de detalle.

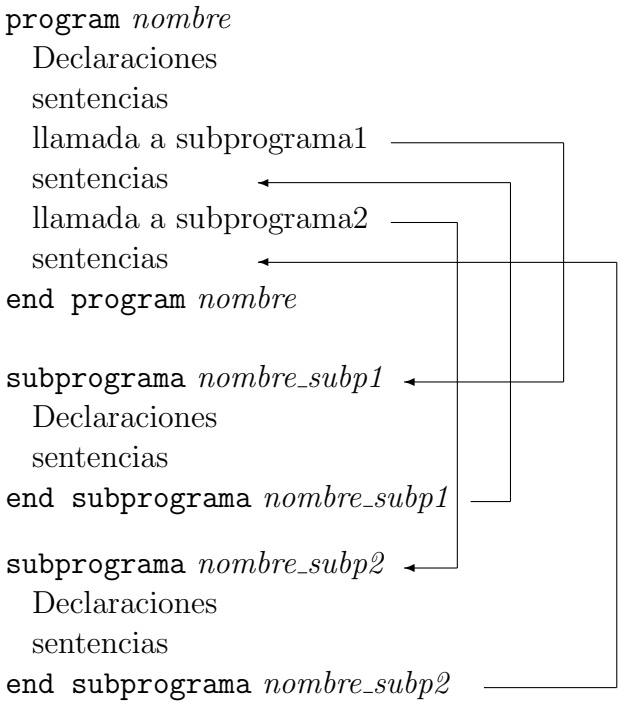
Las ventajas que aportan la programación modular frente a la convencional son, entre otras

Figura 9.1 Diagrama de flujo de la programación modular



- Los programas son más sencillos de escribir y depurar, pues se pueden hacer pruebas parciales con cada una de sus unidades de programa.
- La corrección de una unidad de programa se hace más cómoda y, en general, no tiene por qué afectar al resto de las unidades de programa.
- Un programa se puede ampliar fácilmente con solo diseñar las nuevas unidades de programa necesarias.
- Un subprograma puede ser llamado varias veces desde la misma unidad de programa, evitando la repetición de instrucciones ya escritas.
- Un mismo subprograma puede ser llamado desde otras unidades de programa, e incluso desde diferentes programas principales.

Figura 9.2 Pseudocódigo de la programación modular



Una vez que hemos introducido el concepto de subprograma, la siguiente cuestión es cómo la unidad de programa llamadora conoce la existencia del subprograma, lo que es necesario antes de su invocación. Llegados a este punto se debe introducir la siguiente clasificación de los subprogramas

- *Subprogramas externos*

  Son aquellos cuya definición se encuentra fuera del cuerpo de cualquier otra unidad de programa, tal y como se muestra en la figura 9.2

- *Subprogramas internos*

  Son aquellos cuya definición se encuentra dentro de una unidad de programa

- *Subprogramas module*

  Son aquellos cuya definición se encuentra dentro de un unidad *module*

Por cuestiones de claridad en la exposición, mientras no se indique lo contrario, por ahora vamos a considerar todos los subprogramas como subprogramas externos.

9.2. Funciones

Una función es un subprograma que, indicándole desde la unidad de programa llamadora los datos de entrada (*argumentos*) con los que se desea que realice los cálculos, devuelve un único resultado.

Para entender este concepto pensemos en el caso más sencillo: las funciones intrínsecas con un único argumento. Por ejemplo, en el programa

Figura 9.3 Programa que transfiere el signo de `y` a `x`

```
program main
    real :: x
    real :: y

    write(*,*) 'Introduzca dos reales'
    read(*,*) x, y

    if (y >= 0) then
        x = abs(x)
    else
        x = -abs(x)
    end if

    write(*,*) x
end program main
```

estamos llamando a la función `abs` con la variable `x` del programa principal de la que queremos conocer su valor absoluto. Una vez ejecutada la función, ésta devuelve un único resultado al programa, el valor absoluto del dato de entrada, que en este ejemplo se utiliza para asignarlo a la variable `x`.

Imaginemos ahora que no existe la función intrínseca `abs` y, dada la utilidad de la misma, queremos crearla nosotros. Para ello programamos una función, que trataremos como subprograma externo, y escribimos el siguiente código (*definición* de la función)

Como el programa principal, única unidad de programa conocida hasta este capítulo, la estructura general de un subprograma función consta de (Fig. 9.4)

Figura 9.4 Ejemplo de subprograma `function`

```
1 function valor_abs(x)
2     real, intent(in) :: x
3     real             :: valor_abs
4
5     valor_abs = x
6     if (valor_abs < 0.0) valor_abs = -valor_abs
7
8 end function valor_abs
```

- Un punto de entrada, también conocido como *cabecera* del subprograma. Es la primera línea de código (línea 1 de la figura 9.4), en la que se indica, por el siguiente orden que:

  1. El subprograma es una función. Para ello Fortran dispone de la palabra clave `function`.
  2. La función se llama `valor_abs`, nombre que no puede coincidir con ninguna palabra clave de Fortran ni con el nombre de ninguna función intrínseca de este lenguaje.
  3. `x` es el argumento de la función, denominado de forma genérica *argumento ficticio*.

- Un cuerpo de declaración (líneas 2 y 3 de la figura 9.4), donde se declara el dato de salida o resultado de la función `valor_abs` y el argumento ficticio de entrada a la misma. Se debe resaltar que

  1. El resultado de la función es una variable del subprograma `function` cuyo identificador siempre coincide con el nombre del subprograma, en este caso `valor_abs`.
  2. El argumento `x` está declarado con el atributo `intent(in)` que indica que es un dato de entrada y, por tanto, su valor no se puede variar a lo largo de la ejecución del subprograma.

- Las sentencias de ejecución, entre las que siempre tiene que aparecer la sentencia que define el resultado de la función.

- Un punto de salida (línea 8 de la figura 9.4), en la que se indica que ha terminado la ejecución del programa y se ordena que el control pase a la unidad de programa llamadora.

Para llamar a ejecutar el subprograma `function` que hemos definido, escribimos en el programa de la figura 9.3 las mismas sentencias solo que sustituyendo `abs(x)` por nuestra función `valor_abs(x)` (Fig. 9.5).

Figura 9.5 Programa que transfiere el signo de `y` a `x`

```
program main
    interface
        function valor_abs(x)
            real, intent(in) :: x
            real             :: valor_abs
        end function valor_abs
    end interface
    real :: x
    real :: y

    write(*,*) 'Introduzca dos reales'
    read(*,*) x, y
    if (y >= 0) then
        x = valor_abs(x)
    else
        x = -valor_abs(x)
    end if
    write(*,*) x
end program main
```

Como se trata de un subprograma externo es obligatorio indicarle esta condición a la unidad de programa llamadora. Esta es la única diferencia entre la función intrínseca `abs` que como tal no es necesario declarar, y una función externa codificada por el programador como `valor_abs`. La declaración de un subprograma externo la debe conocer toda unidad de programa que lo llame a ejecutar y, en principio, se hará en dicha unidad de programa dentro del cuerpo de declaraciones de la misma.

La declaración de un subprograma externo se realiza dentro de la construcción `interface` conforme al esquema

```
interface
    cuerpo_de_interface
end interface
```

donde el cuerpo de la interface se especifica por este orden:

1. La cabecera del subprograma.
2. Declaración de los argumentos y del tipo de resultado por tratarse de una función.
3. La sentencia `end function` *nombre_función*

La construcción `interface` es fundamental en el caso de subprogramas externos. Se puede visualizar como un esquema del subprograma donde aparece la información más relevante del mismo: cuáles son los datos de entrada y salida, y de qué tipo son. Esta información es suficiente para que el procesador, durante la compilación, haga una comprobación de la concordancia entre los argumentos verdaderos y ficticios en cada una de las llamadas al subprograma.

Una vez expuestos los pasos básicos que se deben seguir para definir e invocar un subprograma externo de tipo `function`, vamos a modificar nuestro programa de modo que la transferencia de signo entre las variables `x` e `y` (*argumentos verdaderos*) se realice en un subprograma `function` llamado `transf_signo` con dos argumentos ficticios, `a` y `b`. Los correspondientes códigos se muestran en las figuras 9.6 y 9.7.

Figura 9.6 Función de transferencia de signo

```
function transf_signo(a, b)
    real, intent(in) :: a
    real, intent(in) :: b
    real             :: transf_signo

    if (b >= 0) then
        transf_signo = abs(a)
    else
        transf_signo = -abs(a)
    end if
end function transf_signo
```

Figura 9.7 Programa que transfiere el signo de `y` a `x`

```
program main
    interface
        function transf_signo(a, b)
            real, intent(in) :: a
            real, intent(in) :: b
            real             :: transf_signo
        end function transf_signo
    end interface

    real :: x
    real :: y

    write(*,*) 'Introduzca dos reales'
    read(*,*) x, y

    x = transf_signo(x, y)
    write(*,*) x
end program main
```

9.3. Subrutinas

Una subrutina es un subprograma que, indicándole desde la unidad de programa llamadora los datos de entrada con los que se desea que realice los cálculos, proporciona un conjunto de datos de salida a dicha unidad de programa. Tanto los datos de entrada como los de salida constituyen los argumentos ficticios de una subrutina.

En la figura 9.8 se muestra la definición de una subrutina que, dado un ángulo en radianes (argumento de entrada: `theta`), calcula los correspondientes grados, minutos y segundos (argumentos de salida: `deg`, `min`, `sec`).

La estructura general de un subprograma subrutina consta de

- Un punto de entrada o *cabecera* del subprograma. Es la primera línea de código (línea 1 de la figura 9.8), en la que se indica, por el siguiente orden que:

    1. El subprograma es una subrutina. Para ello Fortran dispone de la palabra clave `subroutine`.

Figura 9.8 Subrutina de conversión

```
1  subroutine conversion(theta, deg, min, sec)
2      real, intent(in)  :: theta
3      real, intent(out) :: deg
4      real, intent(out) :: min
5      real, intent(out) :: sec
6
7      real :: pi
8      real :: resto
9
10     pi = acos(-1.0)
11
12     deg = (theta*180.0)/pi
13     resto = deg - aint(deg)
14     deg = aint(deg)
15
16     min = resto*60.0
17     resto = min - aint(min)
18     min = aint(min)
19
20     sec = resto*60.0
21     resto = sec - aint(sec)
22     sec = aint(sec)
23
24 end subroutine conversion
```

2. La subrutina se llama `conversion`, nombre que no puede coincidir con ninguna palabra clave de Fortran ni con el nombre de ninguna función intrínseca de este lenguaje.
3. `theta`, `deg`, `min` y `sec` son los argumentos ficticios del subprograma.

- Un cuerpo de declaración (líneas 2 a 8 de la figura 9.8). En las líneas 2, 3, 4 y 5 se declaran los argumentos ficticios de la subrutina. Se debe resaltar que

1. El argumento `theta` está declarado con el atributo `intent(in)` ya explicado anteriormente que indica que es un dato de entrada al subprograma.
2. Los argumentos `deg`, `min` y `sec` están declarados con el atributo `intent(out)` que indica que son datos de salida del subprograma y, por tanto, no pueden ser utilizados en el subprograma hasta que no se les asigne un valor dentro del mismo.

En las líneas 7 y 8 se declaran las *variables locales* del subprograma. Como su nombre indica se trata de datos que solo conoce este subprograma `subroutine` y que son necesarios para la realización de los cálculos que en él se definen. Su funcionamiento dentro del subprograma es el propio del de cualquier dato de su tipo que hemos estudiado.

- Las sentencias de ejecución, entre las que siempre tienen que aparecer la sentencias que definen los argumentos ficticios de salida declarados con el atributo `intent(out)`.
- Un punto de salida (línea 24 de la figura 9.8), en la que se indica que ha terminado la ejecución del programa y se ordena que el control pase a la unidad de programa llamadora.

Para llamar a ejecutar el subprograma `subroutine` que hemos definido, escribimos el programa de la figura 9.9, teniendo en cuenta que como se trata de un subprograma externo es obligatorio indicarle esta condición a la unidad de programa llamadora. Recordemos que la declaración de un subprograma externo se realiza dentro de la construcción `interface` indicando en el cuerpo de misma, por este orden:

1. La cabecera del subprograma `subroutine` tal y como aparece en su definición.
2. Declaración de los argumentos ficticios tanto de entrada como de salida.
3. La sentencia `end subroutine` *nombre_subrutina*.

La llamada al subprograma `subroutine` (Fig.9.9) se realiza mediante la sentencia `call`, cuya expresión general es

`call` *nombre_subrutina*`(`*lista_argumentos*`)`

siendo *lista_argumentos* los argumentos verdaderos o datos de la unidad de programa llamadora que se van a asociar con los correspondientes argumentos ficticios.

Figura 9.9 Programa para la conversión de ángulos

```
program main
    interface
        subroutine conversion(theta, deg, min, sec)
            real, intent(in)  :: theta
            real, intent(out) :: deg
            real, intent(out) :: min
            real, intent(out) :: sec
        end subroutine conversion
    end interface

    real :: angle
    real :: deg
    real :: min
    real :: sec

    write(*,*) 'Introduzca angulo en radianes'
    read(*,*) angle

    call conversion(angle, deg, min, sec)
    write(*,*) 'El angulo introducido es:'
    write(*,*) deg, 'grados', min, 'minutos', sec, 'segundos'
end program main
```

9.4. Tratamiento de argumentos

Una de las formas, y única vista hasta el momento, de intercambiar información entre una unidad de programa llamadora y un subprograma es a través del paso de argumentos por cabecera. En lenguaje Fortran la llamada a un subprograma es una *llamada por referencia*, es decir, los argumentos ficticios no ocupan nuevas posiciones de memoria sino que el nombre de un argumento ficticio referencia en el momento de la llamada a la posición de memoria de su correspondiente argumento verdadero. Por esta razón es necesario que el programador tenga el control sobre el tratamiento que desea para cada argumento, lo que se consigue a través del atributo `intent`.

Por otro lado Fortran permite variar el número y orden de los argumentos verdaderos en el momento de la llamada, dependiendo de los requerimientos de la unidad de programa

llamadora. Mención especial merecen los argumentos de tipo *array*, los de tipo derivado y el uso de subprogramas como argumentos. En esta sección nos dedicaremos a abordar en detalle estas cuestiones.

9.4.1. El atributo `intent`

Es recomendable no modificar el valor de los argumentos ficticios de entrada dentro de un subprograma, ya que Fortran trabaja con la posición de memoria ocupada por la variable, y esta posición es común para un argumento verdadero y su correspondiente argumento ficticio. Por esta razón una modificación en el valor de un argumento ficticio queda automáticamente reflejada en el valor de su correspondiente argumento verdadero. Sin embargo, la situación habitual en un código es que en ocasiones se requiera que algunos de los argumentos de entrada modifiquen su valor y otros no.

Para evitar la posible confusión se usa el atributo `intent` que permite, en tiempo de compilación, especificar si un argumento es de entrada o de salida o si puede ser modificado o no. El atributo `intent` tiene tres opciones:

- `intent(in)`

  Los argumentos declarados con este atributo no pueden modificar su valor dentro del subprograma. Si en algún momento apareciera una sentencia de asignación que intentara modificar su valor, se produciría un error en tiempo de compilación.

- `intent(out)`

  Los argumentos declarados con este atributo no pueden ser utilizados dentro del subprograma hasta que no se les asigne un valor dentro del mismo, aunque sus correspondientes argumentos verdaderos estuviesen definidos en la unidad de programa llamadora. Si esto ocurriese, daría un error en tiempo de compilación.

- `intent(inout)`

  Los argumentos declarados con este atributo tienen un valor a la entrada al subprograma que puede ser modificado a lo largo del mismo, cambio que se refleja en sus correspondientes argumentos verdaderos.

El único argumento que no se puede declarar con el atributo `intent` es el resultado de un subprograma `function` pues es por sí un dato de salida, y la declaración sería redundante.

En el ejemplo de la figura 9.10 se muestran dos errores típicos por mal uso del atributo `intent`. El primer error ocurre al intentar modificar el valor del dato `a` que ha sido declarado como argumento solo de entrada. El segundo error se produce al intervenir el dato `c` en una operación sin que esté previamente definido, lo que se debía haber hecho por estar declarado con el atributo `intent(out)`. El argumento `b` es de entrada y salida luego su correspondiente argumento verdadero modifica el valor que tenía antes de la llamada, por el valor resultante de los cálculos realizados en el subprograma.

Figura 9.10 Ejemplo del atributo `intent`

```
subroutine ejemplo(a, b, c)
    real, intent(in)    :: a
    real, intent(inout) :: b
    real, intent(out)   :: c

    a = a + 2.0           ! Error de compilacion
    b = b + c             ! Error de compilacion
    c = a + b
    b = b/2.0
end subroutine ejemplo
```

9.4.2. Asociación de argumentos

La regla general que rige la asociación entre argumentos verdaderos y argumentos ficticios impone que ambos deben coincidir en tipo y *kind*, número y orden. Fortran permite alterar tanto el orden en que se disponen los argumentos verdaderos al efectuar la llamada a un subprograma, como el número de argumentos verdaderos, pudiendo ser inferior al número de argumentos ficticios.

Alteración del orden

Para alterar el orden de un argumento verdadero es necesario especificar en el momento de la llamada el nombre de su correspondiente argumento ficticio. Además todos los argumentos verdaderos situados a su derecha en la llamada, deben llevar asociado el nombre de un argumento ficticio. Por ejemplo varias formas de llamar al subprograma de la figura 9.10 pueden ser

```
call ejemplo(x, y, z)              ! Correcta
call ejemplo(a=x, b=y, c=z)        ! Correcta
call ejemplo(x, c=z, b=y)          ! Correcta
call ejemplo(b=y, a=x, z)          ! Incorrecta
```

Alteración del número

Si un argumento ficticio lleva el atributo `optional`, lo estamos declarando argumento opcional en el sentido de que es posible invocar el subprograma asignándole un argumento verdadero o dejándolo libre. A un argumento ficticio declarado con el atributo `optional` se le puede asignar un valor por defecto dentro del subprograma, con lo que queda definido aunque no se le asocie ningún argumento verdadero en el momento de la llamada. Tan pronto como un argumento opcional ha sido omitido de la lista de argumentos verdaderos, todos los restantes a su derecha deben llevar asociado el nombre de su correspondiente argumento ficticio en el momento de la llamada al subprograma.

El estado de un argumento opcional dentro del subprograma se puede conocer con la función lógica intrínseca `present`, que devuelve el valor `.true.` si está asociado su argumento con un argumento verdadero durante la ejecución del subprograma, y `.false.` en caso contrario.

En la figura 9.12 se muestra un programa que realiza diferentes llamadas a un subprograma `subroutine` con dos argumentos opcionales `a` y `b` (Fig.9.11), cuya asociación con sus correspondientes argumentos verdaderos se comprueba con la función `present`. En el programa principal se observa que en el momento que se omite un argumento verdadero en la sentencia de llamada, los restantes a su derecha deben asociarse con el argumento ficticio que les corresponda.

Figura 9.11 Ejemplo del atributo `optional`

```
subroutine suma(a, b, total)
    real, intent(in), optional :: a
    real, intent(in), optional :: b
    real, intent(out)          :: total

    total = 0.0                   ! Valor por defecto
    if (present(a)) then
        total = a
        if (present(b)) then
            total = total + b
        end if
    else if (present(b)) then
        total = b
    end if
end subroutine suma
```

Figura 9.12 Ejemplo de llamada a subprograma con argumentos opcionales

```
program main
    interface
        subroutine suma(a, b, total)
            real, intent(in), optional :: a
            real, intent(in), optional :: b
            real, intent(out)          :: total
        end subroutine suma
    end interface
    real :: a = 2.0
    real :: b = 5.0
    real :: total

    call suma(a, b, total)            ! total = 7.0
    call suma(a, total=total)         ! total = 2.0
    call suma(b=b, total=total)       ! total = 5.0
    call suma(total=total)            ! total = 0.0
end program main
```

9.4.3. Argumentos tipo `character`

Un caso particular de argumento de subprograma lo constituyen los datos de tipo `character`. Una forma muy restrictiva de declarar un argumento ficticio de tipo `character` es hacerlo con una longitud fija, de forma que el argumento verdadero debe tener la misma longitud, en el mejor de los casos, o una longitud mayor, en cuyo caso el argumento ficticio trunca el valor del argumento verdadero.

Una forma sencilla de evitar este problema consiste en declarar el argumento ficticio de tipo `character` con *longitud asumida* utilizando un asterisco como longitud,

```
character(len=*), intent(in)    :: nombre_variable
character(len=*), intent(out)   :: nombre_variable
character(len=*), intent(inout) :: nombre_variable
```

en cuyo caso, el argumento ficticio asume la longitud del argumento verdadero.

9.4.4. Argumentos tipo *array*

El siguiente paso consiste en utilizar *arrays* como datos de entrada y salida de un subprograma. Lo estudiado hasta ahora referente a los atributos de los argumentos ficticios se mantiene, independientemente de si éstos son datos simples o *arrays*.

Hay dos formas de utilizar los datos tipo *array* como argumentos ficticios

Con forma explícita

Se especifican todos los límites del *array* en su declaración. Los argumentos verdadero y ficticio basta con que sean del mismo tipo y *kind*.

El mecanismo de asociación es el siguiente. El argumento verdadero indica únicamente la posición de memoria del primer dato que se va a transferir. El tamaño del argumento ficticio indica el número máximo de datos que se van a recoger. La forma y los límites de las dimensiones del argumento ficticio indican cómo se va a organizar el *array* en el subprograma. La posición del primer dato transferible se especifica a través del correspondiente elemento del argumento verdadero, salvo si se trata del primer elemento que también se puede especificar con el nombre del *array*.

El hecho de que los datos transferidos tengan que ocupar posiciones contiguas de memoria invalida las secciones de un *array* como posibles argumentos verdaderos.

Figura 9.13 Argumento con forma explícita. Subrutina

```
subroutine explicita(Ma, U)
    real, intent(in) :: Ma(2,2)
    real, intent(in) :: U(5)
        .
end subroutine explicita
```

Figura 9.14 Argumento con forma explícita. Programa principal

```
1  program main
2      real :: A(2,2)
3      real :: B(2,3)
4      real :: z(5)
5      real :: t(6)
6          .
7      A = reshape((/11, 12, 13, 14/), (/2,2/))
8      B = reshape((/21, 22, 23, 24, 25, 26/), (/2,3/))
9      z = (/41, 42, 43, 44, 45/)
10     t = (/51, 52, 53, 54, 55, 56/)
11
12     call explicita(Ma=A, U=z)
13     call explicita(Ma=B, U=t)
14     call explicita(Ma=B(2,1), U=t(2))
15     call explicita(Ma=t, U=B)
16     call explicita(Ma=B(2,2), U=t(4))
17
18 end program main
```

En el ejemplo representado por las figuras 9.13 y 9.14, el programa principal define los vectores y matrices

$$z = \begin{pmatrix} 41, & 42, & 43, & 44, & 45 \end{pmatrix}$$

$$t = \begin{pmatrix} 51, & 52, & 53, & 54, & 55, & 56 \end{pmatrix}$$

$$A = \begin{pmatrix} 11 & 13 \\ 12 & 14 \end{pmatrix} \qquad B = \begin{pmatrix} 21 & 23 & 25 \\ 22 & 24 & 26 \end{pmatrix}$$

mientras que la subrutina siempre recoge una matriz `Ma(2,2)` y un vector `U(5)`.

- En la llamada de la línea 12

$$Ma = \begin{pmatrix} 11 & 13 \\ 12 & 14 \end{pmatrix} \qquad U = \begin{pmatrix} 41, & 42, & 43, & 44, & 45 \end{pmatrix}$$

- En la llamada de la línea 13

$$Ma = \begin{pmatrix} 21 & 23 \\ 22 & 24 \end{pmatrix} \qquad U = \begin{pmatrix} 51, & 52, & 53, & 54, & 55 \end{pmatrix}$$

- En la llamada de la línea 14

$$Ma = \begin{pmatrix} 22 & 24 \\ 23 & 25 \end{pmatrix} \qquad U = \begin{pmatrix} 52, & 53, & 54, & 55, & 56 \end{pmatrix}$$

- En la llamada de la línea 15

$$Ma = \begin{pmatrix} 51 & 53 \\ 52 & 54 \end{pmatrix} \qquad U = \begin{pmatrix} 21, & 22, & 23, & 24, & 25 \end{pmatrix}$$

- En la llamada de la línea 16

$$Ma = \begin{pmatrix} 24 & 26 \\ 25 & -500 \end{pmatrix} \qquad U = \begin{pmatrix} 54, & 55, & 56, & 79, & 85 \end{pmatrix}$$

Un método más flexible de trabajar con argumentos con forma explícita consiste en introducir las dimensiones como argumentos de entrada al subprograma, de esta forma el argumento ficticio se adapta fácilmente a la forma del argumento verdadero (Figs. 9.15 y 9.16)

Figura 9.15 Argumento con forma explícita. Subrutina

```
subroutine explicita(n, m, Ma, U, dimU)
    integer, intent(in) :: n
    integer, intent(in) :: m
    integer, intent(in) :: dimU
    real, intent(in)    :: Ma(n,m)
    real, intent(in)    :: U(dimU)
        .
end subroutine explicita
```

Figura 9.16 Argumento con forma explícita. Programa principal

```
program main
    integer, parameter :: n = 2
    integer, parameter :: m = 3
    real :: A(n,n)
    real :: B(n,m)
    real :: z(5)
    real :: t(6)
       .
    A = reshape((/11, 12, 13, 14/), (/2,2/))
    B = reshape((/21, 22, 23, 24, 25, 26/), (/2,3/))
    z = (/41, 42, 43, 44, 45/)
    t = (/51, 52, 53, 54, 55, 56/)

    call explicita(n=n, m=n, Ma=A, U=z, dimU=5)
    call explicita(n=n, m=m, Ma=B, U=t, dimU=6)
    call explicita(n=n, m=n Ma=B(2,2), U=t(2), dimU=5)
    call explicita(n=n, m=m, Ma=t, U=B, dimU=n*m)

end program main
```

Con forma asumida

No se especifica ninguna extensión, solo el rango. Los argumentos verdadero y ficticio deben coincidir en el rango, el tipo y el *kind*.

En este caso, lo que se transfiere es un *array* completo, no un conjunto ordenado de datos. El argumento ficticio asume la forma del argumento verdadero pero no necesariamente los límites de las dimensiones. El único tratamiento especial se refiere al hecho de que no se puede utilizar el atributo `intent(out)` o `intent(inout)` en un argumento ficticio cuando el verdadero es una sección generada con un vector de índices.

Figura 9.17 Argumento con forma asumida. Subrutina

```
subroutine asumida(M, U)
    real, intent(in) :: M(:,:)
    real, intent(in) :: U(:)
        .
end subroutine asumida
```

Figura 9.18 Argumento con forma asumida. Programa principal

```
1  program main
2      real :: A(2,2)
3      real :: B(2,3)
4      real :: z(5)
5      real :: t(6)
6          .
7      A = reshape((/11, 12, 13, 14/), (/2,2/))
8      B = reshape((/21, 22, 23, 24, 25, 26/), (/2,3/))
9      z = (/41, 42, 43, 44, 45/)
10     t = (/51, 52, 53, 54, 55, 56/)
11
12     call asumida(M=A, U=z)
13     call asumida(M=B, U=t)
14     call asumida(M=B(1:2,2:3), U=t(2:4))
15     call asumida(M=B(1:2,2:2), U=t((/1,3/)))
16
17 end program main
```

En el ejemplo representado por las figuras 9.17 y 9.18, el programa principal define los vectores y matrices

$$z = \begin{pmatrix} 41, & 42, & 43, & 44, & 45 \end{pmatrix} \qquad A = \begin{pmatrix} 11 & 13 \\ 12 & 14 \end{pmatrix} \qquad B = \begin{pmatrix} 21 & 23 & 25 \\ 22 & 24 & 26 \end{pmatrix}$$

$$t = \begin{pmatrix} 51, & 52, & 53, & 54, & 55, & 56 \end{pmatrix}$$

mientras que la subrutina recoge,

- En la llamada de la línea 12, una matriz `M(2,2)` y un vector `U(5)`

$$M = \begin{pmatrix} 11 & 13 \\ 12 & 14 \end{pmatrix} \qquad U = \begin{pmatrix} 41, & 42, & 43, & 44, & 45 \end{pmatrix}$$

- En la llamada de la línea 13, una matriz `M(2,3)` y un vector `U(6)`

$$M = \begin{pmatrix} 21 & 23 & 25 \\ 22 & 24 & 26 \end{pmatrix} \qquad U = \begin{pmatrix} 51, & 52, & 53, & 54, & 55, & 56 \end{pmatrix}$$

- En la llamada de la línea 14, una matriz `M(2,2)` y un vector `U(3)`

$$M = \begin{pmatrix} 23 & 25 \\ 24 & 26 \end{pmatrix} \qquad U = \begin{pmatrix} 52, & 53, & 54 \end{pmatrix}$$

- En la llamada de la línea 15, una matriz `M(2,1)` y un vector `U(2)`

$$M = \begin{pmatrix} 23 \\ 24 \end{pmatrix} \qquad U = \begin{pmatrix} 51, & 53 \end{pmatrix}$$

9.4.5. Argumentos tipo subprograma

La filosofía de un subprograma es resolver un problema genérico. El programa llamador especifica cuáles son los datos de entrada del problema concreto a resolver, mientras que el subprograma contiene el algoritmo que, operando esos datos, genera la solución.

Es fácil imaginar situaciones en las que uno de los argumentos es de por sí un subprograma. Por ejemplo, si buscamos en la literatura un algoritmo para integrar una función real, $F(x)$, en un intervalo $[a, b]$, encontraremos varios métodos: Regla del trapecio, Método de Simpson, etc. La programación modular establece que el subprograma `subroutine trapecio` sea capaz de resolver de forma completa el problema. De forma natural, los argumentos de entrada al problema serán: `F`, `a` y `b`; mientras que el argumento de salida será: `integral`, siendo

$$Integral = \int_a^b F(x)\,dx$$

Veamos con un ejemplo los pasos básicos que hay que dar para invocar a un subprograma que cuenta entre sus argumentos ficticios con un subprograma (Figs. 9.19—9.21)

Al igual que los argumentos ficticios `a`, `b` e `integral`, el argumento ficticio `F` debe ser declarado de forma apropiada, es decir, con la construcción `interface` ya que se trata de un subprograma externo para la subrutina `trapecio`.

Figura 9.19 Fichero con las funciones a integrar

```
function F(x)
    real, intent(in) :: x
    real             :: F
        F = x*x + 3.0
end function F

function G(x)
    real, intent(in) :: x
    real             :: G
        G = x*x - cos(x)
end function G
```

Figura 9.20 Fichero con la subrutina de integración

```
subroutine trapecio(F, a, b, integral)
    interface
        function F(x)
            real, intent(in) :: x
            real             :: F
        end function F
    end interface
    real, intent(in)  :: a
    real, intent(in)  :: b
    real, intent(out) :: integral
        .
        .
    integral =
        .
end subroutine trapecio
```

En este ejemplo los argumentos verdaderos que hemos utilizado en la llamada a la subrutina `trapecio` son subprogramas `function` externos declarados con la construcción `interface` en el programa principal (líneas 2–13). La propia subrutina `trapecio`, como subprograma externo, aparece también en una `interface` (líneas 14–26).

Figura 9.21 Fichero con el programa principal de integración

```
1  program main
2      interface
3          function F(x)
4              real, intent(in) :: x
5              real             :: F
6          end function F
7      end interface
8      interface
9          function G(x)
10             real, intent(in) :: x
11             real             :: G
12         end function G
13     end interface
14     interface
15         subroutine trapecio(F, a, b, integral)
16             interface
17                 function F(x)
18                     real, intent(in) :: x
19                     real             :: F
20                 end function F
21             end interface
22             real, intent(in)  :: a
23             real, intent(in)  :: b
24             real, intent(out) :: integral
25         end subroutine trapecio
26     end interface
27
28     real :: IntegralF
29     real :: IntegralG
30     real :: a
31     real :: b
32         .
33     a = 0.0
34     b = 5.0
35     call trapecio(F, a, b, IntegralF)
36         .
37     b = pi
38     call trapecio(G, a, b, IntegralG)
39 end program main
```

Siguiendo las reglas ya establecidas de asociación de argumentos, es en las sentencias de llamada (líneas 35 y 38) donde se especifica el problema concreto que se va a resolver a través de los argumentos verdaderos `F` y `G`.

Igual que un argumento verdadero y su correspondiente ficticio deben coincidir en tipo y *kind*, cuando se trata de subprogramas la concordancia debe ser total (tipo y *kind* de subprograma, número de argumentos, tipo y *kind* de argumentos), excepto el propio nombre del subprograma.

Por último hay que señalar que los únicos subprogramas válidos como argumentos verdaderos son los subprogramas externos y los subprogramas *module*.

9.5. *Array* como resultado de una función

Ya hemos definido un subprograma `function` como aquel objeto que devuelve un único resultado. Este resultado puede ser un dato simple o compuesto. En esta sección nos ocuparemos solo del caso en el que la función devuelva un *array* de datos de tipo intrínseco. Matemáticamente nos referimos, por ejemplo, a $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$.

Las reglas de uso de la función como *array*, son las mismas que las de cualquier otro *array* (acceso a un elemento en particular, operaciones válidas, manejo de secciones, etc). Sin embargo, como argumento del subprograma `function` tiene ciertas limitaciones ya que no puede ser un *array* de forma asumida. Los límites de sus dimensiones deben estar perfectamente definidos en la declaración del *array*. En la figura 9.22 se muestra dos formas típicas de definir una función cuyo resultado es un *array* y en la figura 9.23 un ejemplo de programa principal que las utiliza.

Figura 9.22 Ejemplos de funciones *arrays*

```
function F(n, x)
    integer, intent(in) :: n
    real, intent(in)    :: x
    real, dimension(n)  :: F
    integer :: i

    do i=1,n
        F(i) = x**i
    end do

end function F

function Forz(x)
    real, intent(in)          :: x(:)
    real, dimension(size(x)) :: Forz
    integer :: i

    do i=1,size(x)
        Forz(i) = x(i)*x(i)
    end do

end function Forz

function Normaliza(A)
    real, intent(in)                          :: A(:,:)
    real, dimension(size(A,1), size(A,2)) :: Normaliza
    real :: maximo

    maximo = maxval(A)
    Normaliza = 0.0
    if (maximo /= 0.0) Normaliza = A/maximo

end function Normaliza
```

Figura 9.23 Programa principal para el uso de funciones *arrays*

```
program main
    interface
        function F(n, x)
            integer, intent(in) :: n
            real, intent(in)    :: x
            real, dimension(n)  :: F
        end function F
    end interface
    interface
        function Forz(x)
           real, intent(in)          :: x(:)
           real, dimension(size(x)) :: Forz
        end function Forz
    end interface
    interface
        function Normaliza(A)
            real, intent(in)                          :: A(:,:)
            real, dimension(size(A,1), size(A,2)) :: Normaliza
        end function Normaliza
    end interface
    integer, parameter :: n=2
    real :: x
    real :: U(n+n)
    real :: A(n,n)
    real :: AN(n,n)

    x = 5.0
    U = F(n, x)
    U = Forz(U)
    A = reshape(U, (/2,2/))
    AN = Normaliza(A)
end program main
```

9.6. Variables locales

Las variables locales de un subprograma son las variables, distintas de los argumentos ficticios, que utiliza el subprograma en su desarrollo. Como regla general, las variables locales se crean la primera vez que se invoca al subprograma. Además

- Deben tener un valor asignado antes de ser utilizadas.

- No tienen acceso a ellas ni el (sub)programa llamador ni ningún otro subprograma, por tanto puede haber variables locales que tengan el mismo nombre que variables de cualquier otro (sub)programa, sin que haya ninguna relación entre ellas.
- Una variable local de un subprograma no mantiene su valor de una llamada a otra excepto si
  - Está declarada con el atributo `save`
  - Está inicializada en la línea de declaración.

Figura 9.24 Atributo `save`. Subrutina y programa principal

```
subroutine llamadas
    integer, save :: NLLamadas = 0
    integer       :: Nl

    NumeroLLamadas = NumeroLLamadas + 1
    Nl = 0
    Nl = Nl + 1
    write(*,'(2(a,I2,2x))') 'NLLamadas = ', NLLamadas, 'Nl = ', Nl
end subroutine llamadas
program main
    interface
        subroutine llamadas
        end subroutine llamadas
    end interface
    integer :: i

    do i=1,5
        call llamadas
    end do
end program main
```

En el ejemplo de la figura 9.24, al ejecutar el programa principal, el resultado sería

```
NLLamadas =  1  Nl =  1
NLLamadas =  2  Nl =  1
NLLamadas =  3  Nl =  1
```

9.7. Tipos de subprogramas

Aunque ya se han definido los tipos de subprogramas reconocidos por Fortran, ahora que se tiene un conocimiento más claro de para qué sirve y cómo funciona un subprograma, vamos a ampliar los conceptos con una visión más detallada de cada uno de los tipos de subprograma. Más adelante desarrollaremos los aspectos más importantes de los subprogramas internos y *module*. Sin embargo, en este punto conviene tener una idea global del modo de expresar los subprogramas y cómo interactúan con la unidad de programa llamadora. Esta unidad llamadora puede ser un programa principal u otro subprograma.

Por cuestiones de claridad, en los siguientes esquemas supondremos que la unidad llamadora es el programa principal.

Subprograma externo

Un subprograma, `function` o `subroutine`, externo se caracteriza por las dos propiedades:

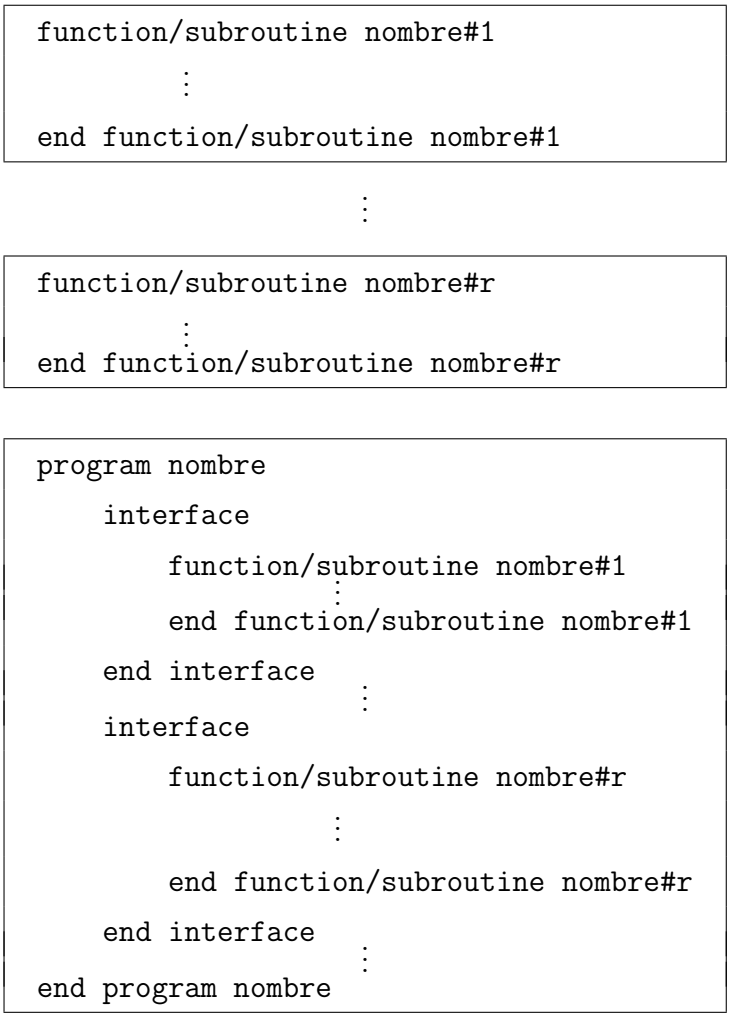
1. Se encuentra fuera del esquema `program` — `end program`
2. Se accede a él a través de una construcción `interface`

Las figura 9.25 muestra un esquema de los puntos descritos anteriormente. Varios subprogramas externos y un programa principal que los utiliza.

El uso habitual de este tipo de esquemas está limitado al caso sencillo en el que un programa principal necesita a lo largo de su ejecución tan solo uno o dos subprogramas que, bien porque son muy largos o bien porque se han extraído de alguna librería, se mantienen en ficheros independientes. Esto obliga a declarados explícitamente con una `interface`. Evidentemente, tener un programa con, exagerando un poco, 200 líneas de `interface`, no es precisamente lo que se entiende por un programa claro y compacto.

Tampoco es muy habitual ver programas con varias `interfaces` anidadas. De nuevo, la razón es la claridad en el estilo de programación.

Figura 9.25 Programa principal que utiliza subprogramas externos



Subprograma interno

Un subprograma, `function` o `subroutine`, interno se caracteriza por las tres propiedades:

1. Se encuentra dentro del esquema `program` — `end program`
2. Se encuentra justo antes de la sentencia `end program`
3. Se encuentra justo después de la sentencia `contains`

Las figura 9.26 muestra un esquema de los puntos descritos anteriormente. Varios subprogramas internos contenidos en un programa principal.

El uso habitual de este tipo de esquemas está limitado al caso, también sencillo en el que un programa principal necesita a lo largo de su ejecución unos pocos subprogramas,

todos ellos cortos en extensión y que probablemente estén relacionados entre sí, en el sentido de que algunos de ellos contengan llamadas a otros subprogramas del mismo `contains`.

Figura 9.26 Programa principal con subprogramas internos

```
program nombre
        ⋮
contains
    function/subroutine nombre#1
        ⋮
    end function/subroutine nombre#1
                ⋮
    function/subroutine nombre#r
        ⋮
    end function/subroutine nombre#r
end program nombre
```

Subprograma *module*

Un subprograma *module*, `function` o `subroutine`, se caracteriza por las cuatro propiedades:

1. Se encuentra dentro del esquema `module` — `end module`
2. Se encuentra justo antes de la sentencia `end module`
3. Se encuentra justo después de la sentencia `contains`
4. Se accede a él con la sentencia `use`

Las figura 9.27 muestra un esquema de los puntos descritos anteriormente. Varios subprogramas contenidos en un `module` y un programa principal que los utiliza.

El uso habitual de este tipo de esquemas se observa en los grandes programas que, para resolver un problema importante deben resolver antes varios problemas intermedios. La idea de este paradigma de programación consiste en manejar varios `modules`, cada uno de ellos asociado a un problema completo en cuestión.

Figura 9.27 Programa principal que utiliza subprogramas module

```
module nombre_module
        ⋮
contains
    function/subroutine nombre#1
        ⋮
    end function/subroutine nombre#1
                ⋮
    function/subroutine nombre#r
        ⋮
    end function/subroutine nombre#r
end module nombre_module
```

```
program nombre_program
    use nombre_module
        ⋮
end program nombre_program
```

Un primer vistazo a los esquemas presentados revela las diferencias entre un subprograma externo y otro de tipo interno o *module*.

En primer lugar, la forma de "declarar" la existencia del subprograma al programa llamador. Si el subprograma es externo, necesitamos la construcción `interface`, que actúa como una línea más de declaración. Igual que para declarar un dato de tipo `integer` usamos el esquema,

```
integer :: nombre_variable
```

para declarar una `function` externa de tipo `real` de una variable `real` utilizamos el esquema

```
interface
    function nombre_funcion(arg)
        real, intent(in) :: arg
        real             :: nombre_funcion
    end function nombre_funcion
end interface
```

Sin embargo, si el subprograma es interno, el hecho de estar físicamente en el mismo fichero que el programa llamador y dentro de un `contains`, basta para que el llamador conozca su existencia. Además, durante la compilación el procesador comprueba la concordancia en los argumentos de las distintas llamadas.

Algo parecido ocurre con un subprograma *module*, la sentencia `use` es suficiente para "declarar" todos los subprogramas contenidos en él, lo que facilita la comprobación de argumentos durante la compilación.

9.8. Subprograma interno

La característica principal de un subprograma interno es que físicamente su definición está contenida dentro de un programa principal o de un subprograma externo, siempre detrás de una sentencia `contains`. Para desarrollar los conceptos relativos al alcance de las variables vamos a ampliar el esquema representado en la figura 9.26 para introducir algunas variables en el programa principal y en los subprogramas.

Hay que hacer constar que vamos a utilizar un ejemplo concreto con un programa principal y dos funciones (Fig. 9.28). Este ejemplo se puede generalizar a un subprograma externo (`function` o `subroutine`) que contiene varios subprogramas `function` o `subroutine`

En la figura 9.28 debemos distinguir los siguientes tipos de variables, según su alcance

Variables locales

Son todas aquellas declaradas en los subprogramas internos que no son argumentos ficticios. Una variable local solo es accesible para el propio subprograma interno que lo contiene. Las variables locales de la figura 9.28 son:

- Para la función `f`, las variables `i` (línea 18), `z` (línea 19) y `t` (línea 20)
- Para la función `g`, las variables `i` (línea 29) y `r` (línea 30)

Variables globales

Son todas aquellas declaradas en el programa principal. Una variable global es accesible, además de para el propio programa principal, para cualquier subprograma interno a él siempre que no contenga una variable local con el mismo nombre. Las variables globales de la figura 9.28 son: `a, b, i, z, zf, zg` (líneas 2 – 7)

Figura 9.28 Programa principal con 2 funciones internas

```
1  program main
2      real    :: a
3      real    :: b
4      integer :: i
5      real    :: z
6      real    :: zf
7      real    :: zg
8       .
9      zf = f(a)
10      .
11     zg = g(a, b)
12      .
13 contains
14     function f(x)
15         real, intent(in)  :: x
16         real              :: f
17
18         integer :: i
19         real    :: z
20         real    :: t
21          .
22     end function f
23
24     function g(x, y)
25         real, intent(in)  :: x
26         real, intent(in)  :: y
27         real              :: g
28
29         integer :: i
30         real    :: r
31          .
32     end function g
33 end program main
```

9.9. Subprograma module

La característica principal de un subprograma *module* es que físicamente su definición está contenida dentro de una unidad `module`, siempre detrás de una sentencia `contains`. Para desarrollar los conceptos relativos al alcance de las variables vamos a ampliar el esquema representado en la figura 9.27 para introducir algunas variables en el `module` y en los subprogramas.

Hay que hacer constar que vamos a utilizar un ejemplo concreto con tres subrutinas (Fig. 9.29). En ella debemos distinguir los siguientes tipos de variables, según su alcance

Variables locales

Son todas aquellas variables declaradas en los subprogramas que no son argumentos ficticios. Una variable local solo es accesible para el propio subprograma que lo contiene.

Variables globales

Son todas aquellas variables declaradas antes de la sentencia `contains`, considerada como *zona de declaración del module*. Una variable global es accesible para cualquier subprograma contenido en el `module` siempre que él mismo no contenga una variable local con el mismo nombre.

La manera de acceder a todos o solo a algunos de los subprogramas contenidos en un subprograma `module` es a través de la sentencia `use`.

En su forma más general, la sentencia

```
use nombre_del_module
```

dentro de un programa principal o cualquier subprograma interno o externo, da acceso a todos los subprogramas contenidos en el `module` llamado *nombre_del_module*, a todos los datos declarados en la zona de declaración del module y a todos los tipos derivados definidos en la zona de declaración del module, siempre que no lleven el atributo `private`.

Por ejemplo, el programa de la figura 9.30 puede utilizar las subrutinas `inversa` y `determinante`, el parámetro `dimA` y el tipo derivado `sistema`. Pero, no tiene acceso a la variable `Xt`.

Si queremos restringir el acceso a únicamente un conjunto de subprogramas y datos, la sentencia sería

```
use nombre_del_module, only : lista_de_objetos
```

que, dentro de un programa principal o cualquier programa interno o externo, da acceso a todos los objetos contenidos en el `module` llamado *`nombre_del_module`* especificados en la *`lista_de_objetos`*.

Por ejemplo, el programa de la figura 9.31 puede utilizar la subrutina `inversa` y el parámetro `dimA`. Pero, no tiene acceso a la subrutina `determinante`, al tipo derivado `sistema` ni a la variable `Xt`.

El ejemplo de los métodos de integración (figuras 9.32–9.34) pone de manifiesto la comodidad del manejo de los subprograma `module`. En este caso estamos usando los `module` como "cajones" para agrupar los subprogramas según su funcionalidad dentro del problema a resolver. Así, hay un módulo que contiene todos los métodos de integración implementados (Fig. 9.32) y otro módulo que contiene todas las funciones que hay que integrar (Fig. 9.33). El programa principal (Fig. 9.34) accede a los contenidos de cada uno de los módulos a través de la sentencia `use`.

Es importante observar que tanto la subrutina `trapecio` como la subrutina `Simpson` mantienen la construcción `interface` – `end interface` para declarar el argumento ficticio `F`, como no podría ser de otra manera, ya que ésta es la única forma de "declarar" un subprograma como argumento.

Figura 9.29 `module` con 3 subrutinas

```
module CalculoMatrices
    type sistema
        real :: A(10,10)
        real :: b(10)
        real :: x(10)
    end type sistema

    integer, parameter :: dimA = 5
    real, private      :: Xt

contains

subroutine inversa(A, Ai)
    real, intent(in)  :: A(:,:)
    real, intent(out) :: Ai(:,:)
    real :: determ
     .
    call determinante(A=A, det=determ)
     .
end subroutine inversa

subroutine determinante(A, det)
    real, intent(in)  :: A(:,:)
    real, intent(out) :: det
     .
end subroutine determinante

subroutine resuelve(A, b, x)
    real, intent(in)  :: A(:,:)
    real, intent(in)  :: b(:)
    real, intent(out) :: x(:)
    type(sistema)     :: Sistem10
     .
end subroutine resuelve
end module CalculoMatrices
```

Figura 9.30

```
program main
    use CalculoMatrices
    integer, parameter :: n = 2
    real    :: A(n,n)
    real    :: InvA(n,n)
    real    :: detA
    real, allocatable :: mA(:,:)
    type(sistema)     :: Sist20
     .
    call inversa(A=A, Ai=InvA)
     .
    call determinante(A=A, det=detA)
     .
    allocate(mA(dimA,dimA))
     .
end program main
```

Figura 9.31

```
program main
    use CalculoMatrices, only : inversa, dimA
    real    :: A(dimA, dimA)
    real    :: InvA(dimA, dimA)
     .
    call inversa(A=A, Ai=InvA)
     .
end program main
```

Figura 9.32 Fichero con las subrutinas de los métodos de integración

```
module metodos

    implicit none

contains

subroutine trapecio(F, a, b, integral)
    interface
        function F(x)
            real, intent(in) :: x
            real             :: F
        end function F
    end interface
    real, intent(in)  :: a
    real, intent(in)  :: b
    real, intent(out) :: integral
        .
        .
    integral =
        .
end subroutine trapecio

subroutine Simpson(F, a, b, integral)
    interface
        function F(x)
            real, intent(in) :: x
            real             :: F
        end function F
    end interface
    real, intent(in)  :: a
    real, intent(in)  :: b
    real, intent(out) :: integral
        .
        .
    integral =
        .
end subroutine Simpson
end module metodos
```

Figura 9.33 Fichero con las funciones a integrar

```
module funciones

    implicit none

contains
    function F(x)
        real, intent(in) :: x
        real             :: F
            F = x*x + 3.0
    end function F

    function G(x)
        real, intent(in) :: x
        real             :: G
            G = x*x - cos(x)
    end function G
```

Figura 9.34 Fichero con el programa principal de integración

```
program main

    use funciones
    use metodos

    implict none

    real :: IntegralF_t, IntegralF_s
    real :: IntegralG_t, IntegralG_s
    real :: a
    real :: b
        .
    a = 0.0
    b = 5.0
    call trapecio(F, a, b, IntegralF_t)
    call Simpson(F, a, b, IntegralF_s)
        .
    b = pi
    call trapecio(G, a, b, IntegralG_t)
    call Simpson(F, a, b, IntegralG_s)

end program main
```

A El conjunto de caracteres ASCII

y \ x	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	‘	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	”	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	’	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
10	lf	sub	∗	:	J	Z	j	z
11	vt	esc	+	;	K	[	k	{
12	ff	fs	,	<	L	\	l	[
13	cr	gs	–	=	M	]	m	{
14	so	rs	.	>	N	↑	n	∼
15	si	us	/	?	O	_	o	del

El número de orden de un carácter, *ch*, se calcula a partir de sus coordenadas en la tabla mediante la fórmula

$$ord(ch) = 16 * x + y$$

Los caracteres con número de orden de 0 a 31 y número 127, se denominan *caracteres de control* y se utilizan para transmisión de datos y control de dispositivos. El carácter número 32 es el blanco.

B Bibliografía

[**1**] Adams J.C., Brainerd W.S., Martin J.T., Smith B.T., Wagener J.N. *Fortran 90 Handbook. Complete ANSI/ISO Reference*. McGraw-Hill, 1992

[**2**] Ellis T.M.R., Philips I.R., Lahey T.M. *Fortran 90 programing*. Addison-Wesley, 1994

[**3**] Metcalf M. *Fortran 90/95 explained*. Oxford University Press, 1996

[**4**] Metcalf M., Reid J., Cohen M. *Fortran 95/2003 explained*. Oxford University Press, 2004

[**5**] Nyhoff L.R., Leestma S.F. *Fortran 90 for engineers & scientists*. Prentice Hall, 1997