



UNIVERSIDAD NACIONAL
AUTONOMA DE MÉXICO

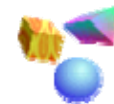
FACULTAD DE INGENIERIA

PROGRAMACIÓN AVANZADA

ING. MARTÍN CARLOS
VELÁZQUEZ

CURSO FORTRAN 90-95

Sección 1: [Elementos básicos del lenguaje](#)



Sección 2: [Organización del programa](#)



Sección 3: [Arrays y punteros](#)



Sección 4: [Entrada/salida de datos](#)





1.1.- Capítulo 1: Forma fuente y declaraciones

- 1.1.1 [Forma fuente - características principales](#)
- 1.1.2 [Forma de un programa en Fortran90](#)
- 1.1.3 [Entrada interactiva de datos: PRINT y READ](#)
- 1.1.4 [Funciones intrínsecas](#)

1.2.- Capítulo 2: Expresiones y asignaciones

- 1.2.1 [Expresiones - operaciones intrínsecas](#)
- 1.2.2 [Evaluación de expresiones](#)
- 1.2.3 [Asignaciones](#)

1.3.- Capítulo 3: Estructuras de control

- 1.3.1 [Estructuras de control: IF](#)
- 1.3.2 [Estructuras de control: CASE](#)
- 1.3.3 [Estructuras de control: DO](#)
- 1.3.4 [Sentencias de control I](#)
- 1.3.5 [Sentencia de control II](#)

1.4.- Capítulo 4: Operadores

- 1.4.1 [Tipos intrínsecos de datos](#)
- 1.4.2 [Tipos de datos](#)
- 1.4.3 [Operadores](#)

1.5.- Capítulo 5: Procedimientos intrínsecos

- 1.5.1 [Procedimientos intrínsecos elementales](#)
- 1.5.2 [Procedimientos numéricos](#)
- 1.5.3 [Procedimientos matemáticos y lógicos](#)
- 1.5.4 [Procedimientos caracter](#)
- 1.5.5 [Funciones para vectores y matrices](#)



Forma fuente - características principales

En el nuevo estándar del Fortran, el Fortran90, se considera una nueva forma de código fuente, la *forma fuente libre*, en ella no existen restricciones sobre las posiciones que deben ocupar las sentencias Fortran, y sus características principales son:

1. Los caracteres en blanco son siempre significativos exceptuando casos en los que se introducen para hacer más legible el programa. Como por ejemplo:

```
s=s+a(i)
s = s + a (i)
```

2. que es la misma sentencia. Además dos o más espacios en blanco consecutivos siempre equivalen a uno.
3. Cada línea no puede tener más de 132 caracteres.
4. El símbolo de exclamación (!) indica al compilador que todo lo que aparece a continuación en la misma línea es un comentario y el compilador lo ignora. En un programa no está limitado el número de comentarios. Además una línea cuyo primer carácter no blanco sea !, se conoce con el nombre de línea de comentario.
5. El símbolo & es un símbolo de continuación y permite al programador continuar una sentencia en la siguiente línea. Una sentencia no puede abarcar mas de 39 líneas. Si el primer caracter no blanco en la línea de continuación es un & los espacios en blanco son ignorados, esto permite escribir una cadena de caracteres en más de una línea. Ejemplo:

```
Print *, '~&
           Quiero escribir una cadena de caracteres &
           & muy larga. ^
```

6. Las líneas de comentario no pueden estar divididas en varias líneas debido a que el compilador consideraría el símbolo & como un carácter más y no como el signo de continuación, puesto que está dentro del comentario (indicado por el signo de admiración).
7. Una línea sin caracteres o solo con caracteres en blanco es considerada como una línea de comentario.
8. En una línea pueden aparecer más de una sentencia separadas por un punto y coma (;). Además, en ausencia del símbolo de continuación,&, el final de la línea lo marca el final de la sentencia.



Forma de un programa en Fortran90

Todo programa en Fortran90 deberá comenzar con la sentencia `program nombre` (Este nombre tiene que comenzar por una letra y puede tener hasta 31 caracteres, tanto letras, como números o el símbolo `_`). Este nombre solo tiene significado para el lector.

Posteriormente, al acabar el programa se usará la sentencia `end program nombre`.

El nombre del fichero fuente del programa deberá acabar en `.f90` (Esta condición no es propia del Fortran90, sino del sistema operativo). Ejemplo:

```
Program suma_de_numeros  
  
.....  
  
end program suma_de_numeros
```

Definición de variables

Las declaraciones de tipo aparecerán entre el principio del programa y el inicio de la parte ejecutable. Cada declaración consta de una palabra de tipo intrínseco específica del Fortran90 seguida de dos pares de dos puntos y una lista de nombres de variables separadas por comas. Ejemplo:

```
Integer::i,j ; real::x,z
```

En Fortran90 es necesario declarar todas las variables. También es posible declararlas con un parámetro de clase particular, para ello escribiremos después de la palabra específica del Fortran90, (`kind = palabra representativa`). Por ejemplo, si quiero que unas determinadas variables reales tengan el doble de números representativos escribiré:

```
Real( kind = 2 )::x,z
```

En el caso de variables carácter yo podré indicar la longitud, en número de caracteres, de la variable con un parámetro de clase, para ello haría:



```
Character( len = 15 ) :: nombre
```

También es posible dar a una variable un valor inicial en el momento de ser declarada, por ejemplo:

```
integer :: iteraciones = 0 , real :: x = 2.1 , y = 0.4
```

Así, el valor de esta variable podrá ser modificado a lo largo de la ejecución del programa. Por otro lado, si queremos darle un valor fijo a una variable usaríamos la sentencia `parameter`. Por ejemplo:

```
Real, parameter :: pi = 3.141592
```

Posteriormente, durante el programa usaremos el nombre del parámetro en vez de su correspondiente valor, y además este no podrá ser modificado.

Como ya está comentado, todas las variables que utilicemos deberán ser declaradas y para que no olvidemos ninguna existe la sentencia `implicit none`. Esta sentencia se colocará al principio de cada programa y toda variable no definida provocará un error de compilación.

Entrada interactiva de datos: **print** y **read**

Para poder trabajar interactivamente con un programa Fortran90 se introducen dos sentencias básicas de entrada y salida directa de datos al terminal.

La sentencia **print**

La sentencia `print` se usa de igual forma que en Fortran77 . Ejemplo:

```
Print * , ' La suma de 2 y 3 es : ', 2 + 3
```

El `*` informa al compilador que el programador no desea especificar el formato exacto en el que desea imprimir la respuesta. Al ejecutar este ejemplo saldrá por pantalla `La suma de 2 y 3 es : 5`. La sentencia entre apóstrofes o entre comillas se conoce con el nombre de cadena de caracteres y puede contener todo tipo de caracteres.

La sentencia **read**

Permite obtener el valor de las variables que usaremos. Ejemplo:



`Read*, x, y`

El `*` nos indica que el formato de entrada de datos es el que se usa por defecto.

Reglas para los nombres de las variables:

1. El primer caracter de un nombre tiene que ser una letra.
2. Los demás pueden ser una mezcla de números, letras o el símbolo `_`.
3. La longitud máxima de un nombre será de 31 caracteres.

Funciones intrínsecas

Las funciones intrínsecas son aquellas incluidas en el propio lenguaje de programación, y que no necesitan ser definidas por el programador para utilizarlas. Existen muchas *funciones intrínsecas* en Fortran90. Para usarlas solamente tenemos que escribir el nombre de la función seguido de los argumentos de ella encerrados entre paréntesis. El número de argumentos no tiene por que ser fijo, es decir, es posible usar la función `MAX`, por ejemplo, para calcular el máximo de dos o tres números.

Función `kind`

La función `kind(x)` devuelve el valor del parámetro de clase o tipo del argumento `x`. Por ejemplo `kind(0)` devuelve la clase por defecto de los enteros, `kind(0.0)` devuelve la clase por defecto de los reales, `kind(.false.)` devuelve la clase por defecto de las variables lógicas y `kind('A')` devuelve la clase por defecto de las variables carácter.

La función intrínseca `selected_real_kind` devuelve un valor cuya representación tiene una cierta precisión y un cierto rango. Por ejemplo `selected_real_kind(7,80)` devuelve reales con 7 cifras decimales y comprendidos entre -10^{80} y 10^{80} . Para los enteros la función es `selected_int_kind`, que funciona de forma análoga. Por ejemplo `selected_int_kind(9)` devuelve enteros comprendidos entre -10^9 y 10^9 .

Funciones de conversión de tipo numérico

Existen funciones que convierten una variable numérica en cada uno de los otros tipos de variables. Estas funciones son, `real`, `int` y `cmplx`. Por ejemplo, el valor de `int(2.4)` es el entero `2`, el valor de `real(2,3.1)` es el real `2` y el valor de `cmplx(3)` es el complejo `3 + 0i`.



Las funciones de conversión de tipo numérico son usadas para convertir variables de una clase en otra con el mismo parámetro de clase, si quiero que esto no sea así tendré que especificar el parámetro de clase. Por ejemplo, `real(i,kind=2)` convierte el valor entero `i` en un real en doble precisión.

Función `logical`

La función denominada `logical` cambia el parámetro de clase de una variable lógica. Por ejemplo, si `l` es de tipo lógico y `k` es entero, `logical(l,k)` da el valor de `l` representado como una variable lógica con parámetro de clase `k` y la función `logical(l)` nos da el valor de `l` representado como una variable lógica con el parámetro de clase por defecto.

Funciones intrínsecas

Las funciones intrínsecas son aquellas incluidas en el propio lenguaje de programación, y que no necesitan ser definidas por el programador para utilizarlas. Existen muchas *funciones intrínsecas* en Fortran90. Para usarlas solamente tenemos que escribir el nombre de la función seguido de los argumentos de ella encerrados entre paréntesis. El número de argumentos no tiene por que ser fijo, es decir, es posible usar la función `MAX`, por ejemplo, para calcular el máximo de dos o tres números.

Función `kind`

La función `kind(x)` devuelve el valor del parámetro de clase o tipo del argumento `x`. Por ejemplo `kind(0)` devuelve la clase por defecto de los enteros, `kind(0.0)` devuelve la clase por defecto de los reales, `kind(.false.)` devuelve la clase por defecto de las variables lógicas y `kind('A')` devuelve la clase por defecto de las variables carácter .

La función intrínseca `selected_real_kind` devuelve un valor cuya representación tiene una cierta precisión y un cierto rango. Por ejemplo `selected_real_kind(7,80)` devuelve reales con 7 cifras decimales y comprendidos entre -10^{**80} y 10^{**80} . Para los enteros la función es `selected_int_kind`, que funciona de forma análoga. Por ejemplo `selected_int_kind(9)` devuelve enteros comprendidos entre -10^{**9} y 10^{**9} .

Funciones de conversión de tipo numérico

Existen funciones que convierten una variable numérica en cada uno de los otros tipos de variables. Estas funciones son, `real`, `int` y `cmplx`. Por ejemplo, el



valor de `int(2.4)` es el entero **2**, el valor de `real(2,3.1)` es el real **2** y el valor de `cmplx(3)` es el complejo **3 + 0i**.

Las funciones de conversión de tipo numérico son usadas para convertir variables de una clase en otra con el mismo parámetro de clase, si quiero que esto no sea así tendré que especificar el parámetro de clase. Por ejemplo, `real(i,kind=2)` convierte el valor entero `i` en un real en doble precisión.

Función `logical`

La función denominada `logical` cambia el parámetro de clase de una variable lógica. Por ejemplo, si `l` es de tipo lógico y `k` es entero, `logical(l,k)` da el valor de `l` representado como una variable lógica con parámetro de clase `k` y la función `logical(l)` nos da el valor de `l` representado como una variable lógica con el parámetro de clase por defecto.

Expresiones - operaciones intrínsecas

Una *expresión* en Fortran90 es usada para indicar muchas clases de cálculos. Las componentes básicas de una expresión pueden ser combinadas usando unos operadores que serán explicados a continuación.

Operaciones intrínsecas

Son aquellas operaciones que están definidas y son conocidas por el compilador. A continuación veremos una tabla de *operadores intrínsecos* con su significado:

CATEGORIA DEL OPERADOR	OPERADOR INTRINSECO	SIGNIFICADO	TIPO DE OPERANDOS
Aritmética	**	Exponente	Numéricos
Aritmética	*	Multiplicación	Numéricos
Aritmética	/	División	Numéricos
Aritmética	+	Adición	Numéricos
Aritmética	-	Sustracción	Numéricos
Caracter	//	Concatenación de cadenas	Caracter con el mismo parámetro de clase.
Relación	.EQ.	Igual	Ambos numéricos o ambos caracteres.
Relación	.NE.	Distinto	Ambos numéricos o ambos caracteres.



Relación	>	Mayor	Ambos numéricos o ambos caracteres.
Relación	>=	Mayor o igual	Ambos numéricos o ambos caracteres.
Relación	<	Menor	Ambos numéricos o ambos caracteres.
Relación	<=	Menor o igual	Ambos numéricos o ambos caracteres.
Lógico	. NOT .	Negación	Ambos numéricos o ambos caracteres.
Lógico	. AND .	y	Lógicos
Lógico	. OR .	o	Lógicos
Lógico	. EQV .	Equivalente	Lógicos
Lógico	. NEQV .	No equivalente	Lógicos

Evaluación de expresiones

Cuando en una expresión concurre más de una operación, los paréntesis indicarán prioridad, es decir, la operación encerrada entre paréntesis se realizará en primer lugar. Además, algunos operadores tendrán preferencia sobre otros. Por ejemplo, en la operación $a + b / c$, primero se realizará b / c y posteriormente se le sumará a . En caso de que el programador quiera que se sume primero a y b para posteriormente dividir por c , tendríamos que hacer $(a + b) / c$. Si todos los operadores de una expresión tienen la misma prioridad, la operación se hará de izquierda a derecha, salvo cuando tengamos exponenciales, en tal caso, el orden será de derecha a izquierda, por ejemplo, al hacer $2^{**}3^{**}2$ resulta el valor $2^{**}9 = 512$.

En la siguiente tabla veremos las prioridades que existen entre los operadores:

Categoría del operador	Operador	Prioridad
Numérica	**	Mayor
Numérica	*, /	.
Numérica	+, -	.
Carácter	//	.
Relación	.eq. , .ne.	.
Relación	.lt. , .le. , .gt. , .ge.	.
Relación	== , /= , < , <= , > , >=	.



Lógica	.not.	.
Lógica	.and.	.
Lógica	.or.	.
Lógica	.eqv. or .neqv.	.
Extensión	Operador binario definido por el usuario	Menor

Hacer $a + b + c + d$ es lo mismo que hacer $(a + b) + (c + d)$ pero en el primer caso el ordenador sumaría a y b , el resultado se lo sumaría a c y este a d , sin embargo en el segundo caso haría $a + b$ y $c + d$ y sumaría ambos resultados.

Al realizar operaciones hay que tener cuidado con el tipo y la clase de las variables a operar, ya que por ejemplo, la división de dos enteros truncaría el resultado, es decir, $3 / 2$ no daría 1.5 sino 1 . El motivo de esto es que el resultado de dividir dos enteros es un entero. Por tanto, no sería lo mismo $i / 2$ que $0.5 * i$, siendo i un entero.

Asignaciones

En muchos casos el resultado obtenido al evaluar una expresión es asignado a una variable cuyo valor será usado posteriormente a lo largo del programa refiriéndose solamente a la variable. Esto se hace con una *sentencia de asignación*.

La forma general del proceso de asignación es:

```
Variable = Expresión
```

La ejecución de la sentencia de asignación provoca que la expresión sea evaluada y luego ese valor sea asignado a la variable que esta a la izquierda del signo igual (=). Este signo igual no es un igual algebraico, solo indica que la variable que esta a la izquierda toma el valor de la expresión de la derecha. Así, a lo largo del programa el valor de la expresión podrá ser reemplazado por el nombre de la variable.

Si tengo por ejemplo $Pi = 3.141592$ y $x = 2$ podré hacer $y = x * Pi$ y así obtener $y = 6.283184$.

En el caso de que la expresión sea escalar esta debería coincidir en tipo y clase con la variable, en caso de no ser así, deberemos utilizar una sentencia de conversión (`int`, `real`, `cmplx`, . . .) para corregirlo.

Si la expresión y la variable son de tipo carácter estas deben tener el mismo valor del parámetro de clase, en caso contrario, es decir, si tienen distintas longitudes, ocurre lo siguiente:



- Si la longitud de la variable es menor que la de la expresión, el valor de la expresión se trunca por la derecha para adecuarse a la longitud de la variable.
- En el caso de que la longitud de la variable sea mayor que la longitud de la expresión, el valor de la expresión se rellenará con espacios en blanco hasta completar la longitud de la variable.

Estructuras de control: IF

Una *estructura de control* consta de uno o más bloques de sentencias Fortran90 que se ejecutarán dependiendo de la veracidad de una sentencia lógica. Existen tres clases de estructuras de control, la estructura **IF**, la estructura **CASE** y la estructura **DO**. Una colección de sentencias cuya ejecución está controlada por una estructura de control se denomina un bloque. No está permitido pasar los valores de un bloque a otro, pero sí que se puede abandonar un bloque en el momento que se desee. Suele ser habitual indentar los bloques para una mayor legibilidad del programa. Las estructuras de control pueden tener un nombre que las identifique, este se situará en la primera sentencia del bloque. En caso de ponerle un nombre a la estructura la sentencia **END** tendrá que ir seguida del nombre de la estructura.

La estructura IF

La estructura **IF** es una estructura de decisión que permite la selección de uno u otro bloque de sentencias dependiendo de una sentencia determinada. La forma general de una estructura **IF** es:

```
IF (expresión lógica ) THEN
    Bloque de sentencias
ELSE IF ( expresión lógica ) THEN
    Bloque de sentencias
ELSE IF ( expresión lógica ) THEN
    Bloque de sentencias
```



```
      . . .  
  
ELSE  
  
    Bloque de sentencias  
  
END IF
```

La sentencia `ELSE IF`, al igual que la sentencia `ELSE`, son opcionales y pueden ser omitidas; sin embargo la sentencia `END IF` es obligatoria.

La sentencia `IF-THEN` será ejecutada siempre y cuando la expresión lógica sea cierta. Si la expresión lógica es falsa la ejecución saltará a la siguiente sentencia, tanto sea un `ELSE IF`, un `ELSE` o un `END IF`.

La ejecución de un bloque `ELSE IF` se hará si se verifica su expresión lógica. Posteriormente, en caso de que ninguna expresión fuese cierta se ejecutará la sentencia `ELSE` y después se acabará la estructura con la sentencia `END IF`. La sentencia `ELSE` solo se ejecutará si todas las expresiones lógicas fuesen falsas.

NOTA: Las expresiones lógicas de las sentencias `IF` y `ELSE IF` no tienen que ser excluyentes, es decir, dentro de una estructura `if` pueden ejecutarse varios bloques de sentencias.

La sentencia `IF`

Es la misma sentencia que en Fortran77 y su forma es:

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

Así, si la expresión lógica es cierta se ejecutará la sentencia y si es falsa no se hará.

Estructuras de control: **CASE**

Esta estructura es una nueva característica del Fortran90. Es bastante parecida a la estructura `IF`, ya que permite la selección de diferentes bloques de instrucciones. La selección se basa en los distintos valores de una expresión escalar; la forma general de la estructura `CASE` es:

```
SELECT CASE ( expresión escalar )  
  
    CASE ( valor o lista de valores )
```



```
        Bloque de sentencias
CASE ( valor o lista de valores )
        Bloque de sentencias
        . . .
CASE DEFAULT
        Bloque de sentencias
END SELECT
```

El valor de la expresión de `SELECT CASE` tiene que ser un entero, un carácter o una variable lógica. Además, el valor o lista de valores del `CASE` tiene que ser del mismo tipo que la expresión. La sentencia `CASE DEFAULT` es opcional. El bloque `CASE` será ejecutado siempre y cuando el valor de la expresión del `SELECT CASE` coincida con el valor de este bloque. La ejecución de las sentencias de los bloques `CASE` es excluyente, es decir, si se realiza un bloque de sentencias ya no se ejecutará ningún otro. En el caso de que ningún valor de las sentencias `CASE` sea válido se ejecutará el bloque de sentencias del `CASE DEFAULT`. La lista de valores de un `CASE` puede ser los valores comprendidos entre dos números o entre dos cadenas de caracteres, en tal caso se escribirá `2 : 5`, para indicar los valores entre el `2` y el `5`, ambos inclusive.

Estructuras de control: DO

La estructura `DO` consta de unas determinadas sentencias que son repetidas bajo el control de una parte de la estructura. La forma general de una estructura `DO` es:

```
DO ( sentencia de control )
        Bloque de sentencias
END DO
```

El bloque de sentencias será repetido tantas veces como indique la sentencia de control. Existen tres tipos de sentencias de control:

1. Que no haya sentencia de control; así el bucle se ejecutará hasta que alguna instrucción, dentro del bloque de sentencias, provoque la parada. Esta instrucción será la sentencia `EXIT`.



2. Que la sentencia de control tome una progresión de valores hasta alcanzar un valor predeterminado.
3. Que la sentencia de control sea ejecutada mientras una expresión lógica sea cierta.

La sentencia `EXIT` provoca el fin de la ejecución de un bucle, mientras que la sentencia `CYCLE` provoca el final de una iteración del bucle. Pasamos ahora a detallar los tres tipos de sentencias de control ya mencionados.

Sentencias de control I

Bucles sin sentencia de control

Para una estructura `DO` sin sentencia de control, el bloque de sentencias entre la sentencia `DO` y la sentencia `END DO` se ejecutará repetidamente hasta que la sentencia `EXIT` provoque la parada.

Bucles con sentencia de variable de control

Habitualmente, los sucesivos valores que toma una variable están relacionados por un valor, como por ejemplo, **1, 2, 3, 4, 5** ó **8, 6, 4, 2**. Como esto ocurre a menudo, hay una forma muy simple de asignar a una variable dichos valores, y es con una sentencia `DO` con una variable de control.

La forma general de un bucle `DO` con variable de control es:

```
DO variable = expresión, expresión, expresión
    Bloque de sentencias
END DO
```

La primera expresión especifica el valor inicial de la variable, la segunda determina el último valor de la variable, mientras que la tercera nos da la frecuencia o el paso de incremento o disminución de la variable. Si se omite la expresión del paso se entiende que este es uno. La variable puede tomar valores enteros o reales, aunque debido a posibles errores de cálculo, sólo deben utilizarse valores enteros. Ejemplo:

```
DO iteraciones = 0, 10, 2
    Print * , ' Estoy en la iteración ', iteraciones
END DO
```

Esta estructura imprimiría los números `0, 2, 4, 6, 8` y `10`.



También es posible poner un paso negativo, por ejemplo:

```
DO i = 10, 0, -2
```

Así los sucesivos valores que iría tomando la variable *i* serían 10, 8, 6, 4, 2 y 0.

NOTA: No es recomendable numerar los bucles, de igual forma que hacíamos en Fortan77 para distinguirlos, sino que para distinguirlos podremos denominarlos con un nombre, por ejemplo:

```
número de iteraciones: DO i = 1,10
                        Print * , i
                        END DO número_de_iteraciones
```

donde *número de iteraciones* es el nombre de la estructura DO.

Sentencia de control II

Bucles con sentencia de control lógica

Son bucles que se ejecutarán mientras una expresión lógica sea cierta. Su forma general es:

```
DO
    IF ( expresión lógica no es cierta ) EXIT
    Bloque de sentencias
END DO
```

Tipos intrínsecos de datos

Existen cinco tipos de datos en Fortran90, *enteros*, *reales*, *complejos*, *lógicos* y *caracteres*.

Cada uno tiene unos valores permitidos y hay algunas sentencias que permiten cambiarlos. Por ejemplo, `'12 + 2'` (entre apóstrofes) es una cadena de caracteres, mientras que `12 + 2` es una expresión cuyo valor es un entero. También hay unas sentencias llamadas "parámetros de clase" que permiten



seleccionar las distintas representaciones en la máquina para cada tipo de datos. Antes de nada profundicemos un poco en esto:

Parámetros de clase

Gracias a ellos para los datos de tipo numérico podremos seleccionar la precisión y el rango de cada variable y para los datos de tipo carácter podremos usar durante el programa distintos alfabetos.

Cada tipo intrínseco de datos tiene un parámetro de clase asociado a él, este es el parámetro de clase por defecto y es el que la máquina le asigna si el programador no lo especifica.

Para los datos numéricos los números **1**, **2** y **3** (ó **4**, **8** y **16** dependiendo de la máquina) denotan al número en simple, doble o cuádruple precisión, respectivamente, por ejemplo `2132_2` ó bien `2132_8`

Por otro lado, para escoger el número de decimales y el rango de un número hay unas determinadas funciones intrínsecas: `selected_int_kind` y `selected_real_kind` , de las cuales ya hemos hablado.

Los datos de tipo lógico, que sabemos que están almacenados bit a bit, se pueden almacenar byte a byte, con lo que ganaremos en eficacia. Estas representaciones podrán ser hechas por el programador especificando el valor del parámetro de clase.

Para los datos de tipo carácter, el parámetro de clase irá situado antes de la variable y separado de esta por un subrayado (`_`), por ejemplo `ASCII_`aeiou`` o `CHEMICAL_`oro``

NOTA: Los diferentes parámetros de clase suelen depender de la máquina en la que estemos trabajando, por ello es conveniente leerse el manual del ordenador para conocerlos.

Tipos de datos

- **Tipo entero**

El tipo entero se usa para representar valores de números enteros. Una constante entera es una cadena que contiene dígitos del **0** al **9**, que pueden ir seguidos por un subrayado (`_`) y otros enteros o por unos valores que designan el parámetro de clase del entero. Por ejemplo, `23` ; `15_short` ; `15_long` ; `3_2` .

Una constante entera con signo es un entero precedido del signo `+` ó `-`.

- **Tipo real**



El tipo real se usa para representar valores de números reales. Hay dos formas en Fortran90 para representar una constante de tipo real, la primera es la *forma posicional*, que consiste en un entero seguido por un punto y por su parte decimal y la segunda es la *forma exponencial*, que es un real escrito en forma posicional seguido de la letra **E** y un entero con signo. Por ejemplo, `2.3E + 05` representa al número $2.3 * 10^{**} 5$, es decir, al número **230000**. Esta forma se usa para representar números muy grandes o muy pequeños.

En ambas formas el número puede ir seguido de un subrayado y de un parámetro de clase (por ejemplo, `double` o `quad`).

- **Tipo complejo**

El tipo complejo en Fortran90 se usa para representar a los numero complejos. Se escribirá (`a, b`) para denotar al complejo $a + b i$. La precisión del complejo será la mayor precisión que tengan **a** y **b**.

- **Tipo lógico**

El tipo lógico en Fortran90 se usa para representar los valores "verdadero" y "falso".

- **Tipo caracter**

El tipo caracter en Fortran90, se usa para representar cadenas de caracteres. Una constante caracter es una secuencia de caracteres entre apóstrofes o entre comillas.

Operadores

- **Operadores aritméticos**

Son los siguientes: `+` , `-` , `*` , `/` y `**`. Estos operadores relacionan operandos numéricos que no tienen que ser del mismo tipo, ya que el de menor clase se convertirá en el de mayor (los enteros tienen menor clase que los reales y estos a su vez tienen menor clase que los complejos) en el momento de realizarse la operación. Por ejemplo, `15.0 / 2` resulta ser `7.5`, ya que `15.0` es un real, `2` se convierte en real y la división de reales es otro real.

En el caso de que los dos operandos sean del mismo tipo pero tengan distinto parámetro de clase el resultado tendrá el mismo parámetro de clase que el operando con mayor precisión.

- **Operadores de relación**



Son los siguientes: `<` , `<=` , `=` , `/=` , `>=` y `>` (ó también `.LT.` , `.LE.` , `.EQ.` , `.NE.` , `.GE.` , `.GT.` , herederos del Fortran77)

- **Operadores lógicos**

Los operadores que se usan para combinar valores lógicos son, `.NOT.` , `.AND.` , `.OR.` , `.EQV.` y `.NEQV.`

El valor que resulta de aplicar estos operadores lo vemos en la siguiente tabla:

X	Y	.NOT. X	X .AND. Y	X .OR. Y	X .EQV. Y	X .NEQV. Y
TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE

- **Operadores caracter**

Existe un único operador caracter que se representa mediante el símbolo (`//`) y que provoca la concatenación de dos cadenas de caracteres, ambas tienen que tener el mismo parámetro de clase. Además, los espacios en blanco que dejamos entre las cadenas y el operador no son representativos. Por ejemplo, `Carlos // Fernández` nos daría `Carlos Fernández` .

Procedimientos intrínsecos elementales

Todos los procedimientos intrínsecos pueden ser llamados con algún argumento adicional; estos son:

- **KIND** : Sirve para darnos la clase del resultado.
- **MASK** : Es una máscara que le podemos poner a los argumentos.
- **DIM** : Dimensión seleccionada para el argumento si este es un array.

El más usado es el argumento **MASK**, que se usa para seleccionar los elementos de uno o más argumentos que operan en la función, los argumentos que son seleccionados por la máscara no tienen que ser definidos al mismo tiempo que la función sea llamada. El argumento **MASK** es de tipo lógico.

Por otro lado, la función **PRESENT** permite examinar la presencia de su argumento a lo largo de un programa; su resultado es de tipo lógico.

Procedimientos numéricos

Funciones numéricas



A continuación daremos una tabla con las más usadas:

FUNCIONES	SIGNIFICADO	TIPO
ABS	Valor absoluto o módulo de un complejo.	Real o entero, dependiendo de su argumento.
AINT	Trunca el valor de su argumento real	Real.
ANINT	Redondea el valor de su argumento real	Real.
AIMAG	Parte imaginaria de un complejo	Real.
CEILING	Entero mayor o igual que su argumento real	Entero.
CONJG	Conjugado de un complejo	Complejo, con números reales.
DPROD	Producto en doble precisión de dos reales	Real en doble precisión.
DIM(X, Y)	Nos da X - Y si esta diferencia es >0 ó 0 en caso contrario	Real o entero, dependiendo de su argumento.
FLOOR	Parte entera de un real	Entero.
MAX	Máximo	Real.
MIN	Mínimo	Real.
MODULO	Resto de una división	Real o entero, dependiendo de su argumento.
SIGN(A, B)	Es ABS(A) si $B \geq 0$ y $-ABS(A)$ si $B < 0$	Real o entero, dependiendo de su argumento.

Procedimientos matemáticos y lógicos

Funciones matemáticas

Las más usadas las detallaremos en la siguiente tabla:

FUNCIONES	SIGNIFICADO
ACOS	Arco coseno
ASIN	Arco seno
ATAN	Arco tangente
COS	Coseno
COSH	Coseno hiperbólico
EXP	Exponencial



LOG	Logaritmo
LOG10	Logaritmo en base 10
SIN	Seno
SINH	Seno hiperbólico
SQRT	Raíz cuadrada
TAN	Tangente
TANH	Tangente hiperbólica

Funciones lógicas

La función denominada **logical** cambia el parámetro de clase de una variable lógica. Por ejemplo, si **l** es de tipo lógico y **k** es entero, **logical(l, k)** da el valor de **l** representado como una variable lógica con parámetro de clase **k** y la función **logical(l)** nos da el valor de **l** representado como una variable lógica con el parámetro de clase por defecto.

Procedimientos carácter

Funciones caracter

Las más importantes son:

FUNCIONES	SIGNIFICADO	TIPO
ICHAR (C)	Posición que ocupa su argumento en el código ASCII.	Entero.
CHAR (I)	Caracter que corresponde al código ASCII I.	Caracter.
IACHAR (C)	Posición que ocupa su argumento en el código ASCII.	Entero.
ACHAR (I)	Caracter que corresponde al código ASCII I.	Caracter.
INDEX(CAD,SUBCAD)	Primera posición de la subcadena en la cadena.	Entero.
LEN_TRIM (CAD)	Longitud sin espacios en blanco de la cadena.	Entero.
REPEAT (CAD, N)	Concatenar N veces la cadena.	Caracter.
VERIFY (STRING,	Posición del caracter mas a la izqda.	Entero.



SET)	de la cadena que no está en SET. Dará cero si todos los caracteres de la cadena aparecen en SET.	
LEN (CAD)	Longitud de la cadena.	Entero.

Funciones para vectores y matrices

La función para multiplicar las matrices **A** y **B** es **MATMUL (A, B)**. Esta función opera con dos matrices, o con una matriz y un vector y devuelve el correspondiente producto.

La función **DOT_PRODUCT (C, D)** calcula el producto escalar de los vectores **C** y **D**.

La función **TRANSPOSE (A)** nos devuelve el array transpuesto de **A**.

Otras funciones menos importantes son:

ALL (MASK, DIM) que devolverá el valor verdadero cuando todos los valores de la máscara sean ciertos.

ANY (MASK, DIM) que devolverá el valor verdadero cuando algún valor de la máscara sea cierto.

COUNT (MASK, DIM) que nos dará el número de elementos verdaderos en la máscara.

MAXVAL (ARRAY, DIM, MASK) nos da el mayor valor del array sujeto a la máscara y a esa dimensión.

MINVAL (ARRAY, DIM, MASK) nos da el menor valor del array sujeto a la máscara y a esa dimensión.

PRODUCT (ARRAY, MASK) multiplica los elementos del array sujetos a la máscara.

SUM (ARRAY, MASK) suma los elementos del array sujetos a la máscara.

MAXLOC (ARRAY, MASK) nos devuelve la posición del mayor elemento del array



sujeto a la máscara.

MINLOC (**ARRAY**, **MASK**) nos devuelve la posición del menor elemento del array sujeto a la máscara.

NOTA: Los argumentos **DIM** y **MASK** son opcionales. Y su funcionamiento ya ha sido explicado con anterioridad.

Otra función para arrays es la función **ALLOCATED** (**A**) que nos devuelve el valor verdadero si el array ha sido dimensionado.

2.1.- Capítulo 1: Unidades del programa

2.1.1 [Unidades del programa](#)

2.1.2 [Subrutinas](#)

2.1.3 [Funciones](#)

2.1.4 [Módulos](#)

2.2.- Capítulo 2: ¿Cómo pasar argumentos?

2.2.1 [¿Cómo se pasan los argumentos?](#)

2.2.2 [Argumentos por referencia y por valor](#)

2.2.3 [Clasificación de los argumentos](#)

2.3.- Capítulo 3: Procedimientos interface

2.3.1 [Procedimientos interface](#)

2.3.2 [Procedimientos genéricos](#)

2.3.3 [Extensión de asignaciones](#)

2.3.4 [Extensión de operadores](#)



Unidades del programa

Existen muchas clases de unidades de un programa, unas ejecutables, como el programa principal o sus subprogramas y otras no ejecutables, como los ficheros de datos y los módulos, los cuales contienen definiciones que usarán las otras unidades del programa. Únicamente los módulos son nuevos en esta versión del Fortran.

Todos y cada uno de los programas en Fortran90 deben contener un único programa principal, desde el cual se llamará a las otras unidades del programa y en el cual se comenzará el programa. La forma de este es:

```
PROGRAM nombre del programa  
  Declaraciones de datos  
  
  Ejecución  
  
  Llamadas a los subprogramas  
  
END PROGRAM nombre del programa
```

Evidentemente, un programa principal no puede hacerse referencia a sí mismo, es decir, no puede ser recursivo.

Una sentencia en Fortran90 que puede usarse desde el programa principal es la sentencia **USE** que permite el acceso a los módulos, es decir, al escribir **USE *nombre del módulo*** podremos usar todos los contenidos del módulo.

Subrutinas



Tienen la estructura de un programa en Fortran90, salvo que comienzan por la sentencia **SUBROUTINE** y terminan con la sentencia **END SUBROUTINE**. Se usan para realizar algún tipo de cálculo y son llamadas mediante la sentencia **CALL** seguida por el nombre de la subrutina y posiblemente por una lista de argumentos entre paréntesis.

Existen subrutinas que empiezan por la sentencia **RECURSIVE**, esto indica que podemos llamarlas repetidas veces dentro de la propia subrutina.

En una subrutina no es necesaria la sentencia **IMPLICIT NONE** debido a que la del programa ya engloba a todas las subrutinas.

La sentencia **CONTAINS** que aparece justo antes del final del programa principal indica que a partir de ella se pueden escribir las subrutinas de tal forma que en las subrutinas no será necesaria la sentencia **RETURN** y en la mayoría de los casos las subrutinas no tendrán ninguna lista de argumentos.

Las subrutinas con argumentos son aquellas en las que es necesario distinguir los argumentos de entrada y de salida y dependiendo de estos dará un valor u otro. También es posible diferenciar un argumento de entrada y otro de salida, para ello usamos las sentencias **INTENT (IN)** e **INTENT (OUT)** después de la definición de la variable. Por ejemplo:

```
REAL, INTENT (IN) :: X , o bien,
COMPLEX, INTENT (OUT) :: C
```

Es posible que un mismo argumento sea de entrada y de salida, en tal caso usaremos la sentencia **INTENT (INOUT)**.

Los argumentos que solo se usan en la subrutina, se tienen que definir en ella y su valor no afecta a otras variables que estén fuera del programa. Además, tampoco tendrán que ser definidos en el programa principal.

Funciones

Una función es un procedimiento similar a una subrutina. Toda función empieza por una sentencia **FUNCTION** con el nombre de la función seguido por una lista de argumentos de entrada para que la función realice su cálculo. Una función no puede llevar argumentos de salida, ya que la salida de la función es el propio valor de la función. Los argumentos de la función, así como la variable que almacena el resultado, deberán ser declarados en la propia función. En el caso de que al declarar la función ya la declaremos como un tipo determinado, no habrá que declarar la variable en la que almacenamos el resultado. En las funciones, también podemos poner la palabra **RECURSIVE** y así la función podrá ser llamada desde la propia función.

La definición de una función sería, por tanto, como sigue:

```
(tipo de la función)FUNCTION nombre (RESULT(nombre del resultado))
```

```
Bloque de sentencias
```



```
END FUNCTION nombre
```

La palabra **RESULT** (*nombre del resultado*) es opcional y en caso de no ponerla el valor de la función se almacenará en el propio nombre de la función, de igual forma que ocurre en el Fortran77.

Un ejemplo ilustrativo sería el siguiente:

```
REAL FUNCTION F(x) RESULT (F_resul)

    Implicit none

    Real :: x

    F_resul = x ** 2

END FUNCTION F
```

Para llamar desde el programa principal a la función no tenemos nada más que escribir su nombre. En nuestro ejemplo, sólo tendríamos que escribir **F(y)** para calcular la imagen de **y** por la función **F**, es decir, **y ** 2**.

Módulos

Un módulo es una unidad del programa no ejecutable que puede contener una subrutina, una función o bien las características de unos datos, que serán utilizadas posteriormente por otras unidades del programa por medio de la sentencia **USE**.

Esta sentencia **USE** seguida por el nombre o nombres de los módulos, debe ir a continuación de la sentencia **PROGRAM** y a partir de ahí ya se conocerán todas las sentencias del módulo y no será necesario volver a definir las variables que intervienen en el programa y que ya están definidas en el módulo.

Existen otras dos posibilidades para la sentencia **USE**, la primera permite cambiar los nombres de las variables definidas en el módulo. Por ejemplo, con la sentencia:

```
USE MOD, NM = NUMERO_DE_MANZANAS
```

la variable **NM** tendrá las mismas características que la variable **NUMERO_DE_MANZANAS** que ya está definida en el módulo **MOD**. La otra posibilidad de la sentencia **USE** permite usar solamente unas determinadas variables definidas en un módulo. Por ejemplo, con la sentencia:

```
USE MOD, ONLY : NUMERO_DE_MANZANAS, NUMERO_TOTAL
```

sólo usaremos en esta unidad del programa las variables **NUMERO_DE_MANZANAS** y **NUMERO_TOTAL** definidas en el módulo.

Por otra parte, también es posible combinar ambas posibilidades de la sentencia **USE**, escribiendo:

```
USE MOD, ONLY : NM = NUMERO_DE_MANZANAS, NUMERO_TOTAL
```

Cómo se pasan los argumentos?



Una de las propiedades importantes de las funciones y de las subrutinas es que la información puede ser pasada a ellas cuando son llamadas y devolverla, al lugar donde fueron llamadas, cuando finalice su ejecución. El paso de esta información será llevado a cabo por los llamados argumentos.

Existe una correspondencia uno a uno entre los argumentos del programa, llamados argumentos actuales, y los argumentos de las subrutinas o funciones, llamados falsos argumentos. Estos últimos, no tienen que tener el mismo nombre que los otros y la correspondencia es temporal, es decir, sólo es válida durante la llamada al procedimiento. Además, el número de ambos argumentos tiene que coincidir, salvo que declaramos a algún falso argumento como opcional, es decir, que pongamos, por ejemplo:

```
Real, opcional :: x
```

De esta forma el argumento `x`, tanto puede ser pasado como argumento, como no. Por otro lado, también es posible identificar a los argumentos con una letra, de esta forma podremos cambiar la correspondencia por defecto entre los argumentos, y realizarla de una forma más clara. Por ejemplo, podríamos llamar a una función `Func (a, b, c)` de las siguientes formas:

```
Func (5, a = 1, c = 0) Func (5, 0, a = 1) o Func (1, 5, 0)
```

Como vemos es posible identificar a algunos argumentos y a otros no. En estos casos, los argumentos que no estén identificados tendrán que ir ordenados entre sí. En cualquiera de los casos, `a = 1`, `b = 5` y `c = 0`.

Por otra parte, el tipo y el parámetro de clase de cada argumento actual deben coincidir con el de su correspondiente falso argumento.

En resumen, podemos decir que existen dos formas típicas de pasar argumentos, por referencia y por valor. A continuación pasaremos a detallarlas.

Argumentos por referencia

Un argumento actual que es una variable, sólo podrá ser pasado por referencia. La referencia al correspondiente falso argumento de la subrutina provoca que el ordenador lo considere como el argumento actual correspondiente. Si en la subrutina se cambia el falso argumento estos cambios también afectarán al argumento actual. Por ello es una mala práctica en la programación, realizar cambios en los argumentos de entrada, que tengan lugar en una subrutina o en una función.

Argumentos por valor

Un argumento actual que sea una constante o una expresión más complicada que una variable, sólo podrá ser pasado por valor al correspondiente falso argumento. Así, el falso argumento no podrá cambiar su valor durante la ejecución del procedimiento.



Clasificación de los argumentos

Antes de nada hay que aclarar que este capítulo sólo es necesario para hacer más legibles los programas.

En Fortran90 es posible clasificar a los falsos argumentos como argumentos de entrada o de salida. Para ello, usaremos las sentencias `INTENT (IN)` e `INTENT (OUT)` que especificarán si el argumento es de entrada o de salida.

Si a un argumento le añadimos la sentencia `INTENT (IN)` su valor no podrá ser cambiado dentro de la subrutina o función en la que nos encontremos.

Si, por otra parte, le colocamos la sentencia `INTENT (OUT)` significa, que al argumento actual correspondiente, le será devuelto un valor por la subrutina o la función correspondiente.

También es posible, usar la sentencia `INTENT (INOUT)` que significa que un falso argumento va a recibir un valor inicial y va a devolver otro valor al correspondiente argumento actual.

Como podemos observar, si el falso argumento es de salida o de entrada-salida, su correspondiente argumento actual será pasado por referencia y tendrá que ser una variable.

Le daremos a los argumentos estos atributos en el momento de ser declarados dentro del procedimiento, ya sea este una subrutina o una función.

Procedimientos interface

Los bloques interface son usados para:

1. Permitir al usuario dar unas propiedades genéricas a los procedimientos.
2. Definir nuevos operadores y extender las propiedades de los operadores intrínsecos.
3. Extender las asignaciones a otros tipos de datos.

En todos los casos, los bloques interface contienen la información necesaria para el compilador, para que este realice una llamada correcta a los procedimientos. Por otro lado, el bloque interface debe ir en la parte específica de la llamada a la rutina o en el módulo usado para llamar a la rutina. La forma general de un bloque interface es:

```
INTERFACE [especificación genérica]  
  [cuerpo del interface]  
MODULE PROCEDURE nombres de los procedimientos
```



```
END INTERFACE
```

Siendo una especificación genérica una de las tres siguientes:

Nombre genérico

`OPERATOR (operador)`

`ASSIGNMENT (=)`

Y siendo el cuerpo del interface una función o una subrutina.

Procedimientos genéricos

Muchos procedimientos intrínsecos son genéricos en el sentido de que permiten argumentos de diferentes tipos. En Fortran90, el programador puede escribir procedimientos genéricos nuevos. Por ejemplo, si tenemos una subrutina cuyos argumentos son enteros, podremos crear otra con otro nombre que opere con reales y dependiendo de cómo sean los argumentos en la llamada, usar una u otra. Para hacer esto, crearemos un bloque interface que elija una u otra subrutina, dependiendo del tipo de argumentos con el que sea llamado el propio bloque interface. Un ejemplo de esta aplicación lo podremos ver en el ejemplo de esta página.

Extensión de asignaciones

A veces, cuando ejecutamos una sentencia de asignación, el tipo de la variable del lado derecho del operador (`=`), se convierte en el tipo de la variable del lado izquierdo de dicho operador. Por ejemplo, la asignación:

```
X = I
```

con `X` real e `I` entero, provoca que `I` sea convertido a tipo real.

Supongamos que se quiere realizar la asignación:

```
I = L
```

siendo `I` un entero y `L` una variable lógica, y que le asigne a `I` el valor `1` si `L = TRUE` y el `0` si `L = FALSE`.

Para ello, crearemos una subrutina que realice esta asignación y un bloque interface que me indique que la asignación ha sido extendida por la subrutina. En el ejemplo que sigue, veremos como se lleva a cabo esta asignación:

NOTA: Toda subrutina que sirva para definir una asignación debe tener exactamente dos argumentos, uno de entrada y otro de salida.



Extensión de operadores

Con el Fortran90 podemos extender operadores, es decir, podemos darle a un operador ya definido otras utilidades que no tenía. Por ejemplo, se le puede dar al operador (+) la posibilidad de sumar variables lógicas, o bien, la capacidad de sumar vectores, etc. . Para ello, crearemos una función que haga esa extensión y un bloque interface que me indique que la extensión ha sido realizada por dicha función. Así, en el ejemplo siguiente crearemos el módulo que lleve a cabo la primera de esas extensiones.

NOTA: Toda función que sirva para extender un operador debe tener uno o dos argumentos, ambos deben ser de entrada.

Definición de operadores

Además de poder extender los operadores y darles otra función, también es posible crear nuevos nombres para operadores. Estos nombres constarán de entre 1 y 31 letras, precedidos y seguidos por un punto, excepto para nombres de constantes lógicas y de operadores intrínsecos.

Para definir un operador, crearemos un bloque interface y una función con el nombre del operador que realice esta definición. Por ejemplo:

```
MODULE INVERSO
    INTERFACE OPERATOR (.INVERS.)
        MODULE PROCEDURE INVERS
    END INTERFACE
CONTAINS
    FUNCTION INVERS (A) RESULT (INVERS_RESUL)
        .....
        .....
    END FUNCTION INVERS
END MODULE INVERSO
```



3.1.- Capítulo 1: Declaración



3.1.1 [Declaración](#)

3.1.2 [La sentencia WHERE](#)

3.2.- Capítulo 2: Reserva dinámica de memoria

3.2.1 [Reserva dinámica de memoria](#)

3.3.- Capítulo 3: Punteros

3.3.1 [Punteros](#)

3.3.2 [Sentencias para punteros](#)



Declaración

En Fortran, una colección de valores del mismo tipo se denomina array. En Fortran77, para trabajar con un array había que usar un bucle en elementos e ir trabajando elemento a elemento; sin embargo, el Fortran90 es mucho más potente en este sentido y permite trabajar con todo el array o con una parte de él. Los programas que están escritos de esta forma son más fáciles de compilar, sobre todo por ordenadores con procesadores paralelos, los cuales aumentarían la velocidad de ejecución.

En la declaración de un array se deben seguir las mismas reglas que en la definición de cualquier otra variable. La forma general de definir un array unidimensional es la siguiente:

Tipo del array, Dimension (n:m) :: nombre del array

siendo **n** y **m** las dimensiones del array. Si quisiéramos definir un array de dos dimensiones tendríamos que escribir **Dimension (n : m, n : m)**. Ejemplos :

```
REAL, DIMENSION ( 0 : 5, 3 ) :: Y
CHARACTER ( LEN = 5 ), DIMENSION ( 8 ) :: LISTA
```

En estos casos, **Y** sería una matriz (array de dos dimensiones) real con 6 filas (de la 0 a la 5) y 3 columnas (de la 1 a la 3) y **LISTA** sería un vector de 8 cadenas de caracteres cada una de longitud 5.

A los arrays se les puede dar un valor inicial. Existen tres clases distintas de valores:

1. Una expresión escalar:

```
X = ( / 3.2, 2.0 / )
```

2. Una expresión vectorial:

```
X = ( / A(I, 1:2), A(I+1, 2:3) / )
```

3. Una expresión con un bucle DO:

```
X = ( / ( I + 2 , I = 1, 2 ) / )
```

Sea **x** un vector de 10 componentes, así, en el Fortran90 es posible realizar este tipo de asignaciones:

x(4:8:2)=3.2. En la cual, los elementos **x(4)**, **x(6)** y **x(8)** toman el valor **3.2**
x(1:3)=x(5:7). Con la cual, hacemos, **x(1)=x(5)**, **x(2)=x(6)** y **x(3)=x(7)**
x(2:4)=x(2:4)*2.0. Con la cual, multiplicamos los elementos **x(2)**, **x(3)** y **x(4)** por **2**.



Declaración

En Fortran, una colección de valores del mismo tipo se denomina array. En Fortran77, para trabajar con un array había que usar un bucle en elementos e ir trabajando elemento a elemento; sin embargo, el Fortran90 es mucho más potente en este sentido y permite trabajar con todo el array o con una parte de él. Los programas que están escritos de esta forma son más fáciles de compilar, sobre todo por ordenadores con procesadores paralelos, los cuales aumentarían la velocidad de ejecución.

En la declaración de un array se deben seguir las mismas reglas que en la definición de cualquier otra variable. La forma general de definir un array unidimensional es la siguiente:

Tipo del array, Dimension (n:m) :: nombre del array
 siendo **n** y **m** las dimensiones del array. Si quisiéramos definir un array de dos dimensiones tendríamos que escribir *Dimension (n : m, n : m)*. Ejemplos :

```
REAL, DIMENSION ( 0 : 5, 3 ) :: Y
CHARACTER ( LEN = 5 ), DIMENSION ( 8 ) :: LISTA
```

En estos casos, **Y** sería una matriz (array de dos dimensiones) real con 6 filas (de la 0 a la 5) y 3 columnas (de la 1 a la 3) y **LISTA** sería un vector de 8 cadenas de caracteres cada una de longitud 5.

A los arrays se les puede dar un valor inicial. Existen tres clases distintas de valores:

1. Una expresión escalar:

```
X = ( / 3.2, 2.0 / )
```

2. Una expresión vectorial:

```
X = ( / A(I, 1:2), A(I+1, 2:3) / )
```

3. Una expresión con un bucle DO:

```
X = ( / ( I + 2 , I = 1, 2 ) / )
```

Sea **x** un vector de 10 componentes, así, en el Fortran90 es posible realizar este tipo de asignaciones:

x(4:8:2)=3.2. En la cual, los elementos **x(4)**, **x(6)** y **x(8)** toman el valor **3.2**
x(1:3)=x(5:7). Con la cual, hacemos, **x(1)=x(5)**, **x(2)=x(6)** y **x(3)=x(7)**
x(2:4)=x(2:4)*2.0. Con la cual, multiplicamos los elementos **x(2)**, **x(3)** y **x(4)** por **2**.



La sentencia WHERE

La sentencia **where** se usa para asignar valores a unos ciertos elementos de un array, cuando una condición lógica es verdadera. La forma general de una sentencia **where** es la siguiente:

```
WHERE ( condición lógica)
      Bloque de sentencias
ELSEWHERE
      Bloque de sentencias
END WHERE
```

Dentro de una sentencia **where** sólo están permitidas asignaciones con arrays. Además, la clase de todos los arrays de las sentencias de asignación, debe coincidir con la clase del array de la expresión lógica del **where**. Por otra parte, las asignaciones serán ejecutadas en el orden que son escritas, primero las del bloque **where** y luego las del bloque **elsewhere**.

Reserva dinámica de memoria

Es una característica nueva del Fortran90 y permite declarar arrays de forma dinámica, es decir, podremos declarar un array según la memoria que vaya a utilizar. Un array dinámico será declarado con una sentencia de declaración de tipo, junto con el atributo **ALLOCATABLE**. Por ejemplo, un array dinámico real de dos dimensiones lo declararíamos así:

```
REAL, DIMENSION(:, :), ALLOCATABLE :: A
```

Con esta declaración, no reservamos ningún espacio de memoria para el array **A**, ya que este espacio será asignado dinámicamente en el programa cuando el array sea usado y mediante la sentencia **ALLOCATE**:

```
ALLOCATE A(0:n, 0:m)
```

Si posteriormente deseamos liberar este espacio de memoria, también podremos hacerlo, para ello usaremos la sentencia **DEALLOCATE**:

```
DEALLOCATE (A)
```

Ambas sentencias, la **ALLOCATE** y la **DEALLOCATE** pueden ir acompañadas por el argumento opcional **STAT**. **STAT** será una variable entera cuyo valor será un número positivo en caso de haber un error en la ejecución de alguna de esas sentencias y valdrá cero si no lo hay.

Con la sentencia **ALLOCATED (A)** sabremos si tenemos reservada memoria para el array **A**.

Un array dinámico puede ser declarado dentro de un módulo, en este caso, la memoria para el array se mantendrá a lo largo de la ejecución de la unidad del programa que use este módulo.



Reglas para el uso de arrays dinámicos:

- Los arrays dinámicos no pueden ser falsos argumentos de un procedimiento y hay que asignarle la memoria en la unidad del programa donde se usen.
- El resultado de una función no puede ser un array dinámico.

Con la sentencia **SIZE** también podemos asignarle memoria a un array, es decir, con la sentencia:

```
REAL, DIMENSION (SIZE(A)) :: C
```

lo que hacemos es definir **C**, como un array de dimensión uno con el mismo tamaño que el array unidimensional **A**.

Punteros

Un puntero, o una variable puntero, es una variable que apunta a un objetivo de su mismo tipo. Un puntero se suele usar como una variable normal aunque tenga otras propiedades. Para poder comprender como trabajan los punteros hay que considerarlos como un "alias". En Fortran90, a diferencia del C, un puntero no contiene ningún dato propio y no es una dirección de memoria. En Fortran90, el puntero y la variable se refieren a lo mismo, son dos nombres para la misma memoria. Los usos más importantes para un puntero son:

- Dar una alternativa mejor a la reserva dinámica de memoria.
- Crear y manipular listas y otras estructuras de datos dinámicas.

La forma general de declarar un puntero es:

```
Tipo (atributos) POINTER :: lista de punteros
```

La variable a la que apunta el puntero tiene que ser declarada como **TARGET**, y tiene que ser del mismo tipo que el puntero. Por otra parte, no es necesario que esta variable tenga un valor definido.

La sentencia de asignación para los punteros es la siguiente:

```
POINTER => TARGET
```

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

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

NOTA: Sean **p1** y **p2** los punteros de las variables **t1** y **t2**. Así, con la sentencia **p2 => p1**, lo que hacemos es que **p2** también sea un puntero de **t1**. Sin embargo, la sentencia **p2 = p1** es equivalente a la asignación **t2 = t1**.



Sentencias para punteros

Las sentencias ALLOCATE y DEALLOCATE

Con la sentencia **ALLOCATE**, creamos espacio de memoria para un valor y hacemos que el puntero se refiera a ese espacio. Si después queremos que el puntero tenga un valor tendremos que asignárselo.

La sentencia **DEALLOCATE**, desecha el espacio de memoria al que apunta su argumento y hace que su objetivo sea nulo. Después de esta sentencia no podremos hacer referencia al valor al que apuntaba este puntero.

La sentencia NULLIFY

Al comienzo del programa una variable puntero, como el resto, no está definida. A veces, es deseable tener una variable puntero que no apunte a nada. Para crear un puntero nulo, que se llama, utilizamos la sentencia **NULLIFY**, que consta de la palabra **NULLIFY** seguida por el nombre de la variable puntero entre paréntesis. Si el objetivo de dos punteros **P1** y **P2** es el mismo, la sentencia **NULLIFY (P1)** no hace nulo a **P2**, sólo a **P1**. Si posteriormente, realizamos la sentencia **P2 => P1**, entonces si que **P2** será nulo. Como vemos, un puntero nulo no está asociado con ningún objetivo de otro puntero.

La función intrínseca ASSOCIATED

La función **ASSOCIATED (P1)**, me indica si el puntero **P1**, que tiene que estar definido, apunta a algún objeto. Esta función tiene otras dos aplicaciones:

- **ASSOCIATED (P1, R)**, me indica si el puntero **P1** es el alias del objetivo **R**.
- **ASSOCIATED (P1, P2)**, me indica si los punteros **P1** y **P2** tienen el mismo objetivo, o si ambos son nulos

NOTA: Si dos punteros son alias de diferentes partes de un array, estos no se considerarán como asociados.



4.1.- Capítulo 1: Registros, ficheros y métodos de acceso

- 4.1.1 [Introducción](#)
- 4.1.2 [Registros - Registros de datos](#)
- 4.1.3 [Registros fin de fichero](#)
- 4.1.4 [Ficheros](#)
- 4.1.5 [Ficheros externos e internos](#)
- 4.1.6 [Métodos de acceso a ficheros](#)
- 4.1.7 [Unidades y ficheros de conexión](#)

4.2.- Capítulo 2: Condiciones de error



4.2.1 [Condiciones de error](#)

4.3.- Capítulo 3: Sentencias de transferencia de datos

- 4.3.1 [Sentencias de transferencia de datos](#)
- 4.3.2 [Transferencia de datos en ficheros internos](#)
- 4.3.3 [Entrada/salida sin formatear](#)
- 4.3.4 [La sentencia OPEN](#)
- 4.3.5 [La sentencia CLOSE](#)
- 4.3.6 [La sentencia INQUIRE I](#)
- 4.3.7 [La sentencia INQUIRE II](#)
- 4.3.8 [Sentencias de situación de ficheros](#)

4.4.- Capítulo 4: Formatos de entrada/salida

- 4.4.1 [Formatos de entrada/salida](#)
- 4.4.2 [Transferencia de datos formateados](#)
- 4.4.3 [Editores numéricos](#)
- 4.4.4 [Editores enteros y reales](#)
- 4.4.5 [Editor complejo, lógico, de caracter, de posición, editor \(/ \) y editor \(: \)](#)
- 4.4.6 [Formato por defecto](#)
- 4.4.7 [Formato por defecto para los distintos tipos de variables](#)

Introducción

Muchos programas necesitan datos para comenzar un cálculo y después de realizado este cálculo, a menudo se quieren imprimir los resultados. También es habitual que una parte del programa necesite los datos que otra parte está calculando.

Ya en la primera sección, hablamos de las sentencias **READ** y **PRINT**,



que son las sentencias de entrada/salida más elementales. En esta sección, describiremos otras características del Fortran para la entrada/salida de datos. Describiremos las sentencias de entrada/salida siguientes:

READ, PRINT, WRITE, OPEN, CLOSE, INQUIRE, BACKSPACE, ENDFILE y **REWIND**.

La sentencia **READ**, es una sentencia de entrada de datos que nos permite leer unos datos, bien sea por pantalla o de un fichero, y asignarles un nombre.

Las sentencias **PRINT** y **WRITE**, son sentencias de salida de datos, estas imprimirán unos datos, tanto por pantalla como en un fichero, respectivamente.

Las sentencias **OPEN** y **CLOSE**, sirven para crear y cerrar un fichero, respectivamente.

La sentencia **INQUIRE**, nos da unos datos sobre la posición en un fichero o unidad.

Las sentencias **BACKSPACE, ENDFILE** y **REWIND**, son sentencias que nos sitúan en unas determinadas posiciones de un fichero.

La primera parte de esta sección contendrá las ideas fundamentales, necesarias para entender como se trabaja en Fortran con la entrada/salida de datos.

En la segunda parte, trataremos las diversas condiciones de error.

En la tercera parte, trataremos las diversas sentencias para la transferencia de datos.

Posteriormente, en la última parte, explicaremos como se formatean estos datos.

Registros

Los datos están organizados en registros, correspondiendo a líneas en la terminal del ordenador, en la pantalla, o a partes de un disco. Existen dos clases de registros, registros de datos y registros fin de fichero:



Registros de datos

Un registro de datos se corresponde con una sucesión de valores. Se suele representar, esquemáticamente, como un conjunto de cajas pequeñas, cada una conteniendo un valor. Los valores se representan de dos formas, formateados o sin formatear. Si los valores de los datos son caracteres leíbles por una persona, cada caracter es un valor y diremos que el dato está formateado. Por ejemplo, la sentencia,

```
WRITE ( *, "( I1, A, I2 )", 3, ^, ^, 10
```

produce un registro conteniendo los valores de los cuatro caracteres " 3 " " , " " 1 " y " 0 " , este sería:

3	,	1	0
---	---	---	---

Los datos no formateados, consisten en valores representados de la forma que los almacena la máquina. Por ejemplo, si los enteros son almacenados como números binarios de ocho bits, la ejecución de la sentencia

```
WRITE ( 6 ) 6, 11
```

produciría un registro sin formatear de la forma siguiente:

00000110	00001011
----------	----------

Por tanto, un registro formateado, será aquel que sólo contiene datos formateados, este será creado por una persona o por un programa Fortran que convierta los valores almacenados internamente, en cadenas de caracteres leíbles que representan a estos caracteres. Cuando un dato formateado es leído por el ordenador este será transformado a la representación interna del ordenador (representación binaria, si el dato es un valor)

La longitud de un registro formateado es el número de caracteres que posee. (Esta puede ser cero).

Por otro lado, un registro sin formatear es aquel que sólo contiene datos sin formatear, estos son creados mientras un programa Fortran se está ejecutando. Los datos sin formatear precisan de menos espacio en una unidad externa. Además, son más rápidos de leer y escribir por un ordenador, debido a que no precisan de ninguna conversión. Sin embargo, los datos sin formatear tienen los inconvenientes de no ser manejables para la lectura del programador y de no ser portables, ya que la representación interna de los datos sin formatear es propia de cada máquina. La longitud de un registro sin formatear vendrá dada por la sentencia **INQUIRE**, que más adelante veremos.

Registros fin de fichero



La otra clase de registros, son los registros fin de fichero, los cuales no poseen valores ni tienen longitud. Los registros fin de fichero tienen que ser los últimos registros de un fichero, estos son usados para señalar el final de un fichero y bien pueden ser escritos explícitamente por el programador, usando la sentencia **ENDFILE**, o bien, pueden estar declarados implícitamente por el ordenador, esto ocurre cuando la última sentencia de transferencia de datos del fichero es una sentencia de salida, y entonces:

1. La sentencia **BACKSPACE** será ejecutada,
2. la sentencia **REWIND** será ejecutada, o
3. el fichero será cerrado.

Impresión de registros formateados

Si queremos imprimir un registro formateado, hay que tener en cuenta que el primer caracter del registro es un caracter de control y que este no será impreso. El resto de los caracteres del registro serán impresos en una línea. Los caracteres de control de la primera línea los detallamos en la siguiente tabla:

CARACTER	ESPACIO VERTICAL ANTES DE IMPRIMIR
Blanco	Una línea (espacio simple)
0	Dos líneas (doble espacio)
1	Primera línea de la próxima página
+	No se avanza (imprimimos a continuación de línea anterior)

En caso de no haber caracteres en el registro, se imprimirá una línea en blanco. A veces, es complicado determinar si el primer caracter de un registro es un caracter de control o no, normalmente lo será, pero para estar completamente seguros, solo hay dos formas saberlo, una es por la propia experiencia y otra leyendo el manual del sistema que tengas.

Ficheros

Un fichero no es mas que un conjunto de registros. Un fichero será representado esquemáticamente como un conjunto de cajas, cada una representando a un registro. Los registros de un fichero deben ser todos del mismo tipo, o todos formateados o todos sin formatear, salvo que el fichero contenga un registro fin de fichero. Un fichero puede tener un nombre, la longitud de este y los caracteres que se pueden usar en el nombre dependen del sistema que usemos.



Existen dos clases de ficheros, los que están situados en una unidad, como un disco, y los ficheros accesibles por un programa que están en memoria. Por tanto, tenemos:

1. Ficheros externos, y
2. Ficheros internos.

Los ficheros internos, poseen cadenas de caracteres, que en la memoria del ordenador se transformarán en los valores de los datos.

Mientras que los ficheros externos, ya poseen los valores de los datos, y la memoria del ordenador puede acceder a ellos, tanto para lectura como para escritura. Una vez dentro de la memoria del ordenador, los datos también podrán ser transformados en cadenas de caracteres

Ficheros externos

Un fichero externo, suele estar almacenado en una unidad periférica, como un disco, una terminal de ordenador,... . Para cada fichero externo, hay una clase de métodos de acceso permitidos, una clase de formas permitidas (formateadas o sin formatear), una clase de acciones permitidas y una clase de longitudes registros permitidas. Todas estas características, están establecidas en el sistema que uses, aunque también pueden ser determinadas por el usuario de acuerdo con el sistema operativo.

Ficheros internos

Los ficheros internos, son almacenados en la memoria del ordenador como valores de variables caracter. Estos valores son creados usando el significado usual de la asignación de variables caracter o son creados con una sentencia de salida usando la variable como un fichero interno. Si la variable es un escalar, el fichero tendrá solo un registro. Si la variable es un array, el fichero tendrá un registro para cada elemento del array.

Por ejemplo, si **C** es un array declarado como:

```
CHARACTER (LEN = 7), DIMENSION (2) :: C
```

la sentencia:

```
WRITE ( C, "(F5.3, F5.3)" ) 4.0, 5.0
```

produce el mismo efecto que las sentencias de asignación:

```
C(1) = 4.0 ; C(2) = 5.0
```

La longitud del registro, es el número de caracteres declarados por la variable caracter. En los ficheros internos solo está permitido el acceso secuencial formateado, más adelante aclararemos este concepto.



Existencia de ficheros

A los ficheros que son conocidos por el procesador y son evaluables al ejecutar el programa, se les llaman ficheros que existen. Un fichero puede no existir, debido por ejemplo, a que no está en ningún disco accesible por el programa. La sentencia **INQUIRE** se usa para saber cuando un fichero existe. Cuando el programa crea un fichero, se dice que este existe. Muchas sentencias de entrada/salida hacen referencia a ficheros que existen para el programa en este punto de la ejecución. Por otro lado, una sentencia **WRITE** crea un fichero que no existe y escribe datos dentro del fichero.

Tengamos en cuenta que un fichero interno siempre existe.

Métodos de acceso a ficheros

Existen dos formas de acceder a ficheros externos, mediante el acceso secuencial o mediante el acceso directo.

Mediante el acceso secuencial, los registros de un fichero son recorridos en el sentido que aparecen en el fichero. No es posible comenzar en un registro particular dentro del fichero sin haber leído los anteriores registros de forma secuencial.

Mediante el acceso directo, los registros son seleccionados por un número de registro. Usando esta identificación, los registros podrán ser leídos y escritos en cualquier orden. Si usamos el acceso directo tenemos las siguientes reglas:

1. Todos los registros deben tener la misma longitud.
2. No es posible borrar un registro.
3. No se puede acceder a un fichero interno.

Si los registros están escritos en orden directo y son leídos en orden secuencial, los registros se comenzarán a leer por el registro número uno, reenumerado cuando fue escrito.

Cada fichero tiene una clase de métodos de acceso permitidos, secuencial o directo. Sin embargo, hay ficheros a los que se puede acceder de ambas formas. Puede ocurrir que a un fichero se acceda de modo secuencial desde un programa, luego se desconecte, y posteriormente se acceda de forma directa desde el mismo programa, al mismo fichero.

Posición de un fichero

Todo fichero que es procesado por un programa tiene una posición determinada. Durante, la ejecución del programa, los ficheros son leídos y escritos, lo que



provoca que la posición del fichero varíe. Además, también hay sentencias en Fortran90 que provocan que la posición del fichero cambie, un ejemplo es la sentencia **BACKSPACE**.

El punto inicial de un fichero, es el punto que está justo antes del primer registro. El punto final de un fichero, es el punto que está justo después del último registro. Se llama registro actual, al registro en el que está el programa. Cuando estoy en un registro actual, la posición del fichero será, la posición inicial del registro, una posición entre los valores del registro o la posición final del registro. Un fichero interno está siempre situado al principio del registro.

Entrada/salida avanzando y sin avanzar

Después de finalizar una operación de entrada/salida avanzando, la posición del fichero siempre estará al final del registro. Por el contrario, al finalizar una operación de entrada/salida sin avanzar, la posición del fichero estará entre dos caracteres dentro de un registro. La operación de entrada/salida sin avanzar, solo está permitido usarla para el acceso secuencial a ficheros externos formateados y no es posible usarla en otros casos. La entrada/salida sin avanzar, se indicará con la sentencia **ADVANCE = "NO"** en la sentencia de control de la entrada/salida.

Unidades y ficheros de conexión

Hay sentencias de entrada/salida que hacen referencia a ficheros particulares especificando su unidad. Por ejemplo, las sentencias **READ** y **WRITE** no hacen referencia a ficheros directamente, sino que hacen referencia a un número de unidad, el cual está conectado a un fichero. El número de unidad para un fichero externo es un entero no negativo. Al nombre de un fichero interno también se le llama unidad, este es una variable caracter.

Algunas reglas y restricciones para las variables caracter son:

1. La unidad * especifica al procesador un número de unidad determinado. En la entrada, es el mismo número que usaría el procesador si en la sentencia **READ** no apareciese el número de unidad. En la salida, es el mismo número que usaría el procesador si en la sentencia **PRINT** no apareciese el número de unidad. La unidad especificada con un * sólo puede ser usada para acceso secuencial formateado.
2. Un número de unidad solo puede hacer referencia a una única unidad en todo el programa Fortran.

Un fichero interno, está siempre conectado a la unidad que lleva el nombre de la variable caracter. Para tratar los datos de un fichero externo, este debe estar



conectado a una unidad. Una vez que la conexión está hecha, las sentencias de entrada/salida usarán el número de unidad en vez del nombre del fichero.

Existen dos formas de establecer esta conexión entre una unidad y un fichero externo:

1. Ejecutando una sentencia **OPEN** durante el programa.
2. Preconectándolos mediante el sistema operativo. Algunas unidades, están preconectadas a ficheros, mediante el sistema operativo, y son conocidas por todo programa Fortran, sin que este realice ninguna ejecución. En estos casos, el programador no necesitará de ninguna sentencia **OPEN** para conectar el fichero.

Condiciones de error

Durante la ejecución de una sentencia de entrada/salida, pueden ocurrir algunos errores. Estos errores los controlaremos usando la palabra **IOSTAT=** en las sentencias de entrada/salida. Cada error en la entrada/salida de datos, dará un valor negativo para la variable **IOSTAT**, este valor es propio de cada sistema. Algunos ejemplos de posibles errores son, intentar abrir un fichero que no existe, intentar leer un fichero con un tipo de datos erróneo, etc. .

Si con la sentencia **READ**, intentamos leer un registro fin de fichero, la variable **IOSTAT**, tomará un valor negativo. Por tanto, para saber cuando se ha alcanzado el final de una unidad, será suficiente ver si la variable **IOSTAT** toma un valor negativo. Habitualmente, tomará el valor **-1**, si llega al final del fichero y el valor **-2**, si llega al final del registro.

Sentencias de transferencia de datos

Estas sentencias son **READ**, **WRITE** y **PRINT**. La forma general para una sentencia de transferencia de datos es la siguiente:

```
READ formato [, lista de entrada]
READ ( lista de control ) [lista de entrada]
WRITE ( lista de control ) [lista de salida]
PRINT formato [, lista de salida]
```

El *formato* es, o una expresión de caracteres que indica el formato de la variable, o un *****, que indica el formato por defecto. Más adelante profundizaremos en este tema.

La *lista de control*, tiene que tener un especificador de unidad de la forma

```
[ UNIT ] = unidad de entrada/salida
```

y puede contener algunos de los siguientes atributos opcionales:

```
[ FMT = ] formato
REC = expresión escalar o entera
IOSTAT = variable escalar o entera
ADVANCE = expresión escalar o caracter
```



La palabra **FMT =** puede ser omitida siempre y cuando la palabra **UNIT =** sea omitida. En este caso, el número de unidad será el primer artículo de la lista y el formato será el segundo. La palabra **REC**, indica el número de registro en el que vamos a leer o escribir. La palabra **IOSTAT**, hace referencia a las condiciones de error y la palabra **ADVANCE**, especifica si la entrada/salida es avanzando o sin avanzar. La diferencia entre la transferencia de datos avanzando y la transferencia de datos sin avanzar, es que en la primera de ellas siempre se abandona el fichero al final del registro, es decir, cuando la transferencia de datos está realizada, y en la segunda de ellas se abandona el fichero después de leer o escribir el último carácter. Esta última, sólo se usa con ficheros externos conectados con acceso secuencial.

La *lista de entrada/salida*, consta básicamente de variables o expresiones para leer o escribir.

Transferencia de datos en ficheros internos

Las sentencias de transferencia de datos que se usan en ficheros internos son:

```
READ ([ UNIT = ] variable caracter, [ FMT = ] formato &
      & [, IOSTAT = ] variable) lista de entrada
WRITE ([ UNIT = ] variable caracter, [ FMT = ] formato &
      & [, IOSTAT = ] variable) lista de salida
```

El símbolo **FMT =** puede ser omitido solo si el símbolo **UNIT =** es omitido. Algunas reglas y restricciones para el uso de ficheros internos son:

1. La unidad debe ser una variable caracter
2. Cada registro de un fichero interno debe ser una variable caracter escalar
3. Si el fichero es un array o una sección de un array, cada elemento tiene que ser una variable caracter, la cual constituye un registro. La longitud de los registros debe ser la misma y debe coincidir con la longitud de un elemento del array.
4. Si la variable caracter es un array, dimensionado de forma dinámica, debe ser dimensionado antes de usarlo como un fichero interno.
5. Si el número de caracteres escritos es menor que la longitud del registro, los caracteres sobrantes serán blancos. En caso contrario los caracteres sobrantes serán truncados.
6. En un fichero interno, a los registros se le asignan valores cuando son escritos.
7. Para poder leer un registro en un fichero interno, este debe estar definido.
8. Antes de que la transferencia de datos ocurra, un fichero interno está siempre posicionado al principio.
9. En un fichero interno solo está permitido el acceso secuencial formateado.



Entrada/salida sin formatear

Para la entrada/salida sin formatear, el fichero consta de unos valores almacenados usando la misma representación que en la memoria del programa. Esto implica que no se necesita ningún tipo de conversión durante la entrada/salida. Además, la entrada/salida de datos se puede hacer mediante acceso secuencial o mediante acceso directo y siempre avanzando. Las sentencias de transferencia de datos sin formatear son de la forma:

```
READ ([UNIT=] núm. de unidad, [ REC= ] núm. de registro &
      & [, IOSTAT= ] variable) lista de entrada
WRITE ([UNIT=] núm. de unidad, [ REC= ] núm. de registro &
      & [, IOSTAT= ] variable) lista de salida
```

Si el atributo **REC=** no está presente indicamos que el acceso es directo, si por el contrario aparece, el acceso es secuencial.

Si el acceso es secuencial, la posición del fichero será al principio del siguiente registro antes de la transferencia de datos y estará posicionado al final del registro cuando la sentencia de entrada/salida finalice.

NOTA: La entrada/salida sin formatear y sin avanzar no está permitida.

Transferencia de datos con acceso directo:

Cuando accedemos a un fichero directamente, el registro que será procesado está dado por el número de registro en el símbolo **REC=**. El fichero puede ser formateado o sin formatear.

La forma general de una sentencia de acceso a datos de forma directa es:

```
READ ([ UNIT = ] núm. de unidad [, FMT = ] formato, REC= núm. de
registro &
      [,IOSTAT = ] variable) lista de entrada
WRITE ([ UNIT = ] núm. de unidad [, FMT = ] formato, REC= núm. de
registro &
      [,IOSTAT = ] variable) lista de salida
```

El símbolo **FMT=** puede ser omitido si se omite el símbolo **UNIT=**. El formato no puede ser *****.

La sentencia OPEN

La sentencia **OPEN** establece la conexión entre una unidad y un fichero externo y determina las propiedades de esta conexión. Después de realizada esta conexión, el fichero podrá ser usado para realizar transferencias de datos, sólo con usar su número de unidad.



Desde que conectamos un fichero a una unidad, el resto del programa conocerá esta conexión y no será posible conectar otro fichero a esta unidad.

La forma general de una sentencia **OPEN** es:

OPEN ([**UNIT** =] lista de propiedades de la conexión)

donde las propiedades de la conexión permitidas son:

ACCESS= **DIRECT** o **SEQUENTIAL**, dependiendo del tipo de acceso al fichero. El valor por defecto es **DIRECT**.

ACTION= **READ**, **WRITE** o **READWRITE**, dependiendo de si queremos leer, escribir o realizar ambas cosas, dentro del fichero. El valor por defecto es **READWRITE**.

BLANK= **NULL** (si queremos ignorar los espacios en blanco) o **ZERO** (si queremos que los espacios en blanco sean interpretados como ceros. Solo se puede usar para conectar un fichero con entrada/salida formateada. El valor por defecto es **NULL**.

DELIM= **APOSTROPHE**, **QUOTE** (indican que el final de la cadena de caracteres está denotado por un apóstrofe o unas comillas) o **NONE** (si la cadena de caracteres no está delimitada). El valor por defecto es **NONE** y este especificador solo se puede usar para ficheros con entrada/salida formateada.

FILE= indica el nombre del fichero que va a ser conectado. En caso de no aparecer, la conexión será hecha con un fichero determinado por el ordenador.

FORM= **FORMATTED** (si el registro está formateado) o **UNFORMATTED** (si el registro está sin formatear). Si el fichero está conectado con acceso secuencial, el valor por defecto es **FORMATTED** y si el fichero está conectado con acceso directo, el valor por defecto es **UNFORMATTED**.

IOSTAT= da un valor entero positivo si hay alguna condición de error cuando ejecutamos la sentencia **OPEN** y cero si no hay error.

PAD= **YES** (indica que en caso de que la lista de entrada tenga una longitud menor que el registro que la contiene, se rellenará con espacios en blanco) o **NO** (indica que la lista de entrada contiene los datos que necesita). El valor por defecto es **YES**.

POSITION= **ASIS** (deja la posición del fichero invariante para un fichero ya conectado, y sin especificar para un fichero que no está conectado), **REWIND** (posiciona al fichero en el punto inicial) o **APPEND** (posiciona el fichero en el final o justo antes del registro fin de fichero). El valor por defecto es **ASIS**. Para usar este especificador el fichero debe estar conectado por acceso secuencial.

RECL= da un valor positivo que especifica la longitud de cada registro, si el método de acceso es directo, o la longitud máxima de los registros, si el acceso es secuencial.

STATUS= **OLD** (indica que el fichero ya existe), **NEW** (indica que el fichero no existe), **UNKNOWN** (indica que el estado depende del procesador), **REPLACE** (si el fichero no existe, se crea y se le da el status **OLD**, si el fichero existe, se borra, se crea otro con el mismo nombre y el estado se cambia a **OLD**), **SCRATCH** (se refiere a ficheros que existen hasta la terminación de la ejecución del programa o hasta la ejecución de una sentencia **CLOSE**).

Algunas reglas y restricciones para la sentencia **OPEN** son:



1. El número de una unidad externa tiene que estar siempre presente. Si la palabra **UNIT=** no está, el número de unidad debe ser el primer artículo de la lista.
2. El especificador **FILE=** debe aparecer si el **STATUS** es **OLD**, **NEW** o **REPLACE** y puede no aparecer si el **STATUS** es **SCRATCH**.

La sentencia CLOSE

La ejecución de la sentencia **CLOSE**, finaliza la conexión entre un fichero y una unidad. Algunas conexiones no son terminadas por la sentencia **CLOSE**, sino que las concluye el propio sistema operativo al concluir el programa. La forma de una sentencia **CLOSE** es:

CLOSE ([**UNIT** =] lista de propiedades de la desconexión)

donde las propiedades de la desconexión son: **IOSTAT** y **STATUS**.

Algunas reglas y restricciones para la sentencia **CLOSE** son:

1. El número de una unidad externa tiene que estar siempre presente. Si la palabra **UNIT** = no está, el número de unidad debe ser el primer artículo de la lista.
2. **STATUS** = toma los valores **KEEP** (si el fichero sigue existiendo después de cerrarlo) o **DELETE** (si el fichero deja de existir después de cerrar el fichero). El valor por defecto es **KEEP**, salvo que en la sentencia **OPEN** aparezca la variable **SCRATCH**.
3. Si la sentencia **CLOSE** hace referencia a una unidad que no está conectada o que no existe, esta sentencia no tiene efecto y no dará ninguna condición de error.
4. Las reglas para el especificador **IOSTAT** =, son las mismas que para la sentencia **OPEN**.
5. Las conexiones que ya hayan sido cerradas podrán volverse a abrir en el último punto en el que se ejecutó el programa.

La sentencia INQUIRE

La sentencia **INQUIRE** nos da información sobre la existencia, conexión y métodos de acceso a ficheros. La forma general de una sentencia **INQUIRE** es:

INQUIRE (lista de especificaciones)

Existen dos formas para realizar la investigación, una y otra son excluyentes:

Por la unidad: Para ello, incluimos en la lista de especificaciones [**UNIT=**]núm. de unidad.

Por el nombre: Para ello, incluimos en la lista de especificaciones **FILE=** nombre del fichero.

La lista de especificación de la sentencia **INQUIRE** está formada por las siguientes propiedades:



ACCESS = mismas características que para la sentencia **OPEN**.
ACTION = mismas características que para la sentencia **OPEN**.
BLANK = mismas características que para la sentencia **OPEN**.
DELIM = mismas características que para la sentencia **OPEN**.
DIRECT = nos devuelve el valor **YES** si el método de acceso directo está permitido, **NO** si el método de acceso directo no está permitido y **UNKNOWN** si el procesador no sabe si está permitido.
EXIST = variable lógica que nos indica si la unidad o el fichero existen o no.
FORM = nos devuelve el valor **FORMATTED** si el fichero está conectado con entrada/salida formateada, el valor **UNFORMATTED** si está sin formatear o el valor **UNDEFINED** si no está conectado.
FORMATTED = nos devuelve el valor **YES** si la entrada/salida formateada está permitida por el fichero, el valor **NO** si no está permitida y el valor **UNKNOWN** si el procesador no sabe si está permitida.
IOWSTAT = mismas características que para la sentencia **OPEN**.
NAME = nos devuelve el nombre del fichero, si tiene un nombre.
NAMED = variable lógica que nos indica si el fichero tiene nombre.
NEXTREC = el entero devuelto es uno más que el número del último registro leído o escrito si el acceso es directo.
NUMBER = nos devuelve el número de unidad conectada al fichero. Si no existe unidad conectada al fichero el valor devuelto es **-1**.
OPENED = si la investigación es por unidad, es una variable lógica que nos indica si la unidad está conectada a algún fichero. Si la investigación es por fichero, es una variable lógica que nos indica si el fichero está conectada a alguna unidad.
POSITION = mismas características que para la sentencia **OPEN**.

La sentencia INQUIRE

CONTINUAMOS CON LAS ESPECIFICACIONES DE LA SENTENCIA INQUIRE:

READ = nos devuelve el valor **YES** si la lectura es una acción permitida por el fichero, el valor **NO** si no lo es y el valor **UNKNOWN** si el procesador no lo sabe.
READWRITE = nos devuelve el valor **YES** si la lectura y escritura son acciones permitidas por el fichero, el valor **NO** si no lo son y el valor **UNKNOWN** si el procesador no lo sabe.
RECL = mismas características que para la sentencia **OPEN**.
SEQUENTIAL = nos devuelve el valor **YES** si el método de acceso secuencial está permitido, **NO** si el método de acceso secuencial no está permitido y **UNKNOWN** si el procesador no sabe si está permitido.
UNFORMATTED = nos devuelve el valor **YES** si la entrada/salida sin formatear está permitida por el fichero, el valor **NO** si no está permitida y el valor **UNKNOWN** si el procesador no sabe si está permitida.
WRITE = nos devuelve el valor **YES** si la escritura es una acción permitida por el fichero, el valor **NO** si no lo es y el valor **UNKNOWN** si el procesador no lo sabe.



PAD = nos da el valor **NO** si el fichero está conectado con el especificador **PAD = NO** y el valor **YES** en otro caso.

Por otro lado, la forma de la sentencia **INQUIRE** usada para calcular la longitud de una lista de salida es:

INQUIRE (IOLENGTH = variable entera) lista de salida
el valor de esa variable entera podrá ser usado después como el valor del especificador **RECL =** en una sentencia **OPEN**.

Sentencias de situación de ficheros

La ejecución de transferencias de datos suele cambiar la posición del fichero. Existen tres sentencias para cambiar la posición de un fichero externo que esté conectado por acceso secuencial, las sentencias **BACKSPACE** , **REWIND** y **ENDFILE**. La forma general de estas es:

BACKSPACE núm. de unidad o **BACKSPACE (lista de especificaciones de posición)**

REWIND núm. de unidad o **REWIND (lista de especificaciones de posición)**

ENDFILE núm. de unidad o **ENDFILE (lista de especificaciones de posición)**

La especificación de posición está compuesta por:

[**UNIT =**] núm. de unidad ; **IOSTAT =**

Si omitimos la palabra **UNIT =** este especificador debe ser el primero de la lista y siempre debe haber un especificador de unidad.

La sentencia BACKSPACE

La ejecución de esta sentencia provoca que el fichero se posicione antes del registro actual si existe un registro actual, o antes del registro precedente si no hay registro actual. En caso de no haber registro actual ni registro precedente, la posición del fichero no se moverá.

Si el fichero está todavía en el punto inicial, la sentencia **BACKSPACE** no tendrá ningún efecto.

La sentencia REWIND

Esta sentencia sitúa al fichero en el punto inicial. Esta sentencia no tendría ningún efecto si la posición del fichero ya fuera la inicial.

La sentencia ENDFILE



Esta sentencia escribe un registro fin de fichero y sitúa el fichero después del registro fin de fichero escrito. Escribir registros después del registro fin de fichero, está prohibido. Después de ejecutar una sentencia **ENDFILE** es necesario ejecutar una sentencia **BACKSPACE** o **REWIND** para posicionar al fichero antes del registro fin de fichero.

Formatos de entrada/salida

Normalmente los datos están almacenados en la memoria como valores de variables en forma binaria. Por otra parte, los registros de datos formateados en un fichero constan de caracteres. Luego, cuando los datos se leen desde un registro formateado, los caracteres deben convertirse a la representación interna de la máquina, así mismo, cuando los datos son escritos en un registro formateado, deben convertirse de la representación interna a cadenas de caracteres. Así, el especificador de formato me da la información necesaria para saber como tienen que ser tratadas estas conversiones. El especificador del formato, básicamente, es una lista de editores, uno para cada valor de la lista de entrada/salida de la sentencia de transferencia de datos.

Un especificador de formato se escribe como una cadena de caracteres. A esta forma de asignar formatos, se le llama formato explícito. También es posible usar el especificador de formato por defecto, esto es, con un *****. A esto se le llama formato implícito. Por otra parte, hay que tener en cuenta que toda información que aparece en el formato después del último paréntesis de la derecha, no tiene ningún efecto.

Especificadores de formato

Los artículos que forman un especificador de formato son los editores, bien editores de datos o editores de control. Los formatos son de una de las siguientes formas:

```
[ r ] editor de datos
editor de control
[ r ] ( lista de formatos )
```

donde **r** es un entero positivo sin parámetro de clase, llamado factor de repetición. Por ejemplo:

```
READ ( 5, " ( 2I3, F6.3 ) " ) n, m, x
PRINT ( *, * ) , " EL VALOR DE X ES: ", x
```

Sea **w** el ancho del campo, es decir, el número de caracteres transferidos desde, o al fichero, **m** el mínimo número de dígitos, **d** el número de decimales, **e** el número de dígitos en el exponente y **n** un entero positivo. Así, una tabla de editores de datos y de editores de control sería la siguiente:

EDITOR DE DATOS	TIPO DE DATOS	EDITOR DE	FUNCIÓN
-----------------	---------------	-----------	---------



		CONTROL	
I w [. m]	Entero decimal	T n	Situarse en la posición n.
B w [. m]	Entero binario	TL n	Situarse n posiciones a la izqda.
O w [. m]	Entero octal	TR n	Situarse n posiciones a la drcha.
Z w [. m]	Entero hexadecimal	[r] /	Pasar al siguiente registro.
F w . d	Real en forma posicional	:	Detener el formato al acabar la lista
E w .d [E e]	Real en forma exponencial	S	Escribir el signo mas por defecto
EN w .d [E e]	Real en forma de ingeniero	SP	Escribir el signo mas
ES w .d [E e]	Real en forma científica	SS	Suprimir el signo mas
L w	Lógico	BN	Ignorar los espacios en blanco
A [w]	Caracter ("alfabeto")	BZ	Convertir los blancos, en ceros

Transferencia de datos formateados

Cuando realizamos una transferencia de datos formateados, el siguiente elemento de la lista de entrada/salida es emparejado con el siguiente editor de datos que determina la conversión entre la representación interna de los datos y la cadena de caracteres en el registro formateado.

Los paréntesis en un especificador de formato

Algunas reglas sobre el uso de los paréntesis en un especificador de formato son:

1. Si después del paréntesis más a la derecha, no hay más datos, la ejecución de la entrada/salida finaliza.
2. Si después del paréntesis más a la derecha, hay más datos, el formato inicia la búsqueda del primer paréntesis derecho, y hacia atrás, los va emparejando con los izquierdos. Si no hay más paréntesis derechos que el



más extremo, el control del formato se situará en el primer paréntesis izquierdo, al principio de la especificación de formato.

3. Si se encuentra un factor de repetición volviendo a los paréntesis izquierdos, la repetición antes de un paréntesis es ignorada.
4. La interpretación de los signos de control y de los blancos, no afecta para el tratamiento de los paréntesis.

Editores numéricos

Los editores numéricos son, **I**, **B**, **O**, **Z**, **F**, **E**, **EN** y **ES**.

Las siguientes reglas se pueden aplicar a todos ellos:

En la entrada:

1. Los espacios en blanco no son significativos.
2. El signo mas (+) es omitido en la entrada de datos.
3. En los números con punto decimal y correspondientes a los editores **F**, **E**, **EN** y **ES**, el punto decimal en la entrada, no hace caso del punto decimal indicado por el editor.

En la salida:

1. Si el número el exponente es muy grande para el especificador del ancho del campo, la salida será de asteriscos (*).
2. Un valor interno positivo o cero tiene un signo mas, dependiendo del editor usado para el signo.
3. Los espacios en blanco son insertados.

Editores enteros

Los editores enteros son, **I**, **B**, **O** y **Z**.

El editor **I**, produce un entero decimal usando los dígitos del 0 al 9. El editor **B** produce un entero binario usando los dígitos 0 y 1. El editor **O** produce un entero octal usando los dígitos del 0 al 7. Y el editor **Z** produce un entero hexadecimal usando los dígitos del 0 al 9 y las letras de la **A** a la **F**.

En la entrada, la cadena de caracteres en el fichero debe ser una constante entera, con o sin signo, usando solamente los dígitos permitidos por el editor.

En la salida, el campo consta de blancos, seguidos, opcionalmente por un signo y por un entero sin signo. Al menos un dígito debe ser de salida, salvo que **m** sea cero y entonces el valor de salida será cero.

Editores reales



Los editores reales son, **F**, **E**, **EN** y **ES**.

El editor **F**, **Fw.d**, representa al campo como una cadena de w dígitos con d dígitos decimales ($w \leq d$). El editor puede tener signo. En la entrada, si el campo de entrada consta de un punto inicial, el valor de d no tiene efecto. Si no hay punto decimal, el punto decimal se inserta al lado derecho del dígito número d . Puede haber más dígitos en el número, que los que necesite el procesador. El número puede contener una **E**, indicando un exponente. En la salida, el número es un número real, con o sin signo, con d dígitos después del punto decimal. Si el número es menor que 1, el procesador puede situar un cero delante del punto decimal. Si el número no se encuentra en el campo de salida, el campo estará lleno de asteriscos (*).

El editor **E**, representa el campo como un número en punto flotante con w caracteres, (incluido el exponente) ($d + e \leq w$). En la entrada, la forma es la misma que el editor **F**, donde **E** puede indicar el exponente. En la salida, la forma del campo es:

`[±] [0] . x1, x2, . . . , xd exponente`

En el editor **EN**, la entrada es la misma que la del editor **F**, mientras que en la salida, el exponente es un número divisible por 3, y $1 \leq |\text{mantis}| \leq 1000$, excepto que la salida sea cero. La forma general es:

`[±] yyy.x1, x2, . . . , xd exponente`

En el editor **ES**, la entrada es la misma que la del editor **F**, mientras que la salida de un número es en notación científica. $1 \leq |\text{mantis}| \leq 10$, excepto que la salida sea cero. La forma general es:

`[±] y.x1, x2, . . . , xd exponente`

Editor complejo

El editor de números complejos necesita de dos editores reales, uno para la parte real y otro para la parte imaginaria. Se pueden usar editores distintos para las dos partes. Por ejemplo:

`WRITE (6 , '(E6.3, E5.2)') C`

Editor lógico

El editor lógico es **Lw**, siendo w el ancho del campo.

En la entrada, el campo para una variable lógica consta de unos blancos, seguidos por una **T** o una **F**, que representan a **TRUE** y **FALSE**, respectivamente.

La salida consta de $w-1$ espacios en blanco seguidos por una **T** o una **F**.

Editor de caracter



El editor para caracteres es **A[w]**, siendo *w* el ancho del campo medido en caracteres. Si omitimos *w*, la longitud de los datos se tomará como el ancho del campo.

En la entrada, sea *long* la longitud de los datos que leemos. Si *w* es mayor que *long*, los caracteres después de la posición *long* también son leídos. Si *w* es menor que *long*, a la entrada le añadiremos espacios en blanco por la derecha. En la salida, si *w* es mayor que *long*, se añaden espacios en blanco por la izquierda. Si *w* es menor que *long*, los caracteres antes de la posición *w* aparecerán en la salida.

Editores de posición

El editor **Tn** sitúa al registro antes del carácter *n*. El editor **TRn** mueve a la derecha *n* caracteres y el editor **TLn** mueve a la izquierda *n* caracteres.

Editor (/)

Cuando encontramos un editor **(/)**, en la especificación de formato, entonces el registro actual finalizará su ejecución.

En la entrada, si el fichero está conectado por acceso secuencial, el fichero se situará al principio del siguiente registro. Si el fichero está conectado por acceso directo, el número de registro se incrementará en uno.

En la salida, si el fichero está conectado por acceso secuencial, el fichero se situará al principio de un nuevo registro. Si el fichero está conectado por acceso directo, el número de registro se incrementará en uno. Y este nuevo registro pasará a ser el registro actual.

Editor (:)

Si la lista de términos en el formato de las sentencias **READ** o **WRITE** se acaba, el editor **(:)** para el proceso de formato en este punto. Claramente, esto no tiene ningún efecto si no existen más datos.

Formato por defecto

Este formato se selecciona usando un ***** en vez de un especificador de formato explícito, en las sentencias **READ**, **WRITE** o **PRINT**. Por ejemplo:

```
WRITE ( 6, * ) N, M
```

Algunas reglas y restricciones sobre el formato por defecto son:



1. Si no hay términos en la lista de variables, dependiendo de la sentencia que usemos, escribiremos un registro de salida vacío, o saltaremos un registro de entrada.
2. El formato por defecto no se puede usar con acceso directo, ni con entrada/salida sin avanzar.
3. Los registros constan de valores y de separadores de valores.

Valores: Los valores permitidos son:

`nulo` : no hay valores entre los separadores

`c` : constante sin signo y sin parámetro de clase

`r*c` : `r` repeticiones de la constante `c` (`r` es una cadena de dígitos)

`r*` : `r` repeticiones del valor nulo (`r` es una cadena de dígitos)

Separadores: Los separadores permitidos son:

(,) Una coma. (/) Una barra. Un espacio en blanco entre dos valores no blancos

Entrada: Los valores permitidos para la entrada son los mismos que los permitidos para formato explícito. Sin embargo, existen algunas excepciones, estas son:

1. Los editores para espacios en blanco no están permitidos, y por tanto, los blancos nunca pueden ser cero. Además, solo están permitidos dentro de constantes caracter y antes o después de una coma.
2. Las cadenas de caracteres deben aparecer entre comillas o entre apóstrofes. Los separadores de valores pueden ser valores representables en la cadena.
3. Los complejos no son tratados como dos reales. $(4.2, 3.3) \equiv 4.2 + 3.3*i$
4. Los términos lógicos no pueden usar separadores de valores seguidos por los valores **T** o **F**.
5. Si `long` es la longitud del siguiente término de la lista de entrada y `w` es la longitud de una constante caracter en la entrada. Entonces, si:
`long ≤ w`, los `long` caracteres de la constante más a la izquierda son usados.
`long > w`, los `w` caracteres de la constante son usados y el campo se rellena con espacios en blanco por la derecha.
6. El tipo debe estar de acuerdo con el siguiente término de la lista.

Valores nulos: Los valores nulos aparecen cuando no hay valores entre separadores, cuando un registro comienza con un separador de valores, o cuando usamos el valor `r*`.

Salida: Los valores permitidos para la salida son los mismos que los permitidos para formato explícito. Las únicas excepciones son, que los caracteres en blanco y el símbolo (/) no son valores de salida.



Formato por defecto para los distintos tipos de variables

Entero:

El resultado es el mismo que si usase el editor **Iw**.

Real:

El resultado es el mismo que si usase los editores **F** o **E**. La salida depende de la magnitud del número, dependiendo de esta, el procesador escogerá una u otra opción.

Complejos:

La parte real y la imaginaria están encerradas entre paréntesis y separadas por una coma. Si la longitud del número complejo es más largo que el registro, el procesador debe separar la parte real y la imaginaria en dos registros.

Lógicos:

La salida por defecto para las variables lógicas es **T** o **F**, dependiendo de si la variable es **TRUE** o **FALSE**, respectivamente.

Caracter:

Los formatos de las variables caracter están supeditados a los valores del especificador **DELIM =** de la sentencia **OPEN**. Si **DELIM = NONE**, entonces:

1. Las constantes caracter no están delimitadas.
2. Las constantes caracter no están rodeadas por separadores de valores
3. Se inserta un espacio en blanco en los nuevos registros como sentencia de control, para continuar la constante caracter.

Si por el contrario, **DELIM = QUOTE** o **DELIM = APOSTROPHE**, entonces:

1. Las constantes caracter están delimitadas.
2. Las constantes caracter están rodeadas por separadores de valores.
3. Los parámetros de clase están permitidos.
4. No se inserta un espacio en blanco en los nuevos registros como sentencia de control, para continuar la constante caracter.