# Fortran 90 - A thumbnail sketch

Michael Metcalf
*CERN, Geneva, Switzerland.*

**Abstract**

The main new features of Fortran 90 are presented.
**Keywords:** Fortran

## 1 New features

In this brief paper, we describe in outline the main new features of Fortran 90 as compared to its predecessor, FORTRAN 77. Full details are given in [1]. Compilers for Fortran 90 are now available on all major systems, further information being accessible on the World Wide Web at the URL `http://www.fortran.com/fortran/market.html`.

### 1.1 Source form

The new source form allows free form source input, without regard to columns. Comments may be in-line, following an exclamation mark (!), and lines which are to be continued bear a trailing ampersand (&). The character set is extended to include the full ASCII set, including lower-case letters. The underscore character is accepted as part of a name which may contain up to 31 characters. Thus, one may write a snippet of code such as

```
SUBROUTINE CROSS_PRODUCT (x, y, z) ! Z = X*Y
:
z1 = x(2) * y(3) -              &
x(3) * y(2)
```

Blanks are significant in the new source form, and may not be embedded inside tokens, such as keywords and variable names.

### 1.2 Alternative style of relational operators

An alternative set of relational operators is introduced. They are

```
<      for .LT.        >      for .GT.
<=     for .LE.        >=     for .GE.
==     for .EQ.        /=     for .NE.
```

enabling us to write statements such as

```
IF (x < y .AND. z**2 >= radius_squared) THEN
```

### 1.3 Specification of variables

The existing form of type declaration is extended to allow all the attributes of the variables concerned to be declared in a single statement. For instance, the statement

```
REAL, DIMENSION(5,5), PARAMETER :: a = (/ (0., i = 1, 25) /),   &
                                   b = (/ ( i, i = 1, 25) /)
```

declares the two objects `a` and `b` to be named constant arrays, whose values are specified by the array constructors (see Section 1.12) following the equals signs.

For those who wish to define explicitly the type of all variables, the statement

```
IMPLICIT NONE
```

turns off the usual implicit typing rules.

## 1.4 Parameterized data types

All the intrinsic data types are now parameterized. The way this is done is best explained by examples. Suppose a given processor gives access to two *kinds* of integers — a default kind, represented by four bytes, and a short kind represented by two bytes. The processor might designate the default kind by a *kind type value* of 4, and the short kind by a value of 2. This value can be associated with the *kind type parameter* of a given variable as shown:

```
INTEGER(KIND=4) :: i      ! same as default
INTEGER(KIND=2) :: j      ! half the size
```

Constants corresponding to these kinds may be written. The value 99 might also be written as 99_4; a short integer constant is 63_2. It is, of course, not very portable to use explicit kind type values, and they can be parameterized, in this example for instance as

```
INTEGER, PARAMETER :: short=2
INTEGER(short)     :: j
:
j = 63_short
```

A similar, optional facility is available for logical entities which might pack down into bits or bytes, depending on the value of the kind type parameter, and for character entities, in order to accommodate non-European languages whose characters require more than one byte for their representation, e.g. Kanji.

In the case of reals, at least two kinds must be provided, corresponding to the present single- and double-precision features. The kinds can be defined in terms of a range and/or precision, allowing a reasonable measure of portability. To define an array whose elements have a range of at least $10^{\pm 99}$ and a precision of at least nine decimal digits, we might write

```
INTEGER, PARAMETER    :: long = SELECTED_REAL_KIND(9, 99)
REAL(long), DIMENSION :: a(1024)
a(1) = 1.23456789_long
```

where the intrinsic function SELECTED_REAL_KIND returns the kind value most closely but completely corresponding to the desired precision and range. This facility is of great importance for portable numerical software. Other intrinsic functions are available as part of this facility.

## 1.5 Binary, octal and hexadecimal constants

The standard allows binary, octal and hexadecimal constants to be used in DATA statements, and integers to be read and written in the corresponding formats:

```
INTEGER :: i, j
DATA j / Z'abcd'/
:
WRITE (unit, '(Z4)' ) i
```

## 1.6 The POINTER attribute

Pointers have been added to Fortran in the form of an *attribute*, (rather than as a separate data type). The specification of an object with the pointer attribute designates a descriptor of the object, rather than the object itself. No storage is reserved — it has to be allocated explicitly.

A pointer may reference any dynamic object, such as an array whose actual bounds are defined at run-time, or any static object defined with the TARGET attribute. The use of the name of the pointer is, in most contexts, such as expressions or I/O lists, interpreted as a reference to the value of the target. This is known as *automatic dereferencing, i.e.* no explicit notation using a

pointer symbol is required. This means that existing code can readily be modified to use pointers, as only the specification statements need to be changed. Simple examples are:

```
REAL, POINTER                  :: arrow ! may point to a real scalar
REAL, DIMENSION(:,:), POINTER :: pfeil ! may point to a 2D array
:
ALLOCATE(pfeil(10, 10))                ! allocate storage to pfeil
:                                      ! define parts of pfeil
pfeil(1,1) = pfeil(5,5) + pfeil(10,10) ! the target values are used
```

Pointers are strongly typed, that is, they may point only at objects of the same type and type parameters as the ones with which they themselves are defined. Sometimes, of course, it is necessary to change the value of a pointer. For this case, a *pointer assignment* is foreseen. An example is shown in

```
   REAL, DIMENSION(:,:), POINTER :: a, b, c
   :
   ALLOCATE (a(10,10), b(10,10)) ! give actual space to a and b
   :
! define a and b
   :
   c => a                        ! c now points to the array a
   DO i = 1, 2
      IF (i == 2) c => b         ! change target of c, no copy of data
   :                             ! on first pass, c refers to a
   :                             ! on second pass to b
! operations on c
   :
   END DO
   :
```

Pointers may be components of a structure (see Section 1.15), and of the same type as the structure. In other words, a recursive definition is allowed which enables structures representing lists, trees and graphs to be defined. An example is

```
   TYPE pixel
      REAL x, y
      INTEGER colour
      TYPE (pixel), POINTER :: next
   END TYPE pixel
```

Pointers become associated with a target by execution of a pointer assignment or of an ALLOCATE statement. They become disassociated by, in general, execution of a NULLIFY statement. Storage occupied by a target may be returned to the processor by execution of a DEALLOCATE statement; this can result in a pointer being left 'dangling', and programmers must guard against their further use until they are newly associated. An intrinsic inquiry function, ASSOCIATED, indicates whether an association exists between a pointer and its target, or whether a pointer is associated at all.

Pointers may be used for static objects. In order that this should not inhibit optimization, any static object which might become a pointer target must be declared with the TARGET attribute. The compiler then knows which assumptions it can and cannot make about the references to static objects.

## 1.7 The CASE construct

The CASE construct allows the execution of zero or one block of code, selected from several blocks, depending on the value of an integer, logical, or character expression as compared against a set of constant values. An example is

```
SELECT CASE(3*i-j)    ! integer expression
CASE(0)               ! for 0  (constant value)
 .                    ! executable code
CASE(2,4:8)           ! for 2, 4 to 8 (constant values)
 .                    ! executable code
CASE DEFAULT          ! for all other values
 .                    ! executable code
END SELECT
```

The default clause is optional; overlapping ranges are not permitted.

## 1.8 The loop construct

A new loop construct is introduced. In a simplified form, its syntax is

```
[name:] DO [(control)]
        block of statements
        END DO [name]
```

(where square brackets indicate optional items). If the control parameter is omitted, an endless loop is implied; if present, it has the form *i = intexp1, intexp2* [*, intexp3*]. The optional name may be used in conjunction with CYCLE and EXIT statements to specify which loop in a set of nested loops is to begin a new iteration or which is to be terminated, respectively.

The DO...WHILE has also been incorporated into the language. It is redundant, as the DO construct and the EXIT statement provide the same functionality.

## 1.9 Program units

An enhanced form of the call to a procedure allows *keyword* and *optional* arguments, with *intent* attributes. For instance, a subroutine beginning

```
SUBROUTINE solve (a, b, n)
OPTIONAL, INTENT(IN) :: b
```

might be called as

```
CALL solve (n = i, a = x)
```

where two arguments are specified in keyword (rather than positional) form, and the third, which may not be redefined within the scope of the subroutine, is not given in this call. The mechanism underlying this form of call requires an *interface block* containing the relevant argument information to be specified.

Procedures may be specified to be recursive, as in

```
RECURSIVE FUNCTION factorial(x)
```

The old form of the statement function is generalized to an *internal procedure* which allows more than one statement of code, permits variables to be shared with the host procedure, and contains a mechanism for extending operators and assignment for use with derived-data types. This we shall return to in Section 1.16.

## 1.10 Extensions to CHARACTER data type

A number of extensions to the existing CHARACTER data type, in addition to the optional kinds, permit the use of strings of zero length and the assignment of overlapping substrings, and introduce new intrinsic functions, such as TRIM, to remove the trailing blanks in a string. Some intrinsics, including INDEX, may operate, optionally, in the reverse sense.

## 1.11 Input/Output

The work in the area of input/output has been mainly on extensions to support the new data types, and an increase in the number of attributes which an OPEN statement may specify, for instance to position a file or to specify the actions allowed on it.

In addition, there are new specifiers on READ and WRITE statements. One is the ADVANCE specifier for formatted, sequential files, allows data to be read and written without being concerned with any record structure. It is, for instance, possible to read a specified number of characters from part of a record, to remain positioned at that point in the record, and to read from the remainder with a subsequent read. At present, a single read would cause the whole record to be read, regardless of the length of the input list.

## 1.12 Array processing

The array processing features are one of the most important new aspects of the language.

An array is defined to have a *shape* given by its number of dimensions, or *rank*, and the *extent* of each dimension. Two arrays are conformable if they have the same shape. The operations, assignments and intrinsic functions are extended to apply to whole arrays on an element-by-element basis, provided that when more than one array is involved they are all conformable. When one of the variables involved is a scalar rather than an array, its value is distributed as necessary. Thus we may write

```
REAL, DIMENSION(5, 20)    :: x, y
REAL, DIMENSION(-2:2, 20) :: z
:
z = 4.0 * y * SQRT(x)
```

In this example we may wish to include a protection against an attempt to extract a negative square root. This facility is provided by the WHERE construct:

```
WHERE (x >= 0.)
   z = 4.0 * y * SQRT(x)
ELSEWHERE
   z = 0.
END WHERE
```

which tests `x` on an element-by-element basis.

A means is provided to define *array sections*. Such sections are themselves array-valued objects, and may thus be used wherever an array may be used, in particular as an actual argument in a procedure call. Array sections are selected using a triplet notation. Given an array

```
REAL, DIMENSION(-4:0, 7) :: a
```

the expression `a(-3,:)` selects the whole of the second row (i.e. the elements a(-3,1),... a(-3,7)), and `a(0:-4:-2, 1:7:2)` selects in reverse order every second element of every second column.

Just as variables may be array-valued, so may constants. It is possible to define a rank-one array-valued constant as in

```
      (/1, 1, 2, 3, 5, 8/)
```
and to reshape it to any desired form:
```
      REAL, DIMENSION(2, 3) :: a
      a = RESHAPE( (/ (I, I = 1, 6) /), (/2, 3/) )
```
where the second argument to the intrinsic function defines the shape of the result, and the first defines an array of the first six natural numbers, using the implied-DO loop notation familiar in I/O lists.

Many vector processors have a gather/scatter capability. This can be directly accessed in Fortran 90 by use of the vector-valued subscript notation:
```
      INTEGER, DIMENSION(1024) :: i, j
      REAL, DIMENSION(1024)    :: a, b
      :
  ! Define i, j and a
      b(j) = a(i)              ! permute the elements of a into b
```

## 1.13 Dynamic storage

Fortran 90 provides three separate mechanisms for accessing storage dynamically. The first mechanism is via the ALLOCATE and DEALLOCATE statements which, as their names imply, can be used to obtain and return the actual storage required for an array whose type, rank, name, and allocatable attribute have been declared previously:
```
      REAL, DIMENSION(:), ALLOCATABLE :: x
      :
      ALLOCATE(x(n:m)) ! n and m are integer expressions
      :
      x(j) = q
      CALL sub(x)
      :
      DEALLOCATE (x)
```
Unless an allocated array has the SAVE attribute, it becomes undefined whenever a RETURN or END statement is executed in the procedure in which it is declared. For good housekeeping, such arrays should be explicitly deallocated. The fact that allocation and deallocation can occur in random order implies an underlying heap storage mechanism.

The second mechanism, for local arrays with variable dimensions, is the *automatic* array:
```
      SUBROUTINE sub(i, j, k)
      REAL, DIMENSION(i, j, k) :: x ! bounds from dummy arguments
```
whose actual storage space is provided (on a stack) when the procedure is called.

Finally, we have the *assumed-shape* array, whose storage is defined in a calling procedure (so it is not strictly dynamic), and for which only a type, rank, and name are supplied:
```
      SUBROUTINE sub(a)
      REAL, DIMENSION(:, :, :) :: a
```
Various enquiry functions may be used to determine the actual bounds of the array:
```
      DO i = LBOUND (a, 3), UBOUND (a, 3)
         DO j = LBOUND (a, 2), UBOUND (a, 2)
            DO k = LBOUND (a, 1), UBOUND (a, 1)
```
where LBOUND and UBOUND give the lower and upper bounds of a specified dimension, respectively.

## 1.14 Intrinsic procedures

Fortran 90 defines about 100 intrinsic procedures. Many of these are intended for use in conjunction with arrays for the purposes of reduction (e.g. SUM), inquiry (e.g. RANK), construction (e.g. SPREAD), manipulation (e.g. TRANSPOSE), and location (e.g. MAXLOC). Others allow the attributes of the working environment to be determined (e.g. the smallest and largest positive real and integer values), and access to the system and real-time clocks is provided. A random number subroutine provides a portable interface to a machine-dependent sequence, and a transfer function allows the contents of a defined area of physical storage to be transferred to another area without type conversion occurring.

Of particular note is that the bit string manipulation functions and subroutine of MIL-STD 1753 have been incorporated into the language; their definitions have been extended to handle integer arrays in a completely upwards-compatible fashion.

## 1.15 Derived-data types

Fortran has hitherto lacked the possibility of building user-defined data types. This will be possible in Fortran 90, using a syntax illustrated by the example

```
TYPE staff_member
   CHARACTER(LEN=20) :: first_name, last_name
   INTEGER           :: id, department
END TYPE
```

which defines a structure which may be used to describe an employee in a company. An aggregate can be defined as

```
TYPE(staff_member), DIMENSION(1000) :: staff
```

defining 1000 such structures to represent the whole staff. Individual staff members may be referenced as, for example, `staff(no)`, and a given field of a structure as `staff(no)%first_name`, for the first name of a particular staff member. More elaborate data types may be constructed using the ability to nest definitions, for instance, to build a structure to define companies:

```
TYPE company
   CHARACTER(LEN=20)                    :: name
   TYPE(staff_member), DIMENSION(1000) :: staff
END TYPE
TYPE(company), DIMENSION(20)           :: companies
```

## 1.16 Data abstraction

It is possible to define a derived-data type, and operations on that data type may be defined in an internal procedure. These two features may be combined into a module which can be propagated through a whole program to provide a new level of data abstraction (see also [2]). As an example we may take an extension to Fortran's intrinsic CHARACTER data type whose definition is of a fixed and pre-determined length. This user-defined derived-data type defines a set of modules to provide the functionality of a variable length character type, which we shall call `string`. The module for the type definition might be

```
MODULE string_type
   TYPE string(maxlen)
      INTEGER               :: length
      CHARACTER(LEN=maxlen) :: string_data
   END TYPE String
```

```
END MODULE String_type
```
With
```
USE string_type
TYPE(string(60)), DIMENSION(10) :: cord
```
we define an array of 10 elements of maximum length 60. An actual element can be set by
```
cord(3) = 'ABCD'
```
but this implies a redefinition, or *extension*, of the assignment operator to define correctly both fields of the element. This can be achieved by the internal procedure
```
SUBROUTINE c_to_s_assign(s,c)
   TYPE (string)    :: s
   CHARACTER(LEN=*) :: c
   s%string_data = c
   s%length      = LEN(c)
END SUBROUTINE c_to_s_assign
```
and the interface block
```
INTERFACE ASSIGNMENT(=)      ! Extend assignment operator
   MODULE PROCEDURE c_to_s_assign
END INTERFACE
```
which can both be included in the module, together with other valid functions such as concatenation, length extraction, etc., to allow the user-defined string type to be imported into any program unit where it may be required, in a uniformly consistent fashion.

## 2 Backwards compatibility

The procedures under which standards are developed require that a period of notice be given before any existing feature is removed from the language. This means, in practice, a minimum period of one revision cycle, which for Fortran means a decade or more. The need to remove features is evident: if the only action of the standards committee is to add new features, the language will become grotesquely large, with many overlapping and redundant items. The solution adopted was to publish as an Appendix to the standard a set of two lists showing which items have been removed or are candidates for eventual removal.

The first list contains the Deleted Features (those which in the previous standard were listed as Obsolescent Features) which have now been removed. There are no Deleted Features in Fortran 90 which contains the whole of FORTRAN 77.

The second list contains the Obsolescent Features, those considered to be redundant and little used, and which should be removed in the next revision (although that is not binding on a future committee). The Obsolescent Features are:

| | |
|---|---|
| Arithmetic IF | Real and double precision DO variables |
| Shared DO termination | DO termination not on CONTINUE (or ENDDO) |
| Alternate RETURN | Branch to ENDIF from outside block |
| PAUSE | ASSIGN and assigned GOTO |

## References


[1] M.Metcalf and J.Reid, *Fortran 90 Explained*, Oxford U. Press, Oxford (1990).

[2] M.Metcalf, "Abstract data types in Fortran 90", *CERN/CN/95/1*, (1995).