The Key Features of Fortran 95

Ninety-Five Key Features of Fortran 95

Jeanne C. Adams
Walter S. Brainerd
Jeanne T. Martin
Brian T. Smith

The Fortran Company

Library of Congress Catalog Card Number

Copyright © 1994-2006 by Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, and Brian T. Smith. All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of either the printed or electronic versions of this book may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the authors and the publisher.

Version 20051122

ISBN

The Fortran Company
6025 N. Wilmot Road
Tucson, Arizona 85750 USA
+1-520-760-1397
info@fortran.com

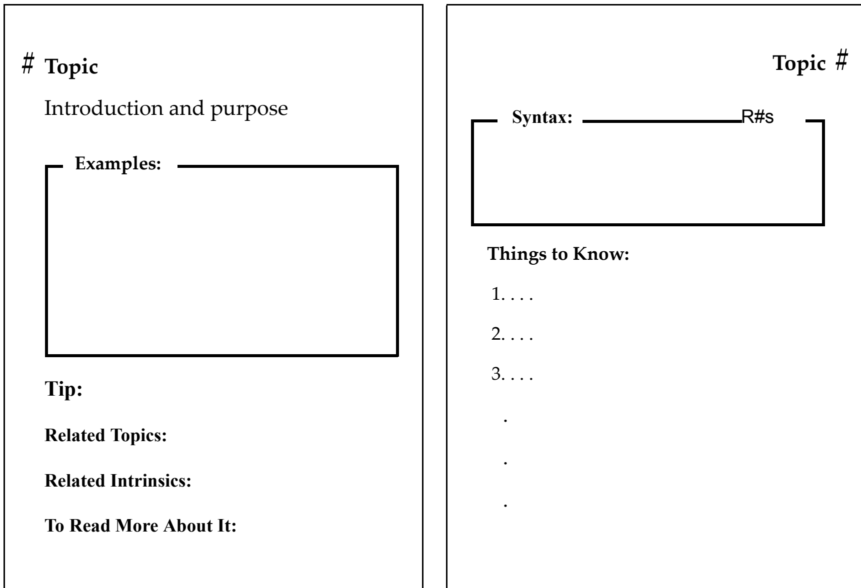
Composition by The Fortran Company

Preface

This guide is intended as a handy quick reference to the 95 features of Fortran 95 that are the most important for contemporary applications of Fortran. Although it is intended to be comprehensive and self-contained, many details are omitted; for completeness each topic contains relevant specific references to the Fortran 95 standard, the comprehensive *Fortran 95 Handbook*, and the *Fortran 95 Using F*.

This quick reference displays each feature in a left-right two-page layout, for a total of 190 pages plus appendices and index.

The normal left-hand page format has an introduction and purpose section, a number of examples, references, and in some cases a tip regarding use of that feature. The right-hand page contains a summary of the syntax and semantics for that feature, including many key "things to know" about it. In some cases the syntax shown has been simplified. For example, sometimes this is done for declaration statements where only one specification is indicated but several, separated by commas, are permitted.



A more appropriate format was used for a few of the 95 topics such as the overviews.

Fortran Top 95—Ninety-Five Key Features of Fortran 95

The electronic version has hypertext links in several contexts:

1. Each of the 95 topics has a link to it in the Bookmark section.
2. Each intrinsic procedure has a link to it in the Bookmark section.
3. The Bookmark section contains links to the List of Topics, the appendix containing the intrinsic procedures, and the index.
4. Each entry in the List of Topics contains a link to the topic.
5. Each Related Topic is linked to the topic.
6. Each Related Intrinsic is linked to the description of the intrinsic procedure in the appendix.
7. Each index entry page number is linked to the appropriate text.
8. Each link to a reference in the book *Fortran 95 Using F* will be active provided that book is available in the same directory.

Selecting any of these will display the corresponding material. Selecting the **back** button will reverse (undo) the link.

Selecting the **topics** button will display a list of all 95 topics, and any topic can be selected from this list. Similarly, selecting the **index** button will bring up the index, which can be scrolled and any entry selected. Selecting an item from either the topic list or index list reinitializes the hypertext browsing path.

The topics are in alphabetical order. Similarly the intrinsic procedures of Appendix A are in alphabetical order. A short example of a complete Fortran 95 application appears at the end of the book (on and inside the back cover of the printed version).

The authors hope that users will find this quick reference to be a handy and useful, if not indispensable, tool in working with Fortran 95.

Jeanne Adams
Walt Brainerd
Jeanne Martin
Brian Smith

2004 May

Topics

1 ALLOCATABLE Attribute and Statement 2
2 ALLOCATE and DEALLOCATE Statements 4
3 Argument Association 6
4 Argument Keywords 8
5 Array Overview 10
6 Array: Constructors 12
7 Array: Data-Parallel Operations 14
8 Array: Declaration Forms 16
9 Array: Sections 18
10 Assignment 20
11 CASE Construct 22
12 Character Substring 24
13 Character Type and Constants 26
14 CLOSE Statement 28
15 COMMON Statement 30
16 Complex Type and Constants 32
17 Data Initialization 34
18 Data Representation Models 36
19 Defined Operators and Assignment 38
20 Defined Type: Default Initialization 40
21 Defined Type: Definition 42
22 Defined Type: Objects 44
23 Defined Type: Structure Component 46

24 Defined Type: Structure Constructor 48
25 DIMENSION Attribute and Statement 50
26 DO Construct 52
27 Dynamic Objects 54
28 Edit Descriptors: Control 56
29 Edit Descriptors: Data and Character String 58
30 Elemental Procedures 60
31 EQUIVALENCE Statement 62
32 Expressions 64
33 Expressions: Initialization 66
34 Expressions: Specification 68
35 EXTERNAL Attribute and Statement 70
36 Files and Records 72
37 File Positioning Statements 74
38 FORALL Construct and Statement 76
39 Format Specifications 78
40 Functions 80
41 Generic Procedures and Operators 82
42 Going Against the Flow 84
43 Host Association 86
44 IF Construct and Statement 88
45 Implicit Typing 90
46 INCLUDE Line 92
47 INQUIRE Statement 94

Fortran Top 95—Ninety-Five Key Features of Fortran 95

48 Integer Type and Constants 96

49 INTENT Attribute and Statement 98

50 Interfaces and Interface Blocks 100

51 Internal Procedures 102

52 INTRINSIC Attribute and Statement 104

53 Intrinsic Function Overview 106

54 Intrinsic Functions: Array 108

55 Intrinsic Functions: Computation 110

56 Intrinsic Functions: Conversion 112

57 Intrinsic Functions: Inquiry and Model 114

58 Intrinsic Subroutines 116

59 Kind Parameters 118

60 Language Evolution 120

61 Logical Type and Constants 122

62 Main Program 124

63 Modules 126

64 Module Procedures 128

65 OPEN Statement 130

66 OPTIONAL Attribute and Statement 132

67 PARAMETER Attribute and Statement 134

68 Pointers 136

69 Pointer Association 138

70 POINTER Attribute and Statement 140

71 Pointer Nullification 142

Fortran Top 95—Ninety-Five Key Features of Fortran 95

72 Portable Precision Control 144

73 Program Units 146

74 PUBLIC and PRIVATE Attributes and Statements 148

75 Pure Procedures 150

76 READ/WRITE General Form 152

77 READ/WRITE: Direct Access Formatted 154

78 READ/WRITE: Direct Access Unformatted 156

79 READ/WRITE: Internal Files 158

80 READ/WRITE: List-directed 160

81 READ/WRITE: Namelist 162

82 READ/WRITE: Sequential Formatted Advancing 164

83 READ/WRITE: Sequential Formatted Nonadvancing 166

84 READ/WRITE: Sequential Unformatted 168

85 Real Type and Constants 170

86 Recursion 172

87 SAVE Attribute and Statement 174

88 Scope, Association, and Definition Overview 176

89 Source Form 178

90 Storage Association 180

91 Subroutines 182

92 TARGET Attribute and Statement 184

93 USE Statement and Use Association 186

94 Variables 188

95 WHERE Construct and Statement 190

Fortran Top 95—Ninety-Five Key Features of Fortran 95

1 ALLOCATABLE Attribute and Statement

The ALLOCATABLE attribute or statement is used to declare an array whose extents in each dimension will be specified by the user at runtime. Thus allocatable arrays are dynamic arrays; only the rank is declared. The bounds are colons indicating a deferred shape that can be specified in an executable ALLOCATE statement.

With dynamic arrays and the ALLOCATE and DEALLOCATE statements, the user is freed from concerns about determining maximum sizes for arrays during the construction of a program and, with the same features, is given more flexibility to utilize memory during execution; this is particularly valuable if memory space is tight.

Examples:

```
INTEGER, ALLOCATABLE :: MILES(:)  ! MILES is deferred shape.
ALLOCATE ( MILES (3) )            ! Allocate 3 elements.
      . . .
DEALLOCATE (MILES)                ! MILES is no longer allocated.
   . . .
ALLOCATE (MILES (-N:N))           ! Allocate with different
                                  !   extents.

REAL X, Y
COMPLEX CAT
DIMENSION X(:,:,:), CAT(:,:,:), Y(:)
ALLOCATABLE X, Y, CAT
INTEGER M, N, P
   . . .
ALLOCATE (CAT(2,2,3), STAT=IS)    ! CAT is allocated
   . . .                          !   12 spaces (2 x 2 x 3).

READ *, M, N, P
ALLOCATE (X(M,N,P),  &            ! M*N*P elements allocated.
          Y(M*N**P))              ! M*N**P elements allocated.
```

Related Topics:

ALLOCATE and DEALLOCATE Statements
Dynamic Objects

Related Intrinsics:

ALLOCATED (ARRAY)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.9, 5.2.6
Fortran 95 Handbook, 2.3.4, 5.3.3, 6.5
Fortran 95 Using F, 4.1.3

ALLOCATABLE Attribute and Statement

Syntax:

A type declaration statement with the ALLOCATABLE attribute is:

type , ALLOCATABLE [, *attribute-list*] :: *entity-list*

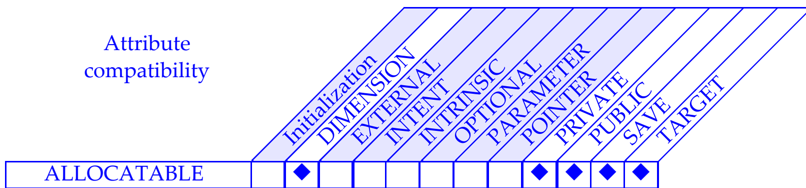
An ALLOCATABLE statement is:

ALLOCATABLE [::] *array-name* [(*deferred-shape-spec-list*)]

A deferred-shape specification is a colon (:).

Things To Know:

1. An allocatable array must not be a component of a defined type and must not appear in a common block.
2. An allocatable array must not be a dummy argument or a function result.
3. Space is not reserved for an allocatable array until an ALLOCATE statement is executed; the space remains allocated until a DEALLOCATE statement is executed.
4. If an allocatable array is not specifically deallocated, it is deallocated automatically when an END or RETURN statement is executed in the program unit that allocates the array unless the allocatable array has the SAVE attribute or is in a module that is being referenced by an active program unit.
5. Other dynamic objects are pointers and automatic data objects.
6. Although pointers provide more functionality, allocatable arrays are simpler and provide more opportunities for compiler optimization.



2 ALLOCATE and DEALLOCATE Statements

The ALLOCATE statement creates space for allocatable arrays and variables with the POINTER attribute. The DEALLOCATE statement frees space previously allocated for allocatable arrays and pointer targets. These statements give the user the ability to manage space dynamically at execution time.

Examples:

```
COMPLEX, POINTER :: HERMITIAN (:, :)    ! Complex array pointer
READ *, M, N
ALLOCATE ( HERMITIAN (M, N) )
   . . .
DEALLOCATE (HERMITIAN, STAT = IERR7)

REAL, ALLOCATABLE :: INTENSITIES(:,:)  ! Rank-2 allocatable array
DO
   ALLOCATE (INTENSITIES (I, J), &      ! IERR4 will be positive
             STAT = IERR4)              !   if there is
   IF (IERR4 == 0) EXIT                 !   insufficient space.
   I = I/2; J = J/2
END DO
   . . .
IF (ALLOCATED (INTENSITIES)) DEALLOCATE (INTENSITIES)

TYPE NODE
   REAL VAL
   TYPE(NODE), POINTER :: LEFT, RIGHT  ! Pointer components
END TYPE NODE
TYPE(NODE) TOP, BOTTOM
   . . .
ALLOCATE (TOP % LEFT, TOP % RIGHT)
IF (ASSOCIATED (BOTTOM % RIGHT)) DEALLOCATE (BOTTOM % RIGHT)

CHARACTER, POINTER :: PARA(:), KEY(:) ! Pointers to char arrays
ALLOCATE (PARA (1000) )
   . . .
KEY => PARA (K : K + LGTH)
```

Related Topics:

ALLOCATABLE Attribute and Statement
Dynamic Objects
Pointers
Pointer Association
POINTER Attribute and Statement
Pointer Nullification

Related Intrinsics:

ALLOCATED (ARRAY)
ASSOCIATED (POINTER, TARGET)
NULL (MOLD)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 6.3.1, 6.3.3
Fortran 95 Handbook, 6.5.1, 6.5.3
Fortran 95 Using F, 4.1.5, 8.1.3

Syntax:

An ALLOCATE statement is:

ALLOCATE (*allocation-list* [, STAT = *scalar-integer-variable*])

An allocation is:

allocate-object [(*allocate-shape-spec-list*)]

An allocate object is one of:

variable-name
structure-component

An allocate shape specification is:

[*lower-bound* :] *upper-bound*

A DEALLOCATE statement is:

DEALLOCATE (*allocate-object-list* [, STAT = *scalar-integer-variable*])

Things To Know:

1. Each allocate object must be an allocatable array or a pointer; the bounds in the shape specification must be scalar integer expressions.
2. The status variable (the variable following STAT=) is set to a positive value if an error is detected and is set to zero otherwise. If there is no status variable, the occurrence of an error causes the program to terminate.
3. For allocatable arrays, an error occurs when there is an attempt to allocate an already allocated array or to deallocate an array that is not allocated. The ALLOCATED intrinsic function may be used to determine whether an allocatable array is allocated.
4. It is not an error to allocate an associated pointer. Its old target connection is replaced by a connection to the newly allocated space. If the previous target was allocated and no other pointer became associated with it, the space is no longer accessible. A pointer may be assigned to point to a portion of an allocated object such as a section of an array. It is not permitted to deallocate such a pointer; only whole allocated objects may be deallocated. It is also not permitted to deallocate a pointer associated with an allocatable array; the allocatable array must be deallocated instead. The ASSOCIATED intrinsic function may be used to determine whether a pointer is associated or if it is associated with a particular target or the same target as another pointer.
5. When a pointer is deallocated, its association status is set to disassociated (as if a NULLIFY statement were also executed). When a pointer is deallocated, the association status of any other pointer associated with the same (or part of the same) target becomes undefined.

3 Argument Association

Argument association is the method of linking the arguments between a procedure reference and a procedure definition, and relies on a correspondence between actual arguments and dummy arguments (or formal parameters). An actual argument may contain input values for the procedure to use or may receive computed values to be returned to the calling program. The actual arguments may be expressions, procedures, and labels (representing alternate returns) or be specified by keyword arguments

Examples:

```
! A function reference

  FCN( 1.2, A+B, D, FCN2, TOLERANCE = 1.0E-6 )

  ! The 1st 2 arguments are expressions that are not variables
  !    -- the corresponding dummy arguments must not be defined.
  ! The 3rd argument is a variable -- its dummy may be defined.
  ! The 4th argument is the name of a procedure -- its dummy
  !    argument must be a dummy procedure.
  ! The 5th argument is a keyword argument that is an expression
  !    -- its corresponding dummy argument must not be defined.

! A CALL statement

  CALL  SUB( *99, Y = (/1.0, 2.0 /) )

  ! The first argument is a label, for an alternate return.
  ! The second argument is an array expression, which must
  !    correspond to a dummy array argument that is not defined.
```

Tip: Fortran 77 permited the ranks of the actual and dummy arguments to differ in certain cases. This is still permitted in Fortran, and the association between the arguments is defined by a concept called **sequence association**. However, it is recommended for safer and more reliable programs that the types, kinds, and ranks of the actual and dummy arguments match in all cases.

Related Topics:

Argument Keywords
Array Overview
Defined Type: Objects
Functions
INTENT Attribute and Statement
Interfaces and Interface Blocks
OPTIONAL Attribute and Statement
Storage Association
Subroutines

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 2.5.6, 12.4, 14.6.1.1
Fortran 95 Handbook, 2.1, 12.1.1.11-12, 12.7, 14.3.1.1
Fortran 95 Using F, 3.5

Things To Know:

1. Each actual argument may be an expression (including a variable), a procedure, or an alternate return (* followed by a label). Each dummy argument may be a variable, dummy procedure, or *.
2. The correspondence between actual and dummy arguments is primarily by position; the first actual argument corresponds to the first dummy argument, the second with the second, and so on. The positional correspondence may be overridden by argument keywords where the keyword name specifies the correspondence to the dummy argument of the same name.
3. An actual argument that it is an expression or a function procedure must match the type and kind of the dummy argument. Also, in most cases, the rank of the actual and dummy arguments must match (both scalars or both arrays of the same rank). When the dummy argument is an explicit-shape or assumed-size array, the actual argument may be an array of a different rank or an array element, provided the procedure does not have a reference to the dummy array that is beyond the storage of the actual argument. Array actual arguments may be passed to scalar dummy arguments of an elemental procedure. When the actual argument is an expression, a procedure name, or an alternate return, the dummy argument must be a variable, a dummy procedure, or an asterisk, respectively.
4. When the ranks or character lengths of the actual and dummy arguments do not match, the array elements or character values are sequence associated. This is accomplished by forming the storage sequences of the actual and dummy arguments and matching them beginning with the first element or character of each sequence. The storage sequence is determined by the array element order (column order of elements) and character order in character strings.
5. The corresponding actual and dummy arguments of defined types are of the same defined type if the structures refer to the same type definition. In addition, they are the same type if a) they refer to different type definitions with the same name, b) they have the SEQUENCE statement in their definition, c) the components have the same names and types and are in the same order, and d) none of the components are of a private type or are of a type that has private access.
6. If the dummy argument has the POINTER attribute, the actual argument must also have the POINTER attribute, and the dummy argument in the procedure behaves as if the actual argument were used in its place. If the dummy argument does not have the POINTER attribute but the actual argument is a pointer, the argument association behaves as if the pointer actual argument were replaced by its target at the time of the procedure reference.

4 Argument Keywords

An argument keyword is a dummy argument name, followed by =, that appears in an actual argument list to identify the actual argument. In the absence of argument keywords, actual arguments are matched to dummy arguments by their position in the actual argument list; however, when argument keywords are used, the actual arguments may appear in any order. This is particularly convenient if some of the arguments are optional and are omitted. An actual argument list may contain both positional and keyword arguments; the positional arguments appear first in the list. If an argument keyword is used in a reference to a user-defined procedure, the procedure interface must be explicit. Argument keywords are specified for all intrinsic procedures.

Examples:

```
! Interface for subroutine DRAW
INTERFACE
   SUBROUTINE DRAW (X_START, Y_START, X_END, Y_END, FORM, SCALE)
      REAL X_START, Y_START, X_END, Y_END
      CHARACTER (LEN = 6), OPTIONAL :: FORM
      REAL, OPTIONAL :: SCALE
   END SUBROUTINE DRAW
END INTERFACE

! References to DRAW
CALL DRAW (5., -4., 2., .6, FORM = “DASHED”)
CALL DRAW (SCALE=.4, X_END=0., Y_END=0., X_START=.5, Y_START=3.)

! References to intrinsics LBOUND, UBOUND, SIZE, and PRODUCT
REAL A (LBOUND (B, DIM=1) : UBOUND (B, DIM=1), SIZE (B, DIM=2) )
A_PROD = PRODUCT (A, MASK = A > 0.0 )
```

Tip: Argument keywords can enhance program reliability and readability. Program construction is easier when the strict ordering of arguments can be relaxed.

Related Topics:

Argument Association
Functions
Generic Procedures and Operators
Interfaces and Interface Blocks
Internal Procedures
Module Procedures
OPTIONAL Attribute and Statement
Subroutines

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 2.5.2, 12.4.1, 13.3, 14.1.2.6
Fortran 95 Handbook, 2.5, 12.7.4, 13.1
Fortran 95 Using F, 3.8.6, A.3

Syntax:

A keyword argument is one of:

keyword = expression
keyword = procedure-name

where a keyword is a dummy argument name.

Things To Know:

1. If an argument keyword is used in a reference to a procedure, the procedure interface must be explicit; that is, the procedure must be:
   - an intrinsic procedure,
   - an internal procedure,
   - a module procedure, or
   - an external procedure (or dummy procedure) with an interface block accessible to the program unit containing the reference.

   Statement function references cannot use keyword calls.
2. After the first appearance of a keyword argument in an actual argument list, all subsequent arguments must use the keyword form.
3. If an optional argument is omitted, the keyword form is required for any following arguments.
4. In an interface block for an external procedure, the keywords do not have to be the same as the dummy argument names in the procedure definition. The keyword names can be tailored to fit their use in the referencing program unit.
5. The positional form is required for alternate returns, because the keyword must be a dummy argument name.
6. When choosing argument keyword names for generic procedures, care must be taken to avoid any ambiguity in the resolution of a generic reference to a specific procedure (see Generic Procedures and Operators, item 2 of the Things to Know).

5 Array Overview

An **array** is an object that consists of a set of objects called the **array elements**, all of the same type and type parameters, arranged in a pattern involving rows, and possibly columns, planes, and higher dimensioned configurations. An array is therefore said to have the DIMENSION attribute and may have up to seven dimensions. The number of dimensions is called the **rank** of the array and is fixed when the array is declared. Each dimension has an **extent** which is the size in that dimension (**upper bound** minus **lower bound** plus one). The **size** of an array is the product of its extents. The **shape** of an array is the vector of its extents in each dimension. Two arrays that have the same shape are **conformable**.

Expressions may contain array operands and be array-valued; function results may be array-valued. Intrinsic operations involving conformable array operands are performed element-by-element to produce an array result of the same shape. There is no implied order in which the element-by-element operations are performed. If such operations appear in an assignment statement where the left-hand side is an array, the effect is as if the right-hand side were completely evaluated before any part of the assignment takes place. A scalar may appear in an array expression and is conformable with any array. The effect is as if the scalar were broadcast to form a conformable array of identical elements.

Examples:

```
REAL A (1000, 1000)              ! A has explicit shape.
REAL, ALLOCATABLE :: B (:,:)     ! B has deferred shape.

READ (5, *) N, M                 ! The size of B is determined
ALLOCATE (B(N,M))                !   by input values (< 1000)
READ (5, *) B                    ! The elements of B are read.
IF (N<=1000 .AND. M<=1000) &
   A(1:N,1:M) = SQRT (B)*.25     ! The constant .25 is broadcast.
```

Related Topics:

ALLOCATABLE Attribute and Statement
Array: Constructors
Array: Declaration Forms
Array: Sections
DIMENSION Attribute and Statement
Dynamic Objects
Elemental Procedures
Intrinsic Functions: Array
Intrinsic Functions: Inquiry and Model
OPTIONAL Attribute and Statement
POINTER Attribute and Statement
WHERE Construct and Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 2.4.5, 4.5, 5.1.2.4, 5.2.5, 6.2, 7.5.3-4, 13.8, C.11
Fortran 95 Handbook, 2.3.3, 4.6, 5.3, 6.4, 7.5.1, 7.5.4-5, 13.6
Fortran 95 Using F, 4.1, 8.2

Array Overview 5

Things To Know:

1. An **array element** is referenced when a scalar integer expression appears for each subscript, for example: A (10, 10) or B (I, J–1). An array element is a scalar. An **array section** is referenced when a triplet section or vector subscript section appears for one or more subscripts. An array section is an array.
2. In addition to vector sections, a **WHERE** or **FORALL** construct or statement may be used to select irregular portions of an array.
3. A **dynamic array** is an array specified with deferred shape and the ALLOCATABLE or POINTER attribute. The size of each dimension is determined when the array is allocated or associated. The bounds of an array in the specification part of a procedure may be specified by expressions that involve variables known only on entry to the procedure at run time, so the array is created then. Such a dynamic array is called an **automatic array**. An automatic array always disappears on exit from a procedure whereas the other dynamic arrays may have the SAVE attribute.
4. **Array constructors** allow an array to be constructed from a list of scalar values, arrays of any rank, and implied-do loops.
5. The order of elements in an array, called **array element order**, is important in certain circumstances, such as for input and output list items, internal files, the DATA statement, argument association involving assumed-size or explicit-shape arrays, certain intrinsic functions (for example, RESHAPE, TRANSFER, PACK, and UNPACK), rank-two or greater arrays in array constructors, and storage association. The order is columnwise; that is, the subscripts along the first dimension vary most rapidly and the subscripts along the last dimension vary most slowly. Thus the order of the elements in a 3 by 2 array is (1,1), (2,1), (3,1), (1,2), (2,2), (3,2).
6. An array dummy argument may be declared to have **assumed shape**; for example, REAL D (:, :, :) is a rank-three real array that will take its shape from the actual argument. Assumed-shape dummy arguments require that the procedure must have an explicit interface in the calling program unit.
7. Intrinsic functions, such as SQRT and SIN, and user-defined functions may have array arguments; they are said to be **elemental functions** and return array results of the same shape as the argument. A number of other array intrinsics perform various array computations or return information about arrays.

6 Array: Constructors

An array constructor generates a rank-one array value from a list of scalar values, arrays of any rank, and implied-do loops. An array constructor may be used as a primary in an expression. If the list contains only constant values, the array constructor may be used as a primary in an initialization expression in a type declaration statement, or in the value of a named constant in a PARAMETER statement. If it is desirable to construct an array of rank greater than one, the RESHAPE intrinsic function may be applied to a constructor.

Examples:

```
X = (/19.3, 24.1, 28.6/)           ! An array is assigned a value.
J = (/4, 10, K(1:5), 2 + L,  &     ! A vector of 16 integer values
   (M(N), N = -7,-2),16,1/)        !   is assigned to J.
A = (/(BASE(K), K=1,5)/)           ! 5 values are assigned.
PARAMETER (T=(/ 36.0, 37.0/))      ! T is vector valued.
Z=RESHAPE((/1,2,3,4,5,6,7,8/), &   ! Z is reshaped as  1 3 5 7
   (/2,4/) )                       !                   2 4 6 8

TYPE SITE                          ! SITE is a defined type.
  CHARACTER *10 PLACE              ! PLACE is a scalar component.
  INTEGER CLIMATE (2)              ! CLIMATE is an array component.
END TYPE SITE
   . . .
TYPE (SITE) ALASKA                 ! ALASKA is declared to be
   . . .                           !   of type SITE.
ALASKA = SITE("NOME",(/-63,4/))    ! An array constructor is used
   . . .                           !   for the second component.

DIAGONAL = (/ (B(I,I), I=1,N) /)
HILBERT = RESHAPE((/ ((1.0/(I+J), I=1,N), J=1,N) /), (/ N,N /) )
IDENT = RESHAPE ( (/ (1, (0, I=1,N), J=1,N-1), 1 /), (/ N,N /) )
```

Related Topics:

Array Overview
Array: Declaration Forms
Array: Sections

Related Intrinsics:

RESHAPE (SOURCE, SHAPE, PAD, ORDER)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.5
Fortran 95 Handbook, 4.6, 7.2.8.1
Fortran 95 Using F, 4.1.4

Syntax:

An array constructor is:

(/ ac-value-list /)

An *ac-value* is one of:

scalar-expression
array-expression
ac-implied-do

An *ac-implied-do* is:

(ac-value-list , scalar-integer-variable-name = &
scalar-integer-expression , scalar-integer-expression &
[, scalar-integer-expression])

Things To Know:

1. The result of an array constructor is a rank-one array.
2. If no values are given (for example, an empty implied-DO), the array is zero sized.
3. If an array of rank greater than one appears in the value list, it is treated as a rank-one array with the values appearing in array element order (see Array Overview).
4. The values in the array constructor value list must be of the same type and type parameters (including character length).
5. The set of values may be a scalar expression, an array expression, or an implied DO specification. These may be mixed in one array constructor.
6. The RESHAPE function may be used to change the shape of the result to any desired shape.
7. An array constructor must not appear in a DATA statement, because only scalar values are allowed there. However, it may appear in a data initialization of a type statement.
8. As illustrated by the last two examples, an array constructor with implied DOs and the RESHAPE function can be used to construct arrays that cannot be expressed conveniently with other notation.

7 Array: Data-Parallel Operations

Fortran is an ideal applications language for what is known as "data parallelism". Data parallelism means applying the same computation simultaneously to similar data objects. The most common example of this is simultaneous operations on the elements of an array. In data-parallel terms, the cosine of each element of a 1000 by 1000 array, for example, can be computed simultaneously on a parallel machine. Computer architectures that can perform such operations in parallel are rapidly becoming practical for "high performance" computing applications. Whole array operations make it easier to program such machines and to program data parallel applications on scalar machines.

The Fortran array semantics specify that array operations are element-by-element, conceptually in parallel. That is, the result of such an operation must be as if the operation is performed on each element independently and simultaneously. Fortran provides many such operations, together with array intrinsics, to support high performance applications.

Examples:

```
REAL, DIMENSION(1000,1000) :: A, B, C, P(1000), Q(1000)

A = 0.0      ! Simultaneously set all million elements of A to 0.
C = COS(B)   ! Simultaneously set each element of C to the
             !   cosine of the corresponding element of B.
B = C-A      ! Simultaneously subtract each element of A from the
             !   corresponding element of C, and store the result
             !   in the corresponding element of B.
B(K,:) = Q   ! Assign all elements of Q to the Kth row of B.
P = C(:,L)   ! Copy the Lth column of C into the vector P.

C = MATMUL(A,B) ! Matrix multiply A and B; result stored in C.
P = DOT_PRODUCT(A(:,K),C(L,:))    ! Dot product of two vectors
A = SQRT(ABS(B))*EXP(A)-C**3      ! Another million computations

B(1:500,1:500) = B(   1: 500,501:1000) &  ! Each element of upper
               + B(501:1000,   1: 500) &  !   left block of B gets
               + B(501:1000,501:1000)     !   sum of other blocks.
```

Related Topics:

Array Overview
Array: Sections
Dynamic Objects
Elemental Procedures
Expressions
FORALL Construct and Statement
Functions
Intrinsic Functions: Array
Variables
WHERE Construct and Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 6.2, 7.1.4-5, 7.5.3-4, 12.7, 13.8, C.11.8
Fortran 95 Handbook, 6.4, 7.2.8, 7.4, 7.5.4-5
Fortran 95 Using F, 4.1

Array: Data-Parallel Operations 7

Things To Know:

1. As the examples illustrate, the Fortran syntax for scalar operations is extended, without exception, to array-valued operands. The meaning is that a number of (simultaneous) scalar operations are performed, one for each element of the operands. The main requirement over the rules for scalar operations is that in any data-parallel operation the operands must all be **conformable**, which means they must have the same shape—that is, the same rank (number of dimensions) with the same number of elements in each dimension.
2. Because an assignment statement may have the same array on both the left- and right-hand sides, data-parallel semantics forces the right-hand side to be fully evaluated before any assignment takes place. This means that in some cases the compiler may create temporary space to hold intermediate results of the computation.
3. Most of the intrinsic functions are **elemental; user-defined functions also may be elemental**. That means they are defined with scalar dummy arguments but may be called with array actual arguments. The function returns an array of the same shape, each element of which is the result of applying the function to the corresponding element of the actual argument. A few of the intrinsic functions are not elemental, but **transformational**. These functions "transform" their array actual arguments "as a whole", rather than element by element, into the result. Elemental functions are data parallel, as they can be envisioned (and implemented) as simultaneous operations on array elements. Transformational functions are not data parallel in this sense, although their results may be used as operands in data-parallel computations.
4. User-defined functions may be array-valued, a feature that is useful in designing data-parallel computations. However, user-defined functions are transformational and therefore the evaluation of such a function must be completed before the result of the function can be used in an expression. (The internal evaluation of a user-defined array-valued function may take advantage of data parallelism.) Because user-defined operations involve user-defined functions, such operations have the characteristics of transformational functions rather than intrinsic operations (which are similar to elemental functions).
5. Array assignments can be masked to avoid both assignment of values and the performance of the elemental operations. This masking is accomplished either by use of the WHERE or FORALL constructs or statements; logical expressions are used to specify which elements participate in the computation. These constructs thus permit data parallel operations with irregular patterns of array elements selected by logical masks.

8 Array: Declaration Forms

An object is an array if an array specifier is used in the declaration of the object. An array specifier is enclosed in parentheses and follows an array name or it may follow the DIMENSION keyword in a type declaration. An array specification determines the rank (number of dimensions) and in some cases the shape of an array. An array declaration is one of:

- explicit shape
- assumed shape
- deferred shape
- assumed size

When the shape is completely specified, the array is an explicit-shape array. For assumed-shape and assumed-size arrays, the lower bound in each dimension may be specified in the array declaration. For a deferred-shape array, only the rank is specified.

Examples:

```
REAL X(10, 1:5, -2:3)          ! X and P have explicit shape.
DIMENSION P(1500)              ! The bounds are constants.
CHARACTER*5 C(N,N)             ! An array of character strings
                               !    of length 5
REAL Y(:), Z(-5:,:)            ! Y and Z have assumed shape and
     . . .                     !    must be dummy arguments.
REAL, ALLOCATABLE,  &          ! S and T have deferred shape
   DIMENSION (:,:) :: S,T      !    and must be subsequently
                               !    allocated.
REAL, POINTER :: Q(:,:,:)      ! A pointer with deferred shape
REAL, DIMENSION (10,5,*) :: &  ! TALLY is an assumed-size
        TALLY                  !   dummy argument.
COMMON /BLOCK_A/ R(10,12,3)    ! An explicit-shape array
                               !    in common
REAL A(N,M)                    ! A is an automatic array.
```

Related Topics:

Array Overview | Dynamic Objects

Related Intrinsics:

ALLOCATED (ARRAY)
ASSOCIATED (POINTER, TARGET)
LBOUND (ARRAY, DIM)
RESHAPE (SOURCE, SHAPE, PAD, ORDER)
SHAPE (SOURCE)
SIZE (ARRAY, DIM)
UBOUND (ARRAY, DIM)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.4, 5.2.5
Fortran 95 Handbook, 5.3.1-2
Fortran 95 Using F, 4.1.3

Syntax:

An array specifier is:

explicit-shape-spec-list
assumed-shape-spec-list
deferred-shape-spec-list
assumed-size-spec-list

An explicit-shape specifier is:

[*lower-bound* :] *upper-bound*

An assumed-shape specifier is:

[*lower-bound*] :

A deferred-shape specifier is a colon (:).

An assumed-size specifier is:

[*explicit-shape-spec-list* ,] [*lower-bound* :] *

Things To Know:

1. An array specifier determines the rank of an array and sometimes its bounds. The bounds may be explicit, deferred, or assumed.
2. The rank is the number of dimensions. The maximum rank is 7. A scalar has rank 0. The extent is the length in any dimension from the lower bound to the upper bound. The shape is a vector of the extents.
3. Bounds may be positive, zero, or negative integer values.
4. Arrays with assumed shape or size must be dummy arguments.
5. An **explicit-shape array** has explicitly declared bounds.
6. An **assumed-shape array** is a dummy argument that takes the shape of the associated actual argument. If a lower bound is specified, a subscript may extend from the specified bound to an upper bound equal to the lower bound plus the extent minus one. An **assumed-size array** is a dummy argument for which the extents are all explicit except for the last, which is an asterisk (*). A subscript in the last dimension may extend from the lower bound to a value that does not cause the reference to go beyond the actual argument.
7. The result of a function must not be an assumed-size array.
8. A **deferred-shape array** has the POINTER attribute or the ALLOCATABLE attribute. The shape is not specified until the array is allocated or pointer assigned. Notice that a deferred-shape and assumed-shape specifier may have the same form; however, they are different because an assumed-shape array must be a dummy argument that does not have the POINTER attribute and a deferred-shape array must have either the ALLOCATABLE or POINTER attribute.

9 Array: Sections

An array section is an array that consists of a selected portion of an array called the **parent** array. It may be used anywhere a whole array may be used. Array section subscripts may identify a smaller section of an array convenient to a calculation. There are two subscript forms used to describe a section: subscript triplets and vector subscripts.

Examples:

```
INTEGER, DIMENSION(3,6) :: X,Y,Z   ! X, Y, and Z are 3x6 arrays.
   . . .
X = 0; Y = 0; Z = 0
X(3,2:4:1)  = 1                 ! Using subscript triplets, the
Y(2,2:6:2)  = 2                 !   selected elements are marked.
Z(1:2,3:6)  = 3

   0 0 0 0 0 0      0 0 0 0 0 0      0 0 3 3 3 3
   0 0 0 0 0 0      0 2 0 2 0 2      0 0 3 3 3 3
   0 1 1 1 0 0      0 0 0 0 0 0      0 0 0 0 0 0

INTEGER, DIMENSION (4) :: &      ! M is a vector defined
   M = (/3,4,8,1/)               !   with an array constructor.
REAL, DIMENSION (10) :: &        ! R is initialized.
   R = (/ (I*1.1, I=1,10) /)
   . . .
PRINT *, R(M)              ! Prints 3.3 4.4 8.8 1.1
   . . .                   ! M is a vector subscript.
R(M) = 20.                 ! R(3), R(4), R(8), R(1) are set to 20.
                           !    using the vector subscript for M.

PRINT *, R      ! Prints 20. 2.2 20. 20. 5.5 6.6 7.7 20. 9.9 11.
```

Tip: Vector subscripts are useful for indirect array addressing, such as indexing into a table.

Related Topics:

Array Overview
Array: Constructors
Array: Declaration Forms
Character Substring

Related Intrinsics:

LBOUND (ARRAY, DIM)
SHAPE (SOURCE)
SIZE (ARRAY, DIM)
UBOUND (ARRAY, DIM)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 6.2.2.3
Fortran 95 Handbook, 6.4.4-6, 12.7.2.2
Fortran 95 Using F, 4.1.6

Syntax:

An array section is:

array-name (section-subscript-list)

An array section subscript is one of:

subscript
subscript-triplet
vector-subscript

A subscript triplet is:

[*subscript*] : [*subscript*] [: *stride*]

A subscript or stride is:

scalar-integer-expression

A vector subscript is a one-dimensional integer array.

Things To Know:

1. At least one of the subscripts for an array section must be a subscript triplet or a vector subscript. A vector subscript is a rank-one array of integer values used to identify elements in the parent array.
2. Use of a subscript triplet produces a regular pattern in that dimension, while vector subscripts can be used to describe an irregular pattern in a dimension.
3. An array section has the attributes of its parent array.
4. The rank of an array section is the number of subscripts that are subscript triplets and vector subscripts. If there are none, it is an array element and is a scalar.
5. If the lower or upper bound of a subscript triplet is omitted, the bound is that of the parent array. (See Array: Declaration Forms.)
6. If the array section is of character type, it also may have a substring range specified. The substring range appears after the section specification. (See Character Substring.)

   ```
   CHARACTER (LEN = 40) :: DESTINATIONS (30)
   DESTINATIONS (20:25)(36:40) = “94510”
   ```

   The last five characters of the 20th through 25th elements of DESTINATIONS receive the zip code 94510.
7. If the section subscript in the last dimension of an assumed-size array is a subscript triplet, the upper bound must be specified.
8. A vector subscript with two or more instances of the same value is a **many-to-one array section**. Such a section must not appear on the left side of an assignment or as an input item in a READ statement because the duplication of elements would make the result unpredictable.

10 Assignment

The assignment statement defines a variable with a specified value and comes in five flavors: an intrinsic assignment gives a value of an expression to a variable (described below); a defined assignment gives a value, determined by a subroutine, to a variable (see Defined Operators and Assignment); a pointer assignment associates a target to a variable with the POINTER attribute (Pointer Association); a masked array assignment statement assigns selected elements of an array expression to the same selected elements of an array variable (WHERE Construct and Statement); and a selected-element assignment assigns to elements, selected by index sets, and logical expressions, scalar values selected by the same index sets and logical expressions (FORALL Construct and Statement).

Examples:

```
INTEGER  I(5)
REAL  X                  ! X is assigned the value 3.0.
X = 3                    ! I is assigned a vector of integer
I = X + (/ 1,2,3,4,5 /)  !   values  4, 5, 6, 7, 8.

REAL Y
COMPLEX  C               ! C becomes equal to the complex
C = Y                    !   value  Y + 0*i.

LOGICAL(1)  LS           ! Assume kind=1 is not default logical.
LOGICAL  L
L = LS                   ! L gets of the value LS but
                         !   converted to default logical.

CHARACTER(4)  CH3
CH3 = 'This string gets truncated to "This"'

TYPE RATIONAL             ! Define type RATIONAL.
    INTEGER  N, D
END TYPE RATIONAL
TYPE(RATIONAL)  R
R = RATIONAL(1,2)         ! R is assigned the rational
                          !   number 1/2.
```

Related Topics:

Defined Operators and Assignment
Expressions
FORALL Construct and Statement
Pointer Association
Variables
WHERE Construct and Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 7.5, C.4.1
Fortran 95 Handbook, 7.1.1, 7.5
Fortran 95 Using F, 1.6.4, 4.1.7-9, 5.1.4, 5.1.14, 8.1.1

Syntax:

The form of an intrinsic assignment is:

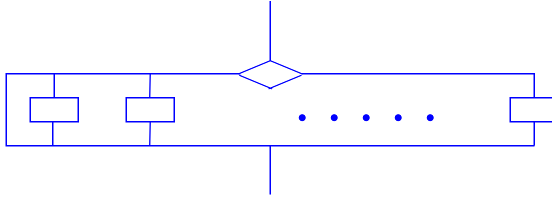
variable = *expression*

Things To Know:

1. If the type of the variable is arithmetic, the expression may be of any arithmetic type and any kind. If the type of the variable is character, the expression must be of type character of the same kind but any length. If the type of the variable is logical, the expression must be of type logical of any kind. If the variable is of a defined type, the expression must be of the same defined type. If the expression is an array, the variable must be an array of the same shape. If the expression is a scalar, the variable may be a scalar or an array of any shape.
2. When the variable and expression are of the same type, kind, shape, and length, the value of the variable becomes that of the expression; otherwise, the value of the expression is converted to the type and kind of the variable and becomes the value of the variable. When the expression and variable are of type character but of different lengths, the expression is truncated or extended on the left with blanks to equal the length of the variable. If the variable and the expression are of the same defined type and no accessible defined assignment applies, the components of the expression are assigned to the corresponding components of the variable, using pointer assignment for pointer components and intrinsic assignment otherwise.
3. The evaluation of the expression and the assignment to the variable must behave as if the expression is evaluated first before any part of the variable is assigned a value.
4. When the variable and expression are arrays, the assignment is element-by-element. When the expression is a scalar and the variable is an array, every element of the array is assigned the value of the expression.
5. If the variable has the POINTER attribute, it must be associated with a target, and the expression is assigned to the target.
6. The variable must not be an assumed-size array; it may be a section of an assumed-size array, provided that for the last dimension there is a subscript, a vector subscript, or a subscript triplet with an upper bound.

11 CASE Construct

The CASE construct may be used to select for execution at most one of the blocks in the construct. Selection is based on a scalar value of type integer, character, or logical. A CASE construct may be named. It permits the following control flow:



Examples:

```
! Character example

 SELECT CASE (STYLE)
  CASE DEFAULT
    CALL SOLID (X1,Y1,X2,Y2)
  CASE ("DOTS")
    CALL DOTS (X1,Y1,X2,Y2)
  CASE ("DASHES")
    CALL DASHES (X1,Y1,X2,Y2)
END SELECT
```

```
! Integer example
                      RANGES: &
SELECT CASE (ITEM)
  CASE (1:7, 52:81)   RANGES
    BIN1 = BIN1 + 1.0
  CASE (8:32, 51, 82) RANGES
    BIN2 = BIN2 + 1.0
  CASE (33:50, 83: )  RANGES
    BIN3 = BIN3 + 1.0
  CASE DEFAULT        RANGES
    WRITE (*, "('BAD ITEM')")
END SELECT            RANGES
```

```
! Logical example

LIMIT: SELECT CASE (X > X_MAX)
CASE (.TRUE.)
      Y = X * 0.9
   CASE (.FALSE.)
      Y = 1.0 / X
END SELECT LIMIT
```

Tip: For program clarity, use an IF-THEN-ELSE construct rather than a logical CASE construct.

Related Topics:

Expressions: Initialization

IF Construct and Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 8.1.3, C.5.2
Fortran 95 Handbook, 8.4
Fortran 95 Using F, 2.3

Syntax:

A CASE construct is:

[*case-construct-name* :] SELECT CASE (*case-expression*)
[CASE (*case-value-range-list*) [*case-construct-name*]
block]...
[CASE DEFAULT [*case-construct-name*]
block]
END SELECT [*case-construct-name*]

A *case-value-range* is one of:

case-value [: *case-value*]
case-value :
: *case-value*

Things To Know:

1. The case expression and all case values must be scalar and of the same type. The case values must be initialization expressions. The types allowed are integer, character, and logical. If the character type is used, different lengths are allowed. If the logical type is used, a case value range (with a :) is not permitted. Overlapping case values are prohibited.
2. The case value range list enclosed in parentheses and the keyword DEFAULT are called selectors. The case expression must select at most one of the selectors. If the case expression matches one of the values or falls in one of the ranges, the block following the matched selector is the one executed. If there is no match, the block following the DEFAULT selector is executed; it need not be last. If there is no match and no DEFAULT selector, no code block is executed and the CASE construct is terminated. A block may be empty.
3. Control constructs may be nested, in which case a program may be easier to read if the constructs are named. If a construct name appears on a SELECT CASE statement, the same name must appear on the corresponding END SELECT statement and is optional on CASE statements of the construct.
4. A construct name must not be used as the name of any other entity in the program unit such as a variable, named constant, procedure, type, namelist group, or another construct.
5. Branching to any statement in a CASE construct, other than the initial SELECT CASE statement, from outside the construct is not permitted. Branching to an END SELECT statement with a GO TO statement from within the construct is permitted.

12 Character Substring

A character substring is a scalar object consisting of zero or more characters that is a contiguous portion of a character string called the **parent** of the substring. A substring has a starting point and an ending point within the parent string.

It is possible to have an array of character strings, all of the same length. It is also possible to have an array of substrings. Because the same syntax is used for an array section as for a substring, when an array of substrings is referenced, the array section must be specified even if the substring applies to the whole array. The array section appears first as in the last two examples below.

Examples:

```
TYPE TEACHER
   INTEGER GRADE
   CHARACTER (LEN = 40) NAME
END TYPE TEACHER
TYPE(TEACHER) PRINCIPAL, TEACHERS (30)
CHARACTER (20) DISTRICT, SCHOOLS (15)
CHARACTER (4) TEMP, TITLES (30)

DISTRICT (7:13) = "UNIFIED"       ! Parent is a scalar variable.
TEMP = "ELMWOOD HIGH" (N:N+3)     ! Parent is a constant.
SCHOOLS (5) (J:J+3) = TEMP        ! Parent is an array element.
PRINCIPAL%NAME (36:) = "SMITH"    ! Parent is structure component.
TEACHERS(:)%NAME(1:4) = TITLES    ! Array of 4-character substrings
SCHOOLS(:)(14:19) = "MIDDLE"      ! Array of 6-character substrings
SCHOOLS(3:5)(14:19) = "  HIGH"    ! Array section of substrings
```

Tip: A substring that references a single character (n : n) is useful for looping over a character string (see example on facing page).

Related Topics:

Array: Sections
Assignment
Character Type and Constants
Defined Type: Structure Component

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 6.1.1
Fortran 95 Handbook, 6.2
Fortran 95 Using F, 5.1.11-15

Syntax:

A substring is:

parent-string ([*starting-position*] : [*ending-position*])

A parent string is one of:

scalar-variable-name
array-element
scalar-structure-component
scalar-constant

Things To Know:

1. The parent string of a substring must be of type character of any kind. The substring is of type character of the same kind.
2. The starting and ending position of a substring must be within the range of the parent except in the case of a null string. If the starting position is greater than the ending position, the substring is zero length. If the starting position is omitted, the default is 1; if the ending position is omitted, the default is the length of the parent.

Example of an elementary encryption scheme:

```
FUNCTION CRYPT (C, DECODE)
   USE CODER, ONLY : KEY          ! Supplies a key from 1 to 26
   CHARACTER (*) CRYPT, C
   LOGICAL, OPTIONAL :: DECODE    ! Encode if absent or .FALSE.
   INTEGER I
   IF (PRESENT (DECODE)) THEN
      IF (DECODE) KEY = 26 - KEY
   END IF
   DO I = 1, LEN(C)
      CRYPT(I:I) =  &
      ACHAR(MOD(IACHAR(C(I:I))-IACHAR('A')+KEY,26)+IACHAR ('A'))
   END DO
END FUNCTION CRYPT

MSG = CRYPT ("SENDMONEY")        ! If KEY = 3, MSG = "VHQGPRQHB"
MSG = CRYPT (MSG, .TRUE.)        ! MSG = "SENDMONEY"
```

The IACHAR intrinsic function returns the integer position in the collating sequence of its character argument. Thus IACHAR (C(I:I)) – IACHAR('A') supplies a value in the range 0-25. Adding the key to this value gives a new value in the range 1-52. This value modulo 26 returns a value in the range 0-25 which when added to IACHAR('A') maps back onto the collating sequence positions for the upper case letters. The decryption process is similar except the addend is 26 minus the key to get back to the original text.

13 Character Type and Constants

The Fortran character type represents strings of characters. The character type is useful in producing readable output. A character constant is a sequence of characters representable by the processor; the constant may be preceded by a kind parameter and an underscore. Concatenation (//) is the only character operator other than the comparison operators.

Examples:

```
CHARACTER (LEN=9) FAMILY_NAME   ! The length is 9 characters.
CHARACTER (*) GIFT_GIVING       ! An asterisk declares an
   . . .                        !   undetermined length.

CHARACTER (LEN=10,KIND=ARABIC)& ! COUNTRY and PERSON have length
   COUNTRY, PERSON              !   10 and are of a nondefault
      . . .                     !   kind with the value ARABIC.

CHARACTER LINE (0:100)          ! LINE is an array
                                !   of single characters.

CHARACTER (LEN = *), PARAMETER :: GREETING = “HELLO WORLD”
```

Examples of character constants are:

```
"JONES"
'isn''t'                     Doubling the " or ' is required
"don't"                         if the delimiting " or ' appears
greek_"πβϕ"                     in a string of characters
```

Character Substring
Expressions
Implicit Typing
Intrinsic Functions: Computation

Related Intrinsics:

ACHAR (I)
ADJUSTL (STRING)
ADJUSTR (STRING)
CHAR (I, KIND)
IACHAR (C)
ICHAR (C)
INDEX (STRING, SUBSTRING, BACK)
KIND (X)
LEN (STRING)
LEN_TRIM (STRING)
LGE (STRING_A, STRING_B)
LGT (STRING_A, STRING_B)
LLE (STRING_A, STRING_B)
LLT (STRING_A, STRING_B)
REPEAT (STRING, NCOPIES)
SCAN (STRING, SET, BACK)
TRIM (STRING)
VERIFY (STRING, SET, BACK)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.3.2.1, 5.1.1.5
Fortran 95 Handbook, 4.3.5, 5.1.6
Fortran 95 Using F, 1.2.8, 1.3.1, 5.1.3

Syntax:

A character type declaration statement is:

```
CHARACTER [ ( [ LEN = ] length-parameter   &
   [ , [ KIND = ] kind-parameter ] ) ]   &
   [ , attribute-list :: ] entity-list
```

A length parameter is one of:

```
specification-expression
*
```

A character constant is one of:

```
[ kind-parameter _ ] ' [ representable-character ] ... '
[ kind-parameter _ ] " [ representable-character ] ... "
```

Things To Know:

1. The length parameter determines the length of the character string. If no character length is specified, the length is 1; if the length is negative, a length of 0 is assumed. A length parameter of ∗ may be used only to declare a dummy argument, a named constant, or the result variable for an external function. The function must not be array valued, pointer valued, or recursive.
2. The kind parameter specifies a kind of character. The value of the kind parameter must be a nonnegative integer and specify a representation method that exists. At least one kind value is for the default character type. The default character set must contain the Fortran character set.
3. Each character of the default character kind occupies one character storage unit. Characters of other kinds, if any, occupy unspecified storage units.
4. In a character constant, the kind parameter precedes the string of characters. This is not true for the other types where the kind parameter follows. An example is

   ```
   MATH_SYMBOLS_"Σxy"
   ```

5. The operator for character concatenation is two slashes (//). The relational operators are used for comparisons giving logical results. There is a processor-dependent collating sequence for these comparisons.
6. Automatic character-length objects are allowed in procedures:

   ```
   CHARACTER (N) DESCRIPTION
   ```

   where N is a variable whose value is known on entry to a procedure. The variable DESCRIPTION is an automatic character object of length N. These and character dummy arguments specified with a length of ∗ are the only character objects whose length may vary.

14 CLOSE Statement

The CLOSE statement closes (terminates) the connection of an external file to a unit. If a unit is not closed explicitly by a CLOSE statement, the connection is always closed when the program terminates. A unit may be preconnected or opened during execution, then closed and subsequently reopened to the same or a different file.

Examples:

```
K9 = 9
CLOSE (IOSTAT = IERR, UNIT = K9)
! IERR is zero if no error occurs.
! Use of the keyword UNIT allows the unit to appear
!   anywhere in the list.

CLOSE (1)
! Close unit 1.
! If an error condition occurs, the program terminates
!   because there is no IOSTAT= or ERR= specifier.

IU = I + J - K
CLOSE (IU, ERR = 10, STATUS = 'KEEP')
! The file is kept on program termination.
! No keyword is used for the unit so the unit number
!   must appear first in the list.

CLOSE (K9, IOSTAT = IE, STATUS = "DELETE")
! The file on unit K9 is deleted after it is closed.
! This means, for example, that the file on unit K9
!   cannot be reopened.
```

Related Topics:

INQUIRE Statement
OPEN Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.3.5, C.6.4
Fortran 95 Handbook, 9.6
Fortran 95 Using F, 9.5

Syntax:

A CLOSE statement is:

CLOSE ([UNIT =] *scalar-integer-expression* [, *close-spec-list*])

The CLOSE specifiers appear in the following table.

Specifier=	Type	Value	Description
ERR=	Lb	Label	Branch target taken on an error condition
IOSTAT=	I	Positive	An error condition occurred
		Zero	No error condition occurred
STATUS=	C	KEEP	Default, file continues to exist
		DELETE	File is deleted after close completes
Lb = label; I = integer variable; C = default character scalar expression; the character values are without regard to case and trailing blanks are ignored			

Things To Know:

1. A specifier may appear only once in the list of close specifiers.
2. Branching to an ERR= label is permitted only when the label is in the same scoping unit.
3. If a CLOSE statement refers to a unit that is not connected or doesn't exist, it has no effect.
4. KEEP indicates that the file is to continue to exist after closing the file.
5. DELETE indicates that the file will not exist after closing the file.
6. An OPEN statement for a unit that is already connected causes a CLOSE statement to be executed on that unit with a default status specification of KEEP.
7. When a program terminates, connections not closed explicitly are closed.
8. A unit that has been closed may be reopened to the same or a different file. The unit must be an external unit.

15 COMMON Statement

Common blocks allow variables in different program units to share storage, thereby permitting data to be global or space to be reused. A COMMON statement places objects in common blocks. Thus common blocks provide a data sharing facility based on storage association.

Examples:

```
COMMON X(100), Y(100), XTEMP(2)  ! Arrays are in blank common.
COMMON/PRESSURE/ P,Q(10:30,100)  ! P is scalar and Q is an array
   . . .                         !   in common block PRESSURE.
REAL, TARGET :: Q
REAL, POINTER :: R(:,:)          ! R is a pointer variable.
COMMON /PRESSURE/ R, S(200), &   ! Extend previously declared
   / / Z, TEMP(2)                !  named and blank common blocks.
SAVE /PRESSURE/

REAL SALARY                      ! Common block EMPLOYEE contains
INTEGER SSN                      !  variables of different types.
CHARACTER *20 NAME
COMMON /EMPLOYEE/ NAME, SSN, SALARY

PARAMETER (N = -1)
COMMON /EXPERIMENT/TRY(1:N)      ! Zero-size common block
```

Tip: Common has been an important part of Fortran for a long time, even though storage association has been a source of programming difficulties and errors. With the advent of modules in Fortran 90, a superior way of making data global is now available; common should therefore be avoided in new Fortran code.

In the past, common blocks were employed frequently to reuse space; this practice is error prone and, with the advent of larger memory machines, no longer advantageous.

Related Topics:

Defined Type: Default Initialization
Storage Association
SAVE Attribute and Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.5.2, C.8.3.1
Fortran 95 Handbook, 5.11.4-5, 14.3.3

Syntax:

A COMMON statement is:

COMMON [/ [*common-block-name*] /] *common-block-object-list*

A common block object is:

variable-name [(*explicit-shape-spec-list*)]

Things To Know:

1. The following are not permitted in common: a dummy argument, an allocatable array, a nonsequence structure, an automatic entity, a function, a result name, or a variable accessible via use association. Only one appearance of a variable name in all common blocks is permitted within a scoping unit.
2. Array bounds in a COMMON statement must be constant specification expressions. Zero-sized common blocks are allowed. If there are explicit bounds for an array in common, it must not have the POINTER attribute.
3. A common block defines a storage sequence. This allows data and storage space to be shared among program units via storage association. An element of a common block storage sequence can have different names in different program units. Variables with different types and attributes may be mixed in a given common block. See Storage Association for the important rules governing such mixtures.
4. A common block without a common block name is called **blank common**. Blank common has the same properties as named common, except:
   - A variable in blank common must not be data initialized.
   - A variable in blank common must not be of a type for which default initialization is specified. (See Defined Type: Default Initialization.)
   - Blank common is always saved; a named common block is not saved unless it appears in a SAVE statement.
   - A named common block must be the same size in all scoping units; blank common may differ in size.
5. A subsequent appearance of a given common block in the same program unit is treated as a continuation of the common block, as illustrated in the examples.
6. If an object in common is of a defined type, it must be a sequence type.

16 Complex Type and Constants

The complex type is used for data that are approximations to the mathematical complex numbers. A complex number consists of a real part and an imaginary part and is often represented as $a + b\iota$ in mathematical terms, where a is the real part and b is the imaginary part.

Examples:

```
COMPLEX CUT, CTEMP, X(10)     ! Complex type declaration

COMPLEX (KIND=LONG) :: CTC    ! CTC has kind parameter LONG
REAL XX, Y
CTC = CMPLX (XX, Y, KIND = LONG)

COMPLEX (SELECTED_REAL_KIND (6,32)) NORTH
! NORTH is a complex variable or function
!   whose parts have at least 6 decimal digits of precision
!   and decimal range of 10^-32 to 10^32.
```

Examples of complex constants are:

```
(1.0,2.0)                 A complex constant:
                              1.0 is the real part.
                              2.0 is the imaginary part.
(4, -.4)                  Integer values are converted to real.
(2, 3.E1)                 One part is integer and the other is
                            is real, but the resulting complex
                            constant is of type default real with
                            both parts of this type.
(1.0_LONG, 2.0_LONG)      The complex constant has the kind LONG.
```

Related Topics:

Expressions
Implicit Typing
Real Type and Constants

Related Intrinsics:

AIMAG (Z)
CMPLX (X, Y, KIND)
KIND (X)
PRECISION (X)
RANGE (X)
REAL (A, KIND)
SELECTED_REAL_KIND (P, R)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.3.1.3, 5.1.1.4
Fortran 95 Handbook, 4.3.3, 5.1.4
Fortran 95 Using F, 1.2.3

Complex Type and Constants 16

Syntax:

A COMPLEX type declaration statement is:

COMPLEX [([KIND =] *kind-parameter*)] [, *attribute-list* ::] *entity-list*

A complex constant is:

(*real-part* , *imaginary-part*)

The real part is one of:

signed-integer-literal-constant
signed-real-literal-constant

The imaginary part is one of:

signed-integer-literal-constant
signed-real-literal-constant

Things To Know:

1. The arithmetic operators are +, –, /, *, **, unary +, and unary –. Only the relational operators == and /=, and synonymously .EQ. and .NE. may be used for comparisons; the result is a default logical value.
2. There are at least two approximation methods for complex, one is default real, and one is default double precision. There are as many complex kinds as there are real kinds.
3. If both parts of a complex constant are integer, they are converted to real. If one part is integer, it is converted to the type and kind of the other part.
4. If both parts of a complex constant are real, but not with the same kind parameter, both take the kind parameter corresponding to the one with the higher precision.
5. The intrinsic function CMPLX (X, Y, KIND) converts complex, real, or integer arguments to complex type. If the first argument is complex, the second argument must not be present. The kind parameter also is optional. The intrinsic function REAL (Z, KIND) extracts the real part of a complex Z and the expression REAL (AIMAG (Z), KIND) extracts the imaginary part of Z, each resulting in a real of kind KIND.
6. Note that there is no default implicit typing for complex.

17 Data Initialization

Initial values may be assigned to variables and pointers may be initially disassociated in a type statement or in a DATA statement prior to the beginning of execution. This is in contrast to the usual situation where variables do not have a value assigned and pointers have an undefined status prior to the beginning of execution.

The DATA statement is the only attribute statement for which there is no corresponding attribute that may appear in a type statement. Instead, an initialization expression is used in a type statement to disassociate pointers and to assign initial values to variables and named constants. If there is a PARAMETER attribute, the declared objects are named constants. Initialized variables, other than those in named common blocks, have the SAVE attribute.

Examples:

```
REAL :: CLIMATE = 16.8                      ! In a type statement
DATA CLIMATE / 16.8 /                       ! In a DATA statement

REAL :: X(3) = (/ 1.2, 3.3, 4.3 /)
DATA (X(I), I=1,3) / 1.2, 3.3, 4.3 /

REAL :: TEMP(365,24) = 0.0
DATA ((TEMP (I,J), I=1,365), J=1,24) / 8760 * 0.0 /

INTEGER :: ICOUNT = 99
DATA ICOUNT / 99 /

REAL, POINTER  :: PRESSURE(:,:) => NULL( )
DATA PRESSURE /NULL( )/

TYPE (RATIONAL) :: R0 = RATIONAL (0,1)
DATA R0 / RATIONAL (0,1) /
```

Tip: A type declaration is preferred for the initialization of a whole variable, because it makes the program easier to read if all the information about a variable is in one place. However, the use of a DATA statement is the only way to data initialize a structure component, a single array element, an array section, or a substring. The use of an assignment statement or NULLIFY statement is another alternative.

Related Topics:

Assignment
Expressions: Initialization
PARAMETER Attribute and Statement

Related Intrinsics:

NULL (MOLD)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1, 5.2.10
Fortran 95 Handbook, 5.5.1
Fortran 95 Using F, 1.3.1

Syntax:

A type declaration statement with data initialization is:

type [, *attribute-list*] :: *object-name* [(*array-spec*)] &
  [* *character-length*] *initialization*

An initialization is one of:

= *initialization-expression* => NULL ( )

A DATA statement is:

DATA *data-object-list* / *data-value-list* /

A data object is one of:

variable
data-implied-do

A data value is one of:

scalar-constant
BOZ-literal-constant
signed-integer-or-real-constant
structure-constructor
NULL( )

A data-implied DO is:

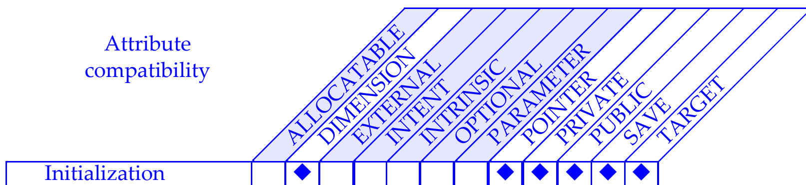
(*data-implied-do-object-list* , *scalar-integer-variable* = &
  scalar-integer-expression , *scalar-integer-expression* &
  [, *scalar-integer-expression*])

A data-implied DO object is one of:

array-element *structure-component* *data-implied-do*

Things To Know:

1. The type of the initial value must be one that could be used in a corresponding intrinsic assignment.
2. The following must not be initialized: a dummy argument, an object made accessible by use or host association, a function result, an automatic object, an allocatable array, an item in blank COMMON, or a procedure name.
3. The data-object list in a DATA statement is expanded to form a sequence of scalar variables. The data-value list is expanded to form a sequence of constant values. The objects and the values must be in one-to-one correspondence.
4. A BOZ data value can be used only in a DATA statement.



18 Data Representation Models

Data representation models suggest how data are represented in the computer and how computations are performed on the data. The computations to be performed by some of the Fortran intrinsic functions are described in terms of these models. There are three such models in Fortran: the bit model, the integer number system model, and the real number system model.

In a given implementation the model parameters are chosen to match the implementation as closely as possible, but an exact match is not required and the model does not impose any particular arithmetic on the implementation.

The intrinsic functions that provide information about the models are BIT_SIZE, DIGITS, EPSILON, HUGE, MINEXPONENT, MAXEXPONENT, PRECISION, RADIX, RANGE, TINY, EXPONENT, FRACTION, NEAREST, RRSPACING, SCALE, SET_EXPONENT, and SPACING.

Examples:

```
INTEGER, PARAMETER :: W = SELECTED_REAL_KIND (10, 99)
REAL (W), PARAMETER :: EPS = 10.0 * EPSILON (0.0_W)
REAL (W) X0, X1
   . . .
X0 = . . .        ! Initial iterate
   . . .
LOOP: DO
   X1 = . . .     ! Next iterate; assume root is not near zero.
   ! Terminate loop when two consecutive iterates become
   ! close enough that their relative difference is negligible
   ! with respect to the precision used.
   IF (ABS (X1 - X0) <= ABS (X0) * EPS) EXIT LOOP
       X0 = X1
          . . .   ! Perform needed calculations
                  ! to compute the next iterate.
END DO LOOP
```

Related Topics:

Intrinsic Functions: Inquiry and Model

Related Intrinsics:

KIND (X)
SELECTED_INT_KIND (R)
SELECTED_REAL_KIND (P, R)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 13.5.7, 13.7
Fortran 95 Handbook, 13.2, 13.3
Fortran 95 Using F, A.6, A.7

The Bit Model. The bit model interprets a nonnegative scalar data object *a* of type integer as a sequence of binary digits (bits), based upon the model $a = \sum_{k=0}^{n-1} b_k 2^k$, where n is the number of bits, given by the intrinsic function BIT_SIZE and each b_k has a bit value of 0 or 1. The bits are numbered from right to left beginning with 0.

The Integer Number System Model. The integer number system is modeled by $i = s \sum_{k=0}^{q-1} d_k r^k$ where

- i is the integer value
- s is the sign (+1 or –1)
- r is the radix given by the intrinsic function RADIX
- q is the number of digits (integer greater than 0), given by the intrinsic function DIGITS
- d_k is the kth digit and is an integer $0 \le d_k < r$

The Real Number System Model. The real number system is modeled by $x = sb^e \sum_{k=1}^{p} f_k b^{-k}$ where

- x is the real value
- s is the sign (+1 or –1)
- b is the base (real radix) and is an integer greater than 1, given by the intrinsic function RADIX
- e is an integer between some minimum and maximum value, given by the intrinsic functions MINEXPONENT and MAXEXPO-NENT
- p is the number of mantissa digits and is an integer greater than 1, given by the intrinsic function DIGITS
- f_k is the kth digit and is an integer $0 \le f_k < b$, but f_1 may be zero only if all the f_k are zero

19 Defined Operators and Assignment

New operators (e.g., .CONVERT.) may be defined and the meaning of an existing operator (e.g, + or .EQ.) may be extended to data types for which the existing operator is not already defined. Assignment may be extended to new combinations of data types or redefined for user-defined types. The action that occurs for a user-defined operator is specified in a function; user-defined assignment is specified by a subroutine. Interface blocks associate these procedures with an operator or assignment. The interface block may contain an external procedure interface or, if the procedure is in a module, a MODULE PROCEDURE statement. Ordinary operator and assignment syntax may then be used to invoke these procedures.

Examples:

```
! User-defined unary operator .EIGENVALUES. that, when applied
! to an object of type matrix, computes its eigenvalues.
INTERFACE OPERATOR ( .EIGENVALUES. )
   TYPE (VECTOR) FUNCTION FIND_EIGENVALUES(MATRIX_1)
      USE NEW_TYPES
      TYPE (MATRIX), INTENT(IN) :: MATRIX_1
   END FUNCTION FIND_EIGENVALUES
END INTERFACE

TYPE (MATRIX) :: A; TYPE (VECTOR) :: B
B = .EIGENVALUES. A           ! Compute the eigenvalues of A.

INTERFACE OPERATOR ( * )      ! Extend the * symbol.
   MODULE PROCEDURE POLAR_MUL, INTERVAL_MUL
END INTERFACE

INTERFACE ASSIGNMENT ( = )    ! Extend assignment.
   MODULE PROCEDURE ASSIGN_POLAR_TO_COMPLEX
END INTERFACE

TYPE (POLAR) :: P1, P2
TYPE (INTERVAL) :: V1, V2, V; COMPLEX :: C
   . . .
V = V1*V2           ! A defined operation and intrinsic assignment
C = P1*P2           ! A defined operation and a defined assignment
```

Related Topics:

Assignment
Defined Type: Definition
Elemental Procedures
Expressions
Generic Procedures and Operators
Interfaces and Interface Blocks

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 7.1.3, 7.3, 7.5.1.3, 7.5.1.6, 12.3, 14.1.2.3, 14.5, C.8.3.6-7
Fortran 95 Handbook, 7.2.7, 7.3.2, 7.5.2, 12.8.4-5, 14.2.7
Fortran 95 Using F, 7.3, 7.4

Defined Operators and Assignment

Syntax:

A defined operator or assignment interface is:

```
INTERFACE operator-or-assignment-spec
      [ interface-body ]...
      [ MODULE PROCEDURE module-procedure-name-list ]...
END INTERFACE
```

An operator or assignment specification is one of:

```
OPERATOR ( . user-defined-operator-name . )
OPERATOR ( intrinsic-operator-symbol )
ASSIGNMENT ( = )
```

A user-defined operator name is:

letter [*letter*]...

Things To Know:

1. A defined operator may be unary or binary and may appear in expressions. A unary operator is defined by a function with a single nonoptional INTENT(IN) argument; a binary operator is defined by a function with two nonoptional INTENT(IN) arguments.
2. A new operator definition must not redefine an existing operator definition for the same operator types. For example, a new definition for "+" must not involve a function whose arguments are an integer and a real, because "+" has an intrinsic meaning for that pattern. The defined operator .PLUS., however, may have such operands.
3. Note that .TRUE. and .FALSE. must not be used as defined operator names and a name may not contain underscores or digits.
4. The precedence of defined operators is as follows: (a) new operations associated with intrinsic operator symbols have the same precedence as the intrinsic operations; (b) unary operations associated with user-defined operator names have the highest precedence; and (c) binary operations associated with user-defined operators with nonintrinsic names have the lowest precedence.
5. The form of a defined assignment is the same as intrinsic assignment:

   variable = *expression*

   The variable and expression must match an accessible defined assignment interface. The subroutine identified in the interface is executed. It has two nonoptional arguments, the first having intent OUT or intent INOUT and the second having intent IN. The subroutine may be elemental.
6. Operator definitions create additional generic forms of procedures, and the rules for resolving generic procedure references apply.

20 Defined Type: Default Initialization

Default initialization may be specified for objects of user-defined type; the initialization is specified in the type definition. It is not necessary for initialization to be specified for every component. A value may be provided for a nonpointer component; the disassociated status may be indicated for a pointer component. When a program unit containing objects of the type begins execution, all objects of the type, except those allocated during the course of execution, are initialized as indicated in the type definition. Initialization for allocated objects occurs at allocation..

Examples:

```
TYPE COLOR                         ! All components initialized
   INTEGER :: HUE = 0
   INTEGER :: SATURATION = 0
   INTEGER :: BRIGHTNESS = 0
END TYPE COLOR
TYPE (COLOR) PRIMARY, CONTRAST     ! Initialized objects

TYPE BRANCH                        ! Only pointer components
   REAL VALUE                      !   initialized
   TYPE(BRANCH), POINTER :: LEFT => NULL( ), RIGHT => NULL( )
END TYPE BRANCH
TYPE(BRANCH), POINTER :: TREE      ! On allocation, initialized

TYPE GOLD_MINE                     ! Partial initialization
   REAL OVERHEAD
   REAL :: EXPANSION_COST = 0.00
   REAL :: EXPECTED_YIELD = 0.00
   CHARACTER (30) NAME
   TYPE(COLOR) :: G_COLOR = COLOR(7276,58150,58637) ! Overrides
END TYPE GOLD_MINE                 ! default initialization for type COLOR
```

Tip: If a pointer is not initialized, its initial status is undefined. It is not possible to query the status of such a pointer using the ASSOCIATED intrinsic function. Thus it is a good idea to initialize pointers to have a disassociated status.

Related Topics:

Data Initialization
Defined Type: Definition
Defined Type: Structure Constructor
Pointers

Related Intrinsics:

ASSOCIATED (POINTER, TARGET)
NULL (MOLD)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.4.1
Fortran 95 Handbook, 4.4.1
Fortran 95 Using F, 6.3.3, 8.4

Syntax:

A component declaration is:

component-name [(*array-spec*)] [* *character-length*] *component-initialization*

A component initialization is one of:

= *initialization-expr*
=> NULL ( )

Things To Know:

1. A double colon separator must appear in a component declaration when default initialization is specified.
2. When the intrinsic function NULL appears in a component initialization, the optional argument must not be present.
3. An object of a type for which default initialization is specified will be initialized. This will occur even if the definition is private or inaccessible.
4. Default initialization of an array component may be specified by a constant expression consisting of an array constructor, an array named constant (or part of one), or a single scalar that becomes the value of each array component.
5. Default initialization of a pointer component may be specified only by reference to the NULL intrinsic function. A pointer cannot be initialized by default to point to a target.
6. Default initialization applies to automatic and allocated objects. It does not apply to dummy arguments unless they have intent OUT.
7. Unlike explicit initialization in a declaration or DATA statement, default initialization does not imply that the object has the SAVE attribute.
8. If a component in a type definition is of a type for which default initialization is specified, the component may be initialized with different values. This initialization overrides the default initialization specified in the type definition for the type of the component. (See the example of type GOLD_MINE on the previous page.) Similarly explicit initialization in a type declaration overrides default initialization. When one initialization overrides another, it is as if only the overriding initialization were specified.
9. An object of a type for which default initialization is specified or any component of such an object must not appear in a DATA statement.

21 Defined Type: Definition

User-defined data types, officially called derived types, are built of components of intrinsic or user-defined type; ultimately, the components are of intrinsic type. This permits the creation of objects, called **structures**, that contain components of different types (unlike arrays, which are homogeneous). It also permits objects, both scalars and arrays, to be declared to be of a user-defined type and operations to be defined on such objects. A component may be a pointer, which provides for dynamic data structures, such as lists and trees. Defined types provide the basis for building abstract data types.

Examples:

```
TYPE TEMP_RANGE                ! This is a simple example of
   INTEGER HIGH, LOW           !    a defined type with two
END TYPE TEMP_RANGE            !    components, HIGH and LOW.

TYPE TEMP_RECORD               ! This type uses the previous
   CHARACTER(LEN=40) CITY      !    definition for one component.
   TYPE (TEMP_RANGE) EXTREMES(1950:2050)
END TYPE TEMP_RECORD

TYPE LINKED_LIST               ! This one has a pointer compon-
   REAL VALUE                  !    ent to provide links to other
   TYPE(LINKED_LIST),POINTER :: NEXT!  objects of the same type,
END TYPE LINKED_LIST           !    thus providing linked lists.

TYPE, PUBLIC :: SET; PRIVATE         ! This is a public type whose
   INTEGER CARDINALITY               !   component structure is
   INTEGER ELEMENT ( MAX_SET_SIZE ) !   private; defined
END TYPE SET                         !   operations provide
                                     !   all functionality.

! Declare scalar and array structures of type SET.
TYPE (SET) :: BAKER, FOX(1:SIZE(HH))
```

Related Topics:

Argument Association
Defined Type: Default Initialization
Defined Operators and Assignment
Defined Type: Objects
Defined Type: Structure Component
Defined Type: Structure Constructor
Generic Procedures and Operators
Interfaces and Interface Blocks
Modules
PUBLIC and PRIVATE Attributes and Statements
Scope, Association, and Definition Overview
USE Statement and Use Association

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.4, C.1.1, C.8.3.3, C.8.3.7
Fortran 95 Handbook, 4.4, 11.6.5.3-5
Fortran 95 Using F, 6.2

Syntax:

A defined-type definition is:

```
TYPE [[ , access-spec ] :: ] type-name
   [ PRIVATE ]
   [ SEQUENCE ]
   component-declaration
   [ component-declaration ]...
END TYPE [ type-name ]
```

A component declaration is:

```
type-spec [[ , component-attribute-list ] :: ] component-list
```

A component attribute is one of:

```
POINTER
DIMENSION ( array-spec )
```

Things To Know:

1. A type name may be any legal Fortran name as long as it is not the same as an intrinsic type name or another local name in that scoping unit. A type definition forms its own scoping unit, which means that the component names are not restricted by the occurrence of any names outside the type definition; the scoping unit has access to host objects by host association so that named constants and accessible types may be used in component declarations.
2. A component array specification must be explicit shape or deferred shape; a deferred-shape component must have the POINTER attribute.
3. A component may itself be a defined type. If, in addition, the POINTER attribute is specified, the component type may even be that of the type being defined.
4. Default initialization may be specified for a component (see Defined Type: Default Initialization).
5. If a type definition is in a module, it may contain a PUBLIC or PRIVATE attribute or an internal PRIVATE statement.
6. The internal PRIVATE statement in a type definition makes the components unavailable outside the module even though the type itself might be available.
7. Two type definitions do not define the same type, even if they have the same name and the same components. To declare two variables to be the same type, for example, access the same type definition by use or host association.
8. The SEQUENCE statement is used: (a) to allow objects of this type to be storage associated, or (b) to allow actual and dummy arguments to have the same type without use or host association (see Argument Association, item 5 of Things To Know).
9. Operations on defined types are defined with procedures and given operator symbols with interface blocks.

22 Defined Type: Objects

Defined-type objects, called **structured objects** or simply **structures**, may be declared, assigned values, used as procedure arguments, and returned as function results. Thus they may be used in much the same way as intrinsic objects. The structure constructor provided automatically for each defined type, and having the same name as the type, may be used to construct data values of that type. Assignment is intrinsically defined for each defined type but may be redefined by the user. Operators appropriate to a defined type may be defined by procedures with the appropriate interfaces. Defined-type input/output is accomplished component by component.

Examples:

```
TYPE WEATHER                    ! WEATHER is a simple defined type
   CHARACTER(LEN=32) PLACE      !   with two character components
   INTEGER HIGH_TEMP, LOW_TEMP  !   and two integer components.
   CHARACTER(LEN=16) CONDITIONS
END TYPE WEATHER

TYPE (WEATHER) JULY(NUM_WS,31) ! A WEATHER array for July

JULY(:,:) % LOW_TEMP = -40     ! Initialize all low temps in JULY

TYPE POLAR                  ! POLAR is a defined type with two
   PRIVATE                  !   real components that cannot be
   REAL RHO, THETA          !   directly accessed in POLAR
END TYPE POLAR              !   objects outside the module.

TYPE POINT                  ! POINT is a defined type with
   REAL X, Y                !   three components, one of which
   TYPE (POLAR) P           !   is itself of defined type.
END TYPE POINT

TYPE (POLAR) R, Q(500)        ! Two variables of type POLAR
TYPE (POINT) A, B, T(100,100) ! Three variables of type POINT

B = POINT(0.,0.,POLAR(0.,0.)) ! Use of two structure constructors
```

Related Topics:

Defined Operators and Assignment
Defined Type: Definition
Defined Type: Structure Component
Defined Type: Structure Constructor
Generic Procedures and Operators
Interfaces and Interface Blocks
Modules
USE Statement and Use Association

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.4, 5.1.1.7, 6.1.2, C.1.1, C.8.3.3, C.9.12
Fortran 95 Handbook, 4.4, 5.1.7, 6.3, 11.6.5.3-5, 6
Fortran 95 Using F, 6.1

Syntax:

A defined-type type declaration statement is:

TYPE (*type-name*) [*attribute-list* ::] *entity-list*

A structure constructor is:

type-name (*expression-list*)

1. Once a type has been defined and made available, possibly via host or use association, objects of that type may be declared and used subject only to normal type restrictions, such as valid operations for that type and actual and dummy argument type matching.
2. A defined-type object may be an array, which may be deferred shape (pointer or allocatable), assumed shape (dummy argument), or assumed size (dummy argument).
3. When a defined-type object is used as a procedure argument, the types of the associated actual and dummy arguments must be the same. For sequenced types (with the SEQUENCE statement in the type definition) different physical type definitions may be used for the actual and dummy arguments, as long as both type definitions specify identical type names, components, and component order. For nonsequence types the same physical type definition must be used, typically accessed via host or use association, for both the actual and dummy arguments.
4. If all the expressions in the expression list of a structure constructor are initialization expressions, the value constructed is a constant and thus can be the value of a named constant.
5. Defined-type objects that have pointer components cannot be a list item in input/output statements; I/O for such objects must be done by other means, such as user-defined procedures.
6. Internally private objects (those whose type definition appears in a module and contains an internal PRIVATE statement) can be declared and used outside the module, but the structure of such objects is not known outside the module and thus the components cannot be referenced and the constructor cannot be used outside the module. Consequently, the last line of the example can be present only in the module in which the definition of POLAR occurs.

23 Defined Type: Structure Component

A structure component is a component of an object of user-defined type. Where the name of the component is accessible, the component may be referenced and used like any other variable. The reference may appear in an expression or as the variable on the lefthand side of an assignment statement. In the latter case, a value is assigned to the component. The name of the component is accessible in a scoping unit that contains the type definition, whose host contains the type definition, or where the type definition is publicly accessible by use association. A component may be a scalar, an explicit-shape array, or, if it has the POINTER attribute, a deferred-shape array.

Examples:

```
TYPE REG_FORM           ! REG_FORM is a defined type.
   CHARACTER (30) LAST_NAME, FIRST_NAME
   INTEGER ID_NUM       ! Note that ID_NUM in REG_FORM does not
   CHARACTER (2) GRADE  ! conflict with ID_NUM in CLASS because
END TYPE REG_FORM       ! each type definition is a scoping unit.

TYPE CLASS                        ! CLASS is a simple defined type
   INTEGER YEAR, QUARTER, ID_NUM !  that includes another
   CHARACTER(30) INSTRUCTOR      !  defined type as a component.
   TYPE (REG_FORM) STUDENT(40)
END TYPE CLASS

TYPE (CLASS) ALGEBRA, CHEMISTRY  ! Two structures of type CLASS
TYPE (REG_FORM) TRANSFERS(20)    ! An array of structures

ALGEBRA % INSTRUCTOR = "Brown"              ! Some typical uses
ALGEBRA % ID_NUM = 101                      !   of structure
ALGEBRA % STUDENT(1) % ID_NUM = 593010040   !   components
CHEMISTRY % STUDENT(39) % LAST_NAME = "Flake"
CHEMISTRY % STUDENT(39) % GRADE = "F-"
   . . .
ALGEBRA % STUDENT(27:33) = TRANSFERS(1:7) ! An array assignment
ALGEBRA % STUDENT(6:8) % GRADE = "B+"     ! The B+ is broadcast.
PRINT *, CHEMISTRY % STUDENT(1:33)        ! Print 33 students.
```

Related Topics:

Character Substring
Defined Type: Definition
Defined Type: Objects
Variables

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 6.1.2, C.3.1
Fortran 95 Handbook, 6.3
Fortran 95 Using F, 6.3.1

Syntax:

A structure component reference is:

part-reference [% *part-reference*]...

A part reference is:

part-name [(*section-subscript-list*)]

A section subscript is one of:

subscript *subscript-triplet* *vector-subscript*

A subscript triplet is:

[*subscript*] : [*subscript*] [: *subscript*]

A vector subscript is:

rank-one-integer-array

A substring of a structure component is:

part-reference [% *part-name*]... (*starting-position* : *ending-position*)

Things To Know:

1. In a structure component reference, each part name except the rightmost one must be of defined type, each part name except the leftmost one must be the name of a component of the preceding defined type, and the leftmost part name is the name of a structured object.
2. The type and type parameters of a structure component are those of the rightmost part name. A structure component is a pointer only if the rightmost part name has the POINTER attribute.
3. If the leftmost part name has the INTENT, TARGET, or PARAMETER attribute, the structure component has that attribute.
4. In a structure component reference, only one part may be array valued, in which case the reference is an array reference. This is an arbitrary restriction in the language, imposed for simplicity.
5. If a structure component reference is an array reference, no part to the right of the array part may have the POINTER attribute. It is possible to declare an array of structures that have a pointer component, but it is not possible to have an array-valued reference to such an object. The reason for this is that Fortran allows pointers to arrays, but does not provide for arrays of pointers.
6. If the type definition is in a module and contains an internal PRIVATE statement, the internal structure, including the number, names, and types of the components are not accessible outside the module. If the type itself is public, objects of this type may be declared and used outside the module but none of the components may be accessed directly.

24 Defined Type: Structure Constructor

A structure constructor is used to construct a value of user-defined type. The value is constructed from a sequence of values, one for each component of the type. A structure constructor is the name of the type followed by a list of component values in parentheses. If a component is of user-defined type, an embedded structure constructor is used to specify that component. If a component is an array, an array constructor is used to specify that component.

Examples:

```
TYPE SKY
   CHARACTER (9) SKY_COLOR
   REAL CLOUD_COVER
END TYPE SKY
TYPE (SKY) :: THE_SKY_TODAY = SKY (“CYAN”, 0.27)

TYPE POSTAL_INFO               ! A simple defined type with
   REAL WEIGHT                 !   two real components
   REAL DIMENSIONS(3)
END TYPE POSTAL_INFO

TYPE TOY
   INTEGER CATALOG_NUMBER
   REAL PRICE                  ! This type has four components,
   TYPE (POSTAL_INFO) TO_MAIL  !  one of which is of defined type
   INTEGER AGE_RANGE(2)        !  and another which is an array.
END TYPE TOY

TYPE (TOY) RED_WAGON           ! Declarations used in the two
REAL RW_SZ (3)                 !    examples below

READ(*,*) RW_SZ

! In the following statement, a value of type
!   TOY is constructed. It contains an embedded
!   structure constructor for the component TO_MAIL
!   and an array constructor for the component AGE_RANGE.

RED_WAGON = TOY (10159, 39.99, POSTAL_INFO(7.3,RW_SZ), (/4,12/))
```

Related Topics:

Array: Constructors
Defined Operators and Assignment
Defined Type: Definition
Defined Type: Structure Component
Expressions

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.4.4, 4.5
Fortran 95 Handbook, 4.4.4, 4.5-6
Fortran 95 Using F, 6.3.2

Syntax:

A structure constructor is:

type-name (expression-list)

Things To Know:

1. A structure constructor is associated with each defined type and is automatically provided when the type is defined.
2. A defined type must be previously defined in or accessible to (via host or use association) the scoping unit in which a structure constructor for that type appears.
3. Any expression may appear in the list as long as it agrees in number, order, and rank with the components of the type. If necessary, each value is converted according to the rules for intrinsic assignment to a value that agrees in type and type parameters.
4. If a component is an array (and not a pointer), the corresponding value must agree in shape.
5. If a component is a pointer, it is pointer assigned. The value in the expression list must be an allowable target for the pointer; a constant is not an allowable target. A reference to the intrinsic function NULL may be used to specify the disassociated state for a pointer component.

```
TYPE LINK
   REAL VALUE
   TYPE (LINK), POINTER :: NEXT
END TYPE LINK

TYPE (LINK) HEAD_OF_LIST

HEAD_OF_LIST = LINK ( 0.0, NULL( ) )
```

6. If all the expressions in the expression list of a structure constructor are initialization expressions, the value constructed is a constant and thus can be the value of a named constant.

```
TYPE (POSTAL_INFO), PARAMETER :: PACKAGE = &
      POSTAL_INFO (9.5, (/10.0, 5.5, 2.25/) )
```

25 DIMENSION Attribute and Statement

A variable declared to have the DIMENSION attribute is an array. An array is a collection of scalar elements all of the same type and kind; the type may be intrinsic or user defined. It is not necessary for the keyword DIMENSION to appear in a declaration for an array to give it the DIMENSION attribute. There are several ways to specify this attribute as well as the rank, possibly the extents, and the bounds of an array. These may be specified in a type statement (possibly containing the DIMENSION attribute) or in a DIMENSION, ALLOCATABLE, COMMON, POINTER, or TARGET statement.

Examples:

```
REAL A (20,2), B (20,2), C (20,2)    ! These 2 declarations
REAL, DIMENSION (20,2) :: A,B,C      !   are equivalent.

DIMENSION X(100), Y(100), Q(:,:,:)   ! X and Y are 1-dimensional.
                                     ! Q is deferred shape and
                                     !   of rank 3.

INTEGER JJ (0:100, -1:1)             ! Lower bounds are specified
                                     !   for JJ.

LOGICAL L                            ! L is 4-dimensional
ALLOCATABLE L(:,:,:,:)               !   and allocatable.

COMPLEX S                            ! S has explicit shape and
TARGET :: S(10,2)                    !   is a target.

DOUBLE PRECISION D                   ! D has 5 dimensions and
COMMON /STUFF/ D(2,3,5,9,8)          !   is declared in common.
```

Tip: There are a number of ways to convey the DIMENSION attribute (arrayness) to variables as illustrated by the examples above. It is best to select one way and use it consistently.

Related Topics:

ALLOCATABLE Attribute and Statement
Array Overview
Array: Declaration Forms
COMMON Statement
POINTER Attribute and Statement
TARGET Attribute and Statement

Related Intrinsics:

LBOUND (ARRAY, DIM)
SHAPE (SOURCE)
SIZE (ARRAY, DIM)
REPEAT (STRING, NCOPIES)
UBOUND (ARRAY, DIM)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.4, 5.2.5
Fortran 95 Handbook, 5.3.2, 6.4
Fortran 95 Using F, 4.1.3

Syntax:

A type declaration statement with the DIMENSION attribute is:

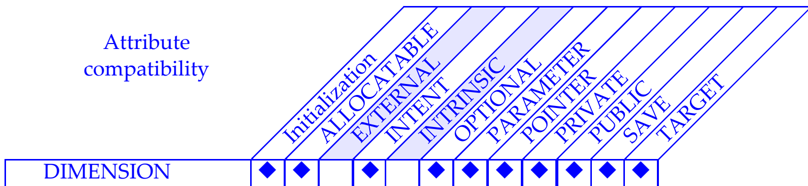
type , DIMENSION (*array-spec*) [, *attribute-list*] :: *entity-list*

A DIMENSION statement is:

DIMENSION [::] *array-name* (*array-spec*) [, *array-name* (*array-spec*)]...

Things To Know:

1. An array specification is either explicit shape, assumed shape, deferred shape, or assumed size. (See Array: Declaration Forms.)
2. The number of dimensions is called the rank. The maximum number of dimensions (rank) is 7.
3. Array specifications can appear following a name to establish array bounds. In a type declaration, such array specifications take precedence over an earlier DIMENSION attribute with bounds declared in the same statement.
4. The extent is the number of elements in a particular dimension. The shape is a vector of the corresponding extents. An array declaration establishes the rank of the array and may establish its shape, extents, upper bounds, and lower bounds. The extent in any dimension is the upper bound minus the lower bound plus 1 in that dimension.
5. In operations involving arrays, there is no implied ordering of execution as there is when loop indexing is involved, except as noted in Array Overview, item 5 of Things to Know.
6. A function may return an array value.

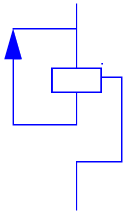


26 DO Construct

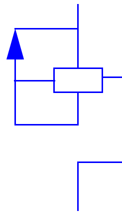
The DO construct may be used to execute a code block repeatedly. The ways to control how many times the block is executed are:

- **simple loop** -There is no control; repeated execution of the block ceases when an exit occurs.
- **indexed loop** – A loop count is calculated that controls the number of times the block is executed, unless a prior exit occurs. A loop variable is incremented or decremented after each execution.
- **while loop** – A condition is tested before each execution of the block; when it is false, execution ceases. An exit may occur at any time.

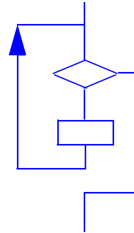
The code block may be executed zero or more times. A DO construct may be named. A CYCLE statement may appear at any point in the block to start the next execution of the block. A loop may contain a statement, such as EXIT, that terminates the loop. Some control flow possibilities are:



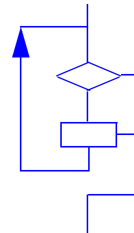
simple loop
with EXIT



with EXIT
and CYCLE



indexed or
WHILE loop



WHILE loop
with EXIT

Examples:

```
! Simple loop with EXIT
DO
   CALL NEW_I (I)
   IF (GET_LOC(I) == SEARCH) EXIT
END DO

! Simple loop with CYCLE and
!   EXIT of OUTER loop
INNER: DO
   READ *, VAL
   IF (VAL < THRESHOLD) CYCLE
   IF (VAL > VMAX) EXIT OUTER
   CALL CALC (VAL, ANS)
END DO INNER
```

```
! Indexed loop
INSIDE: DO I = K-1, 8, -1
  A(I) = A(I-1)*A(I+1)/A(I)
END DO INSIDE

! While loop
DO WHILE (SUM < 100.0)
   SUM = SUM+GET_NEXT(UNIT)
END DO

! Indexed loop with EXIT
DO I = 1, 30
   N(I) = 0; READ *, J
   IF (J < 0) EXIT; N(I)=J
END DO
```

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 8.1.4, C.5.1, C.5.3-4
Fortran 95 Handbook, 8.5
Fortran 95 Using F, 2.4

Syntax:

A DO construct is one of:

```
[ do-construct-name : ] DO [ loop-control ]
    block
END DO [ do-construct-name ]

DO label [ loop-control ]
    block
label CONTINUE
```

Loop control is one of:

```
scalar-integer-variable-name = scalar-integer-expression , &
              scalar-integer-expression [ , scalar-integer-expression ]

WHILE ( scalar-logical-expression )
```

The EXIT and CYCLE statements are:

```
EXIT [ do-construct-name ]
CYCLE [ do-construct-name ]
```

Things To Know:

1. The scalar variable in an indexed loop must be of type integer. The scalar integer expressions denote the initial index value, the limiting value, and the increment, which if not present, is assumed to be 1. It may be negative, in which case the initial value is normally greater than the limiting value. The loop count is calculated from the values of the scalar integer expressions on entry to the loop; if the values change during execution of the block, the loop count is not affected.
2. The scalar logical expression in a WHILE loop is tested prior to each execution of the block. If the condition is true, the block is executed.
3. A loop exit occurs when a statement such as EXIT or RETURN is executed.
4. Control constructs may be nested, in which case a program is easier to read if the inner construct is indented. Construct names must be used when exiting or cycling an outer loop from inside a nested inner loop.
5. If a construct name appears on a DO statement, the same name must appear on the corresponding END DO statement.
6. A number of other possibilities for loops are permitted such as labeled END DO statements, a labeled statement other than END DO or CONTINUE as the loop termination statement, and nested loops terminating on a single labeled statement, but these are considered poor programming practice and some are obsolescent.

27 Dynamic Objects

Dynamic objects are declared but no space is set aside for them at compile time. Their type and rank are declared; their size and location are determined at execution time. There are three classes of dynamic objects:

allocatable arrays   pointers   automatic objects

Pointers are the most general of the dynamic objects. Pointer targets may be scalar objects or arrays. Their size is determined when they are allocated or pointer assigned to a target. Allocatable arrays provide a more restricted and simpler means of dealing with dynamic arrays. Their size is determined when they are allocated. Automatic objects may be arrays of any type or scalar character objects. Their size or length is determined on entry to a procedure when they are created; they disappear on exit from the procedure.

Examples:

```
REAL, ALLOCATABLE :: PLOT (:, :)        ! Allocatable array
REAL, POINTER     :: REGION (:, :)      ! Array pointer
REAL, TARGET      :: GRID (100, 100)    ! Target array
READ *, M, N; ALLOCATE (PLOT (M, N) )
DO J = 1, N/3, 3;     DO I = 1, M/3, 3
      REGION => GRID (I:I+2, J:J+2)
         . . .
SUBROUTINE TASK (X)
   REAL, INTENT (INOUT) :: X (:, :)
   REAL WORKING (SIZE(X,1), SIZE(X,2))  ! Automatic real array
      . . .
SUBROUTINE ERROR_HANDLER (REASON)
   CHARACTER (*), INTENT (IN) :: REASON
   CHARACTER (LEN(REASON) + 13) MSG     ! Automatic character
   MSG = “FATAL ERROR: “ // REASON      !   object
   PRINT *, MSG
END SUBROUTINE ERROR_HANDLER
```

Related Topics:

ALLOCATABLE Attribute and Statement
ALLOCATE and DEALLOCATE Statements
Array Overview
Character Type and Constants
Pointers
Pointer Association

Related Intrinsics:

ALLOCATED (ARRAY)
ASSOCIATED (POINTER, TARGET)
LEN (STRING)
SIZE (ARRAY, DIM)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1, 5.1.2.4.3, 5.1.2.7, 5.2.7, 6.3, 7.5.2, 13.14.9, 13.14.13, 14.6.2, C.1.3, C.2, C.3.2, C.4.3-4, C.8.3.4, C.9.5, C11.1.4
Fortran 95 Handbook, 5.3.1.3, 5.9, 6.5, 7.5.3, 14.3.2, A.9, A.13
Fortran 95 Using F, 4.1.5, 8

Things To Know:

1. The ALLOCATABLE attribute may not be given to a scalar object or a dummy argument; the POINTER attribute may.
2. An allocatable array may not appear in a common block or be a component in a type definition. A pointer may appear in common and be a component. A function may not return an allocatable array. It may return a pointer.
3. Allocatable arrays and pointers may be saved. They cannot be data initialized. Automatic objects cannot be saved or data initialized.
4. The ALLOCATED intrinsic inquiry function is used to inquire about the allocation status of an allocatable array. The ASSOCIATED intrinsic inquiry function is used to inquire about the association status of a pointer, whether a pointer is associated with a given target, or whether two pointers are associated with the same target.
5. The allocation status of an allocatable array may be unallocated, allocated, or undefined. It is unallocated initially. Allocating an allocated array causes an error condition; an allocated allocatable array must be deallocated before it can be allocated again. An error condition will also result if there is an attempt to deallocate an unallocated array. A local allocated allocatable array without the SAVE attribute is deallocated when a RETURN or END statement is executed and no other scoping unit, currently executing, has access to the array.
6. The association status of a pointer may be undefined, disassociated, or associated. It is undefined initially. A pointer becomes disassociated after execution of a NULLIFY or DEALLOCATE statement or a pointer assignment statement with the null target or with a target of NULL(). It is not an error to allocate an associated pointer. It is an error to deallocate a pointer unless its target is a whole allocated object originally declared as a pointer. A pointer with undefined status may subsequently be nullified, allocated, or pointer assigned.
7. Automatic objects may appear only in procedures or procedure interfaces. They are frequently used to create working storage in a procedure.
8. Allocatable arrays, automatic arrays, and pointers may be of a derived type for which default initialization has been specified. Upon creation of such an object (via an ALLOCATE statement or entry to a procedure), the components of such objects are initialized as specified by the default initialization.

28 Edit Descriptors: Control

The input/output control edit descriptors allow the programmer to arrange the way values appear on a line or in a record. These descriptors control skipping, tabbing, scale factors, optional signs, and the interpretation of blanks and do not transfer any data values.

Examples:

```
CLOUDS = 4444.21;  RAIN = -10.2
D = 1.1; I = 9
WRITE (6,100) CLOUDS, RAIN, D
100 FORMAT (F10.2, 3X, F5.1, T19, F4.1)
```

The output record is: *bbb*4444.21*bbb*–10.2*b*1.1

```
PRINT '("First row:", 4I5,(:/" Next row:", 4I5) )', (A(J,1:4),J=1,N)
       ! Output for N=2
       ! First row:    1    2    3    4
       !  Next row:    5    6    7    8
PRINT "(S,I4, SP,I4, SS, I4, SP, I4)", 4, 4, 4, -4
! Output (process default is to suppress optional plus sign)
!    4  +4   4  -4

READ "(BZ, I4, BN, I4)", I, J
! Input (after read, I and J have the values 40 and 871 )
!   4 8 71
READ "(TR4,I4, TL8, I4)", I, J
! Input (after read, I and J have the values 392 and 81 )
!   81 392
PRINT "(A, 2X, I10)","I is",I
```

The output record is: Ibisbbbbbbbb392

```
I = 3; K = -3; J = 4     ! SP controls the plus sign.
PRINT "(SP,3I3)",I,K,J   ! After SP in a format specification,
   . . .                 !   + is printed, as in +3 -3 +4
                         ! An SS suppresses printing +
PRINT "(SS, 3I3)",I,K,J !   The output would be  3 -3  4

PRINT "(2P,E15.1)", .142E+01       ! 14.E-01 is printed.
```

Tip: For most uses of the *k*P edit descriptor, the EN or ES descriptor can be used instead, and are better because they affect only the form of the item being formatted and not subsequent items in the input/output list.

Related Topics:

Edit Descriptors: Data and Character String   Format Specifications

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 10.2.1, 10.4, 10.6, C.10.2
Fortran 95 Handbook, 10.2.2, 10.9
Fortran 95 Using F, 9.8.1, 9.8.12-15

Syntax:

A control edit descriptor is one of:

T *n*	Move to position *n*
TL *n*	Move left *n* positions
TR *n*	Move right *n* positions
n X	Move right *n* positions
[*r*] /	End current record, skip *r*–1 records and start a new record
:	Stop format processing when there are no more list items
S	Printing optional plus sign is processor dependent
SP	Print optional plus sign
SS	Do not print optional plus sign
*k*P	A scale factor of *k* is applied
BN	Ignore nonleading blanks in numeric input
BZ	Nonleading blanks in numeric input are treated as zeros

n and *r* are unsigned integer literal constants with no kind parameter.

k is a possibly signed integer literal constant with no kind parameter.

Things To Know:

1. The scale factor is a signed integer literal constant, taking the value zero initially. After a *k*P descriptor, all numeric fields that follow (except those controlled by EN and ES edit descriptors) have a scale factor of *k* until the next *k*P occurs. A scale factor has no effect if an input field has an exponent. Otherwise, on input, the number stored in the variable is 10^k times the value of the number in the input record. On output, if the value is printed with an E or D edit descriptor, the fractional part is multiplied by 10^k and the exponent part is reduced by *k*.
2. If the previous READ or WRITE is nonadvancing, the tab left edit descriptor cannot move the position further left than the position where the current data transfer began.
3. The colon (:) edit descriptor is used to stop format processing when the list of items in the READ or WRITE statement is exhausted. If there are list items remaining, the colon has no effect.
4. Blanks may be ignored or treated as zero depending on the BN and BZ edit descriptors in a format specification.

   BN ignore nonleading blanks in numeric fields

   BZ treat nonleading blanks in numeric fields as zero

29 Edit Descriptors: Data and Character String

The data edit descriptors convert data from the internal representation to characters on formatted output, and to the internal representation from characters on formatted input. The data in a formatted record consist of characters that are sometimes printed on a line printer for ease of reading. Formatted records may be transferred to an external device as well.

A character string edit descriptor places text in a formatted output record. It is used to create text and headings in an output file or printed page to identify results.

Examples:

```
INTEGER :: TESTS = 13
WRITE (6,100) TESTS
100 FORMAT (' PASS =', I5)      ! Prints: PASS =   13

X = 3218.
Y = 3.2
PRINT 101, X, Y
101 FORMAT (E15.4, F6.1)        ! Prints:      0.3218E+04   3.2

LOGICAL FOUND, SET
READ (5,"(2L7)") FOUND, SET
! Reads .TRUE. or .TEACH with the same result
! Reads .FALSE  or FRENCH with the same result

NAME = " SMITH"
X = 0.4691
PRINT "(A6,EN10.2)", NAME, X    ! Prints:   bSMITH469.10E-03
PRINT "(A6,ES10.2)", NAME, X    ! Prints:   bSMITHbb4.69E-01
PRINT "(A6,G10.4)", NAME, X     ! Prints:   bSMITH0.4691bbbb

L = 10         ! The digits printed for L using O and B
               !  depend on the hardware representation
               !  of the positive integer 10.  For most
               !  current machines, the following results
               !  will be obtained.
PRINT "(O4, B7)", L, L          ! Prints:   bb12bbb1010
```

Related Topics:

Edit Descriptors: Control   Format Specifications

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 10.2.1, 10.5, 10.7
Fortran 95 Handbook, 10.2.1, 10.3, 10.6–10.8, 10.9.7
Fortran 95 Using F, 9.8.1, 9.8.4–11

29 Edit Descriptors: Data and Character String

Syntax:

A data edit descriptor is one of:

I *w* [. *m*]	Decimal integer conversion
B *w* [. *m*]	Binary integer conversion
O *w* [. *m*]	Octal integer conversion
Z *w* [. *m*]	Hexadecimal integer conversion
F *w* . *d*	Real conversion
E *w* . *d* [E*e*]	Real conversion with an exponent
EN *w* . *d* [E*e*]	Engineering notation
ES *w* . *d* [E*e*]	Scientific notation
G *w* . *d* [E*e*]	General conversion, all types
L *w*	Logical conversion
A [*w*]	Character conversion
D *w* . *d*	Similar to E*w*, with D as the exponent

A character string edit descriptor is one of:

[*kind-parameter* _] ' [*representable-character*]... '
[*kind-parameter* _] " [*representable-character*]... "

Things To Know:

1. *w*, *m*, *d*, and *e* are unsigned integer literal constants (no kind parameters). The interpretation of *w*, *m*, *d*, and *e* is:

   w field width, except when *w* is zero
   m least number of digits in the field
   d number of decimal digits in the field
   e number of digits in the exponent

2. On input, *w* must not be zero. On output, if it is zero, the processor selects the field width to be used; otherwise, it specifies the field width.
3. For numeric input editing, leading blanks are not significant and a blank field is zero. A decimal point overrides any descriptor.
4. For numeric output editing, negative zero may be produced if the processor supports it. The number is right justified in the field. The exponent sign is produced. If the field is too small, the field is filled with asterisks.
5. For EN, the exponent is divisible by 3, and $1 \le |\text{significand}| < 1000$. For ES, $1 \le |\text{significand}| < 10$.
6. G is a flexible format specification where the exact format depends on the size of the values in the list. An F or a descriptor with an exponent may be used by generalized editing.
7. In character string editing, doubling the quote or apostrophe permits its appearance within the character string.
8. The B, O, and Z edit descriptors are called BOZ edit descriptors. They can be used to read integer values from or write integer values to formatted records using a binary, octal, or hexadecimal format.

30 Elemental Procedures

Elemental procedures are procedures with scalar dummy arguments. They may be referenced with array actual arguments of the same shape. Elemental procedures called with scalar actual arguments produce a scalar result. Elemental procedures called with array actual arguments produce an array result conformable with the arguments. An elemental procedure is a pure procedure. An elemental procedure must not be recursive.

Examples:

```
ELEMENTAL REAL FUNCTION MODERN_WORLD (PLANET)
! PLANET is a scalar variable.

ELEMENTAL FUNCTION ODD_SINE (X)
REAL, INTENT (IN) :: X
   . . .
END FUNCTION ODD_SINE

U = R + ODD_SINE (ST)
AU = AR + ODD_SINE (AT)
! A scalar value will be returned if ST is a scalar variable.
! An array value will be returned if AT is an array.

ELEMENTAL PURE SUBROUTINE SOURCE (A, B, C)
REAL, INTENT (IN) :: A, B
REAL, INTENT (OUT) :: C
   . . .
END SUBROUTINE SOURCE

REAL, DIMENSION (2,2,3) :: S, T
REAL Q
CALL SOURCE (Q, S, T)
```

Tip: A user can write one scalar procedure that can accept actual arguments that are arrays of any rank. There is no need to write a version of the procedure for each rank, as is necessary when creating a generic interface. Elemental procedures are particularly useful for expressing operations on parallel computers.

Related Topics:

Argument Association
Array: Data-Parallel Operations
Functions
Internal Procedures
Intrinsic Function Overview
Module Procedures
Pure Procedures
Subroutines

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 12.7
Fortran 95 Handbook, 12.1.7, 12.5, 12.7.7
Fortran 95 Using F, 7.1.2, A.2

Syntax:

An elemental function statement is:

```
ELEMENTAL [ PURE ] [ type-spec ] FUNCTION function-name &
   ( dummy-argument-name-list ) [ RESULT ( result-name ) ]
```

An elemental subroutine statement is:

```
ELEMENTAL [ PURE ] SUBROUTINE subroutine-name &
   ( dummy-argument-name-list )
```

Things To Know:

1. There must be at least one scalar dummy argument. The argument intent of all dummy arguments must be specified.
2. A dummy argument must not be a pointer, a procedure, or an alternate return (*). Note that the * is an obsolete feature.
3. An elemental procedure must have an explicit interface.
4. All actual arguments must be conformable. If one actual argument is an array, the result is as if the scalar-valued function is executed for each element of the array. However, no order of execution is implied. The result must not be a pointer.
5. If the procedure is a subroutine, either all actual arguments must be scalar, or all intent OUT or intent INOUT arguments must be arrays of the same shape. Each intent OUT or intent INOUT result must be as if the subroutine were called separately in any order for corresponding elements of the array actual arguments.
6. An example elemental subroutine.

```
ELEMENTAL SUBROUTINE COPE (A, B, C)
REAL, INTENT (OUT) :: B
REAL, INTENT (IN)  :: A, C
   . . .
END SUBROUTINE COPE

REAL Y(10,10), Z(10,10), X
CALL COPE (X, Y, Z)
```

7. No order of execution is implied in using elemental subroutines instead of DO loops where a specific order of execution is implied. Executing the following DO constructs will produce the same results.

```
DO I = 1,10
   DO J = 1,10
      CALL COPE (X, Y(I,J), Z(I,J))
   END DO
END DO
```

31 EQUIVALENCE Statement

The EQUIVALENCE declaration causes two or more variables to share the same storage space. The objects are storage associated. No type conversion is implied among the members of an equivalence set in an EQUIVALENCE statement.

Examples:

```
INTEGER IX
REAL X(4), Y(5), Z(3)
EQUIVALENCE ( X(3), Y(2) )  ! X(3) and Y(2) share storage
EQUIVALENCE ( IX, Z(2) )    ! IX and Z(2) share storage
```

The following alignment occurs for X and Y in storage:

X(1)	X(2)	X(3)	X(4)		
	Y(1)	Y(2)	Y(3)	Y(4)	Y(5)

so that X(2), and Y(1), X(3) and Y(2), X(4) and Y(3) are the same items. The last statement causes only IX and Z(2) to share storage.

```
CHARACTER (LEN = 4) :: A, B
CHARACTER (LEN = 3) :: C (2)
EQUIVALENCE (A, C (1)), (B, C (2))
```

causes the alignment illustrated below:

A(1:1)	A(2:2)	A(3:3)	A(4;4)			
C(1)(1:1)	C(1)(2:2)	C(1)(3:3)	C(2)(1:1)	C(2)(2:2)	C(2)(3:3)	
			B(1:1)	B(2:2)	B(3:3)	B(4:4)

As a result, the fourth character of A, the first character of B, and the first character of C(2) all share the same character storage unit.

Tip: The EQUIVALENCE statement is an early form of storage sharing and first appeared in Fortran 66. With the introduction of modules, dynamic storage, pointers, structures, and the TRANSFER intrinsic function, its use is not recommended. It remains in Fortran so that older Fortran programs may conform to the standard. COMMON and EQUIVALENCE statements are the source of many programming errors; their use is discouraged.

Related Topics:

COMMON Statement   Storage Association

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.5.1, 14.6.3
Fortran 95 Handbook, 5.11.3, 5.11.5

EQUIVALENCE Statement

Syntax:

An EQUIVALENCE statement is:

`EQUIVALENCE` (*equivalence-object* , *equivalence-object-list*)

An equivalence object is one of:

variable-name
array-element
substring

Things To Know:

1. If the objects are not default intrinsic types, the types must be the same with the same kind parameter.
2. In an equivalence set, an array without subscripts means the first element of the array, which is not necessarily A(1, 1, ..., 1).
3. Each subscript or substring range in an equivalence object list must be an integer scalar initialization expression.
4. The same storage unit must occur only once. For example, A(1) and A(2) cannot share the same storage unit.
5. An object in the set must not be a dummy argument, a pointer, an allocatable array, a structure containing a pointer, a structure of nonsequence type, an automatic object, a structure component, a function name, a result name, an entry name, a named constant, or a subobject of any of the above.
6. An EQUIVALENCE statement, when used with a COMMON statement, must not extend a common block before the first storage unit of the common block. An EQUIVALENCE statement must not cause two COMMON blocks to be associated.
7. Data of type default character is equivalenced only with other data of type default character.
8. Conversion or mathematical equivalence is not performed for members of the set, nor is arrayness implied for a scalar. For example, if an integer variable I and a real variable R are equivalenced, and the value 1 is assigned to I, R has no predictable and portable value.

32 Expressions

An expression is the precise and complete description of a computation to be performed. Expressions are used in many contexts in Fortran, such as in assignment statements, procedure references, and output statements. An expression has a value and therefore a type, a kind, and a shape. Expressions are formed from operators and operands using intrinsic as well as defined-type operators.

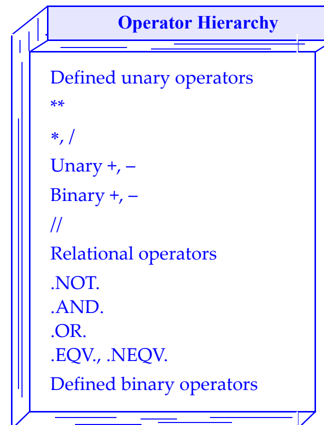
An operand may be a constant, variable, array element, structure component, substring, structure constructor, function reference, or an expression enclosed in parentheses, and may be a scalar or an array.

Examples:

`3.14159`	A constant is an expression.
`V`	A variable is an expression.
`2.0 * A - B ** 3.3`	An expression using *, –, and **
`SIN(A+B) - A * SQRT(B) / D`	An expression using intrinsic functions SQRT and SIN
`A .PLUS. B - C .TIMES. F`	An expression using operators –, .PLUS., and .TIMES.
`(/ 1, 2, 3 /) ** 2 + V`	An expression using array constructor
`FCN(X+Y) * SUM(AA, DIM=1)`	An expression using the intr. func. SUM and external func. FCN
`.NOT. L`	An expression using the unary logical .NOT. intrinsic operator
`(3.0, 5.0) - CONJG(CX)`	An expression using a complex constant
`RATIONAL( 1, 2*J) * &` `  RATIONAL( I, J )`	An expression using the structure constructor RATIONAL and the defined operator *

Related Topics:

Array Overview
Assignment
Defined Operators and Assignment
Expressions: Initialization
Expressions: Specification



To Read More About It:

ISO 1539 : 1997, Fortran Standard, 7.1-4
Fortran 95 Handbook, 7.1-4
Fortran 95 Using F, 1.6

Syntax:

An expression is one of:

unary-operator operand
operand [*binary-operator operand*]...

Things To Know:

1. The simplified syntax above is too restrictive in the sense that it prohibits all adjacent operators; some combinations are allowed as in A .EQV. .NOT. B and C .LT. –B.
2. When an intrinsic binary operator has an array operand, the other operand must be an array of the same shape or be a scalar. When the other operand is a scalar, it is treated as an array of the shape of the first array operand, all of whose elements are equal to the scalar. The result in either case is an array of the shape of the array operand(s). In case a unary intrinsic operator has an array operand, the result is an array of the shape of the operand. For such operations, the corresponding scalar intrinsic operation is performed element-by-element to each corresponding element, and the corresponding element of the result array is given this value. For array operands of intrinsic operators, restrictions on the shapes of the operands and the result of the operation are described in Array Overview.
3. For a defined or extension operator, the operation is performed by a function and returns a result whose type, kind, and shape is determined by the function. The rules for defining such a function and associating a specific operator with the function are described in Defined Operators and Assignment.
4. The operands of the intrinsic arithmetic operators must be of one of the arithmetic types but do not both have to be of the same type or kind. Loosely speaking, the result type and kind is that of the operand of the "bigger" type (the types can be ordered from least to most inclusive type).
5. The intrinsic arithmetic operators (+, –, *, /, **) have their usual mathematical meanings but the results are approximated because of the finite precision and exponent range of the representation of values. The power operator i**j when the second operand j is negative is defined as 1/(i**(–j)). It is invalid to raise a negative value of type integer or real to a real power. When operands of the power operator are both complex, the principal value is returned.
6. If an operand of an intrinsic operation has the POINTER attribute, the target associated with the pointer is used as the operand.

33 Expressions: Initialization

An initialization expression is a special or limited form of an expression used to specify such values as named constants, data initialized variables, kind values, and case values. An initialization expression is in essence an expression that can be evaluated at compile-time and is essentially an expression involving constant operands.

Examples:

```
! Initialization expressions with constant operands
PARAMETER ( N = 3**4 - 19 + 376/13 )  ! Note: restricted to
PARAMETER ( L = N - 3 + 7*N**3 )      !  integer powers

REAL, PARAMETER :: &
      R = 1.01, &
      SQR_EPS = EPSILON( R ) ** 2, &  ! EPSILON can be evaluated
                                      !  at compile time.
      SQRT_EPS = 1.0 / RADIX(1.0) &   ! RADIX and DIGITS can be
      ** ( ( 1 - DIGITS( W ) ) / 2 )  !  evaluated at compile
                                      !  time.

INTEGER, PARAMETER :: &               ! KIND, PRECISION and
      S = KIND(0.0), &                !  SELECTED_REAL_KIND can
      D = KIND(0.0D0), &              !  be evaluated at compile
      Q = SELECTED_REAL_KIND( &       !  time.
          2*PRECISION(1.0D0))
REAL(S)  X                            ! Kind value S is constant.
COMPLEX(D)  Y                         ! Kind value D is constant.
REAL(Q)  Z                            ! Kind value Q is constant.

CHARACTER( KIND('a') ) CH             ! Kind value is an expr.
                                      !  with constant operands.
REAL  A(7,6,5,4,3,2,1)
INTEGER, PARAMETER :: EXT_A = SIZE(A) ! SIZE can be evaluated at
                                      !  compile time.
INTEGER, DIMENSION(EXT_A) :: FLAT_A   ! Declare FLAT_A to have
                                      !   the same number of
                                      !   elements as A.
```

Related Topics:

Assignment
CASE Construct
Character Type and Constants
Complex Type and Constants
Data Initialization
Defined Type: Default Initialization
Expressions
Integer Type and Constants
Logical Type and Constants
PARAMETER Attribute and Statement
Real Type and Constants

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1, 5.2.9-10, 7.1.6.1, 8.1.3.1
Fortran 95 Handbook, 5.1, 5.5.1, 5.5.2, 7.2.9.2, 7.2.9.4-5, 8.4.1
Fortran 95 Using F, 1.2.9, 1.3.1

Syntax:

An initialization expression is an expression of any type limited by the constraints given below.

Things To Know:

1. An initialization expression may consist only of intrinsic operations; in addition, the second operand of ** must be of integer type.
2. The operands of an initialization expression are restricted to:
   - literal constants and named constants, and variables provided they are arguments of inquiry intrinsic functions which return results that can be determined at compile-time (for example, variables such as arguments of the KIND and SIZE intrinsic functions that are not dummy arguments).
   - array elements, array sections, substrings, and structure components provided the parent is a named constant, and the subscripts, section subscript bounds, and substring range endpoints are initialization expressions.
   - structure or array constructors, provided the components and elements are initialization expressions, or are implied-DOs whose DO loop parameters are initialization expressions or are expressions whose primaries are initialization expressions or DO variables of the same or an outer implied DO.
   - elemental intrinsic functions provided the arguments are initialization expressions of type integer or character, and the functions return integer or character results.
   - inquiry functions provided the arguments are initialization expressions or are variables whose attributes can be determined at compile-time (for example, this includes intrinsic functions such as KIND, SIZE, UBOUND, and PRECISION, but excludes the intrinsic function PRESENT).
   - the following transformational functions: NULL, REPEAT, RESHAPE, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER, and TRIM.
   - initialization expressions enclosed in parentheses.
3. When an initialization expression is used as a case value (in a CASE construct), it must be scalar and of type character, integer, or logical; when used as a kind value, it must be scalar and of type integer.
4. An initialization expression may be used as an expression in default initialization or a user-defined type, and as a component of a structure constructor in a DATA statement, a kind value as the argument of the conversion intrinsic functions such as REAL, or a subscript or substring range for a variable in an EQUIVALENCE statement.

34 Expressions: Specification

A specification expression is a special or limited form of an expression that can be evaluated immediately upon entry to a procedure (either external, internal, or module procedure). It must be scalar and of type integer and may involve references to certain user-defined functions. It is used in type declarations and other specification statements to declare bounds of arrays and lengths of character strings.

Examples:

```
INTEGER, PARAMETER :: NSIZE = 10, Q = 2
! The shape of A is determined by two specification expressions
!   with literal and named constants as operands.
INTEGER, DIMENSION(NSIZE + 3, NSIZE - 2 ** Q) :: A

! Assume N is a scalar integer dummy argument and R is a dummy
!   argument of rank 2.
INTEGER, DIMENSION (N, N) :: P       ! N is a spec. expression.
CHARACTER(N*N) CH( N*2, N-3 )        ! Three more specification
REAL R(12, 19)                       !   expressions
COMPLEX CQ
DIMENSION CQ( UBOUND(R,1)*3 - N**2 ) ! The size of CQ is a
                                     !   spec. expression.
COMMON /BLOCK/ M
LOGICAL, DIMENSION (M, M ) :: LQ     ! The extents of LQ are
                                     !   specification expressions.

! The length parameter is a specification expression.
CHARACTER( LEN = LEN(CH) + M - N + M/N ) CHQ( SIZE(CQ) )
! BINARY is a user-defined pure nonrecursive external function.
REAL TABLE(BINARY(N))                ! BINARY(N) is a specification
expression
                                     !   returning the value
                                     !   CEILING(LOG(REAL(N))/LOG(2.0))
```


Array Overview
Character Type and Constants
Complex Type and Constants
Expressions
Integer Type and Constants
Logical Type and Constants
Real Type and Constants

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.1.5, 5.1.2.4, 7.1.6.2
Fortran 95 Handbook, 5.1.6, 5.3.1, 7.2.9.3-5
Fortran 95 Using F, 3.9

Syntax:

A specification expression is a scalar integer expression constructed from intrinsic operations and operands limited as described below.

Things To Know:

1. An operand in a specification expression is one of the following:
   - a literal or named constant or part of a named constant (such as an array element, character substring, or a structure component),
   - a variable that is: a dummy argument; in a common block; accessible from a module or from a host program; or a subobject of any of these variables,
   - an array or structure constructor where the elements or components are expressions with operands satisfying the same restrictions, or are expressions whose operands may in addition be the DO-variables of the array constructor,
   - an elemental intrinsic function whose type is integer or character, and whose arguments are expressions of type integer or character with operands satisfying the same restrictions,
   - one of the transformational intrinsic functions NULL, REPEAT, RESHAPE, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER, or TRIM. The arguments of all but NULL must be expressions of type integer or character with operands satisfying the same restrictions, or
   - an inquiry intrinsic function, except ASSOCIATED, ALLOCATED, or PRESENT, whose arguments are expressions with operands satisfying the same restrictions or are variables whose bounds or type parameters inquired about are not assumed, are not defined by an ALLOCATE statement, or are not defined by a pointer assignment, or
   - a specification function. A specification function is a nonrecursive pure function that is an external or accessible module function. Its arguments and returned results may be of any type and kind,

   where in all cases above, any subscript or subrange is an expression satisfying the same restrictions.

35 EXTERNAL Attribute and Statement

An EXTERNAL attribute or statement specifies that a name may be used as an actual argument in procedure calls and references. The name is either an external procedure, a dummy procedure, or a block data program unit. If such a name has the same name as an intrinsic procedure, the name must be declared to be EXTERNAL. The intrinsic procedure will no longer be available to the program unit.

Examples:

```
SUBROUTINE SUB (FOURIER) ! FOURIER is a dummy procedure.
   REAL FOURIER          ! The actual argument corresponding
   EXTERNAL FOURIER      !   to FOURIER could be an external,
   . . .                 !   intrinsic, or module procedure.

REAL, EXTERNAL :: SIN, COS, TAN
! SIN, COS and TAN are no longer intrinsic procedures.
! Functions with these names must be defined in the program.

COMPLEX, EXTERNAL :: CX, CY

SUBROUTINE GRATX (X, Y)
EXTERNAL INIT_BLOCK_A    ! Specify INIT_BLOCK_A as the block
                         !   data subprogram that initializes
                         !   common block A.
COMMON/A/ TEMP, PRESSURE ! Common block available in GRATX
   . . .
END SUBROUTINE GRATX

BLOCK DATA INIT_BLOCK_A
COMMON/A/ TEMP, PRESSURE ! INIT_BLOCK_A initializes the
                         !   objects in common block  A.
DATA TEMP, PRESSURE/ 98.6, 15.5 /
END BLOCK DATA INIT_BLOCK_A
```

Tip: An interface block also specifies a name to be external in some cases. A name can appear in an EXTERNAL statement, be given the EXTERNAL attribute in a type statement, or appear in an interface block. However, it is recommended that the interface block be used instead of an EXTERNAL attribute or statement, except in the case of a block data subprogram where the interface block is not relevant.

Related Topics:

Interfaces and Interface Blocks
INTRINSIC Attribute and Statement
Module Procedures

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.10, 12.3.2.2
Fortran 95 Handbook, 5.7.1, 12.6.4

35 EXTERNAL Attribute and Statement

Syntax:

A type declaration statement with the EXTERNAL attribute is:

type , EXTERNAL [, *attribute-list*] :: *function-name-list*

An EXTERNAL statement is:

EXTERNAL *external-name-list*

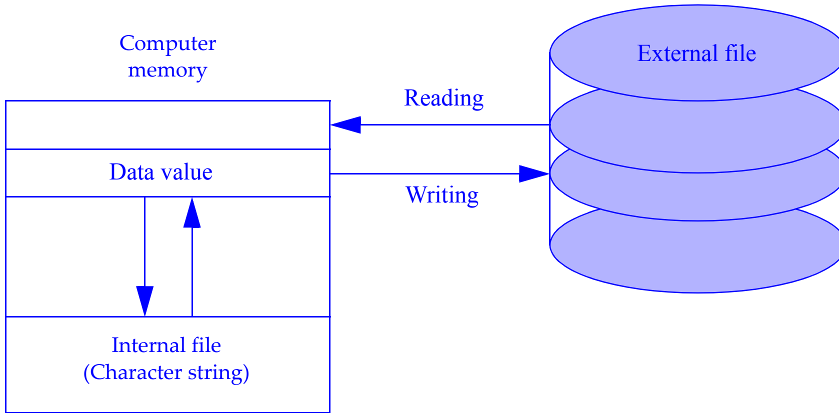
Things To Know:

1. The external name that appears in the type statement with an EXTERNAL attribute must be the name of an external function or a dummy argument that is a function.
2. The external name in an EXTERNAL statement is the name of an external procedure, a dummy procedure, or a BLOCK DATA subprogram.
3. A name can appear only once in an EXTERNAL statement, in a declaration statement with an EXTERNAL attribute, or in an interface body, and never in more than one of these.
4. If the name is a dummy argument, an EXTERNAL statement declares it to be a dummy procedure.
5. If a name of an intrinsic procedure is the name wanted for an external procedure, the EXTERNAL attribute must be declared in every program unit that references it. The named intrinsic procedure is no longer available in such program units.
6. The INTRINSIC and EXTERNAL attributes are mutually exclusive.
7. Note that the EXTERNAL statement does not have optional colons.

Attribute compatibility	Initialization	ALLOCATABLE	DIMENSION	INTENT	INTRINSIC	OPTIONAL	PARAMETER	POINTER	PRIVATE	PUBLIC	SAVE	TARGET
EXTERNAL						◆			◆	◆		

36 Files and Records

A **file** is a collection of **records**. In Fortran a record can be a printed line, a line on a terminal, or a logical record on some storage device such as a tape or a disk. A file may be either an **external file** or an **internal file** as shown in the diagram below.



A file may not **exist** for a program, which means that the program does not have access to the file.

All of the records in a file, except possibly the last one, are **data records**. The data records are all either **formatted** (a collection of characters) or **unformatted** (a collection of machine representable values).

The last record may be an **end-of-file record**, which is a processor-dependent marker for the end of the file. An end-of-file record is written explicitly by the ENDFILE statement or implicitly when the last data transfer is an output statement and a BACKSPACE, REWIND, CLOSE, or OPEN statement is executed or the program terminates without an error condition. An **end-of-file condition** occurs when a READ statement attempts to read past the last data record of a file.

The READ, WRITE, and PRINT statements transfer data to or from a file. The file may be an external file such as a disk, tape, or terminal, or the file may be an internal file, which is a character string in memory.

Related Topics:

CLOSE Statement
File Positioning Statements
INQUIRE Statement
OPEN Statement
READ/WRITE General Form

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.1-2, C.6.1.1-4
Fortran 95 Handbook, 9.1
Fortran 95 Using F, 9.1, 9.2

Things To Know:

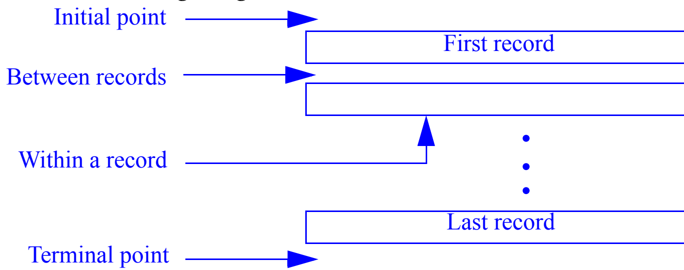
There are two ways to access the records in a file—sequential and direct. Using **sequential access,** the next record read or written is the one immediately following the current record; the records are processed in the order of their appearance in the file. Using **direct access**, a record is identified by its record number and the records may be read or written in any order.

An external file must be **connected** to a **unit** (**opened**) before the program can transfer data to or from the file. For example,

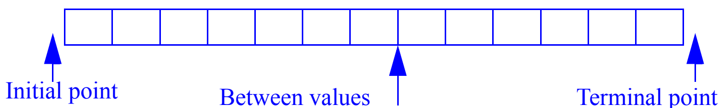
```
WRITE (UNIT = 7) X
```

writes the value of X to the file connected to unit number 7. For an internal file, a character variable is used as the unit, which is always connected.

File position is changed during program execution as records are read or written or by execution of an ENDFILE, REWIND, or BACKSPACE statement. A file may be positioned at its **initial point**, between records, within a record, or at its **terminal point** as shown in the following diagram.



A file positioned within a record may be positioned at the initial point of the record, between values in the record, or at the end of the record.



The position of a file may become indeterminate when an error condition occurs.

In most cases, one or more whole records are read or written by a READ, WRITE, or PRINT statement. However, part of a record may be read or written using **nonadvancing** formatted sequential access; in this case, the file is positioned after the last character read or written. The ADVANCE= specifier in the data transfer statement indicates whether or not the data transfer is advancing or nonadvancing. An **end-of-record condition** occurs when a nonadvancing READ statement attempts to read past the end of a record.

37 File Positioning Statements

BACKSPACE, REWIND, and ENDFILE are file positioning statements on external files connected for sequential access. Each file has a position at any given time during program execution. This position can change during execution of a data transfer statement as well as a file positioning statement. The position of a file for advancing data transfer is between records; the position of a file for nonadvancing data transfer is between values.

Examples:

```
IUNIT = 7 * K              ! The value of IUNIT must be a unit
BACKSPACE IUNIT            !    connected for sequential access.

REWIND 7                   ! 7 identifies an external file.

IU = 8                     ! IU identifies external unit 8.
ENDFILE IU

BACKSPACE (9,ERR = 29)     ! Branch to statement 29 if an
   . . .                   !   error occurs.

REWIND(IOSTAT=IR,UNIT=9)   ! IR has a positive value if an
   . . .                   !   error occurs.

ENDFILE (9, IOSTAT = IR)
```

Related Topics:

Files and Records
INQUIRE Statement
OPEN Statement
READ/WRITE General Form

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.5
Fortran 95 Handbook, 9.8
Fortran 95 Using F, 9.7

Specifier Notes

[UNIT=]	Required; only an external file
ERR=	Branch on error
IOSTAT=	Positive on an error, zero otherwise.

Syntax:

A file positioning statement is one of:

```
BACKSPACE scalar-integer-expression
BACKSPACE ( position-spec-list )

REWIND scalar-integer-expression
REWIND ( position-spec-list )

ENDFILE scalar-integer-expression
ENDFILE ( position-spec-list )
```

A position specifier is one of:

```
[ UNIT = ] scalar-integer-expression
ERR = label
IOSTAT = scalar-default-integer-variable
```

Things To Know:

1. If UNIT= is omitted, the unit specifier must come first. The list must contain exactly one external unit.
2. The file connected to the specified unit must be an external file.
3. The BACKSPACE statement positions the file before the previous record, or if the file is positioned within a current record, the statement positions the file at the beginning of the current record.
4. Backspacing over records written using list-directed or namelist formatting is prohibited.
5. The REWIND statement positions the file before the first record of the file.
6. The ENDFILE statement writes an end-of-file record and positions the file after this record.
7. Writing records after an end-of-file record is prohibited. Use a BACKSPACE or REWIND statement to reposition the file in such a case.
8. The label in the ERR= specifier must be in the same scoping unit as the file positioning statement.

38 FORALL Construct and Statement

The FORALL construct and statement provide a mechanism to specify an indexed parallel assignment of values to an array. They permit a straightforward translation of such formulas as

$$A_{ij} = i + j, \text{ for } i = 1 \text{ to } m, j = 1 \text{ to } n$$

The FORALL construct controls scalar or array assignments (including pointer assignments), masked array assignments (WHERE), and other nested FORALL constructs and statements within its body. The FORALL statement controls only one assignment statement—scalar, array, or pointer. The control in both cases is specified using index sets and scalar mask expressions.

Examples:

```
! Compute a circular matrix in a data-parallel statement
N = 3
FORALL ( I = 1:N, J = 1:N )  A(I,J) = MOD(I+J-2,N) + 1
!                    (  1  2  3  )
! A has the value:  (  2  3  1  )
!                    (  3  1  2  )
! The construct behaves as if expressions on the righthand
! side of the assignment are evaluated before any assignment
! takes place.

N = 10; EVEN(0:10) = (/ (I, I = 0, N) /)
FORALL ( I = 2:N:2 )
  EVEN(I) = EVEN(I-2) + EVEN(I-1)
END FORALL
! EVEN will have the value: ( 0 1 1 3 5 5 9 7 13 9 17 )

! Avoid a computation with a division by zero.
FORALL (I = N:1:-1, J = -N:N, DEM(I,J) /= 0 )
  C(I,J) = (A2(I,J) - 3*B(I,J)) / DEM(I,J)
END FORALL

! Produce a ragged array in parallel using a user-defined type
! with a array pointer component.
FORALL ( I = 1:N ) &
    A1(I) % ROW => NON_ZEROS_BY_ROWS( START(I) : START(I+1)-1 )
```

Related Topics:

Array: Data-Parallel Operations
Assignment
Expressions
Pure Procedures
WHERE Construct and Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 7.5.4, C.4.5-6
Fortran 95 Handbook, 7.5.5
Fortran 95 Using F, 4.1.9

FORALL Construct and Statement

Syntax:

The FORALL construct is:

[*constuct-name*:] FORALL (*index-triplet-list* [, *scalar-logical-expression*])
[*forall-body-construct-or-statement*] ...
END FORALL [*construct-name*]

A FORALL body statement or construct is one of:

forall-assignment-statement *where-construct-or-statement*
forall-statement-or-construct

A FORALL statement is:

FORALL (*index-triplet-list* [, *scalar-logical-expression*]) *forall-assignment-statement*

A forall assignment statement is one of:

assignment-statement *pointer-assignment-statement*

Things To Know:

1. Construct names may be used to identify a FORALL construct.
2. Any procedure referenced in the scalar logical expression of a FORALL, including defined or assignment, must be a pure procedure.
3. No scalar integer expression in the index triplet can reference the integer scalar name used as the index name in the same index triplet list.
4. For each assignment statement or pointer assignment statement within the FORALL construct or statement, the variable being defined or associated must use each index name in the index triplet lists of the FORALL bodies in which it appears.
5. First, the set of values for the index variables are determined; the set is further restricted to those values where the logical expression (if any) is true; then the statements are executed. In the case of the FORALL construct, the execution behaves as if the statements of the construct body are executed in order for the set of values selected by the index triplet list and scalar logical expression, where for each assignment statement its right hand side is fully evaluated over its set of index values before any assignments are performed and then the assignments over its index set are performed in any order.
6. Many-to-one assignments are prohibited.
7. Within a FORALL construct or statement, it is illegal to define the index variable. Index names in nested FORALL constructs or statements must not be the same names as index names in outer FORALL constructs.
8. The scope of the index names in the index triplet lists is the FORALL.

39 Format Specifications

A format specification contains explicit format information that indicates how data is converted to or from the internal representation from or to characters. The internal representation of values is usually binary. These conversions are done with edit descriptors. Format specifications are useful for producing readable program output under control of the programmer, and for controlling how input data is stored in the machine.

Examples:

```
CHARACTER (11) FMT;  REAL X(10)
FMT = '(A, 10F8.2)'
READ (5,"(10F8.2)")  X             ! Format specification in READ statement
PRINT FMT, 'This format specification is a character variable',X
   . . .
WRITE (6,100) "VALUES OF X =", X
100 FORMAT (A, 10F8.2)             ! Specification in FORMAT statement

PRINT "(A22)", "MARY HAD A LITTLE LAMB"
WRITE (6, "(A,3E16.1)" ) "IT WON""T CONVERGE", EPS

CHARACTER * 7, DIMENSION(3) :: FM
   . . .
FM(1) = "(F10.2,"; FM(2) = "ES15.3,"; FM(3) = "5X,L5)"
PRINT FM, XX, XSDCI, LOGG          ! The format is a character array.

PRINT 200, (Y(I), I=1,100)
200 FORMAT (2(3EN15.4,3X)/)

! Variable format for reals--depends on the processor precision
CHARACTER(8) :: VAR_FMT = '(Exx.yy)'          ! Format template
WRITE( VAR_FMT(3:4), '(I2)' ) PRECISION(X) + 9    ! Field width
WRITE( VAR_FMT(6:7), '(I2)' ) PRECISION(X) + 1    ! # of digits
WRITE( 6, VAR_FMT ) X

REAL Z(100)
READ  *, Z                 ! The asterisk specifies default format
   . . .                   !   specifications provided by
                           !   list-directed input formatting.
```

Related Topics:

Edit Descriptors: Control
Edit Descriptors: Data and Character String
READ/WRITE General Form

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 10
Fortran 95 Handbook, 10.1-3
Fortran 95 Using F, 1.7, 9.8

Syntax:

A format specification is:

([*format-item-list*])

A FORMAT statement that includes a format specification is:

FORMAT ([*format-item-list*])

A format item is one of:

[*r*] *data-edit-descriptor*
control-edit-descriptor
character-string-edit-descriptor
[*r*] (*format-item-list*)

Things To Know:

1. A format in a READ, WRITE, or PRINT statement is a format specification represented as one of the following:
   - a character string expression
   - a label indicating a FORMAT statement
   - an asterisk indicating default formatting provided by list-directed READ/WRITE/PRINT statements
2. The format specification may appear in a FORMAT statement or as a value of a character expression in a data transfer statement.
3. FORMAT statements may be used repeatedly and may appear anywhere in the program unit containing the data transfer statement. FORMAT statements should be labeled if they are to be useful.
4. All items in a character expression format specification must be defined. Parentheses are included in the expression, and the first nonblank character must be a left parenthesis. The matching right parenthesis must be in the expression. A format item list appears between the parentheses. Any characters appearing after the matching right parenthesis are ignored.
5. If the character expression is an array element, the entire specification must be within that element. If the character expression is an array, the format specification is the concatenation of the array elements in array element order.
6. Variable format specifications can be created in Fortran in an indirect way. Using internal input/output and a character string as a format specifier, the variable parts can be created and used as illustrated in the next to last example on the previous page. There, a format specification is created for real entities whose width and number of digits printed depends on the processor-dependent number of decimal digits in the fractional part of X.

40 Functions

A function definition is an external, module, or internal subprogram or a statement function statement. A function is used to compute a value to be used in an expression. A function reference appears as a primary in an expression. The reference is the name of the function along with its arguments, if any, enclosed in parentheses, or is in the form of a user-defined or extended operation. The function result is returned to the expression.

Examples:

```
PROGRAM P
   USE MY_OPERATORS      ! Get the unary operator .MYNEWOPERATOR.
   REAL TABLE( BINARY(967.0) )
   INTERFACE
      PURE INTEGER FUNCTION BINARY(X)
         REAL, INTENT(IN) :: X
      END FUNCTION BINARY
   END INTERFACE
   WIND_FUNC(X,Y) = X + C*Y
   PRINT *, F(1.1), .MYNEWOPERATOR. 4.4, WIND_FUNC (2.2, 3.3)
      . . .
   CONTAINS
      FUNCTION F(X)
         F = 2*X + 3
      END FUNCTION F
END PROGRAM P

PURE INTEGER FUNCTION BINARY(X)
   REAL, INTENT(IN) :: X
   INTRINSIC CEILING, LOG
   ! Compute log2(x)
   BINARY = CEILING( LOG(X)/LOG(2.0) )
END FUNCTION BINARY
```

Tip: The statement function was the only mechanism in Fortran 77 to define a function internal to another program unit. It is limited to a single statement that returns a scalar result. With internal and module procedures in Fortran 95, the statement function is redundant and its use is not recommended.

Related Topics:

Argument Association
Argument Keywords
Defined Operators and Assignment
INTENT Attribute and Statement
Interfaces and Interface Blocks
Internal Procedures
Module Procedures
OPTIONAL Attribute and Statement
Pure Procedures
Recursion
Subroutines

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 12.4, 12.5.2.2, 12.5.4, C.4.2
Fortran 95 Handbook, 12.1.1.10, 12.3
Fortran 95 Using F, 3.6

Syntax:

A function subprogram is:

```
[ prefix ] [ type-spec ] FUNCTION  function-name &
      ( [ dummy-argument-name-list ] ) [ RESULT ( result-name ) ]
   [ specification-part ]
   [ execution-part ]
   [ internal-subprogram-part ]
END [ FUNCTION [ function-name ] ]
```

A prefix is one of:

```
ELEMENTAL
PURE
RECURSIVE
```

A function reference is one of:

```
function-name ( [ function-actual-argument-list ] )
[ operand ] defined-operator operand
```

A function actual argument is one of:

```
expression
procedure-name
```

A statement function is:

```
function-name ( [ dummy-argument-name-list ] ) = scalar-expression
```

Things To Know:

1. The type of the function result may be declared implicitly, in the header statement, or in a type statement after the header statement.
2. The interface of an internal function is explicit in its host. The interface of a module function is explicit in all program units using the module.
3. The interface of an external function is implicit, but may be made explicit by the use of an interface block. The interface of a statement function is always implicit. A recursive function that calls itself directly must have a result clause. Other functions may have a result clause.
4. If there is no result clause, the function name becomes the result variable. If there is a result clause, the function name must not be used as the result variable. All attributes, such as DIMENSION or POINTER, must be declared for the result variable.
5. A function reference is a primary in an expression or a defined operator with one or two operands in an expression (see Defined Operators and Assignment). After the invocation of the function, the result is used in the expression in place of the function reference.
6. The END statement of an internal or module function must contain the word FUNCTION.

41 Generic Procedures and Operators

A procedure name is generic if it can be referenced with more than one actual argument type/kind/rank pattern; an operator is generic if its operand(s) can have more than one type/kind/rank pattern. Most intrinsic procedures are generic, and in addition user-defined procedures may be made generic. The interface block is used to specify generic names and operators for user-defined procedures. As with generic intrinsic procedures and operations, it is the type, kind, and rank pattern of the actual argument list in an expression or a reference to a generic procedure that determines which underlying specific user-defined procedure is called.

Examples:

```
INTERFACE SQRT                         ! Extending the generic
   MODULE PROCEDURE SQRT_OF_INTEGER !   properties of SQRT
END INTERFACE

INTERFACE OPERATOR ( + ) ! Extending the generic properties of +
   MODULE PROCEDURE INTEGER_PLUS_RATIONAL, RATIONAL_ADD
END INTERFACE

INTERFACE GENERIC_THING               ! GENERIC_THING defines a new
   INTEGER FUNCTION FIRST_THING(K) !   generic name and three
   END FUNCTION FIRST_THING         !   associated specific
   REAL FUNCTION SECOND_THING(X)    !   procedures with different
   END FUNCTION SECOND_THING        !   argument types.
   MODULE PROCEDURE THIRD_THING
END INTERFACE
```

Example expressions with generic operations and function references.

`SQRT(3.2)`	intrinsic	`K+2`	intrinsic
`SQRT(3.2D0)`	intrinsic	`K+X`	intrinsic
`SQRT(2)`	defined	`K+RATIONAL(N,M)`	defined

`GENERIC_THING(X)` generic reference to function SECOND_THING

Tip: Elemental procedures provide another way to create generic procedures over rank. For elemental procedures, only the type and kind of the actual arguments must match the dummy arguments; the different ranks are handled as if the scalar code were executed element-by-element.

Related Topics:

Defined Operators and Assignment
Elemental Procedures
Interfaces and Interface Blocks
Intrinsic Function Overview
Module Procedures
Scope, Association, and Definition Overview

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 12.3.2, 14.1.2.3-4
Fortran 95 Handbook, 12.7.8, 12.8.3-5, 13.1
Fortran 95 Using F, 7.1

Syntax:

An interface block for user-defined generic names and operators is:

```
INTERFACE [ generic-spec ]
   [ interface-body ]...
   [ MODULE PROCEDURE module-procedure-name-list ]...
END INTERFACE [ generic-spec ]
```

A generic specification is one of:

```
generic-name
OPERATOR ( intrinsic-operator-symbol )
OPERATOR ( . user-defined-operator-name . )
ASSIGNMENT ( = )
```

A user-defined operator name is:

```
letter [ letter ]...
```

Things To Know:

1. Any generic definition is allowed, as long as references to it involve a unique actual argument or operand type/kind/rank pattern or only type/kind pattern if the procedure is elemental. This is the only rule that must be followed to create generic definitions; it applies to generic procedure names, user-defined operators, and user-defined assignment.
2. For generic procedure references, the unique type/kind/rank (or just type/kind if the procedure is elemental) resolution principle applies to both positional and keyword actual argument lists. For example, two functions with dummy arguments named X and K, F1(X,K) and F2(K,X), with real X and integer K in each case, cannot both be associated with generic name F. The reason is that, even though positional actual arguments would be resolvable, reference F(X=.2, K=3) is ambiguous and not resolvable between F1 and F2.
3. The meaning of an intrinsic operation must not be changed by an interface block, but the meaning of an intrinsic procedure reference or assignment using defined types may be replaced by a user-defined procedure.
4. Note that only module procedures and external (and dummy) procedures can be given generic properties; internal procedures and statement functions cannot be given generic properties.
5. Typically, generic definitions will be placed in a module that is used by other program units needing these definitions. There may be any number of generic names active in a given scoping unit; a module procedure may be included in any number of these, but an external (or dummy) procedure may be in at most one in a given scoping unit.

42 Going Against the Flow

With the control constructs (IF, CASE, and DO) control flows in at the top and out at the bottom unless one of the "against the flow" statements appears. To change the flow, GO TO and STOP statements can appear anywhere in a program; RETURN statements can appear only in a subprogram. Also, data transfer statements with ERR=, END=, and EOR= specifiers can change the flow.

Examples:

```
     SUBROUTINE CALC (Y, Z)         ! Subroutine CALC checks the
        . . .                       !   range of Y. If Y exceeds
        . . .                       !   the permitted range, it
     IF (Y > YMAX) GO TO 303        !   calls an error handler
        . . .                       !   and stops the program.
     RETURN                         !   It returns to the caller
303  CALL ERR  &                    !   of CALC if the calculation
        (3, "OUT OF RANGE" )        !   proceeds to normal
     STOP 303                       !   completion.
     END

     SUBROUTINE ERR (MTYPE, MSG)          ! Error handler
        CHARACTER (*) MSG
        GO TO (10, 20, 30) MTYPE          ! Select msg. type.
  10    WRITE (6, "('- WARNING -')" )
        GO TO 40
  20    WRITE (6,"('- NONFATAL -')")
        GO TO 40
  30    WRITE (6, "('- FATAL -')" )
  40    WRITE (6, "(/ A /)" ) MSG         ! Output message.
     END
```

Tip: None of the "against the flow" statements are essential. Although they may sometimes be convenient, their use should be minimized. By using the Fortran control constructs, it is possible to create programs that have no labels and no GO TO statements. If execution reaches the END statement in a subprogram, control will be returned to the caller. If control reaches the last statement of a program, the program will stop.

Related Topics:

CASE Construct
DO Construct
IF Construct and Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 8.2, 8.4, 12.5.2.6
Fortran 95 Handbook, 8.6, 12.6.1
Fortran 95 Using F, 2.4.10, 3.10

Syntax:

A GO TO statement is one of:

GO TO *label*
GO TO (*label-list*) [,] *scalar-integer-expression*

A STOP statement is one of:

STOP [*scalar-default-character-constant*]
STOP *digit* [*digit* [*digit* [*digit* [*digit*]]]]

A RETURN statement is:

RETURN [*scalar-integer-expression*]

Things To Know:

1. A label in a GO TO statement must be a label attached to a branch target statement in the same scoping unit as the GO TO statement.
2. The scalar integer expression in a GO TO statement chooses one of the labels in the list. The same label may appear more than once in the list. If the number of labels is n and the scalar integer expression has a value less than 1 or greater than n, execution continues as though the GO TO statement did not appear. Otherwise, control goes to the statement with the nth label.
3. The STOP statement may contain a constant of default character type or a 1-to-5 digit number, which is made available (printed or displayed) when the program stops. This is useful in determining which STOP statement caused the program to terminate.
4. An expression may appear in a RETURN statement only if alternate returns (1 to n asterisks) are specified as dummy arguments in the FUNCTION, SUBROUTINE, or ENTRY statement of the subprogram. An expression with a value i in the range $1 \leq n$ will return to the ith asterisk argument (specified as **label*) in the actual argument list. Alternate returns are obsolescent.

43 Host Association

Host association is the association of entities between a host and a program unit within a host. A module is a host to module procedures and defined-type definitions; a program unit is a host to internal procedures and defined-type definitions; an interface body is a host to a defined-type definition within it. The unit within the host has access to the data environment of the host.

Examples:

```
MODULE HOST
  TYPE ADDITIONS                    ! Defined type defined in
    . . .                           !   the module
  END TYPE ADDITIONS
  REAL  X, Y                        ! Variables declared in the
  TYPE( ADDITIONS )  JANUARY        !   module
    . . .
  CONTAINS                          ! PARTS and INVENTORY are
                                    !   subroutines whose names
                                    !   (and interfaces) are part
                                    !   of the environment of the
                                    !   module  HOST.
  SUBROUTINE  PARTS( X )            ! X is a local variable that
    . . .                           !   is a dummy argument.  The
                                    !   host variable X is
                                    !   inaccessible.
    CALL  INVENTORY( JANUARY )      ! INVENTORY and JANUARY are
        . . .                       !   accessible by host
  END SUBROUTINE  PARTS             !   association from HOST.

  SUBROUTINE  INVENTORY( MONTH )    ! MONTH is a local variable
    . . .                           !  that is a dummy argument.
    TYPE( ADDITIONS ) MONTH, Y      !   Y is a local variable,
      . . .                         !   local to INVENTORY but
                                    !   the type  ADDITIONS  is
                                    !   accessible by host
                                    !   association from  HOST.
                                    !   The host variable  Y  is
  END SUBROUTINE  INVENTORY         !   inaccessible.
END MODULE HOST
```

Related Topics:

Defined Type: Definition
Interfaces and Interface Blocks
Internal Procedures
Modules

To Read More About It:

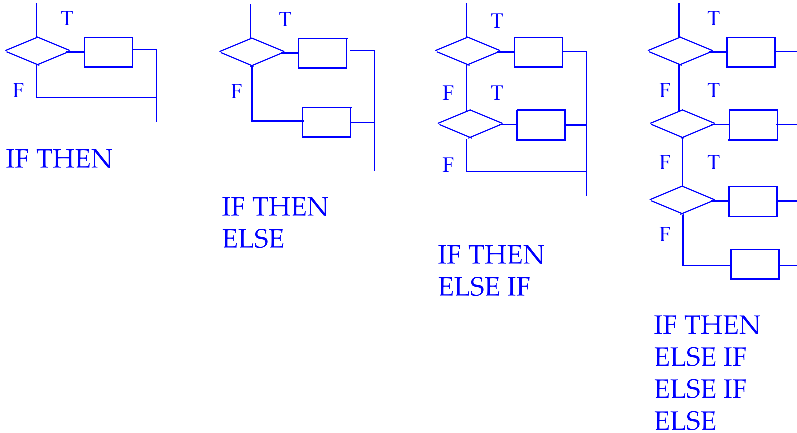
ISO 1539 : 1997, Fortran Standard, 14.6.1.3, C.10.1
Fortran 95 Handbook, 11.4, 12.1.1.6-7, 12.1.4, 14.3.1.3
Fortran 95 Using F, 3.1, 3.11

Things To Know:

1. Host association is the same as use association with no rename or ONLY clause, except for the implicit typing rules. Use association does not access the module's implicit type rules whereas host association uses the host's implicit type rules modified by any explicit IMPLICIT statements within the contained unit.
2. A name is local if it is declared explicitly in the contained unit, regardless of any declarations in the host unit. A dummy argument in a contained procedure is an explicit local declaration.
3. An entity appearing in a contained procedure and not declared there is nevertheless local if and only if that entity is neither explicitly or implicitly declared in the host unit.
4. If an entity is not local by items 2 and 3 above, it is accessible by host association within the contained units.
5. The default implicit type rules in a contained unit are those of the host unit. These are the default rules of the host plus those imposed by explicit IMPLICIT statements in the host.
6. The actual implicit type rules in a contained unit are the implicit type rules of the host plus those imposed by explicit IMPLICIT statements in the contained unit. Note that the actual implicit type rules of the contained unit apply to the dummy arguments of the contained unit. To avoid these rules, it is simpler to put an IMPLICIT NONE statement in the host.
7. An interface body (the part of an interface block beginning with a FUNCTION or SUBROUTINE statement) does not have a host and thus does not have access by host association to the environment of the program unit in which it is placed. However, use association may be used to make other environments accessible in an interface body, and the interface body is the host to any defined-type definition within the interface body.
8. An interface block may reference any module procedure accessible to the program unit containing the interface block.

44 IF Construct and Statement

The IF construct may be used to select for execution at most one code block from one or more in the construct. Selection is based on the value of one or more expressions. An IF construct may be named. It permits several paths of control flow including the following:



The IF statement may be used to selectively execute a single statement.

Examples:

```
! If then
IF (X < 0 .OR. Y < 0) THEN
   Z(I) = 1; CALL C(Z(I))
END IF

! If then; else
ADJUST: IF (INP > M) THEN
   CALL SCALE (INP)
ELSE    ADJUST
   CALL NORMAL (INP)
END IF ADJUST

! If then; else if
IF (VAL < MIN) THEN
   VAL = MIN
ELSE IF (VAL > MAX) THEN
   VAL = MAX
END IF
```

```
! If then; else if; else if; else
                              SIGN: &
IF (LIGHT == GREEN) THEN
   CALL PROCEED
ELSE IF (LIGHT == YELLOW) THEN SIGN
   CALL REDUCE_SPEED
ELSE IF (LIGHT == RED) THEN    SIGN
   CALL IDLE
ELSE
   CALL ERROR_AT (LIGHT)
END IF                         SIGN

! If statements
IF (X > MAX) CALL EXCEEDS_MAX
IF (MSG /= PASSWD) STOP
```

Related Topics:

CASE Construct   Expressions

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 8.1.2
Fortran 95 Handbook, 8.3
Fortran 95 Using F, 2.2

Syntax:

An IF construct is:

```
[ if-construct-name : ] IF ( scalar-logical-expression ) THEN
   block
[ ELSE IF ( scalar-logical-expression ) THEN [ if-construct-name ]
   block ]...
[ ELSE [ if-construct-name ]
   block ]
END IF [ if-construct-name ]
```

An IF statement is:

```
IF ( scalar-logical-expression ) action-statement
```

where an action statement is a single executable statement that is neither an IF statement nor an END statement.

Things To Know:

1. The block following the first expression that is true is the one executed. If no expression is true, the block following the ELSE statement is executed. If there is no ELSE statement and no expression is true, no code block is executed. A block may be empty.
2. Branching to any statement in an IF construct, other than the initial IF-THEN statement, from outside the construct is not permitted. Branching to an ELSE IF or ELSE statement is prohibited. Branching to an END IF statement from any block within the IF construct is allowed. Branching from within the block to other valid branch targets in the block is permitted.
3. Control constructs may be nested, in which case a program may be easier to read if the constructs are named. If a construct name appears on an IF-THEN statement, the same name must appear on the corresponding END IF statement and is optional on ELSE IF and ELSE statements of the construct.
4. A construct name must not be used as the name of any other entity in the program unit such as a variable, named constant, procedure, type, namelist group, or another construct.
5. If the expression in an IF statement is true, the action statement is executed; if the expression is false, the action statement is not executed. The expression may contain a function reference that produces side effects that modify variables in the action statement.

45 Implicit Typing

In Fortran, the type of a variable, function, or named constant can be declared explicitly in a type declaration statement. If it is not declared, it will be typed default real or default integer depending on the first letter of its name. Alternatively, implicit typing can be specified using an IMPLICIT statement. This statement also causes the type of a variable, function, or named constant to be inferred from the first letter of the name. Implicit typing may be disabled by an IMPLICIT NONE statement. If no IMPLICIT statements are present, the default implicit typing specifies that entities of undeclared type and with names beginning with letters I through N are of default integer type, and all other entities with undeclared types are of default real type.

Examples:

```
! Removes any implicit typing; all names must be declared.
IMPLICIT NONE

! Only names beginning with J-L are of integer type
!   (the default mapping).
IMPLICIT REAL (A-I, M-Z)

! Names beginning with Q-S are of defined type MINE
!   unless otherwise declared.
IMPLICIT TYPE (MINE) (Q-S)

! The default mapping is changed by IMPLICIT statements.
IMPLICIT INTEGER (P-Z)
IMPLICIT COMPLEX (KIND = 2) (E-H, O)

! Default      Complex     Default     Complex     Default
!   real       of kind 2   integer     of kind 2   integer

!   ABCD          EFGH     IJKLMN          O           P-Z
```

Tip: To reduce errors, use IMPLICIT NONE and declare all variables.

Related Topics:

Character Type and Constants
Complex Type and Constants
Defined Type: Definition
Integer Type and Constants
Logical Type and Constants
Real Type and Constants

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.3
Fortran 95 Handbook, 5.2, 14.3.1.3

Syntax:

An IMPLICIT statement is one of:

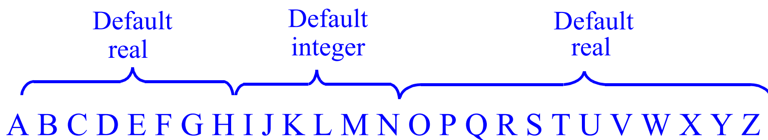
```
IMPLICIT type-spec ( letter-spec-list )
IMPLICIT NONE
```

A letter specification is:

letter [- *letter*]

Things To Know:

1. IMPLICIT NONE states that all names must be declared explicitly in a type declaration. IMPLICIT NONE must precede all specification statements, except USE statements.
2. No other IMPLICIT statements are allowed with IMPLICIT NONE.
3. PARAMETER statements may appear before, between, or after IMPLICIT statements with a letter specifier list. However, each IMPLICIT statement must confirm the type of any specified named constants which precede the IMPLICIT statement.
4. Letters separated by – in a letter specification must follow in alphabetical order. All letters between the letters separated by –, including the letter before and the letter after the –, are specified by this form.
5. A letter must not be specified more than once in an IMPLICIT statement in the same scoping unit.
6. A user-defined type may be the type specification in an IMPLICIT statement.
7. Implicit typing determines a type for names by inference from the first letter of the name. The default mapping for default real and default integer is as follows:



8. An IMPLICIT statement may be used to determine the mapping for certain letters. If a letter doesn't appear or is not within the range of letters in such a specification, the default mapping applies for the letter.

46 INCLUDE Line

An INCLUDE line is used to insert text into a program during compilation. The specified text is substituted for the INCLUDE line before compilation and is treated as if it were part of the original program source text. The location of the text to be included is specified by the value of the character constant in some processor-dependent manner.

A frequent convention is that the character literal constant is the name of a file containing the text to be included. Use of the INCLUDE line provides a convenient way to include source text that is the same in several program units. For example, interface blocks or common blocks may constitute a file that is referenced in the INCLUDE line.

Examples:

```
INCLUDE "MY_COMMON_BLOCKS"

INCLUDE '/usr/include/machine_parameters.h'

READ *, THETA
INCLUDE "FUNCTION_CALCULATION"  ! Program text may be included
   . . .                        !   within the executable part
                                !   of the program as well as
                                !   the specification part.
```

Tip: Modules provide access to data, types, and procedures that can be shared among procedures and thus provide a more effective way to accomplish most of what an INCLUDE line can do. However, as illustrated by the last INCLUDE line in the examples above, it is possible to use an INCLUDE line to include a portion of a subprogram; this is not possible with a module.

Related Topics:

Source Form

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 3.4
Fortran 95 Handbook, 3.4

46 INCLUDE Line

Syntax:

An INCLUDE line is:

`INCLUDE` *character-literal-constant*

Things To Know:

1. The character literal constant must not have a kind parameter that is a named constant.
2. The INCLUDE line is a directive to the compiler; it is not a Fortran statement. The INCLUDE line must appear on one line with no other text except possibly a trailing comment. There must be no statement label. This means, for example, that it is not possible to branch to it and it cannot be the action statement that is part of an IF statement. It is not permitted to put a second INCLUDE or another Fortran 90 statement on the same line using ";" as a separator. Continuing an INCLUDE line using an ampersand ("&") also is not permitted.
3. The INCLUDE line is placed where the included text is to appear in the program.
4. INCLUDE lines may be nested. That is, a second INCLUDE line may appear within the text to be included. The permitted level of nesting is not specified and is processor dependent. However, the text inclusion must not be recursive at any level; for example, included text A must not include text B, which includes text A.
5. The text of the file to be included must not begin or end with an incomplete Fortran statement.

47 INQUIRE Statement

An INQUIRE statement by file or by unit asks about various file properties or the connection status of a file or unit. There are three ways to inquire:

- an inquiry by unit
- an inquiry by file name
- an inquiry by length

An INQUIRE statement by length returns the length of an output list for unformatted direct access. An inquiry by file or unit number returns values that indicate properties of the connection between a unit and a file.

Examples:

```
! Inquiry about unit 7
INQUIRE (ERR = 99, ACCESS = CH34, FORM = CH35, UNIT = 7)

! Inquiry about file named TAPE32
INQUIRE (FILE = "TAPE32", OPENED = OP, ACTION = CH3)

! Inquiry about the length in unformatted direct access
INQUIRE (IOLENGTH = ICOUNT) X, Y, Z

! Inquiry parameters may be tested.
IUT = 12
INQUIRE (IUT, EXIST = LOGEX, SEQUENTIAL = CHARSEQ)
IF (LOGEX) GO TO 40
   . . .
40 IF (CHARSEQ == "NO") STOP
   . . .
```

Related Topics:

CLOSE Statement
Files and Records
OPEN Statement
READ/WRITE General Form

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.6, C.6.5
Fortran 95 Handbook, 9.7
Fortran 95 Using F, 9.2.3, 9.6

47 INQUIRE Statement

Syntax:

An INQUIRE statement is one of:

```
INQUIRE ([ UNIT = ] scalar-integer-expr , inquiry-specifier-list )
INQUIRE ( FILE = scalar-default-character-expr , inquiry-specifier-list )
INQUIRE ( IOLENGTH = scalar-default-integer-variable ) output-item-list
```

The inquiry specifiers appear in the following table.

		INQUIRE by file		INQUIRE by unit	
Specifier= variable		Unconnected	Connected	Connected	Unconnected
ACCESS=	C	UNDEFINED	SEQUENTIAL or DIRECT		UNDEFINED
ACTION=	C	UNDEFINED	READ, WRITE, or READWRITE		UNDEFINED
BLANK=	C	UNDEFINED	NULL, ZERO, or UNDEFINED		UNDEFINED
DELIM=	C	UNDEFINED	APOSTROPHE, QUOTE, NONE, or UNDEFINED		UNDEFINED
DIRECT=	C	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
ERR= label	-	Branch to label on an error			
EXIST=	L	.TRUE. if file exists, else .FALSE.		.TRUE. if unit exists, else .FALSE.	
FORM=	C	UNDEFINED	FORMATTED or UNFORMATTED		UNDEFINED
FORMATTED=	C	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
IOSTAT=	I	0 for no error, a positive integer for an error			
NAME=	C	Filename - Note 1		Note 2	Undefined
NAMED=	L	.TRUE.		Note 3	.FALSE.
NEXTREC=	I	Undefined	Next record #, if direct, else undefined		Undefined
NUMBER=	I	–1	Unit number		–1
OPENED=	L	.FALSE.	.TRUE.		.FALSE.
PAD=	C	YES	YES or NO		YES
POSITION=	C	UNDEFINED	REWIND, APPEND, ASIS, or UNDEFINED		UNDEFINED
READ=	C	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
READWRITE=	C	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
RECL=	I	Undefined	Rec length, if direct, else max rec length		Undefined
SEQUENTIAL=	C	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
UNFORMATTED=	C	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
WRITE=	C	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
IOLENGTH=	I	RECL= length of output-item-list for unformatted direct access			
C, I, L denotes variable type character, integer, or logical, respectively; the character values are without regard to case and trailing blanks are ignored					
Note 1: May not be the same as FILE= value			Note 2: Filename, if named, else undefined		
Note 3: .TRUE. if file is named, else .FALSE.					

48 Integer Type and Constants

The Fortran integer type is used to represent data that are whole numbers. More than one kind of integer is permitted; they are distinguished by kind numbers. At least one kind of integer, designated as the kind for the default integer type, is required.

Examples:

```
! ALONG and AFTER are declared of default integer type.
INTEGER ALONG, AFTER

! MAPPINGS uses kind parameter LONG where LONG is a
!   named integer constant defined previously.
INTEGER (KIND=LONG) :: MAPPINGS(100)

! NUMBER is a scalar variable with KIND parameter 2.
INTEGER (2) NUMBER

! BIG_INT has a specified minimum decimal range of 100.
!   If integers with this large range are not available, the
!   intrinsic function SELECTED_INT_KIND returns -1, which is
!   an invalid kind value that the compiler must diagnose.
INTEGER( SELECTED_INT_KIND(100) )  BIG_INT
```

Examples of integer constants are:

```
249            Default integer constant
-14            Default integer constant
23000_LESS     LESS is a kind parameter
O"15"          An octal integer constant;
                  only in a DATA statement
Z"13A4"        A hexadecimal integer constant;
                  only in a DATA statement
```

Related Topics:

Data Representation Models
Expressions
Implicit Typing
Kind Parameters

Related Intrinsics:

INT (A, KIND)
KIND (X)
RANGE (X)
SELECTED_INT_KIND (R)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.3.1.1, 5.1.1.1, 13.7.1
Fortran 95 Handbook, 4.3.1, 5.1.1, 13.2.2
Fortran 95 Using F, 1.2.1, 1.3.1, A.6.1

Syntax:

An integer type declaration statement is:

INTEGER [([KIND =] *kind-parameter*)] [, *attribute-list* ::] *entity-list*

A decimal constant is:

[*sign*] *digit-string* [_ *kind-parameter*]

A kind parameter is one of:

digit-string
scalar-integer-constant-name

A binary constant is one of:

B ' *digit* [*digit*]... ' B " *digit* [*digit*]... "

where digit is 0 or 1.

An octal constant is one of:

O ' *digit* [*digit*]... ' O " *digit* [*digit*]... "

where digit is 0, 1, 2, ..., or 7.

A hexadecimal constant is one of:

Z ' *digit* [*digit*]... ' Z " *digit* [*digit*]... "

where a digit is 0, 1, 2, ..., 9, A, B, ..., or F.

Things To Know:

1. Integer operators are: +, –, *, /, **, unary +, unary –. The relational operators <, <=, ==, /=, >, >=, .LT., .LE., .EQ., .NE., .GT., and .GE. may be used for comparisons; they yield default logical values. The relational operators <, <=, ==, /=, >, and >= are synonyms for .LT., .LE., .EQ., .NE., .GT., and .GE, respectively.
2. Decimal constants may be used in any numeric expression, in input records, and in DATA statements. Binary, octal, and hexadecimal constants are restricted to DATA statements to initialize integer variables.
3. There is always one integer kind available. There may be more.
4. The value of a kind parameter must indicate a representation method that exists on the processor.
5. The values of the kind parameters are not standard from machine to machine. The SELECTED_INT_KIND intrinsic function provides a portable way to deal with this problem and to specify an integer data type with adequate decimal exponent range.
6. A variable of default integer type occupies one numeric storage unit.

49 INTENT Attribute and Statement

The INTENT attribute declares whether a dummy argument is intended for transferring a value into or out of a procedure or both. The INTENT attribute helps detect errors in the use of arguments that are inconsistent with their intended use. Specifying intent makes the program more readable, and may assist compilers in generating more efficient code.

Examples:

```
SUBROUTINE ELECTRIC (X, Y, Z)  ! X, Y, and Z are dummy arguments.
   REAL, INTENT (IN) :: X, Y   ! X and Y are used only for input.
   COMPLEX, INTENT (INOUT), TARGET :: Z(1000)    ! Z is used for
      . . .                                     !   input and output.

SUBROUTINE PRESSURE (TRUE, TAPE, A, B)
   USE A_MODULE
   TYPE (ACE), INTENT (IN) :: A,B   ! A and B are only for input.
   INTENT (OUT) TRUE, TAPE          ! TRUE and TAPE are used
     . . .                          !   only for output.

SUBROUTINE LAB_TEN (DEGREES, X, Y, Z)
   COMPLEX, INTENT(INOUT) :: DEGREES
   REAL, INTENT(IN), OPTIONAL :: X, Y
   INTENT(IN) Z
      . . .

PROGRAM PXX
   CALL ELECTRIC (A+1, H*C, D)
      . . .
   CALL LAB_TEN (DG, E, F, G+1.0)   ! Valid -- last argument is
END PROGRAM PXX                     !   intent IN so the actual
                                    !   argument can be an expr.
```

Tip: Use of the INTENT attribute is good programming practice and helps to eliminate errors in the use of procedure arguments. It permits the compiler to detect certain incorrect references to a procedure, where, for example, an expression is used as an actual argument corresponding to an intent OUT dummy argument. All dummy arguments in a function should be intent IN to avoid side effects. Use of explicit interfaces with intents specified may make the routines more efficient and certainly makes references to the routines more reliable.

Related Topics:

Argument Association
Argument Keywords
Defined Type: Default Initialization

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.3, 5.2.1, 12.5.2.1
Fortran 95 Handbook, 5.6.2, 12.7.6
Fortran 95 Using F, 3.5.2

Syntax:

A type declaration statement with the INTENT attribute is:

type , `INTENT` (*intent-spec*) [, *attribute-list*] :: *dummy-argument-name-list*

An INTENT statement is:

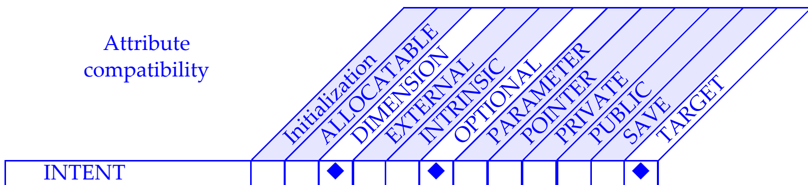
`INTENT` (*intent-spec*) [::] *dummy-argument-name-list*

An intent specification is one of:

`IN` `OUT` `INOUT`

Things To Know:

1. Intent specifications are used only for dummy arguments and appear only in the specification part of a subprogram or interface body. Only dummy arguments with intent OUT may be of a user-defined type that is default initialized.
2. If a dummy argument has intent IN, the procedure must not change it or cause it to become undefined. If the actual argument is defined, this value is passed in as the value of the dummy argument.
3. If a dummy argument has intent OUT, the corresponding actual argument must be definable; for example, it cannot be a constant. When execution of the procedure begins, the dummy argument is undefined unless it is of a type for which default initialization is specified. It is not necessary that the dummy argument be given a value by the procedure.
4. If a dummy argument has intent INOUT, the corresponding actual argument must be definable. If the actual argument is defined, this value is passed in as the value of the dummy argument. It is not necessary that the dummy argument be given a value by the procedure.
5. If there is no intent specified for an argument in a subprogram, the limitations imposed by the actual argument apply to the dummy argument. For example, if the actual argument is an expression that is not a variable, the dummy argument must not redefine its value.
6. The intent of a pointer dummy argument must not be specified.



50 Interfaces and Interface Blocks

A procedure interface consists of the name of the procedure, characteristics of the dummy arguments and the result if the procedure is a function. The interface is explicit in a scoping unit in which these properties are all known, and implicit otherwise. The interface of an external procedure by default is implicit. Explicit interfaces are required for certain Fortran features (see notes below). The interfaces of internal and module procedures are explicit in host and using scoping units, and external procedures may be given explicit interfaces. Interface blocks constitute the mechanism by which this is done, as well as provide additional related functionality.

Examples:

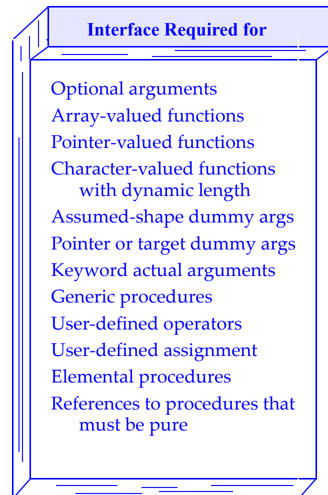
```
INTERFACE                          ! Make explicit the interfaces of
   REAL FUNCTION SPLINE(X,Y,Z)     !   external function SPLINE
   END FUNCTION SPLINE             !   and
   SUBROUTINE SP2(X,Z)             !   external subroutine SP2.
   END SUBROUTINE SP2
END INTERFACE

INTERFACE G_AVE                    ! Make the interface of
   FUNCTION R_AVE(X)               !   function R_AVE explicit, and
      USE AVE_STUFF, ONLY: N       !   give it the generic name G_AVE.
      REAL R_AVE, X(N)
   END FUNCTION R_AVE
END INTERFACE

INTERFACE OPERATOR ( + )               ! Make the interface of
   LOGICAL FUNCTION B_OR(P,Q)          !   external function B_OR explicit,
      LOGICAL, INTENT(IN), P, Q        !   and use it and
         END FUNCTION B_OR             !   module function C_OR
   MODULE PROCEDURE C_OR               !   to extend the "+" operator.
END INTERFACE
```

Related Topics:

Argument Keywords
Defined Operators and Assignment
Elemental Procedures
Generic Procedures and Operators
Pure Procedures
Scope, Association, and Definition Overview

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 12.3, C.8.3.5, C.9.3
Fortran 95 Handbook, 12.8
Fortran 95 Using F, 3.8.8



Syntax:

An interface block is:

```
INTERFACE [ generic-spec ]
   [ interface-body ]...
   [ MODULE PROCEDURE module-procedure-name-list ]...
END INTERFACE
```

An interface body is one of:

```
function-statement
   [ specification-part ]
END FUNCTION [ function-name ]
```

```
subroutine-statement
   [ specification-part ]
END SUBROUTINE [ subroutine-name ]
```

Things To Know:

1. The specification part of an interface body contains specifications pertaining to the dummy arguments of the procedure and, in the case of functions, the function result. These must be described completely, although they may rely on implicit typing and, except for the dummy argument names, must agree with the specifications in the procedure definition. Interface bodies apply only to external (and dummy) procedures.
2. The dummy argument names in an interface block may be different from those in the procedure definition; the names in the explicit interface are those that are used in references with keywords.
3. A procedure must not have more than one explicit interface specified in a given scoping unit. Note that the MODULE PROCEDURE statement is a "pointer" to an accessible explicit interface, not an explicit interface specification itself, whereas an interface body is an explicit interface specification. A procedure name in a MODULE PROCEDURE statement must be the name of a module procedure either in that module (if the host of the interface block is a module) or accessible to the host through use association
4. An interface block is not itself a scoping unit, but rather "walls off" from the surrounding scope any interface bodies it contains by blocking host association and inheritance of the implicit type mapping. Each interface body is a separate scoping unit. Unlike other nested scoping units, nothing "flows into" an interface block from its host. The implicit type mapping in an interface body is the default mapping (namely I-N default integer and A-H, O-Z default real) modified by any IMPLICIT statements in the interface body.
5. The MODULE PROCEDURE statement is allowed only for interface blocks that have a generic specifier.

51 Internal Procedures

An internal procedure is defined by a subroutine or function subprogram that is contained inside a main program or another procedure program. It provides a procedure mechanism that conveniently accesses another data environment (that of its host) and provides a multistatement alternative to the statement function. An internal procedure also provides a means for modular design and contributes to better engineered software.

Examples:

```
PRINT *, F(6.6)
CONTAINS
   FUNCTION F(X); F = X+1.1; END FUNCTION
END

PROGRAM PROG                        ! Host scoping unit
   USE GLOBAL, ONLY: P, Q, R        ! Access names P, Q, R
   IMPLICIT NONE
   TYPE T; . . .; END TYPE          ! Defines T as the name of a type
   TYPE(T) A, B, C                  ! Declares A, B, C (of type T)
   REAL X, Y, Z                     ! Declares X, Y, Z (of type real)
      . . .
CONTAINS
   REAL FUNCTION F_1 (X)        ! F_1 must be explicitly typed due to
      REAL X; . . .             !   IMPLICIT NONE in the host.
   END FUNCTION F_1             ! X is local to F_1;
                                !   not the X in the host.
   SUBROUTINE S_1 (A, J)
     IMPLICIT INTEGER (A-K)! Augments implicit mapping from host
      CI = 3
        . . .
      PRINT *, CI + J           ! Prints an integer
   END SUBROUTINE S_1

   FUNCTION F_2 (H, K)                  ! Typing required, as for F_1
      REAL F_2, H, K
      TYPE S; . . .; END TYPE S         ! New local type
      TYPE T; . . .; END TYPE T         ! Also new type
      TYPE(S) R; TYPE(T) Z; . . .       !   different from host T
      PRINT *, A, R, Y, Z               ! A and Y are accessed from
   END FUNCTION F_2                     !   the host while R and Z
END PROGRAM PROG                        !   are local objects.
```

Related Topics:

Functions
Host Association
Module Procedures
Subroutines
Scope, Association, and Definition Overview

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 2.2.3.3, 12.1.2.2, 12.5.2.7
Fortran 95 Handbook, 11.3, 12.1.1.6

Syntax:

An internal procedure part is:

```
CONTAINS
    internal-subprogram
    [ internal-subprogram ]...
```

An internal subprogram is one of:

```
function-statement
    [ specification-part ]
    [ execution-part ]
END FUNCTION [ function-name ]

subroutine-statement
    [ specification-part ]
    [ execution-part ]
END SUBROUTINE [ subroutine-name ]
```

1. Main programs, external procedures, and module procedures may each have an internal procedure part; however, an internal procedure definition cannot have an internal procedure part.
2. Internal procedures have two restrictions that external and module procedures do not have:
   - ENTRY statements are not allowed in internal procedures.
   - Internal procedure names cannot be actual arguments.
3. An internal procedure name is treated as a local name in its host. The name of an internal procedure must be different from the names of all other internal procedures in that host, other local names in the host, and names made accessible by a USE statement in the host.
4. The interface of an internal procedure is explicit in its host.
5. Each internal procedure comprises a scoping unit that is nested in a host scoping unit. The rules of host association apply to each internal procedure. Local specifications in the internal procedure override any definition inherited by host association.
6. Host association applies to entities defined or accessible in the host scoping unit.
7. The default type mapping in an internal procedure is the type mapping of the host; IMPLICIT statements in the internal procedure may modify part or all of this mapping.

52 INTRINSIC Attribute and Statement

An intrinsic procedure is one defined by the standard and supplied along with the Fortran language as part of the language itself. The INTRINSIC attribute or statement specifies that a name is a specific name or generic name of an intrinsic procedure. An intrinsic subroutine can be specified as intrinsic only in an INTRINSIC statement. In certain circumstances, a specific name of an intrinsic function, when used as an actual argument, must be specified with an INTRINSIC attribute or statement. It may also be used as documentation and to specify that a name is an intrinsic function.

Examples:

```
COMPLEX, INTRINSIC :: CCOS, CSIN ! CCOS and CSIN are specified
X = CCOS(A) + CSIN(B)            !   as intrinsic function
                                 !   names for documentation
                                 !   purposes only.

INTRINSIC TAN, ATAN              ! TAN and ATAN are specific
   . . .                         !   names of intrinsic
                                 !   functions used as actual
CALL TIME (TAN, ATAN)            !   arguments to the subroutine
   . . .                         !   TIME.

INTRINSIC RANDOM_NUMBER          ! Used for documentation only.
      . . .
   CALL RANDOM_NUMBER (R)
```

Tip: The INTRINSIC statement may be used for processor-supplied intrinsic routines not in the Fortran standard. These functions extend the list of standard intrinsic functions by adding new ones to the list of intrinsics. Note that such a program is not portable.

When used to designate nonstandard intrinsic procedures, the statement implies that the specified names have an explicit interface, and therefore their genericity, types, and dummy argument attributes are known to the compiler. In addition, a processor that doesn't support the specified procedures as intrinsic can immediately indicate that the names are not supported as intrinsic procedures.

Related Topics:

EXTERNAL Attribute and Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.11, 12.3.2.3
Fortran 95 Handbook, 5.7.2, 12.6.5
Fortran 95 Using F, 2.4.8

Syntax:

A type declaration statement with the INTRINSIC attribute is:

type , `INTRINSIC ::` *intrinsic-function-name-list*

An INTRINSIC statement is:

`INTRINSIC` *intrinsic-procedure-name-list*

Things To Know:

1. It is rather confusing as to when a name *must* be specified as an intrinsic procedure. The rule is that if an intrinsic procedure is used as an actual argument and no other appearance of the name in the same scoping unit indicates it is a procedure, the name must be declared with an intrinsic attribute. For example, if in a program unit the statement

   ```
   CALL MY_SUBROUTINE (SIN)
   ```

   appears and no other occurrence of SIN appears, the compiler assumes SIN is a variable and not the specific name of the intrinsic function SIN. Therefore, SIN must be declared intrinsic to ensure that the intrinsic function is passed.
2. The name must be that of an intrinsic procedure.
3. Note that the INTRINSIC statement does not have optional colons; this is also true for the EXTERNAL statement.
4. The EXTERNAL and INTRINSIC attributes are mutually exclusive. The INTRINSIC attribute may be declared only once for a name.
5. The INTRINSIC statement must be used to declare intrinsic subroutine names as intrinsic when used as arguments, because subroutine names cannot appear in type statements. Functions, however, may be declared in a type declaration statement or in an INTRINSIC statement.
6. When the INTRINSIC statement is used to identify intrinsic functions added by the processor, they can be immediately identified on another processor that does not support them.

53 Intrinsic Function Overview

Intrinsic functions are "packaged" computations provided with Fortran and accessed via function calls in expressions. Intrinsic functions provide a way to incorporate into Fortran the most common computations important to scientific and engineering applications. Fortran has 109 intrinsic functions, which may be classified as 19 conversion intrinsics, 17 array intrinsics, 28 inquiry and model intrinsics, and 45 computation intrinsics, of which 24 perform various numeric computations, 12 perform character computations, and 9 perform bit computations. In addition, there are 6 intrinsic subroutines. Intrinsic procedures are described in detail in Appendix A.

Examples:

```
COMPLEX        :: Z          ! These declarations apply
REAL           :: R, AR(:)   !   to the example intrinsic
REAL(DOUBLE)   :: A2D(:,:)   !   function references.
CHARACTER(80)  :: C1
CHARACTER(20)  :: C2
LOGICAL        :: L
```

Example references to intrinsic functions:

`COS ( Z )`	Generic call to COS
`COS ( X = R )`	Keyworded generic call to COS
`COS ( AR )`	Elemental generic call to COS
`INDEX ( C1, C2 )`	Optional argument BACK omitted
`INDEX ( C1, C2, L )`	Optional argument BACK supplied
`INDEX ( C1, C2, BACK=L)`	Optional argument keyworded
`SUM ( A2D )`	Summation transformational function
`CSHIFT ( AR, -2 )`	Circular shift transformational function
`RANGE ( R )`	Exponent range for reals

Examples of more complete statements:

```
OCCURS_ONCE = INDEX(C1,C2) .NE. INDEX(C1,C2,BACK=.TRUE.)
R = SUM(COS(AR)*CSHIFT(AR,-2)) ! Rank-one array expression
                               !    is the argument to SUM.
```

Related Topics:

Functions
INTRINSIC Attribute and Statement
Intrinsic Functions: Array
Intrinsic Functions: Computation
Intrinsic Functions: Conversion
Intrinsic Functions: Inquiry and Model
Intrinsic Subroutines

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.11., 12.1.2.1, 12.3.2.3, 13
Fortran 95 Handbook, 5.7.2, 12.3.3, 12.6.5, 13, A
Fortran 95 Using F, 1.5, A

Syntax:

An intrinsic function reference is:

function-name ([*actual-argument-list*])

An actual argument is:

[*dummy-argument-name* =] *expression*

Things To Know:

1. Intrinsic function references may use keywords, in which case the actual argument expression is preceded by the dummy argument name (argument keyword) and an "=" symbol. These argument keywords are shown in the description of each intrinsic function.
2. Some intrinsic function arguments are optional. These are underlined in the lists of intrinsic functions and subroutines in the next five topics.
3. All of the intrinsic functions are generic, except for LGE, LGT, LLE, and LLT. This means that each intrinsic function may be called with more than one argument type/kind/rank pattern. Generally, the kind and type of the result are the same as that of the "principal" argument. For example, the SIN function may be called with any kind of real argument or any kind of complex argument. Many of the intrinsic functions also have nongeneric (specific) names.
4. An intrinsic function is either elemental or transformational; most are elemental. An elemental function has all scalar dummy arguments and delivers a scalar result. It may be called with conformable array arguments, however, which results in a conformable array result. The effect is as if the scalar form of the function were called for each corresponding element of the actual argument arrays supplied—hence the term elemental. A transformational function is one in which at least one dummy argument is an array.
5. An intrinsic function is always "there", in every program unit—with two exceptions. It takes precedence over a user-defined function with an implicit interface except when the user-defined function is a statement function or when the name of the intrinsic function has been given the EXTERNAL attribute. On the other hand, a user-defined function with an explicit interface in the scoping unit takes precedence over an intrinsic function with the same name in that scoping unit, unless that name has been given the INTRINSIC attribute. Thus, for example, module and internal functions take precedence over intrinsic functions with the same name.

54 Intrinsic Functions: Array

The array intrinsic functions are part of the Fortran array processing facilities, supplementing the intrinsic array operations. The array functions may be classified into three categories: (1) array reduction functions, which all return values determined from a given array (or along a specified dimension of the array), (2) array construction functions, which construct new arrays from various pieces of other arrays, and (3) miscellaneous array functions, which either rearrange the elements of an array (CSHIFT, EOSHIFT, RESHAPE, TRANSPOSE) or return location information (MAXLOC, MINLOC).

Eleven of these 17 functions have an optional argument DIM, which if present specifies the dimension along which the operation takes place. (One function, SPREAD, has a nonoptional DIM argument.) Twelve of the functions have a logical array argument MASK, conformable with the argument ARRAY; in 6 of these 12 cases MASK is optional and if present restricts the operation to those elements of ARRAY for which MASK is .TRUE. All optional arguments are underlined in the accompanying list of array functions. All of these functions are transformational, except MERGE, which is elemental.

Examples:

Examples of references to array intrinsic functions:

`REAL X(100,100)`	The array used in the following examples
`SUM(X)`	Sum of all 10,000 elements of X
`SUM(X(:,K))`	Sum of all elements in Kth column of X
`SUM(X, DIM=2)`	Array of sums—one for each row of X
`COUNT(X>0)`	Number of positive values in X
`MAXVAL(X)`	Maximum value in X
`MAXLOC (X)`	(/ row, column /) of the max. value of X
`TRANSPOSE (X)`	Returns the transpose of X
`CSHIFT(X,2,DIM=1)`	Circularly shifts X "up" 2 elements
`MERGE(X,0.0,X.GE.0)`	Replace negative values in X with zero
`RESHAPE(X,(/1000,10/))`	Result is a 1000 by 10 array

Related Topics:

Array Overview
Intrinsic Function Overview

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 13.11.14-19, 13.14
Fortran 95 Handbook, 13.6, A
Fortran 95 Using F, A.8

Array reduction functions

`ALL ( MASK, DIM )`	.TRUE. if MASK is all .TRUE.
`ANY ( MASK, DIM )`	.TRUE. if any of MASK is .TRUE.
`COUNT ( MASK, DIM )`	Count of .TRUE. values in MASK
`MAXVAL ( ARRAY, DIM, MASK )`	Maximum value in ARRAY
`MINVAL ( ARRAY, DIM, MASK )`	Minimum value in ARRAY
`PRODUCT ( ARRAY, DIM, MASK )`	Product of ARRAY values
`SUM ( ARRAY, DIM, MASK )`	Sum of ARRAY values

Array construction functions

`MERGE ( TSOURCE, FSOURCE, MASK )`	Combines values from two sources into an array
`PACK ( ARRAY, MASK, VECTOR )`	Packs ARRAY values to vector[1] under MASK control
`SPREAD ( SOURCE, DIM, NCOPIES )`	Replicates an array by adding a dimension
`UNPACK ( VECTOR, MASK, FIELD )`	Unpacks VECTOR into array[2] conformable with MASK

Miscellaneous array functions

`CSHIFT ( ARRAY, SHIFT, DIM )`	Circular shift ARRAY elements[3]
`EOSHIFT ( ARRAY, SHIFT, BOUNDARY, DIM )`	End-off shift ARRAY elements[3]
`MAXLOC ( ARRAY, DIM, MASK )`	Location of maximum value[4]
`MINLOC ( ARRAY, DIM, MASK )`	Location of minimum value[4]
`RESHAPE ( SOURCE, SHAPE, PAD, ORDER )`	Reshapes SOURCE into an array the shape of SHAPE[5]
`TRANSPOSE ( MATRIX )`	The transpose of MATRIX

Notes:
1. If VECTOR is present in PACK, the packed result has the size of VECTOR; otherwise the result size is COUNT(MASK).
2. FIELD specifies the values to be placed in those unpacked locations that correspond to .FALSE. values in MASK.
3. CSHIFT and EOSHIFT are straightforward if ARRAY is of rank one; otherwise, consult the full descriptions of these functions.
4. If DIM is not present, MAXLOC and MINLOC return a one-dimensional array whose size is the rank of ARRAY; otherwise, consult the full descriptions of these functions.
5. In RESHAPE, SHAPE is a one-dimensional array whose elements are the extents of the respective dimensions of the result; if ORDER is present, it must be conformable with SHAPE and be a permutation of 1 to the size of ORDER.

55 Intrinsic Functions: Computation

Numeric functions with real arguments

`ACOS ( X )`	Arc (inverse) cosine
`ASIN ( X )`	Arc sine
`ATAN ( X )`	Arc tangent
`ATAN2 ( Y, X )`	Angle, given the coordinates
`COSH ( X )`	Hyperbolic cosine
`DPROD ( X, Y )`	Double precision product[1]
`SINH ( X )`	Hyperbolic sine
`TAN ( X )`	Tangent
`TANH ( X )`	Hyperbolic tangent

Numeric functions with real or complex arguments

`ABS ( A )`	Absolute value
`COS ( X )`	Cosine
`EXP ( X )`	Exponentiation, e^x
`LOG ( X )`	Natural logarithm
`LOG10 ( X )`	Base-10 logarithm
`SIN ( X )`	Sine
`SQRT ( X )`	Square root

Numeric functions with real or integer arguments

`DIM ( X, Y )`	MAX (X–Y, 0)
`MAX ( A1, A2, A3, ... )`	Maximum of a set of two or more values
`MIN ( A1, A2, A3, ... )`	Minimum of a set of two or more values
`MOD ( A, P )`	Remainder of A/P; result has sign of A
`MODULO ( A, P )`	Remainder of A/P; result has sign of P
`SIGN ( A, B )`	Apply sign of B to A

Numeric functions with real, integer, or complex arguments

`DOT_PRODUCT ( VECTOR_A, VECTOR_B )`	Dot (or inner) product[2]
`MATMUL ( MATRIX_A, MATRIX_B )`	Matrix multiplication[2]

Notes:
1. Both DPROD arguments must be single precision (default) real.
2. DOT_PRODUCT and MATMUL are transformational; the other intrinsic functions above are elemental.

Related Topics:

Data Representation Models
Intrinsic Function Overview
Intrinsic Subroutines

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 13.5, 13..11.2-3, 13.11.13, 13.14
Fortran 95 Handbook, 13.5, 13.8, A
Fortran 95 Using F, A.5, A.7

Character functions[1,2]

Function	Description
`ADJUSTL ( STRING )`	Move leading blanks to trailing
`ADJUSTR ( STRING )`	Move trailing blanks to leading
`INDEX ( STRING, SUBSTRING, BACK )`	Find position in STRING of first occurrence of SUBSTRING
`LEN_TRIM ( STRING )`	Length without trailing blanks
`LGE ( STRING_A, STRING_B )`	$A \geq B$, based on ASCII
`LGT ( STRING_A, STRING_B )`	$A > B$, based on ASCII
`LLE ( STRING_A, STRING_B )`	$A \leq B$, based on ASCII
`LLT ( STRING_A, STRING_B )`	$A < B$, based on ASCII
`REPEAT ( STRING, NCOPIES )`	Construct NCOPIES of STRING
`SCAN ( STRING, SET, BACK )`	Find position in STRING of first[3] occurrence of any char in SET
`TRIM ( STRING )`	Remove trailing blanks
`VERIFY ( STRING, SET, BACK )`	Find position in STRING of the first character not in SET[3]

Notes:
1. STRING, SUBSTRING, and SET are any character kind.
2. STRING_A and STRING_B are of type default character.
3. If optional BACK is absent or present with a value .TRUE. then the search in STRING is right to left; otherwise it is left to right.

Bit functions (integer arguments)[1,2,5]

Function	Description
`BTEST ( I, POS )`	Bit value in position POS in I
`IAND ( I, J )`	Logical “and” of bit strings I and J
`IBCLR ( I, POS )`	Set the POS bit in I to 0
`IBSET ( I, POS )`	Set the POS bit in I to 1
`IEOR ( I, J )`	Logical “xor” of bit strings I and J
`IOR ( I, J )`	Logical “or” of bit strings I and J
`ISHFT ( I, SHIFT )`	End-off shift bits in I by SHIFT[3]
`ISHFTC ( I, SHIFT, SIZE )`	Circular shift bits in I by SHIFT[3,4]
`NOT ( I )`	Logical “not” of bit string I

Notes:
1. The integers I and J are treated as bit strings in all of these functions; the bits are numbered from the left, starting with zero.
2. Except for BTEST, each bit function returns an integer representing the resulting bit string value, according to the bit model.
3. If SHIFT is positive, the shift is left; if negative, the shift is right.
4. If optional SIZE is omitted, the entire bit string is shifted; otherwise, just the rightmost SIZE bits are circularly shifted.
5. The MVBITS intrinsic subroutine completes the bit intrinsics.

56 Intrinsic Functions: Conversion

Fortran has a number of intrinsic functions to transfer or convert data values from one type and kind combination to another. Many of these functions have an optional argument KIND, which if present must be a scalar integer initialization expression; it specifies the kind of the function result; if KIND is omitted, the returned value is of default kind.

The accompanying list of functions is organized on the basis of the type of the first argument; except as noted, any valid kind is allowed for these arguments. All optional arguments are underlined in the accompanying list of conversion and transfer functions. All of these are elemental, except NULL and TRANSFER, which are transformational.

Examples:

```
INTEGER       :: K    ! Data objects used in the following
REAL          :: X    !   examples (assume integer constants
COMPLEX       :: Z    !   KANJI, BYTE, and QUAD previously set)
CHARACTER(10) :: C
```

Example references to various conversion intrinsic functions:

`INT(X+2.3)`	Integer part of the value of X+2.3
`CEILING (7.2)`	Returns 8
`REAL(Z)`	Real part of the complex value Z
`CHAR(K+1)`	The K+1st character in the default (processor) character set
`CHAR(K, KIND=KANJI)`	Returns the Kth character in the Kanji character set
`ICHAR(C(4:4))`	Collating position of the fourth character of C
`CONJG(Z)`	Computes the complex conjugate of the value of Z
`LOGICAL(X>3, KIND=BYTE)`	Returns true, kind BYTE, if X>3
`TRANSFER(X, K)`	Returns an integer value having the bit pattern of X
`CMPLX(K, KIND=QUAD)`	Returns K+0i in QUAD precision
`IBITS(K, 2, 3)`	Integer value of bits 2:4 of K
`NULL ()`	Returns a null pointer

Related Topics:

Intrinsic Function Overview   Kind Parameters

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 13.5, 13.6, 13.11, 13.14
Fortran 95 Handbook, 13.4, 13.8, A
Fortran 95 Using F, A.5.1, A.5.3, A.5.5, A.6, A.7

Conversion functions, for integer

`ACHAR ( I )`	Returns corresponding ASCII character
`CHAR ( I, KIND )`	Returns character of indicated KIND
`IBITS ( I, POS, LEN )`	Extracts LEN bits beginning at position POS of I
`CEILING ( A, KIND )`	Smallest integer larger than A
`FLOOR ( A, KIND )`	Largest integer smaller than A

Conversion functions, for real

`AINT ( A, KIND )`	Truncates A, but result is still real
`ANINT ( A, KIND )`	Rounds A to integer, but result is real
`NINT ( A, KIND )`	Rounds A to integer, result is integer

Conversion functions, for complex

`AIMAG ( Z )`	Extracts the imaginary portion of Z
`CONJG ( Z )`	Returns the complex conjugate of Z

Conversion function, for logical

`LOGICAL ( L, KIND )`	Converts L to kind KIND

Conversion functions, for character

`IACHAR ( C )`	Returns ASCII collating sequence for C[1]
`ICHAR ( C )`	Returns collating sequence value for C

Conversion functions, for any integer, real, or complex

`CMPLX ( X, Y, KIND )`	Constructs/converts a complex value[2]
`DBLE ( A )`	Converts A to double precision value
`INT ( A, KIND )`	Converts A to integer value, kind KIND
`REAL ( A, KIND )`	Converts A to real value, kind KIND

NULL function

`NULL ( MOLD )`	Returns a disassociated pointer

Transfer function

`TRANSFER ( SOURCE, MOLD, SIZE )`	Bits of SOURCE unchanged, but get type and kind of MOLD[3]

Notes:
1. The argument of IACHAR must be of default character kind.
2. In CMPLX, if Y is present, it must be either integer or real; if X is of type complex, Y must be omitted.
3. The result returned by TRANSFER has the same bit pattern as SOURCE, but has the type and kind of MOLD. If MOLD is a scalar, the result is scalar; otherwise, the result is a one-dimensional array. Note that the results of TRANSFER are not portable.

57 Intrinsic Functions: Inquiry and Model

Miscellaneous inquiry functions

`ASSOCIATED ( POINTER, TARGET )`	Pointer association status[1]
`BIT_SIZE ( I )`	Number of bits in bit string
`KIND ( X )`	Kind value of the argument
`LEN ( STRING )`	Number of characters in string
`PRESENT ( A )`	Presence of optional argument[2]
`SELECTED_INT_KIND ( R )`	Kind corresponding to range
`SELECTED_REAL_KIND ( P, R )`	Kind of precision/range[3]

Notes:
1. If TARGET is present, ASSOCIATED returns true if POINTER is associated with TARGET; if TARGET is not present, ASSOCIATED returns true if POINTER is associated with any target, and returns false if POINTER has been nullified.
2. The argument in a call to PRESENT must be the name of an optional dummy argument of the host procedure; PRESENT returns true if an actual argument is associated with the dummy argument, and false if not.
3. Both arguments in SELECTED_REAL_KIND are optional, but one must be present; the result for SELECTED_REAL_KIND (and SELECTED_INT_KIND) is the kind with the minimum possible precision and range for the properties specified.

Examples:

`LEN(C2//'si, en la cordillera')`	Number of characters in C2 + 20
`SPACING(X+.5)`	Spacing of real numbers near X+.5
`HUGE(1)`	Largest default integer value
`HUGE(1.0_QUAD)`	Largest real value of kind QUAD
`ALLOCATED(RES)`	True if array RES is allocated
`SIZE(RES, DIM=2)`	Extent (number of elements) of the second dimension of array RES
`ASSOCIATED(P,T)`	True if pointer P is currently associated with target T
`ASSOCIATED(P)`	True if P is currently associated with a target; false if P has been nullified; undefined otherwise
`PRECISION(X)`	The actual decimal precision of X

Data Representation Models Intrinsic Function Overview

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 13.5.7, 13.7, 13.8.5, 13.11
Fortran 95 Handbook, 13.3, 13.8, A
Fortran 95 Using F, A.6, A.8.5

Numeric model functions[1]

`EXPONENT ( X )`	Real model exponent value for X
`FRACTION ( X )`	Real model fraction value for X
`NEAREST ( X, S )`	Nearest processor real value[2]
`RRSPACING ( X )`	1/(relative spacing near X)
`SCALE ( X, I )`	X with model exponent changed by I[3]
`SET_EXPONENT ( X, I )`	Set the model exponent of X to I[3]
`SPACING ( X )`	Absolute spacing near X

Environmental inquiry functions[1]

`DIGITS ( X )`	Base digits of precision in model for X
`EPSILON ( X )`	Small value compared to 1 in model for X
`HUGE ( X )`	Largest model number in model for X
`MAXEXPONENT ( X )`	Max. exponent value in model for X
`MINEXPONENT ( X )`	Min. exponent value in model for X
`PRECISION ( X )`	Decimal precision in model for X
`RADIX ( X )`	Base (radix) in model for X
`RANGE ( X )`	Decimal exponent range in model for X
`TINY ( X )`	Smallest model number in model for X

Array inquiry function[4]

`ALLOCATED ( ARRAY )`	True if ARRAY is allocated
`LBOUND ( ARRAY, DIM )`	Lower bound(s) of ARRAY
`SHAPE ( SOURCE )`	Shape of the array SOURCE[5]
`SIZE ( ARRAY, DIM )`	Number of elements in an array
`UBOUND (ARRAY, DIM )`	Upper bound(s) of ARRAY

Syntax:

Syntax:

Notes:
1. Generally X is type real; X may be integer for DIGITS, HUGE, RADIX, and RANGE; X may be complex for PRECISION and RANGE.
2. Argument S tells which direction to select the nearest value to X.
3. Argument I is an integer that specifies the value of the change.
4. DIM is an optional argument that specifies the dimension of ARRAY along which the operation is performed; in the absence of DIM, SIZE returns the number of elements in the entire array, and both LBOUND and UBOUND return one-dimensional arrays of the respective bound values for each dimension.
5. SHAPE returns a one-dimensional array containing the extent of SOURCE in each dimension; the size of SHAPE equals the rank of SOURCE.

58 Intrinsic Subroutines

Intrinsic functions are free of side effects and have well-defined results. Subroutines, rather than functions, are appropriate for those intrinsic procedures that involve side effects or for which the results can take more than one form. Two of the six intrinsic subroutines, RANDOM_NUMBER, and RANDOM_SEED, involve side effects, as described in the accompanying summaries of these procedures. Two intrinsic subroutines, DATE_AND_TIME and SYSTEM_CLOCK, provide several options for the results they return. Intrinsic subroutines are described in detail Appendix A.

Examples:

```
INTEGER       :: J, K, L, M(1000) ! Data objects used in the
REAL          :: R(200)           !   following examples
CHARACTER(10) :: D, T
```

Examples of references to intrinsic subroutines are:

`RANDOM_NUMBER(R(17))`	Sets the value of R(17) to a pseudorandom number
`RANDOM_NUMBER(R(1:40))`	Generates 40 pseudorandom numbers
`CPU_TIME (CURRENT_TIME)`	Gets current CPU time
`SYSTEM_CLOCK(COUNT=K, & COUNT_MAX=L)`	Gets current system clock value and its maximum value
`MVBITS(M(J),K,L,M(J+1),K)`	Copies L bits from M(J) into the same bit positions in M(J+1)
`DATE_AND_TIME(TIME=T, ZONE=D)`	Obtains the current time and time zone
`DATE_AND_TIME(DATE=D, TIME=T)`	Obtains the current date and time
`RANDOM_SEED(SIZE=L)`	Sets the value of L to the current random number seed size
`RANDOM_SEED(PUT=M(J:J+L-1))`	Initializes the random number seed to the value specified by L elements of M
`MVBITS(K, 1, 4, K, 9)`	Copies (replicates) the bits from positions 1:4 of K into positions 9:12 of K

Related Topics:

Intrinsic Function Overview   Subroutines

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 13.10, 13.11, 13.14
Fortran 95 Handbook, 13.7, 13.8, A
Fortran 95 Using F, 4.7, A.10

CPU_TIME (TIME)

TIME is an intent OUT real scalar that is assigned the processor time in seconds.

DATE_AND_TIME (DATE, TIME, ZONE, VALUES)

All of these arguments are optional (although at least one must be supplied in any given call to DATE_AND_TIME) and all are intent OUT. DATE is an 8-character string of the form *ccyymmdd* (century, year, month, day). TIME is a 10-character string of the form *hhmmss.sss* (hours, minutes, seconds-milliseconds). ZONE is a 5-character string of the form ±*hhmm* (hours and minutes) offset from Coordinated Universal Time in hours and minutes). VALUES is an 8-element integer array that returns all of these values as integers.

MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

All five of these arguments are integer and all are intent IN except TO, which is intent INOUT. The LEN bits of FROM, starting from position FROMPOS, are copied into the LEN positions of TO starting at TOPOS; no other bits in TO are changed. MVBITS may be called elementally (it is the only elemental intrinsic subroutine). This could have been an intrinsic function, but because it is a subroutine in the MIL-STD 1753 bit procedures, the same form is used in Fortran.

RANDOM_NUMBER (HARVEST)

HARVEST is of type real, with intent OUT, and may be either a scalar or an array. It is set to contain pseudorandom numbers uniformly distributed in the interval [0,1). Executing RANDOM_NUMBER has the side effect of changing the seed value of the random number generator.

RANDOM_SEED (SIZE, PUT, GET)

All three of these arguments are optional; at most one of them is supplied in any given call to RANDOM_SEED. SIZE is a default scalar integer with intent OUT that is set to the number of integers that the processor uses to hold the pseudorandom number seed. PUT is a one-dimensional default integer array at least the size of the seed, with intent IN that specifies the value to which the seed is set. GET is a rank-one default integer array at least the size of the seed with intent OUT that is set to the current seed value.

SYSTEM_CLOCK (COUNT, COUNT_RATE, COUNT_MAX)

All three of these arguments are optional; at least one must be supplied in any given call to SYSTEM_CLOCK, and all are intent OUT. COUNT is an integer, set to the current value of the processor clock. COUNT_RATE is an integer, set to the number of processor counts per second. COUNT_MAX is an integer set the largest value that COUNT can have.

59 Kind Parameters

Kind parameters provide a way to parameterize a selection from among available machine representations for an intrinsic data type. This parameterization makes selection of numeric precision and range portable. For the character data type, it permits the use of more than one character set, such as Japanese, Chinese, and chemistry symbols, within a program.

Examples:

```
! Declarations of variables with specified kind parameters:
INTEGER, PARAMETER :: SHORT = SELECTED_INT_KIND (3)
INTEGER (KIND = SHORT) :: A, B, C, D

! A literal constant used as a kind parameter:
REAL (KIND = 2)  X, Y
COMPLEX (2) Z1, Z2, Z3

! Declarations of variables with default kind parameters:
REAL X, Y, Z
```

Examples of constants with nondefault kind parameters follow. LONG, SHORT, ASCII, and GREEK are integer named constants.

```
2.7182818284590455_LONG
-2_SHORT
.TRUE._1
ASCII_"abcde"
GREEK_"αβγδε"
```

Tip: It is not a good idea to use literal constants as kind values; named constants are better. If the initialization of a named constant is placed in a module, it needs to be changed only in one place when porting the program to another system.

Related Topics:

Character Type and Constants
Complex Type and Constants
Integer Type and Constants
Intrinsic Functions: Inquiry and Model
Logical Type and Constants
Portable Precision Control
Real Type and Constants

Related Intrinsics:

KIND (X)
SELECTED_INT_KIND (R)
SELECTED_REAL_KIND (P, R)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.3, 5.1, 13.5.5, C.1.2
Fortran 95 Handbook, 4.1, 4.3, 13.3
Fortran 95 Using F, 1.2, 1.5.1, 5.1, A.5.6

Syntax:

A kind selector in a declaration statement is:

[KIND =] *scalar-integer-initialization-expression*

A kind parameter in a literal constant is one of:

digit-string *named-integer-constant*

Things To Know:

1. Each intrinsic data type (integer, real, complex, logical, and character) has a parameter, called its **kind parameter**, associated with it. A kind parameter is used to designate a machine representation for a particular data type. As an example, an implementation might have three real kinds, informally known as single, double, and quad precision.
2. The kind parameter is an integer. These numbers are processor dependent, so that kind parameters 1, 2, and 3 might be single, double, and quad precision on one system, and on a different system, kind parameters 4, 8, and 16, kind parameters 32, 64, and 128, or kind parameters 17, 3, and 25, could be used for the same things. Note that the value of the kind parameter does not necessarily have anything to do with the number of decimal digits of precision or range.
3. Each intrinsic type has a default kind. In addition, there is a second real kind that corresponds to double precision.
4. A complex value consists of two real values with the same kind; hence, the kinds used for complex values are the same as those for real values. In particular, there are complex values with real and imaginary parts that have the kind of double precision.
5. When a kind parameter is a part of another constant, it may be either an integer constant (which does not have a sign) or a named integer constant. In integer, real, and logical constants, it follows the underscore character (_) at the end. In character constants, it occurs at the front and is followed by an underscore.
6. The kind of a complex constant is determined by the kinds of the two real components as follows:
   - If the two parts are type integer, the kind of the complex constant is the default complex kind.
   - If one part is integer and the other is real, the kind of the complex constant is the kind of the real.
   - If both parts are real, the kind of the complex constant is the same as the part with the greater precision.
7. The KIND intrinsic function returns the kind of its argument; the kind value is a positive integer.

60 Language Evolution

As the technology grows and changes, languages such as Fortran must evolve. From the earlier Fortrans to the Fortran of today, there have been many additions to the language but very few deletions. Hollerith data was removed from the Fortran 77 standard; no deletions were made to Fortran 90. In this standard, there are new features and deleted features.

New features often replace those that have become archaic in the language and rarely used. They may make some facilities in Fortran redundant. In order to alert the user to redundant and seldom used features, they are declared obsolescent. There are three classes of features used as guidelines for the language evolution of Fortran. The three classes of features are (1) New, (2) Obsolescent, and (3) Deleted.

New Features

New features provide additional facilities in Fortran in response to user needs and changing technology. In some cases, new features provide alternatives to existing features.

Obsolescent Features

The obsolescent features are those features of Fortran 90 that are redundant and for which better methods are available in Fortran 95. The arithmetic IF, the shared DO termination, and alternate return were obsolescent in Fortran 90 and remain so in Fortran 95. There are seven items on the list.

Obsolescent features may return to the main body of features in the language, remain obsolescent, or be removed. The use of these features is discouraged; however, if their use is still important to users, the feature will not be deleted.

Deleted Features

Deleted features are chosen from the obsolescent features of the previous standard. They are often redundant with a new feature or are considered largely unused. The list of deleted features for Fortran 90 was empty; there were none. In the current Fortran standard, there are some.

To Read More About It:

ISO 1539 : 1997, Fortran Standard, B
Fortran 95 Handbook, C

Language Evolution 60

New Features

FORALL statement and construct
PURE procedures
ELEMENTAL procedures
User-defined functions in specification expressions
Nesting of WHERE construct
Default initialization
NULL intrinsic function
CPU_TIME intrinsic subroutine
Extensions to CEILING, FLOOR, MAXLOC, MINLOC
Automatic deallocation of allocatable arrays
Namelist comments in namelist input
Minimal field widths
Support aspects of IEEE 754/854 arithmetic

Obsolescent Features

Alternate return
Computed GO TO statement
Statement functions
Data statements among executable statements
Assumed-length character-valued functions
Fixed source form
CHARACTER* form of character declaration

Deleted Features

Real and double precision DO variables
Branching to and END IF statement from outside its IF construct
PAUSE statement
ASSIGN statement, assigned GO TO statement, and assigned format
The H edit descriptor

61 Logical Type and Constants

The Fortran logical type is used to represent truth values. The truth values are "true" and "false". The LOGICAL type declaration statement declares variables, named constants, and functions to be of logical type. Kind parameters may be used to select alternative ways to represent logical data; for example, a kind parameter may define a logical data type where the truth values are packed one per bit. Alternatively truth values could be represented in a byte.

Examples:

```
PARAMETER (BIT=1, BYTE=2)

LOGICAL LX                          ! LX is default logical type.
LOGICAL (KIND=BIT) KAPPA            ! Kind parameter is BIT.
LOGICAL, DIMENSION (15:30) :: MX, MY
LOGICAL, SAVE :: X(0:99)
LOGICAL (KIND = BYTE), ALLOCATABLE :: TR (:,:,:)
LOGICAL (KIND = BIT), TARGET :: BTEST(1000)
LOGICAL (KIND=BYTE) :: YTERM        ! BYTE is a kind parameter.
```

Examples of logical constants are:

```
.TRUE.
.FALSE._BIT
.TRUE._BYTE
```

Related Topics:

Implicit Typing
Kind Parameters

Related Intrinsics:

KIND (X)
LOGICAL (L, KIND)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.3.2.2, 5.1.1.6, 7.1.1.6, 7.2.4
Fortran 95 Handbook, 4.3.4, 5.1.5
Fortran 95 Using F, 1.2.7, 1.5.2, A.5.5

Logical Type and Constants

Syntax:

A LOGICAL type declaration statement is:

```
LOGICAL [ ( [ KIND = ] kind-parameter ) ] [ , attribute-list :: ] entity-list
```

A logical constant is one of:

```
.TRUE. [ _ kind-parameter ]
.FALSE. [ _ kind-parameter ]
```

Things To Know:

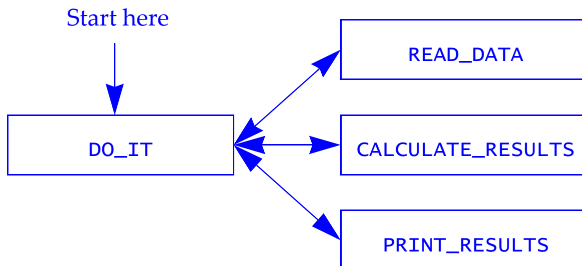
1. At least one kind of logical type must be available; it is the default logical intrinsic type.
2. The default logical constants are .TRUE. and .FALSE.
3. The operators are: .AND., .OR., .EQV., .NEQV. and the unary operator .NOT. The intrinsic logical operators must have both operands of type logical but may be of different kinds. The result is of type logical, of kind equal to the kind of the operands when both have the same kind, and of a processor-dependent kind when they are different kinds.
4. The result of comparing values of other types is a value of default logical type.
5. A variable of default logical type occupies one numeric storage unit.
6. Values assigned to kind parameters are processor dependent. The local documentation must be checked to learn the values of the kind parameters and which representation methods exist.
7. The elemental function LOGICAL converts between objects of type logical with different kind type parameter values. A second argument, KIND, is optional. If the kind argument is not present, the kind type is default logical.

62 Main Program

A main program is one of the kinds of program units; there is exactly one main program in an executable program. Execution always begins with the main program. The main program can determine the overall design and structure of the complete Fortran program and often performs various computations by referencing procedures. A Fortran program may be only a main program, in which case all the program logic is contained within it.

Examples:

```
PROGRAM  ELECTRIC           ! Program header
   REAL  CURRENT            ! Specification part
   CURRENT = 100.5          ! Execution part begins
      . . .
   CALL COMPUTE_RESISTANCE( VOLTAGE, CURRENT, RESISTANCE )
      . . .
   CONTAINS                 ! Internal program part
      SUBROUTINE  COMPUTE_RESISTANCE( V, I, R )
         REAL  I
         R = V / I
      END SUBROUTINE
END PROGRAM  ELECTRIC

! The simplest, but not very useful, main program unit
END

PROGRAM DO_IT  ! The PROGRAM statement is optional.
   . . .
   CALL READ_DATA
   CALL CALCULATE_RESULTS
   CALL PRINT_RESULTS
END PROGRAM DO_IT
```



Related Topics:

Program Units

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 2.1, 11.1, C.8.1
Fortran 95 Handbook, 2.2.1, 2.8, 11.2
Fortran 95 Using F, 1.1

Syntax:

A main program is:

```
[ PROGRAM program-name ]
   [ specification-statement ]...
   [ executable-construct ]...
   [ CONTAINS
      internal-procedure
      [ internal-procedure ]... ]
END [ PROGRAM [ program-name ] ]
```

Things To Know:

1. A main program has three parts (similar to other program units): a specification part, an execution part and an internal procedure part which begins with the CONTAINS statement. All three parts may be empty.
2. The data environment is described in the specification part. The data environment includes such things as the attributes of variables, type definitions, and initial values.
3. An automatic object must not appear in the specification part of a main program.
4. The SAVE attribute in a main program is permitted but has no effect.
5. An executable construct is a CASE, DO, IF, or WHERE construct, and action statements such as an assignment statement, data transfer statement, or IF statement. Neither an ENTRY statement nor a RETURN statement is permitted in a main program.
6. The internal subprogram part contains one or more internal procedures.
7. The PROGRAM statement is optional; if it appears, the program name may be used on the END statement.

63 Modules

Modules are nonexecutable program units that contain type definitions, object declarations, procedure definitions (module procedures), external procedure interfaces, user-defined generic names, user-defined operators and assignments, common blocks, and namelist groups. Any such definitions not specified to be private to the module containing them are available to be shared with those programs that use the module. Thus modules provide a convenient sharing and encapsulation mechanism for data, types, procedures, and procedure interfaces.

Examples:

```
MODULE SHARED                               ! Making data objects
   COMPLEX GTX (100, 6)                     !  and a data type
   REAL, ALLOCATABLE :: Y(:), Z(:,:)        !  sharable via a module
   TYPE PEAK_ITEM
      REAL PEAK_VAL, ENERGY
      TYPE(PEAK_ITEM), POINTER :: NEXT
   END TYPE PEAK_ITEM
END MODULE SHARED

MODULE RATIONAL_ARITHMETIC                  ! Defining a data
   TYPE RATIONAL; PRIVATE                   !  abstraction for
      INTEGER NUMERATOR,DENOMINATOR         !  rational arithmetic
   END TYPE RATIONAL                        !  via a module
   INTERFACE ASSIGNMENT (=)                 ! Generic extension of =
      MODULE PROCEDURE ERR, ERI, EIR
   END INTERFACE
   INTERFACE OPERATOR (+)                   ! Generic extension of +
      MODULE PROCEDURE ARR, ARI, AIR
   END INTERFACE
      . . .
CONTAINS
   SUBROUTINE ERR (. . .)               ! A specific definition of =
      . . .
   FUNCTION ARR (. . .)                 ! A specific definition of +
      . . .
END MODULE RATIONAL_ARITHMETIC
```

Related Topics:

Defined Operators and Assignment
Defined Type: Objects
Host Association
Module Procedures
Program Units
PUBLIC and PRIVATE Attributes and Statements
USE Statement and Use Association

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 2.2.4, 11.3, C.8.3
Fortran 95 Handbook, 2.2.1, 11.6
Fortran 95 Using F, 3.4, 7

Syntax:

A module is:

```
MODULE module-name
   [ specification-part ]
[ CONTAINS
   module-subprogram
   [ module-subprogram ]...]
END [ MODULE [ module-name ] ]
```

Things To Know:

1. A module does not contain executable code except the execution parts of any module subprograms.
2. The specification part of a module must not contain the following attributes or statements: ENTRY, FORMAT, INTENT, OPTIONAL, or statement function statement. Similarly, the specification part of a module must not contain automatic objects; all of these may appear in module procedures, however.
3. PUBLIC and PRIVATE attributes and statements are allowed only in the specification part of a module. PUBLIC specifies the designated entity as sharable by using program units. PRIVATE specifies the designated entity as not sharable but rather private within the module; such entities are fully shared and accessible among the module procedures of the module by host association.
4. A MODULE PROCEDURE statement may appear only in an interface block that has a generic specification. The interface block must be in a module that contains the procedure or in a host that accesses the module.
5. SAVE attributes and statements can be used in a module to preserve data values among uses of the module. If such values are to remain intact when all program units using the module are inactive, SAVE must be specified.
6. Module procedures are like internal procedures in that they access the host environment by host association as well as its implicit type mapping, but otherwise they are like external procedures.
7. Modules are ideal for data abstraction, generic procedure definition, operator extension, and the sharing of such information to all program units of an application that need it.

64 Module Procedures

A module procedure is defined by a function subprogram or a subroutine subprogram appearing in the subprogram part of a module. Module procedures are accessible via use association. The data environment of the module is available to and shared among the procedures in the module.

Examples:

```
MODULE PATH
REAL X, Y, Z                 ! Module data environment
                             ! Module procedures contained in this
                             !   module have access to this data
                             !   environment.

INTERFACE SUBSTANCE                 ! Generic name SUBSTANCE
   MODULE PROCEDURE AIR, WATER      !   for procedures
END INTERFACE                       !   AIR and WATER

INTERFACE OPERATOR (*)
   MODULE PROCEDURE RATIONAL_MULTIPLY
END INTERFACE
   . . .
CONTAINS                    ! Module procedures preceded by CONTAINS
   SUBROUTINE AIR (CONTENTS)
      . . .
   END SUBROUTINE AIR       ! End of module procedure AIR

   SUBROUTINE WATER (X, A, Z)
      A = X + Y             ! X is a dummy argument.
      . . .                 ! Y is from the module data environment.
   END SUBROUTINE           ! End of module procedure WATER

   FUNCTION RATIONAL_MULTIPLY (X, Y)
      TYPE (RATIONAL) :: RATIONAL_MULTIPLY
      TYPE (RATIONAL), INTENT (IN) :: X, Y
         . . .
      RATIONAL_MULTIPLY = . . .
         . . .
   END FUNCTION RATIONAL_MULTIPLY

END MODULE PATH
```

Related Topics:

Functions
Interfaces and Interface Blocks
Internal Procedures
Modules
Subroutines

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 2.2.3.2, 11.3, 12.1.2.2, 12.3.2.1, C.8.3.7
Fortran 95 Handbook, 11.6.3, 12.1.1.7, 12.8.3
Fortran 95 Using F, 3.4, 7

Syntax:

A module subprogram part is:

```
CONTAINS
    module-subprogram
    [ module-subprogram ]...
```

A module subprogram is one of:

```
module-function-subprogram
module-subroutine-subprogram
```

A module function subprogram is:

```
function-statement
    [ specification-part ]
    [ execution-part ]
    [ internal-subprogram-part ]
END FUNCTION [ function-name ]
```

A module subroutine subprogram is:

```
subroutine-statement
    [ specification-part ]
    [ execution-part ]
    [ internal-subprogram-part ]
END SUBROUTINE [ subroutine-name ]
```

Things To Know:

1. The organization, rules, and restrictions for module procedures follow more closely those of external procedures, rather than internal procedures.
2. A module procedure may contain an internal procedure. Also, a module procedure can be passed as a procedure argument.
3. Interfaces to module procedures are explicit when a module is used provided they are not private. Thus it is neither necessary nor allowed to create an interface block for a module procedure. To avoid the need for interface blocks, use module procedures.
4. A module procedure is accessible to those program units that use the module, provided that it is not declared to be PRIVATE. (See Modules.)
5. The rules for host association and implicit typing are those described for internal procedures.
6. At least one procedure must follow the CONTAINS statement.
7. Note that the keyword FUNCTION or SUBROUTINE is required on the END statement of a module procedure. The name of the function or subroutine also may appear on the END statement.

65 OPEN Statement

The OPEN statement connects a unit and a file and establishes various connection properties. The properties are given by keyword specifiers, many of which have default values, if omitted. A file must be connected (either by an OPEN statement or by the processor) in order for data to be read or written. In certain cases, the OPEN statement is used to change the connection properties of files that are already established in a previous connection. If a file does not exist, execution of an OPEN statement for that file may create the file.

Examples:

```
OPEN (11, STATUS="SCRATCH", BLANK="ZERO", IOSTAT=IR)

! Change interpretation of blanks on unit 11.
OPEN (BLANK = "NULL", UNIT = 11)

RLEN = 20
OPEN (9, ACCESS = "DIRECT", FILE = “DATA_FILE”, &
         ACTION = “READ”, RECL = 2*RLEN )

IN = 8
OPEN (UNIT=IN,STATUS = "REPLACE", ERR = 8, FILE = "DISK08" )

! Connect unit 7 to the new file named "info.txt".
!   Because it is not specified explicitly, the file is,
!   by default, assumed to be formatted sequential access.
!   If an error condition occurs, the program terminates.

OPEN (7, STATUS = ’NEW’, FILE = "info.txt1")

! Connect a unit to a processor-determined scratch file
!   for unformatted sequential access.  If an error condition
!   occurs, transfer to the branch target statement labeled 9.

IS = IT * IA
OPEN (ACCESS = ’SEQUENTIAL’, FORM = "UNFORMATTED", UNIT = IS,  &
      ERR = 9, STATUS = ’SCRATCH’)
```

Related Topics:

CLOSE Statement
INQUIRE Statement
READ/WRITE General Form

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.3.4, C.6.2-3
Fortran 95 Handbook, 9.5
Fortran 95 Using F, 9.2.7, 9.4

Syntax:

An OPEN statement is:

OPEN ([UNIT =] *scalar-integer-expression* [, *connection-specifier-list*])

The connection specifiers appear in the following table.

Specifier=		Values	Default	Description
ACCESS=exp	C	DIRECT, SEQUENTIAL	SEQUENTIAL	File access method
ACTION=exp	C	READ, WRITE, READ-WRITE	Proc. dep.	Direction of input/output
BLANK=exp	C	NULL, ZERO	NULL	Interpretation of blanks
DELIM=exp	C	APOSTROPHE, QUOTE, NONE	NONE	Char. str. delimiter for list-directed I/O
ERR=label	Lb	Label	None	Branch target taken on an error condition
FILE=exp	C	Character string	None	Name of file
FORM=exp	C	FORMATTED UNFORMATTED	FORMATTED UNFORMATTED	Formatting, default for sequential access Formatting, default for direct access
IOSTAT=var	I	Positive Zero		An error condition occurred No error condition occurred
PAD=exp	C	YES, NO	YES	Blank padding
POSITION=exp	C	ASIS, REWIND, APPEND	ASIS	Initial position on open
RECL=exp	I	Positive	Proc. dep.	Record length
STATUS=exp	C	OLD, NEW, UNKNOWN, SCRATCH, REPLACE	UNKNOWN	Initial file status
Lb, I, C — label, integer, character default scalar expression; the character values are without regard to case and trailing blanks are ignored				

Things To Know:

1. A unit may not be connected to more than one file at one time, but it may be connected to different files at different times.
2. The following open properties may be changed without closing the file: BLANK, DELIM, and PAD. ERR, and IOSTAT may be present.
3. If the status specifier is OLD, NEW, or REPLACE, a file name specifier must appear.

66 OPTIONAL Attribute and Statement

If a dummy argument has the OPTIONAL attribute, the corresponding actual argument may appear or be omitted in a procedure reference. In cases where there are arguments that generally do not change from one reference to another, it is convenient to specify that the arguments are optional and provide default values for them. They can then be omitted from references in these general cases. The PRESENT intrinsic function may be used within the procedure to determine whether in a particular reference an actual argument has been supplied for an optional dummy argument.

Examples:

```
CALL TRIP ( DISTANCE = 17.0 )          ! PATH is omitted.
   . . .
SUBROUTINE  TRIP ( DISTANCE, PATH )
   OPTIONAL DISTANCE, PATH
      . . .
SUBROUTINE PLOT (PTS, O_XAXIS, O_YAXIS, SMOOTH)
   TYPE (POINT) PTS
   REAL, OPTIONAL :: O_XAXIS, O_YAXIS  ! Origin: default (0.,0.)
   LOGICAL, OPTIONAL :: SMOOTH
   REAL OX, OY
   IF (PRESENT (O_XAXIS)) THEN; OX=O_XAXIS; ELSE; OX=0.; END IF
   IF (PRESENT (O_YAXIS)) THEN; OY=O_YAXIS; ELSE; OY=0.; END IF
   IF (PRESENT(SMOOTH)) THEN
      IF (SMOOTH) THEN
         . . .                         ! Smooth algorithm
         RETURN
      END IF
   END IF
      . . .                            ! Plot points
END SUBROUTINE PLOT
CALL PLOT (POINTS)                     ! Valid calls to PLOT
CALL PLOT (OBSERVED, O_XAXIS = 100., O_YAXIS = 1000.)
CALL PLOT (RANDOM_PTS, SMOOTH = .TRUE.)
```

Tip: Many uses of the ENTRY statement can, and should, be replaced by the use of optional arguments.

Related Topics:

Argument Association   Argument Keywords

Related Intrinsics:

PRESENT (A)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.6, 5.2.2, 12.4.1.5, 13.14.82
Fortran 95 Handbook, 5.6.3, 12.7.5, A.82
Fortran 95 Using F, 3.8.7

Syntax:

A type declaration statement with the OPTIONAL attribute is:

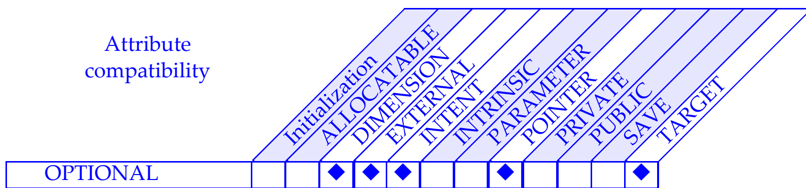
type , `OPTIONAL` [, *attribute-list*] `::` *dummy-argument-name-list*

An OPTIONAL statement is:

`OPTIONAL` [`::`] *dummy-argument-name-list*

Things To Know:

1. The OPTIONAL attribute may be specified only for dummy arguments. This may occur in a subprogram and in any corresponding interface body.
2. An optional dummy argument whose actual argument is not present may not be referenced or defined (or invoked if it is a dummy procedure), except that it may be passed to another procedure as an optional argument and will be considered not present.
3. When an argument is omitted in a procedure reference, all arguments that follow it must use the keyword form. If a procedure has an optional argument, the procedure interface must be explicit.
4. The presence of an optional argument in a procedure may be determined by using the PRESENT function. This function returns a scalar logical value true if the actual argument is present, otherwise the value false is returned. The logical result value may be tested and thus select subsequent program actions based on whether or not an actual argument is present.



67 PARAMETER Attribute and Statement

The PARAMETER attribute in a type declaration declares a named constant. It is used in conjunction with a type declaration and an initialization expression, and the combination specifies the type and a value for the named constant. The PARAMETER statement also specifies a named constant and its value. The type is specified separately in a type declaration statement or by the implicit typing rules in effect. The value of a named constant or parameter does not change during program execution; it is fixed. If the keyword PARAMETER is omitted in a type declaration statement, the name is a variable and not a named constant and may be changed during program execution.

Examples:

```
! Specify CLASSIC as a default logical named constant
!   with value true using a type statement.
LOGICAL, PARAMETER :: CLASSIC = .TRUE.

! Specify WAVE as a real constant with kind parameter HI.
!    Y is default real type.
REAL (KIND = HI) WAVE, T
PARAMETER (T = 1.1, WAVE = 5._HI * T, Y = 42.5)

! Specify X as an array-valued named constant.
! The value is specified using an array constructor.
REAL, PARAMETER, DIMENSION (2) :: X  = (/ 1.1, 2.2 /)

! Specify the origin ORIGIN as a named constant.
TYPE CARTESIAN
  REAL X, Y
END TYPE CARTESIAN
type (CARTESIAN), parameter :: ORIGIN = CARTESIAN (0.0, 0.0)
```

Tip: Using parameters where possible is good programming practice. This is particularly true when portability is a concern or when the forms and values of constants must be changed to move the program to a different computer. With the use of named constants, modifications can be made easily. In addition, the program is typically more readable when named constants are used.

Related Topics:

Data Initialization
Expressions: Initialization
Implicit Typing
Kind Parameters

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.1, 5.2.9
Fortran 95 Handbook, 5.5.2
Fortran 95 Using F, 1.2.9, 5.1.2

Syntax:

A type declaration statement with the PARAMETER attribute is:

type , PARAMETER [, *attribute-list*] :: *name* = *initialization-expression*

A PARAMETER statement is:

PARAMETER (*name* = *initialization-expression*)

Things To Know:

1. When the PARAMETER statement is used, subsequent implicit or explicit type declarations may only confirm the type of the named constant.
2. Arrays in a PARAMETER statement must have all of their array properties declared in the same or a previous statement.
3. Named constants must not appear in a format specification. They also may not appear as part of a literal constant, except as a kind value. In particular, a named constant may not be the real or imaginary part of a complex constant.
4. The PARAMETER attribute must not be declared for dummy arguments, functions, pointers, automatic objects, allocatable arrays, or items in a common block.

68 Pointers

Pointers are used to provide dynamic-data-object and aliasing capabilities in Fortran. By deferring the sizes of objects to execution time, a code can run at the exact size needed; recompilation for unusual cases is no longer required. Dynamic structures such as lists and trees can grow in ways that could not be anticipated when the program was written. The use of pointer aliasing can contribute to more readable, maintainable code.

The elements of the Fortran pointer facility are: two attributes, POINTER and TARGET; four statements, NULLIFY, ALLOCATE, DEALLOCATE, and pointer assignment; and two intrinsic functions, ASSOCIATED and NULL.

Examples:

```
REAL, POINTER :: WEIGHT (:,:,:)     ! Extents are not specified;
REAL, POINTER :: W_REGION (:,:,:)   !   they are determined
READ *, I, J, K                     !   during execution.
   . . .
ALLOCATE (WEIGHT (I, J, K))               ! WEIGHT is created.
W_REGION => WEIGHT (3:I-2, 3:J-2, 3:K-2)  ! W_REGION is an alias
                                         ! for an array section.
AVG_W = SUM (W_REGION) / ( (I-4) * (J-4) * (K-4) )
   . . .
DEALLOCATE (WEIGHT)            ! WEIGHT is no longer needed.

TYPE CATALOG
   INTEGER :: ID, PUB_YR, NO_PAGES
   CHARACTER, POINTER :: SYNOPSIS (:)
END TYPE CATALOG
   . . .
TYPE(CATALOG), TARGET :: ANTHROPOLOGY (5000)
CHARACTER, POINTER :: SYNOPSIS (:)
   . . .
DO I = 1, 5000
   SYNOPSIS => ANTHROPOLOGY(I) % SYNOPSIS! Alias for a component
   WRITE (*,*) HEADER, SYNOPSIS, DISCLAIMER! of an array element
      . . .
END DO
```

Related Topics:

ALLOCATE and DEALLOCATE Statements
Dynamic Objects
Interfaces and Interface Blocks
Pointer Association
POINTER Attribute and Statement
Pointer Nullification
TARGET Attribute and Statement

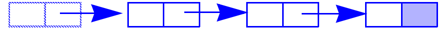
To Read More About It:

ISO 1539 : 1997, Fortran Standard, 2.4.6, 5.1.2.7-8, 6.3, 7.5.2, 13.9, 13.14.13, 13.14.79, 14.6.2, C.1.3, C.2, C.3.2, C.4.3-4

Fortran 95 Handbook, 2.3.4, 5.4, 6.5, 7.5.3, A.13, A.79

Fortran 95 Using F, 8

Linked List Example



```
TYPE LINK
   REAL VALUE
   TYPE (LINK), POINTER :: NEXT => NULL( )
END TYPE LINK
TYPE(LINK), POINTER :: LIST => NULL( ), SAVE_LIST
   . . .
DO
   READ (*, *, IOSTAT = NO_MORE) VALUE
   IF (NO_MORE /= 0) EXIT
   SAVE_LIST => LIST
   ALLOCATE (LIST)                      ! Add link to head of list.
   LIST % VALUE = VALUE
   LIST % NEXT => SAVE_LIST
END DO
   . . .
DO                                      ! Linked list can be
   IF (.NOT.ASSOCIATED (LIST) ) EXIT !   removed when no
   SAVE_LIST => LIST % NEXT             !   longer needed.
   DEALLOCATE (LIST)
   LIST => SAVE_LIST
END DO
```

Things To Know:

1. POINTER is an attribute in Fortran—not a type. An object of any type can have the POINTER attribute. Such an object cannot be referenced until it is associated with a target. A pointer target must have the same type, rank, and kind as the pointer. When the name of an object with the POINTER attribute appears in most executable statements, it is its target that is referenced.
2. To be a candidate for a pointer target, most objects must be given the TARGET attribute; a pointer has this attribute implicitly. A target may be thought of as an object with dynamic names.
3. When the name of an object with the POINTER attribute appears in certain places, it is the pointer that is referenced. These include pointer initialization, the left side of a pointer assignment statement, a NULLIFY, ALLOCATE and DEALLOCATE statement, and arguments of the ASSOCIATED and NULL intrinsic functions. A function may return a pointer or have pointer arguments; if so, the function must have an explicit interface.
4. Recursive procedures are helpful in dealing with dynamic structures such as lists and trees.

69 Pointer Association

The pointer assignment statement causes a variable with a POINTER attribute to become associated with a specified target, where the variable is on the left of the pointer assignment symbol => and the target is on the right or to become dissassociated if the target is a reference to the NULL intrinsic function or to a disassociated pointer.

The ALLOCATE statement creates an association between a pointer and a target, creating space for the target, leaving the target undefined. The DEALLOCATE statement frees the space allocated for the target and dissociates the pointer from its target. The NULLIFY statement and pointer assignment referencing the intrinsic function NULL disassociate the pointer from any target. A pointer with a defined association status (associated or disassociated) can be queried with the ASSOCIATED intrinsic function.

Examples:

```
REAL, TARGET :: X; REAL, POINTER :: PTR_X
PTR_X => X              ! The target of PTR_X becomes X
PTR_X = 13.3            ! This statement changes the target of
                        !   PTR_X, namely X, to 13.3

TYPE(LIST), POINTER :: LP => NULL()  ! LP is initialized as disassociated.

INTEGER, DIMENSION(10), TARGET :: I
INTEGER, DIMENSION(10,20), TARGET :: K
INTEGER, POINTER, DIMENSION(:) :: PTR_I, PTR_K
PTR_I => I              ! The target of PTR_I becomes the
                        !   array I of 10 elements.
PTR_K => K(3,:)         ! The target of PTR_K becomes the
   . . .                !   third row of K of 20 elements.
NULLIFY( PTR_I )        ! PTR_I becomes disassociated.

CHARACTER, POINTER :: J(:,:)
ALLOCATE ( J(3,4) )     ! J is now associated with allocated space
                        !   but the target is not defined.
```

Related Topics:

ALLOCATE and DEALLOCATE Statements
Assignment
Dynamic Objects
Expressions
Pointers
Pointer Nullification
Variables

Related Intrinsics:

ASSOCIATED (POINTER, TARGET)
NULL (MOLD)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 6.3, 7.5.2, 13.14.13, 13.14.79, 14.6.2, C.1.3, C.3.2, C.4.4
Fortran 95 Handbook, 6.5.1, 7.5.3, A.13, A.79
Fortran 95 Using F, 8.1, A.9

Syntax:

A pointer assignment is:

pointer-object => target

A pointer object is one of:

variable
structure-component

Things To Know:

1. A target is an expression, which is limited to a variable with the TARGET attribute, a subobject of a variable with the TARGET attribute, a variable with the POINTER attribute, or a function reference or defined operation that returns a pointer result.
2. The pointer object must have the POINTER attribute.
3. The type, type parameters (kind and length), and rank of the pointer object and target must be the same.
4. If the target is a variable with the TARGET attribute, the pointer, after the pointer assignment statement is executed, is associated with that target.
5. If the target is a variable, function result, or defined operation result with the POINTER attribute, the pointer on the left is associated with the target of the pointer on the right if this pointer is associated with a target, and becomes disassociated or undefined if the pointer is disassociated or undefined.
6. If the pointer is a deferred-shape array, the target must be an array of the same rank. The pointer acquires the extents of the target, if the target has the TARGET attribute or if the target has the POINTER attribute and is associated with a target. As indicated above, the deferred-shape pointer may become disassociated or undefined if the target is a pointer that is disassociated or undefined.
7. The target may not be an assumed-size array, unless the subobject has a subscript or section subscript in the last dimension that specifies the upper bound. The target must not be an array section specified with a vector subscript.
8. See Dynamic Objects for additional ways in which the association status of a pointer is affected.

70 POINTER Attribute and Statement

A POINTER attribute or statement specifies that the named variables may be pointers to some target object. Pointers provide a capability for creating dynamic objects, such as dynamic-sized arrays and linked lists. An object with a pointer attribute initially has no space reserved for its target. A pointer is assigned space for its target when an ALLOCATE statement is executed or it is assigned to point to a target using a pointer assignment statement. A pointer may be thought of as a descriptor containing information about the target it is pointing to, including its location and other attributes such as shape.

Examples:

```
LOGICAL, POINTER, DIMENSION (:,:) :: XPTR
   . . .
ALLOCATE (XPTR(10,10), STAT = IR)

INTEGER P1, P2
TARGET I1
POINTER P1, P2          ! P1 and P2 are scalar and
   . . .                !    have the POINTER attribute.
P1 => I1                ! Pointer assignment statement
P2 => P1                ! The target of P2 is now I1 as well.

PROGRAM POINTER_EXAMPLE
   REAL, DIMENSION (:), POINTER :: PGO => NULL( ), PTAD
      . . .
   ALLOCATE (PTAD(0:N+20))
      . . .
   PGO => PTAD(10:N+1)
      . . .
END PROGRAM POINTER_EXAMPLE
```

Tip: An array with the POINTER attribute is a pointer to an array and not an array of pointers. To create an array of pointers, define a type consisting of a single pointer component and declare an array of this defined type.

Related Topics:

ALLOCATE and DEALLOCATE Statements
Data Initialization
Pointers
Pointer Nullification
TARGET Attribute and Statement

Related Intrinsics:

ASSOCIATED (POINTER, TARGET)
NULL (MOLD)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 2.4.6, 5.1.2.7, 5.2.7, C.2.1
Fortran 95 Handbook, 5.4.1
Fortran 95 Using F, 8.1.1

POINTER Attribute and Statement

Syntax:

A type declaration statement with the POINTER attribute is:

type , POINTER [, *attribute-list*] :: *entity-list*

A POINTER statement is:

POINTER [::] *object-name* [(*deferred-shape-spec-list*)] &
[, *object-name* [(*deferred-shape-spec-list*)]]...

Things To Know:

1. During program execution, a pointer may refer (or point) to an object that has the TARGET attribute or the pointer may be allocated space for its target. In a pointer assignment statement, a pointer may be assigned to point to the target of another pointer.
2. A pointer has both association and definition status. The association status may be tested by the ASSOCIATED intrinsic function.
3. The target may be a scalar or an array, but must have the same rank as the pointer.
4. An array that is a pointer must be a deferred-shape array. That is, the type and rank are declared but the extents are deferred until space is allocated for the target.
5. A pointer must be associated with a target to acquire a value or to be referenced for the target's value.
6. Unless initialized, a pointer has an undefined association status. It may be initialized by pointer assignment to the NULL intrinsic function. This gives it a defined status of disassociated, indicating that it has no target.

71 Pointer Nullification

Pointer nullification causes a pointer to have a disassociated status; that is, its status is defined but it does not point to a target. There are occasions when this is necessary; for example, when the pointer indicates the end of a linked list. There are two ways a pointer may be nullified in executable code: by use of a NULLIFY statement or by use of a pointer assignment statement referencing the NULL intrinsic function.

Examples:

```
REAL, TARGET :: VALUE, X       ! VALUE and X can be targets.
REAL, POINTER :: PT, PV, PX
   . . .
PT => VALUE                    ! Associate PT with VALUE.
NULLILFY (PV, PX)              ! Nullify PV and PX.
   . . .
IF (.NOT.ASSOCIATED(PX)) &     ! The ASSOCIATED intrinsic is valid
      PX=>X                    !   here if (and only if) PT
                               !   previously has been allocated,
                               !   associated or nullified
                               !   (as above).
PT => NULL()                   ! Nullify PT.
TYPE LIST_NODE                 ! Linked list type
    INTEGER VALUE
    TYPE (LIST_NODE), POINTER :: NEXT => NULL()
END TYPE LIST_NODE
TYPE (LIST_NODE), POINTER :: LIST
   . . .
ALLOCATE (LIST)                ! Create new list node.
LIST % VALUE = 28              ! Define the data field.
                               ! The NEXT field is initialized by default
```

Tip: Unless initialized, the initial status of a pointer is undefined. Referencing an undefined pointer in an expression is invalid, so it is a good idea to either initialize a pointer or nullify it. This is unnecessary only if the first executable occurrence of the pointer associates it with a target.

Related Topics:

ALLOCATE and DEALLOCATE Statements
Data Initialization
Defined Type: Default Initialization
Pointers
Pointer Association
POINTER Attribute and Statement
TARGET Attribute and Statement

Related Intrinsics:

ASSOCIATED (POINTER, TARGET)
NULL (MOLD)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.4.1, 5.1, 6.3.2, 7.5.2. 13.14.79
Fortran 95 Handbook, 4.4.1, 5.1, 6.5.2, 7.5.3, A.79
Fortran 95 Using F, 8.1.4

Syntax:

The NULLIFY statement is:

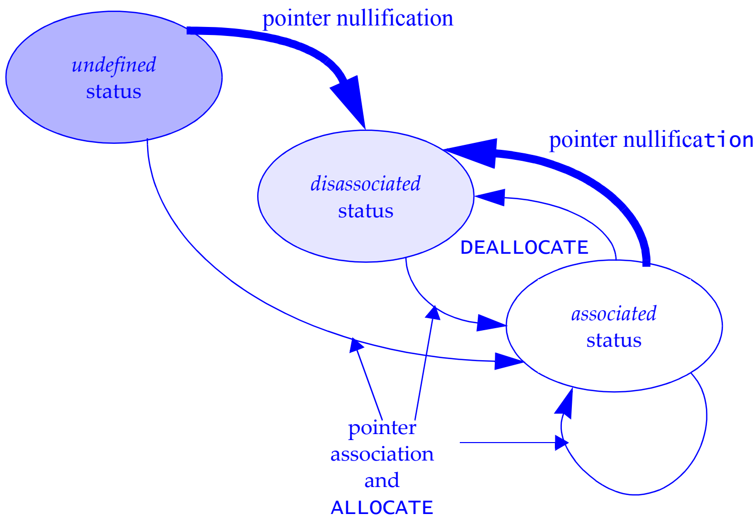
NULLIFY (*pointer-object-list*)

A pointer object is one of:

variable-name
structure-component

Things To Know:

1. All objects in the pointer object list must have the POINTER attribute.
2. Unless initialized, a pointer's status is undefined. Its status may be defined with a disassociated status using a NULLIFY statement or a pointer assignment statement referencing the NULL intrinsic functionL.
3. When a pointer is disassociated, its status is defined but the pointer does not point to anything.
4. If the status of a pointer is undefined, it may not be tested with the ASSOCIATED function.
5. The following diagram illustrates the three association status values that a pointer can have, *undefined*, *disassociated*, and *associated*, and the relationships among them. Note that pointer nullification is the only way to change the status from undefined to disassociated, and is one of two ways, along with the DEALLOCATE statement, to change the status from *associated* to *disassociated*..



72 Portable Precision Control

Fortran supports portable numerical precision and range control by using the kind parameter mechanism and a rich set of inquiry, model inquiry, and computational intrinsic functions.

One common need in writing portable numerical software is to write one program that uses 64-bit arithmetic on both 32-bit single precision machines and 64-bit single precision machines. Using the numerical precision control of Fortran, the same program can be written for both machines to select 64-bit arithmetic as follows.

Declare an integer variable Q using the following declaration:

```
INTEGER, PARAMETER :: Q = SELECTED_REAL_KIND( 10 )
```

All real and complex variables are declared with the kind parameter Q and all constants are specified with that kind value, as follows:

```
REAL(Q)   X, XC, XP             0.1_Q
                                -18.61E25_Q
COMPLEX(Q)  C, CX, CY           ( 0.0_Q, 1.0_Q )
```

The effect of these declarations is to select single precision for long word machines (machines that use 64 bits for single precision variables) and to select double precision for short word machines (machines that use 32 bits for single precision variables and 64 bits for double precision variables). No changes in the source code are necessary to port the program between short word and long word machines.

Computations involving these variables can be controlled using intrinsic functions that return values associated with their representation. For example, suppose we want to compute X*X but are concerned it may overflow. If X is small enough so that X*X does not overflow, we want the computation to proceed. And if it is so large that X*X will overflow, we wish to scale X so as to avoid overflow. Then the following code can be written to accomplish this safe programming:

```
IF( ABS(X) <= SQRT( HUGE(X) ) )  THEN
   . . . ! Perform the computation with X*X.
ELSE
   . . . ! Scale X appropriately using the SCALE
   . . . !   intrinsic function and then compute X*X.
END IF
```

Related Topics:

Complex Type and Constants
Data Representation Models
Integer Type and Constants
Intrinsic Functions: Inquiry and Model
Kind Parameters
Real Type and Constants

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.3.1, 5.1.1.1-4, 13, C.1.2
Fortran 95 Handbook, 4.3, 13.2, 13.3
Fortran 95 Using F, 1.2.11, 1.5.1, A.5.6, A.6

In another case, the intrinsic function EPSILON can be used to terminate an iteration in a portable way. Suppose XC is the current iterate and XP is the previous iterate. In some cases, it is appropriate to terminate the iteration when the difference in the last two iterates is small in a relative sense. Then the statements

```
IF( ABS( XC - XP ) <= 2.0_Q * ABS( XC ) *  EPSILON( XC )  )  THEN
   . . . ! Terminate the iteration.
ELSE
   . . . ! Continue the iteration; convergence not detected.
ENDIF
```

provide portable code that will execute properly no matter what reasonable data types are selected for XC and XP.

A second need is to write a program that uses single precision for production computation but with a minimum number of changes can be rerun using double precision to study the effect of precision on the computation. This can readily be accomplished using a kind parameter defined in a module and referencing the module to define all real or complex variables. For example:

```
MODULE  WORKING_PRECISION
   INTEGER, PARAMETER :: WP = KIND( 0.0 )
END MODULE  WORKING_PRECISION

PROGRAM  MY_PROG
   USE WORKING_PRECISION
      REAL(WP)  X, Y, Z
      COMPLEX(WP)  CX, CY, CZ
         . . . 1.62_WP . . .
END PROGRAM  MY_PROG
```

The program MY_PROG is written using single precision (KIND(0.0) selects the kind parameter value for single precision, the type of the argument for the intrinsic function KIND). To change the program so that it uses double precision, change the argument of the KIND intrinsic to 0.0D0, recompile the program, and it will execute using double precision computations.

Similarly, different integer types can be selected using the intrinsic function SELECTED_INT_KIND rather than the function SELECTED_REAL_KIND. For example, if your compiler supports 16 bit integers and your integer data never exceeds 10^4 in magnitude, the declaration

```
INTEGER( SELECTED_INT_KIND( 4 ) )   I, J, K(100000)
```

uses a “half” word (16 bits) to store I, J, and K, thereby typically reducing the storage for your integer data by one half.

73 Program Units

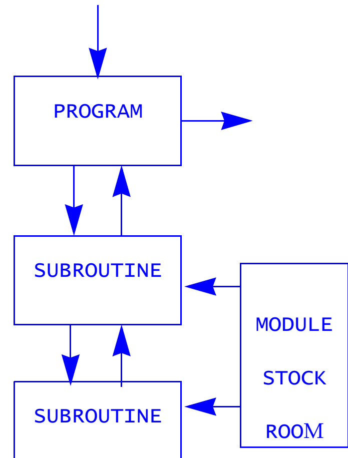
A Fortran program is a collection of program units. One and only one of these units must be a main program. The five kinds of program units are main program unit, external function subprogram unit, external subroutine subprogram unit, module program unit, and block data program unit.

Examples:

```
PROGRAM DRIVER             ! Main program unit
   . . .
   CALL MECHANIC (TUNEUP)
   . . .
END PROGRAM DRIVER

MODULE STOCK_ROOM          ! Module program unit
   . . .                   ! Specifies data shared between
                           !   subroutines MECHANIC and PARTS.
END MODULE STOCK_ROOM

SUBROUTINE MECHANIC (SERVICE)          ! External subprogram unit
   USE STOCK_ROOM
   . . .
   CALL PARTS (PLUGS, "CRX", 1993)
   . . .
END SUBROUTINE MECHANIC

SUBROUTINE PARTS (PART, MODEL, YEAR)  ! External subprogram unit
   USE STOCK_ROOM
   . . .
END SUBROUTINE PARTS
```

Related Topics:

Functions
Generic Procedures and Operators
Interfaces and Interface Blocks
Internal Procedures
Main Program
Modules
Module Procedures
Scope, Association, and Definition Overview
Subroutines



To Read More About It:

ISO 1539 : 1997, Fortran Standard, 11, C.8.1
Fortran 95 Handbook, 2.2.1, 11
Fortran 95 Using F, 3

Syntax:

A program unit is one of:

main-program
external-subprogram
module
block-data-program-unit

An external subprogram is one of:

function-subprogram
subroutine-subprogram

Things To Know:

1. The main program and procedure subprograms are executable. The nonexecutable program units are block data units and modules, which provide only definitions used by other program units.
2. Each program unit is an ordered set of constructs, statements, comments, and include lines. The heading statement identifies the kind of program unit it is, such as a subroutine or a module; it is optional in a main program. An END statement marks the end of the unit.
3. Program execution begins with the first executable statement in the main program. The main program is often used as a "driver" to control computations defined in other program units.
4. A module contains data declarations, defined-type definitions, procedure interfaces, common block declarations, namelist group declarations, and subprogram definitions used by other program units. It also specifies the accessibility (PUBLIC or PRIVATE) of these entities.
5. Block data program units are used only to specify initial values for variables in named common blocks. With the addition of modules to Fortran, block data program units are no longer needed for new programs because modules can provide global data initializations.
6. Main programs, external subprograms, and module subprograms may contain internal subprograms, which may be either subroutines or functions.
7. All program units, except block data, may contain procedure interface blocks. A procedure interface block is used to describe the interface of an external procedure—that is, the procedure name, the number of arguments, their types, attributes, names, and the type and attributes of a function result. It is also used to specify a generic name, assignment, or operator used to invoke a module procedure or external procedure. An interface is required in some cases and, in others, allows the processor to check the validity of a procedure reference.

74 PUBLIC and PRIVATE Attributes and Statements

PUBLIC and PRIVATE attributes in a type statement or in an accessibility statement determine the accessibility of entities such as variables, type definitions, functions, and named constants. These statements may appear only in the specification part of a module. The USE statement may restrict accessibility further. Accessibility statements may be used to control access to subroutines, generic specifiers, and namelist groups because these entities may not appear in a type declaration statement. Public entities in a module are accessible outside the module via use association.

Examples:

```
MODULE FOURIER
   PUBLIC                    ! PUBLIC unless explicitly PRIVATE
   COMPLEX, PRIVATE :: FFT   ! FFT is accessible only in module.

   TYPE (STRUCTURE_NAME), PRIVATE :: STRUCTURE_A, STRUCTURE_B

   PRIVATE  A, B, C          ! A, B, and C are accessible
                             !   only in the module.
   PUBLIC   R, S, T          ! R, S, and T are accessible
                             !   outside the module.
END MODULE FOURIER

MODULE PLACE
   PRIVATE                        ! Change default accessibility
   INTERFACE OPERATOR (.ST.)      !   to PRIVATE.
      MODULE PROCEDURE XST
         . . .
   END INTERFACE

   PUBLIC OPERATOR (.ST.)           ! This makes .ST. public;
   LOGICAL, DIMENSION (100) :: LT   ! everything else is private.
   CHARACTER(20) :: NAME
   INTEGER IX, IY
      . . .
END MODULE PLACE
```

Related Topics:

Defined Type: Definition
Dynamic Objects
Defined Type: Structure Component
Interfaces and Interface Blocks
Modules
USE Statement and Use Association

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.2, 5.2.3, 11.3.1, C.8.2.2
Fortran 95 Handbook, 5.6.1, 11.6.4
Fortran 95 Using F, 3.1.2

PUBLIC and PRIVATE Attributes and Statements

Syntax:

A type declaration statement with the accessibility attribute is one of:

type , PUBLIC [, *attribute-list*] :: *entity-list*
type , PRIVATE [, *attribute-list*] :: *entity-list*

A defined-type statement with an access attribute is one of:

TYPE , PUBLIC :: *type-name*
TYPE , PRIVATE :: *type-name*

An accessibility statement is one of:

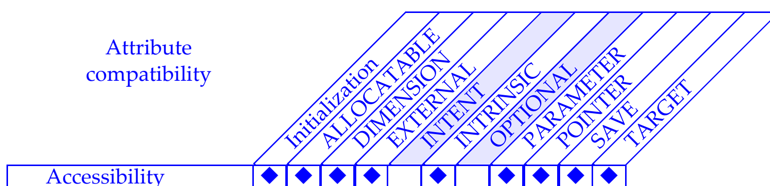
PUBLIC [[::] *access-id-list*]
PRIVATE [[::] *access-id-list*]

An access id is one of:

constant-name
variable-name
procedure-name
defined-type-name
namelist-group-name
OPERATOR (*operator*)
ASSIGNMENT (=)

Things To Know:

1. The PUBLIC attribute allows entities to be available outside the module via use association. The PRIVATE attribute limits access to within a module.
2. The default accessibility in a module is PUBLIC; it can be reaffirmed or changed to PRIVATE using an accessibility statement without a list. Only one PUBLIC or PRIVATE accessibility statement without a list is permitted in a module. See Defined Type: Definition for other uses of the PRIVATE statement.
3. Accessibility specifications for a generic name, operator, or assignment do not apply to any specific name unless the specific name is the same as the generic name.



75 Pure Procedures

A **pure procedure** is a subroutine or function that has the keyword PURE or ELEMENTAL in the procedure heading prefix. In addition, all of the intrinsic functions and the intrinsic subroutine MVBITS are pure. The keyword PURE indicates that the procedure has no side effects, such as changing the value of a global variable or opening a file. This information is used to permit optimizations that otherwise might not be done and to permit invocations of the procedure to appear in a FORALL construct or specification statement.

Examples:

```
PURE FUNCTION VIDA (COSTA, RICA) RESULT (PURA_VIDA)
   REAL :: PURA_VIDA
   REAL, INTENT (IN) :: COSTA, RICA
   PURA_VIDA = 2 * COS (COSTA) + RICA
END FUNCTION VIDA
```

Tip: Although side effects have always been allowed in Fortran functions, the result of using them is never completely predictable and inhibits optimization. Make all functions pure.

Related Topics:

Elemental Procedures
Functions
Internal Procedures
Intrinsic Function Overview
Module Procedures
Subroutines

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 12.6
Fortran 95 Handbook, 12.4
Fortran 95 Using F, 3.7

Syntax:

A pure function statement is:

PURE [RECURSIVE] [*type-spec*] &

FUNCTION *function-name* (*dummy-argument-name-list*) &

[RESULT (*result-name*)]

A pure subroutine statement is:

PURE [RECURSIVE] SUBROUTINE *subroutine-name* [([*dummy-argument-name-list*])]

Things To Know:

1. A dummy argument of a pure function must have intent IN unless it is a procedure or a pointer.
2. A pure subroutine may modify an OUT or INOUT argument.
3. A dummy argument of a pure subroutine must have its intent specified unless it is a procedure, pointer, or alternate return.
4. A procedure dummy argument of a pure procedure must be pure.
5. A local variable in a pure procedure must not have the SAVE attribute.
6. All internal procedures in a pure procedure must be pure.
7. A pure procedure must not contain any input/output statements: except READ or WRITE specifying an internal file.
8. A variable that is in common, is host associated, is use associated, has intent IN, or is storage associated with any such variable, must not be used in any of the following contexts:
   - on the left-hand side of an assignment statement
   - in a pointer assignment statement
   - as a DO variable or implied-DO variable
   - as an input item in a READ statement from an external file
   - as the internal file in a WRITE statement
   - as an IOSTAT variable in an I/O statement
   - as the object to be allocated or deallocated or as the STAT variable in an ALLOCATE or DEALLOCATE statement
   - in a pointer nullification statement
   - in the right-hand side of an assignment statement in which the left-hand side has a pointer component.
9. There are four situations in which a procedure must be pure; in these cases the procedure interface also must be explicit.
   - a function that is referenced in a FORALL construct or statement.
   - a function that is referenced in a specification statement
   - a procedure passed as an actual argument to a pure procedure
   - a procedure referenced by a pure procedure

76 READ/WRITE General Form

READ and WRITE statements transfer data to or from a file connected to either an internal or external unit. A unit may be processor dependent, as in the PRINT statement. A variety of data transfer facilities are available for processing collections of data in a file. These include data transfers that are formatted or unformatted sequential access. Other data transfers refer to data by record number (direct access). List-directed access transfers data without an explicit format specification. Namelist is another form where a name-directed technique with processor-dependent format specifications are used. Part of a record may be read or written by using nonadvancing transfer of data. This sometimes is called **stream input/output**.

In the forms using an I/O specification list, there must be a unit specifier. If a format specifier or a namelist specifier is second, the FMT= or NML= is optional. Only one of these may appear. If neither appears, the data transfer is unformatted.

An array without subscripts as an item in the list indicates that all elements of the array are transferred in array element order. An assumed-size array is prohibited in an input/output list.

If a structure appears in a formatted input/output list, it is as though all components appeared in the order of their declaration. If a structure appears in an unformatted list, the structure is one item, and the order of components is processor dependent.

A pointer in an input/output list must be associated with a target; the target is read or written.

Tip: DO variables in an input/output list of a READ or WRITE statement have the scope of the entire program unit containing the READ or WRITE statement. Therefore, the execution of a READ or WRITE statement with implied DO variables effects the values of program variables of the same names. Often, an easy way to avoid the use of such variables is to use arrays or array sections. It is preferable to use IOSTAT rather than the ERR, EOR or EOF. The END, EOR, and ERR specifiers use labels which, if avoided, reduce the opportunity for program errors and for writing unstructured programs.

Related Topics:

CLOSE Statement
Files and Records
INQUIRE Statement
OPEN Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.4, C.6.1.5
Fortran 95 Handbook, 9.2
Fortran 95 Using F, 9.3

Syntax:

A data transfer statement is one of:

```
READ ( io-control-spec-list ) [ input-item-list ]
READ format [ , input-item-list ]
WRITE ( io-control-spec-list ) [ output-item-list ]
PRINT format [ , output-item-list ]
```

A format is one of:

default-character-expression
*
label

The input/output control specifiers appear in the following table.

Specifier=		Values	Default	Description
UNIT=exp	Ix	Positive integer	Proc. dep.	External unit
UNIT=var	Ca	Character string	None	Internal unit
UNIT=*				Proc. dep. external unit
FMT=exp	C	Character string	None	Format specification
FMT=label	Lb,I	FORMAT statement label	None	FORMAT statement
FMT=*				List-directed formatting
NML=name	Nml			Namelist formatting
ADVANCE=exp	C	YES, NO	YES	(Non)Advancing I/O
END=label	Lb	Branch target stmt	None	Branch target taken on an end-of-file condition
EOR=label	Lb	Branch target stmt	None	Branch target taken on an end-of-record condition
ERR=label	Lb	Branch target stmt	None	Branch target taken on an error condition
IOSTAT=var	I	Positive	None	An error condition occurs
		Negative	None	An end-of-record or end-of-file condition occurs
		Zero	None	No error condition occurs
REC=exp	Ix	Positive	None	Record number
SIZE=exp	I	Positive	Proc. dep.	Record length

Lb, I, C — label, default integer, character default scalar expression; the character values are without regard to case and trailing blanks are ignored
Ix — integer scalar expression of any kind
Ca — scalar or array default character variable
Nml — namelist group name

77 READ/WRITE: Direct Access Formatted

Formatted direct access data transfer statements read or write a specific record to or from a formatted file. Records may be read or written in any order and are identified by record number using the REC= specifier. A file may not be connected for direct access and sequential access at the same time. A file must be closed and reconnected to change access methods. Direct access makes it possible to transfer records in an order that is most convenient for the program logic.

Examples:

```
WRITE (REC=RN, FMT=100, UNIT=9, IOSTAT=IS) SAVE, SAND
100 FORMAT (15F5.3)
! RN is the record number, and IS is set to a zero value,
!   if there is no error condition.

READ (8, 100, REC = 14, ERR = 999) SOFT, WARE
! Formatted record 14 is read using direct access from unit 8;
!   the program branches to statement 999 if there is an error.

IU = 7 * K
WRITE (IU, "(3E10.1)", REC = 20) X, Y, Z
! The format is a character string.
! No error condition options are used;
!   the program terminates on an error condition in this case.
```

Related Topics:

INQUIRE Statement
OPEN Statement
READ/WRITE: Direct Access Unformatted

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.1, 9.2.1.2.2, 9.4.1.3, 9.4.4.4.2
Fortran 95 Handbook, 9.1.1, 9.1.4.2, 9.2.6.1
Fortran 95 Using F, 9.1, 9.2.4, 9.3.8

Specifier Notes

[UNIT=]	Required; an external file
[FMT=]	Required; a format specifier
REC=	Required; a record number
IOSTAT=	Positive on an error, zero otherwise.
ERR=	Branch on error

Syntax:

A direct access formatted data transfer statement is one of:

```
READ ( [ UNIT = ] scalar-integer-expression   &
   , [ FMT = ] explicit-format   &
   , REC = scalar-integer-expression   &
   [ , IOSTAT = scalar-default-integer-variable ]   &
   [ , ERR = label ]   &
   ) [ input-item-list ]

WRITE ( [ UNIT = ] scalar-integer-expression   &
   , [ FMT = ] explicit-format   &
   , REC = scalar-integer-expression   &
   [ , IOSTAT = scalar-default-integer-variable ]   &
   [ , ERR = label ]   &
   ) [ output-item-list ]
```

An explicit format is one of:

default-character-expression
label

Things To Know:

1. The unit must be connected for direct access by using an OPEN statement with ACCESS= "DIRECT" and a record length specifier. If the connection is for formatted access, unformatted access is prohibited. If the connection is for unformatted access, formatted access is prohibited.
2. A record must exist before it can be read. Otherwise, records may be read or written in any order.
3. Records may be rewritten, but not deleted.
4. The record number is established when the record is written and cannot be changed.
5. All records in the same file have the same length which is specified by the RECL= specifier in an OPEN statement.
6. List-directed, namelist, and nonadvancing data transfer are prohibited.

78 READ/WRITE: Direct Access Unformatted

Unformatted direct access data transfer reads or writes a specified record to or from an unformatted file. Records may be read or written in any order and are identified by record number. A file may not be connected for direct access and sequential access at the same time. A file must be closed and reconnected to change access methods. Direct access makes it possible to transfer records in an order that is most convenient for the program logic.

Examples:

```
READ (9, REC = 12) MOUNTAIN, MAGIC
! Read values of MOUNTAIN and MAGIC from record 12, unit 9.

WRITE (ERR = 99, UNIT = 12, REC = 3) PRESSURE
! Writes the value of PRESSURE into record 3.

READ (11, REC = I, IOSTAT = IR, ERR = 99) TEMP_CHANGES
! Read the value of TEMP_CHANGES from record I,  If an
!   error condition occurs, set IR to a positive value
!   and branch to statement 99.

TYPE COAST                 ! Defined-type declaration declares
  INTEGER :: SAND (100)  !   a type called COAST.
                           ! COAST contains SAND and COASTLINE.
  CHARACTER (LEN = 20) :: COASTLINE (100)
END TYPE COAST
   . . .
TYPE (COAST) :: X(100)    ! X is an array of type COAST.
   . . .                   ! Each element of X is a structure.
WRITE (16, REC = N, IOSTAT = IS)  X(J)    ! Write the Jth element
N = N + 1              ! of X as a structure containing 100 integers
   . . .               ! and 100 character strings in one record.
```

Related Topics:

INQUIRE Statement
OPEN Statement
READ/WRITE: Direct Access Formatted

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.1, 9.2.1.2.2, 9.4.1.3, 9.4.4.4.1
Fortran 95 Handbook, 9.1.1, 9.1.4.2, 9.2.6.2
Fortran 95 Using F, 9.1, 9.2.4, 9.3.7, 9.3.8

Specifier Notes

[UNIT=]	Required; an external file
REC=	Required; a record number
IOSTAT=	Positive on an error, zero otherwise.
ERR=	Branch on error

Syntax:

A direct access unformatted data transfer statement is one of:

```
READ ( [ UNIT = ] scalar-integer-expression &
   , REC = scalar-integer-expression &
   [ , IOSTAT = scalar-default-integer-variable ] &
   [ , ERR = label ] &
   ) [ input-item-list ]

WRITE ( [ UNIT = ] scalar-integer-expression &
   , REC = scalar-integer-expression &
   [ , IOSTAT = scalar-default-integer-variable ] &
   [ , ERR = label ] &
   ) [ output-item-list ]
```

Things To Know:

1. The unit must be connected for direct access unformatted data transfer by using an OPEN statement. If a unit is connected for unformatted direct access, formatted data transfer is prohibited. If a unit is connected for formatted direct access, unformatted data transfer is prohibited.
2. A record must exist before it can be read. Otherwise, records may be read in any order.
3. Records may be rewritten, but not deleted. For example, record 10 may be read or written before record 4 or 8.
4. All records in the file have the same length, which is specified by the RECL= specifier in an OPEN statement.
5. List-directed, namelist, and nonadvancing access are prohibited.

79 READ/WRITE: Internal Files

An internal file is a default character variable used as the unit in a formatted (including list-directed) sequential access data transfer statement. Using an I/O statement, data is transferred to/from a variable in memory to/from a character variable (internal file) in a form specified by the format (or using list-directed format). Any character variable read from or written to an internal file must be of type default character. An internal file may be used to convert data values to character strings and vice versa.

Examples:

```
! NAME_FIELD is an internal file.
CHARACTER * 80 :: NAME_FIELD, CHAR_VAR

! If the array NAME_ARRAY has more than 16 elements,
!   an end-of-file condition will occur and the statement
!   labeled 98 will be executed next.
READ (NAME_FIELD, 100, ERR=99, END=98) NAME_ARRAY
100 FORMAT (16I5)

! The WRITE statement transfers data from PRESSURE to the
!   internal file  CHAR_VAR using list-directed formatting.
! It is assumed that no more than 80 characters will be
!   transferred; if the assumption is invalid, the statement
!   is invalid.
WRITE (FMT=*, UNIT=CHAR_VAR, IOSTAT=IER) PRESSURE
```

Related Topics:

Files and Records
READ/WRITE: List-directed
READ/WRITE: Sequential Formatted Advancing

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.2.2
Fortran 95 Handbook, 9.1.2, 9.2.9
Fortran 95 Using F, 9.2.2

Specifier Notes

[UNIT=]	Required; an internal file
[FMT=]	Required; a format specifier
IOSTAT=	Positive on an error, negative on end of file, zero otherwise
ERR=	branch on error
END=	branch on end of file on input

Syntax:

A data transfer statement using an internal file is one of:

```
READ ( [ UNIT = ] default-character-variable  &
       , [ FMT = ] explicit-format  &
       [ , IOSTAT = scalar-default-integer-variable ]  &
       [ , ERR = label ]  &
       [ , END = label ]  &
       ) [ input-item-list ]

WRITE ( [ UNIT = ] default-character-variable  &
       , [ FMT = ] explicit-format  &
       [ , IOSTAT = scalar-default-integer-variable ]  &
       [ , ERR = label ]  &
       ) [ output-item-list ]
```

An explicit format is one of:

```
default-character-expression
*
label
```

Things To Know:

1. A formatted sequential access data transfer statement (including list-directed formatting) may use an internal file. Namelist is prohibited.
2. An internal file must not be accessed directly.
3. If the number of characters in the output item list is less than the length of the record, the remaining characters are set to blank.
4. The character variable used as an internal file must be of default character type. If the character variable is an array with section subscripts, no section subscript may be a vector subscript.
5. An internal file is always positioned ahead of the current record prior to data transfer. Data are transferred with editing. Any remaining characters in the record are blank filled on writing.
6. Both intrinsic and defined type objects are allowed in the input/output item list when an internal file is used.
7. If the character variable representing the unit specifier is a scalar, there is only one record. If it is an array, each element of the array is a record of the file, all of the same length. The order of the records is array element order. Thus the length of the record is the number of characters declared or assumed (assumed-shape only) for the character variable.
8. An end-of-file condition is created on input if there is an attempt to read beyond the end of the scalar variable or beyond the last element of the array representing the internal file.

READ/WRITE: List-directed

List-directed data transfer statements read or write values to and from an internal or external file using a format selected by the processor appropriate for the value being transferred. For example, the format selected to transfer a 10 digit integer must be large enough to transfer the 10 digits and blanks to separate it from the previous item transferred. The data transfer is from and to files or units connected for formatted sequential access.

Examples:

```
REAL X(5)
IVV = 10
READ (IVV,*) ( X (I), I = 1, 5 )        ! Read five values of X.

! Input data:      comma separated
! 1.0, 2.0, 2*3.0, 8.0

READ (5,*,END=99) N, J, A, B            ! N and J are integer.
                                        ! A and B are real.
! Input data:      blank separated
10 7 2.3 2.9

REAL ITEMP(3)
READ (UNIT=4, FMT = *) &   ! Two null values read
      ( ITEMP(K), K=1,3)   ! ITEMP(3) = 5124
                           ! ITEMP(1), ITEMP(2) are unchanged.
! Input data:
! , , 5124

REAL BOOK
CHARACTER*20, LINE         ! LINE is an internal file.
BOOK = 241.5
WRITE (LINE, FMT = *, IOSTAT = IR) BOOK
! The form of the output in LINE depends
!   on the value of BOOK and the processor.
```

Related Topics:

Format Specifications
READ/WRITE General Form

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.4.1.1, 10.8, C.7.2
Fortran 95 Handbook, 9.2.7, 10.10
Fortran 95 Using F, 1.1.3, 9.3.1, 9.8.16

Specifier Notes

[UNIT=]	Required; an internal or external file
[FMT=]	Required; * for list directed
IOSTAT=	Positive on an error, negative on an end of file, zero otherwise
END=	Branch on end of file on input
ERR=	Branch on error

Syntax:

A list-directed data transfer statement is one of:

```
READ ( [ UNIT = ] io-unit &
       , [ FMT = ] * &
       [ , IOSTAT = scalar-default-integer-variable ] &
       [ , END = label ] &
       [ , ERR = label ] &
       ) [ input-item-list ]
READ * [ , input-item-list ]

WRITE ( [ UNIT = ] io-unit &
       , [ FMT = ] * &
       [ , IOSTAT = scalar-default-integer-variable ] &
       [ , ERR = label ] &
       ) [ output-item-list ]
PRINT * [ , output-item-list ]
```

An io-unit is one of:

```
scalar-integer-expression
*
default-character-variable
```

Things To Know:

1. An expression with operators or functions must not appear in the input data when reading. A constant must not be a binary, octal or hexadecimal constant.
2. The value separators are commas, slashes, or blanks. Blanks may precede or follow commas and slashes and are considered one value separator. A null value is read when there is no value between two value separators. A slash used as a value separator completes the input data transfer statement; all unread items remain unchanged.
3. The input record may include null values or values of the form *c*, *r*c*, or *r** where *r* is a digit string and *c* is a literal constant with no embedded blanks (readable by an I, F, A, or L edit descriptor). *r** means *r* successive appearances of the null character. Blanks are never zeros.
4. A real or integer datum must be in a form suitable for an F or I edit descriptor, respectively. Complex data is in the form of the left and right parentheses enclosing a pair of numbers separated by a comma suitable for reading by an F edit descriptor.
5. Logical data on input consists of an optional period (.) followed by T or F followed by any characters except value separators. On output, the field consists of leading blanks followed by T or F.
6. Character data consists of a consecutive sequence of characters that are not value separators, quotes, or double quotes.

81 READ/WRITE: Namelist

A list of variables intended for input or output are given a name called a namelist group name. Data can then be transferred by inserting the namelist name in a READ/ WRITE statement. No list of items is allowed in the READ/WRITE statement. The exact formatting for output is determined by the processor, but examples are given below.

Examples:

```
NAMELIST /THUNDER/ BRIGHT, ELECTRIC, XY
READ (*, THUNDER, END = 99)          ! Uses default input unit.
```

The input might be:

```
&THUNDER BRIGHT = 24.35/             ! Only BRIGHT is changed.

WRITE (UNIT=6, NML=THUNDER, ERR=98) ! All variables are output.
```

The output might be:

```
&THUNDER BRIGHT = 24.35, ELECTRIC = "STORM", XY = 1000./

PROGRAM CLOUD_COVER
NAMELIST /CLOUDS/ DARK, LIGHT, CUMULUS
   . . .
READ (NML = CLOUDS, ERR = 97, UNIT = 11)
WRITE (10, NML = CLOUDS, IOSTAT = IS)
END
```

The input might be:

```
&CLOUDS DARK = 4, ! This is a comment.
CUMULUS = 8.3/
```

Tip: This feature has several restrictions when used with other Fortran features. In certain cases, it offers some advantages. For example, a significant use of this feature is to read in a few input values from among many that might potentially change (but only a few do) from their current value.

Related Topics:

CLOSE Statement
Files and Records
INQUIRE Statement
OPEN Statement
READ/WRITE General Form
READ/WRITE: List-directed

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.4, 9.4.1.2, 10.9
Fortran 95 Handbook, 5.10, 9.2.2.1, 9.2.8, 10.11

Specifier Notes	
[UNIT=]	Required; an external file
[NML=]	Required; namelist group name
IOSTAT=	Positive on an error, negative on an end of file, zero otherwise
END=	Branch on end of file on input
ERR=	Branch on error

Syntax:

A namelist group declaration is:

```
NAMELIST / namelist-group-name / variable-name-list
```

A namelist data transfer statement is one of:

```
READ ( [ UNIT = ] external-io-unit  &
   , [ NML = ] namelist-group-name  &
   [ , IOSTAT = scalar-default-integer-variable ]  &
   [ , END = label ]  &
   [ , ERR = label ]  &
   )

WRITE ( [ UNIT = ] external-io-unit  &
   , [ NML = ] namelist-group-name  &
   [ , IOSTAT = scalar-default-integer-variable ]  &
   [ , ERR = label ]  &
   )
```

An external-io-unit is one of:

```
scalar-integer-expression
*
```

Things To Know:

1. Note that a format specification must not appear in a namelist input/output statement. Direct access and nonadvancing input/output must not be used. Unformatted data transfer is prohibited, as is transfer to and from an internal file.
2. The form of a namelist input record is an ampersand followed by the namelist group name followed by name-value pairs separated by value separators and terminated by a slash.
3. Namelist output is the same as namelist input, except for the form of logical and real constants. Namelist output formatting is processor dependent and is based on the type and values of the variables in the namelist group.
4. Namelist comments are permitted only in namelist input at the beginning of a line or after a value separator. Comments start with ! and continue to the end of a line.
5. The form of a name-value pair is:

   ```
   variable = value
   ```

   where value has a form described in item 3 of the topic READ/WRITE: List-directed.
6. Data in the group may be of intrinsic or defined type. The name-value pairs are processed in the order of appearance.
7. The value in the pair must be acceptable to a format specification for data of that type. A name-value pair may be omitted.

82 READ/WRITE: Sequential Formatted Advancing

The sequential access formatted advancing READ/WRITE statement transfers data under format control beginning at the first record. The transfer is in sequential order. The position of the file following an advancing data transfer is after the last record read or written. The file consists of characters determined by the format specification. A data transfer is sequential if the records are read in the order of their appearance on the external device. Internal files are allowed for this form of input/output, provided the ADVANCE= specifier is not present. (See READ/WRITE: Internal Files.)

Examples:

```
READ (5,100) X, (Y(I), I = 1,N) ! The default is advancing.

IU = 7
READ (IU,"(10F10.2)", ADVANCE = "YES" ) XLT

PRINT 101,  PAPERS               ! Default output unit is used.

WRITE (*, "(A, 5F3.1)" ) "G = ", G
! The format is a character string.
```

Tip: It is preferable to use IOSTAT rather than ERR or END. The END and ERR specifiers require labels, which increase the opportunity for program errors and for writing unstructured programs.

Related Topics:

Edit Descriptors: Control
Edit Descriptors: Data and Character String
READ/WRITE General Form
READ/WRITE: Internal Files

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.2.1.2.1, 9.2.1.3.1, 9.4
Fortran 95 Handbook, 9.1.4, 9.2.3
Fortran 95 Using F, 9.1, 9.2.6, 9.3.4

Specifier Notes

[UNIT=]	Required
[FMT=]	Required; a format specifier
IOSTAT=	Positive on an error, negative on an end of file, zero otherwise
END=	Branch on end of file on input
ERR=	Branch on error
ADVANCE="YES"	

Syntax:

A sequential access advancing formatted data transfer statement is one of:

```
READ ( [ UNIT = ] io-unit &
    , [ FMT = ] format &
    [ , IOSTAT = scalar-default-integer-variable ] &
    [ , END = label ] &
    [ , ERR = label ] &
    [ , ADVANCE = 'YES' ] &
    ) [ input-item-list ]

READ format [ , input-item-list ]

WRITE ( [ UNIT = ] io-unit &
    , [ FMT = ] format &
    [ , IOSTAT = scalar-default-integer-variable ] &
    [ , ERR = label ] &
    [ , ADVANCE = 'YES' ] &
    ) [ output-item-list ]
PRINT format [ , output-item-list ]
```

An input/output unit is one of:

```
scalar-integer-expression
*
default-character-variable
```

A format is one of:

```
default-character-expression
*
label
```

Things To Know:

1. The data transfer takes place with editing determined by the format specification.
2. Records are processed in the order they appear in the file.
3. In cases where a nonadvancing data transfer is followed by an advancing data transfer and the position of the file was in the middle of the record, the advancing input/output starts at the very next character.
4. Execution terminates when:
   - no items remain in the list,
   - on input, an end-of-file condition occurs, or
   - an error condition is encountered.

83 READ/WRITE: Sequential Formatted Nonadvancing

The sequential access formatted nonadvancing READ and WRITE statements transfer data in sequential order under format control. The file position is after the last character read or written and does not advance (nonadvancing) to the beginning of the next record. It is sometimes called stream input/output. Nonadvancing data transfer is character oriented and provides the capability of reading or writing parts of a record and not skipping to the end of a record as in advancing input/output. Varying length character strings can be read with this method. It is very con-venient to use in setting up a prompt for data input from a terminal.

Examples:

```
READ (*, "(A)", ADVANCE = "NO", ERR = 99) WHITE_CHARS

IUNIT = 7
WRITE (UNIT = IUNIT, FMT = '(A)', ADVANCE = 'NO') BLUE_CHARS

! The number of characters read is returned in the variable ICH.
READ (5, 101, ADVANCE = 'NO', SIZE = ICH, EOR = 97) RED_CHARS

PROGRAM COUNT_CHARACTERS
   INTEGER, PARAMETER :: EOR = -2, EOF = -1
   CHARACTER (1) :: C
   INTEGER :: COUNT = 0
   DO
      READ (*,"(A)", ADVANCE = "NO", IOSTAT = IOS) C
      IF (IOS .EQ. EOR) THEN; CYCLE
      ELSE IF (IOS == EOF) THEN; EXIT
      ELSE; COUNT = COUNT + 1; END IF
   END DO    ! Then print the character count.
END PROGRAM COUNT_CHARACTERS
```

Tip: IOSTAT is preferable to ERR, EOR, or END. The END, ERR, and EOR specifiers use labels, which increase the opportunity for program errors and for writing unstructured programs.

Specifier Notes

Specifier	Note
[UNIT=]	Required; an external file
[FMT=]	Required; explicit format
ADVANCE='NO'	
SIZE=	The number of characters read
IOSTAT=	Positive on an error, negative on an end of file or end of record, zero otherwise
END=	Branch on end of file on input
EOR=	Branch on end of record on input
ERR=	Branch on error

Related Topics:

Edit Descriptors: Control
Edit Descriptors: Data and Character String
READ/WRITE General Form

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.1.1, 9.4.1.8, 9.4.4, C.6.1.5
Fortran 95 Handbook, 9.1.3, 9.1.4.1, 9.2.5, 10.4
Fortran 95 Using F, 9.2.6, 9.3.5

READ/WRITE: Sequential Formatted Nonadvancing

Syntax:

A sequential nonadvancing formatted data transfer statement is one of:

```
READ ( [ UNIT = ] external-io-unit   &
    , [ FMT = ] explicit-format   &
    , ADVANCE = 'NO'   &
    [ , SIZE = scalar-default-integer-variable ]   &
    [ , IOSTAT = scalar-default-integer-variable ]   &
    [ , END = label ]   &
    [ , EOR = label ]   &
    [ , ERR = label ]   &
    ) [ input-item-list ]

WRITE ( [ UNIT = ] external-io-unit   &
    , [ FMT = ] explicit-format   &
    , ADVANCE = 'NO'   &
    [ , IOSTAT = scalar-default-integer-variable ]   &
    [ , ERR = label ]   &
    ) [ output-item-list ]
```

An external input/output unit is one of:

```
*      scalar-integer-expression
```

An explicit format is one of:

```
default-character-expression
label
```

Things To Know:

1. Data is transferred in a "stream" of characters. The files are formatted external files connected for sequential access.
2. The data transfers may be of varying length and are performed with explicit formatting. The unit may be * meaning some processor-dependent preconnected external unit using formatted sequential access.
3. The format may not be an asterisk (*) for list-directed formatting.
4. On output, execution of the WRITE statement terminates when there are no more items in the list and a data edit descriptor in the format is encountered, or when an error condition occurs.
5. On input, execution of the READ statement terminates when there are no more items in the list, or when an end-of-record, an end-of-file, or an error condition occurs if there are further items in the input list.
6. The negative value for an end-of-file condition is different from the negative value for an end-of-record condition.

84 READ/WRITE: Sequential Unformatted

A sequential access unformatted READ/WRITE statement transfers data in sequential order. The representation of data values in a record is processor dependent. Unformatted sequential data transfer has never been very portable, but it is probably the most efficient way to store data and recover it in subsequent executions of a program, because the transfer takes place without format conversion.

Examples:

```
READ (5, IOSTAT = IR) X, Y        ! IR is the status specifier.

WRITE (UNIT=9,IOSTAT=ES) A, B, C! The value of ES may be tested.

JU = 10 * ITT
READ (JU, END = 99, ERR = 9) XTE ! Read to an end of file.

WRITE (IOSTAT=IR,UNIT=11) X      ! UNIT= is required.
```

Tip: It is preferable to use IOSTAT rather than the ERR or END. The END and ERR specifiers use labels, which increase the opportunity for program errors and for writing unstructured programs.

Related Topics:

CLOSE Statement
INQUIRE Statement
OPEN Statement
READ/WRITE General Form

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 9.1.2, 9.4
Fortran 95 Handbook, 9.1.4.1, 9.2.4
Fortran 95 Using F, 9.2.4, 9.3.7

Specifier Notes

[UNIT=]	Required; external file
IOSTAT=	Positive on an error, negative on an end of file, zero otherwise
END=	Branch on end of file on input
ERR=	Branch on error

Syntax:

A sequential access unformatted data transfer statement is one of:

```
READ ( [ UNIT = ] scalar-integer-expression   &
    [ , IOSTAT = scalar-default-integer-variable ]   &
    [ , END = label ]   &
    [ , ERR = label ]   &
    ) [ input-item-list ]

WRITE ( [ UNIT = ] scalar-integer-expression   &
    [ , IOSTAT = scalar-default-integer-variable ]   &
    [ , ERR = label ]   &
    ) [ output-item-list ]
```

Things To Know:

1. Exactly one record is read or written without editing. The record is either a data record or an end-of-file record. The data in the records may be of intrinsic or defined type.
2. There is a correspondence of type and kind parameters between the value in the record and the input list item. If a value is complex, it may be represented by two reals. Character values must have the same length as well as kind type parameters.
3. The file consists of values in machine representation which is typically close to or the same as the representation in computer memory.
4. No format or namelist specifier is allowed.
5. Execution terminates if the list is exhausted or an error condition occurs. On input, execution terminates if an end-of-file record is encountered.

85 Real Type and Constants

The Fortran real type is used for data that approximate the mathematical real numbers. There are at least two kinds of real numbers—single precision (default real) and double precision real. A kind type parameter may be used to select a representation method. A processor may provide additional approximation methods that can be specified explicitly with a kind parameter. There is a real number representation model with related inquiry functions pertaining to each method.

Examples:

```
! X and Y are declared to be of real type.
REAL X, Y

! R and S are arrays that have a kind parameter HIGH.
REAL (KIND = HIGH) :: R(10,10), S(15:35,2)

! Z has at least 6 decimal digits and a decimal exponent range
!   between 10-32 and 10+32
REAL (SELECTED_REAL_KIND (6, 32)) Z

! A and B are double precision real arrays.
DOUBLE PRECISION, DIMENSION (100) :: A, B
```

Examples of real constants are:

42.96E-03	a real constant
48.33333333333333	a real constant
2.6D2	a double precision constant
10E5_LOW	a real constant with kind parameter LOW

Related Topics:

Data Representation Models
Expressions
Implicit Typing
Intrinsic Functions: Computation
Intrinsic Functions: Conversion
Intrinsic Functions: Inquiry and Model

Related Intrinsics:

KIND (X)
PRECISION (X)
RANGE (X)
REAL (A, KIND)
SELECTED_REAL_KIND (P, R)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.3.1.2, 5.1.1.2, 13.5, 13.7, C.1.1-2
Fortran 95 Handbook, 4.3.2, 5.1.2-3. 13.2.3, 13.3
Fortran 95 Using F, 1.2.2, 1.3.1, A.6.1-3

Syntax:

A real type declaration statement is one of:

REAL [([KIND =] *kind-parameter*)] [, *attribute-list* ::] *entity-list*
DOUBLE PRECISION [, *attribute-list* ::] *entity-list*

A real constant is one of:

[*sign*] *digit-string* *exponent-letter* *exponent* [_ *kind-parameter*]
[*sign*] *whole-part* **.** [*fraction-part*] [*exponent-letter* *exponent*] &
[_ *kind-parameter*]
[*sign*] **.** *fraction-part* [*exponent-letter* *exponent*] [_ *kind-parameter*]

An exponent letter is one of:

E
D

An exponent is:

[*sign*] *digit-string*

Things To Know:

1. Arithmetic operators that may have real operands are +, –, *, /, **, unary +, unary –. The relational operators <, <=, ==, /=, >, >=, .LT., .LE., .EQ., .NE., .GT., and .GE. may be used for comparisons; they yield default logical values.
2. At least two approximation methods, one for default real type and one for double precision real type, must be available. Double precision real has more precision than default real (single precision). A processor may provide additional representation methods that may be declared using an explicit kind parameter. The values of the kind parameters are processor dependent.
3. If both a kind parameter and an exponent letter are present, the exponent letter must be E.
4. More decimal digits may be written in a real literal constant than a processor can represent.
5. In storage association contexts, a default real variable occupies one numeric storage unit; a double precision variable occupies two numeric storage units. Other kinds of real, if provided, have no specified storage units.
6. A kind parameter used in a program must correspond to a processor approximation method. If a particular kind value is not supported, the processor must be able to indicate this invalid use.

86 Recursion

Fortran subroutines and functions may be recursive. Such procedures may call themselves, either directly or indirectly. Recursion is often a natural and powerful way of expressing and defining a computation.

Examples:

```
! This example computes the factorial function
!   whose recursive definition is given by:
!      0! = 1
!      n! = n * (n - 1)!  for n > 0

RECURSIVE FUNCTION FACTORIAL (N)  RESULT (FACTORIAL_RESULT)

   INTEGER, INTENT (IN) :: N
   INTEGER :: FACTORIAL_RESULT

   IF (N <= 0) THEN
      FACTORIAL_RESULT = 1
   ELSE
      FACTORIAL_RESULT = N * FACTORIAL (N - 1)
   END IF

END FUNCTION FACTORIAL

! In practice, factorial is more efficiently written
!  iteratively and is given above only as a concise and
!  compact illustration of recursion.
```

Tip: Recursion is particularly useful for expressing operations on linked lists that cannot easily be expressed iteratively.

Related Topics:

Functions
Internal Procedures
Module Procedures
SAVE Attribute and Statement
Subroutines

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 12.5.2.2-5
Fortran 95 Handbook, 2.4.3, 11.3, 12.1.3, 12.2-3
Fortran 95 Using F, 3.15, 3.16, 5.3, 7.7.7, 8.3, 8.4

Syntax:

A recursive function statement is:

RECURSIVE [*type-spec*] FUNCTION *function-name* &
([*dummy-argument-list*]) [RESULT (*result-name*)]

A recursive subroutine statement is:

RECURSIVE SUBROUTINE *subroutine-name* &
[([*dummy-argument-list*])]

Things To Know:

1. A procedure involved in either direct or indirect recursion must have the keyword RECURSIVE in the FUNCTION or SUBROUTINE statement of the procedure definition. The RECURSIVE keyword may help the implementation with the optimization of procedure calls.
2. For a recursive procedure, local variables that are not explicitly data initialized or saved are automatic; that is, there is a different copy of each such variable for each execution of the recursive procedure. On the other hand, local variables that are data initialized or saved are static; that is, there is only one copy for all executions of the recursive procedure. This implies that, if such a variable is given a value, that value is the same for all (recursive) instances of the program, until it is changed in which case the changed value is the same for all instances of the program.
3. The RESULT clause is required in a function that calls itself directly; otherwise, in general, there would be no way to distinguish the use of the function name as a result variable from its use in a recursive call, as is done in the second assignment statement of the recursive function FACTORIAL.

```
RECURSIVE FUNCTION REVERSE (PHRASE) RESULT (FLIPPED)
   CHARACTER (*), INTENT (IN) :: PHRASE
   CHARACTER (LEN(PHRASE)) :: FLIPPED
   INTEGER :: K, N
   INTRINSIC :: LEN, LEN_TRIM, INDEX

   K = LEN_TRIM (PHRASE)
   N = INDEX (PHRASE(1:K), " ", BACK=.TRUE.)
   IF (N == 0) THEN
      FLIPPED = PHRASE
   ELSE
      FLIPPED = PHRASE(N+1:K) // " " // REVERSE(PHRASE(1:N-1))
   END IF
END FUNCTION REVERSE
```

87 SAVE Attribute and Statement

A variable with the SAVE attribute retains its value and definition, association, and allocation status on exit from a procedure. All variables accessible to a main program are saved implicitly. An entire common block may be saved in order to maintain the integrity of the storage when none of the procedures using the common block are active. Similarly, saving a variable in a module preserves its value when no procedure using the module is active.

Examples:

```
MODULE FLOWERS
REAL, SAVE, ALLOCATABLE :: FOLIAGE(:) ! FOLIAGE is real type and
   . . .                              !  has the SAVE attribute.
END MODULE FLOWERS

SAVE A, B, TEMP, /BLOCKXY/           ! A common block BLOCKXY
                                     !   has the SAVE attribute.

RECURSIVE SUBROUTINE ATLATL (X, Y)
  INTEGER :: COUNT = 0               ! COUNT is saved
      . . .                          !   automatically.
  COUNT = COUNT + 1
      . . .
  CALL ATLATL (X, Y)
      . . .
END SUBROUTINE ATLATL

SUBROUTINE DAISY
   SAVE                              ! This saves everything.
      . . .
END SUBROUTINE DAISY
```

Tip: Even though many early implementations of Fortran saved all variables and named common blocks, a standard-conforming program may not rely on this. Modern systems are more complex and more attention should be paid to variables that must retain their value. Unless the SAVE attribute has been declared, a variable might not be saved. For the sake of portability, the SAVE attribute should always be declared for variables that need to retain their value.

Related Topics:

Data Initialization
Recursion

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.5, 5.2.4 , 5.2.10, 12.5.2.4
Fortran 95 Handbook, 5.5.1, 5.6.4, 12.1.3
Fortran 95 Using F, 3.1.2

Syntax:

A type declaration statement with the SAVE attribute is:

type , SAVE [, *attribute-list*] :: *entity-list*

A SAVE statement is:

SAVE [[::] *saved-entity-list*]

A saved entity is one of:

data-object-name
/ *common-block-name* /

Things To Know:

1. If the list in a SAVE statement is omitted in a scoping unit, everything in that scoping unit that can be saved is saved. No other explicit occurrences of the SAVE attribute or SAVE statement are allowed.
2. A variable in a common block must not be saved individually. If a common block is saved in one program unit, it must be saved everywhere it appears other than in a main program.
3. A SAVE statement in a main program has no effect because all variables and common blocks are saved implicitly in a main program.
4. There is only one copy of saved variables in all activations in a recursive procedure. If a local variable is not saved, there is a different copy for each activation.
5. Initialization in a DATA statement or in a type declaration implies that a variable has the SAVE attribute, unless the variable is in a named common block in a block data subprogram. Default initialization does not cause a variable to be saved.
6. The SAVE attribute may be declared in the specification part of a module. A variable in a module that is not saved becomes undefined when the module is not being used by any active program unit.

88 Scope, Association, and Definition Overview

Scope, association, and definition are the glue that binds statements and program units of Fortran into an executing program. Scope specifies the portion of a program where an entity is known or is accessible by an identifier. Association is the pathway by which an entity communicates with another entity in the same or a different scope. Definition, and its opposite undefinition, characterize the ways entities attain, retain, and lose their values through actions and associations in an executable program.

Examples:

```
SUBROUTINE GET(I, J)              ! I and J are local names.
   COMMON  /BUFFER/ X, Y          ! GET and BUFFER are global.
      . . .                       ! X and Y are local names.
   READ(5, *)  X, Y               ! X and Y defined
END

MODULE STACK_DATABASE                        ! STACK_DATABASE is global.
   TYPE STACK_TYPE
      INTEGER TOP; REAL, POINTER :: PTR(:)
   END TYPE STACK_TYPE
     . . .
   CONTAINS
      SUBROUTINE  CREATE( STACK )            ! CREATE is local.
         TYPE(STACK_TYPE)  :: STACK

         ALLOCATE  ( STACK % PTR(1000) )  ! Component allocated, but 
undefined
         STACK % TOP = 0                     ! Component defined
      END SUBROUTINE CREATE
        . . .
END MODULE STACK_DATABASE

PROGRAM MAIN                                 ! MAIN is global.
   INTEGER A, B
   COMMON /BUFFER/ T(2)                      ! T is local but BUFFER is
      . . .                                  !   global; thus T is storage
   CALL GET(A, B)                            !   associated with X and Y
      . . .                                  !   and T becomes defined.
END
```

Related Topics:

Argument Association
COMMON Statement
EQUIVALENCE Statement
Host Association
Program Units
Storage Association
USE Statement and Use Association

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 14
Fortran 95 Handbook, 2.1, 2.2, 2.4.2, 14
Fortran 95 Using F, 3.11

Things To Know:

In the example, the names BUFFER, GET, STACK_DATABASE, and MAIN have the scope of the entire executable program, and are called **global names**. Unit number 5 in the READ statement also has global scope. Each of the program units MAIN and GET are regions of scope called **scoping units**. A scoping unit in general is not an entire program unit, but is a unit with holes in it. The holes occur wherever a scoping construct such as another program unit or defined-type definition appears within it. For example, the scoping unit corresponding to the module STACK_DATABASE does not include the inner parts of the procedure CREATE or the defined type STACK_TYPE.

Lines of communication or association are established between the local entities of two or more scoping units. For example, the CALL statement to GET in MAIN associates the dummy and actual arguments (all are **local names** in this case) so that while GET is being executed, the dummy argument I is the same as A and the dummy argument J is the same as B; this is **argument association**. Via storage association, the local variables X and Y in GET communicate with the local variable T in MAIN.

Another Example:

```
FUNCTION POP( SODA )                 ! POP is global.
   INTEGER K; REAL A(5), B(5), X, STMT_FUNC
   DATA (A(K),K=1,5) / 1.0, 2.0, 3.0, 4.0, 5.0 /
   STMT_FUNC(K) = K * 3 + A(3)       ! K has a statement scope.
      . . .
   K = SODA
   B = (/ (K, K=1,5) /)              ! K has a statement scope.
      . . .                          ! K still has the value SODA.
   PRINT *, (B(K), K=1,5)            ! K has a local scope.
   POP = SUM (A)                     ! K reset to the value 6.
END
```

The body of the subprogram POP has several examples of names with a scope that consists of a single statement or part of a statement. The variable K in the DATA statement has the scope of the implied DO in the DATA statement and is not associated with any other variable in the program, including the variable K appearing in other places in the program. Similarly, the dummy argument K in the statement function STMT_FUNC has as its scope only the statement function, and the variable K in the array constructor in the assignment to B has as its scope the array constructor. However, the declared K, the K in the assignment statement, and the K in the implied DO of the PRINT statement are the same variable; this K has a scope that is the entire program unit, excluding the DATA statement, statement function, and the array constructor.

89 Source Form

There are two source forms that may be used to write Fortran programs. One is called **fixed source form**. The other, **free source form**, is described here. Fixed source form is obsolete and is a candidate for deletion from the next Fortran standard.

Examples:

```
PROGRAM NICE

   !  This is a nice way to write a program!
      PRINT *
      PRINT *,  &
            12.0 + 34.6

END PROGRAM NICE


                   PROGRAM &
UGH  ! This is a terrible way to write a program!
          PRINT                                              &
              *;   PRINT &
     *     ,                &
               12.0         +&
    34.6
        END
```

Tip: Pick a consistent style for writing programs, using a consistent amount of indentation, placement of comments, etc.

A source form conversion program is available at no cost from the free software section of the Fortran Market: http://www.fortran.com/fortran.

It is possible to write programs in a way that is acceptable as both free source form and fixed source form. The rules are:

- Put labels in positions 1-5.
- Put statement bodies in positions 7-72.
- Begin comments with an exclamation (!) in any position except 6.
- Indicate all continuations with an ampersand in position 73 of the line to be continued and an ampersand in position 6 of the continuing line.

Related Topics:

INCLUDE Line

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 3.3
Fortran 95 Handbook, 3.3, 3.4
Fortran 95 Using F, 1.4

Things To Know:

1. A Fortran program consists of a sequence of statements, comments, and include lines; they are written on lines that contain from 0 to 132 characters.
2. A statement can be continued onto more lines if the last character of the line (not in a comment) to be continued is an ampersand (&).

   ```
   PRINT *,  &
         "I hope this is the right answer."
   ```

   An ampersand must be used on the continuing line if a keyword or character string is split between lines in free source form. For example

   ```
   PRINT *,  "I hope this is t&
         &he right answer."
   ```

   A statement may not have more than 40 lines.
3. The semicolon (;) symbol is used to separate multiple statements on the same line; it may be used in both free and fixed source form programs.

   ```
   A = 0; B = 0
   ```

4. The ! symbol for a comment may be used in both free and fixed source form programs. Any occurrence of the exclamation symbol (!) other than within a character context or a comment marks the beginning of a **comment**. The comment is terminated by the end of the line. All comments are ignored by the Fortran system.
5. In the absence of a continuation symbol, the end of a line marks the end of a statement.
6. Blank characters are significant in a Fortran program written using free source form. In general, they must not occur within things that normally would not be typed with blanks in English text, such as names and numbers. On the other hand, they must be used between two things that look like “words”. An example is that in the first line of a program the keyword PROGRAM and the name of the program must be separated by one or more blanks.
7. Keywords and names such as PRINT and NUMBER must contain no blank characters, except that keywords that consist of more than one English word may contain blanks between the words, as in the Fortran statement END DO. Two or more consecutive blanks are always equivalent to one blank unless they are in a character string.
8. Statements may begin anywhere, including positions 1 to 6.
9. Labels may appear anywhere before the main part of the statement, even in a position to the right of position 6.
10. A construct name followed by a colon may appear anywhere before the main part of the statement.

90 Storage Association

Storage association is a data communication mechanism that relies on the location of objects in computer memory rather than any names given to the objects. It permits a memory cell to be accessed by different names or by the same name but from different scoping units. Elements of the language that depend on storage association are common blocks, equivalence groups, ENTRY statements in function subprograms, and objects of sequence defined type.

Examples:

```
SUBROUTINE X
REAL A, B, C
COMMON /GLOBAL/ A, B, C
   . . .

SUBROUTINE Y             ! Subroutines X and Y access the same storage
   TYPE TRIPLET          !   through the common block GLOBAL, but
      SEQUENCE           !   with different variable names. A and
      REAL P, Q, R       !   TRIO % P refer to the same storage.
   END TYPE TRIPLET
   TYPE(TRIPLET) TRIO
   COMMON /GLOBAL/ TRIO

REAL D; INTEGER I        ! D and I are of different types.
EQUIVALENCE (D, I)       ! D and I refer to the same storage.
D = 3.0                  ! I is undefined and cannot be referenced.

REAL FUNCTION CALC (X, Y, I)
   REAL X, Y; INTEGER I, INDEX
      . . .
ENTRY INDEX (Y, I)       ! Function results CALC and INDEX
   . . .                 !   share storage.
END FUNCTION CALC
```

Tip: For new programs, avoid the use of features that depend on storage association. They are error prone and inflexible. Modules provide a better mechanism for handling global data.

Related Topics:

COMMON Statement
Defined Type: Definition
EQUIVALENCE Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 4.4, 5.5, 12.5.2.5, 14.6.3
Fortran 95 Handbook, 4.4, 5.11, 12.6.3, 14.3.3

Things To Know:

Fortran 77 had only two kinds of storage units: numeric and character. A common block may contain numeric storage units or character storage units but not both. It is not possible to equivalence a character object with an object of any other type nor can a character function have an entry with a different type of result.

Fortran 90 introduced several new objects: objects of intrinsic type and nondefault kinds, objects of user-defined type, and pointers. Each of these occupies a different unspecified storage unit, which complicates the storage association picture.

Objects of nondefault kind can appear in common, but every instance of that common block must have the same sequence of storage units. Objects of nondefault kind can be equivalenced, but only with objects of the same type and kind.

If an object of user-defined type is to appear in common, it must be a sequence structure. There are three kinds of sequence structures: numeric sequence structures (all components occupy numeric storage units or are of numeric sequence type), character sequence structures (all components occupy character storage units or are of character sequence type), and sequence structures (containing a mixture of components including those that occupy numeric, character, and unspecified storage units). A numeric sequence structure may appear freely in a common block that contains only objects that occupy numeric storage units. It may appear in equivalence groups with such objects. Similarly, a character sequence structure may appear freely in a common block that contains only objects that occupy character storage units. It may appear in equivalence groups with such objects. If a sequence structure appears in a common block, every instance of the common block must have the same sequence of storage units. Such a structure may be equivalenced only with objects of the same type.

A pointer may appear in a common block, but if it does, every instance of the common block must have the same sequence of storage units. A pointer may not be equivalenced.

If an ENTRY statement appears in a function, the results must be such that they could appear in an equivalence group. Thus if a function contains an ENTRY statement, no result can be a pointer.

Note that Fortran allows objects occupying numeric storage units and objects occupying character storage units to appear in the same common block, but if they do, every instance of the common block must contain the same sequence of storage units. In such cases the names of objects are not required to be the same.

91 Subroutines

A subroutine is one of two kinds of procedures (the other is a function). A subroutine performs a specific task such as calculating values for variables or performing input/output. A subroutine may change the value of arguments, variables in a common block, or variables in a module. In addition, a subroutine, in conjunction with an assignment interface, can be used to specify a defined assignment. A subroutine definition is an external, module, or internal subprogram. Both subroutines and functions are useful to encapsulate a well defined task in a program.

Examples:

```
PROGRAM REJECT
      . . .
   CALL EXCHANGE (A,T)              ! A subroutine reference
   CALL ALTITUDE (*90, LAT = 49)
      . . .
END

SUBROUTINE EXCHANGE (X, Y)          ! A subroutine definition
   TEMP = X; X = Y; Y = TEMP        !   with two arguments
END SUBROUTINE EXCHANGE

SUBROUTINE ALTITUDE (*, LONG, LAT)  ! An alternate return
   IMPLICIT NONE
   INTEGER, OPTIONAL :: LONG, LAT
      . . .
   RETURN 1
END SUBROUTINE ALTITUDE

! Examples of subroutine statements
SUBROUTINE PRESSURE_SURFACE         ! No arguments
SUBROUTINE TAFFY ()                 ! Also no arguments
RECURSIVE SUBROUTINE FACT (N,X)
```

Tip: A subroutine should be used instead of a function if there are side effects or if a result is to be returned in more than one variable.

Related Topics:

- Argument Association
- Argument Keywords
- Defined Operators and Assignment
- Elemental Procedures
- Functions
- Generic Procedures and Operators
- INTENT Attribute and Statement
- Internal Procedures
- Module Procedures
- OPTIONAL Attribute and Statement
- Pure Procedures
- Recursion

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 12.5.2.3, 13.10
Fortran 95 Handbook, 12.2, 13.7
Fortran 95 Using F, 3.3, A.10

Syntax:

A subroutine subprogram is:

```
[ prefix ] SUBROUTINE subroutine-name [ ( [ dummy-argument-list ] ) ]
    [ specification-part ]
    [ execution-part ]
    [ internal-subprogram-part ]
END [ SUBROUTINE [ subroutine-name ] ]
```

A dummy argument is one of:

```
name
*
```

A subroutine reference is one of:

```
CALL subroutine-name [ ( [ subroutine-actual-argument-list ] ) ]
variable = expression
```

A subroutine actual argument is one of:

```
expression
procedure-name
* label
```

Things To Know:

1. A subroutine prefix may be RECURSIVE, ELEMENTAL, or PURE, as it is in a function.
2. The keyword SUBROUTINE must appear on the END statement if the subroutine is a module or internal procedure.
3. Argument keywords may be used in a CALL statement.
4. A CALL statement or defined assignment (*variable* = *expression*) is used to invoke a subroutine. Execution of the subroutine begins with the first executable statement of the appropriate subroutine. An ENTRY statement in a subroutine subprogram creates another subroutine.
5. Data may be communicated to and from the subroutine using argument, host, use, sequence, or storage association.
6. The interface of an internal subroutine is explicit in its host. The interface of a module subroutine is explicit within the module, and if it is public, it is explicit in all program units using the module. The interface of an external subroutine is implicit, but may be made explicit by the use of an interface block.
7. An asterisk in a subroutine dummy argument list designates an alternate return.
8. An internal subroutine must not contain an ENTRY statement or an internal subprogram part.

92 TARGET Attribute and Statement

The TARGET attribute or statement specifies that the named object is a target that may be pointed to by a pointer. A target may be either a scalar or an array. The TARGET attribute allows the compiler to generate efficient code because only those objects specified with the TARGET or POINTER attribute can be dynamically aliased. The TARGET attribute also permits the programmer to specify clearly what objects in the program are dynamically aliased. This provides documentation for the program.

Examples:

```
INTEGER, POINTER, DIMENSION(:,:) :: P  ! P is a pointer.
INTEGER, TARGET :: T(10, 20, 30)       ! T is an array with
                                       !   the TARGET attribute.
P => T(10,1:10,2:5)                    ! P points to a rank-2
                                       !   section of T.

REAL, POINTER :: NOOTKA(:), TALK(:)
REAL, ALLOCATABLE, TARGET :: X(:)
ALLOCATE (X(1:100), STAT = IS)
NOOTKA => X(51:100)                    ! Pointer assignment
TALK => X(1:50)                        !   statements
      . . .

REAL R, P1, P2
TARGET R
POINTER P1, P2
R = 4.7
P1 => R                                ! P1 and P2 are both
P2 => P1                               !   aliases of R.
   . . .
ALLOCATE (P1)                          ! P1 no longer has R as
                                       !   its target
P1 = 9.4                               ! P1 now has 9.4 as its
   . . .                               !   value but not its
                                       !   target.
```

Related Topics:

ALLOCATE and DEALLOCATE Statements
Pointers
Pointer Association
POINTER Attribute and Statement
Pointer Nullification

Related Intrinsics:

ASSOCIATED (POINTER, TARGET)

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 5.1.2.8, 5.2.8, C.2.2
Fortran 95 Handbook, 5.4.2
Fortran 95 Using F, 8.1

TARGET Attribute and Statement

Syntax:

A type declaration statement with the TARGET attribute is:

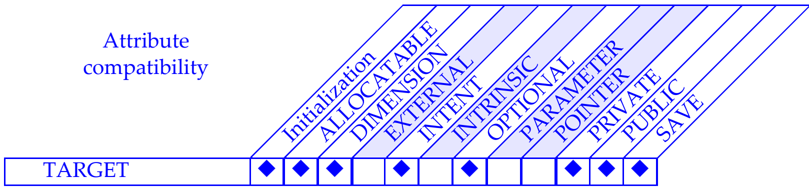
type , TARGET [, *attribute-list*] :: *entity-list*

A TARGET statement is:

TARGET [::] *object-name* [(*array-spec*)] &
    [, *object-name* [(*array-spec*)]]...

Things To Know:

1. A target may be a scalar or an array.
2. If the target in a pointer assignment is a variable, then:
   - it must have the TARGET attribute, or
   - it must be the component of a structure, the element of an array variable, or the substring of a character variable that has the TARGET attribute, or
   - it must have the POINTER attribute.



93 USE Statement and Use Association

The USE statement provides access to a module's public specifications and definitions. These include declared variables, named constants, defined-type definitions, procedure interfaces, procedures, generic identifiers, and namelist groups. The method of access is called **use association**. Such access may be limited by an ONLY clause on the USE statement, or the accessed entities may be renamed.

Examples:

```
MODULE RAT_ARITH           ! All entities are public except ZERO.
   TYPE RAT
     INTEGER  N, D
   END TYPE

   TYPE(RAT), PRIVATE, PARAMETER :: ZERO = RAT(0,1)
   TYPE(RAT), PUBLIC, PARAMETER  :: ONE = RAT(1,1)
   TYPE(RAT)  R1, R2
   NAMELIST  /NML_RAT/  R1, R2

   INTERFACE  OPERATOR( + )
     MODULE PROCEDURE  RAT_PLUS_RAT, INT_PLUS_RAT, ...
   END INTERFACE
    CONTAINS
      FUNCTION RAT_PLUS_RAT(L, R)
         . . .
      END FUNCTION
     . . .
END MODULE

PROGRAM  MINE
   ! From the module  RAT_ARITH, access only the entities
   !   RAT, ONE, R1, R2, NML_RAT
   !   but use the name ONE_RAT for the rational value ONE

   USE RAT_ARITH, ONLY: RAT, ONE_RAT => ONE, R1, R2, NML_RAT

   ! The OPERATOR + for rationals and the procedures
   !   RAT_PLUS_RAT and INT_PLUS_RAT are not available
   !   because of the ONLY clause.

   READ *, R2; R1 = ONE_RAT; WRITE( *, NML = NML_RAT)
     . . .
END PROGRAM
```

Related Topics:

Modules   Scope, Association, and Definition Overview

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 11.3.2, 14.6.1.2, C.8.2.1, C.8.3
Fortran 95 Handbook, 11.6.4-5, 12.1.4, 14.3.1.2
Fortran 95 Using F, 3.1.3

Syntax:

A USE statement is one of:

```
USE module-name
USE module-name , rename-list
USE module-name , ONLY : access-list
```

A rename is:

```
local-name => module-entity-name
```

An access is one of:

```
[ local-name => ]  module-entity-name
OPERATOR ( operator )
ASSIGNMENT ( = )
```

Things To Know:

1. All USE statements must be after the procedure header and before any other statements. More than one USE statement may be present, including more than one referring to the same module.
2. Modules may contain USE statements referring to other modules, except that the references must not directly or indirectly be recursive.
3. The first two forms of the USE statement make available by use association all publicly accessible entities in the module, except that the USE statement may rename some module entities. The third form makes available only those entities specified in the access list with possible renames of some module entities.
4. Entities made accessible by a USE statement include public entities from other modules referenced by USE statements in the referenced module. The implicit typing rules of the module are not accessed by a USE statement.
5. No accessible entity from a module, except a generic specifier, may be respecified in the program unit containing a USE statement. Entities with the same identifier as inaccessible entities may be specified anew in the program unit.
6. The same name or specifier may be made accessible via two or more USE statements. Such an entity must not be referenced in the scoping unit containing the USE statements, except if the specific procedures can be distinguished by the overload rules. A rename or ONLY clause is used to restrict access to one name or may be used to remove one name so that both entities are accessible.

Variables

A variable is a data object whose value can be defined and redefined during program execution. A variable may be a named scalar object or a named array; it may also be an array element, an array section, a structure component, or a character substring. Type declaration statements are generally used to declare variables unless reliance is placed on any implicit typing rules that may be in effect. A program could be written with all data specified as constants, but when the data changed the program would have to be revised and recompiled. By using variables and reading data as input, the program is generalized and can be used for various data sets without recompilation.

Examples:

```
REAL A, B (10, 50), C (10)
CHARACTER (20) PLACE
INTEGER, EXTERNAL :: NEW_CODE
TYPE STORE
   INTEGER BAR_CODES (10000)
   CHARACTER (2) STATE
END TYPE STORE
TYPE(STORE) VARIETY, DISCOUNT

C (5) = A + 1.0       ! Array element = scalar expression
B (10, 21:30) = C     ! Array section = named array
                      ! Structure component = character substring
VARIETY % STATE = PLACE (LEN(PLACE)-1 : LEN(PLACE))
                      ! Array element = function result
DISCOUNT % BAR_CODES(8952) = NEW_CODE(VCR,10)
```

Tip: It is good programming practice to declare all variables explicitly and not rely on implicit typing. Inserting an IMPLICIT NONE statement early in a program unit will instruct the compiler to detect undeclared variables.

Related Topics:

Array Overview
Assignment
Character Substring
Character Type and Constants
Complex Type and Constants
Defined Type: Structure Component
Expressions
Implicit Typing
Integer Type and Constants
Logical Type and Constants
READ/WRITE General Form
Real Type and Constants

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 6
Fortran 95 Handbook, 6
Fortran 95 Using F, 1.3, 3.5, 4, 5, 6

Syntax:

A variable is one of:

scalar-variable-name
array-variable-name
array-element
array-section
structure-component
substring

Things To Know:

1. Variables, as well as constants and function results, are used as operands in expressions. However, there are places where only variables may appear: as the left-hand side of an assignment statement, as an input object, as a do variable, as an initialized object in a DATA or type declaration statement, and as a STAT= specifier in an ALLOCATE statement, an IOSTAT= specifier in an OPEN or CLOSE statement, or an inquiry specifier in an INQUIRE statement. In addition, if a dummy argument has an intent of OUT or INOUT, the associated actual argument must be a variable.
2. In Fortran 77, all variables were scalar; this is no longer the case. A variable can be a scalar, an array, a structure, or a structure array.
3. A scalar is a data object that can have a single value from the set of values that characterize its type. A scalar has rank 0. A scalar may be a structure, an array element, a scalar component, or a substring of a scalar variable.
4. An array is a set of scalar data all of the same type and kind (and length, if character) that is arranged in a rectangular pattern. An array may have rank one up to rank seven. An array may be an array section. It may also be a substring of an array variable, an array component of a scalar variable, a structure array, a scalar component of an array variable, or a section of any of the previous arrays.

95 WHERE Construct and Statement

Certain array elements, selected by a mask, can be assigned in array assignment statements using the WHERE statement or WHERE construct. For any elemental operation in the array assignments, only the elements selected by the mask participate in the computation. The elemental operations include the usual intrinsic operations and the elemental intrinsic functions such as ABS. Masked array assignments are useful when certain elemental operations involving arrays need to be avoided because of program exceptions.

Examples:

```
REAL, DIMENSION(150) :: A, RECIPROCAL_A
REAL(DOUBLE), DIMENSION(10,20,30) :: B, SQRT_B
   . . .
! Assign 1.0/A to RECIPROCAL_A only where A is nonzero.
WHERE( A /= 0.0 ) RECIPROCAL_A = 1.0 / A
   . . .
WHERE( B .GE. 0.0 )
   SQRT_B = SQRT(B)    ! Assign to SQRT_B only where B is
                       !   positive.
ELSEWHERE
   SQRT_B = 0.0        ! Assign SQRT_B where B is negative.
END WHERE

WHERE (TEMPERATURE > 100)
   STATE = "GAS"
ELSEWHERE (TEMPERATURE > 0)
   STATE = "LIQUID"
ELSEWHERE
   STATE = "SOLID"
END WHERE

INTEGER, DIMENSION(NO_OF_TESTS, STUDENT) :: SCORE
CHARACTER, DIMENSION(NO_OF_TESTS, STUDENT) :: LETTER_GRADE

! Assign letter grades appropriate to numeric scores.
WHERE( SCORE >= 92 ) LETTER_GRADE = 'A'
WHERE( SCORE >= 82 .AND. SCORE <= 91 ) LETTER_GRADE = 'B'
WHERE( SCORE >= 72 .AND. SCORE <= 81 ) LETTER_GRADE = 'C'
WHERE( SCORE >= 62 .AND. SCORE <= 71 ) LETTER_GRADE = 'D'
WHERE( SCORE >=  0 .AND. SCORE <= 61 ) LETTER_GRADE = 'F'
```

Related Topics:

Array: Data-Parallel Operations
Assignment
Expressions
FORALL Construct and Statement

To Read More About It:

ISO 1539 : 1997, Fortran Standard, 7.5.3
Fortran 95 Handbook, 7.5.4
Fortran 95 Using F, 4.1.8

Syntax:

A WHERE statement is:

```
WHERE ( array-logical-expression ) where-assignment-statement
```

A WHERE construct is:

```
[ construct-name: ] WHERE ( array-logical-expression )
   [ where-body-statement ] . . .
[ ELSEWHERE ( array-logical-expression ) [ construct-name: ]
   [ where-body-statement ] . . . ] . . .
[ ELSEWHERE [ construct-name: ]
   [ where-body-statement ] . . . ]
END WHERE [ construct-name ]
```

A WHERE body statement is one of:

```
array = array-expression
where-statement
where-construct
```

Things To Know:

1. The shape of the array logical expression and the arrays on the left of each assignment statement must be the same; they may be of size zero.
2. The assignment statements must be either intrinsic array assignment statements or defined assignment statements whose subroutine is elemental.
3. Each elemental operation in the array assignment statement is either intrinsic or user defined; if user-defined, it must be defined with an elemental function. The operation is masked by the effective control mask, provided it is not an operand in an expression that is an argument of a transformational or nonelemental user-defined function. A nonelemental user-defined operation is treated like a nonelemental function wherein the operands as the arguments are not masked by the mask expression.
4. The elements of the arrays that are used in the WHERE part are those corresponding to the true elements of the array logical expression. The elements of the arrays that are used in the ELSEWHERE part are those corresponding to the elements corresponding to the false elements of the effective mask prior to that point and true elements of the ELSEWHERE mask, if any.
5. The array assignment statements are all executed in the order they appear in both the WHERE and ELSEWHERE parts of the WHERE construct.
6. In a WHERE construct, only the statement beginning with the keyword WHERE may be a branch target statement.
7. A construct name may be used to identify a WHERE construct.

Appendix A: Intrinsic Procedures

This appendix contains detailed specifications of the generic intrinsic procedures in alphabetical order.

For each procedure there are examples. The examples use type kind parameters for which the following assumptions are made:

1. The default real type has eight decimal digits of precision.
2. The value of the integer named constant HIGH is a kind parameter value for a real data type with 14 decimal digits of precision and an exponent range of at least 100.
3. The value of the integer named constant GREEK is a kind parameter value for a character data type that contains Greek letters.
4. The value of the integer named constant BIT is a kind parameter value for a logical data type that is an alternative to the default logical data type.
5. The value of the integer named constant SHORT is a kind parameter value for an integer data type with eight bits to represent integer values, that is, s in the bit model (18) for this integer type is 8.

All real values cannot be represented exactly in any processor; therefore, when the following text says something like "ACOS (.1_HIGH) has the value 1.4706289056333", it means that the value is a processor approximation to 1.4706289056333. The Fortran standard does not specify how accurate the approximation must be.

ABS (A)

Description. Absolute value.

Class. Elemental function.

Argument. A must be of type integer, real, or complex.

Result Type and Type Parameter. The same as A except that if A is complex, the result is real.

Result Value. If A is of type integer or real, the value of the result is $|A|$; if A is complex with value (x, y), the result is equal to a processor-dependent approximation to $\sqrt{x^2 + y^2}$.

Examples. ABS (–1) has the value 1. ABS (–1.5) has the value 1.5. ABS ((3.0, 4.0)) has the value 5.0.

ACHAR (I)

Description. Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the IACHAR function.

Class. Elemental function.

Argument. I must be of type integer.

Result Type and Type Parameter. Character of length one with kind type parameter value KIND ('A').

Result Value. If I has a value in the range $0 \le I \le 127$, the result is the character in position I of the ASCII collating sequence, provided the processor is capable of representing that character; otherwise, the result is processor dependent. If the processor is not capable of representing both uppercase and lowercase letters and I corresponds to a letter in a case that the processor is not capable of representing, the result is the letter in the case that the processor is capable of representing. ACHAR (IACHAR (C)) must have the value C for any character C capable of representation in the processor.

Examples. ACHAR (88) is 'X'. ACHAR (42) is '*'.

ACOS (X)

Description. Arccosine (inverse cosine) function.

Class. Elemental function.

Argument. X must be of type real with a value that satisfies the inequality $|X| \le 1$.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to arccos(X), expressed in radians. It lies in the range $0 \le \text{ACOS}(X) \le \pi$.

Examples. ACOS (0.54030231) has the value 1.0. ACOS (.1_HIGH) has the value 1.4706289056333 with kind HIGH.

ADJUSTL (STRING)

Description. Adjust to the left, removing leading blanks and inserting trailing blanks.

Class. Elemental function.

Argument. STRING must be of type character.

Result Type. Character of the same length and kind type parameter as STRING.

Result Value. The value of the result is the same as STRING except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

Examples. ADJUSTL (′ WORD′) is ′WORD ′. ADJUSTL (GREEK_′ τρια′) is GREEK_′τρια ′.

ADJUSTR (STRING)

Description. Adjust to the right, removing trailing blanks and inserting leading blanks.

Class. Elemental function.

Argument. STRING must be of type character.

Result Type. Character of the same length and kind type parameter as STRING.

Result Value. The value of the result is the same as STRING except that any trailing blanks have been deleted and the same number of leading blanks have been inserted.

Examples. ADJUSTR (′WORD ′) has the value ′ WORD′. ADJUSTR (GREEK_′τρια ′) has the value GREEK_′ τρια′.

AIMAG (Z)

Description. Imaginary part of a complex number.

Class. Elemental function.

Argument. Z must be of type complex.

Result Type and Type Parameter. Real with the same kind type parameter as Z.

Result Value. If Z has the value (x, y), the result has value y.

Examples. AIMAG ((2.0, 3.0)) has the value 3.0. AIMAG ((2.0_HIGH, 3.0)) has the value 3.0 with kind HIGH; the parts of a complex literal constant have the same precision, which is that of the part with the greatest precision.

AINT (A, KIND)

Optional Argument. KIND

Description. Truncation to a whole number.

Class. Elemental function.

Arguments.

A	must be of type real.
KIND (optional)	must be a scalar integer initialization expression.

Result Type and Type Parameter. The result is of type real. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.

Result Value. If $|A| < 1$, AINT (A) has the value 0; if $A \geq 1$, AINT (A) has a value equal to the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

Examples. AINT (2.783) has the value 2.0. AINT (–2.783) has the value –2.0. AINT (2.1111111111111_HIGH) and AINT (2.1111111111111, HIGH) have the value 2.0 with kind HIGH.

ALL (MASK, DIM)

Optional Argument. DIM

Description. Determine whether all values are true in MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$ where n is the rank of MASK. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n - 1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

Result Value.

Case (i): The result of ALL (MASK) has the value true if all elements of MASK are true or if MASK has size zero, and the result has value false if any element of MASK is false.

Case (ii): If MASK has rank one, ALL (MASK, DIM) has a value equal to that of ALL (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of ALL (MASK, DIM) is equal to ALL (MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$).

Examples.

Case (i): The value of ALL ((/ .TRUE., .FALSE., .TRUE. /)) is false. ALL ((/ .TRUE._BIT, .TRUE._BIT, .TRUE._BIT /)) is the value true with kind parameter BIT. Note that all values in an array constructor must have the same type and type parameter (4.6).

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$ then ALL (B .NE. C, DIM = 1) is (true, false, false) and ALL (B .NE. C, DIM = 2) is (false, false).

ALLOCATED (ARRAY)

Description. Indicate whether or not an allocatable array is currently allocated.

Class. Inquiry function.

Argument. ARRAY must be an allocatable array.

Result Type, Type Parameter, and Shape. Default logical scalar.

Result Value. The result has the value true if ARRAY is currently allocated and has the value false if ARRAY is not currently allocated. The result is undefined if the allocation status (6.5.1.1) of the array is undefined.

Example. If the following statements are processed

```
REAL, ALLOCATABLE :: A(:,:)
ALLOCATE (A(10,10))
PRINT *, ALLOCATED (A)
```

then T is printed.

ANINT (A, KIND)

Optional Argument. KIND

Description. Nearest whole number.

Class. Elemental function.

Arguments.

A	must be of type real.
KIND (optional)	must be a scalar integer initialization expression.

Result Type and Type Parameter. The result is of type real. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.

Result Value. If $A > 0$, ANINT (A) has the value AINT (A + 0.5); if $A \le 0$, ANINT (A) has the value AINT (A – 0.5).

Examples. ANINT (2.783) has the value 3.0. ANINT (–2.783) has the value –3.0. ANINT (2.7837837837837_HIGH) and ANINT (2.7837837837837, HIGH) have the value 3.0 with kind HIGH.

ANY (MASK, DIM)

Optional Argument. DIM

Description. Determine whether any value is true in MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \mathrm{DIM} \le n$, where n is the rank of MASK. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

Result Value.

Case (i): The result of ANY (MASK) has the value true if any element of MASK is true and has the value false if no elements are true or if MASK has size zero.

Case (ii): If MASK has rank one, ANY (MASK, DIM) has a value equal to that of ANY (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, ..., s_n)$ of ANY (MASK, DIM) is equal to ANY (MASK $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$).

Examples.

Case (i): The value of ANY ((/ .TRUE., .FALSE., .TRUE. /)) is true. ANY ((/ .FALSE._BIT, .FALSE._BIT, .FALSE._BIT /)) is false with kind parameter BIT.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, ANY (B .NE. C, DIM = 1) is (true, false, true) and ANY (B .NE. C, DIM = 2) is (true, true).

ASIN (X)

Description. Arcsine (inverse sine) function.

Class. Elemental function.

Argument. X must be of type real. Its value must satisfy the inequality $|X| \le 1$.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to arcsin(X), expressed in radians. It lies in the range $-\pi/2 \le \mathrm{ASIN}\ (X) \le \pi/2$.

Examples. ASIN (0.84147098) has the value 1.0. ASIN (1.0_HIGH) has the value 1.5707963267949 with kind HIGH.

ASSOCIATED (POINTER, TARGET)

Optional Argument. TARGET

Description. Returns the association status of its pointer argument or indicates the pointer is associated with the target.

Class. Inquiry function.

Arguments.

POINTER	must be a pointer and may be of any type. Its pointer association status must not be undefined.
TARGET (optional)	must be a pointer or target. If it is a pointer, its pointer association status must not be undefined.

Result Type. The result is scalar of type default logical.

Result Value.

Case (i): If TARGET is absent, the result is true if POINTER is currently associated with a target and false if it is not.

Case (ii): If TARGET is present and is a target, the result is true if POINTER is currently associated with TARGET and false if it is not.

Case (iii): If TARGET is present and is a pointer, the result is true if both POINTER and TARGET are currently associated with the same target, and is false otherwise. If either POINTER or TARGET is disassociated, the result is false.

Examples.

Case (i): ASSOCIATED (PTR) is true if PTR is currently associated with a target.

Case (ii): ASSOCIATED (PTR, TAR) is true if the following statements have been processed:

```
REAL, TARGET  :: TAR (0:100)
REAL, POINTER :: PTR(:)
PTR => TAR
```

The subscript range for PTR is 0:100. If the pointer assignment statement is either of

```
PTR => TAR (:)
PTR => TAR (0:100)
```

ASSOCIATED (PTR, TAR) is still true, but in both cases the subscript range for PTR is 1:101 (5.3.1.3). However, if the pointer assignment statement is

```
PTR => TAR (0:99)
```

ASSOCIATED (PTR, TAR) is false, because TAR (0:99) is not the same as TAR.

Case (iii): ASSOCIATED (PTR1, PTR2) is true if the following statements have been processed:

```
REAL, POINTER :: PTR1(:), PTR2(:)
ALLOCATE (PTR1(0:10))
PTR2 => PTR1
```

After the execution of either of the statements:

```
NULLIFY (PTR1)
NULLIFY (PTR2)
```

ASSOCIATED (PTR1, PTR2) is false.

ATAN (X)

Description. Arctangent (inverse tangent) function.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to arctan(X), expressed in radians, that lies in the range $-\pi/2 \leq \text{ATAN}(X) \leq \pi/2$.

Examples. ATAN (1.5574077) has the value 1.0. ATAN (2.0_HIGH/3.0) has the value 0.58800260354757 with kind HIGH.

ATAN2 (Y, X)

Description. Arctangent (inverse tangent) function. The result is the principal value of the argument of the nonzero complex number (X, Y).

Class. Elemental function.

Arguments.

Y — must be of type real.

X — must be of the same type and kind type parameter as Y. If Y has the value zero, X must not have the value zero.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the principal value of the argument of the complex number (X, Y), expressed in radians. It lies in the range $-\pi < \text{ATAN2}(Y, X) \le \pi$ and is equal to a processor-dependent approximation to a value of arctan(Y/X) if $X \neq 0$. If $Y > 0$, the result is positive. If $Y = 0$, the result is zero if $X > 0$ and the result is π if $X < 0$. If $Y < 0$, the result is negative. If $X = 0$, the absolute value of the result is $\pi/2$.

Examples. ATAN2 (1.5574077, 1.0) has the value 1.0. If Y has the value $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$ and X has the value $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$, the value of ATAN2 (Y, X) is $\begin{bmatrix} \frac{3\pi}{4} & \frac{\pi}{4} \\ \frac{-3\pi}{4} & -\frac{\pi}{4} \end{bmatrix}$.

BIT_SIZE (I)

Description. Returns the number of bits s defined by the model of topic 18 for integers with the kind parameter of the argument.

Class. Inquiry function.

Argument. I must be of type integer.

Result Type, Type Parameter, and Shape. Scalar integer with the same kind type parameter as I.

Result Value. The result has the value of the number of bits s in the model integer defined for bit manipulation contexts in topic 18 for integers with the kind parameter of the argument.

Examples. BIT_SIZE (1) has the value 32 if s in the model is 32. BIT_SIZE (1_SHORT) is 8 with kind SHORT.

BTEST (I, POS)

Description. Tests a bit of an integer value.

Class. Elemental function.

Arguments.

I	must be of type integer.
POS	must be of type integer. It must be nonnegative and be less than BIT_SIZE (I).

Result Type. The result is of type default logical.

Result Value. The result has the value true if bit POS of I has the value 1 and has the value false if bit POS of I has the value 0. The model for the interpretation of an integer value as a sequence of bits is in topic 18.

Examples. BTEST (8, 3) has the value true. BTEST (8_SHORT, 3) has the value true. If A has the value $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, the value of BTEST (A, 2) is $\begin{bmatrix} \text{false} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$, and the value of BTEST (2, A) is $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$.

CEILING (A, KIND)

Optional Argument. KIND

Description. Returns the least integer greater than or equal to its argument.

Class. Elemental function.

Argument.

A	must be of type real.
IIND (optional)	must be a scalar integer initialization expression.

Result Type and Type Parameter. Default integer if KIND is not present; otherwise, integer of kind specified by KIND.

Result Value. The result has a value equal to the least integer greater than or equal to A.

Examples. CEILING (3.7) has the value 4. CEILING (–3.7) has the value –3. CEILING (20.0_HIGH/3) has the value 7.

CHAR (I, KIND)

Optional Argument. KIND

Description. Returns the character in a given position of the processor collating sequence associated with the specified kind type parameter. It is the inverse of the function ICHAR.

Class. Elemental function.

Arguments.

I	must be of type integer with a value in the range $0 \le I \le n - 1$, where n is the number of characters in the collating sequence associated with the specified kind type parameter.
KIND (optional)	must be a scalar integer initialization expression.

Result Type and Type Parameters. Character of length one. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default character type.

Result Value. The result is the character in position I of the collating sequence associated with the specified kind type parameter. ICHAR (CHAR (I, KIND (C))) must have the value I for $0 \le I \le n - 1$ and CHAR (ICHAR (C), KIND (C)) must have the value C for any character C capable of representation in the processor.

Examples. CHAR (88) is 'X' on a processor using the ASCII collating sequence. CHAR (97, GREEK) might be 'α' on a processor that supports a character type containing Greek letters.

CMPLX (X, Y, KIND)

Optional Arguments. Y, KIND

Description. Convert to complex type.

Class. Elemental function.

Arguments.

X	must be of type integer, real, or complex.
Y (optional)	must be of type integer or real. It must not be present if X is of type complex.
KIND (optional)	must be a scalar integer initialization expression.

Result Type and Type Parameter. The result is of type complex. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default real type.

Result Value. If Y is absent and X is not complex, it is as if Y were present with the value zero. If Y is absent and X is complex, it is as if Y were present with the value AIMAG (X). CMPLX (X, Y, KIND) has the complex value whose real part is REAL (X, KIND) and whose imaginary part is REAL (Y, KIND).

Examples. CMPLX (–3) is –3.0 + 0ι. CMPLX ((4.1, 0.0), KIND=HIGH), CMPLX ((4.1, 0), KIND=HIGH), and CMPLX (4.1, KIND=HIGH) are each 4.1 + 0ι with kind HIGH.

CONJG (Z)

Description. Conjugate of a complex number.

Class. Elemental function.

Argument. Z must be of type complex.

Result Type and Type Parameter. Same as Z.

Result Value. If Z has the value (x, y), the result has the value $(x, -y)$.

Examples. CONJG ((2.0, 3.0)) is 2.0 – 3.0ι. CONJG ((0, –4.1_HIGH)) is 0 + 4.1ι with kind HIGH.

COS (X)

Description. Cosine function.

Class. Elemental function.

Argument. X must be of type real or complex.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to cos(X). If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

Examples. COS (1.0) has the value 0.54030231. COS ((1.0_HIGH, 1.0)) has the value 0.83373002513115 – 0.98889770576287ι with kind HIGH.

COSH (X)

Description. Hyperbolic cosine function.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to cosh(X).

Examples. COSH (1.0) has the value 1.5430806. COSH (0.1_HIGH) has the value 1.0050041680558 with kind HIGH.

COUNT (MASK, DIM)

Optional Argument. DIM

Description. Count the number of true elements of MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \mathrm{DIM} \le n$, where n is the rank of MASK. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type default integer. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

Result Value.

Case (i): The result of COUNT (MASK) has a value equal to the number of true elements of MASK or has the value zero if MASK has size zero.

Case (ii): If MASK has rank one, COUNT (MASK, DIM) has a value equal to that of COUNT (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, ..., s_n)$ of COUNT (MASK, DIM) is equal to COUNT (MASK $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$).

Examples.

Case (i): The value of COUNT ((/ .TRUE., .FALSE., .TRUE. /)) is 2.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, COUNT (B .NE. C, DIM = 1) is (2, 0, 1) and COUNT (B .NE. C, DIM = 2) is (1, 2).

CPU_TIME (TIME)

Description. Returns the processor time.

Class. Subroutine

Argument. TIME must be scalar and of type real. It is an INTENT(OUT) argument that is set to a processor-dependent approximation to the processor time in seconds. If the processor cannot return a meaningful time, the value is set to a processor dependent negative value.

Example.

```
PROGRAM TIME_SOME_CODE
REAL START_TIME, STOP_TIME
```

```
   . . .
CALL CPU_TIME (START_TIME)   ! This may or may not be the start of the program.
   . . .          !  Code to be timed.
CALL CPU_TIME(STOP_TIME)      ! This may or may not be the end of the program.
WRITE(*,*)  'Time taken by code was ', STOP_TIME - START_TIME , ' seconds'
END PROGRAM TIME_SOME_CODE
```

The CPU_TIME subroutine is useful for comparisons of different algorithms to find the most efficient one to use for the program or to determine which parts of a program are taking the most time to pinpoint those areas that need more work or optimization.

The duration of a complete program may be timed, or a section of code such as a subroutine or function may be timed. Processor time is not precisely defined in Fortran. Different processors may provide processor times with considerable variability. However when running different algorithms on the same computer, comparisons may be made that are useful indicators.

When seeking processor time for a given program, the system you are using may or may not include system overhead and there is a misconception that this time is somehow related to the time elapsed for a real time evaluation. This has no apparent connection to "wall clock" time.

An additional subroutine could be provided for a parallel processor, for example, where the argument of the subroutine is an array.

CSHIFT (ARRAY, SHIFT, DIM)

Optional Argument. DIM

Description. Perform a circular shift on an array expression of rank one or perform circular shifts on all the complete rank one sections along a given dimension of an array expression of rank two or greater. Elements shifted out at one end of a section are shifted in at the other end. Different sections may be shifted by different amounts and in different directions (positive for left shifts, negative for right shifts).

Class. Transformational function.

Arguments.

ARRAY — may be of any type. It must not be scalar.

SHIFT — must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank $n-1$ and of shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

DIM (optional) — must be a scalar and of type integer with a value in the range $1 \le DIM \le n$, where n is the rank of ARRAY. If DIM is omitted, it is as if it were present with the value 1.

Result Type, Type Parameter, and Shape. The result is of the type and type parameters of ARRAY, and has the shape of ARRAY.

Result Value.

Case (i): If ARRAY has rank one, element i of the result is ARRAY (1 + MODULO (i + SHIFT – 1, SIZE (ARRAY))).

Case (ii): If ARRAY has rank greater than one, section $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$ of the result has a value equal to CSHIFT (ARRAY $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$, sh, 1), where sh is SHIFT or SHIFT $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$.

Examples.

Case (i): If V is the array (1, 2, 3, 4, 5, 6), the effect of shifting V circularly to the left by two positions is achieved by CSHIFT (V, SHIFT = 2) which has the value (3, 4, 5, 6, 1, 2); CSHIFT (V, SHIFT = –2) achieves a circular shift to the right by two positions and has the value (5, 6, 1, 2, 3, 4).

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts. If M is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, the value of CSHIFT (M, SHIFT = –1, DIM = 2) is $\begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}$, and the value of CSHIFT (M, SHIFT = (/ –1, 1, 0 /), DIM = 2) is $\begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$.

DATE_AND_TIME (DATE, TIME, ZONE, VALUES)

Optional Arguments. DATE, TIME, ZONE, VALUES

Description. Returns data on the real-time clock and date in a form compatible with the representations defined in ISO 8601:1988.

Class. Subroutine.

Arguments.

DATE (optional) must be scalar and of type default character, and must be of length at least 8 in order to contain the complete value. It is an INTENT (OUT) argument. Its leftmost 8 characters are set to a value of the form *CCYYMMDD*, where *CC* is the century, *YY* the year within the century, *MM* the month within the year, and *DD* the day within the month. If there is no date available, they are set to blank.

TIME (optional) must be scalar and of type default character, and must be of length at least 10 in order to contain the complete value. It is an INTENT (OUT) argument. Its leftmost 10 characters are set to a value of the form *hhmmss.sss*, where *hh* is the hour of the day, *mm* is the minutes of the hour, and *ss.sss* is the seconds and milliseconds of the minute. If there is no clock available, they are set to blank.

ZONE (optional) must be scalar and of type default character, and must be of length at least 5 in order to contain the complete value. It is an INTENT (OUT) argument. Its leftmost 5 characters are set to a value of the form ±*hhmm*, where *hh* and *mm* are the time difference with respect to Coordinated Universal Time (UTC) in hours and parts of an hour expressed in minutes, respectively. If there is no clock available, they are set to blank.

VALUES (optional) must be of type default integer and of rank one. It is an INTENT (OUT) argument. Its size must be at least 8. The values returned in VALUES are as follows:

VALUES (1) the year (for example, 1990), or –HUGE (0) if there is no date available;

VALUES (2) the month of the year, or –HUGE (0) if there is no date available;

VALUES (3) the day of the month, or –HUGE (0) if there is no date available;

VALUES (4) the time difference with respect to Coordinated Universal Time (UTC) in minutes, or –HUGE (0) if this information is not available;

VALUES (5) the hour of the day, in the range of 0 to 23, or –HUGE (0) if there is no clock;

VALUES (6) the minutes of the hour, in the range 0 to 59, or –HUGE (0) if there is no clock;

VALUES (7) the seconds of the minute, in the range 0 to 60, or –HUGE (0) if there is no clock;

VALUES (8) the milliseconds of the second, in the range 0 to 999, or –HUGE (0) if there is no clock.

HUGE is described in A.38.

Example.

```
INTEGER DATE_TIME (8)
CHARACTER (LEN = 10) BIG_BEN (3)
CALL DATE_AND_TIME (BIG_BEN (1), BIG_BEN (2),  &
                    BIG_BEN (3), DATE_TIME)
```

if called in Geneva, Switzerland on 1985 April 12 at 15:27:35.5 would have assigned the value "19850412 " to BIG_BEN (1), the value "152735.500" to BIG_BEN (2), and the value "+0100 " to BIG_BEN (3), and the following values to DATE_TIME: 1985, 4, 12, 60, 15, 27, 35, 500.

Note that UTC is defined by CCIR Recommendation 460-2 (and is also known as Greenwich Mean Time).

DBLE (A)

Description. Convert to double precision real type.

Class. Elemental function.

Argument. A must be of type integer, real, or complex.

Result Type and Type Parameter. Double precision real.

Result Value.

Case (i): If A is of type double precision real, DBLE (A) = A.

Case (ii): If A is of type integer or real, the result is as much precision of the significant part of A as a double precision real datum can contain.

Case (iii): If A is of type complex, the result is as much precision of the significant part of the real part of A as a double precision real datum can contain.

Examples. DBLE (–.3) is –0.3 of type double precision real. DBLE (1.0_HIGH/3) is 0.33333333333333 of type double precision real.

DIGITS (X)

Description. Returns the number of significant digits in the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has the value q if X is of type integer and p if X is of type real, where q and p are as defined in topic 18 for the model representing numbers of the same type and kind type parameter as X.

Examples. DIGITS (X) has the value 24 for real X whose model described in topic 18. DIGITS (ARRAY_A), where ARRAY_A is declared as

```
REAL (KIND=HIGH) ARRAY_A (100)
```

might have the value 48 for a model somewhat different from the one described in topic 18.

DIM (X, Y)

Description. The difference X–Y if it is positive; otherwise zero.

Class. Elemental function.

Arguments.

X	must be of type integer or real.
Y	must be of the same type and kind type parameter as X.

Result Type and Type Parameter. Same as X.

Result Value. The value of the result is X–Y if X > Y and zero otherwise.

Examples. DIM (5, 3) has the value 2. DIM (–3.0, 2.0) has the value 0.0.

DOT_PRODUCT (VECTOR_A, VECTOR_B)

Description. Performs dot-product multiplication of numeric or logical vectors.

Class. Transformational function.

Arguments.

VECTOR_A	must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one.

VECTOR_B must be of numeric type if VECTOR_A is of numeric type or of type logical if VECTOR_A is of type logical. It must be array valued and of rank one. It must be of the same size as VECTOR_A.

Result Type, Type Parameter, and Shape. If the arguments are of numeric type, the type and kind type parameter of the result are those of the expression VECTOR_A * VECTOR_B determined by the types of the arguments according to 7.2.8. If the arguments are of type logical, the result is of type logical with the kind type parameter of the expression VECTOR_A .AND. VECTOR_B according to 7.2.8. The result is scalar.

Result Value.

Case (i): If VECTOR_A is of type integer or real, the result has the value SUM (VECTOR_A*VECTOR_B). If the vectors have size zero, the result has the value zero.

Case (ii): If VECTOR_A is of type complex, the result has the value SUM (CONJG (VECTOR_A)*VECTOR_B). If the vectors have size zero, the result has the value zero.

Case (iii): If VECTOR_A is of type logical, the result has the value ANY (VECTOR_A .AND. VECTOR_B). If the vectors have size zero, the result has the value false.

Examples.

Case (i): DOT_PRODUCT ((/ 1, 2, 3 /), (/ 2, 3, 4 /)) has the value 20.

Case (ii): DOT_PRODUCT ((/ (1.0, 2.0), (2.0, 3.0) /), (/ (1.0, 1.0), (1.0, 4.0) /)) has the value 17 + 4ι.

Case (iii): DOT_PRODUCT ((/ .TRUE., .FALSE. /), (/ .TRUE., .TRUE. /)) has the value true.

DPROD (X, Y)

Description. Double precision real product.

Class. Elemental function.

Arguments.

X must be of type default real.

Y must be of type default real.

Result Type and Type Parameters. Double precision real.

Result Value. The result has a value equal to a processor-dependent approximation to the product of X and Y.

Example. DPROD (–3.0, 2.0) has the value –6.0 of type double precision real.

EOSHIFT (ARRAY, SHIFT, BOUNDARY, DIM)

Optional Arguments. BOUNDARY, DIM

Description. Perform an end-off shift on an array expression of rank one or perform end-off shifts on all the complete rank-one sections along a given dimension of an array expression of rank two or greater. Elements are shifted off at one end of a section and copies of a boundary value are shifted in at the other end. Different sections may have different boundary values and may be shifted by different amounts and in different directions (positive for left shifts, negative for right shifts).

Class. Transformational function.

Arguments.

ARRAY may be of any type. It must not be scalar.

SHIFT must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank $n - 1$ and of shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

BOUNDARY (optional) must be of the same type and type parameters as ARRAY and must be scalar if ARRAY has rank one; otherwise, it must be either scalar or of rank $n - 1$ and of shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$. BOUNDARY may be omitted for the data types in the following table and, in this case, it is as if it were present with the scalar value shown.

Type of ARRAY	Value of BOUNDARY
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	false
Character (*len*)	*len* blanks

DIM (optional) must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY. If DIM is omitted, it is as if it were present with the value 1.

Result Type, Type Parameter, and Shape. The result has the type, type parameters, and shape of ARRAY.

Result Value. Element $(s_1, s_2, ..., s_n)$ of the result has the value ARRAY $(s_1, s_2, ..., s_{DIM-1}, s_{DIM} + sh, s_{DIM+1}, ..., s_n)$ where sh is SHIFT or SHIFT $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ provided the inequality LBOUND (ARRAY, DIM) $\le s_{DIM} + sh \le$ UBOUND (ARRAY, DIM) holds and is otherwise BOUNDARY or BOUNDARY $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$.

Examples.

Case (i): If V is the array (1, 2, 3, 4, 5, 6), the effect of shifting V end-off to the left by 3 positions is achieved by EOSHIFT (V, SHIFT = 3) which has the value (4, 5, 6, 0, 0, 0); EOSHIFT (V, SHIFT = –2, BOUNDARY = 99) achieves an end-off shift to the right by 2 positions with the boundary value of 99 and has the value (99, 99, 1, 2, 3, 4).

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts and the boundary elements can be the same or different. If M is the array

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$$, then the value of EOSHIFT (M, SHIFT = –1, BOUNDARY = '*', DIM = 2) is

$$\begin{bmatrix} * & A & B \\ * & D & E \\ * & G & H \end{bmatrix}$$, and the value of EOSHIFT (M, SHIFT = (/ –1, 1, 0 /), BOUNDARY = (/ '*', '/', '?' /),

DIM = 2) is $$\begin{bmatrix} * & A & B \\ E & F & / \\ G & H & I \end{bmatrix}$$.

EPSILON (X)

Description. Returns a positive model number that is almost negligible compared to unity in the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Scalar of the same type and kind type parameter as X.

Result Value. The result has the value b^{1-p} where b and p are as defined in topic 18 for the model representing numbers of the same type and kind type parameter as X.

Examples. EPSILON (X) has the value 2^{-23} for real X whose model is described in topic 18. EPSILON (Y), where Y has kind parameter HIGH, would be 2^{-47} if p is 48 for the model of kind HIGH.

EXP (X)

Description. Exponential.

Class. Elemental function.

Argument. X must be of type real or complex.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to e^{X}. If X is of type complex, its imaginary part is regarded as a value in radians.

Examples. EXP (1.0) has the value 2.7182818. EXP (2.0_HIGH/3.0) has the value 1.9477340410547 with kind HIGH.

EXPONENT (X)

Description. Returns the exponent part of the argument when represented as a model number.

Class. Elemental function.

Argument. X must be of type real.

Result Type. Default integer.

Result Value. The result has a value equal to the exponent e of the model representation (18) for the value of X, provided X is nonzero and e is within the range for default integers. The result is undefined if the processor cannot represent e in the default integer type. EXPONENT (X) has the value zero if X is zero.

Examples. EXPONENT (1.0) has the value 1 and EXPONENT (4.1) has the value 3 for reals whose model is described in topic 18.

FLOOR (A, KIND)

Optional Argument. KIND.

Description. Returns the greatest integer less than or equal to its argument.

Class. Elemental function.

Argument.

A	must be of type real.
KIND (optional)	must be a scalar integer initialization expression.

Result Type and Type Parameter. Default integer, if KIND is not present; otherwise, integer of kind specified by KIND..

Result Value. The result has value equal to the greatest integer less than or equal to A.

Examples. FLOOR (3.7) has the value 3. FLOOR (–3.7) has the value –4. FLOOR (10.0_HIGH/3) has the value 3.

FRACTION (X)

Description. Returns the fractional part of the model representation of the argument value.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has the value $X \times b^{-e}$, where b and e are as defined in topic 18 for the model representation of X. If X has the value zero, the result has the value zero.

Example. FRACTION (3.0) has the value 0.75 for reals whose model is described in topic 18.

HUGE (X)

Description. Returns the largest number in the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Scalar of the same type and kind type parameter as X.

Result Value. The result has the value $r^q - 1$ if X is of type integer and $(1 - b^{-p})b^{e_{max}}$ if X is of type real, where r, q, b, p, and e_{max} are as defined in topic 18 for the model representing numbers of the same type and kind type parameter as X.

Example. HUGE (X) has the value $(1 - 2^{-24}) \times 2^{127}$ for real X whose model is described in topic 18.

IACHAR (C)

Description. Returns the position of a character in the ASCII collating sequence.

Class. Elemental function.

Argument. C must be of type default character and of length one.

Result Type and Type Parameter. Default integer.

Result Value. If C is in the collating sequence defined by the codes specified in ISO 646:1983 (International Reference Version), the result is the position of C in that sequence and satisfies the inequality ($0 \leq \text{IACHAR (C)} \leq 127$). A processor-dependent value is returned if C is not in the ASCII collating sequence. The results are consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE (C, D) is true, IACHAR (C) .LE. IACHAR (D) is true where C and D are any two characters representable by the processor.

Examples. IACHAR ('X') has the value 88. IACHAR ('*') has the value 42.

IAND (I, J)

Description. Performs a logical AND.

Class. Elemental function.

Arguments.

I — must be of type integer.

J — must be of type integer with the same kind type parameter as I.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IAND (I, J)
1	1	1
1	0	0
0	1	0
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in topic 18.

Examples. IAND (1, 3) has the value 1. IAND (2_SHORT, 10_SHORT) is 2 with kind SHORT.

IBCLR (I, POS)

Description. Clears one bit to zero.

Class. Elemental function.

Arguments.

I — must be of type integer.

POS — must be of type integer. It must be nonnegative and less than BIT_SIZE (I).

Result Type and Type Parameter. Same as I.

Result Value. The result has the value of the sequence of bits of I, except that bit POS of I is set to zero. The model for the interpretation of an integer value as a sequence of bits is in topic 18.

Examples. IBCLR (14, 1) has the result 12. If V has the value (1, 2, 3, 4), the value of IBCLR (POS = V, I = 31) is (29, 27, 23, 15). The value of IBCLR ((/ 15_SHORT, 31_SHORT, 7_SHORT /), 3) is (7, 23, 7) with kind SHORT.

IBITS (I, POS, LEN)

Description. Extracts a sequence of bits.

Class. Elemental function.

Arguments.

I	must be of type integer.
POS	must be of type integer. It must be nonnegative and POS + LEN must be less than or equal to BIT_SIZE (I).
LEN	must be of type integer and nonnegative.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value of the sequence of LEN bits in I beginning at bit POS, right-adjusted and with all other bits zero. The model for the interpretation of an integer value as a sequence of bits is in topic 18.

Examples. IBITS (14, 1, 3) has the value 7. The value of IBITS ((/ 15_SHORT, 31_SHORT, 7_SHORT /), 2_SHORT, 3_SHORT) is (3, 7, 1) with kind SHORT.

IBSET (I, POS)

Description. Sets one bit to one.

Class. Elemental function.

Arguments.

I	must be of type integer.
POS	must be of type integer. It must be nonnegative and less than BIT_SIZE (I).

Result Type and Type Parameter. Same as I.

Result Value. The result has the value of the sequence of bits of I, except that bit POS of I is set to one. The model for the interpretation of an integer value as a sequence of bits is in topic 18.

Examples. IBSET (12, 1) has the value 14. If V has the value (1, 2, 3, 4), the value of IBSET (POS = V, I = 0) is (2, 4, 8, 16). The value of IBSET ((/ 15_SHORT, 31_SHORT, 7_SHORT /), 3) is (15, 31, 15) with kind SHORT.

ICHAR (C)

Description. Returns the position of a character in the processor collating sequence associated with the kind type parameter of the character.

Class. Elemental function.

Argument. C must be of type character and of length one. Its value must be that of a character capable of representation in the processor.

Result Type and Type Parameter. Default integer.

Result Value. The result is the position of C in the processor collating sequence associated with the kind type parameter of C and is in the range $0 \le \text{ICHAR (C)} \le n - 1$, where n is the number of characters in the collating sequence. For any characters C and D capable of representation in the processor, C .LE. D is true if and only if ICHAR (C) .LE. ICHAR (D) is true and C .EQ. D is true if and only if ICHAR (C). EQ. ICHAR (D) is true.

Examples. ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence for the default character type. ICHAR ('*') has the value 42 on such a processor.

IEOR (I, J)

Description. Performs an exclusive OR.

Class. Elemental function.

Arguments.

I	must be of type integer.
J	must be of type integer with the same kind type parameter as I.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IEOR (I, J)
1	1	0
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in topic 18.

Examples. IEOR (1, 3) has the value 2. IEOR ((/ 3_SHORT, 10_SHORT /), 2_SHORT) is (1, 8) with kind SHORT.

INDEX (STRING, SUBSTRING, BACK)

Optional Argument. BACK

Description. Returns the starting position of a substring within a string.

Class. Elemental function.

Arguments.

STRING	must be of type character.
SUBSTRING	must be of type character with the same kind type parameter as STRING.
BACK (optional)	must be of type logical.

Result Type and Type Parameter. Default integer.

Result Value.

Case (i): If BACK is absent or present with the value false, the result is the minimum positive value of I such that STRING (I : I + LEN (SUBSTRING) – 1) = SUBSTRING or zero if there is no such value. Zero is returned if LEN (STRING) < LEN (SUBSTRING) and one is returned if LEN (SUBSTRING) = 0.

Case (ii): If BACK is present with the value true, the result is the maximum value of I less than or equal to LEN (STRING) – LEN (SUBSTRING) + 1 such that STRING (I : I + LEN (SUBSTRING) – 1) = SUBSTRING or zero if there is no such value. Zero is returned if LEN (STRING) < LEN (SUBSTRING) and LEN (STRING) + 1 is returned if LEN (SUBSTRING) = 0.

Examples. INDEX (’FORTRAN’, ’R’) has the value 3. INDEX (’FORTRAN’, ’R’, BACK = .TRUE.) has the value 5. INDEX (GREEK_"τρια", GREEK_"ι") has the value 3. INDEX ("XXX", "") has the value 1. INDEX ("XXX", "", BACK=.TRUE.) has the value 4.

INT (A, KIND)

Optional Argument. KIND

Description. Convert to integer type.

Class. Elemental function.

Arguments.

A	must be of type integer, real, or complex.
KIND (optional)	must be a scalar integer initialization expression.

Result Type and Type Parameter. Integer. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default integer type.

Result Value.

Case (i): If A is of type integer, INT (A) = A.

Case (ii): If A is of type real, there are two cases: if $|A| < 1$, INT (A) has the value 0; if $|A| \geq 1$, INT (A) is the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

Case (iii): If A is of type complex, INT (A) is the value obtained by applying the case (ii) rule to the real part of A. The result is undefined if the processor cannot represent the result in the specified integer type.

Examples. INT (–3.7) has the value –3. INT (9.1_HIGH/4.0_HIGH, SHORT) is 2 with kind SHORT.

IOR (I, J)

Description. Performs an inclusive OR.

Class. Elemental function.

Arguments.

I — must be of type integer.

J — must be of type integer with the same kind type parameter as I.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IOR (I, J)
1	1	1
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in topic 18.

Examples. IOR (1, 3) has the value 3. IOR ((/ 3_SHORT, 2_SHORT /), (/ 1_SHORT, 10_SHORT /)) is (3, 10) with kind SHORT.

ISHFT (I, SHIFT)

Description. Performs a logical shift.

Class. Elemental function.

Arguments.

I — must be of type integer.

SHIFT — must be of type integer. The absolute value of SHIFT must be less than or equal to BIT_SIZE (I).

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by shifting the bits of I by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if SHIFT is zero, no shift is performed. Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end. The model for the interpretation of an integer value as a sequence of bits is in topic 18.

Examples. ISHFT (3, 1) has the value 6. ISHFT (3, –1) has the value 1.

ISHFTC (I, SHIFT, SIZE)

Optional Argument. SIZE

Description. Performs a circular shift of the rightmost bits.

Class. Elemental function.

Arguments.

I — must be of type integer.

SHIFT	must be of type integer. The absolute value of SHIFT must be less than or equal to SIZE.
SIZE (optional)	must be of type integer. The value of SIZE must be positive and must not exceed BIT_SIZE (I). If SIZE is absent, it is as if it were present with the value of BIT_SIZE (I).

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by shifting the SIZE rightmost bits of I circularly by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if SHIFT is zero, no shift is performed. No bits are lost. The unshifted bits are unaltered. The model for the interpretation of an integer value as a sequence of bits is in topic 18.

Examples. ISHFTC (3, 2, 3) has the value 5. ISHFTC (3_SHORT, –2_SHORT) is 192 with kind SHORT.

KIND (X)

Description. Returns the value of the kind type parameter of X.

Class. Inquiry function.

Argument. X may be of any intrinsic type.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the kind type parameter value of X.

Examples. KIND (0.0) has the kind type parameter value of default real. KIND (1.0_HIGH) has the value of the named constant HIGH.

LBOUND (ARRAY, DIM)

Optional Argument. DIM

Description. Returns all the lower bounds or a specified lower bound of an array.

Class. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type default integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): For an array section or for an array expression other than a whole array or array structure component, LBOUND (ARRAY, DIM) has the value: 1. For a whole array or array structure component, LBOUND (ARRAY, DIM) has the value: a) equal to the lower bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have extent zero or if ARRAY is an assumed-size array of rank DIM, or (b) 1, otherwise.

Case (ii): LBOUND (ARRAY) has a value whose *i*th component is equal to LBOUND (ARRAY, i), for i = 1, 2, ..., n, where n is the rank of ARRAY.

Examples. If the following statements are processed

```
REAL, TARGET :: A (2:3, 7:10)
REAL, POINTER, DIMENSION (:,:) :: B, C, D
B => A
C => A(:,:)
ALLOCATE ( D(-3:3,-7:7) )
```

LBOUND (A) is (2, 7), LBOUND (A, DIM=2) is 7, LBOUND (B) is (2,7), LBOUND (C) is (1,1), LBOUND (D) is (–3,–7),

LEN (STRING)

Description. Returns the length of a character entity.

Class. Inquiry function.

Argument. STRING must be of type character. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the number of characters in STRING if it is scalar or in an element of STRING if it is array valued.

Example. If C and D are declared by the statements

```
CHARACTER (11) C (100)
CHARACTER (KIND=GREEK, LEN=31) D
```

LEN (C) has the value 11 and LEN (D) has the value 31.

LEN_TRIM (STRING)

Description. Returns the length of the character argument without counting trailing blank characters.

Class. Elemental function.

Argument. STRING must be of type character.

Result Type and Type Parameter. Default integer.

Result Value. The result has a value equal to the number of characters remaining after any trailing blanks in STRING are removed. If the argument contains no nonblank characters, the result is zero.

Examples. LEN_TRIM (' A B ') has the value 4 and LEN_TRIM (' ') has the value 0.

LGE (STRING_A, STRING_B)

Description. Test whether a string is lexically greater than or equal to another string, based on the ASCII collating sequence.

Class. Elemental function.

Arguments.

STRING_A	must be of type default character.
STRING_B	must be of type default character.

Result Type and Type Parameters. Default logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if the strings are equal or if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is true if both STRING_A and STRING_B are of zero length.

Example. LGE ('ONE', 'TWO') has the value false.

LGT (STRING_A, STRING_B)

Description. Test whether a string is lexically greater than another string, based on the ASCII collating sequence.

Class. Elemental function.

Arguments.

STRING_A	must be of type default character.
STRING_B	must be of type default character.

Result Type and Type Parameters. Default logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is false if both STRING_A and STRING_B are of zero length.

Example. LGT ('ONE', 'TWO') has the value false.

LLE (STRING_A, STRING_B)

Description. Test whether a string is lexically less than or equal to another string, based on the ASCII collating sequence.

Class. Elemental function.

Arguments.

STRING_A	must be of type default character.
STRING_B	must be of type default character.

Result Type and Type Parameters. Default logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if the strings are equal or if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is true if both STRING_A and STRING_B are of zero length.

Example. LLE ('ONE', 'TWO') has the value true.

LLT (STRING_A, STRING_B)

Description. Test whether a string is lexically less than another string, based on the ASCII collating sequence.

Class. Elemental function.

Arguments.

STRING_A	must be of type default character.
STRING_B	must be of type default character.

Result Type and Type Parameters. Default logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is false. Note that the result is false if both STRING_A and STRING_B are of zero length.

Example. LLT ('ONE', 'TWO') has the value true.

LOG (X)

Description. Natural logarithm.

Class. Elemental function.

Argument. X must be of type real or complex. If X is real, its value must be greater than zero. If X is complex, its value must not be zero.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to $\log_e x$. A result of type complex is the principal value with imaginary part ω in the range $-\pi < \omega \le \pi$. The imaginary part of the result is π only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

Examples. LOG (10.0) has the value 2.3025851. LOG ((–0.5_HIGH,0)) has the value –0.69314718055994 + 3.1415926535898ι with kind HIGH.

LOG10 (X)

Description. Common logarithm.

Class. Elemental function.

Argument. X must be of type real. The value of X must be greater than zero.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to $\log_{10} X$.

Examples. LOG10 (10.0) has the value 1.0. LOG10 (10.0E1000_HIGH) has the value 1001.0 with kind HIGH.

LOGICAL (L, KIND)

Optional Argument. KIND

Description. Converts between kinds of logical.

Class. Elemental function.

Arguments.

L	must be of type logical.
KIND (optional)	must be a scalar integer initialization expression.

Result Type and Type Parameter. Logical. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default logical.

Result Value. The value is that of L.

Examples. LOGICAL (L .OR. .NOT. L) has the value true and is of type default logical, regardless of the kind type parameter of the logical variable L. LOGICAL (L, BIT) has kind parameter BIT and has the same value as L.

MATMUL (MATRIX_A, MATRIX_B)

Description. Performs matrix multiplication of numeric or logical matrices.

Class. Transformational function.

Arguments.

MATRIX_A	must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one or two.
MATRIX_B	must be of numeric type if MATRIX_A is of numeric type and of logical type if MATRIX_A is of logical type. It must be array valued and of rank one or two. If MATRIX_A has rank one, MATRIX_B must have rank two. If MATRIX_B has rank one, MATRIX_A must have rank two. The size of the first (or only) dimension of MATRIX_B must equal the size of the last (or only) dimension of MATRIX_A.

Result Type, Type Parameter, and Shape. If the arguments are of numeric type, the type and kind type parameter of the result are determined by the types of the arguments according to 7.2.8.2. If the arguments are of type logical, the result is of type logical with the kind type parameter of the arguments according to 7.2.8.2. The shape of the result depends on the shapes of the arguments as follows:

Case (i): If MATRIX_A has shape (n, m) and MATRIX_B has shape (m, k), the result has shape (n, k).

Case (ii): If MATRIX_A has shape (m) and MATRIX_B has shape (m, k), the result has shape (k).

Case (iii): If MATRIX_A has shape (n, m) and MATRIX_B has shape m, the result has shape n.

Result Value.

Case (i): Element (i, j) of the result has the value SUM (MATRIX_A (i, :) * MATRIX_B (:, j)) if the arguments are of numeric type and has the value ANY (MATRIX_A (i, :) .AND. MATRIX_B (:, j)) if the arguments are of logical type.

Case (ii): Element (j) of the result has the value SUM (MATRIX_A (:) * MATRIX_B (:, j)) if the arguments are of numeric type and has the value ANY (MATRIX_A (:) .AND. MATRIX_B (:, j)) if the arguments are of logical type.

Case (iii): Element (i) of the result has the value SUM (MATRIX_A (i, :) * MATRIX_B (:)) if the arguments are of numeric type and has the value ANY (MATRIX_A (i, :) .AND. MATRIX_B (:)) if the arguments are of logical type.

Examples. Let A and B be the matrices $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}$ and $\begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}$; let X and Y be the vectors (1, 2) and (1, 2, 3).

Case (i): The result of MATMUL (A, B) is the matrix-matrix product AB with the value $\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$.

Case (ii): The result of MATMUL (X, A) is the vector-matrix product XA with the value (5, 8, 11).

Case (iii): The result of MATMUL (A, Y) is the matrix-vector product AY with the value (14, 20).

MAX (A1, A2, A3, ...)

Optional Arguments. A3, ...

Description. Maximum value.

Class. Elemental function.

Arguments. The arguments must all have the same type which must be integer or real and they must all have the same kind type parameter.

Result Type and Type Parameter. Same as the arguments.

Result Value. The value of the result is that of the largest argument.

Examples. MAX (–9.0, 7.0, 2.0) has the value 7.0. MAX (–1.0_HIGH/3, –0.1_HIGH) is –0.1 with kind HIGH.

MAXEXPONENT (X)

Description. Returns the maximum exponent in the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has the value e_{max}, as defined in topic 18 for the model representing numbers of the same type and kind type parameter as X.

Example. MAXEXPONENT (X) has the value 127 for real X whose model is described in topic 18.

MAXLOC (ARRAY, MASK) or MAXLOC (ARRAY, DIM, MASK)

Optional Argument. MASK

Description. Determine the location of the first element of ARRAY along dimension DIM having the maximum value of the elements identified by MASK.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
DIM	must be a scalar integer with value $1 \leq \text{DIM} < n$, where n is the rank of the array. The corresponding actual argument must not be an optional dummy argument.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of type default integer. If DIM is absent, it is an array of rank one and of size equal to the rank of ARRAY; otherwise, the result is of rank n–1 and shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, d_n), where (d_1, d_2, ..., d_n) is the shape of ARRAY.

Result Value.

Case (i): The result of MAXLOC(ARRAY) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the maximum value of all of the elements of ARRAY. The ith subscript returned lies in the range 1 to e_i, where e_i is the extent

of the ith dimension of ARRAY. If more than one element has the maximum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, the value of the result is processor dependent.

Case (ii): The result of MAXLOC(ARRAY, MASK=MASK) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value equals the maximum value of all such elements of ARRAY. The ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of ARRAY. If more than one such element has the maximum value, the element whose subscripts are returned is the first such element taken in array element order. If there are no such elements (that is, if ARRAY has size zero or every element of MASK has the value false), the value of the result is processor dependent.

Case (iii): If ARRAY has rank one, MAXLOC(ARRAY, DIM=DIM [, MASK=MASK]) is a scalar whose value is equal to that of the first element of MAXLOC(ARRAY [, MASK=MASK]). Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of the reault is equal to MAXLOC(ARRAY(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n), DIM=1 [, MASK=MASK(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)]).

Examples.

Case (i): The value of MAXLOC ((/ 2, 6, 4, 6 /)) is (2). If the array B is declared

```
INTEGER, DIMENSION(4:7) :: B = (/ 8, 6, 3, 1 /)
```

the value of MAXLOC (B) is (1).

Case (ii): If A has the value $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$, MAXLOC (A, MASK = A .LT. 6) has the value (3, 2). Note that this is true even if A has a declared lower bound other than 1.

Case (iii): The value of MAXLOC((/ 5, –9, 3 /), DIM=1) is 1. If B has the value $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$, MAXLOC(B, DIM=1) is (2, 1, 2) and MAXLOC(B, DIM=2) is (2,3). Note that this is true even if B has a declared lower bound other than 1.

MAXVAL (ARRAY, MASK) or MAXVAL (ARRAY, DIM, MASK)

Optional Arguments. MASK

Description. Maximum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY — must be of type integer or real. It must not be scalar.

DIM (optional) — must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$ where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

MASK (optional) — must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of MAXVAL (ARRAY) has a value equal to the maximum value of all the elements of ARRAY or has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if ARRAY has size zero.

Case (ii): The result of MAXVAL (ARRAY, MASK = MASK) has a value equal to the maximum value of the elements of ARRAY corresponding to true elements of MASK or has the value of the

negative number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if there are no true elements.

Case (iii): If ARRAY has rank one, MAXVAL (ARRAY, DIM+DIM [,MASK=MASK]) has a value equal to that of MAXVAL (ARRAY [,MASK = MASK]). Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of MAXVAL (ARRAY, DIM [,MASK]) is equal to MAXVAL (ARRAY (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) [, MASK = MASK (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)]).

Examples.

Case (i): The value of MAXVAL ((/ 1, 2, 3 /)) is 3.

Case (ii): MAXVAL (C, MASK = C .LT. 0.0) finds the maximum of the negative elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, MAXVAL (B, DIM = 1) is (2, 4, 6) and MAXVAL (B, DIM = 2) is (5, 6).

MERGE (TSOURCE, FSOURCE, MASK)

Description. Choose alternative value according to the value of a mask.

Class. Elemental function.

Arguments.

TSOURCE	may be of any type.
FSOURCE	must be of the same type and type parameters as TSOURCE.
MASK	must be of type logical.

Result Type and Type Parameters. Same as TSOURCE.

Result Value. The result is TSOURCE if MASK is true and FSOURCE otherwise.

Examples. If TSOURCE is the array $\begin{bmatrix} 1 & 6 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, FSOURCE is the array $\begin{bmatrix} 0 & 3 & 2 \\ 7 & 4 & 8 \end{bmatrix}$ and MASK is the array $\begin{bmatrix} T & . & T \\ . & . & T \end{bmatrix}$, where "T" represents true and "." represents false, then MERGE (TSOURCE, FSOURCE, MASK) is $\begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 6 \end{bmatrix}$. The value of MERGE (1.0, 0.0, K > 0) is 1.0 for K = 5 and 0.0 for K = –2.

MIN (A1, A2, A3, ...)

Optional Arguments. A3, ...

Description. Minimum value.

Class. Elemental function.

Arguments. The arguments must all be of the same type which must be integer or real and they must all have the same kind type parameter.

Result Type and Type Parameter. Same as the arguments.

Result Value. The value of the result is that of the smallest argument.

Examples. MIN (–9.0, 7.0, 2.0) has the value –9.0. MIN (–0.4_HIGH, –1.0_HIGH/3) is –0.4 with kind HIGH.

MINEXPONENT (X)

Description. Returns the minimum (most negative) exponent in the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has the value e_{min}, as defined in topic 18 for the model representing numbers of the same type and kind type parameter as X.

Example. MINEXPONENT (X) has the value –126 for real X whose model is described in topic 18.

MINLOC (ARRAY, MASK) or MINLOC (ARRAY, DIM, MASK)

Optional Argument. MASK

Description. Determine the location of the first element of ARRAY along dimension DIM having the minimum value of the elements identified by MASK.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
DIM	must be a scalar integer with value $1 \leq \text{DIM} < n$, where n is the rank of the array. The corresponding actual argument must not be an optional dummy argument.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of type default integer. If DIM is absent, it is an array of rank one and of size equal to the rank of ARRAY; otherwise, the result is of rank n–1 and shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of MINLOC(ARRAY) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the minimum value of all of the elements of ARRAY. The ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of ARRAY. If more than one element has the minimum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, the value of the result is processor dependent.

Case (ii): The result of MINLOC(ARRAY, MASK=MASK) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value equals the minimum value of all such elements of ARRAY. The ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of ARRAY. If more than one such element has the minimum value, the element whose subscripts are returned is the first such element taken in array element order. If there are no such elements (that is, if ARRAY has size zero or every element of MASK has the value false), the value of the result is processor dependent.

Case (iii): If ARRAY has rank one, MINLOC(ARRAY, DIM=DIM [, MASK=MASK]) is a scalar whose value is equal to that of the first element of MINLOC(ARRAY [, MASK=MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of the reault is equal to MINLOC(ARRAY($s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n$), DIM=1 [, MASK=MASK($s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n$)]).

Examples.

Case (i): The value of MINLOC ((/ 4, 3, 6, 3 /)) is (2). If the array B is declared

```
INTEGER, DIMENSION(4:7) :: B = (/ 8, 6, 3, 1 /)
```

the value of MINLOC (B) is (4).

Case (ii): If A has the value $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$, MINLOC (A, MASK = A .GT. –4) has the value (1, 4). Note that this is true even if A has a declared lower bound other than 1.

Case (iii): The value of MINLOC((/ 5, –9, 3 /), DIM=1) is 2. If B has the value $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$, MINLOC(B, DIM=1) is (1, 2, 1) and MINLOC(B, DIM=2) is (3, 1). Note that this is true even if B has a declared lower bound other than 1.

MINVAL (ARRAY, MASK) or MINVAL (ARRAY, DIM, MASK)

Optional Arguments. MASK

Description. Maximum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$ where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of MINVAL (ARRAY) has a value equal to the minimum value of all the elements of ARRAY or has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if ARRAY has size zero.

Case (ii): The result of MINVAL (ARRAY, MASK = MASK) has a value equal to the minimum value of the elements of ARRAY corresponding to true elements of MASK or has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if there are no true elements.

Case (iii): If ARRAY has rank one, MINVAL (ARRAY, DIM+DIM [,MASK=MASK]) has a value equal to that of MINVAL (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of MINVAL (ARRAY, DIM [,MASK]) is equal to MINVAL (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$ [, MASK = MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of MINVAL ((/ 1, 2, 3 /)) is 1.

Case (ii): MINVAL (C, MASK = C .GT. 0.0) finds the minimum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, MINVAL (B, DIM = 1) is (1, 3, 5) and MINVAL (B, DIM = 2) is (1, 2).

MOD (A, P)

Description. Remainder function.

Class. Elemental function.

Arguments.

A	must be of type integer or real.
P	must be of the same type and kind type parameter as A.

Result Type and Type Parameter. Same as A.

Result Value. If $P \neq 0$, the value of the result is A – INT (A / P) $*$ P. If $P = 0$, the result is processor dependent.

Examples. MOD (3.0, 2.0) has the value 1.0. MOD (8, 5) has the value 3. MOD (–8, 5) has the value –3. MOD (8, –5) has the value 3. MOD (–8, –5) has the value –3. MOD (2.0_HIGH, 3.0_HIGH) has the value 2.0 with kind HIGH.

MODULO (A, P)

Description. Modulo function.

Class. Elemental function.

Arguments.

A	must be of type integer or real.
P	must be of the same type and kind type parameter as A.

Result Type and Type Parameter. Same as A.

Result Value.

Case (i): A is of type integer. If $P \neq 0$, MODULO (A, P) has the value R such that $A = Q \times P + R$, where Q is an integer, the inequalities $0 \leq R < P$ hold if $P > 0$, and $P < R \leq 0$ hold if $P < 0$. If $P = 0$, the result is processor dependent.

Case (ii): A is of type real. If $P \neq 0$, the value of the result is A –FLOOR (A / P) $*$ P. If $P = 0$, the result is processor dependent.

Examples. MODULO (8, 5) has the value 3. MODULO (–8, 5) has the value 2. MODULO (8, –5) has the value –2. MODULO (–8, –5) has the value –3. MODULO (3.0, 2.0) has the value 1.0. MODULO (2.0_HIGH, 3.0_HIGH) has the value 2.0 with kind HIGH.

MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

Description. Copies a sequence of bits from one data object to another.

Class. Elemental subroutine.

Arguments.

FROM	must be of type integer. It is an INTENT (IN) argument.
FROMPOS	must be of type integer and nonnegative. It is an INTENT (IN) argument. FROMPOS + LEN must be less than or equal to BIT_SIZE (FROM). The model for the interpretation of an integer value as a sequence of bits is in topic 18.
LEN	must be of type integer and nonnegative. It is an INTENT (IN) argument.
TO	must be a variable of type integer with the same kind type parameter value as FROM and may be the same variable as FROM. It is an INTENT (INOUT) argument. TO is set by copying the sequence of bits of length LEN, starting at position FROMPOS of FROM to position TOPOS of TO. No other bits of TO are altered. On return, the LEN bits of TO starting at TOPOS are equal to the value that the LEN bits of FROM starting at FROMPOS had on entry. The model for the interpretation of an integer value as a sequence of bits is in topic 18.
TOPOS	must be of type integer and nonnegative. It is an INTENT (IN) argument. TOPOS + LEN must be less than or equal to BIT_SIZE (TO).

Examples. If TO has the initial value 6, the value of TO after the statement CALL MVBITS (7, 2, 2, TO, 0) is 5. After the statement

```
CALL MVBITS (PATTERN, 0_SHORT, 1_SHORT, PATTERN, 7_SHORT)
```

is executed, the integer variable PATTERN of kind SHORT has a leading bit that is identical to its terminal bit.

NEAREST (X, S)

Description. Returns the nearest different machine representable number in a given direction.

Class. Elemental function.

Arguments.

X	must be of type real.
S	must be of type real and not equal to zero.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to the machine representable number distinct from X and nearest to it in the direction of the infinity with the same sign as S.

Example. NEAREST (3.0, 2.0) has the value $3 + 2^{-22}$ on a machine whose representation is that of the model described in topic 18.

NINT (A, KIND)

Optional Argument. KIND

Description. Nearest integer.

Class. Elemental function.

Arguments.

A	must be of type real.
KIND (optional)	must be a scalar integer initialization expression.

Result Type and Type Parameter. Integer. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. If $A > 0$, NINT (A) has the value INT (A+0.5); if $A \le 0$, NINT (A) has the value INT (A–0.5). The result is undefined if the processor cannot represent the result in the specified integer type.

Examples. NINT (2.783) has the value 3. NINT (–1.99999999999_HIGH) has the value –2.

NOT (I)

Description. Performs a logical complement.

Class. Elemental function.

Argument. I must be of type integer.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by complementing I bit-by-bit according to the following truth table:

I	NOT (I)
1	0
0	1

The model for the interpretation of an integer value as a sequence of bits is in topic 18.

Example. If I is an integer of kind SHORT and has a value that is equal to 01010101 (base 2), NOT (I) has the value which is equal to 10101010 (base 2).

NULL (MOLD)

Optional Argument. MOLD

Description. Returns a disassociated pointer.

Class. Transformational function.

Argument. MOLD must be a pointer and may be of any type. Its pointer association status may be undefined, disassociated, or associated. If its status is associated, the target need not be defined with a value.

Result Characteristics. The same as MOLD if MOLD is present; otherwise, determined by context.

Result. The result is a pointer with disassociated association status.

Example. REAL, POINTER, DIMENSION(:) :: VEC => NULL(). This specification statement causes the initial association status of VEC to be disassociated.

PACK (ARRAY, MASK, VECTOR)

Optional Argument. VECTOR

Description. Pack an array into an array of rank one under the control of a mask.

Class. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
MASK	must be of type logical and must be conformable with ARRAY.
VECTOR (optional)	must be of the same type and type parameters as ARRAY and must have rank one. VECTOR must have at least as many elements as there are true elements in MASK. If MASK is scalar with the value true, VECTOR must have at least as many elements as there are in ARRAY.

Result Type, Type Parameter, and Shape. The result is an array of rank one with the same type and type parameters as ARRAY. If VECTOR is present, the result size is that of VECTOR; otherwise, the result size is the number t of true elements in MASK unless MASK is scalar with the value true, in which case the result size is the size of ARRAY.

Result Value. Element i of the result is the element of ARRAY that corresponds to the ith true element of MASK, taking elements in array element order, for $i = 1, 2, ..., t$. If VECTOR is present and has size $n > t$, element i of the result has the value VECTOR (i), for $i = t + 1, ..., n$.

Examples. The nonzero elements of an array M with the value $\begin{bmatrix} 0 & 0 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{bmatrix}$ may be "gathered" by the function PACK. The result of PACK (M, MASK = M .NE. 0) is (9, 7) and the result of PACK (M, M .NE. 0, VECTOR = (/ 2, 4, 6, 8, 10, 12 /)) is (9, 7, 6, 8, 10, 12).

PRECISION (X)

Description. Returns the decimal precision in the model representing real numbers with the same kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type real or complex. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has the value INT $((p - 1) * \text{LOG10}(b)) + k$, where b and p are as defined in topic 18 for the model representing real numbers with the same value for the kind type parameter as X, and where k is 1 if b is an integral power of 10 and 0 otherwise.

Example. PRECISION (X) has the value INT (23 ∗ LOG10 (2.)) = INT (6.92...) = 6 for real X whose model is described in topic 18.

PRESENT (A)

Description. Determine whether an optional argument is present.

Class. Inquiry function.

Argument. A must be the name of an optional dummy argument that is accessible in the procedure in which the PRESENT function reference appears.

Result Type and Type Parameters. Default logical scalar.

Result Value. The result has the value true if A is present (12.5.5) and otherwise has the value false.

Example.

```
SUBROUTINE SUB (A, B, EXTRA)
   REAL A, B, C
   REAL, OPTIONAL :: EXTRA
   . . .
   IF (PRESENT (EXTRA)) THEN
      C = EXTRA
```

```
   ELSE
      C = (A+B)/2
   END IF
   . . .
END
```

If SUB is called with the statement

```
CALL SUB (10.0, 20.0, 30.0)
```

C is set to 30.0. If SUB is called with the statement

```
CALL SUB (10.0, 20.0)
```

C is set to 15.0. An optional argument that is not present must not be referenced or defined or supplied as a nonoptional actual argument, except as the argument of the PRESENT intrinsic function.

PRODUCT (ARRAY, DIM, MASK)

Optional Arguments. DIM, MASK

Description. Product of all the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer, real, or complex. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of PRODUCT (ARRAY) has a value equal to a processor-dependent approximation to the product of all the elements of ARRAY or has the value one if ARRAY has size zero.

Case (ii): The result of PRODUCT (ARRAY, MASK = MASK) has a value equal to a processor-dependent approximation to the product of the elements of ARRAY corresponding to the true elements of MASK or has the value one if there are no true elements.

Case (iii): If ARRAY has rank one, PRODUCT (ARRAY, DIM [,MASK]) has a value equal to that of PRODUCT (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of PRODUCT (ARRAY, DIM [,MASK]) is equal to PRODUCT (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$ [, MASK = MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of PRODUCT ((/ 1, 2, 3 /)) and PRODUCT ((/ 1, 2, 3 /), DIM=1) is 6.

Case (ii): PRODUCT (C, MASK = C .GT. 0.0) forms the product of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, PRODUCT (B, DIM = 1) is (2, 12, 30) and PRODUCT (B, DIM = 2) is (15, 48).

RADIX (X)

Description. Returns the base of the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has the value r if X is of type integer and the value b if X is of type real, where r and b are as defined in topic 18 for the model representing numbers of the same type and kind type parameter as X.

Example. RADIX (X) has the value 2 for real X whose model is described in topic 18.

RANDOM_NUMBER (HARVEST)

Description. Returns one pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 \le x < 1$.

Class. Subroutine.

Argument. HARVEST must be of type real. It is an INTENT (OUT) argument. It may be a scalar or an array variable. It is set to contain pseudorandom numbers from the uniform distribution in the interval $0 \le x < 1$.

Examples.

```
REAL X, Y (10, 10)
! Initialize X with a pseudorandom number
CALL RANDOM_NUMBER (HARVEST = X)
CALL RANDOM_NUMBER (Y)
! X and Y contain uniformly distributed random numbers
```

RANDOM_SEED (SIZE, PUT, GET)

Optional Arguments. SIZE, PUT, GET

Description. Restarts or queries the pseudorandom number generator used by RANDOM_NUMBER.

Class. Subroutine.

Arguments. There must either be exactly one or no arguments present.

SIZE (optional)	must be scalar and of type default integer. It is an INTENT (OUT) argument. It is set to the number N of integers that the processor uses to hold the value of the seed.
PUT (optional)	must be a default integer array of rank one and size $\ge N$. It is an INTENT (IN) argument. It is used by the processor to set the seed value.
GET (optional)	must be a default integer array of rank one and size $\ge N$. It is an INTENT (OUT) argument. It is set by the processor to the current value of the seed. If no argument is present, the processor sets the seed to a processor-dependent value.

Examples.

```
CALL RANDOM_SEED                     ! Processor initialization
CALL RANDOM_SEED (SIZE = K)          ! Sets K = N
CALL RANDOM_SEED (PUT = SEED (1 : K)) ! Set user seed
CALL RANDOM_SEED (GET = OLD (1 : K))  ! Read current seed
```

RANGE (X)

Description. Returns the decimal exponent range in the model representing integer or real numbers with the same kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type integer, real, or complex. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value.

Case (i): For an integer argument, the result has the value INT (LOG10 ($huge$)), where $huge$ is the largest positive integer in the model representing integer numbers with same kind type parameter as X (topic 18).

Case (ii): For a real or complex argument, the result has the value INT (MIN (LOG10 ($huge$), – LOG10 ($tiny$))), where $huge$ and $tiny$ are the largest and smallest positive numbers in the

model representing real numbers with the same value for the kind type parameter as X (topic 18).

Example. RANGE (X) has the value 38 for real X whose model is described in topic 18, because in this case $huge = (1 - 2^{-24}) \times 2^{127}$ and $tiny = 2^{-127}$.

REAL (A, KIND)

Optional Argument. KIND

Description. Convert to real type.

Class. Elemental function.

Arguments.

A	must be of type integer, real, or complex.
KIND (optional)	must be a scalar integer initialization expression.

Result Type and Type Parameter. Real.

Case (i): If A is of type integer or real and KIND is present, the kind type parameter is that specified by KIND. If A is of type integer or real and KIND is not present, the kind type parameter is the processor-dependent kind type parameter for the default real type.

Case (ii): If A is of type complex and KIND is present, the kind type parameter is that specified by KIND. If A is of type complex and KIND is not present, the kind type parameter is the kind type parameter of A.

Result Value.

Case (i): If A is of type integer or real, the result is equal to a processor-dependent approximation to A.

Case (ii): If A is of type complex, the result is equal to a processor-dependent approximation to the real part of A.

Examples. REAL (–3) has the value –3.0. REAL (Z) has the same kind type parameter and the same value as the real part of the complex variable Z. REAL (2.0_HIGH/3.0) is 0.66666666666666 with kind HIGH.

REPEAT (STRING, NCOPIES)

Description. Concatenate several copies of a string.

Class. Transformational function.

Arguments.

STRING	must be scalar and of type character.
NCOPIES	must be scalar and of type integer. Its value must not be negative.

Result Type, Type Parameter, and Shape. Character scalar of length NCOPIES times that of STRING, with the same kind type parameter as STRING.

Result Value. The value of the result is the concatenation of NCOPIES copies of STRING.

Examples. REPEAT ('H', 2) has the value HH. REPEAT ('XYZ', 0) has the value of a zero-length string.

RESHAPE (SOURCE, SHAPE, PAD, ORDER)

Optional Arguments. PAD, ORDER

Description. Constructs an array of a specified shape from the elements of a given array.

Class. Transformational function.

Arguments.

SOURCE	may be of any type. It must be array valued. If PAD is absent or of size zero, the size of SOURCE must be greater than or equal to PRODUCT (SHAPE). The size of the result is the product of the values of the elements of SHAPE.
SHAPE	must be of type integer, rank one, and constant size. Its size must be positive and less than 8. It must not have an element whose value is negative.
PAD (optional)	must be of the same type and type parameters as SOURCE. PAD must be array valued.

ORDER (optional)	must be of type integer, must have the same shape as SHAPE, and its value must be a permutation of (1, 2, ..., n), where n is the size of SHAPE. If absent, it is as if it were present with value (1, 2, ..., n).

Result Type, Type Parameter, and Shape. The result is an array of shape SHAPE (that is, SHAPE (RESHAPE (SOURCE, SHAPE, PAD, ORDER)) is equal to SHAPE) with the same type and type parameters as SOURCE.

Result Value. The elements of the result, taken in permuted subscript order ORDER (1), ..., ORDER (n), are those of SOURCE in normal array element order followed if necessary by those of PAD in array element order, followed if necessary by additional copies of PAD in array element order.

Examples. RESHAPE ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /)) has the value $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$. RESHAPE ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 4 /), (/ 0, 0 /), (/ 2, 1 /)) has the value $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$.

RRSPACING (X)

Description. Returns the reciprocal of the relative spacing of model numbers near the argument value.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has the value $|X \times b^{-e}| \times b^p$, where b, e, and p are as defined in topic 18 for the model representation of X.

Example. RRSPACING (–3.0) has the value 0.75×2^{24} for reals whose model is described in topic 18.

SCALE (X, I)

Description. Returns $X \times b^I$ where b is the base in the model representation of X.

Class. Elemental function.

Arguments.

X	must be of type real.
I	must be of type integer.

Result Type and Type Parameter. Same as X.

Result Value. The result has the value $X \times b^I$, where b is defined in topic 18 for model numbers representing values of X, provided this result is within range; if not, the result is processor dependent.

Example. SCALE (3.0, 2) has the value 12.0 for reals whose model is described in topic 18.

SCAN (STRING, SET, BACK)

Optional Argument. BACK

Description. Scan a string for any one of the characters in a set of characters.

Class. Elemental function.

Arguments.

STRING	must be of type character.
SET	must be of type character with the same kind type parameter as STRING.
BACK (optional)	must be of type logical.

Result Type and Type Parameter. Default integer.

Result Value.

Case (i): If BACK is absent or is present with the value false and if STRING contains at least one character that is in SET, the value of the result is the position of the leftmost character of STRING that is in SET.

Case (ii): If BACK is present with the value true and if STRING contains at least one character that is in SET, the value of the result is the position of the rightmost character of STRING that is in SET.

Case (iii): The value of the result is zero if no character of STRING is in SET or if the length of STRING or SET is zero.

Examples.

Case (i): SCAN ('FORTRAN', 'TR') has the value 3.

Case (ii): SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.

Case (iii): SCAN ('FORTRAN', 'BCD') has the value 0.

SELECTED_INT_KIND (R)

Description. Returns a value of the kind type parameter of an integer data type that represents all integer values n with $-10^R < n < 10^R$.

Class. Transformational function.

Argument. R must be scalar and of type integer.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the value of the kind type parameter of an integer data type that represents all values n in the range of values n with $-10^R < n < 10^R$, or if no such kind type parameter is available on the processor, the result is –1. If more than one kind type parameter meets the criteria, the value returned is the one with the smallest decimal exponent range, unless there are several such values, in which case the smallest of these kind values is returned.

Examples. SELECTED_INT_KIND (6) has the value KIND (0) on a machine that supports a default integer representation method with $r = 2$ and $q = 31$ as defined in the model for the integer number systems in topic 18. SELECTED_INT_KIND (2) has the value of SHORT on a machine that supports this integer kind.

SELECTED_REAL_KIND (P, R)

Optional Arguments. P, R

Description. Returns a value of the kind type parameter of a real data type with decimal precision of at least P digits and a decimal exponent range of at least R.

Class. Transformational function.

Arguments. At least one argument must be present.

P (optional)	must be scalar and of type integer.
R (optional)	must be scalar and of type integer.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to a value of the kind type parameter of a real data type with decimal precision, as returned by the function PRECISION, of at least P digits and a decimal exponent range, as returned by the function RANGE, of at least R, or if no such kind type parameter is available on the processor, the result is –1 if the precision is not available, –2 if the exponent range is not available, and –3 if neither is available. If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision, unless there are several such values, in which case the smallest of these kind values is returned.

Examples. SELECTED_REAL_KIND (6, 70) has the value KIND (0.0) on a machine that supports a default real approximation method with $p = 16$, $p = 6$, $e_{min} = -64$, and $e_{max} = 63$ as defined in the model for the real number system in topic 18. SELECTED_REAL_KIND (P=14) returns the value of HIGH on a machine that supports this real kind.

SET_EXPONENT (X, I)

Description. Returns the model number whose fractional part is the fractional part of the model representation of X and whose exponent part is I.

Class. Elemental function.

Arguments.

X	must be of type real.
I	must be of type integer.

Result Type and Type Parameter. Same as X.

Result Value. The result has the value $X \times b^{I-e}$, where b and e are as defined in topic 18 for the model representation of X, provided this result is within range; if not, the result is processor dependent. If X has value zero, the result has value zero.

Example. SET_EXPONENT (3.0, 1) has the value 1.5 for reals whose model is as described in topic 18.

SHAPE (SOURCE)

Description. Returns the shape of an array or a scalar.

Class. Inquiry function.

Argument. SOURCE may be of any type. It may be array valued or scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. It must not be an assumed-size array.

Result Type, Type Parameter, and Shape. The result is a default integer array of rank one whose size is equal to the rank of SOURCE.

Result Value. The value of the result is the shape of SOURCE.

Examples. The value of SHAPE (A (2:5, –1:1)) is (4, 3). The value of SHAPE (3) is the rank-one array of size zero.

SIGN (A, B)

Description. Absolute value of A times the sign of B.

Class. Elemental function.

Arguments.

A	must be of type integer or real.
B	must be of the same type and kind type parameter as A.

Result Type and Type Parameter. Same as A.

Result Value. The value of the result is $|A|$ if $B \geq 0$ and $-|A|$ if $B < 0$.

Example. SIGN (–3.0, 2.0) has the value 3.0.

SIN (X)

Description. Sine function.

Class. Elemental function.

Argument. X must be of type real or complex.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to sin(X). If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

Examples. SIN (1.0) has the value 0.84147098. SIN ((0.5_HIGH, 0.5)) has the value 0.54061268571316 + 0.45730415318425ı with kind HIGH.

SINH (X)

Description. Hyperbolic sine function.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to sinh(X).

Examples. SINH (1.0) has the value 1.1752012. SINH (0.5_HIGH) has the value 0.52109530549375 with kind HIGH.

SIZE (ARRAY, DIM)

Optional Argument. DIM

Description. Returns the extent of an array along a specified dimension or the total number of elements in the array.

Class. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. If ARRAY is an assumed-size array, DIM must be present with a value less than the rank of ARRAY.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the extent of dimension DIM of ARRAY or, if DIM is absent, the total number of elements of ARRAY.

Examples. The value of SIZE (A (2:5, –1:1), DIM=2) is 3. The value of SIZE (A (2:5, –1:1)) is 12.

SPACING (X)

Description. Returns the absolute spacing of model numbers near the argument value.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. If X is not zero, the result has the value b^{e-p}, where b, e, and p are as defined in topic 18 for the model representation of X, provided this result is within range; otherwise, the result is the same as that of TINY (X).

Example. SPACING (3.0) has the value 2^{-22} for reals whose model is described in topic 18.

SPREAD (SOURCE, DIM, NCOPIES)

Description. Replicates an array by adding a dimension. Broadcasts several copies of SOURCE along a specified dimension (as in forming a book from copies of a single page) and thus forms an array of rank one greater.

Class. Transformational function.

Arguments.

SOURCE	may be of any type. It may be scalar or array valued. The rank of SOURCE must be less than 7.
DIM	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n+1$, where n is the rank of SOURCE.
NCOPIES	must be scalar and of type integer.

Result Type, Type Parameter, and Shape. The result is an array of the same type and type parameters as SOURCE and of rank $n+1$, where n is the rank of SOURCE.

Case (i): If SOURCE is scalar, the shape of the result is (MAX (NCOPIES, 0)).

Case (ii): If SOURCE is array valued with shape (d_1, d_2, ..., d_n), the shape of the result is (d_1, d_2, ..., $d_{\text{DIM}-1}$, MAX (NCOPIES, 0), d_{DIM}, ..., d_n).

Result Value.

Case (i): If SOURCE is scalar, each element of the result has a value equal to SOURCE.

Case (ii): If SOURCE is array valued, the element of the result with subscripts (r_1, r_2, ..., r_{n+1}) has the value SOURCE (r_1, r_2, ..., $r_{\text{DIM}-1}$, $r_{\text{DIM}+1}$, ..., r_{n+1}).

Examples.

Case (i): SPREAD ("A", 1, 3) is the character array (/ "A", "A", "A" /).

Case (ii): If A is the array (2, 3, 4), SPREAD (A, DIM=1, NCOPIES=NC) is the array $\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$ if NC has the value 3 and is a zero-sized array if NC has the value 0.

SQRT (X)

Description. Square root.

Class. Elemental function.

Argument. X must be of type real or complex. Unless X is complex, its value must be greater than or equal to zero.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the square root of X. A result of type complex is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

Examples. SQRT (4.0) has the value 2.0. SQRT (5.0_HIGH) has the value 2.23606774998 with kind HIGH.

SUM (ARRAY, DIM, MASK)

Optional Arguments. DIM, MASK

Description. Sum all the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY — must be of type integer, real, or complex. It must not be scalar.

DIM (optional) — must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

MASK (optional) — must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of SUM (ARRAY) has a value equal to a processor-dependent approximation to the sum of all the elements of ARRAY or has the value zero if ARRAY has size zero.

Case (ii): The result of SUM (ARRAY, MASK = MASK) has a value equal to a processor-dependent approximation to the sum of the elements of ARRAY corresponding to the true elements of MASK or has the value zero if there are no true elements.

Case (iii): If ARRAY has rank one, SUM (ARRAY, DIM [,MASK]) has a value equal to that of SUM (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of SUM (ARRAY, DIM [,MASK]) is equal to SUM (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$ [, MASK= MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of SUM ((/ 1, 2, 3 /)) and SUM ((/ 1, 2, 3 /), DIM=1) is 6.

Case (ii): SUM (C, MASK= C .GT. 0.0) forms the arithmetic sum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, SUM (B, DIM = 1) is (3, 7, 11) and SUM (B, DIM = 2) is (9, 12).

SYSTEM_CLOCK (COUNT, COUNT_RATE, COUNT_MAX)

Optional Arguments. COUNT, COUNT_RATE, COUNT_MAX

Description. Returns integer data from a real-time clock.

Class. Subroutine.

Arguments.

COUNT (optional) must be scalar and of type default integer. It is an INTENT (OUT) argument. It is set to a processor-dependent value based on the current value of the processor clock or to –HUGE (0) if there is no clock. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX is reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.

COUNT_RATE (optional) must be scalar and of type default integer. It is an INTENT (OUT) argument. It is set to the number of processor clock counts per second, or to zero if there is no clock.

COUNT_MAX (optional) must be scalar and of type default integer. It is an INTENT (OUT) argument. It is set to the maximum value that COUNT can have, or to zero if there is no clock.

Example. If the processor clock is a 24-hour clock that registers time in 1-second intervals, at 11:30 A.M. the reference

```
CALL SYSTEM_CLOCK (COUNT = C, COUNT_RATE = R, COUNT_MAX = M)
```

sets C = 11 × 3600 + 30 × 60 = 41400, R = 1, and M = 24 × 3600 – 1 = 86399.

TAN (X)

Description. Tangent function.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to tan(X), with X regarded as a value in radians.

Examples. TAN (1.0) has the value 1.5574077. TAN (2.0_HIGH) has the value –2.1850398632615 with kind HIGH.

TANH (X)

Description. Hyperbolic tangent function.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to tanh(X).

Examples. TANH (1.0) has the value 0.76159416. TANH (2.0_HIGH) has the value 0.96402758007582 with kind HIGH.

TINY (X)

Description. Returns the smallest positive number in the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Scalar with the same type and kind type parameter as X.

Result Value. The result has the value $b^{e_{min} - 1}$ where b and e_{min} are as defined in topic 18 for the model representing numbers of the same type and kind type parameter as X.

Example. TINY (X) has the value 2^{-127} for real X whose model is described in topic 18.

TRANSFER (SOURCE, MOLD, SIZE)

Optional Argument. SIZE

Description. Returns a result with a physical representation identical to that of SOURCE but interpreted with the type and type parameters of MOLD.

Class. Transformational function.

Arguments.

SOURCE	may be of any type and may be scalar or array valued.
MOLD	may be of any type and may be scalar or array valued.
SIZE (optional)	must be scalar and of type integer. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of the same type and type parameters as MOLD.

Case (i): If MOLD is a scalar and SIZE is absent, the result is a scalar.

Case (ii): If MOLD is array valued and SIZE is absent, the result is array valued and of rank one. Its size is as small as possible such that its physical representation is not shorter than that of SOURCE.

Case (iii): If SIZE is present, the result is array valued of rank one and size SIZE.

Result Value. If the physical representation of the result has the same length as that of SOURCE, the physical representation of the result is that of SOURCE. If the physical representation of the result is longer than that of SOURCE, the physical representation of the leading part is that of SOURCE and the remainder is undefined. If the physical representation of the result is shorter than that of SOURCE, the physical representation of the result is the leading part of SOURCE. If D and E are scalar variables such that the physical representation of D is as long as or longer than that of E, the value of TRANSFER (TRANSFER (E, D), E) must be the value of E. If D is an array and E is an array of rank one, the value of TRANSFER (TRANSFER (E, D), E, SIZE (E)) must be the value of E.

Examples.

Case (i): TRANSFER (1082130432, 0.0) has the value 4.0 on a processor that represents the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000.

Case (ii): TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /)) is a complex rank-one array of length two whose first element is (1.1, 2.2) and whose second element has a real part with the value 3.3. The imaginary part of the second element is undefined.

Case (iii): TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /), 1) has the value 1.1 + 2.2ι, which is a rank-one array with one complex element.

TRANSPOSE (MATRIX)

Description. Transpose an array of rank two.

Class. Transformational function.

Argument. MATRIX may be of any type and must have rank two.

Result Type,Type Parameters, and Shape. The result is an array of the same type and type parameters as MATRIX and with rank two and shape (n, m) where (m, n) is the shape of MATRIX.

Result Value. Element (i, j) of the result has the value MATRIX (j, i), $i = 1, 2, ..., n$; $j = 1, 2, ..., m$.

Example. If A is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, then TRANSPOSE (A) has the value $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$.

TRIM (STRING)

Description. Returns the argument with trailing blank characters removed.

Class. Transformational function.

Argument. STRING must be of type character and must be a scalar.

Result Type and Type Parameters. Character with the same kind type parameter value as STRING and with a length that is the length of STRING less the number of trailing blanks in STRING.

Result Value. The value of the result is the same as STRING except any trailing blanks are removed. If STRING contains no nonblank characters, the result has zero length.

Examples. TRIM (′ A B ′) is ′ A B′. TRIM (GREEK_′ Π ′) is GREEK_′ Π′.

UBOUND (ARRAY, DIM)

Optional Argument. DIM

Description. Returns all the upper bounds of an array or a specified upper bound.

Class. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. If ARRAY is an assumed-size array, DIM must be present with a value less than the rank of ARRAY.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type default integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): For an array section or for an array expression, other than a whole array or array structure component, UBOUND (ARRAY, DIM) has a value equal to the number of elements in the given dimension; otherwise, it has a value equal to the upper bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero.

Case (ii): UBOUND (ARRAY) has a value whose *i*th component is equal to UBOUND (ARRAY, i), for $i = 1, 2, ..., n$, where n is the rank of ARRAY.

Examples. If the following statements are processed

```
REAL, TARGET :: A (2:3, 7:10)
REAL, POINTER, DIMENSION (:,:) :: B, C, D
B => A;  C => A(:,:)
ALLOCATE (D(-3:3,-7:7))
```

UBOUND (A) is (3, 10), UBOUND (A, DIM = 2) is 10, UBOUND (B) is (3, 10), UBOUND (C) is (2, 4), and UBOUND (D) is (3, 7); see Section 7.5.3, rules and restrictions, item 9.

UNPACK (VECTOR, MASK, FIELD)

Description. Unpack an array of rank one into an array under the control of a mask.

Class. Transformational function.

Arguments.

VECTOR	may be of any type. It must have rank one. Its size must be at least t where t is the number of true elements in MASK.
MASK	must be array valued and of type logical.
FIELD	must be of the same type and type parameters as VECTOR and must be conformable with MASK.

Result Type, Type Parameter, and Shape. The result is an array of the same type and type parameters as VECTOR and the same shape as MASK.

Result Value. The element of the result that corresponds to the *i*th true element of MASK, in array element order, has the value VECTOR (i) for $i = 1, 2, \ldots, t$, where t is the number of true values in MASK. Each other element has a value equal to FIELD if FIELD is scalar or to the corresponding element of FIELD if it is an array.

Examples. Specific values may be "scattered" to specific positions in an array by using UNPACK. If M is the array $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, V is the array (1, 2, 3),

and Q is the logical mask $\begin{bmatrix} . & T & . \\ T & . & . \\ . & . & T \end{bmatrix}$, where "T" represents true and "." represents false, then the result of

UNPACK (V, MASK = Q, FIELD = M) has the value $\begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ and the result of UNPACK (V, MASK = Q,

FIELD = 0) has the value $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

VERIFY (STRING, SET, BACK)

Optional Argument. BACK

Description. Verify that a set of characters contains all the characters in a string by identifying the position of the first character in a string of characters that does not appear in a given set of characters.

Class. Elemental function.

Arguments.

STRING	must be of type character.
SET	must be of type character with the same kind type parameter as STRING.
BACK (optional)	must be of type logical.

Result Type and Type Parameter. Default integer.

Result Value.

Case (i): If BACK is absent or present with the value false and if STRING contains at least one character that is not in SET, the value of the result is the position of the leftmost character of STRING that is not in SET.

Case (ii): If BACK is present with the value true and if STRING contains at least one character that is not in SET, the value of the result is the position of the rightmost character of STRING that is not in SET.

Case (iii): The value of the result is zero if each character in STRING is in SET or if STRING has zero length.

Examples.

Case (i): VERIFY ('ABBA', 'A') has the value 2.

Case (ii): VERIFY ('ABBA', 'A', BACK = .TRUE.) has the value 3.

Case (iii): VERIFY ('ABBA', 'AB') has the value 0.

Index

Symbols

! 179
– 33, 97, 171
% 47
& 179
* 33, 97, 171
** 33, 97, 171
+ 33, 97, 171
.AND. operator 123
.EQ. 33, 97, 171
.EQV. operator 123
.FALSE. 123
.GE. 97, 171
.GT. 97, 171
.LE. 97, 171
.LT. 97, 171
.NE. 33, 97, 171
.NEQV. operator 123
.NOT. operator 123
.OR. operator 123
.TRUE. 123
/ 33, 97, 171
/ edit descriptor 57
// 26, 27
/= 33, 97, 171
: edit descriptor 57
; 179
< 97, 171
<= 97, 171
== 33, 97, 171
> 97, 171
>= 97, 171

A

A edit descriptor 59
ABS function 110, 193
access
    direct 73, 152
    id 149
    in USE statement 187
    sequential 73, 74, 152
ACCESS= specifier 95, 131, 155
accessibility
    attribute 149
    statement 148, 149
ACHAR function 113, 193
ACOS function 110, 193
ACTION= specifier 95, 131
actual argument 6, 7, 8, 9, 99, 107, 132, 183
    function 81
ADJUSTL function 111, 194
ADJUSTR function 111, 194
ADVANCE= specifier 153, 166
advancing input/output 74, 164, 165
AIMAG function 113, 194
AINT function 113, 194
aliasing 136
ALL function 109, 194
allocatable array 2, 3, 4, 5, 54
ALLOCATABLE attribute 2, 11, 17, 55
ALLOCATABLE statement 2
allocate object 5
ALLOCATE statement 2, 3, 4, 136, 137, 138, 140
allocated array 55
ALLOCATED function 5, 55, 115, 195
allocation 5
    status 174
alternate return 7, 9, 85, 183
ampersand (&) 179
ANINT function 113, 195
ANY function 109, 196
apostrophe edit descriptor 59
approximation method 170
argument
    actual 6, 7, 8, 9, 99, 107, 132, 183
    association 6, 7, 43, 177, 183
    dummy 6, 7, 8, 9, 99, 133, 183
    function 81
    keyword 7, 8, 9
    optional 9, 107, 132
    positional 8
    presence 132, 133
    procedure 129
arithmetic operator 33, 97, 171
array 10, 50, 189
    allocatable 2, 3, 4, 5, 54

allocated 55
assumed-shape 11, 16, 17
assumed-size 7, 16, 17, 21
automatic 11
bound 2, 17, 68
component 43
constructor 11, 12, 13, 67, 69
deferred-shape 16, 17
dynamic 11
element 7, 10, 11, 67
element order 11
explicit-shape 7, 16, 17
intrinsic function 108
masked assignment 190
parent 18, 19
rank 19
section 11, 18, 19, 67
shape 191
specification 16, 51
specifier 16, 17
unallocated 55
whole 18
array section
many-to-one 19
array-valued function 15
ASCII collating sequence 193, 199, 207, 208, 212, 213
ASIN function 110, 196
assignment
defined 20, 39
defined type 44
intrinsic 20, 21, 35
masked array 20, 190
pointer 20, 55, 136, 139
specification 39
structure 44
user-defined 38
assignment statement 10, 20, 191
ASSOCIATED function 5, 55, 114, 136, 137, 138, 141, 143, 196
associated pointer 5, 143
association 176
argument 6, 7, 43, 177, 183
host 43, 45, 86, 87, 103, 127, 129, 183
pointer 138, 139, 141
sequence 6, 7, 183
status 5, 174
storage 30, 31, 62, 180, 183
use 45, 49, 87, 128, 148, 183, 186
assumed-shape array 11, 16, 17
assumed-size array 7, 16, 17, 21
asterisk (*) 183
ATAN function 110, 197
ATAN2 function 110, 197
attribute
accessibility 149
ALLOCATABLE 2, 11, 17, 55
DIMENSION 10, 50, 51
EXTERNAL 70, 71, 107
INTENT 98, 99
INTRINSIC 104, 105, 107
OPTIONAL 133
PARAMETER 134, 135
POINTER 4, 7, 11, 17, 20, 21, 43, 46, 47, 55, 65, 136, 137, 138, 139, 140, 141, 143
PRIVATE 43, 127, 148, 149
PUBLIC 43, 127, 148, 149
SAVE 3, 11, 34, 55, 125, 127, 174, 175
TARGET 136, 137, 139, 141, 184, 185
automatic array 11
automatic character object 27
automatic data object 3, 54

B

B edit descriptor 59
BACKSPACE statement 72, 73, 74, 75
binary
constant 97
operator 39, 65
bit model 36, 37
BIT_SIZE function 36, 114, 198
blank
character 179
interpretation of 56
padding 212, 213
blank common 31
blank padding 21

BLANK= specifier 95, 131
block 52
    common 30, 31, 63, 174, 175, 180, 181
    interface 9, 38, 70, 82, 83, 87, 100, 101, 127, 147
block data program unit 70, 146, 147
BN edit descriptor 57
body
    interface 87, 101
bound
    array 2, 17, 68
    lower 10, 17
    upper 10, 17
BOZ constant 35
BOZ edit descriptor 59
branch target statement 85
BTEST function 111, 198
BZ edit descriptor 57

C

CALL statement 183
case
    expression 23
    value 23
CASE construct 22, 23, 67
CASE statement 23
CEILING function 113, 198
CHAR function 113, 199
character
    automatic object 27
    blank 179
    constant 26, 27
    default 27
    kind 27
    operator 26, 27
    sequence structure 181
    set
        default 27
        Fortran 27
    storage unit 27
    substring 24
    type 26
    variable 159
CHARACTER statement 27
character string edit descriptor 58, 59
clause
    ONLY 186, 187
    RESULT 173
CLOSE statement 28, 29, 72
CMPLX function 113, 199
collating sequence 25, 27, 193, 199, 207
    ASCII 193, 199, 207, 208, 212, 213
colon edit descriptor 57
comment 179
    namelist 163
common block 30, 31, 63, 174, 175, 180, 181
    named 31
common block object 31
COMMON statement 31, 62, 63
comon
    blank 31
complex
    constant 33
    default 33
    kind 33, 119
    operator 33
complex number 32
COMPLEX statement 33
complex type 32
component
    array 43
    declaration 43
    name 43
    structure 46, 47, 67
        reference 47
computation intrinsic function 110, 144
concatenation 26, 27
condition
    end-of-file 72, 159, 167
    end-of-record 73, 167
conformable 10, 15
CONJG function 113, 199
connected file 73
connected unit 73
connection specifier 131
constant
    binary 97

BOZ 35
character 26, 27
complex 33
decimal 97
hexadecimal 97
integer 119
literal 67, 69
logical 123
named 45, 49, 67, 69, 134
octal 97
real 171
constant specification expression 31
construct
CASE 22, 23, 67
control 53
DO 52, 53
FORALL 11
IF 88, 89
name 23, 52, 53, 77, 89, 179, 191
WHERE 11, 190, 191
constructor
array 11, 12, 13, 67, 69
structure 44, 45, 48, 49, 67, 69
CONTAINS statement 125, 129
CONTINUE statement 53
continued statement 179
control construct 53
control edit descriptor 56, 57
conversion intrinsic function 67, 112
COS function 110, 199
COSH function 110, 200
COUNT function 109, 200
CPU_TIME subroutine 117, 200
CSHIFT function 109, 201
CYCLE statement 52, 53

D

D edit descriptor 59
data
edit descriptor 58, 59
initialization 34
object 35
automatic 3, 54
parallelism 14
record 72
representation model 36
value 35
DATA statement 13, 34, 35, 67, 97
data transfer statement 153, 155, 157, 158, 159, 160, 161, 163, 165, 167, 169
data-implied DO 35
data-implied DO object 35
DATE_AND_TIME subroutine 117, 202
DBLE function 113, 203
DEALLOCATE statement 3, 4, 55, 136, 137, 138, 143
decimal constant 97
declaration
component 43
defined type 45
type 71, 99, 105, 133, 134, 135, 141, 149, 175, 185, 188
default character 27
default character set 27
default complex 33
default integer 96
DEFAULT keyword 23
default kind 119
default logical 123
default real 170, 171
deferred-shape array 16, 17
defined assignment 20, 39
defined operator 39
defined type 7, 43, 45, 83
assignment 44
declaration 45
definition 43, 86
input/output 44
object 44
defined-type statement 149
definition 176
defined type 43, 86
function 80
pointer 141
procedure 6
status 174
type 43
DELIM= specifier 95, 131
derived type 42
digit 97

DIGITS function 36, 37, 115, 203
DIM function 110, 203
DIMENSION attribute 10, 50, 51
DIMENSION statement 51
direct access 73, 152
direct access input/output 154, 155, 156, 157
DIRECT= specifier 95
disassociated pointer 143
disassociation
  pointer 138, 139
DO
  data-implied 35
DO construct 52, 53
DO statement 53
DOT_PRODUCT function 110, 203
double precision
  real 170, 171
DOUBLE PRECISION statement 171
DPROD function 110, 204
dummy argument 6, 7, 8, 9, 99, 133, 183
dummy procedure 7, 9, 70, 71
dynamic array 11
dynamic object 54

E

E edit descriptor 59
edit descriptor
  / 57
  : 57
  A 59
  apostrophe 59
  B 59
  BN 57
  BOZ 59
  BZ 57
  character string 58, 59
  colon 57
  control 56, 57
  D 59
  data 58, 59
  E 59
  EN 59
  ES 59
  F 59
  G 59
  I 59
  L 59
  O 59
  P 57
  quote 59
  S 57
  slash 57
  SP 57
  SS 57
  T 57
  TL 57
  TR 57
  X 57
  Z 59
element
  array 7, 10, 11, 67
ELEMENTAL
  keyword 183
elemental function 11, 15
elemental intrinsic function 67, 69, 107, 190
elemental operation 191
elemental procedure 7
ELSE IF statement 89
ELSE statement 89
ELSEWHERE statement 191
EN edit descriptor 59
END DO statement 53
END IF statement 89
END INTERFACE statement 101
END SELECT statement 23
END statement 81, 129, 147, 183
END WHERE statement 191
END= specifier 153, 158, 160, 162, 164, 166, 168
ENDFILE statement 72, 73, 74, 75
ending position
  substring 25
end-of-file condition 72, 159, 167
end-of-file record 72
end-of-record condition 73, 167
ENTRY statement 132, 180, 181, 183
EOR= specifier 153, 166
EOSHIFT function 109, 204

EPSILON function 36, 115, 145, 205
equivalence 180
  group 181
  object 63
EQUIVALENCE statement 62, 63, 67
ERR= specifier 29, 95, 131, 153, 154, 156, 160, 164, 166, 168
ES edit descriptor 59
exclamation mark (!) 179
executable subprogram 147
execution
  part 125
  program 147
EXIST= specifier 95
existence of a file 72
EXIT statement 53
EXP function 110, 206
explicit format 155, 159, 167
explicit interface 100
explicit-shape array 7, 16, 17
exponent 171
  letter 171
EXPONENT function 36, 115, 206
expression 21, 64, 65
  case 23
  constant specification 31
  initialization 12, 23, 45, 49, 66, 67, 134
  specification 68, 69
extent 10
EXTERNAL attribute 70, 71, 107
external file 72, 160, 167
external input/output unit 152, 163, 167
external procedure 9, 70
EXTERNAL statement 70, 71, 105
external subprogram 80, 147, 182
external subprogram unit 146
external subroutine 183
external subroutine subprogram unit 146

F

F edit descriptor 59
file 72, 130, 131, 152
  connected 73
  existence 72
  external 72, 160, 167
  formatted 154, 167
  initial point 73
  internal 72, 158, 159, 160
  position 73, 74, 166
  terminal point 73
  unformatted 156
file name
  inquiry by 94
file positioning statement 74, 75
FILE= specifier 131
fixed source form 178
FLOOR function 113, 206
FMT= specifier 153, 154, 158, 160, 164, 166
FORALL construct 11
form
  program 179
  source 178
FORM= specifier 95, 131
format 79, 153, 165
  explicit 155, 159, 167
  item 79
  specification 78
  specifier 152
FORMAT statement 79
formatted file 154, 167
formatted input/output 152, 154, 155, 158, 159, 164, 165, 166, 167
formatted record 72
FORMATTED= specifier 95
Fortran character set 27
FRACTION function 36, 115, 206
free source form 178
function
  ACHAR 113
  ACOS 110
  actual argument 81
  ADJUSTL 111
  ADJUSTR 111
  AIMAG 113
  AINT 113
  ALL 109
  ALLOCATED 5, 55, 115

ANINT 113
ANY 109
array intrinsic 108
array valued 15
ASIN 110
ASSOCIATED 5, 114, 136, 137, 138, 141, 143
ATAN 110
ATAN2 110
BIT_SIZE 36, 114
BTEST 111
CEILING 113
CHAR 113
CMPLX 113
computation intrinsic 110, 144
CONJG 113
conversion intrinsic 67, 112
COS 110
COSH 110
COUNT 109
CSHIFT 109
DBLE 113
definition 80
DIGITS 36, 37, 115
DIM 110
DOT_PRODUCT 110
DPROD 110
elemental 11, 15
elemental intrinsic 67, 69, 107, 190
EOSHIFT 109
EPSILON 36, 115, 145
EXP 110
EXPONENT 36, 115
FLOOR 113
FRACTION 36, 115
HUGE 36, 115
IACHAR 25, 113
IAND 111
IBCLR 111
IBITS 113
IBSET 111
ICHAR 113
IEOR 111
INDEX 111
inquiry intrinsic 67, 69, 144
INT 113
intrinsic 36, 106
IOR 111
ISHFT 111
ISHFTC 111
KIND 67, 114, 119, 145
LBOUND 115
LEN 114
LEN_TRIM 111
LGE 111
LGT 111
LLE 111
LLT 111
LOG 110
LOG10 110
LOGICAL 113
MATMUL 110
MAX 110
MAXEXPONENT 36, 37, 115
MAXLOC 109
MAXVAL 109
MERGE 109
MIN 110
MINEXPONENT 36, 37, 115
MINLOC 109
MOD 110
model intrinsic 144
NEAREST 36, 115
NINT 113
nonelemental 191
NOT 111
NULL 49, 113, 136, 137, 138, 141, 142, 143
PACK 109
PRECISION 36, 115
PRESENT 114, 132, 133
PRODUCT 109
RADIX 36, 37, 115
RANGE 36, 115
REAL 113
recursive 81, 172
reference 80, 81
REPEAT 111
RESHAPE 12, 13, 109
result clause 81
RRSPACING 36, 115
SCALE 36, 115

SCAN 111
SELECTED_INT_KIND 97, 114, 145
SELECTED_REAL_KIND 114, 145
SET_EXPONENT 36, 115
SHAPE 115
SIGN 110
SIN 110
SINH 110
SIZE 67, 115
SPACING 36, 115
SPREAD 109
SQRT 110
statement 80, 81
subprogram 81, 128
SUM 109
TAN 110
TANH 110
TINY 36, 115
TRANSFER 62, 113
transformational 191
transformational intrinsic 15, 67, 69, 107
TRANSPOSE 109
TRIM 111
type 81
UBOUND 115
UNPACK 109
VERIFY 111
function ABS 110
FUNCTION statement 173

G

G edit descriptor 59
generic intrinsic procedure 82
generic name 83
generic operator 82, 83
generic procedure 9, 82
name 82
reference 83
generic specification 83
GO TO statement 84, 85
group
equivalence 181
namelist 163

H

hexadecimal constant 97
host association 43, 45, 86, 87, 103, 127, 129, 183
HUGE function 36, 115, 207

I

I edit descriptor 59
IACHAR function 25, 113, 207
IAND function 111, 207
IBCLR function 111, 207
IBITS function 113, 208
IBSET function 111, 208
ICHAR function 113, 208
IEOR function 111, 208
IF construct 88, 89
IF statement 89
IF-THEN statement 89
implicit interface 100
IMPLICIT NONE statement 90, 91, 188
IMPLICIT statement 90, 91, 103
implicit type rule 87
implicit typing 90, 127, 129
INCLUDE line 92, 93
INDEX function 111, 209
indexed loop 52
initial point
file 73
initial value 34, 35, 147
initialization 35, 175
data 34
expression 12, 23, 45, 49, 66, 67, 134
input/output
advancing 74, 164, 165
defined type 44
direct access 154, 155, 156, 157
formatted 152, 154, 155, 158, 159, 164, 165, 166, 167
list directed 152, 158, 159, 160, 161
namelist 152, 162, 163
nonadvancing 57, 73, 74, 152, 166, 167
sequential access 158, 159, 164,

165, 166, 167, 168, 169
stream 152
unformatted 152, 156, 157, 168, 169
input/output unit 161, 165
INQUIRE statement 94, 95
inquiry
by file name 94
by length 94
by unit 94
intrinsic function 67, 69, 114, 144
specifier 95
INT function 113, 209
integer
constant 119
default 96
kind 96
model 36, 37
operator 97
type 96
INTEGER statement 97
intent 39
specification 99
INTENT attribute 98, 99
INTENT statement 99
interface 183
block 9, 38, 70, 82, 83, 87, 100, 101, 127, 147
body 86, 87, 101
explicit 100
procedure 8, 9, 100
INTERFACE statement 101
internal file 72, 158, 159, 160
internal input/output unit 152
internal procedure 9, 86, 102, 129
internal procedure part 103, 125
internal subprogram 80, 103, 147, 182
internal subroutine 183
interpretation of blanks 56
intrinsic assignment 20, 21, 35
INTRINSIC attribute 104, 105, 107
intrinsic function 36, 106, 190
array 108
computation 110
conversion 112
inquiry 114
model 114
reference 107
intrinsic operation 15, 67, 83
intrinsic operator 65
intrinsic procedure 9, 83, 104, 193–234
generic 82
INTRINSIC statement 104, 105
intrinsic subroutine 116
IOR function 111, 210
IOSTAT= specifier 29, 95, 131, 153, 154, 156, 158, 160, 162, 164, 166, 168
ISHFT function 111, 210
ISHFTC function 111, 210
item
format 79

K

keyword 179
argument 7, 8, 9
DEFAULT 23
ELEMENTAL 183
PURE 183
RECURSIVE 173, 183
kind
character 27
complex 33, 119
default 119
integer 96
logical 123
number 96
parameter 26, 97, 112, 118, 119, 122, 144, 171
real 170
selector 119
KIND function 67, 114, 119, 145, 211

L

L edit descriptor 59
label 85, 179
LBOUND function 115, 211
LEN function 114, 211
LEN_TRIM function 111, 212
length
inquiry by 94

parameter 27
string 68
letter
exponent 171
LGE function 111, 212
LGT function 111, 212
line
INCLUDE 92, 93
source 179
list-directed input/output 152, 158, 159, 160, 161
literal constant 67, 69
LLE function 111, 213
LLT function 111, 213
local name 87, 177
local variable 173
LOG function 110, 213
LOG10 function 110, 213
logical
constant 123
default 123
kind 123
operator 123
type 122
LOGICAL function 113, 214
LOGICAL statement 122, 123
loop
indexed 52
simple 52
termination statement 53
while 52, 53
lower bound 10, 17

M

main program 124, 125, 146, 147, 174
many-to-one array section 19
masked array assignment 20, 190
MATMUL function 110, 214
MAX function 110, 215
MAXEXPONENT function 36, 37, 115, 215
MAXLOC function 109, 215
MAXVAL function 109, 216
MERGE function 109, 217
method
approximation 170
representation 171
MIN function 110, 217
MINEXPONENT function 36, 37, 115, 217
MINLOC function 109, 218
MINVAL function 109, 219
MOD function 110, 219
model
bit 36, 37
data representation 36
integer 36, 37
intrinsic function 114, 144
real 36, 37
module 30, 86, 126, 127, 128, 147, 187
function subprogram 129
procedure 9, 86, 128
program unit 146
subprogram 80, 129, 147, 182
subprogram part 129
subroutine 183
subroutine subprogram 129
MODULE PROCEDURE statement 38, 101, 127
MODULO function 110, 220
MVBITS subroutine 117, 220

N

name 179
component 43
construct 23, 52, 53, 77, 89, 179, 191
generic 82, 83
intrinsic procedure 105
local 87, 177
namelist group 162
procedure 82
type 43
user-defined operator 83
NAME= specifier 95
named common block 31
named constant 45, 49, 67, 69, 134
NAMED= specifier 95
namelist

comment 163
group 163
name 162
input/output 152, 162, 163
specifier 152, 162
NEAREST function 36, 115, 220
NEXTREC= specifier 95
NINT function 113, 221
NML= specifier 153, 162
nonadvancing input/output 57, 73, 74, 152, 166, 167
nonelemental function 191
nonelemental operation 191
nonexecutable program unit 147
NOT function 111, 221
NULL function 49, 113, 136, 137, 138, 141, 142, 143, 221
null string 25
nullification
pointer 142, 143
NULLIFY statement 5, 34, 55, 136, 137, 138, 142, 143
number
complex 32
kind 96
record 155, 156
NUMBER= specifier 95
numeric sequence structure 181

O

O edit descriptor 59
object
allocate 5
automatic 3, 54
automatic character 27
common block 31
data 35
data-implied DO 35
defined type 44
dynamic 54
equivalence 63
pointer 139
structured 44
octal constant 97
ONLY clause 186, 187
open specifier 130
OPEN statement 29, 72, 130, 131, 155, 157
OPENED= specifier 95
operand 64, 67, 69
operation
elemental 191
intrinsic 15, 67, 83
nonelemental 191
user defined 15
operator 39
.AND. 123
.EQV. 123
.NEQV. 123
.NOT. 123
.OR. 123
arithmetic 33, 97, 171
binary 39, 65
character 26, 27
complex 33
defined 39
generic 82, 83
integer 97
intrinsic 65
logical 123
precedence 39, 64
real 171
relational 27, 33, 97, 171
unary 39, 65
user defined 38, 83
optional argument 9, 107, 132
OPTIONAL attribute 133
optional sign 56
OPTIONAL statement 133

P

P edit descriptor 57
PACK function 109, 222
PAD= specifier 95, 131
padding
blank 21, 212, 213
parallelism
data 14
parameter
kind 26, 97, 112, 118, 119, 122,

144, 171
length 27
PARAMETER attribute 134, 135
PARAMETER statement 12, 91, 135
parent 67
array 18, 19
string 25
substring 24
part
reference 47
pointer 3, 5, 42, 49, 54, 136, 140, 141, 181, 184
assignment 20, 55, 136
associated 5, 143
association 138, 139, 141
definition 141
disassociated 143
disassociation 138, 139
nullification 142, 143
object 139
target 4
undefined 139, 143
pointer assignment statement 137, 138, 139, 140, 142
POINTER attribute 4, 7, 11, 17, 20, 21, 43, 46, 47, 55, 65, 136, 137, 138, 139, 140, 141, 143
POINTER statement 140, 141
position
file 73, 74, 166
specifier 75
POSITION= specifier 95, 131
positional argument 8
precedence
operator 39, 64
precision 144
PRECISION function 36, 115, 222
prefix 81
PRESENT function 114, 132, 133, 222
PRINT statement 72, 153, 161, 165
PRIVATE attribute 43, 127, 148, 149
PRIVATE statement 43, 45, 47, 149
procedure 182
argument 129
definition 6
dummy 7, 9, 70, 71
elemental 7
external 9, 70
generic 9, 82, 83
interface 8, 9, 100
internal 9, 86, 102, 129
intrinsic 9, 83, 104, 193–234
module 9, 86, 128
name 82
pure 150
recursive 137, 173
reference 6, 83, 132
specific 9
subprogram 147
user defined 8
PRODUCT function 109, 223
program 146, 179
execution 147
form 179
main 124, 125, 146, 147, 174
unit 86, 124, 126, 146, 147
block data 146, 147
external 146
module 146
nonexecutable 147
PROGRAM statement 125
program unit
block data 70
PUBLIC attribute 43, 127, 148, 149
PUBLIC statement 149
PURE
keyword 183
pure
procedure 150

Q

quote edit descriptor 59

R

RADIX function 36, 37, 115, 223
RANDOM_NUMBER subroutine 117, 224
RANDOM_SEED subroutine 117, 224
range 144
substring 19
RANGE function 36, 115, 224

rank 10, 17, 51, 189
- array 19

READ statement 72, 152, 153, 155, 157, 159, 161, 162, 163, 164, 165, 166, 167, 168, 169
READ= specifier 95
READWRITE= specifier 95
real
- constant 171
- default 170, 171
- double precision 170, 171
- kind 170
- model 36, 37
- operator 171
- single precision 170, 171
- type 170

REAL function 113, 225
REAL statement 171
REC= specifier 153, 154, 156
RECL= specifier 95, 131, 155, 157
record 72, 155, 157, 169
- data 72
- end-of-file 72
- formatted 72
- number 155, 156
- unformatted 72

recursion 172
recursive function 81, 172
recursive function statement 173
RECURSIVE keyword 173, 183
recursive procedure 137, 173
recursive subroutine 172
recursive subroutine statement 173
reference
- function 80, 81
- generic procedure 83
- intrinsic function 107
- part 47
- procedure 6, 132
- subroutine 183

relational operator 27, 33, 97, 171
rename 187
renaming 186
REPEAT function 111, 225
representation method 171
RESHAPE function 12, 13, 109, 225
RESULT clause 173
result clause 81
RETURN statement 84, 85
REWIND statement 72, 73, 74, 75
RRSPACING function 36, 115, 226

S

S edit descriptor 57
SAVE attribute 3, 11, 34, 55, 125, 127, 174, 175
SAVE statement 31, 127, 175
saved entity 175
scalar 189
scale factor 56, 57
SCALE function 36, 115, 226
SCAN function 111, 226
scope 176
scoping unit 180
section
- array 11, 18, 19, 67
- subscript 18, 19
- triplet 11
- vector subscript 11, 19, 47

SELECT CASE statement 23
SELECTED_INT_KIND function 97, 114, 145, 227
SELECTED_REAL_KIND function 114, 145, 227
selector
- kind 119

semicolon (;) 179
sequence
- association 6, 7, 183
- collating 25, 27, 193, 199, 207
- storage 7, 31
- structure
  - character 181
  - numeric 181
- type 180

SEQUENCE statement 7, 43, 45
sequential access 73, 74, 152, 158, 159, 164, 165, 166, 167
sequential access input/output 168, 169
SET_EXPONENT function 36, 115, 227
shape 10

    array 191
SHAPE function 115, 228
sign
    optional 56
SIGN function 110, 228
simple loop 52
SIN function 110, 228
single precision real 170, 171
SINH function 110, 228
size 10
SIZE function 67, 115, 229
SIZE= specifier 153, 166
skipping records 56
slash edit descriptor 57
source form 178
    fixed 178
    free 178
SP edit descriptor 57
SPACING function 36, 115, 229
specific procedure 9
specification
    array 16, 51
    assignment 39
    expression 68, 69
    format 78
    generic 83
    intent 99
    part 125, 127
specifier
    ACCESS= 95, 131, 155
    ACTION= 95, 131
    ADVANCE= 153, 166
    array 16, 17
    BLANK= 95, 131
    connection 131
    DELIM= 95, 131
    DIRECT= 95
    END= 153, 158, 160, 162, 164, 166, 168
    EOR= 153, 166
    ERR= 29, 95, 131, 153, 154, 156, 160, 164, 166, 168
    EXIST= 95
    FILE= 131
    FMT= 153, 154, 158, 160, 164, 166
    FORM= 95, 131
    format 152
    FORMATTED= 95
    inquiry 95
    IOSTAT= 29, 95, 131, 153, 154, 156, 158, 160, 162, 164, 166, 168
    NAME= 95
    NAMED= 95
    namelist 152, 162
    NEXTREC= 95
    NML= 153, 162
    NUMBER= 95
    open 130
    OPENED= 95
    PAD= 95, 131
    position 75
    POSITION= 95, 131
    READ= 95
    READWRITE= 95
    REC= 153, 154, 156
    RECL= 95, 131, 155, 157
    SIZE= 153, 166
    STAT= 5
    STATUS= 29, 131
    unit 152
    UNIT= 153, 154, 156, 158, 160, 162, 164, 166, 168
SPREAD function 109, 229
SQRT function 110, 230
SS edit descriptor 57
starting position
    substring 25
STAT= specifier 5
statement
    accessibility 148, 149
    ALLOCATABLE 2
    ALLOCATE 3, 4, 136, 137, 138, 140
    assignment 10, 20, 191
    BACKSPACE 72, 73, 74, 75
    branch target 85
    CALL 183
    CASE 23
    CHARACTER 27
    CLOSE 28, 29, 72

COMMON 31, 62, 63
COMPLEX 33
CONTAINS 125, 129
CONTINUE 53
continued 179
CYCLE 52, 53
DATA 13, 34, 35, 67, 97
data transfer 153, 155, 157, 158, 159, 160, 161, 163, 165, 167, 169
DEALLOCATE 3, 4, 55, 136, 137, 138, 143
defined type 149
DIMENSION 51
DO 53
DOUBLE PRECISION 171
ELSE 89
ELSE IF 89
ELSEWHERE 191
END 81, 129, 147, 183
END DO 53
END IF 89
END INTERFACE 101
END SELECT 23
ENDFILE 72, 73, 74, 75
ENDWHERE 191
ENTRY 132, 180, 181, 183
EQUIVALENCE 62, 63, 67
EXIT 53
EXTERNAL 70, 71, 105
file positioning 74, 75
FORMAT 79
FUNCTION 173
GO TO 84, 85
IF 89
IF-THEN 89
IMPLICIT 90, 91, 103
IMPLICIT NONE 91, 188
INQUIRE 94, 95
INTEGER 97
INTENT 99
INTERFACE 101
INTRINSIC 104, 105
length 179
LOGICAL 122, 123
loop termination 53
MODULE PROCEDURE 38, 101, 127
NULLIFY 5, 34, 55, 136, 137, 138, 142, 143
OPEN 29, 72, 130, 131, 155, 157
OPTIONAL 133
PARAMETER 12, 91, 135
POINTER 140, 141
pointer assignment 137, 138, 139, 140, 142
PRINT 72, 153, 161, 165
PRIVATE 43, 45, 47, 149
PROGRAM 125
PUBLIC 149
READ 72, 152, 153, 155, 157, 159, 161, 162, 163, 164, 165, 166, 167, 168, 169
REAL 171
recursive function 173
recursive subroutine 173
RETURN 84, 85
REWIND 72, 73, 74, 75
SAVE 31, 127, 175
SELECT CASE 23
separator 179
SEQUENCE 7, 43, 45
statement function 81
STOP 84, 85
SUBROUTINE 173, 183
TARGET 184, 185
TYPE 43, 45, 149
type 34, 35, 51, 97, 148
USE 103, 186, 187
WHERE 11, 190, 191
where body 191
WRITE 72, 152, 153, 155, 157, 159, 161, 162, 163, 164, 165, 166, 167, 168, 169
statement function 80, 81
status
allocation 174
association 5, 174
definition 174
status variable 5
STATUS= specifier 29, 131
STOP statement 84, 85

storage
    association 62, 180, 183
    sequence 7
    unit 63
        character 27, 181
        numeric 181
storage association 30, 31
storage sequence 31
stream input/output 152
stride 19
string
    length 68
    null 25
    parent 25
structure 42, 44
    component 46, 47, 67
        reference 47
    constructor 44, 45, 48, 49, 67, 69
structure constructor 45, 49
structured object 44
subprogram
    executable 147
    external 80, 147, 182
    function 81, 128
    internal 103, 147, 182
    module 80, 129, 147, 182
    part 128
    procedure 147
    subroutine 128, 183
    unit
        external 146
subroutine 182
    CPU_TIME 117, 200
    DATE_AND_TIME 117
    external 183
    internal 183
    intrinsic 116
    module 183
    MVBITS 117
    RANDOM_NUMBER 117
    RANDOM_SEED 117
    recursive 172
    reference 183
    subprogram 128, 183
    SYSTEM_CLOCK 117
SUBROUTINE statement 173, 183
subscript 19
    section 18, 19
    triplet 18, 19, 47
    vector 18, 19, 47
    vector section 11, 19, 47
substring 25, 67
    character 24
    ending position 25
    parent 24
    range 19
    starting position 25
SUM function 109, 230
SYSTEM_CLOCK subroutine 117, 231

T

T edit descriptor 57
tabbing 56
TAN function 110, 231
TANH function 110, 231
target 7, 137, 138, 139, 140, 141, 184
    pointer 4
TARGET attribute 136, 137, 139, 141, 184, 185
TARGET statement 184, 185
terminal point
    file 73
TINY function 36, 115, 231
TL edit descriptor 57
TR edit descriptor 57
TRANSFER function 62, 113, 232
transformational function 191
transformational intrinsic function 15, 67, 69, 107
TRANSPOSE function 109, 232
TRIM function 111, 232
triplet
    section 11
    subscript 18, 19, 47
type 90
    character 26
    complex 32
    declaration 71, 99, 105, 133, 134, 135, 141, 149, 175, 185, 188
    defined 7, 43, 45, 83, 91

definition 43
derived 42
function 81
integer 96
logical 122
name 43
real 170
sequence 180
user defined 46, 48
TYPE statement 43, 45, 149
type statement 34, 35, 51, 148
typing
implicit 90, 129

U

UBOUND function 115, 233
unallocated array 55
unary operator 39, 65
undefined pointer 139, 143
undefinition 176
unformatted file 156
unformatted input/output 152, 156, 157, 168, 169
unformatted record 72
unit 130, 131, 155, 157, 158
connected 73
external 163, 167
external input/output 152
input/output 161, 165
inquiry by 94
internal input/output 152
program 86, 124, 126, 146, 147
scoping 180
specifier 152
storage 63, 181
UNIT= specifier 153, 154, 156, 158, 160, 162, 164, 166, 168
UNPACK function 109, 233
upper bound 10, 17
use association 45, 49, 87, 128, 148, 183, 186
USE statement 103, 186, 187
user-defined assignment 38
user-defined operation 15
user-defined operator 38, 83
user-defined procedure 8
user-defined type 46, 48, 91

V

value
case 23
initial 147
variable 21, 67, 69, 188, 189
character 159
local 173
status 5
vector
subscript 18, 19, 47
vector subscript 11, 19, 47
VERIFY function 111, 234

W

where bodystatement 191
WHERE construct 11, 190, 191
WHERE statement 11, 190, 191
while loop 52, 53
whole array 18
WRITE statement 72, 152, 153, 155, 157, 159, 161, 162, 163, 164, 165, 166, 167, 168, 169

X

X edit descriptor 57

Z

Z edit descriptor 59

Example

```
MODULE PRECISION
   ! ADEQUATE is a kind number of a real representation with at least
   !    10 digits of precision and 99 digits range, which results in
   !    64-bit arithmetic on most machines.
   INTEGER, PARAMETER :: ADEQUATE = SELECTED_REAL_KIND(10,99)
END MODULE PRECISION

MODULE LINEAR_EQUATION_SOLVER

   USE PRECISION
   IMPLICIT NONE
   PRIVATE ADEQUATE
   CONTAINS

   SUBROUTINE SOLVE_LINEAR_EQUATIONS (A, X, B, ERROR)

      ! Solve the system of linear equations Ax = B.
      ! ERROR is true if the extents of A, X, and B are incompatible
      !    or a zero pivot is found.
      REAL (ADEQUATE), DIMENSION (:, :), INTENT (IN) :: A
      REAL (ADEQUATE), DIMENSION (:), INTENT (OUT) :: X
      REAL (ADEQUATE), DIMENSION (:), INTENT (IN) :: B
      LOGICAL, INTENT (OUT) :: ERROR
      REAL (ADEQUATE), DIMENSION (SIZE (B), SIZE (B) + 1) :: M
      INTEGER :: N

      ! Check for compatible extents.
      ERROR = SIZE (A, DIM=1) /= SIZE (B) .OR. SIZE (A, DIM=2) /= SIZE (B)
      IF (ERROR) THEN
          X = 0.0
          RETURN
      END IF

      ! Append the right-hand side of the equation to M.
      N = SIZE (B)
      M (1:N, 1:N) = A; M (1:N, N+1) = B

      ! Factor M and perform forward substitution in the last column of M.
      CALL FACTOR (M, ERROR)
      IF (ERROR) THEN
          X = 0.0
          RETURN
      END IF

      ! Perform back substitution to obtain the solution.
      CALL BACK_SUBSTITUTION (M, X)

   END SUBROUTINE SOLVE_LINEAR_EQUATIONS
```

```
   SUBROUTINE FACTOR (M, ERROR)

      ! Factor M in place into a lower and upper tranular matrix
      !    using partial pivoting.
      ! Terminate when a pivot element is zero.
      ! Perform forward substitution with the lower triangle
      !    on the right-hand side M(:,N+1)
      REAL (ADEQUATE), DIMENSION (:, :), INTENT (INOUT) :: M
      LOGICAL, INTENT (OUT) :: ERROR
      INTEGER, DIMENSION (1) :: MAX_LOC
      REAL (ADEQUATE), DIMENSION (SIZE (M, DIM=2)) :: TEMP_ROW
      INTEGER :: N, K
      INTRINSIC MAXLOC, SIZE, SPREAD, ABS

      N = SIZE (M, DIM=1)

      TRIANG_LOOP: &
      DO K = 1, N

         MAX_LOC = MAXLOC (ABS (M (K:N, K)))
         TEMP_ROW (K:N+1) = M (K, K:N+1)
         M (K, K:N+1) = M (K-1+MAX_LOC(1), K:N+1)
         M (K-1+MAX_LOC(1), K:N+1) = TEMP_ROW (K:N+1)

         IF (M (K, K) == 0) THEN
            ERROR = .TRUE.
            EXIT TRIANG_LOOP
         ELSE
            M (K, K:N+1) = M (K, K:N+1) / M (K, K)
            M (K+1:N, K+1:N+1) = M (K+1:N, K+1:N+1) -  &
               SPREAD (M (K, K+1:N+1), 1, N-K) *  &
               SPREAD (M (K+1:N, K), 2, N-K+1)
         END IF

      END DO TRIANG_LOOP

   END SUBROUTINE FACTOR

   SUBROUTINE BACK_SUBSTITUTION (M, X)

      ! Perform back substitution on the upper triangle
      !    to compute the solution.
      REAL (ADEQUATE), DIMENSION (:, :), INTENT (IN) :: M
      REAL (ADEQUATE), DIMENSION (:), INTENT (OUT) :: X
      INTEGER :: N, K
      INTRINSIC SIZE, SUM

      N = SIZE (M, DIM=1)

      DO K = N, 1, -1
         X (K) = M (K, N+1) - SUM (M (K, K+1:N) * X (K+1:N))
      END DO

   END SUBROUTINE BACK_SUBSTITUTION

END MODULE LINEAR_EQUATION_SOLVER
```

```
PROGRAM EXAMPLE

   USE PRECISION                                      ! Uses modules shown
   USE LINEAR_EQUATION_SOLVER                          !    previously

   IMPLICIT NONE
   REAL (ADEQUATE)  A(3,3), B(3), X(3)
   INTEGER  I, J
   LOGICAL  ERROR

   DO I = 1,3
      DO J = 1,3
         A(I,J) = I+J
      END DO
   END DO

   A(3,3) = -A(3,3)
   B = (/ 20, 26, -4 /)

   CALL  SOLVE_LINEAR_EQUATIONS (A, X, B, ERROR)

   PRINT *, ERROR
   PRINT *, X

END PROGRAM EXAMPLE



! Coefficient matrix A:

!  2.0   3.0   4.0
!  3.0   4.0   5.0
!  4.0   5.0  -6.0

! Constants on right-hand side of equation:

!  20.0
!  26.0
!  -4.0

! Error flag:

! F

! Solution:

!  1.0   2.0   3.0
```

The Fortran Company
6025 N. Wilmot Road
Tucson, Arizona 85750 USA
www.fortran.com
info@fortran.com