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
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.
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. ALLOCATE and DEALLOCATE Statements 2
2. Argument Keywords 4
3. CASE Construct 6
4. Complex Type and Constants 8
5. Defined Type: Definition 10
6. Defined Type: Structure Component 12
7. Modules 14
8. Pointers 16
9. SAVE Attribute and Statement 18
10. Source Form 20
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:

```fortran
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), &, STAT = IERR4)  ! IERR4 will be positive
   IF (IERR4 == 0) EXIT  ! if there is 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:

- ALLOCATE and DEALLOCATE Statements
- Pointer Association
- Dynamic Objects
- Pointer Association and Statement
- Pointers
- Pointer Nullification

Related Intrinsics:

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

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, 4.1.3
**ALLOCATE and DEALLOCATE Statements**

### 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.

### 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 ] )
```
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
- Internal Procedures
- Functions
- Module Procedures
- Generic Procedures and Operators
- OPTIONAL Attribute and Statement
- Interfaces and Interface Blocks
- 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
Argument Keywords

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).
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
SELECT CASE (ITEM)
  CASE (1:7, 52:81)
    BIN1 = BIN1 + 1.0
  CASE (8:32, 51, 82)
    BIN2 = BIN2 + 1.0
  CASE (33:50, 83:
    BIN3 = BIN3 + 1.0
  CASE DEFAULT
END SELECT

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

**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:**

- Fortran 95 Handbook, 8.4
- Fortran 95 Using F, 2.3
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 from within the construct is permitted.
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 \( a + bi \) in mathematical terms, where \( a \) is the real part and \( b \) is the imaginary part.

Examples:

```fortran
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
- Real Type and Constants
- Implicit Typing

Related Intrinsic:

- 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

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.
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*
Defined Type: Definition

Syntax:

A defined-type definition is:

\[
\text{TYPE} \ [ \ [ \ [ , \text{access-spec} \ ] \ :: \ ] \text{type-name} \\
\ [ \ \text{PRIVATE} \ ] \\
\ [ \ \text{SEQUENCE} \ ] \\
\ \text{component-declaration} \\
\ [ \ \text{component-declaration} \ ]... \\
\text{END TYPE} \ [ \ \text{type-name} \ ]
\]

A component declaration is:

\[
\text{type-spec} \ [ \ [ , \text{component-attribute-list} \ ] \ :: \ ] \text{component-list}
\]

A component attribute is one of:

- \text{POINTER}
- \text{DIMENSION} ( \text{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 \text{POINTER} attribute.

3. A component may itself be a defined type. If, in addition, the \text{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 \text{PUBLIC} or \text{PRIVATE} attribute or an internal \text{PRIVATE} statement.

6. The internal \text{PRIVATE} statement in a type definition makes the components unavailable outside the module even though the type itself might be available.

7. The \text{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).

8. Operations on defined types are defined with procedures and given operator symbols with interface blocks.
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:**

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

TYPE CLASS
  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-"

! An array assignment
ALGEBRA % STUDENT(27:33) = TRANSFERS(1:7)
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
- Defined Type: Variables

**To Read More About It:**

- *Fortran 95 Handbook,* 6.3
- *Fortran 95 Using F,* 6.3.1
Defined Type: Structure Component

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:

\[ subscript \hspace{1em} subscript-triplet \hspace{1em} vector-subscript \]

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.
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:

```fortran
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
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.
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:

```fortran
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.
```

```fortran
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
- POINTER Attribute and Statement
- Dynamic Objects
- Pointer Nullification
- Interfaces and Interface Blocks
- TARGET Attribute and Statement
- Pointer Association

To Read More About It:

- *Fortran 95 Handbook*, 2.3.4, 5.4, 6.5, 7.5.3, A.13, A.79
- *Fortran 95 Using F*, 8
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.
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.

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.

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
```

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
SAVE Attribute and Statement

**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.

![SAVE Attribute compatibility diagram](image-url)
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:

```fortran
PROGRAM NICE

! This is a nice way to write a program!
PRINT *
PRINT *, &
12.0 + 34.6

END PROGRAM NICE
```

```fortran
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](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:

- *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 (&).

    \texttt{PRINT *, \&}\n    \quad \"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. 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.

   \texttt{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.
## Index

### Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>21</td>
</tr>
<tr>
<td>–</td>
<td>9</td>
</tr>
<tr>
<td>%</td>
<td>13</td>
</tr>
<tr>
<td>&amp;</td>
<td>21</td>
</tr>
<tr>
<td>*</td>
<td>9</td>
</tr>
<tr>
<td>**</td>
<td>9</td>
</tr>
<tr>
<td>+</td>
<td>9</td>
</tr>
<tr>
<td>.EQ.</td>
<td>9</td>
</tr>
<tr>
<td>.NE.</td>
<td>9</td>
</tr>
<tr>
<td>/</td>
<td>9</td>
</tr>
<tr>
<td>/=</td>
<td>9</td>
</tr>
<tr>
<td>;</td>
<td>21</td>
</tr>
<tr>
<td>==</td>
<td>9</td>
</tr>
</tbody>
</table>

### A

- actual argument 4, 5
- aliasing 16
- allocatable array 2, 3
- allocate object 3
- ALLOCATE statement 2, 16, 17
- ALLOCATED function 3
- allocation 3
  - status 18
- alternate return 5
- ampersand (&) 21

### argument

- actual 4, 5
- association 11
- dummy 4, 5
- keyword 4, 5
- optional 5
- positional 4

### arithmetic operator 9

### array

- allocatable 2, 3
- component 11

### assignment

- pointer 16

### ASSOCIATED function 3, 16, 17

### associated pointer 3

### association

- argument 11

### host 11, 15

### status 3, 18

### attribute

- POINTER 2, 11, 12, 13, 16, 17
- PRIVATE 11, 15
- PUBLIC 11, 15
- SAVE 15, 18, 19
- TARGET 16, 17

### B

- blank
  - character 21
- block
  - common 18, 19
  - interface 5, 15

### C

- case
  - expression 7
  - value 7
- CASE construct 6, 7
- CASE statement 7
- character
  - blank 21
  - comment 21
- common block 18, 19
- complex
  - constant 9
  - default 9
  - kind 9
  - operator 9
- complex number 8
- COMPLEX statement 9
- complex type 8
- component
  - array 11
  - declaration 11
  - name 11
  - structure 12, 13
  - reference 13
- constant
  - complex 9
- construct
CASE 6, 7
name 7, 21
continued statement 21

D
DEALLOCATE statement 2, 16, 17
declaration
  component 11
type 19
default complex 9
DEFAULT keyword 7
defined type 11
definition
  defined type 11
  status 18
type 11
derived type 10
dummy argument 4, 5
dummy procedure 5

E
END SELECT statement 7
exclamation mark (!) 21
expression
  case 7
  initialization 7
external procedure 5

F
fixed source form 20
form
  program 21
  source 20
free source form 20
function
  ALLOCATED 3
  ASSOCIATED 3, 16, 17
  NULL 16, 17

G
generic procedure 5

H
host association 11, 15

I
implicit typing 15
initialization 19
  expression 7
interface
  block 5, 15
  procedure 4, 5
internal procedure 5
intrinsic procedure 5

K
keyword 21
  argument 4, 5
  DEFAULT 7
kind
  complex 9

L
label 21
line
  source 21

M
main program 18
module 14, 15
  procedure 5
MODULE PROCEDURE statement 15

N
name 21
  component 11
  construct 7, 21
type 11
NULL function 16, 17
NULLIFY statement 3, 16, 17
number
  complex 8
O

object
    allocate 3
operator
    arithmetic 9
    complex 9
    relational 9
optional argument 5

P

part
    reference 13
pointer 3, 10, 16
    assignment 16
    associated 3
    target 2
pointer assignment statement 17
POINTER attribute 2, 11, 12, 13, 16, 17
positional argument 4
PRIVATE attribute 11, 15
PRIVATE statement 11, 13
procedure
    dummy 5
    external 5
    generic 5
    interface 4, 5
    internal 5
    intrinsic 5
    module 5
    recursive 17
    specific 5
    user defined 4
program 21
    form 21
    main 18
    unit 14
PUBLIC attribute 11, 15

R

recursive procedure 17
reference
    part 13
relational operator 9

S

SAVE attribute 15, 18, 19
SAVE statement 15, 19
saved entity 19
section
    vector subscript 13
SELECT CASE statement 7
semicolon (;) 21
SEQUENCE statement 11
source form 20
    fixed 20
    free 20
specific procedure 5
specification
    part 15
specifier
    STAT= 3
    STAT= specifier 3
statement
    ALLOCATE 2, 16, 17
    CASE 7
    COMPLEX 9
    continued 21
    DEALLOCATE 2, 16, 17
    END SELECT 7
    length 21
    MODULE PROCEDURE 15
    NULLIFY 3, 16, 17
    pointer assignment 17
    PRIVATE 11, 13
    SAVE 15, 19
    SELECT CASE 7
    separator 21
    SEQUENCE 11
    TYPE 11
status
    allocation 18
    association 3, 18
    definition 18
status variable 3
structure 10
    component 12, 13
    reference 13
subscript
    triplet 13
    vector 13
vector section 13

T

target 17
  pointer 2
TARGET attribute 16, 17
triplet
  subscript 13
type
  complex 8
  declaration 19
  defined 11
  definition 11
  derived 10
  name 11
  user defined 12
TYPE statement 11

U

unit
  program 14
user-defined procedure 4
user-defined type 12

V

value
  case 7
variable
  status 3
vector
  subscript 13
vector subscript 13
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

END MODULE LINEAR_EQUATION_SOLVER
SUBROUTINE FACTOR (M, ERROR)

! Factor M in place into a lower and upper triangular 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 (:, :) :: 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)
ENDIF

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 (:, :) :: 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    ! inside back cover

IMPLICIT NONE
REAL (ADECATE)  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

ISBN

The Fortran Company
6025 N. Wilmot Road
Tucson, Arizona 85750 USA