Intel® Fortran Programmer's Reference

Issued in U.S.A.
Order Number: 687928-5001

World Wide Web: http://developer.intel.com

Contents

Appendix A Intel Fortran Extensions

Tables

About This Manual

This manual describes the Intel® Fortran Language for programmer's reference. It also provides description of all the library functions and intrinsic procedures.

This manual is organized as follows:

Chapter 1	"Introduction to Intel Fortran." Summarizes features of Intel Fortran that distinguish it from FORTRAN 77.
Chapter 2	"Language Elements." Describes the basic language elements of Intel Fortran, including character set, names, statement types and order, source program format, and the `INCLUDE` line.
Chapter 3	"Data Types and Data Objects." Describes intrinsic and derived types, the type declaration statement, attributes, constants, implicit typing, storage association, alignment, and dynamic data objects.
Chapter 4	"Arrays." Describes arrays and array-handling features of Intel Fortran.
Chapter 5	"Expressions and Assignments." Describes expressions, operators, assignment, and the `WHERE` construct.
Chapter 6	"Execution Control." Describes the constructs and statements that control program execution.
Chapter 7	"Program Units and Procedures." Describes program units and procedures, argument correspondence and association, interfaces, and modules.

Chapter 8	"I/O and File Handling." Describes the types of records and files, file connection and access, the I/O data list, the implied-`DO` loop, and ASA carriage control. At the end of this chapter are example programs that illustrate various feature of Intel Fortran I/O, including nonadvancing I/O.
Chapter 9	"I/O Formatting." Describes the syntax and use of format specifications and edit descriptors, as used with formatted I/O.
Chapter 10	"Intel Fortran Statements." Describes the syntax and function of all Intel Fortran statements and attributes. The statements and attributes are described in alphabetical order.
Appendix A	"Intel Fortran Extensions." Briefly summarizes all Intel Fortran extensions to the Fortran standard.
Glossary	Defines terms used in this manual.

Related Publications

The following documents provide additional information relevant to the Intel Fortran Compiler:

- *Fortran 95 Handbook*, Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. The MIT Press, 1997. Provides a comprehensive guide to the standard version of the Fortran 95 Language
- *Fortran 90/95 Explained*, Michael Metcalf and John Reid. Oxford University Press, 1996. Provides a concise description of the Fortran 95 language.
- For Win32-specific information, see the documentation included with the *Microsoft Win32 Software Development Kit*.
- For Microsoft Fortran PowerStation 32 information, see the documentation included with the *Microsoft Fortran Powerstation 32 Development System for Windows NT, Version 1.0*.

Information about the target architecture is available from Intel and from most technical bookstores. Some helpful titles are:

- *Intel® Fortran Libraries Reference*, Intel order number 687929
- *Intel® Fortran Programmer's Reference*, Intel order number 687928

- *Intel® C/C++ Compiler User's Guide*, order number 741901
- *Intel® Architecture Optimization Reference Manual*, Intel Corporation, order number 245127
- *Intel Architecture Software Developer's Manual:*
  - Volume 1: *Basic Architecture*, order number 243190
  - Volume 2: *Instruction Set Reference Manual*, order number 243191
- *Intel Processor Identification with the CPUID Instruction*, order number 241618

Most Intel documents are also available from the Intel Corporation web site at *www.intel.com*

Notational Conventions

This manual uses the following conventions:

`This type style`	indicates an element of syntax, a reserved word, a keyword, a filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
`THIS TYPE STYLE`	Fortran source text appears in upper case. `l` is lowercase letter L in examples. `1` is the number 1 in examples. `O` is the uppercase O in examples. `0` is the number 0 in examples.
`This type style`	indicates the exact characters you type as input.
`This type style`	indicates a place holder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the place holder.
`[`*`items`*`]`	items enclosed in brackets are options.
`{`*`item`*` | `*`item`*`}`	Select only one of the items listed between braces. A vertical bar (|) separates the items.
`...`	Ellipses indicate that you can repeat the preceding item.
`This type style`	indicates an Intel Fortran Language extension format.

This type style	indicates an Intel Fortran Language extension discussion. Throughout the manual, extensions to the ANSI standard Fortran language appear in this font and color to help you easily identify when your code uses a non-standard language extension.

Introduction to Intel® Fortran Compiler 1

This manual is a complete reference description of the Intel® Fortran compiler. Intel Fortran is fully compliant with ISO/IEC 1539:1995, hereafter referred to as “the Fortran 95 Standard” or as “the Standard Fortran” or as “Fortran 95.” Intel Fortran also includes a number of extensions to the Standard, as well as command-line options that allow you to override the default actions of the compiler. This manual describes the standard features, the extensions and the command-line options.

The rest of this chapter briefly summarizes the standard features of Fortran 95 that are not found in FORTRAN 77. It is chiefly of interest to the developer who is familiar with FORTRAN 77 but new to Fortran 95. If you are already familiar with Fortran 95, you may want to turn to Appendix A, “Intel Fortran Extensions” which lists all of the Intel extensions and refers to other parts of this manual, where the extensions are more fully described. For a full description of the command-line options, see the *Intel® Fortran Compiler User’s Guide.*

New Features in Fortran 95

Some extensions to FORTRAN 77 are included in Fortran 95 and other completely new features have been added. The following list summarizes features of Fortran 95 that are not in standard FORTRAN 77 and indicates where they are described in the manual.

Source Format

The fixed source form of FORTRAN 77 is extended by the addition of the semicolon (`;`) statement separator, and the (`!`) trailing comment, and also a free source form is provided.

The format used in a source program file is normally indicated by the file suffix, but the default format can be overridden by the command-line options /FI and /FR, as described in *Intel Fortran Compiler User's Guide*.

Data Types

Intrinsic data types are now parameterized. Each data type can have one or more kinds identified by a `KIND` type parameter. This is an integer value that determines the range and/or precision of values that entities of that type may hold.

Several `KIND` types may be implemented for each intrinsic data type; also, intrinsic inquiry functions are provided to establish what is available making "precision portability" possible.

In Intel Fortran, the `KIND` type parameter value is typically the number of bytes used to represent an entity of that type, except for `COMPLEX` entities, where the number of bytes required is double the `KIND` type value.

On Windows NT*, Intel Fortran provides an interface to the NT Unicode support for Fortran programs. See the *Intel Fortran User's Guide* for more information.

Derived data types are available: they are defined by the user and can be composed of components that are of the intrinsic types (`INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, and `CHARACTER`) or of previously defined derived data types. Scalar and array entities of derived data types may be declared.

For more details on data types, see Chapter 3, "Data Types and Data Objects."

Operators

Intrinsic operators can be extended and new operations defined, for use with operands of intrinsic or derived data types. The intrinsic assignment operator can be extended similarly.

User-defined operations and defined assignment are implemented by means of user-written procedures; see Chapter 7, "Program Units and Procedures." for details.

Control Constructs

The CASE construct enables one of a set of statement blocks to be executed on the basis of a case selector value (that can be INTEGER, CHARACTER or LOGICAL).

- Additional forms of the DO statement are provided, as well as the CYCLE and EXIT statements, to branch to the end of a DO loop and out of a DO loop respectively.

These facilities are described in Chapter 6, "Execution Control."

Arrays

Fortran 95 greatly extends array facilities which include the following:

- Array sections that permit selection of a subset of array elements have been introduced. Operations for processing whole arrays and array sections are included, and expressions, functions, and assignments can be array-valued. The WHERE construct and statement provide for masked-array assignment.
- Array constructors are provided. An array constructor is an unnamed, rank-one array value, the elements of which may be constant or variable in value. The RESHAPE intrinsic function can be used to produce an array value of higher rank from an array constructor.
- Several new sorts of array (extensions in FORTRAN 77) are provided in Fortran 95:
  - Assumed-shape ("assumed" meaning "taking on the characteristics of")
  - Deferred-shape (an allocatable array or array pointer)
  - Automatic, which is a new sort of explicit-shape array
- Many intrinsic array functions are provided in Fortran 95, classed as elemental, transformational, or inquiry.

Arrays are discussed in Chapter 4, "Arrays."

Procedures

A large number of new intrinsic procedures and procedure-related features are provided in the language. Many of these features are "elemental." Thus they accept either scalar or array arguments. In the latter case, the result is as if the procedure were applied separately to each element of the array.

Other additions are transformational functions, which operate on their arguments in a nonelemental fashion. These functions return properties of the arguments rather than values computed from them.

The following are some of these new procedures and feature-related procedures:

- *interface block*—This feature is the basis of a number of new facilities:
  - The interface block enables explicit specification of procedure interfaces, so that the names and properties of the dummy arguments of such a procedure are known in the scoping unit invoking the procedure. This makes it possible for the compiler to ensure that the dummy and actual arguments match.
  - Optional arguments and keyword-identified arguments are also available when the procedure interface is explicit.
  - In addition, the procedure interface block enables user-defined generic procedures to be written, and is the mechanism used to specify defined operators and defined assignment.
- *intent Attribute*—Dummy arguments to procedures can now be given an `INTENT` attribute (`IN`, `OUT` or `INOUT`).
- *local scoping*—Internal subprograms can be defined within a module subprogram, an external subprogram, or a main program unit. They are local to the scoping unit in which they are declared.
- *recursive procedures—Recursive procedures that can invoke* themselves, directly or indirectly (an extension in FORTRAN 77), are available as a standard feature in Fortran 95.

These facilities are discussed in Chapter 7, "Program Units and Procedures."

Pointers

Arrays and scalar variables can be given the `POINTER` attribute in Fortran 95. A pointer is an alias, and the variable (or allocated space) for which it is an alias is its target. Pointer facilities enable data to be accessed and handled dynamically. Allocatable arrays (noted in the array discussion earlier) are similar to array pointers, but are slightly simpler, more limited, and more efficient.

In Intel Fortran, the pointers are of two styles: Fortran 95 pointers and Cray. Cray pointers are non-standard and are discussed in "Cray-Style Pointers" in Chapter 3.

Modules

A module is a new type of program unit that allows the specification of data objects, parameters, derived types, procedures, operators, and `NAMELIST` groups. Partial or complete access to these module entities is provided by the `USE` statement. An entity may be declared `PRIVATE` to limit visibility to the module itself.

Typical applications of modules are the specification of global data (in preference to the more troublesome common block mechanism) or the specification of a derived type and its associated operations.

Modules are discussed in "Modules" in Chapter 7.

Non-advancing I/O

In FORTRAN 77, after a record-based I/O operation, the file pointer is moved to the start of the next record.

In Fortran 95, use of the I/O specifier `ADVANCE=NO`, causes the file pointer to be positioned after the characters just read or written, but not automatically at the start of the next record.

This makes character I/O operations much easier to handle. It is also possible to read a variable length record and determine its length.

I/O facilities are discussed in Chapters 8, "I/O and File Handling." and 9, "I/O Formatting."

Namelist I/O

Namelist I/O, similar to that provided in FORTRAN 77, is available in Intel Fortran 95. The `READ/WRITE` specifier `NML=namelist-group-name` has been added together with the `NAMELIST` statement that allows specification of the variables belonging to a `NAMELIST` group.

Language Elements 2

This chapter describes the basic elements of an Intel® Fortran program, including the character set, lexical tokens, and names, and describes the source program formats. It also summarizes the categories of statements in Intel Fortran and the rules controlling their use and ordering within program units. The `INCLUDE` line facility is described at the end of the chapter.

Character Set

The Fortran 95 character set consists of letters, digits, the underscore character, and special characters, as detailed in Table 2-1.

The processor character set consists of the Fortran 95 character set, plus:

- Control characters (Tab, Newline , and Carriage Return). Carriage Return and Tab are usually treated as “white space” in a source program. Their use may affect the appearance of a listing file.
- You can use the hash character (#), which may appear in column 1 to initiate a comment. The hash character is an Intel Fortran extension.

Table 2-1 Fortran 95 Character Set

Category	Characters
Letters	A to Z, a to z*
Digits	0 to 9
Underscore	_
Special characters	blank (space) = + - * / ( ) , . ' : ! " % & ; < > ?** $

* Lowercase alphabetic characters are equivalent to uppercase characters except when they appear in character strings or Hollerith constants.

** Although "?" has been designated a special character, it has no special meaning in the Fortran 90 language.

Intel Fortran supports the default character type, which has kind parameter = 1, as described in Chapter 3, "Data Types and Data Objects." However, support is provided for the use of conversions from Unicode to multibyte character sets (MBCS) and back. For more information see the *Intel® Fortran Compiler User's Guide.*

Lexical Tokens

Lexical tokens are the building blocks of a program; they consist of sequences of characters. They denote names, operators, literal constants, labels, keywords, delimiters, and can include the following characters and character combinations:

```
, = => : :: ; %
```

Names

Names are used in Fortran 95 to denote entities such as variables, procedures, derived types, named constants, and common blocks. A name must start with a letter and consists thereafter of any combination of letters, digits, and underscore (_) characters.

As an extension to Standard Fortran, the dollar sign may also be used in a name, but not as the first character.

Standard Fortran 95 allows a maximum length of 31 characters in a name; in Intel Fortran this limit is extended to 255 characters, and all are significant—that is, two names that differ only in their 255th character are treated as distinct. However, names and keywords are case insensitive; thus `Title$23_Name` and `TITLE$23_NAME` are the same name.

Program Structure

A complete executable program contains one main program unit and zero or more other program units, where each of these can be compiled separately.

A program unit is one of the following:

- Main program unit
- External function subprogram unit
- External subroutine subprogram unit
- Block data program unit
- Module program unit

Execution of the program starts in the main program and then control can be passed between the main program and the other program units.

The Fortran 95 program units, and the transfer of control between them, are described in Chapter 7, "Program Units and Procedures."

Statement Labels

A Fortran 95 statement can have a preceding label, composed of one to five digits. All statement labels in the same scoping unit must be unique; leading `0`s are not significant in distinguishing them. Although most Fortran 95 statements can be labeled, not all statements can be branched to. The `FORMAT` statement must have a label.

The `INCLUDE` line (which is not a statement but a directive to the compiler) must not have a label.

Construct Names

Fortran 95 has these types of constructs. CASE, IF, DO, FORALL and WHERE. These constructs can optionally be given names. When names are used with the DO construct, they can affect the operation of the CYCLE and EXIT statements.

The construct name appears before the first statement of the construct, followed by a ":" character. It should then be repeated at the end of the final statement of the construct. Chapter 6, "Execution Control," describes Fortran 95 constructs.

Statements

All Intel Fortran statements are listed in Table 2-2, with the following categorization codes. They are fully described in Chapter 10, "Intel Fortran Statements," in alphabetical order.

All control statements are executable statements.

a Assignment statement

c Control statement

e Executable statement

i I/O statement

n Nonstandard statement (extension)

p Program structure statement

s Specification statement

t Can be a terminal statement (of a DO construct)

Table 2-2 Intel Fortran Statement Categories

Statement	Code	Statement	Code	Statement	Code
ACCEPT	e,i,n	END INTERFACE		OPEN	e,i,t
ALLOCATABLE	s	END MAP	n,p	OPTIONAL	s
ALLOCATE	e,t	END MODULE	p	PARAMETER	s
ASSIGN	a,e,t	END SELECT	c,e	PAUSE	c,e,t

continued

Table 2-2 Intel Fortran Statement Categories (continued)

Statement	Code	Statement	Code	Statement	Code
Assignment statement	a,e,t	END STRUCTURE	n,p	POINTER	s
AUTOMATIC	n,s	END SUBROUTINE	e,p	POINTER (Cray)	n,s
BACKSPACE	e,i,t	END TYPE	p	Pointer assignment	a,e
BLOCK DATA	p	END UNION	n,p	PRINT	e,i,t
BYTE	n,s	END WHERE	c,e	PRIVATE	s
CALL	c,e,t	ENDFILE	e,i,t	PROGRAM	p
CASE	c,e	ENTRY	p	PUBLIC	s
CHARACTER	s	EQUIVALENCE	s	READ	e,i,t
CLOSE	e,i,t	EXIT	c,e	REAL	s
COMMON	s	EXTERNAL	s	RECORD	n,s
COMPLEX	s	FORMAT	i	RETURN	c,e
CONTAINS	p	FUNCTION	p	REWIND	e,i,t
CONTINUE	c,e,t	GOTO (assigned)	c,e	SAVE	s
CYCLE	c,e	GOTO (computed)	c,e	SELECT CASE	c,e
DATA	s	GOTO (unconditional)	c,e	SEQUENCE	s
DEALLOCATE	e,t	IF (arithmetic)	c,e	Statement function	p
DECODE	e,i,n	IF (block)	c,e	STATIC	n,s
DEFINE FILE	e,i,n	IF (logical)	c,e,t	STOP	c,e
DIMENSION	s	IMPLICIT	s	STRUCTURE	n,s
DO	c,e	INCLUDE	p	SUBROUTINE	p
DOUBLE COMPLEX	n,s	INQUIRE	e,i,t	TARGET	s
DOUBLE PRECISION	s	INTEGER	s	TYPE (declaration)	p,s
ELSE	c,e	INTENT	s	TYPE (definition)	s
ELSE IF	c,e	INTERFACE	p	TYPE (I/O)	e,i,n
ELSEWHERE	c,e	INTRINSIC	s	UNION	n,p

continued

Table 2-2 Intel Fortran Statement Categories (continued)

Statement	Code	Statement	Code	Statement	Code
ENCODE	e,i,n	LOGICAL	s	USE	s
END (program unit)	e,p	MAP	n,p	VIRTUAL	n,s
END BLOCK DATA	p	MODULE	p	VOLATILE	n,s
END DO	c,e,t	MODULE PROCEDURE	s	WHERE	c,e,t
END FORALL	c, e	FORALL	c, e		
END FUNCTION	e,p	NAMELIST	s	WRITE	e,i,t
END IF	c,e	NULLIFY	a,e,t		

Statement Order

Table 2-3 summarizes the rules for statement ordering. It should be read in conjunction with Table 2-4, which checks off those statements that can appear in the various categories of scoping unit.

In Table 2-3, vertical lines separate statements that can be interspersed; horizontal lines separate statements that cannot be interspersed. Thus, for example, the tables indicate that:

- the USE statements, if present, must come immediately after the initial statement of the program unit.
- the FORMAT statements can appear anywhere in the program unit between the USE statement position and the CONTAINS statement position (but not in modules, because Table 2-4 prohibits their appearance in modules).
- the DATA statements can be interspersed with executable constructs.

Table 2-3 Statement Ordering Requirements

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA statement</td></tr>
<tr><td colspan="3">USE statements</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER statements</td><td>IMPLICIT statements</td></tr>
<tr><td>PARAMETER and DATA statements</td><td>Derived- type definitions,
Interface blocks,
type declaration statements,
Statement function statements,
and Specification statements</td></tr>
<tr><td>DATA statements</td><td>Executable constructs</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">INTERNAL subprograms or MODULE subprograms</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Table 2-4 Statements Allowed in Scoping Units (Y=yes)

Scoping Unit **Statement**	**Main Program**	**Pubprogram**	**External**	**Module**	**Pubprogram**	**Module**	**Subprogram**	**Internal**	**Body**	**Interface**	**Data**	**Block**
USE statement	**Y**		**Y**	**Y**		**Y**		**Y**		**Y**		**Y**
ENTRY statement			**Y**			**Y**						
FORMAT statement	**Y**		**Y**		**Y**		**Y**					
PARAMETER statement	**Y**		**Y**	**Y**		**Y**		**Y**		**Y**		**Y**
IMPLICIT statement	**Y**		**Y**	**Y**		**Y**		**Y**		**Y**		**Y**
Type declaration statement	**Y**		**Y**	**Y**		**Y**		**Y**		**Y**		**Y**
Specification statement	**Y**		**Y**	**Y**		**Y**		**Y**		**Y**		**Y**

continued

Table 2-4 Statements Allowed in Scoping Units (continued) **(Y=yes)**

Scoping Unit / Statement	Main Program	External Subprogram	Module	Module Subprogram	Internal Subprogram	Interface Body	Block Data
DATA statement	Y	Y	Y	Y	Y		Y
Derived-type definition	Y	Y	Y	Y	Y	Y	Y

Source Program Forms

Fortran 95 has a free source form, but also accepts FORTRAN 77's fixed source form (as it must, since FORTRAN 77 is a subset of Fortran 95). Although the two forms are quite different, it is possible to use an "intersection" form that satisfies both. This would be necessary, for example, when preparing Fortran text that was intended to be `INCLUDE`d (see "INCLUDE Line" on page 14) in a Fortran program whose source form, fixed or free, was not known.

The fixed source form, the free source form, and the intersection form are described below.

The Intel Fortran compilation system assumes that source files are in the fixed format form, unless the source file being compiled has the suffix `.f90`. However, you can use the command-line options `/FI` and `/FR` to change the format accepted by the compiler. See the *Intel® Fortran Compiler User's Guide*.

Fixed Source Form

Statements or parts of statements must be written between character positions 7 and 72. Any text following position 72 is ignored. The `/Qextend_source` option (see the *Intel Fortran Compiler User's Guide*) extends the statement to position 132. Positions 1-6 are reserved for special use. Blanks are not significant except within a character context.

For example:

```
RETURN
R E T U R N
```

are equivalent, but:

```
c = "abc"
c = "a b c"
```

are not equivalent.

Multiple statements may appear on one line separated by a semicolon (;).

There are three classes of lines in Fortran 95 fixed source form:

1. Initial line
2. Continuation line
3. Comment line

Initial Line

An initial line has the following form:

- Positions 1 to 5 may contain a statement label.
- Position 6 contains a space or the digit 0.
- Positions 7 to 72 (optionally, to 254) can contain the statement.

Continuation Line

A continuation line has the following form:

- Positions 1 to 5 are blank.
- Position 6 contains any character other than 0 or a space. One practice is to number continuation lines consecutively from 1.
- Positions 7 to 72 (optionally, to 254) contain the continuation of a statement.

The Fortran 95 Standard specifies that a statement must not have more than 19 continuation lines.

In Intel Fortran, a statement consists of an initial line and up to 99 continuation lines.

Comment Line

Comment lines may be included in a program. Such lines do not affect the program in any way but can be used by the programmer to include explanatory notes. The letter `C,` or `c` , or an asterisk (`*`) in position 1 of a line, designates that line as a comment line; the comment text is written in positions 1 to 72. A line containing only blank characters in positions 1 to 72 is also treated as a comment line.

An exclamation mark (`!`) in position 1 or in any position except position 6, causes the rest of the line to be treated as a comment.

In Intel Fortran, a line with `D` or `d` in position 1 is by default treated as a comment. A command-line option, the `/Qd_lines` option, treats lines with `D` or `d` in position 1 as statements to be compiled. This facility is useful in program debugging. See the *Intel Fortran Compiler User's Guide* for more information about the `+dlines` option.

Also, Intel Fortran provides the extension that a line with # in position 1 is treated as a comment. This allows source files that have been preprocessed with `fpp` to be compiled.

Tab-format Lines

In Intel Fortran a tab character in the first position of a line can be used to skip past the statement label positions. If the character following the tab character is a digit, this is assumed to be in position 6, the continuation indicator position. Any other character following the tab character is assumed to be in position 7, the start of a new statement. A tab character in any other position of a line is treated as a space.

Free Source Form

In this form the source line is not divided into fields of predefined width, as in the fixed form. This makes it more convenient for input of text at an interactive terminal. The details of the free source form are as follows.

Source Lines

Lines can contain from 0 to 132 characters. The `/Qextend_source` option (see the *Intel Fortran Compiler User's Guide*) can extend the statement to position 132. Several Fortran 95 statements can be placed on a single source line, separated by "`;`" characters, and a single Fortran 95 statement can extend over more than one source line, as described below in "Statement Continuation" on page 12.

Statement Labels

Statement labels are not required to be in columns 1-5, but must be separated from the statement itself by at least one space.

Spaces

Spaces are significant:

- Spaces do not always appear within a lexical token, such as a name or an operator.
- In general one or more spaces are required to separate adjacent statement keywords, names, constants, or labels. Within the following keyword pairs, however, the space is optional:

```
BLOCK DATA              GO TO
DOUBLE PRECISION        IN OUT
ELSE IF                 SELECT CASE
END keyword
```

The *keyword* after `END` can be any allowed by the Fortran 95 syntax, including the following: `BLOCK DATA`, `DO`, `FILE`, `FUNCTION`, `IF`, `INTERFACE`, `MAP`, `MODULE`, `PROGRAM`, `SELECT`, `SUBROUTINE`, `STRUCTURE`, `TYPE`, `UNION`, or `WHERE`.

Spaces are not required between a name and an operator because the latter begins and ends with special symbols that cannot be part of a name. Multiple spaces, unless in a character context, are equivalent to a single space.

Examples Using Spaces

Spaces are denoted here (and throughout this manual where it is necessary to stress their presence) by ~~b~~.

```
IF~~bb~~(TEXT.EQ.'~~bbb~~YES') ...   ! Valid
```

Valid: the two spaces after `IF` are equivalent to one space. No spaces are required before or after `.EQ.`, because there is no ambiguity. Note that the three spaces in the character constant are significant.

```
IF(MbARY.bGE.MIKE) ...      ! Faulty
```

Faulty: the space is invalid in `MbARY`, and the space is invalid in `.bGE.`. (This example would be valid in the fixed form.)

Comments

In free source form, the only way of indicating a comment is by use of the "`!`" symbol. Unless it appears in a character context, the occurrence of a "`!`" symbol defines the start of a comment, which always continues to the end of the source line. It is thus not possible to embed a comment inside program text within a single source line, but it can follow program text on a source line. Furthermore, a Fortran 95 statement on a line with a trailing comment can be continued on subsequent lines.

Statement Continuation

A statement can be split over two or more source lines by appending an ampersand (`&`) symbol to each source line involved except the last. The ampersand must not be within a character constant.

In this way, in Standard Fortran 95, a statement can occupy up to 40 source lines. As an extension, Intel Fortran increases this limit to 100 source lines.

The `END` statement cannot be split by means of a continuation line.

The text of the source statement in a continuation line is assumed to resume from column 1, unless the first nonblank symbol in the line is an ampersand, in which case the text resumes from the first position after the ampersand.

Example of Statement Continuation

Consider the following two statements:

```
INTEGER  marks, total, difference,&      ! work variables
       mean, average
INTEGER  marks, total, difference, mean_&   ! work variables
       &value average
```

The second of these statements declares an integer variable called `mean_value`. Any spaces appearing in the variable name as a result of the continuation would have been invalid, which is why another "`&`" was used in the continuation line. (Alternatively "`value`" could have been positioned at column 1.) Splitting lexical tokens, including character constants, across source lines in this way, is permitted but should be avoided if possible. Comments cannot be continued.

Intersection Source Form

It is possible to write programs in a way that is acceptable as both free source form and fixed source form, unless in extended fixed format source mode. The rules are:

- Put labels in positions 1-5.
- Put statement bodies in positions 7-72.
- Begin comments with an exclamation mark (`!`) 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.
- Do not insert blanks in tokens.
- Separate adjacent names and keywords with a space.

INCLUDE Line

An INCLUDE line inserts text into a program during compilation. The INCLUDE line is a directive to the compiler; it is not a Fortran 95 statement. The format of an INCLUDE line is:

INCLUDE *character-literal-constant*

character-literal-constant is the name of a file containing the text to be included. The character literal constant must not have a kind parameter that is a named constant.

The contents of the specified file are substituted for the INCLUDE line before compilation and are treated as if they were part of the original program source text.

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.

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 that follow, it is possible to use an INCLUDE line to include a portion of a subprogram; this is not possible with a module.

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. Putting a second INCLUDE or another Fortran 95 statement on the same line using ";" as a separator is not permitted. Continuing an INCLUDE line using "&" is also not permitted.

INCLUDE lines may be nested. That is, a second INCLUDE line may appear within the text to be included, and the text that it includes may also have an INCLUDE line, and so on. Intel Fortran has a maximum INCLUDE line nesting level of 10. However, the text inclusion must not be recursive at any level; for example, included text A must not include text B if B includes text A.

The text of the file to be included must consist of complete Fortran 95 statements.

Example of INCLUDE Lines

```
INCLUDE "MY_COMMON_BLOCKS"
INCLUDE "/usr/include/machine_parameters.h"
...
! Program text may be included within the
!  executable part of the program as well as
!  the specification part.
READ *, theta
INCLUDE "FUNCTION_CALCULATION"
...
```

If *character-literal-constant* is only a *filename*, in other words no *pathname* is specified, the compiler searches a user-specified path. See the *Intel Fortran Compiler User's Guide* for information about the /I*directory* option, which tells the compiler to search directories specified by *directory* to locate files to be included.

Data Types and Data Objects

3

This chapter describes both intrinsic and derived data types and the form of the declaration statements used to assign these data types to data objects and functions. It defines the format of constants for each of the intrinsic data types. It illustrates the definition, declaration, and use of derived types. It outlines implicit typing and data initialization. Finally there is a brief discussion of the mechanisms available for storage association and for dynamic storage allocation.

Terminology

A data type defines a set of values and a means of representing, manipulating, and interpreting them. Intrinsic numeric and nonnumeric types are defined in the language, and a user can define additional types, known as derived types, which are structures composed of the intrinsic types and of other derived types.

A data object is a constant, a variable, a subobject of a variable, or a subobject of a constant, and has a data type.

A constant has a value that cannot be changed during execution of the program. A constant is a literal constant, unless it has the `PARAMETER` attribute, in which case it is a named constant.

A variable may have a value and this value can be defined and redefined during execution of the program. It can be a scalar variable, an array variable, or a subobject of a variable.

A subobject of a variable can be an array element, an array section, a character substring, or a structure component. A subobject of a constant is a portion of the constant; the portion referenced may depend on the value of a variable.

Intrinsic Data Types

The numeric types are INTEGER, REAL, and COMPLEX; the nonnumeric types are CHARACTER and LOGICAL.

Each Fortran 95 implementation defines a set of representations for each of these types. Each representation corresponds to a different range of values that can be attained by entities or constants declared to be of the corresponding type.

For real and complex types, different representations also have different levels of precision. Each representation is assigned an identifying KIND parameter, which is an integer value. One of the representations for each type is designated the default representation for that type. Table 3-1 shows the options available with Intel Fortran.

Table 3-1 Types and KIND Parameters

Type	KIND Parameter	Range	Storage Bytes	Alignment
INTEGER (BYTE)	1	-128 to 127	1	1
INTEGER	2	-2^{15} to $2^{15}-1$	2	2
INTEGER	4 (default)	-2^{31} to $2^{31}-1$	4	4
INTEGER	8	-2^{63} to $2^{63}-1$	8	8
REAL Precision: 6 to 9 decimal digits	4 (default)	-3.402823×10^{38} to -1.175495×10^{-38} and 0.0 and $+1.175495\times10^{-38}$ to $+3.402823\times10^{38}$	4	4

continued

Table 3-1 Types and KIND Parameters (continued)

Type	KIND Parameter	Range	Storage Bytes	Alignment
REAL Precision: 15 to 17 decimal digits	8	-1.797693x10^{+308} to -2.225073x10^{-308} and 0.0 and +2.225073x10^{-308} to +1.797693x10^{+308}	8	8
REAL Precision: 33 to 35 decimal digits	16	-1.189731x10^{+4932} to -3.362103x10^{-4932} and 0.0 and +3.362103x10^{-4932} to +1.189731x10^{+4932}	16	16
COMPLEX*	4 (default)	same as for REAL(4)	8	4
COMPLEX**	8	same as for REAL(8)	16	8
COMPLEX(KIND=16)	16	two components of REAL(16)	32	16
CHARACTER	1 (default)	ASCII character set***	1	1
LOGICAL	1	.TRUE. .FALSE.****	1	1
LOGICAL	2	.TRUE. .FALSE.**	2	2
LOGICAL	4 (default)	.TRUE. .FALSE.**	4	4
LOGICAL	8	.TRUE. .FALSE.**	8	8

* COMPLEX (KIND=4) is the same as COMPLEX o r COMPLEX*8.

** COMPLEX (KIND=8) is the same as DOUBLE COMPLEX or COMPLEX*16.

*** The ASCII character set uses only the values 0 to 127, but the Intel Fortran implementation allows use of all 8 bits of a CHARACTER entity. The processing of character sets requiring multibyte representation for each character makes use of all 8 bits.

****In a standard conforming program, .TRUE. is represented by 1 and .FALSE. is represented by 0. In nonstandard conforming programs involving arithmetic operators with logical operands, a logical variable may be assigned a value other than 0 or 1. In this case, any nonzero value is considered to be .TRUE. and only the value zero is considered to be .FALSE.

The KIND parameter for an intrinsic data type is the same as the storage requirements for that data type except for COMPLEX where the KIND parameter is the KIND parameter of the real or imaginary part.

Examples of simple type declarations are:

```
INTEGER :: i,j
! i and j are default 4-byte integers
INTEGER(KIND=2) :: i2
! i2 is a 2-byte integer
REAL,DIMENSION(5,5) :: a
! a is a 5x5 array of default reals
CHARACTER(LEN=10) :: c10
! c10 is a variable of 10 characters
```

Derived Types

Fortran 95 allows the creation of new data types that are constructed from the intrinsic data types and previously defined new data types. These new data types are known as derived types.

For example, a derived type for manipulating coordinates consisting of two real numbers can be defined as follows:

```
TYPE coord
  REAL :: x,y
END TYPE coord
```

`x and y are` ***`components`*** `of the derived type coord.`

```
Variables of type coord, named a and b, can be
declared as follows:
TYPE(coord) :: a, b
```

An assignment statement

```
a = b
```

copies the values of all the defined components of `b` to those of `a`.

The individual components of `a` and `b` are referenced as `a%x, a%y, b%x,` and `b%y`. By coding appropriate procedures (as described in "Scope and Association" in Chapter 7), the scope of the standard operators can be extended so that, for example,

```
a = a + b
```

could be defined to be equivalent to

```
a%x = a%x + b%x; a%y = a%y + b%y
```

or to anything else, depending on the user-defined procedure that is provided to implement the operation.

A derived-type entity can be used as an argument to a procedure and can be the result of a function— that is, a function of derived type can be defined.

Type Declarations

The general form of a type declaration statement is:

type-spec[[,*attribute-spec*] ... ::]*entity-list*

type-spec

is one of :

- INTEGER [*kind-selector*]
- REAL [*kind-selector*]
- DOUBLE PRECISION [*kind-selector*]
- CHARACTER [*char-selector*]
- LOGICAL [*kind-selector*]
- TYPE (*type-name*)
- DOUBLE COMPLEX (Intel Fortran extension)
- BYTE (Intel Fortran extension)

BYTE is equivalent to INTEGER (KIND=1). DOUBLE PRECISION is equivalent to REAL (KIND=8) and DOUBLE COMPLEX is equivalent to COMPLEX (KIND=8).

kind-selector is

([KIND=]*scalar-int-init-expr*)

scalar-int-init-expr is a scalar integer initialization expression that must evaluate to one of the KIND parameters available (see Table 3-1).

char-selector see "CHARACTER" in Chapter 10 for details.

type-name is the name of a derived type.

attribute-spec is one or more compatible items from the following:

- PARAMETER
- *access-spec*

- ALLOCATABLE
- DIMENSION (*array-spec*)
- EXTERNAL
- INTENT (*intent-spec*)
- INTRINSIC
- OPTIONAL
- POINTER
- SAVE
- TARGET

Table 10-1 contains a matrix of attribute compatibility.

access-spec is one of:

- PUBLIC
- PRIVATE

array-spec is a list of array bounds. Chapter 4, "Arrays," describes the formats.

intent-spec is one of:

- IN
- OUT
- INOUT

entity is one of:

- *variable-name* [(*array-spec*)] [* *character-length*] [= *initialization-expression*]
- *function-name* *[(array-spec)]* [* *character-length*]

Note that when there is an *initialization-expression* in the *entity* there must also be a :: separator in the statement.

Examples of Type Declarations

Below are examples of type declaration statements, some of which include data initialization components.

```
INTEGER i, j, k
! Default, KIND=4, integers i j k.
INTEGER :: i,j,k
! Using optional separator.

INTEGER(KIND=8) :: i=2**40
! An 8-byte initialized integer.

INTEGER(8),DIMENSION(10) :: i
! 10 element array of 8-byte integers.

REAL, DIMENSION(2,2):: a = &
RESHAPE((/1.,2.,3.,4./),(/2,2/))
! Using an array constructor for initialization.
COMPLEX :: z=(1.0,2.0)
! Initialized complex.

COMPLEX z=(1.0,2.0)  ! FAULTY
! Syntax error - no :: present.

CHARACTER(KIND=1) :: c
! One character (default length).

CHARACTER(LEN=10) :: c
! A 10-byte character string.

CHARACTER(*),PARAMETER :: title='Ftn 95 MANUAL'
! Length can be * for a named constant.
! title is a 13-byte character string.
```

```
CHARACTER(LEN=n) :: c
! If the statement is in a subprogram,
!  n must be known at entry, otherwise
!  it must be a constant.
SUBROUTINE x(c)
 CHARACTER*(*) :: c
 ! c assumes the length of the actual argument.
END

TYPE(node):: list_element
! A single entity, of derived type node.

TYPE(coord) :: origin = coord(0.0,0.0)
! Declaration and initialization of a
! user-defined variable
```

Alternative Form of Intrinsic Type Spec Declaration

As an extension, Intel Fortran allows, for noncharacter types, the *type-spec* to be given in the form:

*type * length*

where *type* is an intrinsic type excluding CHARACTER, and *length* is the number of bytes of storage required, as in Table 3-1. Alternatively, **length* may be placed after the entity name. If the entity is an array with an *array-spec* following it, **length* may also be positioned after the *array-spec.*

Example

```
REAL*8 r8(10)
REAL r8*8(10)
REAL r8(10)*8
```

are all equivalent to the following preferred notation:

```
REAL(8), DIMENSION(10) :: r8
```

Except for COMPLEX, *length* is the same as the equivalent KIND parameter; for COMPLEX, the KIND parameter is the KIND parameter of the real or imaginary part and so:

```
COMPLEX*8 is equivalent to COMPLEX(KIND=4).
COMPLEX*16 is equivalent to COMPLEX(KIND=8).
```

Alternative Form of Initialization Within Declaration

Intel Fortran permits the use of slashes delimiting the data values rather than the equal sign introducing the data item, although it is recommended that this format is not used. The :: separator must not be used and array constructors and structure constructors cannot be used. Arrays may be initialized by defining a list of values that will be sequence associated with the elements of the array (the DATA statement, which permits the use of implied-DO loops may be more appropriate).

Examples

```
INTEGER i/1/,j/2/
REAL a(2,2)/1.1,2.1,1.2,2.2/ ! a(i,j)=i.j
```

Increasing Default Sizes

Intel Fortran provides command-line options (/4I2, /4I4, /4I8) that increase the default sizes of integer, logical, real, and complex items. The options are described in the *Intel® Fortran Compiler User's Guide for Win32* Systems*.

Intrinsic Inquiry Functions

Two intrinsic functions, SELECTED_INT_KIND and SELECTED_REAL_KIND, are provided to determine the most appropriate KIND parameter to use for a given range and precision. These functions can be used to significantly enhance the portability of programs. The value of the KIND parameter can be set to the result returned by one of these functions.

SELECTED_INT_KIND has one argument which specifies the range required. For example:

```
PARAMETER(intkind=SELECTED_INT_KIND(11))
```

will set the parameter intkind to a value of 8, as the function will return the KIND parameter with the smallest storage requirement that can contain integers with a magnitude of 10^{11}. Integer variables can then be declared using the value of intkind thus:

```
INTEGER(KIND=intkind) :: i, j, k
```

SELECTED_REAL_KIND has two arguments corresponding to the range and precision required. For example:

```
PARAMETER(rlkind=SELECTED_REAL_KIND(P=10,R=99))
```

returns a value of 8, supporting at least 10 digits of precision for values with magnitude of at least 10^{99}. Declaring all REAL entities with REAL(rlkind) enables easy modification if it becomes necessary to change the range or precision.

Attributes

The attributes that may be included in a type declaration are individually described in Chapter 10, "Attributes," with further references from there to appropriate sections of the manual. A one-line summary of the purpose of each attribute is given here.

ALLOCATABLE	Storage for the array is to be explicitly allocated during execution.
DIMENSION (*array-spec*)	Declares an array.
EXTERNAL	Defines a subprogram or block data to be in another program unit.
INTENT (*intent-spec*)	Defines the mode of use of a dummy argument.
INTRINSIC	Allows the use of a specific intrinsic name as an actual argument.
OPTIONAL	Declares the presence of an actual argument as optional.

PARAMETER	Defines named constants.
POINTER	Declares the entity to be a pointer.
PRIVATE	Inhibits visibility outside a module.
PUBLIC	Provides visibility outside a module.
SAVE	Ensures the entity retains its value between calls of a procedure.
TARGET	Enables the entity to be the target of a pointer.

All the above attributes can also be specified by using separate statements, although an attribute may not be specified more than once for an entity.

Note that in Intel Fortran there are two POINTER statements with differing syntax. One supports the Standard Fortran 95 definition and the other supports Cray-style pointers.

The following additional attributes are Intel Fortran extensions and can only be specified using separate statements:

AUTOMATIC	Entity does not retain its value between procedure calls.
STATIC	Entity retains its value between procedure calls.
VOLATILE	Provides data sharing between asynchronous processes.

An attribute compatibility table and further information and examples can be found in the relevant entries in Chapter 10, Intel Fortran Statements.

Representation of Literal Constants

The formats of constants for each of the intrinsic data types are described below.

Integer Constants

A signed integer literal constant is:

[*sign*] *digit-string* [*_kind-parameter*]

sign is one of:

- +
- -

digit-string is:

- *digit*[*digit*] ...

kind-parameter is one of:

- *digit-string*
- the name of a scalar integer constant (PARAMETER)

For example:

- -123
- 123_1
- 123_ILEN where ILEN is a named integer constant that must have a value which is a valid KIND parameter, either 1, 2, 4, or 8

BOZ Constants

In DATA statements, additional forms of unsigned constants are permitted for initializing integer variables. The values are expressed in binary, octal, or hexadecimal notation, and are collectively known as BOZ constants. The formats are:

A binary constant is one of:

- B '*digit-string* '
- B "*digit-string* "

where *digit-string* contains only the digits 0 and 1.

An octal constant is one of:

- O '*digit-string* '
- O "*digit-string* "

where *digit-string* contains only the digits 0, ..., 7.

A hexadecimal constant is one of:

- Z '*hex-digit-string* '
- Z "*hex-digit-string* "

where *hex-digit-string* contains only the characters 0, ..., 9, A, ..., F,a,...,f .

For example:

```
INTEGER i ,j ,k
DATA i/B'01001010'/
DATA j/O'112'/
DATA k/Z'4A'/
```

initializes `i`, `j`, and `k` to the decimal value 74.

As an extension, Intel Fortran also allows octal constants with a trailing `O`, and hexadecimal constants with a trailing `X`. For example:

```
'112'O  '4A'X
```

are alternative representations to those used in the example above.

Intel Fortran also extends the range of use of these constants to contexts other than initializing integers. These extensions are described in "Typeless Constants", page 3-16.

Real Constants

A signed real literal constant is one of:

- *`[sign] digit-string [exponent] [_kind-parameter]`*
- *`[sign] digit-string.[digit-string] [exponent] [_kind-parameter]`*
- *`[sign] [digit-string] .digit-string [exponent] [_kind-parameter]`*

`exponent` is

`exponent-letter [sign] digit-string`

`exponent-letter` is one of:

- `E`
- `D`
- `Q`

`sign` and *`digit-string`* are explained in the section "Integer Constants", page 3-11.

The use of Q is an Intel Fortran extension.

If no KIND parameter is present, or if the *exponent letter* E is present, then the default KIND representation will be used. If the *exponent letter* is D, the KIND parameter used will be 8, and if the *exponent letter* is Q, the KIND parameter will be 16. If both an *exponent* and a KIND parameter are specified, the *exponent letter* must be E.

For example:

3.4E-4	0.00034
42.E2	4200
1.234_8	1.234 with approximately 15 digits precision
-2.53Q-300	-2.53×10^{-300} with approximately 34 digits precision

Complex Constants

A complex literal constant has the form:

(*real-part , imaginary-part*)

real-part, imaginary-part are each one of:

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

The KIND parameter of the complex value will correspond to the KIND parameter of the part with the larger storage requirement. For example:

(1.0E2, 2.3E-2)	default complex value
(3.0_8,4.2_4)	complex value with KIND=8

Character Constants

A character literal constant is one of:

- [*kind-parameter_*] '*character-string*'
- [*kind-parameter_*] "*character-string*"
- *[kind-parameter_] 'character-constant'C*

The delimiting characters are not part of the constant. If it is required to place a single quote in a string delimited by single quotes then two single quotes must be used; and similarly for double quotes. For example:

```
1_'A.N.Other'
'Bach''s Preludes'
```
(actual constant is `Bach's Preludes`)

`""` (a zero length constant)

For compatibility with C, Intel Fortran has an extension that allows you to specify a null-terminated character constant. You can use this where a character constant can appear. You must take into account that the last byte of the character entity will be taken up by a zero. For example,

```
CHARACTER*5 STRING
DATA STRING/'1234'C/
```

This works correctly because the fact that the initializing `STRING` as a C character constant means that the last byte is initialized as zero. However,

```
CHARACTER*5 STRING
DATA STRING/'12345'C/
```

will generate a warning because there are too many data values to fit in the five bytes of `STRING`.

Also for compatibility with C usage, Intel Fortran allows the backslash character (\) to be used as an escape character in character strings. You can use a compile-time option, the `/nbs` option (see the *Intel Fortran Compiler User's Guide*) to disable this feature. When the `/nbs` option is not used, the default behavior is to ignore the backslash character, and either substitute an alternative value for the character following, or to interpret the character as a quoted value. The escape characters that are recognized and their effects are described in Table 3-2.

Table 3-2 Escape Characters

Escape Character	Effect
`\n`	newline
`\t`	horizontal tab
`\v`	vertical tab
`\b`	backspace

Table 3-2 Escape Characters (continued)

Escape Character	Effect
\f	form feed
\0	null
\'	apostrophe (does not terminate a string)
\"	double quote (does not terminate a string)
\\	\
\x	*x*, where *x* is any other character

Thus:

'ISN\'T' is a valid string where /nbs is not used.

The backslash is not counted in the length of the string.

If \& appears at the end of a line when the /nbs option is active, the & will not be treated as a continuation indicator.

Logical Constants

The form of a logical literal constant is one of:

- .TRUE. [_kind-parameter]
- .FALSE. [_kind-parameter]

Example

```
.TRUE.
.FALSE._2
```

Typeless Constants

Intel Fortran extends the uses of binary, octal, and hexadecimal constants beyond those prescribed in the Fortran 95 Standard. Binary, octal, and hexadecimal constants (BOZ constants) can be used wherever an intrinsic literal constant of any numeric or logical type is permitted. Intel Fortran also allows Hollerith constants to be used in these contexts and where a character type is required.

Extended Use of BOZ Constants

The format of BOZ constants is described in "BOZ Constants", page 3-12.

If possible, the type attached to a typeless constant is derived from the magnitude of the constant and the context in which it appears. When used as one operand of a binary operator, it assumes the type of the other operand. If it is used as the right-hand side of an assignment, the type of the object on the left-hand side is assumed. When used to define the value within a structure constructor, it assumes the type of the corresponding component. If appearing in an array constructor, it assumes the type of the first element of the constructor.

Further rules:

- If the context does not determine the type, a warning is issued and the type attached to the constant is:
  - — `INTEGER(4)` if the constant occupies 1-4 bytes.
  - — `INTEGER(8)` if the constant occupies more than 4 bytes.

  Leading zeros are considered significant in determining the size.

  For example:

  `Z'00000001'` assumes `INTEGER(4)`

  `Z'000000001'` assumes `INTEGER(8)`

The compiler truncates and issues a warning about constants that can only be represented by more than 8 bytes (for example, `Z'12345678123456781234'`). The resulting truncated value differs from that specified in the source code.

- When the size of type determined by context does not match the size of the actual constant, the constant is either extended with zeros on the left or truncated from the left as necessary.
- If a single constant is assigned to a complex entity, it is assumed to represent the real part only and will assume the real type with the same length as the complex entity.
- In user generic procedure resolution (see Chapter 7 for details), an actual argument that is a BOZ constant is considered to match a logical or numeric dummy argument; however, an ambiguous reference is likely to occur.
- Except for the intrinsic conversion procedures, a BOZ constant used as an actual argument for an intrinsic procedure assumes the integer type.

- The intrinsic functions INT, LOGICAL, REAL, DBLE, DREAL, CMPLX, and DCMPLX are available to cast a BOZ constant to a specific type. If a BOZ constant is given as argument *arg* to these functions, the type assumed for *arg* is as follows:
  - For functions INT and LOGICAL the assumed type will be respectively INTEGER(KIND=*a*) and LOGICAL(z=*a*), where *a* is 4 if the constant occupies 1 to 4 bytes, and 8 otherwise.
  - For the functions REAL, DBLE, DREAL, CMPLX, and DCMPLX an argument of type REAL(KIND=*b*) is assumed, where *b* is 4 if the constant occupies 1 to 4 bytes, 8 if it occupies 5 to 8 bytes, and 16 otherwise.

Examples

`Z'4A1'`	Hexadecimal constant is `INTEGER(4)`.
`10_2 + Z'1000A'`	The value is `20` (constant treated as `INTEGER(2)` and truncated on the left).
`LOGICAL(2) :: lgl2` `lgl2 = B'1'`	Constant treated as `LOGICAL(2)`, the type of the variable.
`ABS(Z'41')`	Constant treated as `INTEGER(4)`; `IABS` is used.
`REAL(Z'3FF0000000000000')`	Constant treated as `REAL(8)` as it is more than 4 bytes.

Hollerith Constants

Hollerith constants have the format:

lenHstring where

len	is the number of characters in the constant and
string	contains exactly *len* characters. The value of the constant is the value of the pattern of bytes generated by the ASCII values of the characters.

For example:

```
3HABC
5HABCbb
```

bb represents two space characters, to make the length equal to 5.

Hollerith constants may appear anywhere that a BOZ constant can appear, and additionally where a character string is valid. When there is a mismatch in lengths the constant will be truncated on the right, or padded on the right with space characters.

If a Hollerith constant is used as an argument to the conversion functions INT and LOGICAL, KIND=1 and KIND=2 are added as possible values for KIND=*a* (see the BOZ rules earlier in this section); these apply when the length of the constant is 1 or 2 characters/bytes.

Character Substrings

A *character-substring* is a contiguous portion of a scalar character entity, referred to as the *parent-string*. The substring is defined by giving the character positions of its start and end. The format is:

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

starting-position is a scalar expression. If *starting-position* is omitted, a value of 1 is assumed. The *starting-position* must be greater than or equal to 1 unless the substring has zero length.

ending-position is a scalar integer expression. If *ending-position* is omitted the value of the length of the character string is assumed.

The length of the substring is:

MAX (*ending-position - starting-position* + 1, 0)

Example

```
'ABCDEFGH' (3:5)
is a character substring of length 3 equal to
'CDE'
'ABC' (-1 : 2)
is invalid.
'ABC'(2: -1)
has a zero length.
```

Derived-type Definition

The format of a derived-type definition is:

```
TYPE [[, access-spec] ::] type-name
   [private-sequence-statement] ...
   component-definition-statement
   [component-definition-statement] ...
END TYPE [type-name]
```

access-spec Specifies one of the following:

```
PRIVATE
PUBLIC
```

type-name is the name of the type being defined. *type-name* must not conflict with the intrinsic type names.

private-sequence- is a PRIVATE statement or a SEQUENCE statement.

The use of PRIVATE and PUBLIC is only allowed if the type definition is within a module. Their use is explained in "PRIVATE (Statement and Attribute)," and "SEQUENCE" in Chapter 10.

The SEQUENCE statement is explained below.

component-definition-statement is:

```
type-spec [[component-attr-list]::]
component-declaration
```

component-attr-list can only contain the DIMENSION and POINTER attributes.

component-declaration is:

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

where *component-initialization* has one of the following forms:

```
 = initialization-expression
=> NULL()
```

The first form of *component-initialization* is used for components that are not of POINTER type.

The second form of *component-initialization* is used for POINTER components, and indicates that the pointer has an initial status of *disassociated* (or nullified).

The ability to initialize a pointer in this manner is useful because there are several instances in the language where a pointer may not be used unless it has a defined association status.

A component array without the POINTER attribute must have an explicit-shape specification with constant bounds.

The presence of the SEQUENCE statement implies that the components of the type will be arranged in storage in the order in which they are defined. The type is then known as a *sequence derived type*. If all components are of character type it has *character sequence type*, and if all the components are of numeric type it has *numeric sequence type*.

Equivalencing variables of derived type which have different sequence types is a supported extension.

If a component is of the same derived type as the type being defined then the component must have the POINTER attribute.

For example, a singly linked list can be created as a set of “nodes”, each containing a value and a pointer to the next node. A type node can be defined as follows:

```
TYPE node
  INTEGER :: value
  TYPE(node), POINTER :: next
  ! next must have the POINTER
  ! attribute
END TYPE node
```

Structure Constructor

A *structure constructor* specifies a scalar value for a derived type by specifying the values for the components in the order that they appear in the definition. For example:

```
TYPE employee
  CHARACTER(LEN=30) :: surname
  CHARACTER(LEN=20) :: firstname
  INTEGER :: id
END TYPE employee

TYPE(employee) :: programmers(30)
TYPE(employee) :: &
  robjones=employee('Jones','Rob',20)
programmers(1) = employee('Smith','John',34)
```

`employee('Smith','John',34)` is a structure constructor that contains the values assigned to the components of the first element of the array `programmers` by the final statement above.

Note that the name of the type must precede the parenthesized data values of the components, a value must be present for each component, and objects of derived type may be initialized by a structure constructor in a derived-type declaration.

Implicit and Explicit Typing

If an entity is declared or used without being explicitly typed, then the entity's type will be determined from the initial symbol of its name, known as implicit typing. The default implicit typing rules are as follows:

- Names with initial letter `A` to `H` or `O` to `Z`: `REAL`
- Names with initial letter `I` to `N`: `INTEGER`

Thus:

```
DIMENSION a(5), i(10)
k=1
b=k
```

implicitly declares *a* and *b* as default reals and `i` and `k` as default integers.

Do not use implicit typing if you can avoid it because your types can be converted to a type that you do not want. Instead explicitly type all entities using declaration statements. Implicit typing can be disabled with the IMPLICIT NONE statement, as described below, ensuring that any appearance of an entity that has not appeared in an explicit type declaration statement will be the subject of an error message and render the program invalid.

IMPLICIT Statement

The IMPLICIT statement provides a means of changing or canceling the default implicit typing. This takes effect for the scoping unit in which it appears, except where overridden by explicit type statements.

The statement is one of:

- IMPLICIT NONE
- IMPLICIT *implicit-spec-list*

implicit-spec-list is:

type-spec (*letter-spec-list*)

letter-spec is one of:

letter
letter - letter

IMPLICIT NONE overrides the predefined implicit type specification. If this statement is included in a scoping unit then all the names in that unit must have their types explicitly declared. It must appear before any PARAMETER statements. A scoping unit that includes an IMPLICIT NONE statement may not include any other IMPLICIT statements. A command-line option, the /4{Y|N}d option, can be specified that has the effect of including an IMPLICIT NONE statement in every program unit (See the *Intel Fortran Compiler User's Guide*).

Examples

```
IMPLICIT NONE
! Enforce explicit typing
IMPLICIT REAL(a-h,o-z),INTEGER(i-n)
! This is equivalent to the default typing:
! a through h and o through z implies REAL
```

```
! i through n implies INTEGER

IMPLICIT REAL(KIND=8)(d),COMPLEX(8)(z)
! d implies REAL(8) z implies COMPLEX(8);
! other letters retain any assigned types

IMPLICIT TYPE(node)(l,n)
! Derived types can be included
```

A scoping unit may contain more than one "active" IMPLICIT statement, but any letter must be included in only one *letter-spec*. IMPLICIT statements must precede all other specification statements except PARAMETER statements. The IMPLICIT statement has no effect on the default types of intrinsic functions.

The implicit rules of a host scoping unit will apply to a contained scoping unit, but can be completely or partially overridden by implicit statements within the contained scoping unit.

Data Initialization

Compile-time data initialization can be carried out using type declaration statements and DATA statements. The format of the DATA statement is fully described in "DATA" in Chapter 10, where the initialization formats for each of the intrinsic type declarations are also specified.

Examples

```
INTEGER i
LOGICAL test
CHARACTER(LEN=10):: string
REAL,DIMENSION(2,4) :: array
COMPLEX,DIMENSION(3) :: zz

DATA i,test,string/21,.TRUE.,'10 letters'/
DATA zz/3*(1.0,2.0)/  ! Using a repeat factor.
DATA ((array(i,j),i=1,2),j=1,4)/1.0,2*2.0,5*3.0/
! Using an implied DO loop.
```

```
! i = 21 test = .TRUE.
! string = '10 letters'
! All elements of zz = (1.0,2.0)
! The 8 elements of array are
! 1.0 2.0 3.0 3.0
!  2.0 3.0 3.0 3.0
```

Storage Association and Alignment

In general, no assumptions about the relative storage locations of any entities can be made. The use of `COMMON` and `EQUIVALENCE` statements enable storage association to be established. The detailed syntax and description of these statements is given in Chapter 10.

The `COMMON` statement enables common blocks of storage to be established. The use of a `COMMON` statement referring to the same common block in more than one program unit ensures that the same storage locations are referenced in each of the program units.

The `EQUIVALENCE` statement enables more than one name to be given to the same storage location within a program unit. Where common block elements or array elements are referenced, restrictions apply because of the imposed sequencing of these elements within storage.

The `SEQUENCE` statement appears only in a derived-type definition (see "Derived-type Definition" on page 20). It enables derived-type variables to be located in common blocks and to be named in `EQUIVALENCE` statements.

Storage Association Alignment Rule

The `COMMON` and `SEQUENCE` `storage` statements enforce an ordering of variables within storage. Association may be established between variables with different type and `KIND` parameters. The general rules for the alignment of variables in storage are as follows:

- A variable will be stored at an address that is a multiple of the alignment required for storage of a scalar variable with the same type and `KIND` parameters (see Table 3-1). This is an extension to the Fortran 95 Standard.

- A sequence derived type will have the same alignment as the component that has the most restrictive alignment requirement.

Examples

The following code illustrates how a sequence of variables would be stored within a derived type.

```
TYPE t
  SEQUENCE
  CHARACTER(LEN=7) :: c
  INTEGER(2) :: i2
  REAL(8) :: r8
  REAL(4) :: r4
END TYPE t

TYPE (t), DIMENSION(5) :: ta
```

Each element of `t` is allocated as in the following table. The first component of `t` starts at an address that is a multiple of 8:

Table 3-3 Example of Structure Storage

Component	Byte Offset	Length
`c`	0	7
`i2`	8	2
`r8`	16	8
`r4`	24	4
padding	28	4

The four trailing padding bytes are necessary to preserve the alignment of `r8` in each element of the array.

Dynamic Data Objects

Allocatable arrays, pointers and automatic objects are allocated dynamically.

Allocatable Arrays

The definition and description of use of allocatable arrays is in "Allocatable Arrays" in Chapter 4.

Pointers

A variable with the `POINTER` attribute is referred to as a pointer. It can be in one of three states: undefined, disassociated, or associated. On entry to a program, all pointers are undefined.

If variable `p` is a pointer:

- `ALLOCATE(p)` acquires storage and associates `p` with this storage, which becomes its target.
- `DEALLOCATE(p)` disassociates `p` from its target (which must have been previously `ALLOCATE`d) and frees the storage occupied by the target.
- `NULLIFY(p)` disassociates `p` from any target but does not alter the status of the target.

The `ASSOCIATED` intrinsic function inquires if a pointer is associated with:

- Any target
- A specific target
- The same target as another pointer

A pointer can also be associated with an existing target using pointer assignment (see Chapter 5 for details). Briefly, `p => t` associates pointer `p` with target `t`. If `t` is a pointer then `p` becomes associated with the target with which `t` is associated.

Cray-Style Pointers

For compatibility with earlier versions of Fortran, Intel Fortran supports Cray-style pointer variables; see Chapter 10, Intel Fortran Statements for the syntax and examples. The use of Cray-style pointers is not recommended.

Automatic Objects

An automatic object is an explicit-shape array or character string whose size is determined by values which are known only on entry to the procedure in which it is declared. It cannot be a dummy argument and cannot possess the SAVE attribute. Its storage space is dynamically allocated upon invocation of the subprogram and is released on return from the subprogram.

Example

```
SUBROUTINE sub(n,...)
  ! a and c are not in the dummy argument list
  INTEGER, INTENT(IN) :: n
  ! n must have a value on entry
  REAL, DIMENSION(n) :: a
  CHARACTER(LEN=n) :: c
  ...
END SUBROUTINE sub
```

Array a is dynamically allocated on entry to the subroutine sub, by which time the value of n has been defined. Similarly, character variable c will be dynamically allocated, with length n. The storage for both of these automatic objects will be released on return from the subroutine.

Records and Structures

Intel Fortran also provides STRUCTURE and RECORD statements to provide compatibility with earlier implementations. The Fortran 95 derived-type (TYPE) feature now provides similar facilities.

The STRUCTURE and RECORD statements are extensions to the Fortran 95 Standard. For details see the "STRUCTURE" and "RECORD" in Chapter 10, Intel Fortran Statements.

Arrays 4

Array processing is a feature of many Fortran programs and one of the major features of Fortran 95 is the ability to process an array as a whole, or in part, rather than on an element-by-element basis as in traditional Fortran. Fortran 95 has also introduced new array categories that include automatic arrays, pointer arrays, arrays that may be allocated dynamically, and functions that return an array result. These new array categories and the concepts introduced to support them are described in this chapter.

New Features

The following is a summary of new array features provided in Fortran 95:

Array categories Fortran 95 provides a number of different categories of arrays: explicit-shape (including automatic and adjustable), assumed-shape, deferred-shape (including allocatable and pointer), and assumed-size. They are each described later in this chapter.

Whole array processing Expressions may contain array operands and be array-valued. They may also contain array sections, which are array-valued. Function results may also be array valued. 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 also be used in array expressions; more details of array expressions appear later in this chapter.

Masked array assignment	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 assignments, only the elements selected by the mask participate in the computation. Masked array assignments are described later in "Masked Array Assignment" in Chapter 5 and under the `WHERE` statement in "WHERE (Statement and Construct)" in Chapter 10.
Intrinsic functions	A number of new intrinsic functions have been provided to manipulate arrays. They are mostly classed as transformational functions.
Array sections	A selected portion of an array, called an array section, can be specified. It is then treated as an array in its own right and can be used as such. The section can be specified by the use of subscript triplets, vector subscripts, or both.
Array substrings	It is possible to attach a substring specifier to the subscript list of a character array; the result is considered to be an array section, that (as noted above) is itself an array.
Array constructors	An array constructor allows an array to be constructed from a list of scalar values and arrays of any rank. An array constructor is a one-dimensional array and can be used wherever such an array is valid. Arrays of higher rank can be constructed by combining an array constructor with the `RESHAPE` intrinsic function.

Zero-sized arrays Fortran 95 has introduced the concept of an array with no elements. These arrays are known as zero-sized arrays and allow certain algorithms to be written naturally without having to allow for edge conditions; more details are given later in this chapter.

Examples of these features are given in the appropriate sections below.

Array Properties

A Fortran array is a single, named entity consisting of a set of objects called array elements, all of the same type and type parameters, arranged in a rectangular pattern of one or more dimensions. An array is therefore said to have the `DIMENSION` attribute, and arrays in Fortran 95 may have up to seven dimensions. An array has the following properties:

rank The number of dimensions of the array. This is fixed for a given array, and is determined from the array declaration. If an object is not an array, then it is said to be scalar and to have rank zero.

lower bound, *upper bound*, *extent* Each array dimension has a *lower bound*, an *upper bound*, and an *extent* that is defined as:

```
MAX (upper bound - lower bound + 1, 0)
```

Bounds are integer valued and may be positive, zero, or negative. Unlike FORTRAN 77, it is permissible for the *lower bound* to be greater than the *upper bound*; if this happens then there are no elements in the dimension and the extent of the dimension is defined to be zero.

size The *size* of an array is the total number of array elements, computed as the product of all its extents. If the extent of any dimension is zero, the size of the array is zero and the array contains no elements. An array with no elements is known as a zero-sized array.

shape The *shape* of an array is a vector of the extents of each dimension of the array; the shape can thus be expressed as a one-dimensional array of size equal to the rank of the array being described. For example, if given the following declarations:

```
REAL    :: a1(10)
INTEGER :: a2(2,4)
LOGICAL :: a3(5,5,0)
COMPLEX :: s1
```

The rank of `a1` is 1 as it only has one dimension, the extent of the single dimension is 10, and the size of `a1` is also 10. `a1` has a shape represented by the vector [10].

`a2` has been declared with two dimensions and consequently has a rank of 2, the extents of the dimensions are 2 and 4 respectively, and the size of `a2` is 8. The vector [2, 4] represents the array's shape.

`a3` has a rank of 3, the extent of the first two dimensions is 5, and the extent of the third dimension is zero. The size of `a3` is the product of all the extents and is therefore zero. The shape of `a3` is [5, 5, 0].

`s1` is a scalar and therefore has a rank of zero, and its shape is represented by an empty vector.

Array Declaration

An object is declared as an array if its declaration includes an array specifier. An array specifier is enclosed in parentheses and defines the rank (number of dimensions), or the rank and shape, of the array and may either follow the `DIMENSION` keyword in a type declaration statement or may follow the declaration of a name.

See "Examples of Type Declarations" in Chapter 3 and "ALLOCATABLE (Statement and Attribute)" in Chapter 10 for descriptions of the statements that can be used to declare arrays.

Syntax

In Fortran 95, an array specifier is used to classify an array as explicit-shape, assumed-shape, deferred-shape, or assumed-size; these different classes of array are discussed later under the section "Array Categories", page 4-7.

The syntax of an array specifier is:

array-spec is either a comma separated list of one of the following:

- *explicit-shape-spec*
- *assumed-shape-spec*
- *deferred-shape-spec*
- *assumed-size-spec*

explicit-shape-spec
[*lower-bound* :] *upper-bound*
assumed-shape-spec
[*lower-bound*] :
deferred-shape-spec
:
assumed-size-spec
[*explicit-shape-spec-list* ,] [*lower-bound* :] *

Each set of bounds defines one dimension of the array, and the number of sets of bounds defines the rank of the array. If a lower bound is not specified then the default lower bound for that dimension is 1.

Examples of Array Specifiers

The following declarations illustrate various forms of an array specifier.

```
REAL :: x(10, 1:5, -2:3)
DIMENSION p(1500)
! x and p have explicit shape, in this example
! the bounds are constant
INTEGER :: ibuff (i:,j:), obuff (:)
! ibuff and obuff are assumed-shape arrays
```

```
INTEGER :: cnts (mdim,ndim)
! an array with an explicit shape, the bounds
! are not constant

COMPLEX, ALLOCATABLE, DIMENSION (:,:) :: coords
! declares an array with deferred shape

REAL, POINTER :: ptr(:,:,:)
! a pointer with deferred shape and a rank of
! three

CHARACTER*5 :: text(10,*)
! the array text has an assumed size
```

Array Element Storage Order

The sequence in which elements in an array are stored in memory (the array element order) is important in certain circumstances, such as:

- Input and output list items
- Internal file I/O
- The DATA statement
- Argument association involving assumed-size or explicit-shape arrays
- Certain intrinsic functions (for example, RESHAPE, TRANSFER, PACK, and UNPACK)
- Array constants in array constructors
- Storage association (for example, as entailed by use of the COMMON or EQUIVALENCE statements)

Array elements are stored in column major order — that is, the order is columnwise: 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 an array declared with the bounds (3,2) is (1,1), (2,1), (3,1), (1,2), (2,2), (3,2).

In general, for an array a declared as

```
DIMENSION a(1:u1, 1:u2, 1:u3
```

the position of array element a(*s1,s2,s3*) is given by the formula

```
s1 + (s2-1) x u1 + (s3-1) x u1 x u2
```

If the array has more dimensions, the formula is extended accordingly, as implied by its structure. If the lower bound of any dimension is not 1, then the formula has to be elaborated slightly, but the general form is unaffected.

Notice that the upper bound of the rightmost dimension (*u3*) does not appear. An assumed-size array, described below, is characterized in its declaration by the rightmost upper bound being given as an asterisk (*). This is possible because its value is not needed in order to compute the position of any array element.

Array Categories

There are several different categories of arrays in Fortran 95. Each category is based on the shape of the array as defined by its array specifier.

Explicit-shape Arrays

An *explicit-shape array* has explicitly declared bounds for each dimension; they are neither taken from an actual array argument (“assumed”) nor otherwise specified later, prior to use (“deferred”). Each dimension of an explicit-shape array is of the form:

```
[lower bound:] upper bound
```

For a given dimension, the values of the lower bound and upper bound define the range of the array in that dimension. The bounds may be positive, negative, or zero. Normally the lower bound will be less than the upper bound; if the lower bound is the same as the upper bound then that dimension will contain only one element; if it is greater, then the dimension contains no elements, the extent of the dimension will be zero, and the array will be zero-sized. If a lower bound is not specified then it will assume the default value of 1.

More generally, the bounds of a dimension may be any specification expression. A specification expression is always a scalar and of type integer; it is either a constant expression, or one in which all variables are available at the time the subprogram is activated. Chapter 5, "Expressions and Assignment," describes specification expressions in more detail.

There are various forms of explicit-shape array; the simplest form is represented by an array declaration in which the name of the array is not a dummy argument and all the bounds are constant expressions. This form of array may have the SAVE attribute and you can declare it in any program unit.

An automatic array is an explicit-shape array that is not a dummy argument, and which has at least one nonconstant bound. You can declare automatic arrays in a subroutine or function, but they may not have the SAVE attribute nor can they be initialized. Large automatic arrays may adversely affect the performance of your programs on Windows*-based operating systems.

A dummy array is identified by the appearance of its name in a dummy argument list; its bounds may be constants or expressions. Dummy arrays can only be declared in a subroutine or function.

An adjustable array is a particular form of a dummy array; its name is specified in a dummy argument list but at least one of its bounds is a nonconstant specification expression.

Explicit-shape arrays may also be used as function results; these are described in the section "Array Functions," page 4-34, and also in Chapter 7, Program Units and Procedures.

Example

The subroutine below demonstrates how explicit-shape arrays may be declared.

```
SUBROUTINE sort(list1,list2,m,n)
!  examples of arrays with explicit shape
INTEGER :: m,n
INTEGER :: cnt1(2:99)
! a rank-one array, having an explicit shape
! represented by the vector [ 98 ]
```

```
REAL :: list1(100), list2(0:m-1,-m:n)
! two dummy arrays with explicit shape, list1
! is a rank-one array with an extent of 100 and
! list2 is a rank-two array with an extent of
! m * (m+n+1). Note that list2 is also an
! adjustable array

REAL :: work(100,n)
! work is an automatic array as it does not
! appear in the dummy argument list and its
! bounds are not constant

INTEGER, PARAMETER :: buffsize = 0
REAL :: buffer (1: buffsize)
! the array buffer has explicit shape, in this
! example however it has no elements and is
! zero-sized
...
END SUBROUTINE SORT
```

Assumed-shape Arrays

An assumed-shape array is a dummy argument that assumes the shape of the corresponding actual argument. This should be compared with an explicit-shape dummy array in which the shape of the array is specified locally.

Each dimension of an assumed-shape array has the form:

[*lower bound*] :

where

lower bound is a specification expression; it can be omitted and would then take the default value of 1. Note that it is the shape of the actual argument that is assumed and not its bounds and that the actual and dummy argument may have different lower (and upper) bounds for each dimension.

An assumed-shape array subscript may extend from the specified *lower bound* to an upper bound that is equal to the lower bound plus the extent in that dimension of the actual argument minus one.

A procedure that declares an assumed-shape dummy argument must have an explicit interface in the calling program unit; this is explained more thoroughly in Chapter 7, "Program Units and Procedures."

Examples

The subroutine below demonstrates various forms of an assumed-shape array declaration.

```
SUBROUTINE initialize (a,b,c,n)

! examples of assumed-shape arrays

INTEGER :: n

INTEGER :: a(:)
! the array a is a rank-one assumed-shape array,
! it assumes (or inherits) its shape and size
! from the corresponding actual argument; its
! lower bound is 1 regardless of the lower bound
! defined for the actual argument

COMPLEX :: b(ABS(n):)
! a rank-one assumed-shape array, the lower
! bound is ABS(n) and the upper bound will be
! the lower bound plus the extent of the
! corresponding actual argument minus one
```

```
REAL, DIMENSION(:,:,:,:,:) :: c
! an assumed-shape array with 5 dimensions
! (rank=5), and all the lower bounds are 1

...

END SUBROUTINE initialize
```

As mentioned previously, if a procedure has an argument that is an assumed-shape array, its interface must be known to the calling program unit. For example, if subroutine `initialize` is an external subroutine, then it must appear in an interface block as follows:

```
PROGRAM main

INTEGER :: parts(0:100)
COMPLEX :: coeffs(100)
REAL  :: omega(-2:+3, -1:+3, 0:3, 1:3, 2:3)

INTERFACE

  SUBROUTINE initialize (a,b,c,n)
  INTEGER :: n
  INTEGER :: a(:)
  COMPLEX :: b(ABS(n):)
  REAL, DIMENSION(:,:,:,:,:) :: c
END SUBROUTINE initialize

END INTERFACE

CALL initialize &
(parts,coeffs,omega,lbound(omega,1))

...

END PROGRAM main
```

Interface blocks are described further in Chapters 7 and 10.

Deferred-shape Arrays

A deferred-shape array is either an allocatable array or it is a pointer array. The array specification for a deferred-shape array is of the form:

```
: [ , : ] ...
```

It defines the rank of the array but not the bounds. The array is therefore said to have deferred-shape.

The shape of the array becomes defined either when the array is allocated or when a pointer array becomes associated with a target. Note that the form of array specifier for assumed-shape arrays and deferred-shape arrays is similar, but a deferred-shape array has either the `ALLOCATABLE` attribute which defines an allocatable array or it has the `POINTER` attribute which defines a pointer array; an assumed-shape array may have neither of these attributes.

Pointer Arrays

A pointer array is an array that has the `POINTER` attribute and may therefore be used to point to some target object. Initially a pointer array has no shape and may not be referenced until it becomes associated either through an `ALLOCATE` statement or through a pointer assignment statement.

Chapter 3, Data Types and Data Objects describes in more detail the concept of pointers and how they may become associated, and disassociated, while "POINTER (Statement and Attribute)" in Chapter 10 explains how the `POINTER` statement may be used to declare a pointer.

Once a pointer array has become associated you can use it in any context in which an array is allowed. Note that a pointer array is not an array of pointers; that is, its elements do not have the `POINTER` attribute. To create an array of pointers, define a derived type consisting of a single pointer component and declare an array of this derived type.

Examples

The following declarations illustrate the concepts associated with declaring a pointer array.

```
REAL, POINTER, DIMENSION(:) :: p1
! p1 is declared as a pointer to a rank-one
! array of type real, p1 is not associated
! with any target

INTEGER, POINTER :: p2(:,:)
! p2 is a pointer to an integer array of
!rank-two,
! p2 must be associated with a target before it
! can be referenced

TYPE err_type
  INTEGER :: class
  REAL :: code
END TYPE err_type
TYPE(err_type), POINTER, DIMENSION(:,:,:) :: err
! err is a pointer to a rank-3 array of type
! err_type

INTEGER, POINTER :: p3(n)
! this is ILLEGAL, pointers cannot have an
! explicit shape
```

Allocatable Arrays

An allocatable array has only its name and rank declared at compile-time, plus the ALLOCATABLE attribute. It can be allocated and deallocated as required by use of the ALLOCATE and DEALLOCATE statements. These statements give the user the ability to manage space dynamically at execution time.

The ALLOCATABLE statement and attribute, the ALLOCATE statement, and the DEALLOCATE statement are described in Chapter 10.

An allocatable array has an allocation status which is initially set to *not-allocated*. The array may not be referenced while it is in this state except as an argument to the ALLOCATED intrinsic inquiry function, which may be used to determine the allocation status of an allocatable array. Once the allocatable array is allocated, its allocation status becomes *allocated* and the array may be used in any context in which an array may appear. If an allocatable array is deallocated then its allocation status returns to

not-allocated. It is an error to either allocate an allocatable array whose status is *allocated*, or to deallocate an allocatable array when its status is *not-allocated*.

The allocation status of a local allocatable array that does not have the SAVE attribute becomes undefined if the allocation status of the array is *allocated* when the procedure in which it is defined exits. In Intel Fortran such an array will be automatically deallocated.

Although pointer arrays provide more functionality, allocatable arrays are simpler and provide more opportunities for compiler optimization. When exiting a particular scope, any array that is ALLOCATABLE and is not SAVEd is automatically deallocated. This prevents memory leaks.

Example

The following subroutine contains an example of an allocatable array declaration and uses the ALLOCATED intrinsic function to illustrate how its allocation status may change.

```
SUBROUTINE foo

! demonstrate the use of an allocatable array

REAL, ALLOCATABLE, DIMENSION(:,:) :: matrix
! the array matrix is rank-2 allocatable
! array, it has no shape and no storage

INTEGER :: n
LOGICAL :: a1
LOGICAL :: a2
LOGICAL :: a3

a1 = ALLOCATED(matrix)
! a1 is assigned the value .FALSE. as the
! allocation status of the array is
! not allocated
READ *,n
ALLOCATE(matrix(n,n))
```

```
! dynamically create the array matrix; after
! it has been allocated the array will have
! the shape [ n, n ]

a2 = ALLOCATED(matrix)
! a2 is assigned the value .TRUE. as the
! allocatable array does exist and its
! allocation status is therefore allocated

DEALLOCATE (matrix)
a3 = ALLOCATED (matrix)
! a3 is assigned the value .FALSE. as the
! allocation status of the array is
! not-allocated

END SUBROUTINE foo
```

Assumed-size Arrays

An *assumed-size array* is an older FORTRAN 77 feature that has been modernized in Fortran 90 with the introduction of assumed-shape arrays; the use of assumed-size arrays in new code is discouraged.

An assumed-size array is a dummy argument whose size is not specified; this is in contrast to an explicit-shape dummy array where the extents of each dimension are specified, and an assumed-shape array where the extents of each dimension are assumed from the corresponding actual argument. The form of an assumed-size array specifier is the same as for an explicit-shape array except that the upper bound of the last dimension is an asterisk (*).

All dummy array arguments and their corresponding actual argument share the same initial element and are storage-associated. In the case of explicit-shape and assumed-size arrays, the actual and dummy array do not have to have the same shape or even rank. However the size of the dummy array must not exceed the size of the actual argument. Therefore a subscript

in the last dimension of an assumed-size array may extend from the lower bound to another value, providing that the value does not cause the reference to go beyond the storage associated with the actual argument.

Because the last dimension of an assumed-size array has no upper bound, the dimension has no extent and the array consequently has no shape. The name of an assumed-size array therefore cannot be used in contexts in which a shape is required, such as the name of a function result or in a whole array reference.

Example

The example below shows how an assumed-size array may be declared.

```
SUBROUTINE foo(a,n)
! an example of an assumed-size array
INTEGER :: n
REAL :: a(n,3:*)
! declares a to be a rank-two array, the array
! has no shape and its size must not be greater
! than the size of associated dummy argument;
! the bounds of the first dimension range from 1
! through to n, the lower bound of the second
! dimension starts at 3, and its upper bound is
! not specified
...
END SUBROUTINE foo
```

Whole Arrays and Array Subobjects

An array may be referred to either as a whole or in part. Any part of an array that may be referenced independently of other parts of the array is known as a subobject of the array, and includes either an array element or an array section. These terms are explained below.

Array Elements

An individual element of an array is a scalar and has the same type and type parameters as the array; an element of an array may be referred to by an array element reference that takes the form of the array name followed by a subscript list enclosed in parentheses. A subscript list is an ordered set of subscript expressions separated by commas, one expression for each array dimension. Each subscript expression must be scalar and of type integer and must have a value that lies within the declared bounds for that dimension.

Intel Fortran also allows a subscript expression of type real; the expression will automatically be converted to type integer after it has been evaluated.

An array element may be used in any expression in which a scalar is allowed.

Example

The example below declares various arrays and then shows how elements of these arrays may be referenced. The example also contains some invalid array element references and explains why they are illegal.

```
SUBROUTINE foo(a,b,c,n)

INTEGER :: n
INTEGER :: a(:)
! a is an assumed-shape array
REAL  :: b(-100:n)
! b is an adjustable dummy array
REAL  :: c(100,100)
! c is an explicit-shape dummy array

REAL, ALLOCATABLE :: d(:,:,:)
! d is an allocatable array
REAL, POINTER   :: e(:,:)
! e is a pointer to a rank-2 array
```

```
REAL, TARGET   :: f(5,5)
! f is an explicit-shape array with rank 2 and
! size 25
INTEGER :: i,j

! examples of valid array element references

a(1) = 100b(a(n)) = b(a(n)) + ABS(c(10*i,j)) /
ABS(a(n))

ALLOCATE(d(10,10,n))
! allocate d with shape [10, 10, n]
d(5,5,n) = LOG10 (n)

e => f
! associate pointer e with array f
e(1,1) = n
! assign n to the first array element of e,
! this is equivalent to assigning n to f(1,1)

! examples of INVALID array element references

a(0) = 100
! illegal - a reference outside the array
! bounds, the default lower bound of an
! assumed shape array is 1

c(100) = 123
! illegal - the array has a rank of two but
only! one subscript has been specified

c(101,1) = 0.0
! illegal - the subscript 101 is outside the
! bounds of its dimension

END SUBROUTINE foo
```

Whole Arrays

All the elements of an array are referenced if the array name is used without any bracketed subscript list; this is known as a whole array reference. Whole array references may be used in such contexts as input/output statements and argument lists, and also in any array-valued expression. An array-valued expression is an expression whose value is an array and may be formed from operations involving arrays; this is discussed in more detail in a following section.

Example

The subroutine in the example below illustrates various contexts in which a whole array reference may be used. It also contains some examples of simple array operations, which will be explained later in the section "Array Expressions".

```
SUBROUTINE change (a,b)

! examples of references to a whole array

REAL, DIMENSION(:,:) :: a,b
! declare a and b to be rank-two assumed-shape
! arrays

REAL, ALLOCATABLE  :: temp(:,:)
! temp is an allocatable array, it will be
! assigned storage below

ALLOCATE(TEMP(SIZE(a,1),SIZE(a,2))
! create temp with the same shape as the
! assumed-shape array a
temp = a
! copy all of array a into temp

a = b - 1.0
! subtract 1.0 from each element of b and
! assign to the corresponding element of a
```

```
b = b - temp
! decrement each element of b by the
! corresponding element in temp

WRITE(*,*) b
DEALLOCATE(temp)

END SUBROUTINE change
```

Array Sections

The term array section is used in Fortran 95 to denote an array that is a selected portion of another array, known as the parent. An array section is an array even if it consists of only one element (or possibly none) and can therefore be specified wherever an array name may be specified.

In Fortran 95, an array section may be defined:

- By a section subscript list
- By an array of derived-type components
- By an array of character substrings

Each of these will be described in turn below.

Section Subscript List

There are two subscript forms used to describe a section: subscript triplets and vector subscripts.

- The subscript triplet notation enables a lower bound, an upper bound, and a stride to be specified for any dimension of the parent array. A subscript triplet selects elements in a regular manner from a dimension; the stride can, for example, be used to select every second element.
- A vector subscript is any expression that results in a rank-one integer value; the values of the array select the corresponding elements of the parent array for a given dimension. Vector subscripts can be used to describe an irregular pattern and may be useful for indirect array addressing such an indexing by a table.

Syntax

An array section reference using a section subscript list is:

array-name (section-subscript-list)

section-subscript-list - is a comma-separated list of *section-subscript*.

section-subscript is one of:

- subscript
- subscript-triplet
- vector-subscript

subscript is:

scalar-integer-expression

subscript-triplet is

[subscript] : [subscript] [: stride]

stride is

scalar-integer-expression

vector-subscript is a rank-one integer array expression.

A *section-subscript-list* must specify a *section-subscript* for each dimension of the parent array. The rank of the array section is the number of *subscript-triplets* and *vector-subscripts* that appear in the *section-subscript-list*; and because an array section is also an array, at least one *subscript-triplet* or *vector-subscript* must be specified.

Subscript triplet

The first subscript of a subscript triplet specifies the lower bound for the dimension, the second subscript specifies the upper bound, and the stride defines the increment between subscript values. All three components of a subscript triplet are optional; if a bound is left out, then that bound is taken from the parent array; if the stride is omitted, then the increment between subscript values is assumed to be one. However, you must specify an upper bound if a subscript triplet is used in the last dimension of an assumed-sized array.

The stride must not be zero; if it is positive then the subscripts range from the lower bound up to and including the upper bound, in steps of stride. Note that when the difference between the upper bound and lower bound is not a multiple of the stride then the last subscript value selected by the subscript triplet will be the largest integer value that is not greater than the upper bound; thus the array expression `a(1: 9: 3)` will select subscripts 1, 4, and 7 from `a`.

It therefore follows that a bound in a subscript triplet need not be within the declared bounds for that dimension of the parent array so long as all the elements selected are within its declared bounds.

Strides may be negative as well as positive. A negative stride selects elements from the parent array starting at the lower bound and proceeds backwards through the parent array in steps of the stride down the last value that is greater than the upper bound. For example, the expression `a(9: 1:- 3)` will select the subscripts 9, 6, and 3 in that order from `a`.

If the section bounds are such that no elements are selected in a dimension, the section has zero-size; for example, the section `a(2:1)`.

Example

The following example shows the power of the subscript triplet notation in assigning the same value to a regular pattern of array elements.

```
INTEGER, DIMENSION(3,6) :: x,y,z
! x, y, and z are 3x6 arrays.
x = 0; y = 0; z = 0
! These are whole-array assignments.
x(3,2:4:1) = 1
y(2,2:6:2) = 2
z(1:2,3:6) = 3
! Using subscript triplets, elements of x, y,
!  and z have been assigned, as follows.
!    X         Y         Z
! 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
```

Vector Subscripts

A vector subscript is an array expression that evaluates into a rank-one integer array. The values of the expression represent the subscript value of the elements to be selected. For example, if `v` represents a rank-one array initialized with the values 4, 3, 1, 7, then the array section `a(v)` is a rank-one array composed of the array elements `a(4)`, `a(3)`, `a(1)`, and `a(7)` — in that order. Note that vector subscripts are commonly specified using array constructors that are described in the next section; as an example of these, the expressions `a(v)` and `a((/ 4, 3, 1, 7/))` have the same section of the array `a`.

There are various restrictions associated with the use of an array section with vector subscript; they may not appear:

- On the right hand side of a pointer assignment statement.
- In an I/O statement as an internal file.
- As an actual argument that is associated with a dummy argument declared with `INTENT(OUT)` or `INTENT(INOUT)` or with no `INTENT`.

It is permissible for a vector subscript to specify the same element more than once. When a vector subscript of this form is used to specify an array section, the array section is known as a *`many-one array section`*. An example of a *`many-one array section`* is:

```
a( (/ 4, 3, 4, 7/) )
```

where element 4 has been selected twice. Fortran 95 does not define the order in which elements are selected in array operations and it is therefore illegal for a many-one array section to appear in either an input list or on the left-hand side of an assignment statement.

A vector subscript allows irregular patterns of elements to be selected as opposed to subscript triplets that select elements in a uniform pattern.

Example

The following example illustrates the concept of an array section using a section subscript list.

```
INTEGER, DIMENSION(4) :: m = (/ 2, 3, 8, 1/)
! m is a rank-1 array that has been
! initialized with the values of an array
! constructor
```

```
INTEGER :: i
REAL, DIMENSION(10) :: a = (/ (i*1.1, i=1,10) /)
! a is a rank-1 array that has been
! initialized with the values
! 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9,
! 11.0

REAL, DIMENSION(4,2) :: b
! b is an uninitialized 4x2 array

PRINT *,a(m)
! prints 2.2, 3.3, 8.8, and 1.1

b(:,1) = a( (/ 5, 10, 6, 5/) )
! assigns the values 5.5, 11.0, 6.6, and 5.5
! to the first column of b; this is an example
! of a many-one array section

b(:,2) = b(MIN(m,4),1)
! the vector subscript MIN(m,4) represents a
! rank-1 array with the values 2, 3, 4, 1 and
! the second column of b is assigned with
! 11.0, 6.6, 5.5, 5.5

a(m) = a(m) + 20.0
! increments a(2), a(3), a(8), and a(1) by 20.0

PRINT *,a
! prints 21.1, 22.2, 23.3, 4.4, 5.5, 6.6, 7.7,
! 28.8, 9.9, 11.0
```

Array of Derived-type Components

Chapter 3, Data Types and Data Objects describes how derived types may be declared. They may be scalars or arrays and may contain components which are scalars or arrays of intrinsic type. The components may also be variables of derived type nested to any arbitrary level. Fortran 95 requires that in any variable name reference at most one component of the name has a non-zero rank and a reference to an array of derived-type components is one in which a component of the name, but not the last, is an array. Thus given the declaration:

```
TYPE sampletype
character(8) :: time
real     :: reading
logical   :: flags(4)
END TYPE sampletype

TYPE (sampletype) :: sample(100)
```

then the operands below are examples of a whole array:

```
sample
sample(n)%flags
```

and the following operand is an example of an array of derived-type components:

```
sample%reading
```

This array has a rank of `1` and a size of `100`; it consists of the array elements `sample(1)%reading, sample(2)%reading, ..., sample(100)%reading.`

Similarly the operand below:

```
sample%time
```

represents a `CHARACTER(8)` array, its rank is `1`, its size is `100`, and it is composed of the elements `sample(1)%time`, `sample(2)%time`, `...`, `sample(100)%time.`

This concept may be taken one step further, for although the term `sample%flags` is not allowed, `sample%flags(4)` is valid and represents an array of derived-type components that is composed of all the final elements of the component *`flags`* that belong to the `array sample`.

Example

The following subroutine includes the declaration of the derived-type `sample` and demonstrates some simple uses of an array of derived-type components.

```
SUBROUTINE process(sample,case)

TYPE sampletype
   CHARACTER(8):: time
   REAL    :: reading
   LOGICAL   :: flags(4)
END TYPE sampletype
TYPE (sampletype) :: sample(100)
! declare sample to be a rank-1 dummy array
! of type sampletype, the array has 100
! elements

INTEGER :: case

IF (case == 1) THEN
  sample%reading = 0.0
  ! initializes every component reading of the
  ! array sample to zero
ENDIF

IF (case == 2) THEN
  sample%reading = DIM(sample%reading,0.0)
  ! any negative value in the array
  ! sample%reading is replaced by zero
ENDIF

IF (case == 3) THEN
  WRITE(*,10) &
(sample(i)%time,sample(i)%reading, i=1,100)
  ! prints two columns, the first column
```

```
  ! contains the 100 components time and the
  ! second column contains the 100 components
  ! reading
ENDIF

10 FORMAT(A,E15.7)
END SUBROUTINE process
```

Array of Character Substrings

An array section of type character may also have a substring range specified. An array section of this form is known as an array substring and is composed of array elements whose values only include the specified substring from each corresponding element of the array section.

Example

This example illustrates the concept of an array substring.

```
CHARACTER(11) :: dates(40)
dates(5:10)(8:) = "1996"
```

The variable `dates(5:10)` is an array section that includes elements `5` through to `10` of the parent array `dates`, and the variable `dates(5:10)(8:11)` is also an array section of the array `dates` but only contains the last 4 character positions of the elements `5` through to `10`.

Array Constructors

An array constructor allows a rank-one array to be constructed from a list of scalar values, arrays of any rank, and implied `DO` specifications. The type of an array constructor is taken from the values in the list which must all have the same type and type parameters (including character length), and its shape is taken from the number of values specified.

An array constructor may appear in any context in which a rank-one array expression is allowed. An array with a rank greater than one may be constructed by passing the array constructor to the `RESHAPE` intrinsic function.

If the list contains only constant values, the array constructor may be used to initialize a named constant, or it may be used in an initialization expression in a type declaration statement; however an array constructor may not be used to initialize variables in a DATA statement as this statement may only contain scalar constants.

Syntax

The syntax of an array constructor is:

(/ *array-constructor-value-list* /)

array-constructor-value-list is a comma-separated list of:

array-constructor-value

array-constructor-value is one of:

- *scalar-expression*
- *array-expression*
- *array-constructor-implied-do*

array-constructor-implied-do is:

(*array-constructor-value-list , scalar-int-var-name = &*
scalar-int-expression , scalar-int-expression &
[, *scalar-int-expression*])

If an array expression appears in a value list, it is treated as a succession of values appearing in array element order (see "Array Element Storage Order" on page 6).

The extent of an array constructor is the number of values supplied. If no values were supplied then the array constructor is zero-sized. For example, the size of the following array constructor:

```
(/ ( i, i=10,n) /)
```

depends on the value of the variable n; if the value of the variable is less than 10 then the constructor will contain no values.

Intel Fortran allows the use of the square brackets square brackets ([]) in place of slashed (/ ... /). This notation is an extension to the Fortran 95 Standard.

Examples

```
x = (/19.3, 24.1, 28.6/)
! Array x is assigned three real values.

j = (/4, 10, k(1:5), 2 + l, &
  (m(n), n = -7,-2),16, 1/)
! One vector, consisting of 16 integer values,
! is assigned to j.

a = (/(base(k), k=1,5)/)
! 5 values are assigned.

REAL,DIMENSION(2):: t
PARAMETER (t=(/ 36.0, 37.0/))
! The named constant t is a rank-one array
! initialized with the values 36.0 and 37.0.

z=RESHAPE((/1,2,3,4,5,6,7,8/), (/2,4/) )
! the array constructor is reshaped as 1 3 5 7
!                                      2 4 6 8
! and is then assigned to z

alaska = site("NOME",(/-63,4/))
! An array constructor is used for the second
! component of the structure constructor.
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 /) )
```

As illustrated by the last three examples, an array constructor with implied DOs and the RESHAPE function can be used to construct arrays that cannot be expressed conveniently with alternative notations.

Zero-sized Arrays

The size of an array is the product of the extents of each dimension; if any extent is zero then the array has no size and is known as a zero-sized array. Any of the arrays described in this chapter may be zero-sized apart from assumed-sized arrays, which have no specified size. This concept of a zero-sized arrays is important if a number of algorithms are to be expressed naturally.

Note that while zero-sized arrays have no elements, they do still have a shape and this is important when they are used in array expressions which are described below. Operations involving zero-sized arrays are generally null operations.

Examples

Some examples of zero-sized arrays are given below:

```
INTEGER, PARAMETER :: cases = 0
REAL :: data1(cases,2), data2(cases,3)
! both data1 and data2 are explicit-shape
! arrays and have zero size; data1 has
! shape [ 0 , 2] and data2 has the shape
! [ 0 , 3]
DO i = 1,n+1
  a(i) = SIN(x(i))
  b(i:n) = b(i:n) + a(i)
ENDDO
! the array section b(i:n) becomes a
! zero-sized array in the last iteration of
! the DO loop
WRITE(*,"(9a)",ADVANCE="NO") &
(/ (title(i),i=1,cols) /)
! if the variable cols is less than 1, then
! the array constructor contains no values
! and no data will be output
```

Array Expressions

The preceding sections have primarily been concerned with describing the concepts of arrays in Fortran 95, the various categories of arrays, and the different ways that they may be referenced. An important feature of Fortran 95 is also the ability to use arrays as operands in expressions; for example, in traditional Fortran an expression of the form

```
a + b
```

must include scalars, but in Fortran 95 the variables `a` and `b` may equally be arrays. Operations involving arrays are performed elementally — that is, an equivalent scalar operation is performed on each element of the arrays.

Array operations generally require the arrays involved to be conformable —that is, they must have the same rank (number of dimensions), same shape, and the extents of corresponding dimensions must be the same. Assumed-size arrays therefore may not be used in an array operation, although a section of such an array is allowed. Note that conformability does not require the lower and upper bounds of corresponding dimensions to be the same.

The Fortran array semantics specifies that array operations are conceptually performed in parallel — that is, the result of such an operation must be as if the operation were performed on each element independently and in any order. The practical effect of this is that, because an assignment statement may have the same array on both the left and right-hand sides, the right-hand side is 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.

A scalar may appear in an array expression. The effect is as if the scalar were evaluated and then broadcast to form a conformable array of elements, each having the value of the scalar. Thus a scalar used in an array context is regarded as conformable with the array or arrays involved.

Zero-sized arrays may also be used in an array expression, but while they have no elements they do have a shape and must therefore follow the rule of conformable arrays. Note that scalars are conformable with any array and may therefore be used in an operation involving a zero-sized array.

Example

The following example contains valid and invalid examples of array operations.

```
SUBROUTINE foo(a,b,c)

REAL     :: a(:)
! a is an assumed-shape array with rank-one
REAL, POINTER :: b(:,:)
! b is a pointer to a rank-two array
REAL     :: c(*)
! c is an assumed-size array

REAL, ALLOCATABLE :: d(:)
! d is an allocatable array, its shape can
! only be defined in an ALLOCATE statement

ALLOCATE(d(SIZE(a)))
! creates the array d with the same size as a;
! in this example a and d are conformable as
! they have the same shape

d = a
! copies the array a into d

b = 0.0
! sets each element of the array associated
! with b to 0.0; the effect is as if the scalar
! were broadcast into a temporary array, with
! the same shape as b,that is then assigned
! to the left-hand side
```

```
d = a + d
! corresponding elements of a and d are added
! together and then stored back into the
! corresponding array element of d

d = a + SQRT(d)
! conceptually the operand SQRT(d) is evaluated
! into an intermediate array with the same shape
! as d; each element of the intermediate array
! will be added to the corresponding element of
! a and stored into the corresponding element of
! d

DEALLOCATE(d)

! examples of invalid uses of arrays

a = c
! illegal - c is an assumed-size array and so
! has no shape; an assumed-size array may not be
! used as a whole array operand (except in an
! argument list)

a = a + b
! illegal - the arrays a and b do not have the
! same shape and are therefore not conformable

a = a + d
! illegal - in this example d has already been
! deallocated and may not be referenced
! subsequently

END SUBROUTINE foo
```

Array Functions

Functions may be used in array expressions. As well as returning a scalar result, a function may also be defined to return an array result. Array functions may be used in any array expression provided that they do not appear:

- In an input list
- On the left side of an assignment statement (unless returning the result from within a function)

Array functions may also be used in an array expression wherever a scalar function reference is allowed but must be conformable—that is, the function result must have the same shape as the expression. Functions that return arrays are also known as array-valued functions and may be either:

- Intrinsic functions
- User-defined functions

Intrinsic Functions

The group of functions known as elemental procedures and transformation procedures have particular relevance to array expressions. Elemental procedures are specified for scalar arguments, but when used with an array argument will return an array result with the same shape as its argument(s); each element of the result is as if the function were applied to each corresponding element of the argument. Examples of elemental intrinsic procedures are the mathematical functions `SQRT` and `SIN`.

A transformational procedure on the other hand generally has one or more array arguments that the procedure operates on as a whole, and usually returns an array result whose elements may depend not only on the corresponding elements of the arguments but also on the values of other elements of the arguments. The `RESHAPE` intrinsic mentioned earlier in the chapter is an example of a transformational procedure; other examples are the intrinsic functions `SUM` and `MATMUL`.

User-defined Functions

User-defined functions are not elemental in that they are defined to return either a scalar result or to return an array result; also they cannot be used interchangeably with scalar or array arguments. A scalar function may of course appear in an array expression but the effect, as with any other scalar, is to first broadcast its value throughout a conformable array. A reference to a user-defined array function must obey the rules for functions in general, and must also conform to the shape of the expression in which it appears.

User-defined functions are described in Chapter 7, Program Units and Procedures.

Example

The following example shows how an array-valued function may be referenced.

```
PROGRAM main
! the following interface block describes the
! characteristics of a function genrand; the
! function inputs a single integer scalar and
! returns a real array of rank-one with an
! extent equal to the value of its argument

INTERFACE
  FUNCTION genrand(n)
  INTEGER:: n
  REAL, DIMENSION (n)::genrand
  END FUNCTION genrand
END INTERFACE

REAL :: a(100)
REAL :: b(10,10)

a = genrand(SIZE(a))
! the array a is set to the result returned by
! the function genrand, note that the left
```

```
! and right hand side are conformable.

b = RESHAPE(a + genrand(100),(/ 10, 10 /))
! each element of a is added with the
! corresponding element of the result returned
! by genrand to form an intermediate rank-one
! result that is passed into the intrinsic
! function RESHAPE. In this example, the
! RESHAPE intrinsic transforms its argument
! into a 10 by 10 array; again the left and
! right hand side are conformable.

...

END PROGRAM main
```

Array Inquiry Functions

Fortran 95 has a number of intrinsic inquiry functions that may be used to interrogate the properties of an array. The array need not be defined as these functions examine the array itself rather than its values, but in general, an allocatable array must have been allocated and a pointer array must either be associated with a target or have been explicitly disassociated.

The inquiry functions that can be used to return the properties of an array are:

`ALLOCATED`	interrogates whether an allocatable array is allocated.
`ASSOCIATED`	examines the association status of a pointer to determine whether it is associated with a target.
`LBOUND`	returns either the lower bound of a specific dimension or the lower bounds of the array as a whole.
`SHAPE`	returns the shape of the array as a rank-one integer array.
`SIZE`	returns the size of the array or the extent of a particular dimension.
`UBOUND`	is similar to `LBOUND` but returns an upper bound for a dimension or the upper bounds for all the dimensions of the array.

Expressions and Assignment

5

This chapter describes the syntax and uses of the different forms of expressions and assignments in Fortran 95.

Expressions

An expression can consist of operands, operators, and parentheses, and defines a computation that upon evaluation yields a result. This result can be an operand in a larger expression.

Expressions are used in many contexts in Fortran 95, for example, in assignment statements, in procedure references, and in output statements. An expression has a value and therefore a type and a kind. Expressions are formed from operands and operators that may be intrinsic or user-defined.

An operand may be a constant, variable, array element, array section, structure component, substring, array constructor, structure constructor, function reference, or an expression enclosed in parentheses. Parentheses have the usual mathematical meaning.

An operator can be an intrinsic operator or a user-defined operator. The intrinsic operators are defined within the language; each has a specific meaning for a set of defined operand types. The range of types that an intrinsic operator accepts can be extended, and entirely new operators can be defined, by inclusion of an appropriate interface block and function subprogram definitions. Details are given in Chapter 7, Program Units and Procedures.

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 user-defined
! operators -,.plus., and .times..

(/ 1, 2, 3 /) ** 2 + v
! An array expression, using an array
! constructor.

fcn(x+y) * SUM(aa, DIM=1)
! An expression using the intrinsic
! function SUM and an external function
! fcn.

.NOT. l
! An expression using the unary logical
! .NOT.intrinsic operator.

(3.0, 5.0) - CONJG(cx)
! An expression using a complex constant
! and intrinsic function CONJG.

rational( 1, 2*j) *  rational( i, j )
```

```
! An expression using the structure
! constructor rational and an extended
! definition of the intrinsic operator *.
```

Formation of Expressions

The expressions can be primary, consisting of operands only, and more complex, including both operator(s) and operand(s). In addition, there could be special forms of expressions. All these forms are described in the following sections.

Primary

A primary, the simplest form of expression, consists only of an operand, that can be any of:

- A constant or variable
  `1.0, 'ab', a`
- An array element or array section
  `a(1,3), a(1,2:3)`
- A character substring or structure component
  `ch(1:3), employee%name`
- An array constructor
  `(/1.0,2.0/)`
- A structure constructor
  `employee(8, "Wilson", 123876)`
- A function reference
  `SQRT(x)`
- An expression in parentheses
  `(b + SIN(y)**2)`

When the primary is an array variable, the complete array is referenced. An assumed-size array variable cannot be a primary. An array section of an assumed-size array can be a primary if the extent of the last dimension of the section is defined by the use of a subscript, a section subscript with an extent for the upper bound, or a vector subscript. (See Chapter 4, Arrays for a discussion of arrays.)

If the primary has the `POINTER` attribute, then the target associated with it is used as the operand.

Operators

The more general form of an expression is:

[operand1] operator operand2

If *operand1* is present then the operator is binary (operates on two operands), otherwise it is a unary operator (operates on only one operand). Table 5-1 lists the intrinsic operators and the types of operands for which they have a defined meaning.

Note that:

- The operators +, -, /, *, and ** are used for addition, subtraction, division, multiplication, and exponentiation respectively.
- The operators + and - can be used as unary or binary operators.
- The operator // is used to concatenate two strings.
- The Standard does not allow two adjacent operators. For example, `i + -j` is not valid; this example should be rewritten as `i + (-j)`. However, Intel Fortran does allow the exponentiation operator to be followed by a signed entity, for example, `i ** -j` is permitted and is equivalent to `i ** (-j)`.
- The relational operators `.EQ.`, `.NE.`, and others are used to compare values.
- Logical operators are available to perform Boolean arithmetic; these are `.NOT.`, `.AND.`, `.OR.`, `.EQV.`, and `.NEQV.`. Their behavior is described in Table 5-3.
- As an extension, Intel Fortran also supports the `.XOR.` operator, which is equivalent to `.NEQV.`.

A more detailed description of the interpretation of the operators is given in Table 5-1.

Table 5-1 Intrinsic Operators

Category	Operators	Valid Operand Types
Arithmetic	** * / + -	Numeric, of any combination of types and kind parameters
Character	//	Character, of any length but same kind parameters
Relational	.EQ. .NE. == /=	Both operands of numeric type (mixed kind parameters allowed), or both of character type, with same kind parameters
Relational	.GT. .GE. .LT. .LE. > >= < <=	Both operands of numeric type except complex (mixed kind parameters allowed), or both of character type (same kind parameters)
Logical	.NOT. .AND. .OR. .EQV. .NEQV. .XOR.	Logical (mixed kind parameters)

Precedence of Operators

When an expression expands to:

operand1 operator1 operand2 operator2 operand3 ...

it is necessary to define the order in which the operators will be applied. Each operator is assigned a precedence. The defined order of evaluation is that any subexpressions containing an operator with higher precedence than the adjacent operators will be evaluated first. Where operators are of equal precedence, evaluation will be from left to right, except for the exponentiation operator (**), which is evaluated from right to left. Any expression or subexpression may be enclosed in parentheses; such expressions are always evaluated first using the rules explained above. This usage of parentheses is therefore equivalent to normal mathematical usage. Table 5-2 lists the precedence of the operators; it is followed by some examples.

Table 5-2 Operator Precedence

Precedence	Operators
Highest	User defined unary operators
	**
	* /
	Unary + Unary -
	+ -
	//
	.EQ. .NE. .LT. .LE. .GT. .GE. == /= < <= > >=
	.NOT.
	.AND.
	.OR.
	.EQV. .NEQV. .XOR.
Lowest	User defined binary operators

Examples

```
a+b*c
```
is `a + (b*c)` (* has a higher precedence than +)

```
a/b*c
```
is `(a/b)*c` (/ and * have the same precedence, and evaluation is left to right)

```
a**b**c
```
is `a**(b**c)` (** evaluates right to left)

```
a.AND.b.AND.c.OR.d
```
is `((a.AND.b).AND.c).OR.d)`

Special Forms of Expression

Within certain language constructs only strictly defined forms of expression are permitted. For example, the value of an entity with the PARAMETER attribute—that is, a named constant—may be defined by an expression, but it must be possible to evaluate the expression during compilation—the expression must be an initialization expression, a strictly defined form of constant expression.

The bound of an array that is a dummy array argument in a subprogram may be an expression, but it must be possible to evaluate this expression on entry to the subprogram: the expression must be a specification expression.

Constant expressions, initialization expressions, and specification expressions are defined in the following sections.

Constant Expression

A constant expression is either a constant or an expression containing only intrinsic operators and constant operands. In this context, a constant includes any well-defined part of a constant—for example, a substring with constant start and end points, or an array or structure constructor where all the expressions used are constants or constant expressions. A constant expression can also include references to intrinsic functions that can be fully evaluated at compilation time.

Certain intrinsics cannot be evaluated by the compiler; these are ALLOCATED, ASSOCIATED, and PRESENT, and any inquiry intrinsic with arguments such that the property inquired about (for example type parameters or array bounds) is not constant.

A constant expression may appear in any context in which a general expression may be used.

Examples of a constant expression are:

```
123                     !an integer literal

"Hello " // " World"    ! a character constant
                        ! expression
3.0_single              ! a real literal constant
                        ! where single is a named
```

```
                          ! integer constant
coord(0.0,infinity)       ! a structure constructor
                          ! in which "infinity" is
                          ! a named constant
(/ SQRT(x), x, x*x /)     ! an array constructor in
                          ! which x is a named real
                          ! constant
x*x + 2*x*y + y*y         ! a constant numeric
                          ! expression where x and
                          ! y are named constants
SUM(iterations,DIM=1)     ! reference to a
                          ! transformational
                          ! intrinsic where
                          ! iterations is an
                          ! array-valued named
                          ! constant
SHAPE(matrix)             ! a reference to an
                          ! inquiry intrinsic in
                          ! which "matrix" is an
                          ! array with constant
                          ! bounds
```

Initialization Expression

An initialization expression is a constant expression with the following further restrictions:

- Exponentiation is only allowed if the second operand is an integer.
- Any subexpression used within the expression must be an initialization expression.
- All arguments to intrinsic function references must be initialization expressions.
- Only the following transformational intrinsic functions may be referenced:
  - REPEAT
  - RESHAPE
  - SELECTED_INT_KIND

  - SELECTED_REAL_KIND
  - TRANSFER
  - TRIM
- If you use an inquiry intrinsic, you can only use it to ask about an aspect of an entity that is a compile-time constant. For example, you can use an inquiry intrinsic to ask about the bounds of an array, or a KIND-type parameter.
- Intel Fortran allows references to elemental intrinsic functions with floating-point constants or named constants of a floating-point type, so long as the parameters to the elemental function reference are rank-zero, compile-time constant expressions.

Initialization expressions are required in the following situations:

- When defining values of named constants.
- When specifying a kind parameter in a type specification statement.
- When specifying the KIND dummy argument of a type conversion intrinsic function.
- For initial values in type declaration statements.
- For expressions in structure constructors in DATA statements.
- For case values in CASE statements.
- For subscript expressions or substring ranges in EQUIVALENCE statements.

The following are valid initialization expressions:

```
-456                    ! an integer literal
("Hello "// "World")    ! a character constant
                        ! expression
pi * r ** 2             ! a constant numeric
                        ! expression where
                        ! pi and r are named
                        ! constants
ABS(i * j)              ! a reference to an
                        ! elemental intrinsic in
                        ! which i and j are
                        ! named integer constants
SELECTED_REAL_KIND(7)   ! a reference to a
                        ! transformational
                        ! intrinsic
```

The following are not valid initialization expressions:

```
x ** 2.5                ! x is not a compile-time
                        ! constant
SUM( (/ i, 2 /) )       ! reference to a
                        ! prohibited function
```

Specification Expression

A specification expression is an expression that has a scalar value, is of type integer, and can be evaluated on entry to the scoping unit in which it appears. This imposes the following conditions on primaries used in a specification expression:

- Constants or variables must be available by argument, host, or use association or be in common.
- Any variable referenced must not be a dummy argument with either the `OPTIONAL` attribute or the `INTENT(OUT)` attribute.
- All arguments to intrinsic function references must be specification expressions.
- Elemental intrinsic function references must return integer results.
- Only the following transformational intrinsic functions may be referenced:
  - `SELECTED_INT_KIND`
  - `SELECTED_REAL_KIND`
  - `TRANSFER`
- The inquiry intrinsics `ALLOCATED`, `ASSOCIATED`, and `PRESENT` may not be referenced.
- Other inquiry intrinsics may be referenced provided that the property interrogated is not defined by either a pointer assignment or `ALLOCATE` statement; furthermore, an inquiry intrinsic may not interrogate the following properties of an assumed size array:
  - Upper bound of the last dimension
  - Extent of the last dimension
  - Size of the array
  - Shape of the array

Note that there are some important differences between specification expressions and initialization expressions; the differences are summarized below:

- Initialization expressions
  - — Must be a constant expression
  - — Can be either scalar or array valued
  - — Can be any type
  - — Can reference an inquiry intrinsic (except for `ALLOCATED`, `ASSOCIATED`, and `PRESENT`) to interrogate a property of an entity provided that the property is constant
- Specification expressions
  - — Must be scalar valued
  - — Must be integer type
  - — Can reference variables via host, argument, or use association
  - — Can reference variables in common
  - — Subject to certain restrictions, can reference an inquiry intrinsic (except for `ALLOCATED`, `ASSOCIATED`, and `PRESENT`) to interrogate a property of an entity; the property need not be constant.

Specification expressions may be used where any arbitrary expression is allowed, and they may also be used to declare the bounds of an array and the length of a character variable. Do not use them as follows:

- as subscripts or substring ranges in an `EQUIVALENCE` statement
- in a `CASE` statement
- as a `KIND` parameter in a type declaration statement
- as initial values in a `PARAMETER` or type declaration statement
- as the limits or increment of an implied `DO` loop in a `DATA` statement
- as a `KIND` dummy argument to type conversion intrinsics

Examples of specification expressions are:

```
789                     ! an integer literal
                        ! constant
MAX(m+n,0)              ! m and n are integer
                        ! dummy arguments
LEN(c)                  ! c is a character
```

```
                                ! variable accessible
                                ! via host association
SELECTED_INT_KIND(5)            ! a reference to a
                                ! transformational
                                ! intrinsic
UBOUND(arr,DIM=n)               ! a reference to an array
                                ! inquiry intrinsic in
                                ! which arr is an array
                                ! accessible via USE
                                ! association and n is a
                                ! variable in common
```

Interpretation of Expressions

The expressions can be interpreted differently depending on the type and the KIND type parameters and operator types.

Intrinsic Operators

- Arithmetic operators (+, -, /, *, **)

  The two operands may be of different numeric types or different KIND type parameters. The type of the result is as follows:

  — The type of either operand if the types and KIND type parameters are the same.

  — The type of the operand with the larger KIND type parameter if the types are the same but not the kind type parameters.

  — Complex if either operand is complex and the other is not.

  — Real if either operand is real and the other is not complex.

- Except for a value raised to an integer power, each operand that differs in type or kind type parameter from that of the result is converted to a value with the type and kind type of the result before the operation is performed.

  The arithmetic operators behave as expected, with the following qualifications:

  — The division of an integer by an integer is defined to be the integer closest to the true result that is between zero and the true result.

— Exponentiation of an integer to a negative integer, `i1**i2`, where `i2` is negative, is interpreted as `1/(i1**(-i2))`, where the division is interpreted as described for division of one integer by another.

— If `x1` and `x2` are real with `x1` negative, then `x1**x2` could be an invalid expression, as the result could be complex. Note, however, that `CMPLX(x1)**x2` is valid; the result is the principal value.

- Relational operators (`.EQ., .NE., .GT., .GE., .LT., .LE., ==, /=, >, >=, <, <=`)

  If the operands of a relational operator are numerical expressions with different type or kind type parameters, the operands are converted to the type and kind type parameters that the sum of the operands have, and then they are compared. If the operands are character expressions, the shorter operand is blank padded to the length of the other prior to the comparison. The comparison starts at the first character and proceeds until a character differs or equality is confirmed. The collating sequence is defined in Appendix C.

- Character operators (`//`)

  In a character concatenation operation, each operand must be a character type and have the same kind type parameter. The character length parameter of the result is the sum of the character length parameters of the operands.

- Logical operators (`.AND., .OR., .EQV., .NEQV., .XOR., .NOT.`)

  In a standard conforming program the two operands must be of logical type but may be of different kind type parameters. The type of the result is as follows:

  — the type of either operand if the kind type parameters are the same

  — the type of the operand with the larger kind type parameter if the kind type parameters are not the same

  An operand that differs in kind type from that of the result is converted to a value with the type and kind type of the result before the operation is performed.

As an extension, Intel Fortran permits the operands to be of type `integer`.The behavior of the logical operators is as shown in Table 5-3:

Table 5-3 Logical operators

opnd1	*opnd2*	.AND.	.OR.	.EQV.	.NEQV. .XOR.	.NOT. *opnd1*
.TRUE.	.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.

Intel Fortran accepts `.XOR.` as an alternative notation for `.NEQV.`.

Array Operands

If both operands are arrays, then they must have the same shape. If one operand is a scalar, then it is treated as an array of the same shape as the other operand in which all elements have the value of the scalar. The result of the operation is an array in which each element is the result of applying the operator repeatedly to corresponding elements of the two operands.

Example

```
REAL,DIMENSION(3):: a, b, c

a =  a+ 1.5
! Increases each element of a by 1.5
c = a * b
! It is equivalent to
! DO i = 1,3
! c(i) = a(i) * b(i)
! ENDDO
```

Evaluation of Expressions

The definition of the language allows the compiler to generate code that evaluates an expression by any sequence that produces a result mathematically equivalent to the sequence implied by the Fortran 95 statement. This permits optimization of the code, including, for example, the reordering of expressions and the promotion of common subexpressions.

Because the order of evaluation of an expression is not defined, it is invalid for any function reference within an expression to modify any of the other components appearing within the expression. Thus, for example, `fun(x)+x` is indeterminate if the reference to `fun` modifies the value of the argument `x`.

Logical Operators and Integer Operands

The logical operators can be used with integer operands to perform bit operations. The logical operations are performed for each bit of the binary representations of the integers. When the operands are of different lengths, the shorter is considered to be extended to the length of the other operand as if it were a signed integer, and the result has the length of the longer operand.

The following example shows the use of logical operators to perform bit-masking operations.

```
INTEGER(2) mask2
INTEGER(4) mask4
DATA mask2/ -4 /
DATA mask4/Z"ccc2"/

mask4 = mask4 .NEQV. mask2      !set mask4 to
                                !Z"ffff333e"
mask2 = .NOT. mask4             !set mask2 to
                                !Z"ccc1"
```

Arithmetic Operators and Logical Operands

Logical and integer types can be combined with the arithmetic operators. The logical variable is treated as an integer of equivalent size, and the result of the operation is an integer value. When different lengths of operands are involved, the shorter is considered extended as a signed integer.

The following example shows how logical operands can be used interchangeably with integer operands

```
LOGICAL(1) :: boolean1 = -4
LOGICAL(4) :: boolean4 = 2**16 + 27
INTEGER(1) :: flag1
INTEGER(4) :: flag4
flag4 = boolean4 - boolean1     !set flag4 to
                                !2**16 + 31
IF (boolean4 > 65536) THEN      !an example of
                                !a relational
                                !operator with
                                !a logical
                                !operand
    flag1 = -(boolean4/65536)   !set flag1 to -1
ENDIF
```

Integer and Logical Functions

References to functions are classified as expressions, and Intel Fortran allows integer function results to be used in logical expressions, and also user-defined logical function results to be used in integer expressions.

Bit Manipulation Intrinsics

In general, an integer actual argument may not be used in a reference to a procedure when the corresponding dummy argument is of type logical, nor may a logical actual argument be used when the dummy argument is of type integer. The only relaxation of this rule allowed by Intel Fortran is in calls to bit manipulation intrinsics, when logical and integer arguments may be used interchangeably.

The following code contains a standard-conforming reference to a bit manipulation intrinsic:

```
INTEGER :: mask = 65535
LOGICAL :: is_even = .TRUE.
IF (IAND(mask,1) /= 0) is_even = .FALSE.
```

The following code contains a similar but nonstandard reference supported by Intel Fortran:

```
LOGICAL :: mask = z"ffff"
INTEGER :: is_even = .TRUE.
IF (IAND(mask,1)) is_even = .FALSE.
```

Logical Truth Values

In a standard-conforming program, a logical variable or expression will be `.TRUE.` (the value `1` in Intel Fortran) or `.FALSE.` (the value `0` in Intel Fortran). In nonstandard conforming programs involving logical operators with integer operands or arithmetic operators with logical operands, a logical variable or expression may have a value other than `1` to return `.TRUE.`. In this case, any nonzero value is considered to be `.TRUE.` and a zero value `.FALSE.`.

Typeless Entities

The Fortran 95 Standard defines a specific set of integer literals known collectively as BOZ constants that represent values in binary, octal, or hexadecimal. These constants may be used in DATA statements as initial values. In Intel Fortran, BOZ constants assume a type and kind that is compatible with the context in which they appear, and may be used interchangeably wherever integer, logical, real, or complex literals are allowed.

Intel Fortran allows Hollerith constants to be used in the same contexts as BOZ constants and also wherever a character literal may appear.

BOZ constants and Hollerith constants are collectively known as typeless constants and are described in "Representation of Literal Constants" in Chapter 3; the rules associated with the use of these constants are also described in "Typeless Constants" in Chapter 3.

Assignment

This section discusses assignment statement and two varieties of assignments: pointer and masked array.

Assignment Statement

An assignment statement transfers the value of an expression to a variable.

The syntax of an assignment statement is:

variable = expression

The interpretation of the assignment is defined for the allowed intrinsic type combinations of variables and expressions; these are *intrinsic assignments*. Assignments for additional combinations can be defined by inclusion of the appropriate defined assignment interfaces and corresponding subroutine subprograms, as detailed in "Defined Assignment" in Chapter 7.

Intrinsic Assignment

The variable may be any nonpointer variable or a pointer variable that is associated with a target.

The valid combinations of types for the variable and the expression are given in the following table. The intrinsic functions used to describe the conversions are detailed in the *Intel Fortran Compiler User's Guide*.

Table 5-4 Conversion of variable=expression

Variable Type	*Expression Type*	Conversion
integer	integer, real, or complex	INT(*expression*, KIND(*variable*))
real	integer, real, or complex	REAL(*expression*, KIND(*variable*))
character	character (same kind parameters)	CMPLX(*expression*, KIND(*variable*))
logical	logical	Truncate *expression* if *expression* length is greater than *variable* length; otherwise, pad value assigned to *variable*, with blanks if necessary.
logical	logical	LOGICAL(*expression*, KIND(*variable*))
derived type	same derived type	None

As described in the section "Interpretation of Expressions" on page 12, Intel Fortran allows integer and logical operands to be used interchangeably. Intel Fortran also allows logical expressions to be assigned to integer variables and integer expressions to logical variables. So, from

Table 5-4, a logical expression may also be assigned to real or complex variables, and similarly, a real or complex expression may be assigned to a logical variable.

If the variable is a scalar, the expression must be scalar. If the variable is an array or an array section, the expression must be an array valued expression of the same shape or a scalar. If the variable is an array or an array section, and the expression is a scalar, the value of the expression is assigned to all elements of the variable. If the variable and expression are arrays, the assignment is carried out element by element with no ordering implied.

The expression is evaluated completely before the assignment is started. For example:

```
CHARACTER (LEN=4):: c
c(1:4) = 'abcd'
c(2:4) = c(1:3)
```

sets `c(2:4)` to `"abc"`, not to `"aaa"`, which might result from a left-to-right character-by-character assignment.

If the variable is a pointer, then it must be associated with a target; the value of the expression is assigned to the target.

Examples of Intrinsic Assignment

```
INTEGER ICNT
TYPE CIRCLE
   REAL RADIUS
   REAL X, REAL Y
END TYPE
TYPE (CIRCLE) CIRCLE1, CIRCLE2
REAL AREA, PI
LOGICAL BOOLX, BOOLY, PIXEL(10,10)
INTEGER A(10,5)
INTEGER, DIMENSION (10,10):: MATRIX1, MATRIX2
CHARACTER*3 INITIALS
CHARACTER*10 SURNAME
CHARACTER*20 NAME
```

```
ICNT = ICNT + 1
!example of an integer assignment

CIRCLE1 = CIRCLE2
!example of a derived-type assignment

AREA = PI * CIRCLE%RADIUS**2
!example of a real assignment

PIXEL(X,Y) = BOOLX .AND. BOOLY
!assigns a logical expression to an element of
!the logical array pixel

A(:,1:2) = 0
!first two columns of A are set to zero
MATRIX1 = MATRIX2
!each element of MATRIX2 is assigned to the
!corresponding element of MATRIX1

NAME = INITIALS // SURNAME
!example of a character assignment
```

Pointer Assignment

The pointer assignment statement establishes an association between a pointer object and a target.

The syntax is:

pointer-object=> *target*

subject to the following constraints:

- *pointer-object* is a variable or variable component with the POINTER attribute; if *target* is a variable, it must have the TARGET or POINTER attribute. If *target* is an expression, then it must either be a reference to a function that returns a pointer result or a user-defined operation that returns a pointer result.

- The type, kind parameters and rank of *`pointer-object`* and *`target`* must be the same.
- *`target`* cannot be an assumed-size whole array or an array section with a vector subscript.

If *`target`* is a pointer already associated with a target, then *`pointer-object`* becomes associated with the target of *`target`*. If *`target`* is a pointer that is disassociated or undefined, then *`pointer-object`* inherits the disassociated or undefined status of *`target`*.

Examples of Pointer Assignment

The following examples show association of scalar and array pointers with scalar and array targets.

```
INTEGER, POINTER :: P1, P2, P3(:)
INTEGER, TARGET :: T1, T2(10)
! P1, P2 and P3 are currently undefined.
P1 => T1       ! P1 is associated with T1.
P2 => P1       ! P2 is associated with T1.
               ! P1 remains associated with T1.
P1 => T2(1)    ! P1 is associated with T2(1).
               ! P2 remains associated with T1.
P3 => T2       ! P3 is associated with T2.
P1 => P3(2)    ! P1 is associated with T2(2).
NULLIFY(P1)    ! P1 is disassociated.
P2 => P1       ! Now P2 is also disassociated.
```

Masked Array Assignment

In masked array assignment, a logical array expression, the mask, controls evaluation of the array expressions and assignment to the array variables.

Masked array assignment is provided in Fortran 95 by the `WHERE` statement and the `WHERE` construct.

The syntax of the WHERE statement is:

WHERE (*array-logical-expression*) *array = expression*

array-logical-expression, *array*, and *expression* must all be conformable. The *array-logical-expression* (the mask) is evaluated for each element and the outcome (.TRUE. or .FALSE.) used to determine whether an assignment is made to the corresponding element of *array*.

The syntax of the WHERE construct is:

```
WHERE ( array-logical-expression )
        array = expression
        [array = expression] ...
[ELSEWHERE
        array = expression
        [array = expression] ... ]
END WHERE
```

The WHERE construct is similar to the WHERE statement, but more general in that several *array = expression* clauses can be controlled by one *array-logical-expression*. In addition, an optional ELSEWHERE part of the construct may be used to assign array elements whose corresponding *array-logical-expression* elements evaluate .FALSE..

When a WHERE construct is executed, *array-logical-expression* is evaluated just once and therefore any subsequent assignment in a WHERE block (the block following the WHERE statement), or ELSEWHERE block to an entity of *array-logical-expression* has no effect on the masking. Thereafter, successive assignments in the WHERE block are evaluated in sequence as if they were specified as:

WHERE (*array-logical-expression) array = expression*

Each assignment in the ELSEWHERE is executed as if it were:

WHERE (.NOT.*array-logical-expression) array = expression*

For example, the following WHERE construct:

```
WHERE (a > b)
  a = b
  b = 0
```

```
ELSEWHERE
  b = a
  a = 0
END WHERE
```

is evaluated as if it was specified as:

```
mask = a > b
WHERE (mask) a = b
WHERE (mask) b = 0
WHERE (.NOT.mask) b = a
WHERE (.NOT.mask) a = 0
```

Only assignment statements may appear in a `WHERE` block or an `ELSEWHERE` block. Within a `WHERE` construct, only the `WHERE` statement may be the target of a branch.

Examples of Mask Array Assignment

```
REAL, dimension(5) :: a
WHERE (a > 0.0) a = SQRT(a)
Each positive element of array a is replaced by its
square root.
REAL, DIMENSION(5) :: a
COMPLEX, DIMENSION(5) :: ca
WHERE (a > 0.0)
   ca = CMPLX(0.0)
   a  = SQRT(a)
ELSEWHERE
   ca = SQRT(CMPLX(a))
   a  = 0.0
END WHERE
```

Each positive element of array `a` is replaced by its square root; the remaining elements calculate the complex square roots of their values, which are then stored in the corresponding elements of the complex array `ca`. Note that in the `ELSEWHERE` clause the assignment to array `a` should not appear before the assignment to array `ca`; otherwise, all of `ca` will be set to zero.

The form of a WHERE construct is similar to that of an IF construct, but with the following important difference: no more than one block of an IF construct may be executed, but in a WHERE construct at least one (and possibly both) of the WHERE and ELSEWHERE blocks will be executed. In a WHERE construct, this difference has the effect that results in a WHERE block may feed into, and hence affect, variables used in the ELSEWHERE block. Notice, however, that results generated in an ELSEWHERE block cannot feed back into variables used in the WHERE block.

The following demonstrates how results in a WHERE block could affect assignments in the ELSEWHERE block:

```
REAL, DIMENSION x(100)
WHERE (x(2:100) /= 0.0)
  x(2:100) = 1.0/x(2:100)  !the last 99
                           !non-zero elements
                           !of x are replaced
                           !by their reciprocal
ELSEWHERE
  x(2:100) = x(1:99)       !the last 99 zero
                           !elements of x are
                           !replaced by their
                           !preceding neighbor
                           !which may have
                           !been modified by
                           !the WHERE block
END WHERE
```

Execution Control 6

The normal flow of execution in a Fortran 95 program is sequential: statements execute in the order of their appearance in the program. However, you can alter this flow, using Fortran 95 control constructs and flow control statements.

This chapter describes the operations performed by control constructs and flow control statements. For a full description of each Fortran 95 control statement, see "Statements and Attributes" in Chapter 10. The `WHERE` construct is described in "Masked Array Assignment" in Chapter 5.

Control Constructs and Statement Blocks

A control construct consists of a statement block whose execution logic is defined by one of the following control statements:

- `CASE` statement
- `DO` statement
- `FORALL` statement
- `IF` statement

A statement block is a sequence of statements delimited by a control statement and its corresponding terminal statement. A statement block consists of zero or more statements and can include nested control constructs. However, any nested construct must have its beginning and end within the same statement block.

Although the Fortran Standard forbids transferring control *into* a statement block except by means of its control statement, Intel Fortran allows it. The Fortran Standard does permit transferring control *out of* a statement block. For example, the following `IF` construct contains a `GO TO` statement that legally transfers control to a label that is defined outside the IF construct:

```
  IF (var > 1) THEN
    var1 = 1
  ELSE
    GO TO 2
  END IF
   .
   .
   .
2 var1 = var2
```

The next logical IF statement is nonstandard (but permitted by Intel Fortran) because it would transfer control into the DO construct:

```
IF (.NOT.done) GO TO 4 ! nonstandard!
 .
 .
 .
DO i = 1, 100
  sum = b + c
 4 b = b + 1
END DO
```

The following sections describe the operations performed by the control constructs.

CASE Construct

The CASE construct selects (at most) one out of a number of statement blocks for execution.

```
[ construct-name :] SELECT CASE ( case-expr )
[ CASE ( case-selector ) [ construct-name ]
  statement-block ]
.
.
.
[ CASE DEFAULT [ construct-name ]
  statement-block ]
END SELECT [ construct-name ]
```

Notes on Syntax

- *case-selector* is one of the following:
  - *case-value*
  - *low :*
  - *: high*
  - *low : high*

  For additional information about case-selector, see the description of the CASE statement in "CASE" in Chapter 10.
- a *case-selector* must be mutually exclusive and must agree in type with case-expr.
- case-expr must evaluate to a scalar value and must be an integer, logical, or character type.
- If construct-name is given in the SELECT CASE statement, the same name can appear after any CASE statement within the construct, and must appear in the END CASE statement. The construct name cannot be used as a name for any other entity within the program unit.
- CASE constructs can be nested. Construct names can then be useful in avoiding confusion.
- Although the Standard forbids branching to any statement in a CASE construct other than the initial SELECT CASE statement from outside the construct, Intel Fortran allows it. The Standard allows branching to the END SELECT statement from within the construct.

Execution Logic

The execution sequence of the CASE construct is as follows:

1. *case-expr* is evaluated.
2. The resulting value is compared to each case-selector.
3. If a match is found, the corresponding *statement-block* executes.
4. If no match is found but a CASE DEFAULT statement is present, its statement-block executes.
5. If no match is found and there is no CASE DEFAULT statement, execution of the CASE construct terminates without any block executing.
6. The normal flow of execution resumes with the first executable statement following the END SELECT statement, unless a statement in statement-block transfers control.

Example

The following CASE construct prints an error message according to the value of ios_err:

```
INTEGER :: ios_err
...
SELECT CASE (ios_err)
CASE (:900)
 PRINT *, "Unknown error"
CASE (913)
 PRINT *, "Out of free space"
CASE (963:971)
 PRINT *, "Format error"
CASE (1100:)
 PRINT *, "ISAM error"
CASE DEFAULT
 PRINT *, "Miscellaneous Error"
END SELECT
```

DO Construct

The DO construct repeatedly executes a statement block. The syntax of the DO statement provides two ways to specify the number of times the statement block executes:

- By specifying a loop count.
- By testing a logical expression as a condition for executing each iteration.

You can also omit all control logic from the DO statement, in effect creating an infinite loop. The following sections describe the three variations of the DO construct.

You can use the CYCLE and EXIT statements to alter the execution logic of the DO construct. For information about these statements, see "Flow Control Statements" on page 15.

Counter-controlled DO Loop

A counter-controlled DO loop uses an index variable to determine the number of times the loop executes.

Syntax

```
[ construct-name : ] DO index = init, limit [ , step ]
statement-block
END DO [ construct-name ]
```

Intel Fortran also supports the older, FORTRAN 77-style syntax of the DO loop:

```
DO label index = init, limit [ , step ]
statement-sequence
```

label terminal-statement

A third form, combining elements of the other two, is also supported:

```
[ construct-name : ] DO label index = init, limit [ ,
step ]
```

For a full description of the DO loop syntax—including a list of legal *terminal-statements*—see "DO" in Chapter 10.

Execution Logic

The following execution steps apply to all three syntactic forms, except as noted:

1. The loop becomes active, and *index* is set to *init*.
2. The iteration count is determined by the following expression:

```
MAX( INT ( limit - init + step ) / step, 0 )
```

step is optional, with the default value of 1. It may not be 0.

Note that the iteration count is 0 if either of the following conditions is true:

- *step* (if present) is a positive number and *init* is greater than *limit*.
- *step* is a negative number and *init* is less than *limit*.

3. If the iteration count is 0, the construct becomes inactive and the normal flow of execution resumes with the first executable statement following the `END DO` or terminal statement.
4. The *statement-block* executes. (In the case of the old-style syntactic form, both *statement-sequence* and *terminal-statement* execute.)
5. The iteration count is decremented by *1*, and *index* is incremented by *step*, or by *1* if *step* is not specified.
6. Go to Step 3.

NOTE. *To ensure compatibility with older versions of Fortran, you can use the* `/Q` *command-line option to ensure that, when a counter-controlled* `DO` *loop is encountered during program execution, the body of the loop executes at least once. For more information about this option, see the Intel® Fortran Compiler User's Guide.*

Example

This example uses nested `DO` loops to sort an array into ascending order:

```
INTEGER :: scores(100)
DO i = 1, 99
 DO j = i+1, 100
```

```
  IF (scores(i) > scores(j)) THEN
   temp = scores(i)
   scores(i) = scores(j)
   scores(j) = temp
  END IF
 END DO
END DO
```

The following example uses the older syntactic form. Note that, unlike the newer form, old-style nested `DO` loops can share the same terminal statement:

```
 DO 10 i = 1, 99
  DO 10 j = i+1, 100
   if (scores(i) <= scores(j)) GO TO 10
   temp = scores(i)
   scores(i) = scores(j)
   scores(j) = temp
10 CONTINUE
```

Conditional DO Loop

A conditional `DO` loop uses the `WHILE` syntax to test a logical expression as a condition for executing the next iteration.

Syntax

`[` *`construct-name`*`:] DO WHILE (` *`logical-expression`*`)`

`statement-block`

`END DO [` *`construct-name`*`]`

Intel Fortran also supports the older syntax of the `DO WHILE` loop:

`DO` *`label`* `WHILE (` *`logical-expression`*`)`

`statement-sequence`

`label terminal-statement`

Execution Logic

1. The loop becomes active.
2. The logical-expression is evaluated. If the result of the evaluation is false, the loop becomes inactive, and the normal flow of execution resumes with the first executable statement following the `END DO` statement, or in the old `DO`-loop syntax, the terminal statement.
3. The *`statement-block`* executes. (In the case of the old-style syntactic form, both statement-sequence and terminal-statement execute.)
4. Go to Step 2.

Example

```
! Compute the number of years it takes to
! double the value of an investment earning
! 4% interest per annum
REAL :: money, invest, interest
INTEGER :: years

money = 1000
invest = money
interest = .04
years = 0
DO WHILE (money < 2*invest) ! doubled our money?
 years = years + 1
 money = money + (interest * money)
END DO
PRINT *, "Years =", years
```

Infinite DO Loop

The `DO` statement for the infinite `DO` loop contains no loop control logic. It executes a statement block for an indefinite number of iterations, until it is terminated explicitly by a statement within the block; for example, a `RETURN` or `EXIT` statement.

Syntax

```
[ construct-name:] DO
 statement-block
END DO [ construct-name ]
```

Execution Logic

The execution sequence of an infinite DO loop is as follows:

1. The loop becomes active.
2. statement-block executes.
3. Go to Step 2.

Example

```
! Compute the average of input values;
! press 0 to exit
INTEGER :: i, sum, n
sum = 0
n = 0
average: DO
 PRINT *, 'Enter a new number or 0 to quit'
 READ *, i
 IF (i == 0) EXIT
 sum = sum + i
 n = n + 1
END DO average
PRINT *, 'The average is ', sum/n
```

FORALL Construct and Statement

The FORALL construct and statement are similar to the DO statement, providing indexed repetitive execution of a statement or block of statements. However, when you use FORALL, you are specifying that the operations within a statement on array elements in the body of the FORALL construct may be executed in parallel. The result stored in each array element is independent of the result stored in other elements. FORALL allows indexed parallel assignment of values to an array.

Syntax

The syntax of the FORALL statement is:

```
[construct-name :] FORALL  ( forall-
   triplet-specification-list
   [ , scalar-logical-expression ] )
[ forall-body-construct ]
END FORALL [ forall-construct-name ]
```

The syntax of the FORALL construct is:

```
[construct-name : ] FORALL  ( forall-
   triplet-specification-list
   [ , scalar-logical-expression ] )
forall-assignment-statement
```

where:

forall-construct-name is an optional identifier that must be unique within the program unit

forall-triplet-specification-list is:

index-name = *scalar-integer-expression* : *scalar-integer-expression* : [*scalar-integer-expression*]

forall-body-construct is one of:

an assignment statement
WHERE construct
FORALL construct
FORALL statement

scalar-logical-expression
is a mask value, indicating whether the operation should be carried out on each array element.

forall-assignment-statement
is an assignment statement or a pointer assignment statement

A FORALL construct has more than one statement in the *forall-body-construct*, while a FORALL statement has a single FORTRAN statement as its *forall-body-construct*. A FORALL statement does not need an END FORALL to mark the end of the iterative statement group, since there is only one statement in the group. For example,

```
FORALL (I=1:N) A(I,I) = B(I)
```

Each operation on an array element in a statement of FORALL construct must complete execution before an operation on the same array element in the next sequential statement can begin execution.

Example

The following code —

```
DIMENSION A(10,10), B(10)
REAL A,B,C
DATA A/100*2.0/,B/10*1.0/,C/4.0/
INTEGER I
FORALL(I=1:10:2)
 A(I,I) = A(I,I) + C * B(I)
 A(I,I) = A(I,I) + 1
 B(I) = A(I,1)
END FORALL
PRINT *, B
END
```

produces these results:

```
7.000000     1.000000     2.000000     1.000000     2.000000
1.000000     2.000000     1.000000     2.000000     1.000000
```

The code above is equivalent to:

```
DIMENSION A(10,10), B(10)
REAL A,B,C
DATA A/100*2.0/,B/10*1.0/,C/4.0/
INTEGER I,J
DO I=1,10,2
 A(I,I) = A(I,I) + C * B(I)
 A(I,I) = A(I,I) + 1
```

```
 B(I) = A(I,1)
ENDDO
PRINT *, B
END
```

Usage Rules

Follow these rules when using the FORALL construct and statement:

- With a FORALL construct name, the same identifier must appear on the FORALL and END FORALL statements.
- The index variable for a FORALL statement or construct must be a scalar integer variable. The index variable may be a formal parameter with the INTENT(IN) attribute. If the index variable is INTENT(IN), the increment or decrement of the variable is not reflected on return to the calling routine.
- The *forall-triplet-specification-list* must not contain a reference to any scalar variable from the list in which the expression appears. For example, the following FORALL statement is invalid:

  ```
  FORALL (N= 1:K, K=1,L)  A(N,K) = 0.0
  ```

  This statement is invalid because the limit for the first index triplet is K, and the next part of the specification list uses K as an index variable.
- The FORALL statement may specify a *scalar-logical-expression* that forms a mask. Operations in the FORALL construct are carried out on the array elements whose indices have a .TRUE. value relative to the mask expression. For example:

  ```
  DIMENSION A(100)
  DO I=1,100
   A(I) = REAL(I)
  ENDDO
  FORALL (I=1:100, A(I) > 20.0 .AND. A(I) < 90.0)
   A(I) = A(I) * A(I)
  END FORALL
  PRINT *, A
  END
  ```

This FORALL construct squares each array element of A from A(20) to A(89) inclusive. The mask is based upon the value of the array elements themselves, not the value of the index variable I.

- Any procedure referenced in the *scalar-logical-expression* that forms the mask, or in any statement in the FORALL body construct must be a PURE procedure.
- Within a FORALL body construct, you must not alter the value of the scalar index variable used to control the FORALL construct. This includes changing the value of the index variable by using the same index variable for a nested FORALL construct.

```
FORALL(I=1:10)
  FORALL(I=1:10)
  END FORALL
END FORALL
```

The above construct is invalid.

- A FORALL body construct may not be the target of a GOTO or other branch construct.
- The stride portion of a *forall-triplet-specification-list* may not equal zero.
- You cannot perform a many-to-one assignment within a single statement of a FORALL construct.

For example:

```
DIMENSION F(10),A2(10)
DATA F/1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0/
A2 = F * 3.0
FORALL (J=1:10)
 F((/1,1,2,2,3,3,4,4,5,5/)) = A2(J)
END FORALL
END
```

will yield:

```
badfor.f
   main program
         F((/1,1,2,2,3,3,4,4,5,5/)) = A2(J)
         ^
```

Error 513 at (5:badfor.f) : The FORALL with index J causes more than one assignment to this (sub)object.

Execution Logic

The execution logic of a FORALL statement is as follows:

1. When your program executes a FORALL construct, it first determines the values for the FORALL index variables, then evaluates the mask expression, if there is one, and finally executes the body of the FORALL construct.
2. The expressions forming each component of the *forall-triplet-specification-list* may be evaluated in any order, and if necessary, converted to the type and KIND value of the index variable.
3. If there is no mask expression, it is as if the mask were present, and all values were .TRUE. If the mask is .TRUE. for a particular index value, then that index value is in the set of **active index values** for the FORALL construct.
4. FORALL body constructs are executed in the order in which they appear, for all active index values. The statements executed may be assignment statements, pointer assignment statements, a WHERE construct or statement, or a nested FORALL construct.

IF Construct

The IF construct selects between alternate paths of execution. The executing path is determined by testing logical expressions. At most, one statement block within the IF construct executes.

Syntax

```
[construct-name :] IF (logical-expression1) THEN
  statement-block1
[ELSE IF (logical-expression2) THEN [construct-name]
  statement-block2 ] ...
[ELSE [construct-name]
  statement-block3]
END IF [construct-name]
```

Execution Logic

1. The *logical-expression1* is evaluated. If it is true, *statement-block1* executes.
2. If *logical-expression1* evaluates to false and ELSE IF statements are present, the logical-expression for each ELSE IF statement is evaluated. The first expression to evaluate to true causes the associated statement-block to execute.
3. If all expressions evaluate to false and the ELSE statement is present, its statement-block executes. If the ELSE statement is not present, no statement block within the construct executes.
4. The normal flow of execution resumes with the first executable statement following the END IF statement.

Example

```
! Compare two integer values
IF ( num1 < num2 ) THEN
  PRINT *, "num1 is smaller than num2."
ELSE IF ( num1 > num2 ) THEN
  PRINT *, "num1 is greater than num2."
ELSE
  PRINT *, "The numbers are equal"
END IF
```

Flow Control Statements

Flow control statements alter the normal flow of program execution or the execution logic of a control construct. For example, the GO TO statement can be used to transfer control to another statement within a program unit, and the EXIT statement can terminate execution of a DO construct.

This section describes the operations performed by the following flow control statements:

- CONTINUE statement
- CYCLE statement
- EXIT statement
- Assigned GO TO statement

- Computed `GO TO` statement
- Unconditional `GO TO` statement
- Arithmetic `IF` statement
- Logical `IF` statement
- `PAUSE` statement
- `STOP` statement

For additional information about these statements, see Chapter 10, Intel Fortran Statements.

CONTINUE Statement

The `CONTINUE` statement has no effect on program execution. It is generally used to mark a place for a statement label, especially when it occurs as the terminal statement of a FORTRAN 77-style `DO` loop.

Syntax

```
CONTINUE
```

Execution Logic

No action occurs.

Example

```
! find the 50th triangular number
  triangular_num = 0
  DO 10 i = 1, 50
    triangular_num = triangular_num + i
10 CONTINUE
  PRINT *, triangular_num
```

CYCLE Statement

The `CYCLE` statement interrupts execution of the current iteration of a `DO` loop.

Syntax

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

Execution Logic

1. The current iteration of the enclosing `DO` loop terminates. Any statements following the `CYCLE` statement do not execute.
2. If do-construct-name is specified, the iteration count for the named `DO` loop decrements. If do-construct-name is not specified, the iteration count for the immediately enclosing `DO` loop decrements.
3. If the iteration count is nonzero, execution resumes at the start of the statement block in the named (or enclosing) `DO` loop. If it is zero, the relevant `DO` loop becomes inactive.

Example

```
LOGICAL :: even
INTEGER :: number
loop: DO i = 1, 10
  PRINT *, "Enter an integer: "
  READ *,number
  IF (number == 0) THEN
   PRINT *, "Must be nonzero."
   CYCLE loop
  END IF
  even = (MOD(number, 2) == 0)
  IF (even) THEN
   PRINT *, "Even"
  ELSE
   PRINT *, "Odd"
  END IF
END DO loop
```

EXIT Statement

The `EXIT` statement terminates a `DO` loop. If it specifies the name of a `DO` loop within a nest of `DO` loops, the `EXIT` statement terminates all loops by which it is enclosed, up to and including the named `DO` loop.

Syntax

EXIT [*do-construct-name*]

Execution Logic

If the *do-construct-name* is specified, execution terminates for all DO loops that are within range, up to and including the DO loop with that name. If no name is specified, execution terminates for the immediately enclosing DO loop.

Example

```
DO
 PRINT *, "Enter a nonzero integer: "
 READ *, number
 IF (number == 0) THEN
  PRINT *, "Bye"
  EXIT
 END IF
 even_odd = MOD(number, 2)
 IF (even_odd == 0) THEN
  PRINT *, "Even"
 ELSE
  PRINT *, "Odd"
 END IF
END DO
```

Assigned GO TO Statement

The assigned GO TO statement transfers control to the statement whose statement label was assigned to an integer variable by an ASSIGN statement.

Syntax

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

If *label-list* is present, then the label previously assigned to integer-variable must be in the list.

Execution Logic

Control transfers to the executable statement at *integer-variable*.

Example

```
INTEGER int_label
  ...
  ASSIGN 20 TO int_label
  ...
GOTO int_label
...
20 ...
```

Computed GO TO Statement

The computed `GO TO` statement transfers control to one of several labeled statements, as determined by the value of an arithmetic expression.

Syntax

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

Execution Logic

1. integer-expression is evaluated.
2. The resulting integer value (the index) specifies the ordinal position of the label that is selected from label-list.
3. Control transfers to the executable statement with the selected label. If the value of the index is less than 1 or greater than the number of labels in label-list, the computed `GO TO` statement has no effect, and control passes to the next executable statement in the program.

Example

```
 DO
  PRINT *, "Enter a number 1-3: "
  READ *, k
  GO TO (20, 30, 40) k
  PRINT *, "Number out of range."
```

```
  EXIT
20 i = 20
  GO TO 100
30 i = 30
  GO TO 100
40 i = 40
100 print *, i
 END DO
```

Unconditional GO TO Statement

The unconditional `GO TO` statement transfers control to the statement with the specified label.

Syntax

`GO TO` *`label`*

Execution Logic

Control transfers to the statement at *`label`*.

Example

Older, "dusty-deck" Fortran programs often combine the `GO TO` statement with the logical `IF` statement to form a kind of leap-frog logic, as in the following:

```
  IF ( num1 /= num2) GO TO 10
  PRINT *, "num1 and num2 are equal."
  GO TO 30
10 IF ( num1 > num2 ) GO TO 20
  PRINT *, "num1 is smaller than num2."
  GO TO 30
20 PRINT *, "num1 is greater than num2."
30 CONTINUE
```

Arithmetic IF Statement

The arithmetic `IF` transfers control to one of three labeled statements, as determined by the value of an arithmetic expression.

Syntax

`IF` (*`arithmetic-expression`*) *`label1`*, *`label2`*, *`label3`*

Execution Logic

1. arithmetic-expression is evaluated.
2. If the resulting value is negative, control transfers to the statement at *`label1`*.
3. If the resulting value is 0, control transfers to the statement at *`label2`*.
4. If the resulting value is positive, control transfers to the statement at *`label3`*.

Example

Note that, as in this example, two or more labels in the label list can be the same.

```
i = MOD(total, 3) + 1
IF ( i ) 10, 20, 10
```

Logical IF Statement

The logical `IF` statement executes a single statement, conditional upon the value of a logical expression. The statement it executes must not be any of the following:

- a statement used to begin a construct
- an `END` statement
- an `IF` statement

Syntax

`IF` (logical-expression) executable-statement

Execution logic

1. logical-expression is evaluated.
2. If it evaluates to true, executable-statement executes.
3. The normal flow of execution resumes with the first executable statement following the `IF` statement. (If executable-statement is an unconditional `GO TO` statement, control resumes with the statement specified by the `GO TO` statement.)

Example

```
LOGICAL :: finished
.
.
.
IF ( finished ) PRINT *, "Done."
```

PAUSE Statement

The PAUSE statement temporarily suspends program execution until the user or the system resumes execution. The PAUSE statement is an obsolescent feature in Fortran 90 that has been deleted from the standard language in Fortran 95. However, Intel Fortran fully supports the PAUSE statement.

Syntax

PAUSE [*pause-code*]

where pause-code is one of the following optional messages:

- a scalar character constant of type default character
- a string of up to six digits; leading zeros ignored. (Fortran 95 and FORTRAN 77 standards limit the number of digits to five.)

Execution Logic

1. If you specify a [*pause-code*] message, the PAUSE statement displays the message specified and then the default prompt.

   or

   If you do not specify a [*pause-code*] message, the following prompt is then displayed:

   on Windows NT and Windows 95 Systems:

   ```
   Fortran Pause - Enter <CR> to continue
   ```

   where <CR> is the carriage control character. The program looks for input from stdin (typically your terminal keyboard). If you enter a blank line, execution resumes at the next executable statement. Anything else is treated as a DOS command, and executed by a system()call. The program loops, letting you execute multiple DOS commands until a you enter a blank line. The program resumes at the next executable statement.

If the standard input device is other than a your keyborad terminal, the message is:

```
To resume execution, execute the following a command:
kill -15 pid
```

pid is the unique process identification number of the suspended program. You can issue the kill command at any terminal that you are logged into.

Example

```
PAUSE 999
```

STOP Statement

The STOP statement terminates program execution.

Syntax

```
STOP [ stop-code ]
```

where stop-code is a character constant, a named constant, or a list of up to 5 digits.

Execution logic

Program terminates execution. If stop-code is specified, the following is written to standard output:

STOP *stop-code*

Example

```
STOP "Program has stopped executing."
```

Program Units and Procedures

7

This chapter describes the internal structure of each type of program unit, how it is used, and how information is communicated between program units and shared by them. All Fortran 95 statements are described in detail in this chapter.

Overview

This overview summarizes the main features of program units, procedures, scope, and association.

Program Units

A program unit is one of the following:

- Main program unit
- External function subprogram unit
- External subroutine subprogram unit
- Block data program unit
- Module program unit

A complete executable program contains one main program unit and zero or more other program units, where each of these can be compiled separately.

Program Unit Concepts

A program unit corresponds to the following characteristics:

- The main program, subroutine subprogram, and function subprogram are all executable. The nonexecutable program units are block data units and modules, which provide only definitions used by other program units.
- 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; it is optional in a main program unit. An `END` statement marks the end of a program unit.
- 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.
- A module program unit contains data declarations, user-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.
- Block data program units are used only to specify initial values for variables in named common blocks. With the provision of modules in Fortran 95, block data program units are no longer needed for new programs because modules can provide global data initializations.
- Main programs, external subprograms, and module subprograms may contain internal subprograms.
- All program units, except block data, may contain procedure interface blocks.

Procedures

Procedures can be defined by the following characteristics:

- A procedure is either a function or a subroutine. It encapsulates an arbitrary sequence of computations that may be invoked directly during program execution.
- A procedure is defined by a subprogram— that is, a subprogram defines (or is an implementation of) a procedure. A procedure can also be implemented by means other than the Fortran Language.

- If a subprogram contains one or more ENTRY statements, it defines a procedure for each ENTRY statement, as well as a procedure for the SUBROUTINE or FUNCTION statement.

Scope and Association

Two further concepts are required for a full understanding of program structure: *scope* and *association*.

Scope

All defined Fortran entities have a *scope* within which their properties are known. For example, a label used within a subprogram cannot be referenced directly from outside the subprogram; the subprogram is the scoping unit of the label. A variable declared within a subprogram has a scope that is the subprogram. A common block name can be used in any program unit and it refers to the same entity — that is, the name has global scope. At the other extreme, the index variable used within an implied DO list in a DATA statement or array constructor, for example, has a scope consisting only of the implied DO list construct itself.

Association

Entities may be associated by *host*, *storage*, *use*, *pointer* or *argument association*. One scoping unit can encapsulate others, and an entity declared in an outer unit may also, by default, be known within the contained subprogram. This is an example of host association. Examples of storage association are: use of an EQUIVALENCE statement (sharing data within a subprogram), use of a COMMON statement (sharing data across program units). The USE statement provides access to entities defined in a module to other program units by use association. A pointer and its target have pointer association. The actual and dummy arguments of a subprogram have argument association when the subprogram is invoked.

Example

Different kinds of scope and association are illustrated in the following:

```
SUBROUTINE get(i, j)
  INTEGER i,j
```

```
  COMMON /buffer/ x, y
  ! i,j,x, and y are local names.
  ! get and buffer are global.
   •
   •
   •
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
     ! Host association of stack_type.
   •
   •
   •
   END SUBROUTINE create
   •
   •
   •
END MODULE stack_database
PROGRAM main
! main is global
  USE stack_database
  INTEGER a, b
  TYPE (stack_type) :: main_stack
  ! Use association of stack_type.
```

```
  COMMON /buffer/ t(2)
  ! t is local but buffer is global; thus t is
  ! storage associated with x and y.
    •
    •
    •
  CALL get(a,b)
  CALL create(main_stack)
  ! Use association of create.
    •
    •
    •
END
```

In this example, the names `buffer`, `get`, `stack_database`, `create`, and `main` have the scope of the entire executable program, and are called global names. All of the program units `main`, `get`, `create` and `stack_database` are “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 derived-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`.

Lines of communication or association can be 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. The local variables `x` and `y` in `get` communicate with the local variable `t` in `main` by storage association.

Procedures

Fortran 95 procedures are implemented either as function subprograms or as subroutine subprograms. A function subprogram returns a value, the function-result, for use within an expression evaluation.

Procedure Categories

There are several categories of procedures described in the following sections.

Intrinsic Procedures

Intrinsic procedures are those available for use without any declaration or definition. They are described in detail in the *Intel® Fortran Compiler User's Guide*.

Intrinsic procedures provide a way to incorporate into Standard Fortran 95 the most common computations important to scientific and engineering applications. Standard Fortran 95 has 110 intrinsic functions and 5 intrinsic subroutines. Table 12 lists the different categories of intrinsic functions and gives the total number of intrinsic functions for each category.

Table 7-1 Categories of intrinsic functions

Category	Total
Conversion intrinsics	16
Array intrinsics	17
Inquiry and model intrinsics	28
Numeric computation	26
Character computation	12
Bit computation	9

Some intrinsic procedures are known as elemental—that is, they can take scalar arguments to produce a scalar result, and they can accept conformable arrays as arguments, in which case they operate on each array element separately and return an array as a result.

Several different intrinsic procedures can be called using the same name when the actual arguments are of different kinds, types and ranks. For example, when using `SQRT(`*x*`)`, *x* can be of type real or complex, it can be of any defined kind, and it can be an array of any of these types, of any rank. The name used is a generic name for a set of procedures, each of which accepts an argument of a fixed kind, type, and rank.

External Procedures

An external procedure is a separately compilable program unit whose name and any additional entry points have global scope.

Module Procedures

A module procedure can appear only within a module. Its name can be made available outside the scope of its host only by use association. It is not otherwise accessible outside of its host.

Internal Procedures

An internal procedure can appear only within a main program, an external subprogram, or a module subprogram. It cannot have additional entry points, and it is not accessible outside its host. It appears between a `CONTAINS` and `END` statement of its host.

Referencing Procedures

The following sections describe how to reference a procedure subprogram:

Subroutine Subprogram

A subroutine subprogram is referenced by using the `CALL` statement specifying the subroutine name, one of its entry point names, or when the subroutine is implementing an assignment operation.

The syntax of the `CALL` statement is:

```
CALL subroutine-name &
   [([actual-argument-spec-list])]
actual-argument-spec
   is [keyword =] actual-argument
keyword
```

is *dummy-argument*

actual-argument

is one of the following:

- expression
- variable
- procedure-namealternate-return

alternate-return

is one of the following:

- *label
- &label

label is permitted in Intel Fortran in fixed source form only.

Alternate returns are arguments that permit control to branch to a particular label following the call. *label* is the statement label of an executable statement in the same scoping unit as the CALL statement.

Function Subprogram

A function subprogram is referenced by its name, by one of its entry point names or by the operator defined by the function.

The syntax of a function reference is:

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

where *actual-argument-spec* is as above, except that an alternate return cannot be included.

On invocation, the calling program unit can identify actual arguments that are then associated with dummy arguments in the procedure definition. This is also applicable to subroutine subprograms.

Interfaces

The interface of a procedure is the information required to compile a call to the procedure. This information includes the characteristics of the dummy arguments and of the result for a function procedure. It is explicit if all of these characteristics are defined within the scope of the reference. If they are not, the compiler may be able to make sufficient assumptions about them and the interface is then implicit. Under many circumstances an explicit interface is mandatory. Internal procedures and module procedures always have an explicit interface.

Explicit interfaces can be specified by means of an INTERFACE block. An INTERFACE block is also used to define the extended use of the standard operators, to define new operators and to extend the definition of the assignment operator. Additionally, it provides the capability of using a generic name to reference any one of a set of procedures.

Generic Referencing

The user can define a generic name and code several procedures to cater for the required different combinations of argument characteristics. Then, by including the specifications of these routines within an interface definition, use just the generic name for referencing any of the procedures (the choice of procedure being determined by matching actual and dummy arguments). The technique can also be used to extend the selection of argument types.

Built-in Functions

Intel Fortran provides two functions that are extension to the language: %VAL and %REF. These extensions can be used to communicate with procedures written in programming languages other than Fortran 95 that have different argument passing conventions. These functions specify how an actual argument is to be passed in a procedure reference.

- %VAL(a) specifies that the value of the actual argument a is to be passed to the called procedure. The argument a can be a constant, variable, array element, or derived-type component. %VAL approximates the default argument passing mechanism of the C programming language, and it is to pass a Fortran object to a procedure written in C where %VAL is typically used.
- %REF(a) specifies that the actual argument a is to be passed as a reference to its value. This is how Intel Fortran normally passes arguments except those of type character; for each character argument, Intel Fortran normally passes both a reference to the argument and its length, with the length being appended to the end of the actual argument list. Passing a character argument using %REF disables the passing of the character length argument.

These two routines may only be used in either an interface block, or in the actual CALL statement or function reference.

For information about %VAL and %REF, see the description of the CALL statement in Chapter 10, Intel Fortran Statements.

Example

This first example uses %VAL and %REF in an interface block:

```
PROGRAM foobar
  INTERFACE
    SUBROUTINE fred(%VAL(X))
     INTEGER :: x
    END SUBROUTINE fred

    FUNCTION foo (%REF(ip))
     INTEGER :: ip, foo
    END FUNCTION foo
  END INTERFACE
  ...
CALL fred(i) ! The value of i is passed to fred
j = foo(i)  ! i passed to foo by reference,
       ! foo receives a reference to
       ! the value of i.
END PROGRAM
```

The next example employs %REF and %VAL in the actual procedure references:

```
  PROGRAM foobar
   INTEGER :: foo
   EXTERNAL foo, fred
   ...
   CALL fred (%VAL(i))
   j = foo(%REF(i))
  END PROGRAM
```

Procedure interfaces and interface blocks are described later in this chapter.

Procedure Definition

Fortran procedures mostly consist of functions and subroutines defined in the following sections.

Functions and Subroutines

Functions and subroutines are defined in Fortran 95 by means of subprograms.

A subroutine subprogram has the following form:

subroutine-statement
 [specification-part]
 [execution-part]
 [internal-subprogram-part]
end-subroutine-statement

A function subprogram has the following form:

function-statement
 [specification-part]
 [execution-part]
 [internal-subprogram-part]
end-function-statement

internal-subprogram-part is:

 CONTAINS
 [*internal-subprogram*]...

internal-subprogram

 is a subprogram without any *internal-subprogram-part*.

end-subroutine-statement

 is

 END [SUBROUTINE [*subroutine-name*]]

end-function-statement

 is

 END [FUNCTION [*function-name*]]

Statements Introducing Procedures

The following sections describe specific statements with which each procedure is introduced in the code.

Subroutine Statement

The SUBROUTINE statement introduces a subroutine subprogram. The syntax for *subroutine-statement* is:

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

Function Statement

The FUNCTION statement introduces a function subprogram. The syntax for *function-statement* is:

[*prefix*] FUNCTION *function-name* &

([*dummy-argument-list*]) [RESULT (*result-name*)]

prefix

is one of:

- *type-spec* [RECURSIVE]
- RECURSIVE [*type-spec*]
- PURE [*type-spec*]
- ELEMENTAL [*type-spec*]

The *type-spec* may appear before or after the attribute.

Dummy arguments are discussed in "Subprogram Arguments" on page 18; actual arguments are described in "Referencing Procedures" on page 7.

The result-name must be different from the function-name, and must be given if the function is recursive. If there is a result clause, the result-name must be used as the result variable, otherwise the function name must be used.

Entry Statement

The ENTRY statement defines a procedure entry. Its syntax is as follows:

ENTRY *entry-name* ([*dummy-argument-list*]) &

[RESULT (*result-name*)]

The RESULT clause may appear only if the ENTRY statement is in a function subprogram.

ENTRY statements can appear only within the execution part of a subprogram; they provide additional names by which the subprogram can be invoked. Execution will commence at the statement immediately following the referenced ENTRY statement, but the ENTRY statement does not interrupt the sequencing of execution across it. ENTRY statements may not appear within an internal subprogram.

When used in a function subprogram, all ENTRY statements should return results of the same type, kind, and shape; otherwise they should all return results that are scalars without the POINTER attribute and which are of intrinsic numeric or logical type.

Examples of the ENTRY statement are given in Chapter 10, Intel Fortran Statements.

Internal Procedures

Internal procedures are defined by internal subprograms in the internal subprogram part of a main, external, or module program unit. Internal subprograms can be recursive. The following restrictions apply to internal subprograms:

- They cannot have ENTRY statements.
- They cannot be passed as arguments.
- Their names are local to the host.
- The interface of an internal procedure is explicit in the host.
- Declarations in the host are inherited by the internal subprogram, but can be overridden.
- They cannot be the host of another internal procedure.

Example of an internal function:

```
SUBROUTINE printav
! Start of external subprogram.
 REAL, DIMENSION(3) :: x
 ! Specification part.
 x=(/2.0,5.0,7.0/)
 PRINT *,av(x)
 ! Reference to function av.
```

```
 CONTAINS
  REAL FUNCTION av(a)
  ! Start of internal subprogram.
   REAL a(:)
   av=SUM(a)/SIZE(a)
   ! References to intrinsic functions.
  END FUNCTION av
END SUBROUTINE printav
```

RECURSIVE Procedures

An internal or external procedure that directly or indirectly invokes itself is recursive. Such a procedure must have the word RECURSIVE added to the FUNCTION or SUBROUTINE statement.

If both RECURSIVE and a result clause are specified in the FUNCTION statement, then the interface of the function being defined is explicit within the subprogram.

Example of a recursive function:

```
RECURSIVE FUNCTION factorial (n) RESULT(r)
 INTEGER :: n, r
 IF (n.ne.0) THEN
  r = n*factorial(n-1)
 ELSE
  r = 1
 ENDIF
END FUNCTION factorial
```

PURE Procedures

A PURE procedure is a routine that does not change any value external to the procedure implicitly. For a FUNCTION, this means that the only value a pure function may change is its return value. For a SUBROUTINE, the routine may change only the values of those formal parameters that are declared INTENT(OUT). Variables in COMMON blocks or variables that are host-associated or use-associated are regarded as external to the PURE

procedure, and may not be modified. Any local variable that is storage-associated with an variable in a COMMON block or a variable that is host or use associated may not be modified.

Host association occurs when the declaration point of a variable is in an outer scope from the current procedure. USE association occurs when a variable is made available in the current routine via a USE statement.

Variables in COMMON blocks or variables that are host or use associated may not appear in the right-hand side of an assignment statement whose left-hand side has a pointer component at any level, within a PURE procedure.

All intrinsic functions and the intrinsic subroutine MVBITS are PURE procedures.

Local variables of PURE procedures must not have the SAVE attribute, and may not be used in a STATIC or VOLATILE statement.

This means also that you cannot give a local variable of a PURE procedure an initial value in the specification portion of the routine, since this automatically implies the SAVE attribute.

A PURE procedure may not contain a STOP, PRINT, OPEN, CLOSE, BACKSPACE, REWIND, ENDFILE, or INQUIRE statement. WRITE and READ statements may appear only if they reference internal files as the logical unit.

Any routine CONTAINed within a PURE procedure must also be PURE.

Example

```
PURE FUNCTION ADDIT(I)
 INTENT (IN) I
 ADDIT = REAL(I) + 2.0
END FUNCTION

PURE SUBROUTINE SUBIT(RESULT,AIN1,AIN2)
 INTENT (OUT) RESULT
 INTENT (IN) AIN1,AIN2
 RESULT = AIN1 - AIN2
END SUBROUTINE
```

In the following cases the use of a PURE procedure is required, if any procedure is used at all:

- a function referenced in a FORALL statement
- a function referenced in a specification statement
- a procedure that is passed as an actual argument to a PURE procedure
- a procedure referenced in the body of a PURE procedure, including defined operators or defined assignment

In these contexts, with the exception of intrinsic functions, you must define an explicit INTERFACE for the referenced procedure, and the INTERFACE must contain the PURE attribute.

ELEMENTAL Procedures

ELEMENTAL procedures allow you to specify routines that have scalar formal parameters, but whose actual parameters are arrays of any rank, so long as the actual parameters are conformable. A formal parameter may be a scalar if it is used as a dimension specifier in declaration statement. An ELEMENTAL procedure is implicitly a PURE procedure. The keywords ELEMENTAL and PURE may appear on the same routine declaration, but in this case PURE is redundant.

The result of executing an ELEMENTAL procedure is the same as if the procedure were applied to each element of the array(s) in the actual parameter(s) in any order, including simultaneously. ELEMENTAL functions must return a scalar result.

It should be noted that some side effects are not recognized by standard Fortran, such as IEEE floating point exception bits. It is the programmer's responsibility to make sure these side effects either do not occur, or have acceptable results.

The following rules apply to ELEMENTAL procedures:

- Formal parameters must be INTENT(IN), and cannot be a procedure parameter, or a pointer.
- The INTENT of each formal parameter must be specified.
- Local variables have the same restrictions as in a PURE procedure (i.e. no SAVE attribute, cannot be used in a STATIC or VOLATILE statement. See "PURE Procedures.").

- Since an ELEMENTAL procedure is also PURE, an ELEMENTAL procedure may not contain a STOP, PRINT, OPEN, CLOSE, BACKSPACE, REWIND, ENDFILE, or INQUIRE statement. WRITE and READ statements may appear only if they reference internal files as the logical unit.
- An ELEMENTAL procedure cannot be RECURSIVE.
- A formal parameter of an ELEMENTAL procedure must not have a POINTER type.
- The result of an ELEMENTAL function must not have a POINTER type.
- You cannot use alternate returns with an ELEMENTAL subroutine.

Statement Functions

If an evaluation of a function with a scalar value can be expressed in just one Fortran assignment statement, such a definition can be included in the specification part of a main program unit or subprogram. This definition is known as a statement function, and is local to the scope in which it is defined. The syntax is:

function-name (*dummy-argument-list*) =
scalar-expression

All dummy arguments must be scalars. All entities used in the expression must have been declared earlier in the specification part. A statement function can reference another statement function that has already been declared. The name cannot be passed as a procedure-name argument.

Example

```
vol(r,h) = 3.14*h*r**2
! Definition, within the specification part of
! the subprogram.
  ...
   total = n * vol(0.5*d,x)
! Reference to the statement function, from
! within the same program unit.
  ...
```

Returning to the Calling Unit

When the END statement of the subprogram is encountered, control will be returned to the calling program unit. The RETURN statement can be used to the same effect at any point within the subprogram.

The syntax of the RETURN statement is:

RETURN [*scalar-integer-expression*]

When alternate returns have been declared, the value of the scalar integer expression determines which argument corresponds to the requested return point. The dummy arguments corresponding to the alternate returns must each be declared as an asterisk (*). If the value of the expression is n, then the return is to the labeled statement referred to by the actual argument corresponding to the nth dummy argument declared as an asterisk.

Subprogram Arguments

Actual arguments appear in a procedure reference and specify the actual entities to be used by the procedure during its execution. Dummy arguments are specified when the procedure is defined and are the name by which the actual arguments are known within a procedure. When a procedure is referenced during program execution, the actual arguments become associated with the dummy arguments through argument association.

A dummy argument is one of:

- *name*
- * (subroutine only)

A subprogram actual argument is one of:

- *expression*
- *procedure-name*
- *alternate return*

Arguments allow a calling program unit and a called subprogram to communicate with each other. The calling unit provides a list of actual arguments and the called subprogram will have been declared with a list of dummy arguments.

Argument Correspondence

If no keyword= component of an actual-argument-spec is included in the reference, then each actual argument is assumed to correspond with the dummy argument in the equivalent position in the dummy argument list. The actual arguments may appear in any order if the keyword option is used for all arguments. Actual arguments without the keyword option may be followed by an argument with the keyword option. However, an argument with the keyword option must only be followed by other arguments with the keyword option.

Dummy arguments can be declared with the `OPTIONAL` attribute. If they appear at the end of the dummy argument list, the reference can omit any trailing arguments that are not required. Otherwise, keywords must be provided to maintain an identifiable correspondence.

Example of Argument Correspondence

The intrinsic function `SUM` (has three arguments: `array`, `dim`, and `mask`, in that order, and dim and mask are optional arguments. The following are valid references:

```
SUM(a,2)
SUM(a,MASK=a.gt.0)
SUM(DIM=2,ARRAY=a)
```

The following is an invalid reference—the mask keyword should have been specified:

```
SUM(a,DIM=2,a.gt.0)   ! Invalid
```

Argument Association

Dummy and actual arguments must agree in kind, type, and usually in rank—that is, both scalars, or both arrays of the same dimensionality. An actual argument that is an expression or a reference to a function procedure must match the type and kind of the dummy argument.

Scalar Dummy Argument

If the dummy argument is a scalar, then the corresponding actual argument must be a scalar, or scalar expression, of the same kind and type.

If the dummy argument is a character variable and has assumed length then it will inherit the length of the actual argument. Otherwise the length of the actual argument must be at least that of the dummy argument, and only the characters within the range of the dummy argument can be accessed by the subprogram. The lengths may differ only for default character types.

Array Dummy Argument

The different sorts of Fortran 95 arrays are described in Chapter 4, "Arrays."

If the dummy argument is an assumed-shape array, then the corresponding actual argument must match in kind, type, and rank; the dummy argument assumes its shape (the number of elements in each dimension) from the actual argument, resulting in an element-by-corresponding-element association between the actual and dummy arguments.

If the dummy argument has explicit shape or assumed size, the kind and type of the actual argument must match but the rank need not, as the elements are matched by sequence association. That is, the actual and dummy arguments are each considered to be a linear sequence of elements in storage without regard to rank or shape, and corresponding elements in each sequence are associated with each other.

A consequence of this sequence association is that the overall size of the actual argument must be at least that of the dummy argument, and only elements within the overall size of the dummy argument can be accessed by this subprogram.

Example

The following example illustrates sequence association.

Assuming that an actual array argument is declared thus:

```
REAL a(0:3,0:2)
```

And that the corresponding dummy array argument is declared thus:

```
REAL d(2,3,2)
```

Then the following correspondence between elements of the actual and dummy argument is achieved:

```
Dummy    <=> Actual
```

```
d(1,1,1) <=> a(0,0)d(2,1,1) <=> a(1,0)
d(1,2,1) <=> a(2,0)
         ...
d(2,3,2) <=> a(3,2)
```

When an actual argument and associated dummy argument are default character arrays, they may be of unequal character length. If this is the case, then the first character of the dummy and actual arguments will be matched, and the successive characters, rather than array elements, will be matched.

Example

The following example illustrates character sequence association.

Assuming that an actual argument array is declared thus:

```
CHARACTER*2 a(3,4)
```

and that the corresponding dummy array argument is declared thus:

```
CHARACTER*4 d(2,3)
```

then the following correspondence between elements of the actual and dummy argument is achieved:

```
Dummy   <=>  Actual
d(1,1)  <=>  a(1,1)//a(2,1)
d(2,1)  <=>  a(3,1)//a(1,2)
         ...
d(2,3)  <=>  a(2,4)//a(3,4)
```

An actual argument may be an array section, but passing an array section to a nonassumed shape dummy argument may cause a copy of the array section to be generated and is likely to result in a degradation in performance.

Derived-type Dummy Argument

The corresponding dummy and actual arguments of derived types are of the same derived type if the structures refer to the same type definition. Alternatively, they are of the same type if all the following are true:

- They refer to different type definitions with the same name.
- They both have the `SEQUENCE` statement in their definition.

- The components have the same names and types and are in the same order.
- None of the components is of a private type or is of a type that has private access.

Pointer Dummy Argument

If the dummy argument has the POINTER attribute, the actual argument must also have the POINTER attribute, and match in kind, type, and rank; the dummy argument in the procedure then 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.

Procedure Dummy Argument

If a dummy argument of a procedure is used as a procedure name within the procedure, then the actual argument must be the name of an appropriate subprogram, and its name must have been declared as EXTERNAL in the calling unit or defined in an interface block. Internal procedures, statement functions and generic names may *not* be passed as actual arguments.

If the actual argument is an intrinsic procedure, then the appropriate specific name must be used in the reference, and must be declared as INTRINSIC in the calling unit.

Example

```
DOUBLE PRECISION dsin,x,y,fun
INTRINSIC dsin
  ...
  y=fun(dsin,x)
  ...
DOUBLE PRECISION FUNCTION fun(proc,y)
DOUBLE PRECISION y, proc
   ...
  fun=proc(y)
   ...
END
```

Duplicated Association

If a subroutine call or function reference would cause a data object to be associated with two or more dummy arguments, then that data object must not be redefined within the subroutine or function. For example, in the following:

```
PROGRAM p
  CALL s (a,a)
CONTAINS
  SUBROUTINE s (c,d)
   c = 22.01    ! invalid definition of
            ! one of the dummy
            ! arguments associated
            ! with data object a
  ...
 END SUBROUTINE
END PROGRAM
```

both dummy arguments, c and d, are associated with the actual argument a. The definition of a, through the assignment to the dummy argument c, is invalid. The above rule is extended to when the actual arguments are overlapping sections of the same array.

Similarly, if a data object is available to a procedure through both argument association and either use, host, or storage association, then the data object must be defined and referenced only through the dummy argument.

In the following code, the data object a is available to the subroutine as a consequence of argument association and host association. The reference to a directly in the subroutine is illegal.

```
PROGRAM p
  CALL s (a,b)
CONTAINS
  SUBROUTINE s (c,d)
   c = 22.01    ! valid definition of a
            ! through the dummy
            ! argument
   d = 3.0*a    ! reference to a directly
```

```
                ! is illegal
  ...
 END SUBROUTINE
END PROGRAM
```

INTENT Attribute

To enable additional checking to be performed on argument matching and to avoid possible unwanted side effects, an `INTENT` attribute can be declared for each dummy argument, which may be specified as `INTENT(IN)`, `INTENT(OUT)` or `INTENT(INOUT)`.

The values that may be specified for the `INTENT` attribute have the following significance:

- `IN` is used if the argument is not to be modified within the subprogram.
- `OUT` implies that the actual argument must not be used within the subprogram before it is assigned a value.
- `INOUT` (the form `IN OUT` is also permitted) implies that the actual argument must be defined on entry and is definable within the subprogram.

Interfaces

The interface to a procedure (referred to as the procedure interface) is that information, about the procedure, which is pertinent when that procedure is invoked. This information includes:

- The name of the procedure.
- The properties (type, kind, and attributes) of the result, if the procedure is a function.
- The names, types, kinds, attributes, and order of the dummy arguments of the procedure.

The procedure interface is said to be explicit if the above information is available to a program unit containing a reference to the procedure; when the above information is not known, the procedure interface is implicit. In FORTRAN 77 all procedure interfaces are implicit, giving no way to ensure that the actual arguments supplied in a procedure reference match the dummy arguments within the procedure itself. In Fortran 95 procedure interfaces can be either implicit or explicit.

A number of new Fortran 95 features, as listed below, require that the procedures involved have explicit procedure interfaces available within the scoping units invoking them. An explicit procedure interface is required when:

- The procedure reference uses the keyword form of an actual argument.
- The procedure has `OPTIONAL` arguments.
- Any dummy argument is an assumed-shape array or pointer.
- The result of the procedure is an array or pointer.
- The procedure is a character function, the length of which is determined dynamically.
- The procedure reference is to a generic name.
- The procedure reference is a consequence of a user-defined operator function or operation.
- The procedure reference is a consequence of a user-defined assignment.

Even where an explicit procedure interface is not required, making a procedure interface explicit allows the compiler to check the validity of references to the procedure.

In Fortran 95, all procedure interfaces are implicit except for procedures which are:

- Intrinsic procedures—the interface of every intrinsic procedure is explicit.
- Internal procedures—the interface of an internal procedure is explicit within its host.
- Module procedures—the interface of a module procedure is explicit within a program unit using the module, and within the module itself.
- Recursive functions which specify a result clause—the interface of such a function is explicit within the function itself.
- External procedures whose interfaces have been made explicit by the provision of an interface block.

INTERFACE Block

When an external procedure (one which is outside the prevailing scope) or a dummy procedure is referenced, it is sometimes necessary for its interface to be made explicit. This is achieved by the provision of an interface block, which is accessible to the scoping unit containing the procedure reference. An interface block can also be used to:

- Define a generic procedure name, specify the set of procedures to which the generic name applies, and make explicit the interfaces of any external procedures contained within the set.
- Define a new operator or extend an already defined operator, specify the set of functions which implement the operator, and make the interfaces of any of these functions, which are external, explicit.
- Define new defined assignment operations, specify the set of subroutines to which to implement these operations, and make the interfaces of any of these subroutines, which are external, explicit.
- An explicit interface is described by an interface block, which appears in the specification part of the programming unit containing the procedure reference. An interface block may appear in any program unit, except a block data program unit.

The syntax for an `INTERFACE` *block* is:

`INTERFACE` [*generic-spec*]

[*interface-body*]...

[`MODULE PROCEDURE` *module-procedure-name-list*]

`END INTERFACE`

generic-spec

is one of:

- *generic-name*
- `OPERATOR (`*operator*`)ASSIGNMENT (=)`

generic-name

is the name of the generic procedure that is referenced in the subprogram containing the interface block

operator

is one of the Fortran 95 unary or binary intrinsic operators, or a user-defined unary or binary operator of the form:

.*letter*[*letter*]....

interface-body

is

function-statement
[*specification-part*]
end-function-statement

or

subroutine-statement
[*specification-part*]
end-subroutine-statement

The MODULE PROCEDURE statement is permitted in an interface block only if there is a *generic-spec* present.

In the following example, the procedure interface for the function av is made explicit by the inclusion of the interface block in the main program.

Example

```
REAL FUNCTION av(a)
! External function av with one assumed-shape
! dummy argument.
 REAL a(:)
 av = SUM(a)/SIZE(a)
END

PROGRAM main
REAL,DIMENSION(3) :: x
INTERFACE
 REAL FUNCTION av(a)
 REAL, INTENT(IN) :: a(:)
 END FUNCTION av
END INTERFACE
```

```
x=(/2.0,4.0.7.0/)
PRINT *, av(x)
END
```

INTERFACE TO Block

The INTERFACE TO is an Intel Fortran extension that serves the same purpose as the INTERFACE block with the following exceptions:

- The header is on the same line as the as the key phrase INTERFACE TO.
- The block contains the specifications for just one subroutine or function

The syntax for an INTERFACE TO block is:

```
INTERFACE TO [function-statement
|subroutine-statement]
   [formal-parameter-declarations]...
END
```

where,

function-statement is a function declaration statement.

subroutine-statement is a subrotuine declaration statement

formal-parameter-declarations is variable declaration statement.

The following is an example:

```
INTERFACE TO Integer*4 function CreateMutex
   [stdcall, alias: '_CreateMutexA@12']
   (security, owner, string)
   integer*4 security [value]
   logical*4 owner [value]
   integer*4 string [value]
END
```

In the preceding example, the procedure CreateMutex is being referenced. The ALIAS attribute is being used to change the name of the function. (See "Attributes" in Appendix A for a description of the ALIAS attribute. Also, three variables (security, owner, string) and their data types are being prototyped.

Generic Names and Procedures

The concept of generic names and procedures was introduced in FORTRAN 77 with the provision of generic intrinsic procedures. In Fortran 95 this concept is extended to allow user-defined generic procedures.

Two or more procedures are said to be generic if they can be referenced with the same name; the name by which the procedures can be referenced is the generic name.

A generic name is defined by an interface block containing a *generic-spec* and the specifications of the procedures that may be invoked by referencing the generic name. The procedure specifications in the interface block must be distinguishable from each other in one or more of the following ways:

- The number of dummy arguments differ.
- Dummy arguments, from the specific procedure specifications, that occupy the same position in the argument lists differ in type, kind, or rank.
- The name of a dummy argument differs from the names of the other dummy arguments in dummy argument lists of other procedure specifications, or all dummy arguments with the same name differ in either type, kind, or rank.

There may be more than one interface block with the same generic name, but the procedure specifications in all such interface blocks must be distinguishable by the above criteria.

When a generic name is referenced it must be possible to determine to which of the specific procedures the generic name refers; the generic reference must resolve to a unique specific procedure name. Selection of the specific procedure is based on the properties of the actual argument list, including:

- The number of actual arguments.
- The type, kind, and rank of each actual argument.
- The argument keyword, if supplied, of an actual argument.

The specific procedure whose dummy argument list matches the actual argument list is selected and invoked from the list of procedure specifications contained in the interface block that defines the generic

name. The dummy argument list of exactly one of the procedure specifications contained in the interface block must match the actual arguments in the reference of the generic name.

The MODULE PROCEDURE statement can be used to extend the list of procedure specifications that comprises the interface block, by naming procedures that are accessible to the program unit containing the interface block. In the MODULE PROCEDURE statement only the specific names of the procedures are given as their procedure interfaces are already explicit. The MODULE PROCEDURE statement may only appear in an interface block that has a generic specification, and the interface block must either be in the module containing the definitions of the named procedures or, in a program unit in which the procedures are accessible through use association.

Example

In the following, it is assumed that two subroutines have been coded for solving linear equations: rlineq for when the coefficients are real, and zlineq for when the coefficients are complex. A generic name, lineq, is declared as follows and then used for either reference.

```
INTERFACE lineq
  SUBROUTINE rlineq(ra,rb,rx)
   REAL,DIMENSION(:,:) :: ra
   REAL,DIMENSION(:) :: rb,rx
  END SUBROUTINE rlineq
  SUBROUTINE zlineq(za,zb,zx)
   COMPLEX,DIMENSION(:,:) :: za
   COMPLEX,DIMENSION(:) :: zb,zx
  END SUBROUTINE zlineq
END INTERFACE lineq
```

Defined Operators

The OPERATOR (*operator*) generic specification can be used to either define a new user-defined operator symbol, or to extend the behavior of an already defined or intrinsic operator.

When the OPERATOR (*operator*) generic specification is present in the INTERFACE statement, the procedure specifications that immediately follow must only describe function subprograms. The functions described

are those that are to be used to implement the operator for various type, kind, and rank combinations of operand. These functions must have only one or two mandatory arguments, which correspond to the operand(s) of a unary or binary operator. The functions return the result of an expression of the form:

```
operator operand
```

or

```
operand1 operator operand2
```

as appropriate. Each dummy argument of the functions described in the interface block must have the `INTENT(IN)` attribute. If `operator` is one of the Fortran 95 intrinsic operators, then each of the specified functions must take the same number of arguments as the intrinsic operator has operands, and the arguments must be distinguishable from those normally associated with the intrinsic operation.

Argument keywords must not be specified in a reference to a user-defined operator function when the operator syntax, rather than the name of the specific function, is used in an expression.

An interface block that defines or extends an operator is analogous to defining a generic procedure name, with the operator being the generic name. Similarly a reference to a user-defined operator must resolve to a unique specific function name. The selection of the function is accomplished by matching the number, type, kind, and rank of the operand(s) with the dummy argument lists of the function specifications contained in the interface block. As with generic names exactly one such specification must match the properties of the operands, and the function whose specification does match is selected and invoked.

See the examples in the next section.

Defined Assignment

The `ASSIGNMENT(=)` option allows you to specify one or more subroutines that extend the assignment operation. Each subroutine must have only two mandatory arguments; the first argument can have either the `INTENT(OUT)` or the `INTENT(INOUT)` attribute; the second argument

must have the INTENT(IN) attribute. The first argument corresponds to the variable on the left-hand side of the assignment statement, and the second to the expression on the right-hand side.

In a similar manner to generic names and defined operators, defined assignment must resolve to a unique specific subroutine. The subroutine whose dummy arguments match the left and right-hand sides of the assignment statement in all of kind, type, and rank is selected and invoked from the list of subroutine specifications contained in the defined assignment interface block.

Examples

The following example illustrates the definition of a 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
! Compute the eigenvalues of a.
b = .eigenvalues. a
```

The next example extends the * operator and assignment in order to work with entities of derived types.

```
! Extend the * operator.
INTERFACE OPERATOR (*)
  MODULE PROCEDURE polar_mul, interval_mul
END INTERFACE
! Extend assignment.
INTERFACE ASSIGNMENT (=)
  MODULE PROCEDURE assign_polar_to_complex
END INTERFACE
TYPE (polar) :: p1, p2
```

```
TYPE (interval) :: v1, v2, v
COMPLEX :: c
   •
   •
   •
! A defined operation and an intrinsic
! assignment.
v = v1*v2
! A defined operation and an defined assignment.
c = p1*p2
```

Modules

Modules contain the definitions of data objects, derived-types, procedures, and procedure interface blocks. These definitions may be used in other program units. A module does not contain executable code, except in the execution parts of module subprograms.

Typically, modules are used for:

- Definition and declaration of data types
- Definition and declaration of global data areas
- Definitions of operators
- Creation of subprogram libraries

The definitions within a module become available to another program unit if that program unit contains a USE statement nominating the module. These definitions are then said to be accessible by the other program unit through use association.

A USE statement may appear within a module program unit, but such a statement must not cause a module to reference itself either directly or indirectly.

The syntax of a module program unit is:

MODULE *module-name*

[*specification-part*]

[*module-subprogram-part*]

END [MODULE [*module-name*]]

module-subprogram-part

is

CONTAINS

module-subprogram[*module-subprogram*]...

module-subprogram

is one of:

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

A module subprogram can contain internal subprograms. It differs from an external subprogram in the following ways:

- A module subprogram name does not have global scope—it is known only within the module. However, it can be made accessible to other program units, however, through use association.
- The END statement of a module subprogram must contain the SUBROUTINE or FUNCTION keyword, as appropriate; for an external subprogram this is optional.

Points to note about module definition and use:

- Module entities that are accessible by use association are:
  - Declared variables
  - Named constants
  - Derived-type definitions
  - Procedure interfaces
  - Module procedures
  - Generic names
  - Namelist groups
- The procedure interface of a module subprogram is automatically made explicit in the program unit using the module—no interface block needs to be created.
- A module subprogram can be passed as an actual argument.
- The specification part of a module must not contain statement function definitions, automatic objects, or FORMAT statements.

- The SAVE attribute can be specified when declaring an entity within a module or, alternatively, the entity may appear in a SAVE statement within the module. This will preserve the entity's value even when there are no active program units using the module.
- Specifying the SAVE attribute within a module is unnecessary in Intel Fortran, as entities declared within a module retain their value(s) by default.
- Each entity declared in the module specification part, and each of the module subprogram names, has either the PUBLIC or PRIVATE attribute. By default all of the declared entities have the PUBLIC attribute and thereby become accessible to other program units accessing the module by use association. The PRIVATE attribute and statement can be specified to inhibit access.

The PUBLIC and PRIVATE attributes and statements are described further in Chapter 10, Intel Fortran Statements.

Example

The following schematic example shows a possible structure for an application using modules.

MODULE datatypes

[*derived-type definitions*] ...

END MODULE datatypes

```
MODULE global
 USE datatypes
 ! Gives access to module datatypes.
 [global data area definitions] ...
END MODULE global

MODULE operators
 USE datatypes
 ! Gives access to module datatypes.
 [generic interface definitions] ...
 [code for operator,assignment definitions]
 ...
```

```
END MODULE operators
MODULE library
 USE operators
 ! Gives access to datatypes and operators.
 CONTAINS
   •
   •   ! Module subprograms.
   •
END MODULE library

PROGRAM main
 USE global
 ! Gives access to data areas. USE library
 ! Gives access to subprogram library,
 ! generic interfaces and operator
 ! definitions.
   •
   •
   •
END
```

Use Statement

USE statements appear at the head of the specification part of a program unit that requires access to information from modules. Such shared information has use association.

The syntax of the USE statement is:

- USE *module-name* [, *rename-list*]

or

- USE *module-name*, ONLY : [*access-list*]

rename-list

is a comma separated list of *rename*

rename

is

local-name => *module-entity-name*

access -list

is a comma separated list of *access*

access

is one of:

- [*local-name =>*] *module-entity-name*
- OPERATOR (*operator*)
- ASSIGNMENT (=)

Notes

- The form USE *module-name*, without the ONLY option, provides access to all PUBLIC entities within *module-name* available.
- The ONLY option restricts the information that is accessed through use association to a listed subset of public items in the named module. Each item can be renamed if necessary.
- A *rename-list* may be added to avoid name clashes.
- When more than one USE statement for the same module is present in a scoping unit, then:
- If one of the USE statements is without the ONLY qualifier, then all of the PUBLIC entities from the module are available and all of the renames from the rename-lists and access-lists are interpreted as a single concatenated rename-list.
- Otherwise, all of the USE statements have the ONLY qualifier and the *access-lists* from these ONLY qualifiers are interpreted as a single concatenated access-list.

Example 1

In this first example, the module `linearsolver` contains two module procedures `rlineq` and `zlineq`; they are given the generic name `lineq`, which is renamed to `lq` by the program unit using `linearsolver`. `lq` is then invoked twice.

```
MODULE linearsolver
INTERFACE lineq
```

```
  MODULE PROCEDURE rlineq, zlineq
END INTERFACE
CONTAINS
  SUBROUTINE rlineq(ra,rb,rx)
  REAL,DIMENSION(:,:) :: ra
  REAL,DIMENSION(:) :: rb, rx

              ! Code for rlineq.

  END SUBROUTINE rlineq
  SUBROUTINE zlineq(za,zb,zx)
  COMPLEX,DIMENSION(:,:) :: za
  COMPLEX,DIMENSION(:) :: zb, zx
   •
   •     ! Code for zlineq.
   •
  END SUBROUTINE zlineq
END MODULE linearsolver

PROGRAM main
  USE linearsolver, lq => lineq
  REAL ra(4,4),rb(4),rx(4)
  COMPLEX za(5,5),zb(5),zx(5)
   •
   •
   •
  CALL lq(ra,rb,rx)    ! Invokes rlineq.
   •
   •
   •
  CALL lq(za,zb,zx)    ! Invokes zlineq.
   •
   •
   •
END PROGRAM main
```

Example 2

The next extended example entails the use of two modules, `precision` and `linear_equation_solver`. The `precision` module is very short and it is used to communicate a kind type parameter (`adequate`) to the other program units in the program, and thus exemplifies precision portability. The `linear_equation_solver` is a typical example of a "real life" module that also demonstrates the power of Fortran 95 array language. This module contains three module procedures, the first of which, `solve_linear_equations`, uses the other two; `solve_linear_equations` is itself invoked by the main program.

```
MODULE precision
 ! adequate is a kind number of a real representation with
at least 10
 ! digits of precision and 99 digits range, that normally
results
 ! in 64-bit arithmetic.
  INTEGER, PARAMETER :: adequate =
SELECTED_REAL_KIND(10,99)
END MODULE precision

MODULE linear_equation_solver
  þþUSE 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
```

```
  n = SIZE (b)
  ! Check for compatible extents.
  error = SIZE(a, DIM=1) /= n .OR. SIZE(a, DIM=2) /= n &
           .OR. SIZE(x).LT. n
  IF (error) THEN
   x = 0.0
   RETURN
  END IF

  ! Append the right-hand side of the equation to m.
  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 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 (:, :), 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 the two modules defined above.
 USE precision
 USE linear_equation_solver
```

```
 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
```

Main Program

A main program is a program unit. There must be 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 95 program and often performs various computations by referencing procedures. A Fortran program may consist of only a main program, in which case all the program logic is contained within it.

A main program has the form:

[PROGRAM *program-name*]

  [*specification-part*] ...

  [*execution-part*] ...

  [*internal-subprogram-part*]

END [PROGRAM [*program-name*]]

NOTE. *If you use a* PROGRAM *statement, the* *program-name* *is global to the executable program, and must not be the same as the name of any other program unit, external procedure, or common block in the executable program, nor the same as any local name in the main program.*

Like other program units, a main program has three parts: a specification part, an execution part, and an internal procedure part that begins with the CONTAINS statement. All three parts are optional.

The data environment is described in the specification part. The data environment includes USE statements, declarations and specifications of the attributes of variables, type definitions, and initial values. An automatic object must not appear in the specification part of a main program.

The execution-part of a program unit contains executable-constructs such as the CASE, DO, IF, or WHERE constructs, and action statements such as assignment statements, data transfer statements, or IF statements.

Neither ENTRY nor RETURN statements are permitted in a main program.

The internal subprogram part contains one or more internal procedures.

The PROGRAM statement is optional; if it appears, the program name may be used on the END statement.

Note that the smallest valid Fortran 95 program consists of the single statement:

```
END
```

Block Data

A block data program unit initializes data values in common blocks. The syntax of a block data program unit is:

BLOCK DATA [*block-data-name*]

[*specification-part*]

END [BLOCK DATA [*block-data-name*]]

The specification part of a block data program unit can contain:

- Type declaration statements
- USE statements
- IMPLICIT statements
- COMMON statements
- DATA statements
- EQUIVALENCE statements
- Derived-type definitions
- Allowable attribute specification statements (see list below)

The following attributes can be specified:

Table 7-2 Allowable Block Data Attributes

PARAMETER	INTRINSIC	SAVE
DIMENSION	POINTER	TARGET

There must not be more than one unnamed BLOCK DATA program unit in an executable program. A named common block can be initialized in only one BLOCK DATA program unit. Pointer objects cannot be initialized.

An Intel Fortran extension allows data objects in an unnamed common block to be initialized as shown in the following example.

Example

```
BLOCK DATA blank
 COMMON//aa(3),ab(5)
 DATA aa/3*1.0/
 DATA ab/1.0,2.0,3*4.0/
END BLOCK DATA blank
```

Another extension in Intel Fortran allows the DATA initialization of variables in COMMON blocks, in any program unit or subprogram, and not just in BLOCK DATA. However a given COMMON block can only be initialized in one program unit only.

I/O and File Handling 8

This chapter describes input/output (I/O) and file handling as supported by Intel Fortran. Included at the end of the chapter are example programs that illustrate different types of I/O.

Records

The record is the basic unit of Fortran 95 I/O operations. It consists of either characters or binary values, depending upon whether the record is formatted or unformatted. The following sections describe both formatted and unformatted records, plus the special case of the end-of-file record.

Note that nonadvancing I/O makes it possible to read and write partial records. For more information, see "Nonadvancing I/O,".

Formatted Records

A formatted record consists of characters that have been edited during list-directed or namelist-directed I/O, or by a format specification during a data transfer. (For information about format specifications, see Chapter 9, I/O Formatting.) The length of a formatted record is measured in characters; there is no predefined maximum limit to the length of a formatted record.

Unformatted Records

An unformatted record consists of binary values in machine-representable format. The length of an unformatted record is measured in bytes. Unformatted records cannot be processed by list-directed or namelist-directed I/O statements or by I/O statements that use format specifications to edit data.

End-of-file Record

The end-of-file record is a special case: it contains no data and is the last record of a sequential file. The end-of-file record is written:

- By the `ENDFILE` statement
- When the file is closed—either explicitly by the `CLOSE` statement or implicitly when the program terminates—immediately following a write operation
- When a `BACKSPACE` statement executes after a write operation, before the file is backspaced

Files

A file is a collection of data, organized as a sequence of logical records. Records in a file must be either all formatted or all unformatted, except for the end-of-file record.

The following sections describe the two types of files, external files and internal files.

External Files

An external file is stored on disk, magnetic tape, or some other peripheral device. External files can be accessed sequentially or directly as described in "File Access Methods."

Scratch Files

A scratch file is a special type of external file. It is an unnamed, temporary file that exists only while it is open—that is, it exists no longer than the life of the program. Intel Fortran uses the *`tempnam`*`(3S)` system routine to

name the scratch file. The name becomes unavailable through the file system immediately after it is created, and it cannot be seen by the `ls(1)` command and cannot be opened by any other process.

To create a scratch file, you must include the `STATUS='SCRATCH'` specifier in the `OPEN` statement, as in the following:

```
OPEN (25, STATUS='SCRATCH')
```

In all other respects, a scratch file behaves like other external files. For an example of a program that uses a scratch file, see "Sequential- and Direct-access Example."

Internal Files

An internal file is stored in a variable where it exists for the life of the variable. Its main use is to enable programs to transfer data internally between a machine representation and a character format, using edit descriptors to make the conversions. (For more information about edit descriptors, see Chapter 9, I/O Formatting.)

An internal file can be one of the following:

- A character variable
- A character array
- A character array element
- A character substring
- An integer or real array (Intel Fortran extension)
- Any of the above that is either a field of a structure or a component of a derived type

Note, however, that a section of a character array with a vector subscript cannot be used as an internal file.

Accessing records in an internal file is analogous to accessing them in a formatted sequential file; see "Formatted I/O." For an example program that uses an internal file, see "Internal-file Example."

An internal file is not connected to a unit number and therefore does not require an `OPEN` statement. It is referenced as a character variable. In the following example, the `WRITE` statement transfers the data from `char_var`

to the internal file int_file, using list-directed formatting. Because int_file is declared to be 80 characters long, it is assumed that the length of char_var will be no more than 80 characters.

```
CHARACTER(LEN=80) :: int_file
...
WRITE (FILE=int_file, FMT=*) char_var
```

Connecting a File to a Unit

Before a program can perform any I/O operations on an external file, it must establish a logical connection between the file and a unit number. Once the connection is established, the program can reference the file by specifying the associated unit number (a non-negative integer expression). In the following example, the OPEN statement connects unit number 1 to the file my_data, allowing the WRITE statement to write the values in total_acct and balance to my_data:

```
OPEN (UNIT=1, FILE='my_data')
WRITE (1, '(F8.2)') total_acct, balance
```

The following sections describe three types of unit numbers:

- Those that are explicitly connected by means of the OPEN statement
- Preconnected unit numbers
- Automatically opened unit numbers

Connecting to an External File

Typically, the connection between an external file and a unit number is established by the OPEN statement. When the program is finished using the file, the connection is terminated by the CLOSE statement. Once the connection is terminated, the unit number can be assigned to a different file by means of another OPEN statement. Similarly, a file whose connection was broken by a CLOSE statement can be reconnected to the same unit number or to a different unit number.

A unit cannot be connected to more than one file at a time.

The following code establishes a connection between unit 9 and the external file `first_file`, which is to be by default opened for sequential access. When the program is finished with the file, the `CLOSE` statement terminates the connection, making the unit number available for connection to other files. Following the `CLOSE` statement, the program connects unit 9 to a different external file, `new_file`:

```
! connect unit 9 to first_file
 OPEN (9, FILE='first_file')
 ...
! process file
 ...
! terminate connection
 CLOSE (9)
! connect same unit number to new_file
 OPEN (9, FILE='new_file')
 ...
! process file
 ...
! terminate connection
 CLOSE (9)
```

Preconnected Unit Numbers

Unit numbers 5, 6, and 0 are preconnected; that is, they do not have to be explicitly opened and are connected to system-defined files, as follows:

- Unit 5 is connected to standard input—by default, the keyboard of the machine on which the program is running.
- Unit 6 is connected to standard output—by default, the terminal/display of the machine on which the program is running.
- Unit 0 is connected to standard error—by default, the terminal/display of the machine on which the program is running.

Each predefined logical unit is automatically opened when a Fortran 95 program begins executing and remains open for the duration of the program. This means, for example, that standard output can be used by a `PRINT` statement without prior execution of an `OPEN` statement. Attempting to `CLOSE` a preconnected logical unit has no effect.

A preconnected unit number can be reused with an OPEN statement that assigns it to a new file. Once a preconnected unit number is connected to a new file, however, it cannot be reconnected to its original designation.

You can use the input/output redirection (< and >) and piping (|) operators to redirect from standard input, standard output, or standard error to a file of your own choosing.

Automatically Opened Unit Numbers

Unit numbers that have not been associated with a file by an OPEN statement can be automatically opened using the READ or WRITE statement. When a file is automatically opened, a string is created of the form:

```
fort.XX
```

where *XX* is replaced by the unit number in the range 01 to 99.

If you have made an environment variable assignment of the form `fort.XX=path`, the file named in *path* is opened. Otherwise, the file whose name is `fort.XX` is opened in the current directory. If the file does not exist, it is created.

The following program:

```
PROGRAM Auto
WRITE (11,'(A)') 'Hello, world!'
END
```

writes the string

```
Hello, world!
```

to the file `fort.11`.

If this program is compiled to `a.out` and is run as follows (using `/bin/sh` or `/bin/ksh`)

```
fort.11=datafile
export fort.11
a.out
```

the output string is written to the file `datafile` instead of `fort.11`.

Automatically opened files are always opened as sequential files. Other characteristics of an automatically opened file, such as record length and format, are determined by the data transfer statement that creates the file. If the statement does not specify formatted, list-directed, or namelist-directed I/O, the file is created as an unformatted file.

File Access Methods

Intel Fortran allows both sequential access and direct access. You specify the access method with the `OPEN` statement when you connect the file to a unit number. The following example opens the file `new_data` for direct access:

```
OPEN(40, ACCESS='DIRECT', RECL=128,  &
   FILE='new_data')
```

If you do not specify an access method, the file is opened for sequential access.

The following sections describe both sequential and direct methods.

Sequential Access

Records in a file opened for sequential access can be accessed only in the order in which they were written to the file. A sequential file may consist of either formatted or unformatted records. If the records are formatted, you can use list-directed, namelist-directed, and formatted I/O statements to operate on them. If the records are unformatted, you must use unformatted I/O statements only. The last record of a sequential file is the end-of-file record.

The following sections describe the types of I/O that can be used with sequential files, namely:

- Formatted I/O
- List-directed I/O
- Namelist-directed I/O
- Unformatted I/O

Formatted I/O

Formatted I/O uses format specifications to define the appearance of data input to or output from the program, producing ASCII records that are formatted for display. (Format specifications are described in detail in Chapter 9, I/O Formatting.) Data is transferred and converted, as necessary, between binary values and character format. You cannot perform formatted I/O on a file that has been connected for unformatted I/O; see "Unformatted I/O."

Formatted I/O can be performed only by data transfer statements that include a format specification. The format specification can be defined in the statement itself or in a `FORMAT` statement referenced by the statement.

For an example of a program that accesses a formatted file, see Sequential- and Direct-access Example.

List-directed I/O

List-directed I/O is similar to formatted I/O in that data undergoes a format conversion when it is transferred but without the use of a format specification to control formatting. Instead, data is formatted according to its data type. List-directed I/O is typically used when reading from standard input and writing to standard output.

List-directed I/O uses the asterisk (`*`) as a format identifier instead of a list of edit descriptors, as in the following `READ` statement, which reads three floating-point values from standard input:

```
READ *, A, B, C
```

List-directed I/O can be performed only on internal files and on formatted, sequential external files. It works identically for both file types.

List-directed Input

Input data for list-directed input consists of values separated by one or more blanks, a slash, or a comma preceded or followed by any number of blanks. (No values may follow the slash.) An end-of-record also acts as a separator except within a character constant. Leading blanks in the first record read are not considered to be part of a value separator unless followed by a slash or comma.

Input values can be any of the values listed in Table 8-1. (A blank is indicated by the symbol b.)

Table 8-1 Input Values for List-directed I/O

Value	Meaning
z	A null value, indicated by two successive separators with zero or more intervening blanks (for example, ,b/).
c	A literal constant with no embedded blanks. It must be readable by an I, F, A, or L edit descriptor. Binary, octal, and hexadecimal data are illegal.
r*c	Equivalent to r (an integer) successive occurrences of *c* in the input record. For example, 5*0.0 is equivalent to 0.0 0.0 0.0 0.0 0.0.
r*z	Equivalent to *r* successive occurrences of *z*.

Reading always starts at the beginning of a new record. Records are read until the list is satisfied, unless a slash in the input record is encountered. The effect of the slash is to terminate the READ statement after the assignment of the previous value; any remaining data in the current record is ignored.

Table 8-2 outlines the rules for the format of list-directed input data.

Table 8-2 Format of list-directed Input Data

Data Type	Input Format Rules
Integer	Conforms to the same rules as integer constants.
Real and Double Precision	Any valid form for real and double precision. In addition, the exponent can be indicated by a signed integer constant (the Q, D, or E can be omitted), and the decimal point can be omitted for those values with no fractional part.

continued

Table 8-2 Format of list-directed Input Data (continued)

Data Type	Input Format Rules
Complex and Double Complex	Two integer, real, or double precision constants, separated by a comma and enclosed in parentheses. The first number is the real part of the complex or double complex number, and the second number is the imaginary part. Each of the numbers can be preceded or followed by blanks or the end of a record.
Logical	Consists of a field of characters, the first nonblank character of which must be a T for true or an F for false (excluding the optional leading decimal point). Integer constants may also appear.
Character	Same form as character constants. (Delimiting with single or double quotation marks is needed only if the constant contains any separators; delimiters are discarded upon input.) Character constants can be continued from one record to the next. The end-of-record does not cause a blank or any other character to become part of the constant. If the length of the character constant is greater than or equal to the length, *len*, of the list item, only the leftmost *len* characters of the constant are transferred. If the length of the constant is less than *len*, the constant is left-justified in the list item with trailing blanks.

List-directed Output

The format of list-directed output is determined by the type and value of the data in the output list and by the value of the `DELIM=` specifier in the `OPEN` statement. (For information about the `DELIM=` specifier, see the description of the `OPEN` statement in Chapter 10, Intel Fortran Statements.)
Table 8-3 summarizes the rules governing the display of each data type.

Table 8-3 Format of List-directed Output Data

Data Type	Output Format Rules
Integer	Output as an integer constant.
Real and Double Precision	Output with or without an exponent, depending on the magnitude. Also, output with field width and decimal places appropriate to maintain the precision of the data as closely as possible.
Complex	Output as two numeric values separated by commas and enclosed in parentheses.
Logical	If the value of the list element is `.TRUE.`, then `T` is output. Otherwise, `F` is output.
Character	Output using the A*len* format descriptor, where *len* is the length of the character expression (adjusted for doubling). If `DELIM='NONE'` (the default), no single (') or double (") quotation marks are doubled, and the records may not be suitable list-directed input. If the value specified by `DELIM=` is not `'NONE'`, only the specified delimiter is doubled. Character strings are output without delimiters, making them also unsuitable for list-directed input.

With the exception of character values, all output values are preceded by exactly one blank. A blank character is also inserted at the start of each record to provide ASA carriage control if the file is to be printed (see "ASA Carriage Control"). For example, the following statement:

```
PRINT *, 'Hello, world!'
```

outputs the line (where `b` indicates a blank):

```
bHello,bworld!
```

If the length of the values of the output items is greater than 79 characters, the current record is written and a new record started.

Slashes, as value separators, and null values are not output by list-directed `WRITE` statements.

Namelist-directed I/O

Namelist-directed I/O enables you to transfer a group of variables by referencing the name of the group, using the NML= specifier in the data transfer statement. The NAMELIST statement specifies the variables in the group and gives the group a name.

Like list-directed I/O, namelist-directed I/O does not use a format specification when formatting data but uses default formats, as determined by the data types.

In the following example, the NAMELIST statement defines the group name_group, which consists of the variables i, j, and c. The READ statement reads a record from the file connected to unit number 27 into name_group. The PRINT statement then writes the data from the variables in name_group to standard output. (As an extension, Intel Fortran allows this use of the PRINT statement in namelist I/O.)

```
INTEGER :: i, j
CHARACTER(LEN=10) :: c
NAMELIST /name_group/ i, j, c
...
READ (UNIT=27,NML=name_group)
PRINT name_group
```

Each namelist-directed output record begins with a blank character to provide for ASA carriage control if the records are to be printed (see "ASA Carriage Control").

Namelist-directed I/O can be performed only on formatted, sequential external files.

The following program illustrates namelist-directed I/O:

```
PROGRAM namelist
INTEGER, DIMENSION(4) :: ivar
CHARACTER(LEN=3), DIMENSION(3,2) :: cvar
LOGICAL :: lvar
REAL :: rvar
```

```
NAMELIST /nl/ ivar, cvar, lvar, rvar
READ (*,nl)
PRINT nl

END PROGRAM namelist
```

If the input data is:

```
&nl
ivar = 4,3,2,1
lvar=toodles
cvar=,,'QRS',2*,2*'XXX'
rvar=5.75E25, cvar(3,2)(1:2)='AB'
/
```

the output from this program will be:

```
b&NLbIVAR  = 4 3 2 1bCVAR  ='', 'QRS', '', '', 'XXX',
'ABX'bLVAR  = TbRVAR  = 5.75000E+25b/
```

The following sections describe the format of namelist-directed input and output. For detailed information about the NAMELIST statement, see Chapter 10, Intel Fortran Statements.

Namelist-directed Input

A namelist-directed input record takes the following form:

1. An ampersand character (&) immediately followed by a namelist group name. The group name must have been previously defined by a NAMELIST statement.

   As an extension, the dollar sign ($) can be substituted for the ampersand (&).

2. A sequence of name-value pairs and value separators. A name-value pair consists of the name of a variable in the namelist group, the equals sign (=), and a value having the same format as for list-directed input (*z*, *c*, *r*c*, and *r**). A name-value pair can appear in any order in the sequence or can be omitted.

A value separator may be one of the following:

- Blanks
- Tabs
- Newlines
- Any of the above with a single comma

A NAMELIST comment may appear following a value separator. It begins with an exclamation mark (!), except when the exclamation mark is a character in a literal constant. The comment extends to the end of the NAMELIST record. A slash appearing within the comment does not end the record; the comment itself ends the input record. A comment may appear as the first non-blank character in an input record, but in this case, the input record consists only of the comment. NAMELIST comments are ignored.

3. A terminating slash (/).

As an extension, ($END) can be substituted for the slash.

Names of character type may be qualified by substring range expressions and array names by subscript/array section expressions. If the name in a name-value pair is that of an array, the number of the values following the equals sign must be separated by value separators and must not exceed the number of elements in the array. If there are fewer values than elements, null values are supplied for the unfilled elements.

Namelist-directed input values are formatted according to the same rules as for list-directed input data; see Table 8-2.

Namelist-directed Output

The output record for namelist-directed I/O has the same form as the input record, but with these exceptions:

- The namelist group name is always in uppercase.
- Logical values are either T or F.
- As in list-directed output, character values are output without delimiters by default, making them unsuitable for namelist-directed input. However, you can use the DELIM= specifier in the OPEN statement to specify the single or double quotation mark as the delimiter to use for character constants.
- Only character and complex values may be split between two records.

Unformatted I/O

Unformatted I/O does not perform format conversion on data it transfers. Instead, data is kept in its internal, machine-representable format. You cannot perform unformatted I/O on files that have been connected for formatted I/O (see "Formatted I/O").

Unformatted I/O is more efficient than formatted, list-directed, or namelist-directed I/O because the transfer occurs without the conversion overhead. However, because unformatted I/O transfers data in internal format, it is not portable.

Direct Access

When performing I/O on a direct-access file, records can be read or written in any order. The records in a direct-access file are all of the same length.

Reading and writing records is accomplished by `READ` and `WRITE` statements containing the `REC=` specifier. Each record is identified by a record number that is a positive integer. For example, the first record is record number 1; the second, number 2; and so on. If `REC=` is not specified:

- The `READ` statement inputs from the current record, and the file pointer moves to the next record.
- The `WRITE` statement outputs to the record at the position of the file pointer, and the file pointer is advanced to the next record.

As an extension, Intel Fortran allows sequential I/O statements to access a file connected for direct access.

Once established, a record number of a specific record cannot be changed or deleted, although the record may be rewritten. A direct-access file does not contain an end-of-file record as an integral part of the file with a specific record number. Therefore, when accessing a file with a direct-access read or write statement, the `END=` specifier is not valid and is not allowed.

Direct-access files support both formatted and unformatted record types. Both formatted and unformatted I/O work exactly as they do for sequential files. However, you cannot perform list-directed, namelist-directed, or nonadvancing I/O on direct-access files.

For an example program that uses direct access, see "Sequential- and Direct-access Example."

Nonadvancing I/O

By default, a data transfer leaves the file positioned after the last record read or written. This type of I/O is called advancing. Fortran 95 also allows nonadvancing I/O, which positions the file just after the last character read or written, without advancing to the next record. It is character-oriented and can be used only with external files opened for sequential access. It cannot be used with list-directed or namelist-directed I/O.

To use nonadvancing I/O, you must specify `ADVANCE='NO'` in the `READ` or `WRITE` statement. The example program, "Sequential- and Direct-access Example" uses nonadvancing I/O in the first `WRITE` statement, which is reproduced here:

```
WRITE (6, FMT='(A)', ADVANCE='NO') ' Enter number to
insert in list: '
```

The effect of nonadvancing I/O on the `WRITE` statement is to suppress the newline character that is normally output at the end of a record. This is the desired effect in the example program: by using a nonadvancing `WRITE` statement, the user input to the `READ` statement stays on the same line as the prompt.

(You can get the same effect with the newline ($) edit descriptor, an Intel Fortran extension that also suppresses the carriage-return/linefeed sequence at the end of a record; see Chapter 9, I/O Formatting.)

For an example program that illustrates nonadvancing I/O in a `READ` statement, see "Nonadvancing-I/O Example." For more information about nonadvancing I/O and the `ADVANCE=` specifier, see the `READ` and `WRITE` statements in Chapter 10, Intel Fortran Statements.

I/O Statements

Intel Fortran supports three types of I/O statements:

- Data transfer statements (see Table 8-4)
- File positioning statements (see Table 8-5)
- Auxiliary statements (see Table 8-6)

For detailed information about all I/O statements, refer to Chapter 10, Intel Fortran Statements.

Table 8-4 Data Transfer Statements

Statement	Use
ACCEPT	Inputs data from the preconnected default input device (standard input). (Extension)
DECODE	Inputs data from an internal file. (Extension)
ENCODE	Outputs data to an internal file. (Extension)
PRINT	Outputs data to the preconnected default output device file (standard output).
READ	Inputs data from a connected or automatically opened unit.
TYPE	Synonym for the PRINT statement. (Extension)
WRITE	Outputs data to a connected or automatically opened unit.

NOTE. *Although the* DECODE *and* ENCODE *statements are available as compatibility extensions for use with internal files, they are nonportable and are provided for compatibility with older versions of Fortran. To keep your programs standard-conforming and portable, you should use the* READ *and* WRITE *statements with both external and internal files.*

ACCEPT and TYPE are also available as compatibility extensions for reading from standard input and writing to standard output. However, if you wish your program to be portable, you should use the READ *and* PRINT *statements instead of the* ACCEPT *and* TYPE *statements.*

Table 8-5 File Positioning Statements

Statement	Use
BACKSPACE	Moves the file pointer of the connected sequential file to the start of the previous record.
ENDFILE	Writes an end-of-file record as the next record of the sequential file.
REWIND	Moves the file pointer of the connected file to the initial point of the file.

Table 8-6 Auxiliary Statements

Statement	Use
CLOSE	Disconnects a unit from a file.
INQUIRE	Requests information about a file or unit.
OPEN	Connects an existing file to a unit, creates a file and connects it to a unit, or changes certain specifiers of a connection between a file and a unit.

Syntax of I/O Statements

The general syntactic form of file-positioning and auxiliary statements is:

statement-name (io-specifier-list)

where

statement-name is one of the statements listed in Table 8-5 or Table 8-6.

io-specifier-list is a comma-separated list of I/O specifiers that control the statement's operation.

The general form of a data-transfer statement is:

statement-name (io-specifier-list) data-list

where

statement-name	is one of the statements listed in Table 8-4.
io-specifier-list	is a comma-separated list of I/O specifiers that control the data transfer.
data-list	is a comma-separated list of data items.

The following sections describe the I/O specifiers and the form of *data-list*. For detailed information about the syntax of individual I/O statements, see Chapter 10, Intel Fortran Statements.

I/O Specifiers

I/O specifiers provide I/O statements with additional information about a file or a data transfer operation. They can also be used (especially with the INQUIRE statement) to return information about a file. Table 8-7 lists all I/O specifiers supported by Intel Fortran and identifies the statements in which each can appear. Note that the ACCEPT, DECODE, ENCODE, and TYPE statements are not listed in the table as they are nonstandard. All I/O specifiers and statements are fully described in Chapter 10, Intel Fortran Statements.

Table 8-7 I/O Statements and Specifiers (Y=Yes)

	I/O statements								
I/O Specifiers	BACKSPACE	CLOSE	ENDFILE	INQUIRE	OPEN	PRINT	READ	REWIND	WRITE
ACCESS=				**Y**	**Y**				
ACTION=				**Y**	**Y**				

continued

Table 8-7 I/O Statements and Specifiers (continued) **(Y=Yes)**

	I/O statements								
I/O Specifiers	BACKSPACE	CLOSE	ENDFILE	INQUIRE	OPEN	PRINT	READ	REWIND	WRITE
ADVANCE=							Y		Y
ASSOCIATEVARIABLE=					Y				
BINARY=				Y					
BLANK=				Y	Y				
BLOCKSIZE=				Y	Y				
BUFFERCOUNT=					Y				
CARRIAGECONTROL=				Y					
DEFAULTFILE=				Y	Y				
DELIM=				Y	Y				
DIRECT=				Y					
DISPOSE= (same as STATUS)		Y							
DISP[OSE]=					Y				
EXTENDSIZE=					Y				
END=							Y		
EOR=							Y		
ERR=	Y	Y	Y	Y	Y		Y	Y	Y
EXIST=				Y					
FILE=				Y	Y				
FILEOPT=					Y				

continued

Table 8-7 I/O Statements and Specifiers (continued) **(Y=Yes)**

	I/O statements								
I/O Specifiers	BACKSPACE	CLOSE	ENDFILE	INQUIRE	OPEN	PRINT	READ	REWIND	WRITE
FMT=							Y		Y
FORM=				Y	Y				
FORMATTED=				Y					
INITIALSIZE=					Y				
IOLENGTH=				Y					
IOFOCUS=					Y				
IOSTAT=	Y	Y	Y	Y	Y		Y	Y	Y
MAXREC ONOPEN=				Y	Y				
NAME=				Y	Y				
NAMED=				Y					
NEXTREC=				Y					
NML=							Y		Y
NOSPANBLOCKS=					Y				
NUMBER=				Y					
OPENED=				Y					
ORGANIZATION=				Y	Y				
PAD=				Y	Y				
POSITION=				Y	Y				
READ=				Y					
READONLY=					Y				

continued

Table 8-7 I/O Statements and Specifiers (continued) **(Y=Yes)**

	I/O statements								
I/O Specifiers	BACKSPACE	CLOSE	ENDFILE	INQUIRE	OPEN	PRINT	READ	REWIND	WRITE
READWRITE=				Y					
RECORDTYPE=				Y	Y				
REC=							Y		Y
RECL= or RECORDSIZE=				Y	Y				
SEQUENTIAL=				Y					
SHARE=				Y	Y				
SHARED=					Y				
STATUS=		Y			Y				
TITLE on OPEN=					Y				
UNFORMATTED=				Y					
UNIT=	Y	Y	Y	Y	Y		Y	Y	Y
USEROPEN=					Y				
WRITE=				Y					

Table 8-8 I/O Specifiers Values

I/O Specifiers	Possible Values	Default Values
ACCESS=	'DIRECT' SEQUENTIAL	'SEQUENTIAL'
ACTION=	'READ', 'WRITE', 'READWRITE'	Processor-Dependent
ADVANCE=	'YES', 'NO'	'Yes'

continued

Table 8-8 I/O Specifiers Values (continued)

I/O Specifiers	Possible Values	Default Values
ASSOCIATEVARIABLE=	*variable-name*	No default
BINARY=	'YES', 'NO', 'UNKNOWN'	Formatted: 'Yes' Unformatted: 'No'
BLANK=	'NULL', 'ZERO', 'UNDEFINED'	'NULL'
BLOCKSIZE=	*integer-expression*	512
BUFFERCOUNT=	*integer-expression*	1
CARRIAGECONTROL=	'FORTRAN', 'NONE', 'LIST'	Formatted: 'LIST' Unformatted: 'NONE'
DEFAULTFILE=	*character-expression*	Current working directory
DELIM=	'APOSTROPHE', 'QUOTE', 'NONE'	'NONE'
DIRECT=	'YES', 'NO', 'UNKNOWN'	'NO'
DISPOSE= (same as STATUS)	'OLD', 'NEW', 'UNKNOWN', 'REPLACE', 'SEARCH'	'NEW'
DISP[OSE]=	'KEEP', 'SAVE', 'DELETE', 'PRINT', 'PRINT/DELETE', 'SUBMIT', 'SUBMIT/DELETE'	'KEEP'
EXTENDSIZE=	*integer-expression*	512
END=	*Label*	No default
EOR=	*Label*	No default
ERR=	*Label*	No default
EXIST=	.TRUE. .FALSE.	No default
FILEOPT=	*character-constant*	
FMT=	*character-expression, *,* *label*	No default
FORM=	'FORMATTED'	'FORMATTED' for sequential access

continued

Table 8-8 I/O Specifiers Values (continued)

I/O Specifiers	Possible Values	Default Values
FORMATTED=	'YES', 'NO', 'UNKNOWN'	Direct access: 'No' Sequential access: 'Yes'
INITIALSIZE=	*integer-expression*	
IOFOCUS=	*logical expression*	If unit is '*', defaults to .FALSE., otherwise .TRUE.
IOLENGTH=	*length*	
IOSTAT=	*scalar-default-integer-value*	No default
MODE=	'READ', 'WRITE', 'READWRITE'	'READWRITE'
NAME=	*filename* or undefined	
NAMED=	.TRUE. or .FALSE.	
NEXTREC=	next-record-#, undefined	
NML=	namelist-group-name	No default
NOSPANBLOCKS	no values	Spanning blocks OK (no value)
NUMBER=	*num*	
OPENED=	.FALSE., .TRUE.	
ORGANIZATION=	'SEQUENTIAL', 'RELATIVE', 'UNKNOWN'	
PAD=	'YES', 'NO'	'YES'
POSITION=	'ASIS', 'REWIND', 'APPEND'	'ASIS'
READ=	'YES', 'NO', 'UNKNOWN'	
READONLY	No value	

continued

Table 8-8 I/O Specifiers Values (continued)

I/O Specifiers	Possible Values	Default Values
RECORDTYPE=	'FIXED', 'VARIABLE', 'SEGMENTED', 'STREAM', 'STREAM_CR'	Direct access: 'FIXED' Otherwise: 'VARIABLE'
REC=	*scalar-integer expression*	No default
RECL=	*positive-scalar-integer expression*	Processor-Dependent
SEQUENTIAL=	'YES', 'NO', 'UNKNOWN'	
SHARE=	*scalar-integer-expression*	
SHARED=	No value	
SIZE=	*scalar-default-integer variable*	
STATUS=	'OLD', 'NEW', 'UNKNOWN', 'REPLACE', 'SEARCH'	'UNKNOWN'
TITLE=	*character expression*	
UNFORMATTED=	'YES', 'NO', 'UNKNOWN'	
UNIT=	*scalar-integer-expression*	No default
USEROPEN=	*procedure-name*	
WRITE=	'YES', 'NO', 'UNKNOWN'	

I/O Data List

The I/O data list can be used with any data transfer statement except namelist I/O (see "Namelist-directed I/O." The general form of the I/O data list is:

item1[,*item2*. . .]

where

item is a either a simple data element or an implied-DO loop.

The following sections describe simple data elements and the implied-DO loop.

Simple Data Elements

In a read operation, the simple data element specifies a variable, which can include:

- A scalar
- An array
- An array element or section
- A character substring
- A structure
- A component of a structure
- A record
- A field of a record
- A pointer

In a write operation, the simple data element can include any variable that is valid for a read operation, plus most expressions. Note that, if the expression includes a function reference, the function must not itself perform I/O.

The output list in the following `PRINT` statement contains two simple list elements, a variable named `radius` and an expression formed from `radius`:

```
99 FORMAT('Radius = ', F10.2, 'Area = ', F10.2)
  PRINT 99, radius, 3.14159*radius**2
```

The next `READ` statement contains three simple elements: a character substring `(name(1:10))`, a variable `(id)`, and an array name `(scores)`:

```
88 FORMAT(A10,I9,10I5)
  READ(5, 88) name(1:10), id, scores
```

If an array name is used as a simple data element in the I/O list of a `WRITE` statement, then every element in the array will be displayed. If a format specification is also used, then the format will be reused if necessary to display every element. For example, the following code

```
  INTEGER :: i(10) = (/1,2,3,4,5,6,7,8,9,10/)
88 FORMAT('  N1:',I5, ' N2:',I5, ' N3:',I5)
  PRINT 88, i
```

will output the following:

```
N1:  1 N2:  2 N3:  3
N1:  4 N2:  5 N3:  6
N1:  7 N2:  8 N3:  9
N1:  10 N2:
```

The following restrictions apply to the use of arrays in input and output:

- Sections of character arrays that specify vector-valued subscripts cannot be used as internal files.
- An assumed-size array cannot be referenced as a whole array in an input or output list.

The following restrictions apply to the use of structures and records in input and output:

- All components of the structure or fields of the record must be accessible within the scoping unit that contains the data transfer statement.
- Every component of the structure or field of the record is written.
- A structure in an I/O list must not contain a pointer that is an ultimate component—that is, the last component in a variable reference. In the expression *a%b%c*, *a* and *b* can be pointers, but not *c*.

Implied-DO Loop

An implied-DO loop consists of a list of data elements to be read, written, or initialized, and a set of indexing parameters. The syntax of an implied-DO loop in an I/O statement is:

```
(list , index = init , limit [, step ])
```

where

list	is an I/O list, which can contain other implied-DO loops.
index	is an integer variable that controls the number of times the elements in *list* are read or written. The use of real variables is supported but obsolescent.
init	is an expression that is the initial value assigned to *index* at the start of the implied-DO loop.
limit	is an expression that is the termination value for *index*.

step is an expression by which *index* is incremented or decremented after each execution of the DO loop. *step* can be positive or negative. Its default value is 1.

Inner loops can use the indexes of outer loops.

The implied-DO loop acts like a DO construct. The range of the implied-DO loop is the list of elements to be input or output. The implied-DO loop can transfer a list of data elements that are valid for a write operation. *index* is assigned the value of *init* at the start of the loop. Execution continues in the same manner as for DO loops (see Chapter 6, Execution Control).

The implied-DO loop is generally used to transmit arrays and array elements, as in the following:

```
INTEGER :: b(10)
PRINT *, (b(i), i = 1,10)
```

If b has been initialized with the values 1 through 10 in order, the PRINT statement will produce the following output:

```
1 2 3 4 5 6 7 8 9 10
```

If an unsubscripted array name occurs in the list, the entire array is transmitted at each iteration. For example:

```
REAL :: x(3)
PRINT *, (x, i=1, 2)
```

If x has been initialized to be [1 2 3], the output will be:

```
 1.0 2.0 3.0 1.0 2.0 3.0
```

The list can contain expressions that use the index value. For example:

```
REAL :: x(10) = (/.1, .2, .3, .4, .5, .6, &
          .7, .8, .9, 1 /)
PRINT *, (i*2, x(i*2), i = 1, 5)
```

print the numbers

```
2 .2 4 .4 6 .6 8 .8 10 1
```

Implied-DO loops can also be nested. The form of a nested implied-DO loop in an I/O statement is:

(((*list*, *index1* = *init1*, *limit1*, *step1*), *index2* = *init2*, *limit2*, *step2*)
... *indexN* = *initN*, *limitN*, *stepN*)

Nested implied-DO loops follow the same rules as do other nested DO loops. For example, given the following statements:

```
REAL :: a(2,2)

a(1,1) = 1
a(2,1) = 2
a(1,2) = 3
a(2,2) = 4

WRITE(6,*)((a(i,j),i=1,2),j=1,2)
```

the output will be:

```
 1.0 2.0 3.0 4.0
```

The first, or nested DO loop, is completed once for each execution of the outer loop.

ASA Carriage Control

If you are on a UNIX* system, the program *asa(1)* processes the output of a Fortran 95 program that uses ASA carriage control characters so that it can be properly handled by many printers.

The syntax of asa is:

asa [*file-names*]

where

file-names is a list of file names to be output with carriage control characters interpreted according to ASA rules.

Table 8-9 lists the ASA carriage-control characters and their meanings.

Table 8-9 ASA Carriage-control Characters

Character	Meaning
blank	Advance one line.
0	Advance two lines.

continued

Table 8-9 ASA Carriage-control Characters (continued)

Character	Meaning
1	Advance to top of next page.
+	Do not advance; overstrike previous line.

The `asa` reads input from *`file-names`* or from standard input if *`file-names`* is not specified. The first character of each line is interpreted as a control character. Lines beginning with any character other than those listed in Table 8-9 are interpreted as if they began with a blank, and an appropriate diagnostic appears on standard error. The first character of each line is not printed. The `asa` program interprets input lines and sends its output to standard output. Each input file begins on a new page.

To properly view the output of programs that use `asa` carriage control characters, `asa` should be used as a filter. For example, the following example pipes the output of `fortran_asa`, an executable Intel Fortran program that outputs lines with ASA carriage control characters, through the `asa` filter to the line printer command, `lp`:

```
fortran_asa | asa | lp
```

On Windows NT systems, if you have the Mortice Kern Systems (MKS) toolkit, you can use the MKS version of `asa` and use the `print` command rather than `lp`.

Example Programs

This section gives example programs that illustrate I/O and file-handling features of Intel Fortran.

Internal-file Example

The following program illustrates how internal files can use edit descriptors internally. The comments within the program explain in detail what the program does.

```
! ifile.f90
```

This program is a driver for the function roundoff, which truncates and rounds a floating-point number to a requested number of decimal places. The main program prompts for two numbers, a double-precision number

and an integer. These are passed to the function roundoff as arguments. The double-precision argument (x) is the value to be rounded, and the integer (n) represents the number of decimal places for rounding. The function converts both arguments to character format, storing them in separate internal files. The function uses the F edit descriptor (to which n in character format has been appended) to round x. This rounded value is finally converted back from a character string to a double-precision number, which the function returns.

```
PROGRAM internal_file
REAL (KIND=8) :: x, y, roundoff
```

Use nonadvancing I/O to suppress the newline and keep the prompt on the same line as the input.

```
  WRITE (6, ’(X, A)’, ADVANCE=’NO’) ’Enter a real number: ’
  READ (5, ’(F14.0)’) x
  WRITE (6, ’(A)’) ’How many significant digits (1 - 9) to the’
  WRITE (6, '(X, A)', ADVANCE='NO') 'right of the decimal point?’
! Don’t enter a number greater than you input into x!
  READ (5, '(I1)') n
  y = roundoff(x, n)
  PRINT *, y
  END
```

This function truncates and rounds x to the number of decimal places specified by n. The function performs no error checking on either argument.

```
REAL (KIND=8) FUNCTION roundoff(x, n)
INTEGER :: n
REAL (KIND=8) :: x
CHARACTER (LEN=14) :: dp_val
CHARACTER :: dec_digits
```

Use an edit descriptor to convert the value of n to a character, writing the result to the internal file dec_digits.

```
WRITE (dec_digits, '(I1)') n
```

Concatenate dec_digits to the string 'F14.'. The complete string forms an edit descriptor that will convert the binary value of x to a formatted character string that formats the value. The character represents the requested level of precision. The formatted number is stored in the internal file dp_val.

```
WRITE (dp_val, '(F14.''//dec_digits//'')') x
```

Re-convert the formatted record in dp_val to a binary value that the function will return.

```
READ (dp_val, '(F14.0)') roundoff
END
```

When compiled with the command line:

```
f90 -o ifile ifile.f90
```

the program writes the following to standard output:

```
Enter a real number: 3.1415927
How many significant digits (1 - 9) to the
right of the decimal point? 3
3.142
```

Nonadvancing-I/O Example

The following program illustrates nonadvancing I/O on input. It reads a formatted sequential file, each record of which consists of a name followed by one or more grades. For each record, the program first reads the name, then uses a DO loop to read all grades in the record. After reading the last grade, the program computes and displays the average. The comments explain what the program does.

```
! nonadv.f90
! assumptions: no errors in file (hence no
error-checking),
! name field occupies 20 characters, and at least one
grade

PROGRAM proc_grades

  INTEGER :: grade, count, sum, average
```

```
  CHARACTER(LEN=20) name

  OPEN(20, FILE='grades')
  WRITE (6, 10) "Name", "Average"
  WRITE (6, *) "-------------------------"
  DO
   sum = 0
   count = 0
```

Read the first field of the record, using nonadvancing I/O so as not to advance beyond that field. Note that the `END=` specifier causes the program to exit the loop and branch to the statement labeled 999 when it detects end-of-file.

```
READ(20, "(A20)", ADVANCE='NO', END=999) name
! read in grades
DO
```

Again, use non-advancing I/O to avoid advancing to the next record after each read. The `EOR=` specifier causes the program to break out of the loop and resume execution at the statement labeled 99.

```
    READ(20, "(I3)", ADVANCE='NO', EOR=99) grade
    count = count + 1
    sum = sum + grade
   END DO
99  average = sum/count
   ! Write each student's name and average.
   WRITE(6, 20) name, average
  END DO
10 FORMAT (X, A, T21, A)
20 FORMAT (X, A, I3)

999 CLOSE(20)
END PROGRAM proc_grades
```

Use the following command line to compile the program:

```
f90 -o nonad nonad.f90
```

If the file `grades` contains the following records:

```
SandrabDelfordbbbbb79b85b81b72100100
JoanbArunsoeltonbbbbbb8b64b77b79
EdenbPhilpottsbbbbbb100b92b87b65bb0
SoamesbJenynsbbbbbbbb97b78b58b75b88b73
AnitabJaysonbbbbbbbbb93b85b90b95b68b72b93
JoebKorzeniowskibbbbbb9b27b35b49
HarrietbMyrlebbbbbbbb84b78b93b95b97b92b84b93
PetebHartleybbbbbbbbb67b54b58b71b93b58
```

the program will produce the following output:

```
Name             Average
-------------------------
Sandra Delford     86
Joan Arunsoelton   57
Eden Philpotts     68
Soames Jenyns      78
Anita Jayson       85
Joe Korzeniowski   30
Harriet Myrle      89
Pete Hartley       66
```

Sequential- and Direct-access Example

The following program illustrates both sequential and direct access on external files. The file opened for direct access is a scratch file. The comments explain what the program does.

```
! dir_acc.f90
```

This program uses an external file and a scratch file to insert a number into a list of numerically sorted numbers. The sorted list is held in a external file. The program uses the scratch file as a temporary holding place. The program uses the direct access method with the scratch file.

```
PROGRAM direct_access

REAL :: number_to_insert, number_in_list
INTEGER :: rec_num, ios1, ios2, i

! Initialize counter.
```

```
rec_num = 0
```

`ios1` must be initialized to 0 so that the error-handling section at the end of the program will work correctly

```
ios1= 0
! Open the scratch file and the sequential data file
OPEN (18, FILE='list', STATUS='UNKNOWN', IOSTAT=ios1,
ERR=99)
OPEN (17, STATUS='SCRATCH', ACCESS='DIRECT',
FORM='FORMATTED', &
        IOSTAT=ios1, ERR=99, RECL=16)
```

Use non-advancing I/O to suppress newline at the end of output record, thus keeping the prompt on the same line with the input.

```
WRITE (6, FMT='(A)', ADVANCE='NO') ' Enter number to
insert in list: '
READ *, number_to_insert
```

Read from sorted list and write to scratch file until we find where to insert number; then, write `number_to_insert`, and continue writing rest of sorted numbers to scratch file.

```
DO WHILE (ios1 >= 0) Enter loop only if OPEN didn't
encounter EOF
The END=15 specifier in the READ statement gets us out
of the loop, once we're in it.
READ (18, *, END=10, IOSTAT=ios2, ERR=99) number_in_list
IF (number_to_insert <= number_in_list) THEN
rec_num = rec_num + 1  ! add the new record
WRITE(17, 100, REC=rec_num) number_to_insert
DO
rec_num = rec_num + 1
WRITE(17, 100, REC=rec_num) number_in_list
READ (18, *, END=15, IOSTAT=ios2, ERR=99)
number_in_list
```

```
END DO
ELSE
rec_num = rec_num + 1
WRITE (17, 100, REC=rec_num) number_in_list
END IF
END DO
```

The file is empty or the item goes at the end of file. Add 1 to `rec_num` for the record to be inserted.

```
10 rec_num = rec_num + 1
WRITE (17, 100, REC=rec_num) number_to_insert
```

Copy the scratch file to the data file. But first rewind so that we start writing at beginning of the data file.

```
15 REWIND 18
```

Read from scratch file and write to data file

```
DO i = 1, rec_num
 READ (17, 100, REC=i) number_in_list
 WRITE (18, *) number_in_list
END DO

CLOSE (18)
CLOSE (17)
STOP 'Inserted!'
! Error handling section
99 IF (ios1 /= 0) THEN
 WRITE (7, 200) "Open error = ", ios1
ELSE
 WRITE (7, 200) "Read error = ", ios2
END IF

100 FORMAT (F16.6)
200 FORMAT (A, 2I6)

END
```

Use the following command line to compile the program:

```
f90 -o dir_acc dir_acc.f90
```

If the file `list` contains the following records:

```
3.01
6.0
6.22
7.54
27.9
```

and the input is

```
6.15
```

the file rewritten by the program will contain the following numbers:

```
3.010000
6.000000
6.150000
6.220000
7.540000
27.900000
```

I/O Formatting

9

I/O formatting occurs during data transfer operations when data is converted between its machine-readable binary representation and human-readable character format. Although unformatted data transfers are faster because they do not incur the overhead of data conversion, I/O formatting is useful for displaying data in a human-readable form and for transferring data between machines with different machine representations for a data type.

I/O formatting can be implicit or explicit. Implicit formatting occurs during list-directed and namelist-directed I/O: data is converted without programmer intervention, based on the data types of the I/O list items. (For information about `list-directed` and `namelist-directed` I/O, see Chapter 8, I/O and File Handling.) Explicit formatting occurs under the control of the programmer, who specifies how the data is to be converted.

This chapter describes explicit I/O formatting and includes information about the following:

- `FORMAT` statement
- Format specification
- Edit descriptors
- Format specification in character expressions
- Nested format specifications
- Interaction between format specification and I/O list

FORMAT Statement

The function of the FORMAT statement is to specify formatting information that can be used by one or more of the following data transfer statements:

- ACCEPT (extension)
- DECODE (extension)
- ENCODE (extension)
- PRINT
- READ
- TYPE (extension)
- WRITE

The syntax of the FORMAT statement is:

label FORMAT (*format-spec*)

where

label	is a statement label.
format-spec	is a format specification consisting of a comma-separated list of edit descriptors. For detailed information about edit descriptors, see the next section.

The FORMAT statement must include *label* so that the data transfer statements can reference it. One FORMAT statement can be referenced by many data transfer statements. In the following example, both the READ and WRITE statements reference the same FORMAT statement:

```
READ(UNIT=22, FMT=10)ivar, fvar
WRITE(17, 10)ivar, fvar
...
10 FORMAT(I7, F14.3)
```

For additional information about the FORMAT statement and data transfer statements, see Chapter 10, Intel Fortran Statements.

Format Specification

A format specification consists of a list of edit descriptors that define the format of data to be read with a `READ` statement, or written with a `WRITE` or `PRINT` statement. A format specification can appear either in a `FORMAT` statement or in a character expression in a data transfer statement.

The syntax of a format specification is:

`[`*`descriptor1`*`[, descriptor2...]]`

where

`descriptor` is an edit descriptor that is used to convert data between its internal (binary) format and an external (character) format. Edit descriptors are described in detail in the following section.

Note that format specifications are not used in list-directed and namelist-directed I/O.

Variable Expressions in Formats

With variable expressions, you can replace an integer constant in any arbitrary expression. You must enclose the expression in angle brackets. For example, in the following statement

```
FORMAT( 4f8.2 )
```

you can replace the 8 with the variable X as in the following:

```
FORMAT( 4f<X>.2 )
```

Also, you can use more complicated expressions within the brackets as follows:

```
FORMAT( 4f<2*X+Y>.2 )
```

Further, you can replace the 4 or the 2 by any expression.

The following rules apply to using variable expressions in formats:

- The expression is re-evaluated each time it is found in a format scan.
- If necessary, the expression is converted to integer type.
- All valid Fortran 95 statements are allowed, including function calls.

- You cannot use variable expressions in formats generated at runtime.
- The single exception is the n in an nH... edit descriptor. See the following section for a description of edit descriptors.

Edit Descriptors

Edit descriptors are encoded characters that describe data conversion between an internal (binary) format and an external (character) format. There are three types of edit descriptors:

- Data edit descriptors define the format of data to be read or written, such as its type and width (in characters). All data edit descriptors are repeatable; that is, they can be preceded by a positive integer that specifies the number of times the edit descriptor is to be replicated.
- Control edit descriptors specify editing information, such as the number of spaces between input items, treatment of blanks in input, and scale factors. Of the control edit descriptors, only the slash (/) is repeatable.
- Character string edit descriptors output text. None of these is repeatable.

Output format edit descriptors can produce default minimum field widths that eliminate "white space" on output, for formatting numeric values. To specify a minimum field width, the width *w* should be zero when used with the I,B,O,Z, or F edit descriptors.

All of the edit descriptors supported by Intel Fortran are listed in Table 9-1. As indicated by the syntax descriptions included in the table, the field width specification (*w*) is optional for all data edit descriptors in Intel Fortran. (Note that the Standard defines the field width specifier to be optional only for the A edit descriptor.) The table also identifies which edit descriptors are repeatable and which can be used on input, output, or both.

Table 9-1 Edit Descriptors

Descriptor	Type	Repeatable?	I/O Use	Function
'...' or '...'	Character String	No	Output	Output enclosed string.
$	Control	No	Output	Suppress newline at end of output.

continued

Table 9-1 Edit Descriptors (continued)

Descriptor	Type	Repeatable?	I/O Use	Function
/ (slash)	Control	Yes	Input and output	End current record and begin new record.
: (colon)	Control	No	Input and output	Stop formatting if I/O list is exhausted.
A[*w*] or R[*w*]	Data	Yes	Input and output	Convert character AND Non-character data.
B[*w*[.*m*]]	Data	Yes	Input and output	Convert integer data, using binary base.
BN	Control	No	Input and output	Ignore blanks in numeric input data.
BZ	Control	No	Input and output	Treat blanks as zeroes in numeric input data.
D[*w.d*[E*e*]]	Data	Yes	Input and output	Convert real type data with exponent.
E[*w.d*[E*e*]]	Data	Yes	Input and output	Convert real type data with exponent.
EN[*w.d*[E*e*]]	Data	Yes	Input and output	Convert real type data, using engineering notation.
ES[*w.d*[E*e*]]	Data	Yes	Input and output	Convert real type data, using scientific notation.
F[*w.d*]	Data	Yes	Input and output	Convert real type data without exponent.
G[*w.d*[E*e*]]	Data	Yes	Input and output	Convert numeric data, all types.
*n*H*s*	Character String	No	Output	Output following *n* characters.
I[*w*[.*m*]]	Data	Yes	Input and output	Convert integer numeric data.
L[*w*]	Data	Yes	Input and output	Convert logical data.
O[*w*[.*m*]]	Data	Yes	Input and output	Convert integer data, using octal base.

continued

Table 9-1 Edit Descriptors (continued)

Descriptor	Type	Repeatable?	I/O Use	Function
*k*P	Control	No	Input and output	Set scale factor to *k*.
Q[*w.d*]	Data	Yes	Input and output	Convert real type data with exponent.
Q	Control	No	Input	Return number of bytes remaining to be read in current input record.
[*n*]R	Control	No	Input and output	Changes the Radix for integer-formatted I/O.
S or SP	Control	No	Output	Print optional plus sign.
SS	Control	No	Output	Do not print optional plus sign.
T*c*	Control	No	Input and output	Move to column *c*.
TL*c*	Control	No	Input and output	Move *c* columns to the left.
TR*c* or *c*X	Control	No	Input and output	Move *c* columns to the right.
X[w]				
Z[*w*[*.m*]]	Data	Yes	Input and output	Convert integer data, using hexadecimal base.

The following sections describe the edit descriptors.

Character String ('...' or "...") Edit Descriptor

NOTE. *There is no single edit descriptor that defines a field for complex data. Instead, you must use two real edit descriptors--the first for the real part of the number, and the second for the imaginary part. The two edit descriptors may be different or the same, and you can insert control and character string edit descriptors between them.*

Likewise, there are no edit descriptors for formatting derived types and pointers. For derived types, you must specify the appropriate sequence of edit descriptors that match the data types of the derived type's components. For pointers, you must specify the edit descriptor that matches the type of the target object.

The character string edit descriptor is used to write a character constant to a formatted output record. It cannot be used to format input. You can use either apostrophes or quotation marks to delimit the constant. Whichever you use, they must be balanced. That is, if you begin with an apostrophe, you must also end with it. If the enclosed character constant includes a delimiting character, it must be of the other type; or you can escape the delimiter by giving another of the same type. The width of the field is the number of characters enclosed by the character string edit descriptors, including any blanks.

Table 9-2 gives examples of the character string edit descriptor on output. Note that b represents a blank.

Table 9-2 Character String Edit Descriptor: Output Examples

Descriptor	Field Width	Output
`'Enter data:'`	11	Enter data:
`"David's turn"`	12	David's turn
"bbbSpacesbbb"	12	~~bbb~~Spaces~~bbb~~
`'That''ll do.'`	11	That'll do.
`"""That'll do!"""`	13	"That'll do!"
`""""`	1	"
`'"'`	1	"

Newline ($) Edit Descriptor

The newline edit descriptor is an Intel Fortran extension that suppresses the generation of the newline character (that is, the carriage-return/linefeed sequence) during formatted, sequential output. By default, the cursor moves to a newline after each output statement. The newline edit descriptor causes the cursor to remain on the same line, immediately to the right of the last character output.

NOTE. *Nonadvancing I/O also suppresses the newline at the end of a record. Unlike the newline* `($)` *edit descriptor, it is a standard feature of Fortran 95, and can be used in input and output. For more information, see* Chapter 8, I/O and File Handling *and the* `ADVANCE=` *I/O specifier in the description of the* `OPEN` *statement in* Chapter 10, Intel Fortran Statements.

NOTE. *Also, as an extension in Intel Fortran, you can use the backslash (\) character as the new line edit descriptor.*

Slash (/) Edit Descriptor

The slash edit descriptor terminates the current record and begins processing a new record (such as a new line on a terminal). This edit descriptor has the same result for both input and output: it terminates the current record and begins a new one. For example, on output a newline character is printed, and on input a new line is read.

Keep in mind the following considerations when using the slash edit descriptor:

- If a series of two or more slashes are written at the beginning of a format specification, the number of records skipped is equal to the number of slashes.
- If *n* slashes appear other than at the beginning of a format specification (where *n* is greater than 1), processing of the current record terminates and *n* - 1 records are skipped.
- If a format contains only *n* slashes (and no other format specifiers), *n* + 1 records are skipped.

The / edit descriptor does not need to be separated from other descriptors by commas.

Colon (:) Edit Descriptor

The colon edit descriptor (:) is used when performing formatted I/O to terminate format control when the I/O list has been exhausted. If all items in an I/O list have been read or written, the colon edit descriptor stops any further format processing. If more items remain in the list, the colon edit descriptor has no effect.

Consider the following example:

```
WRITE (*, 40) 1, 2
WRITE (*, 50) 1, 2
40 FORMAT(3(' value =', I2))
50 FORMAT(3(:, ' value =', I2))
```

The first `WRITE` statement outputs the line:

```
value = 1 value = 2 value =
```

The descriptor 'value =' is repeated a third time because format control is not terminated until the descriptor I2 is reached and not satisfied.

The second WRITE statement outputs the line:

```
 value = 1 value = 2
```

This time, the colon descriptor terminates format control before the string ' value=' is output a third time.

A and R (character) Edit Descriptors

The A and R edit descriptors define fields for character data. The A edit descriptor specifies left-justification, and the R edit descriptor specifies right-justification.

The R edit descriptor is an Intel Fortran extension.

The syntax for the character edit descriptors is:

[*r*]A[*w*]

[*r*]R[*w*]

where

r	is a positive integer constant, specifying the repeat factor.
w	is the field width. If *w* is not specified, the default is the length in bytes of the corresponding I/O list item.

As a portability extension, the list item can be of any data type.

When the A and R edit descriptors are used for input and output, the results can differ according to whether the width (*w*) specified for the edit descriptor is less than, greater than, or equal to the length of the I/O list item. The results on input are summarized in Table 9-3; the results on output are summarized in Table 9-4.

Table 9-3 Contents of Character Data Fields on Input

Descriptor	Width/Length Relationship	Result
A	width < length	Data is left-justified in variable, followed by blanks.
	width >= length	Data is taken from rightmost characters in the field.
R	width < length	Data is right-justified in variable, preceded by nulls.
	width >= length	Data is taken from rightmost characters in the field.

Table 9-4 Contents of Character Data Fields on Output

Descriptor	Width/Length Relationship	Result
A	width <= length	Data is taken from leftmost characters in the field.
	width > length	Output the value, preceded by blanks.
R	width <= length	Data is taken from rightmost characters in the field.
	width > length	Output the value, preceded by blanks.

For examples of the use of character edit descriptors on input, see Table 9-5; for output examples, see Table 9-6. In the tables, ~~b~~ represents a blank and *z* represents a Null.

Table 9-5 A and R Edit Descriptors: Input Examples

Edit Descriptor	Input Field	Variable Length	Value Stored
A3	XYZ	3	XYZ
R3	XYZ	4	*z*XYZ
A5	ABC~~bb~~	10	ABC~~bbbbbbb~~
R9	RIGHTMOST	4	MOST
R8	CHAIR~~bbb~~	8	CHAIR~~bbb~~
R4	CHAIR	8	*zzzz*CHAI
A4	ABCD	2	CD

Table 9-6 A and R Edit Descriptors: Output Examples

Edit Descriptor	Internal Characters	Variable Length	Output
A6	ABCDEF	6	ABCDEF
R4	ABCDEFGH	8	EFGH
A4	ABCDE	5	ABCD
A8	STATUS	6	~~bb~~STATUS
R8	STATUS	6	~~bb~~STATUS
R8	STATUS	8	STATUS~~bb~~

B (binary) Edit Descriptor

The B edit descriptor defines a field for binary data. It provides for conversion between an external binary number and its internal representation.

The syntax for the binary edit descriptor is:

[*r*]B[*w*[.*m*]]

where

r is a positive integer constant, specifying the repeat factor.

w is a positive integer constant, specifying the field width.

m is an unsigned integer constant, specifying the minimum number of digits that must be in the field and forcing leading zeroes as necessary up to the first nonzero digit. The *m* value is ignored on input. If *m* is not specified, a default value of 1 is assumed. If *m* is larger than *w*, the field is filled with *w* asterisks.

On Input

Variables to receive binary input must be of type integer. The only legal characters are 0s and 1s. Nonleading blanks are ignored, unless the file is opened with `BLANK='ZERO'`.

If the file is opened with `BLANK='ZERO'`, nonleading blanks are treated as zeroes. (For more information about the `BLANK=` specifier, see the description of the `OPEN` statement in Chapter 10, Intel Fortran Statements.) Plus and minus signs, commas, or any other symbols are not permitted. If a nonbinary digit appears, an error occurs. The presence of too many digits for the integer variable (or I/O list item) is illegal.

Table 9-7 gives examples of the binary edit descriptor on input.

Table 9-7 B Edit Descriptor: Input Examples

Descriptor	Input field (Binary)	Value Stored (Binary)
B8	1111	1111
B8	01111	1111
B4	10101	1010
B8	1.1	error: illegal character

On Output

Unlike input, list items on output may be of any type, though character values are output only as the binary equivalent of their ASCII representation (without a length descriptor). If *w* is greater than the number of converted binary digits (excluding leading zeroes), the binary digits are right-justified in the output field.

If *w* is less than the number of converted binary digits, the field is filled with *w* asterisks. This primarily affects the output of negative values. Because negative values are output in twos complement form, their high-order bits are nonzero and cause the field to be filled with asterisks when *w* is less than the number of binary digits in the entire output value.

The field width required to fully represent the binary value of an item is eight times its size in bytes. For example, an `INTEGER*4` item could require a field *w* of up to 32 characters.

Only 1s and 0s are printed on output.

Table 9-8 gives examples of the binary edit descriptor on output.

Table 9-8 B Edit Descriptor: Output Examples

Descriptor	Internal Value	Output
B5	27	11011
B8	27	~~bbb~~11011
B8.6	27	~~bb~~011011
B8	-27	********

BN and BZ (blank) Edit Descriptors

The BN and BZ edit descriptors control the interpretation of embedded and trailing blanks in numeric input fields. The syntax of the blank edit descriptors is:

```
BN
BZ
```

At the beginning of the execution of an input statement, blank characters within numbers are ignored except when the unit is connected with `BLANK='ZERO'` specified in the `OPEN` statement. `BN` and `BZ` override the `BLANK=` I/O specifier for the current `READ` statement. For more details about the `BLANK=` I/O specifier, see the `OPEN` statement in Chapter 10, Intel Fortran Statements.

If a `BZ` edit descriptor is encountered in the format specification, trailing and embedded blanks in succeeding numeric fields are treated as zeroes. The `BZ` edit descriptor remains in effect until a `BN` edit descriptor or the end

of the format specification is encountered. If BN is specified, all embedded blanks are removed and the input number is right justified within the field width.

The BN and BZ edit descriptors affect only I, B, O, Q, F, D, E, EN, ES, G, and Z format descriptors during the execution of an input statement. The BN and BZ edit descriptors do not affect character and logical edit descriptors.

Table 9-9 gives examples of the BN and BZ edit descriptors on input.

Table 9-9 BN and BZ Edit Descriptors: Input Examples

Data Descriptor	Input Characters	BN Editing in Effect	BZ Editing in Effect
I4	1b2b	12	1020
F6.2	b4b.b2	4.2	40.02
E7.1	5b.bE1b	5.0×10^{1}	5.0×10^{11}
E5.0	3E4bb	3.0×10^{4}	3.0×10^{400} (overflow)

The BN and BZ edit descriptors are ignored during the execution of an output statement.

D, E, EN, ES, F, G, and Q (real) Edit Descriptors

The D, E, EN, ES, F, G, and Q edit descriptors define fields for real numbers. The I/O list item corresponding to a real descriptor must be a numeric type. (The Standard permits real and complex types only; as an extension, Intel Fortran allows integers.)

NOTE. *For the edit descriptors E, G, and D, you can use a comma as a delimiter to terminate an input field.*

The syntax for these edit descriptors is:

```
[r]D[w.d]
[r]E[w.d [{E|D|Q}e ]]
```

```
[r]EN[w.d [Ee ]]
[r]ES[w.d [Ee ]]
[r]F[w.d]
[r]G[w.d E[{|D|Q}e ]]
[r]Q[w.d]
```

where

r	is a positive integer constant, specifying the repeat factor.
w	is a positive integer constant, specifying the field width.
d	is a nonnegative integer constant, specifying the number of decimal places on output.
e	is a positive integer constant, specifying the number of digits in the exponent.

For formatting complex data, you can use two real edit descriptors—the first for the real part of the number and the second for the imaginary part. The two edit descriptors may be different or the same, and you can insert control and character string edit descriptors between them.

Real Edit Descriptors on Input

The input field for the real descriptors consists of an optional plus or minus sign followed by a string of digits that may contain a decimal point. If the decimal point is omitted in the input string, then the number of digits equal to *d* from the right of the string are interpreted to be to the right of the decimal point. If a decimal point appears in the input string and conflicts with the edit descriptor, the decimal point in the input string takes precedence. This basic form can be followed by an exponent in one of the following forms:

- A signed integer constant
- An E followed by an optionally signed integer constant
- A D followed by an optionally signed integer constant
- A Q followed by an optionally signed integer constant

All four exponent forms are processed in the same way. Note, however, that *e* has no effect on input.

The EN and ES edit descriptors are the same as the F edit descriptor on input. The Q edit descriptor (an Intel Fortran extension) is the same as the E edit descriptor on input.

Table 9-10 gives examples of the real edit descriptors on input. (The BZ edit descriptor listed in the "Descriptor" column treats nonleading blanks in numeric fields as zeroes.)

Table 9-10 D, E, F, and G Edit Descriptors: Input Examples

Descriptor	Input Field	Value Stored
F6.5	4.51E4	45100
G4.2	51-3	.00051
E8.3	7.1bEb5	710000
D9.4	~~b~~bb45E+35	.0045 x 10^{35}
BZ, F6.1	-54E3b	-5.4 x 10^{30}

Real Edit Descriptors on Output

The output field for the real descriptors consists of *w* character positions, filled with leading blanks (if necessary) and an optionally signed real constant with a decimal point, rounded to *d* digits after the decimal point. The following sections describe the real edit descriptors on output in detail.

D and E edit descriptors

The D and E edit descriptors define a normalized floating-point field for real and complex values. The value is rounded to *d* digits. The exponent part consists of *e* digits. If E*e* is omitted in a D or E edit descriptor, then the exponent occupies two or three positions, depending on its magnitude. The field width, *w*, should follow the general rule: *w* is greater than or equal to d+7. If E*e* is used, *w* is greater than or equal to *d*+*e*+5. This rule provides positions for a leading blank, the sign of the value, the decimal point, *d* digits, the exponent letter (D, E, or **Q**), the sign of the exponent, and the exponent. The E*e*, D*e*, and Q*e* specifications, which are available with the E edit descriptor, control which exponent letter is output.

Table 9-11 gives examples of the D and E edit descriptors on output.

Table 9-11 D and E Edit Descriptors: Output Examples

Descriptor	Internal value	Output
D10.3	+12.342	~~bb~~.123D+02
E10.3E3	-12.3454	-.123E+002
E12.4	+12.34	~~bbb~~.1234E+02
D12.4	-.00456532	~~bb~~-.4565D-02
D10.10	+99.99913	**********
E11.5	+999.997	~~b~~.10000E+04
E10.3E4	+.624 x 10^{-30}	.624E-0030

EN and ES edit descriptor

The EN and ES descriptors format floating-point values, using engineering and scientific notation, respectively. They are similar in form to the E descriptor, except:

- The field produced by the EN descriptor has an exponent that is divisible by 3 and a significand that is in the range 1 to 999.
- The field produced by the ES descriptor has one digit before the decimal point.

Table 9-12 gives examples of the EN and ES edit descriptors on output.

Table 9-12 EN and ES Edit Descriptors: Output Examples

Descriptor	Internal value	Output
EN12.3	+3.141	~~bbb~~3.141E+00
ES12.3	+3.141	~~bbb~~3.141E+00
EN12.3	+.00123	~~bbb~~1.230E-03
ES12.3	+.00123	~~bb~~b1.230E-03
EN12.3	-.7	-700.000E-03
ES12.3	-.7	~~bb~~-7.000E-01
EN12.3	+1234.5	~~bb~~b1.235E+03
ES12.3	+1234.5	~~bb~~b1.235E+03

F Edit Descriptor

The F edit descriptor defines a field for real and complex values. The value is rounded to *d* digits to the right of the decimal point. The field width, *w*, should be four greater than the expected length of the number to provide positions for a leading blank, the sign, the decimal point, and a roll-over digit for rounding if needed.

Table 9-13 gives examples of the F edit descriptor on output.

Table 9-13 F Edit Descriptor: Output Examples

Descriptor	Internal value	Output
F5.2	+10.567	10.57
F3.1	-254.2	***
F6.3	+5.66791432	b5.668
F8.2	+999.997	b1000.00
F8.2	-999.998	-1000.00
F7.2	-999.997	*******
F4.1	+23	23.0

G Edit Descriptor

The G edit descriptor can be used with any data type but is commonly used to define a field for real and complex values.

According to the magnitude of the data, the G edit descriptor is interpreted as either an E or F descriptor. (For more information on these edit descriptors, refer to “D and E edit descriptors” on page 17 and “F Edit Descriptor” on page 19.) The E edit descriptor is used when one of the following conditions is true:

- The magnitude is less than 0.1 but not zero.
- The magnitude is greater than or equal to `10**`*d* (after rounding to *d* digits).

If the magnitude does not fit either of these rules, the F edit descriptor is used. When F is used, trailing blanks are included in the field where the exponent would have been.

For fixed- or floating-point format descriptors, the field width is *w*. The value is rounded to *d* digits, and the exponent consists of *e* digits. If E*e* is omitted, the exponent occupies two positions. If *Ee* is omitted and the exponent is greater than 99 (that is, it requires three digits), the exponent letter is dropped from the output. The field width, *w*, should follow the general rule: *w* is greater than or equal to the sum of *d*+7; or, if E*e* is specified, *w* is greater than or equal to the sum of *d*+*e*+5. This rule provides positions for a leading blank, the sign of the value, *d* digits, the decimal point, and, if needed, the exponent letter (D, E, or Q), the sign of the exponent, and the exponent. Note that the E*e*, D*e*, and Q*e* specifications control which exponent letter is output.

When used to specify I/O fields for integer, character, and logical data, the G edit descriptor has the same syntax and same effect as the integer, character, and logical edit descriptors. The *d* and *e* values (if specified) have no effect.

Table 9-14 gives examples of the G edit descriptor on output.

Table 9-14 G Edit Descriptor: Output Examples

Edit Descriptor	Internal value	Interpreted as	Output
G10.3	+1234.0	E10.3	~~b~~0.123E+04
G10.3	-1234.0	E10.3	-0.123E+04
G12.4	+12345.0	E12.4	bb0.1235E+05
G12.4	+9999.0	F8.0, 4X	~~bb~~b9999.~~bbbb~~
G12.4	-999.0	F8.1, 4X	~~bb~~-999.0~~bbbb~~
G7.1	+.09	E7.1	0.9E-01
G5.1	-.09	E5.1	*****
G11.1	+9999.0	E11.1	~~bbb~~b0.1E+05
G8.2	+9999.0	E8.2	0.10E+05
G7.2	-999.0	E7.2	*******

Q Edit Descriptor

The Q edit descriptor (an Intel Fortran extension) has the same effect as the E edit descriptor on output, except that it outputs a Q for the exponent instead of an E.

The Q edit descriptor can also be used to determine the number of bytes remaining to be read in an input record; see page 9-28, "Q (bytes remaining) Edit Descriptor."

H (Hollerith) Edit Descriptor

The H edit descriptor outputs a specified number of characters. The syntax is:

nHcharacter-sequence

where

n is a positive integer that specifies the number of characters to output. This number must exactly match the actual number of characters in *character-sequence*.

character-sequence is the string of representable characters (including blanks) to output.

Table 9-15 gives examples of the Hollerith edit descriptor on output.

NOTE. *You should be careful if you split a Hollerith edit string across the Fortran source lines. By default, the compiler treats the end of line character as the end of the source line, even if it occurs before column 72 in fixed form source. If your Hollerith depends on treating the remaining characters up to column 72 as blanks, use the -Qpad_ source option. For details, see Intel Fortran Compiler User's Guide.*

Table 9-15 H Edit Descriptor: Output Examples

Edit Descriptor	Field Width	Output
12H␣␣␣Spaces␣␣␣	12	␣␣␣Spaces␣␣␣
14H"It␣isn't␣so."	14	"It␣isn't␣so."

I (integer) Edit Descriptor

The I edit descriptor defines a field for an integer number. As an Intel Fortran extension, it can also be used on real and logical data. The corresponding I/O list item must be a numeric or logical type.

NOTE. *For the* I *edit descriptor, you can use a comma as a delimiter to terminate an input field.*

The syntax of the integer edit descriptor is:

[*r*I][w[.*m*]]

where

r is a positive integer constant, specifying the repeat factor.

w is a positive integer constant, specifying the field width.

m is a nonnegative integer constant, specifying the minimum number of digits that must be in the field and forcing leading zeroes as necessary up to the first nonzero digit. The *m* value is ignored on input. If *m* is not specified, a default value of 1 is assumed. If *m* is larger than *w*, the field is filled with *w* asterisks. If *m* = 0 and the list item is zero, only blanks are output.

On Input

The integer edit descriptor causes the interpretation of the next *w* positions of the input record. The number is converted to match the type of the list item currently using the descriptor. A plus sign is optional for positive values. A decimal point must not appear in the field.

Table 9-16 gives examples of the integer edit descriptor on input.

Table 9-16 I Edit Descriptor: Input Examples

Descriptor	Input field	Value Stored
I4	~~b~~1~~bb~~	1
I5	bbbbb	0
I5	~~bbbbb~~1	0
I2	-1	-1
I4	-123	-123
I3	~~b~~12	12
I3	12~~b~~	12
I3	12~~b~~	120
I3	1.1	error: illegal character

On Output

The integer edit descriptor outputs a numeric variable as a right-justified integer value (truncated, if necessary). The field width, `w`, should be one greater than the expected number of digits to allow a position for a minus sign for negative values. If *m* is set to 0, a zero value is output as all blanks.

Table 9-17 gives examples of the integer edit descriptor on output.

Table 9-17 I Edit Descriptor: Output Examples

Descriptor	Internal Value	Output
I4	+452.25	~~b~~452
I2	+6234	**
I3	-11.92	-11
I5	-52	~~bb~~-52
I10	123456.5	~~bbbb~~123456
I6.3	3	~~bbb~~003
I3.0	0	bbb
I3	0	~~bb~~0

L (logical) Edit Descriptor

The L edit descriptor defines a field for logical data. Its syntax is:

[*r*]L[*w*]

where

r	is a positive integer constant, specifying the repeat factor.
w	is a positive integer constant, specifying the field width.

NOTE. *For the* L *edit descriptor, you can use a comma as a delimiter to terminate an input field.*

The I/O list item corresponding to an L edit descriptor must be of type logical, short logical, or byte.

On Input

The field width is scanned for optional blanks followed by an optional decimal point, followed by T (or t) for true or F (or f) for false. The first nonblank character in the input field (excluding the optional decimal point) determines the value to be stored in the declared logical variable. It is an error if the first nonblank character is not T, t, F, f, or a period(.). Table 9-18 gives examples of the logical edit descriptor on input.

Table 9-18 L Edit Descriptor: Input Examples

Edit Descriptor	Input Field	Value Stored
L1	T	.TRUE.
L1	f	.FALSE.
L6	.TRUE.	.TRUE.
L7	.false.	.FALSE.
L2	.t	.TRUE.
L8	~~bbbb~~TRUE	.TRUE.
L3	ABC	error: illegal character

On Output

The character T or F is right-justified in the output field, depending on whether the value of the list item is true or false. Table 9-19 gives examples of the logical edit descriptor on output.

Table 9-19 L Edit Descriptor: Output Examples

Descriptor	Internal value	Output (logical)
L5	false	~~bbbb~~F
L4	true	~~bbb~~T
L1	true	T

O (octal) Edit Descriptor

The O edit descriptor defines a field for octal data. It provides conversion between an external octal number and its internal representation.

The syntax for the octal edit descriptor is:

[*r*]O[*w*[*.m*]]

where

r is a positive integer constant, specifying the repeat factor.

w is a positive integer constant, specifying the field width.

m is a nonnegative integer constant, specifying the minimum number of digits that must be in the field and forcing leading zeroes as necessary up to the first nonzero digit. The *m* value is ignored on input. If *m* is not specified, a default value of 1 is assumed. If *m* is larger than *w*, the field is filled with *w* asterisks.

On Input

The presence of too many digits for the integer variable (or list item) to receive produces undefined results. Legal octal digits are 0 through 7. Plus and minus signs are illegal.

Table 9-20 gives examples of the octal edit descriptors on input.

Table 9-20 O Edit Descriptor: Input Examples

Descriptor	Input Field (Octal)	Value Stored (Octal)
O8	12345670	12345670
O2	77	77
O3	064	64
O8	45r	error: illegal character

On Output

List items may be of any type, though character variables are output only as the octal equivalent of their ASCII representation (no length descriptor).

If *w* is greater than the number of converted octal digits (including blanks between words but excluding leading zeroes), the octal digits are right-justified in the output field. If *w* is less than the number of converted octal digits, the field is filled with asterisks. This primarily affects the output of negative values. Because negative values are output in twos complement form, their high-order bits are nonzero and cause the field to be filled with asterisks when *w* is less than the number of octal digits in the entire output value. If *m* is set to 0, a zero value is output as all blanks.

Table 9-21 gives examples of the octal edit descriptors on output.

Table 9-21 O Edit Descriptor: Output Examples

Descriptor	Internal Value	Output (Octal)
O6	80	~~bb~~b120
O2	80	**
O14	-9	~~bb~~b37777777767
O11	32767	~~bbbbbb~~77777
O6.4	79	~~bb~~0117
O12	1.1	~~b~~b7743146315

continued

Table 9-21 O Edit Descriptor: Output Examples (continued)

Descriptor	Internal Value	Output (Octal)
O12	'A'	~~b~~101
O12	'ABC'	b101b102b103

P (scale factor) Edit Descriptor

The *k*P edit descriptor causes a scale factor of *k* to be applied to all subsequent F, D, E, *EN*, ES, and G edit descriptors in the format specification.

If the P edit descriptor does not precede an F, D, E, EN, ES, or G edit descriptor, it should be separated from other edit descriptors by a comma. If the P edit descriptor immediately precedes an F, D, E, EN, ES, or G edit descriptor, the comma is optional. For example, the format specification:

```
(3P, I2, F4.1, E5.2)
```

is equivalent to

```
(I2, 3PF4.1, E5.2)
```

When a format specification is interpreted, the scale factor is initially set to 0. When a P edit descriptor is encountered, the specified scale factor takes effect for the format specification and remains in effect until another P edit descriptor is encountered.

The effect of the scale factor differs for input and output as follows:

On Input

If the value in the input field does not have an exponent, the internal number is equal to the field value multiplied by 10^{-k}. If the value in the input field has an exponent, the scale factor has no effect. See Table 9-22 for examples of the scale factor on input.

On Output

The scale factor has no effect on the EN, ES, F and G (interpreted as F) edit descriptors. For the D, E, and G (interpreted as E) edit descriptors, the value of the list item is multiplied by 10^{k} as it is output but the exponent part is decreased by *k*.

The value specified for the scale factor (*k*) must be in the range:

-d < k < d+2

where

d is the number of digits in the fractional part of the number being written.

k is a signed integer that specifies the scale factor.

See Table 9-22 for examples of the scale factor on output.

Table 9-22 P Edit Descriptor: Input and Output Examples

Format Specification	Input Field	Internal Value	Output
(-2PG15.5)	1.97E-4	1.97×10^{-4}	~~bbbbb~~.00197E-01
(2P, F15.5)	27.982	.2798199	~~bbbbbbb~~27.98200
(2P,ES15.5)	3518.	35.18	~~bbb~~3.51800E+01
(-2P,EN15.5)	7.91E+5	7.91×10^{5}	~~bb~~791.00000E+03
(-2PE15.5)	.17694	17.694	~~bbbbb~~.00177E+04

When part or all of a format specification is repeated, the current scale factor is not changed until another scale factor is encountered.

Q (bytes remaining) Edit Descriptor

The Q edit descriptor is an Intel Fortran extension that returns the number of bytes remaining to be read in the input record, placing the result into the corresponding integer variable in the I/O list. The return value can be used to control the remaining input items.

The Q edit descriptor is valid on input only; it is ignored on output. It can be used for reading formatted, sequential, and direct-access files. The following program segment reads variable-length strings from a sequential file:

```
CHARACTER(LEN=80) :: string
INTEGER :: n, i
...
READ (11,'(Q,80A1)') n, (string (i:i), i=1, n)
```

For information about the Q*w.d* edit descriptor for editing real data, see page 9-15, "D, E, EN, ES, F, G, and Q (real) Edit Descriptors."

S, SP, and SS (plus sign) Edit Descriptors

The S, SP, and SS edit descriptors control printing of the plus sign character in numeric output. The default behavior of Intel Fortran is not to print the plus sign. However, an SP edit descriptor in the format specification causes the plus sign to appear in any subsequent numeric output where the value is positive. The SS descriptor suppresses the plus sign in subsequent numeric output. The S edit descriptor restores the default behavior.

The sign edit descriptors have no effect on input.

T, TL, TR, and X (tab) Edit Descriptors

The tab edit descriptors position the cursor on the input or output record. Their syntax is:

T*n*

TL*n*

TR*n*

*n*X

where

n is a positive integer constant, specifying the number of column positions to skip for positioning within the current output or input record.

The T edit descriptor references an absolute column number, while the descriptors TL and TR reference a relative number of column positions to the left (TL) or right (TR) of the current cursor position. Note that the TR descriptor is identical to the X edit descriptor.

Z (hexadecimal) Edit Descriptor

The Z edit descriptor defines a field for hexadecimal data. This descriptor provides for conversion between an external hexadecimal number and its internal representation.

NOTE. *For the Z edit descriptor, you can use a comma as a delimiter to terminate an input field*.

The syntax for the hexadecimal edit descriptor is:

[*r*]Z[*w* [.*m*]]

where

r	is a positive integer constant, specifying the repeat factor.
w	is a positive integer constant, specifying the field width.
m	is a nonnegative integer constant, specifying the minimum number of digits that must be in the field and forcing leading zeroes as necessary up to the first nonzero digit. The *m* value is ignored on input. If *m* is not specified, a default value of 1 is assumed. If *m* is larger than *w*, the field is filled with *w* asterisks.

On Input

Variables to receive hexadecimal input must be of type integer. Legal hexadecimal digits are 0 through 9, and A through F (or a through f). Nonleading blanks are ignored, unless the file is opened with BLANK='ZERO'. If the file is opened with BLANK='ZERO', nonleading blanks are treated as zeroes. (For more information about the BLANK=

specifier of the OPEN statement, see Chapter 10, Intel Fortran Statements.) Plus and minus signs, commas, or any other symbols are neither permitted on input nor printed on output. The presence of too many digits for the integer variable (or list item) produces undefined results.

Table 9-23 gives examples of the hexadecimal edit descriptor on input.

Table 9-23 Z Edit Descriptor: Input Examples

Descriptor	Input Field (Hexadecimal)	Value Stored (Hexadecimal)
Z4	FF3B	FF3B
Z4	fFfF	FFFF
Z2	ABCD	AB
Z3	1.1	error: illegal character

On Output

List items may be of any type, though character variables are output only as the hexadecimal equivalent of their ASCII representation (without a length descriptor). If *w* is greater than the number of converted hexadecimal digits (excluding leading zeroes), the hexadecimal digits are right-justified in the output field. If *w* is less than the number of converted hexadecimal digits, the field is filled with asterisks. This primarily affects the output of negative values. Because negative values are output in twos complement form, their high-order bits are nonzero and cause the field to be filled with asterisks when *w* is less than the number of hexadecimal digits in the entire output value. If *m* is set to 0, a zero value is output as all blanks.

The field width required to fully represent the hexadecimal value of an item is twice its size in bytes. For example, a CHARACTER*12 item would require a field width of 24 characters.

Table 9-24 gives examples of the hexadecimal edit descriptor on output.

Table 9-24 Z Edit Descriptor: Output Examples

Descriptor	Internal value	Output
Z2	27	1B
Z6.4	27	~~bb~~001B
Z	'A'	~~b~~41
Z8	'ABCD'	41424344
Z8	1.1	3F8CCCCD

Embedded Format Specification

A format specification can be embedded in a data transfer statement as a character expression. Parentheses are included in the expression, and the first nonblank character must be a left parenthesis. The matching right parenthesis must also be in the expression. A list of edit descriptors appears between the parentheses. Any characters appearing after the matching right parenthesis are ignored.

If the character expression is a character constant, it must be delimited by either apostrophes or quotation marks. If the character constant contains another character constant, the nested character constant must also be delimited. If the inner set of delimiters is the same as the outer set they must be doubled. Each of the following statements is correct and will produce the same results:

```
PRINT "('i = ', i2)", i
PRINT "(""i = "", i2)", i
PRINT '("i = ", i2)', i
PRINT '(''i = '', i2)', i
WRITE (6, "('i = ', i2)") i
```

If the character expression is an array element, the entire specification must be within that element. If the expression is a whole character array, the format specification is the concatenation of the array elements in array element order. (As an extension, Intel Fortran allows the use of an integer array to contain a format specification.)

The following illustrates the use of a character array to hold the format specification:

```
CHARACTER(LEN=6), DIMENSION(2) :: fspec
fspec(1) = '(F8.3,'
fspec(2) = ' I5)'
PRINT fspec, fvar, ivar
```

If the value of `fvar` is 12.34567 and `ivar` is 123, the output would be:

```
bb12.346bb123
```

Nested Format Specifications

A format specification can include a nested format specification (another set of edit descriptors, enclosed in parentheses). You can also precede the nested format specification with a repeat factor, as in the following example:

(1H , 2(I5, F10.5))

This is equivalent to:

```
(1H, I5, F10.5, I5, F10.5)
```

Each nested specification is known as a group at nested level *n*. The value of *n* begins at 1. For each successive level of nesting, *n* is incremented by 1. Each group at nested level 1 can contain one or more groups at nested level 2, and so on.

For example:

```
(E9.3,I6,(2X,I4))
```

contains one group at nested level 1.

```
(L2,A3/(E10.3,4(A2,L4)))
```

has one group at nested level 1 and one at nested level 2.

```
(A,(3X,(I2,(A3)),I3),A)
```

contains one group at nested level 1, one at level 2, and one at level 3.

A nested format specification can be preceded by a repeat specification. For example, the following input record:

```
b26b6.4336b373.86b39bb49.79bb4bbb4395.4972
```

could be accessed with the following FORMAT statement:

```
10 FORMAT (I3,F7.4,2(F7.2,I3),F12.4)
```

The list of variables following READ statement corresponds to the preceding FORMAT statement:

```
READ 10,i,a,b,j,d,k,f
```

The READ statement would read values for i and a; repeat the nested format specification F7.2,I3 twice to read values for b, j, d, and k; and, finally, read a value for f.

Interaction Between Format Specification and I/O Data List

A formatted I/O statement references each item in an I/O list, and the corresponding format specification is scanned to find a format descriptor for each item. As long as an item is matched to an edit descriptor, normal execution continues.

NOTE. *Default values are provided for the w, d, and e fields regardless of which edit descriptors you select.*

If there are more edit descriptors than list items, format control terminates with the last list item. If there are fewer edit descriptors than list items, the following three steps are performed:

1. The current record is terminated.
2. A new record is started.

3. Format control is returned to the format specification based upon the following hierarchy:
   a. Control returns to the repeat specification for the rightmost group at nested level 1. (For information about nested levels, see “Nested Format Specifications” on page 33.)
   b. If no repeat specification exists in the rightmost group at nested level 1, control returns to the group itself.
   c. If there is no group at nested level 1, control returns to the first descriptor in the format specification.

Table 9-25 provides examples showing how control is returned to the format specification in different circumstances.

Table 9-25 Format Control and Nested Format Specifications

Format Specification	Control Returns to:	Explanation
(I5,2(3X,I2,(I4)))	2(3X,I2,(I4))	The rightmost group at nested level 1 is 3X,I2,(I4). Control returns to the repeat specifier for this group.
(F4.1,I2)	(F4.1,I2)	There is no group at nested level 1. Control returns to the first descriptor in the format specification.
(A3,(3X,I2),4X,I4)	(3X,I2),4X,I4	Control returns to the group at nested level 1.

Intel Fortran Statements 10

This chapter describes the Intel Fortran statements and attributes, arranged in alphabetical order and providing syntactic descriptions, applicable rules, and examples. This chapter does not describe assignment statements (see Chapter 4, Arrays) or statement functions (see Chapter 7, Program Units and Procedures). For general information about type declaration statements, see Chapter 3, Data Types and Data Objects. For information about any of the following specific type declaration statements, see this chapter:

- BYTE
- CHARACTER
- COMPLEX
- DOUBLE COMPLEX
- DOUBLE PRECISION
- INTEGER
- LOGICAL
- REAL
- RECORD
- TYPE(*type-name*)

This chapter describes statements and attributes only, not constructs. For example, for information about the CASE statement, look here; for information about the CASE construct, see Chapter 6, Execution Control.

Attributes

Table 10-1 lists all the attributes that an Intel Fortran entity may possess and indicates their compatibility. If the box at the intersection of two attributes contains a check mark, then the attributes are mutually compatible and can be held simultaneously by a Fortran 95 entity. The attributes are referred to throughout this chapter as well as in the rest of the book.

Table 10-1 Attribute Compatibility (Y=YES)

	ALLOCATABLE	AUTOMATIC	DIMENSION	EXTERNAL	**Initialization**	INTENT	INTRINSIC	OPTIONAL	PARAMETER	POINTER	PRIVATE	PUBLIC	SAVE	STATIC	TARGET	VOLATILE
ALLOCATABLE	Y	Y	3								Y	Y	Y		Y	Y
AUTOMATIC	Y	Y	Y							Y					Y	Y
DIMENSION	Y	Y	Y		Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y
EXTERNAL				Y				Y			Y	Y				
INITIALIZATION			Y		Y				Y		Y	Y	Y	Y	Y	Y
INTENT			Y			Y		Y							Y	Y
INTRINSIC							Y				Y	Y				
OPTIONAL			Y	Y		Y		Y		Y					Y	Y
PARAMETER			Y		Y				Y		Y	Y				
POINTER		Y	Y					Y		Y	Y	Y	Y	Y		Y
PRIVATE	Y		Y	Y	Y		Y		Y	Y	Y		Y	Y	Y	Y
PUBLIC	Y		Y	Y	Y		Y		Y	Y		Y	Y	Y	Y	Y
SAVE	Y		Y		Y					Y	Y	Y	Y	Y	Y	Y
STATIC			Y		Y					Y	Y	Y	Y	Y	Y	Y

continued

Table 10-1 Attribute Compatibility (Y=YES) (continued)

	ALLOCATABLE	AUTOMATIC	DIMENSION	EXTERNAL	Initialization	INTENT	INTRINSIC	OPTIONAL	PARAMETER	POINTER	PRIVATE	PUBLIC	SAVE	STATIC	TARGET	VOLATILE
TARGET	Y	Y	Y		Y	Y		Y			Y	Y	Y	Y	Y	Y
VOLATILE	Y	Y	Y		Y	Y		Y		Y	Y	Y	Y	Y	Y	Y

NOTE. AUTOMATIC, STATIC, *and* VOLATILE *may be specified in a statement of the same name but not as attributes in a type declaration statement*

Statements and Attributes

The remainder of this chapter describes all of the statements and attributes that you can use in an Intel Fortran program. The statement and attribute descriptions are listed in alphabetical order. Not described here are the statement function (see Chapter 7, Program Units and Procedures) and the general form of a type declaration statement (see Chapter 3, Data Types and Data Objects). For general information about statements and attributes (including the order in which statements are required to appear in a legal program), refer to Chapter 2, Language Elements.

ACCEPT

Reads from standard input.

The syntax of the ACCEPT statement can take one of two forms:

- Formatted and list-directed syntax:
  ACCEPT *format* [, *input-list*]

- Namelist-directed syntax:
  ACCEPT *name*

format is one of the following:

- An asterisk (*), specifying list-directed I/O. For detailed information about list-directed I/O, see Chapter 8, I/O and File Handling.
- The label of a FORMAT statement containing the format specification.
- An integer variable that has been assigned the label of a FORMAT statement.
- An embedded format specification. For information about the format specifications, see Chapter 9, I/O Formatting.

input-list is a comma-separated list of data items. The data items can include variables and implied-DO lists; see Chapter 8, I/O and File Handling for more detailed information.

name is the name of a namelist group, as previously defined by a NAMELIST statement. Using this syntax, the ACCEPT statement accepts data from standard input and transfers it to the namelist group. To perform namelist-directed I/O with a connected file, you must use the READ statement and include the NML= specifier.

Description

The ACCEPT statement is an Intel Fortran extension and is provided for compatibility with other versions of Fortran. The standard READ statement performs the same function, and standard-conforming programs should use it.

The ACCEPT statement transfers data from standard input to internal storage. (Unit 5 is preconnected to the Intel standard input.) The ACCEPT statement can be used to perform formatted, list-directed, and namelist-directed I/O only.

To read data from a connected file, use the READ statement.

Examples

The following example of the ACCEPT statement reads an integer and a floating-point value from standard input, using list-directed formatting:

```
INTEGER :: i
REAL :: x
ACCEPT *, i, x
```

Related Statements

`FORMAT`, `NAMELIST`, `PRINT` and `READ`

Related Concepts

For information about I/O concepts, see Chapter 8, I/O and File Handling, which also presents example programs performing I/O. For information about I/O formatting, see Chapter 9, I/O Formatting.

ALLOCATABLE (Statement and Attribute)

Declares an allocatable array with deferred shape.

The syntax of a type declaration statement with the `ALLOCATABLE` attribute is:

`type, attrib-list :: entity-list`

`type` is a valid type specification (`INTEGER`, `REAL`, `LOGICAL`, `CHARACTER`, `TYPE (`type-name`)`, and so on), as described in Chapter 3, Data Types and Data Objects.

`attrib-list` is a comma-separated list of attributes including `ALLOCATABLE` and optionally those attributes compatible with it, namely:

`DIMENSION`	`PUBLIC`	`TARGET`
`PRIVATE`	`SAVE`	

entity-list is a comma-separated list of entities. Each entity is of the form:

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

If (*deferred-shape-spec-list*) is omitted, it must be specified in another declaration statement.

array-name is the name of an array being given the attribute ALLOCATABLE.

deferred-shape-spec-list is a comma-separated list of colons, each colon representing one dimension. Thus the rank of the array is equal to the number of colons specified.

The syntax of the ALLOCATABLE statement is:

```
ALLOCATABLE [::] array-name
[(deferred-shape-spec-list)]
   [,array-name [(deferred-shape-spec-list)]]...
```

If (*deferred-shape-spec-list*) is omitted from the ALLOCATABLE statement, it must be specified in another declaration statement, such as a type or DIMENSION statement.

The ALLOCATED intrinsic inquiry function can be used to determine whether an allocatable array is currently allocated.

Description

The ALLOCATABLE attribute or statement is used to declare an array whose extents in all its dimensions will be specified when an ALLOCATE statement is executed at run-time; for this reason it is known as "deferred-shape". When an allocatable array is declared, only its name and rank are given.

Examples

The following statements declare a rank-one deferred-shape array and illustrate its use with different extents.

```
! mls is deferred shape.
INTEGER, ALLOCATABLE :: mls(:)
ALLOCATE (mls (3))     ! Allocate 3 elements.
```

```
DEALLOCATE (mls)        ! mls is no longer
                        ! allocated.
ALLOCATE (mls (-n:n))   ! Allocate with
                        ! different extent.
```

Related Statements

ALLOCATE and DEALLOCATE

Related Concepts

See Chapter 4, Arrays for a full description of ALLOCATABLE arrays and the conditions applying to their use.

Array pointers provide a more general mechanism for the manipulation of deferred-shape arrays; see Chapter 4, Arrays.

ALLOCATE

Provides storage space for allocatable arrays and pointer targets.

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

allocation-list	is a comma-separated list of allocation.
allocation	is allocate-object [(allocate-shape-spec-list)].
allocate-object	is *variable-name* or *structure-component*. Each *allocate-object* must be an allocatable array or a pointer.
allocate-shape-spec-list	is a comma-separated list of allocate-shape-spec.

allocate-shape-spec
: is *[lower-bound :] upper-bound*. The bounds in an allocate-shape-spec must be scalar integer expressions.

STAT=*scalar-integer-variable*

returns the error status after the statement executes. If given, it is set to a positive value if an error is detected, and to zero otherwise. If there is no status variable, the occurrence of an error causes the program to terminate.

Description

The ALLOCATE statement creates space for allocatable arrays and targets for variables (scalars or arrays) with the POINTER attribute. The ALLOCATE and DEALLOCATE statements give the user the ability to manage space dynamically at execution time.

For allocatable arrays, an error occurs when an attempt is made 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.

A pointer can be associated with a target, either with the pointer assignment statement or by use of the ALLOCATE statement. It is not an error to allocate an already associated pointer; its old target connection is replaced by a connection to the newly allocated space. However, if the previous target was allocated and no other pointer became associated with it, the space is no longer accessible.

Examples

In the following example, a complex array with the POINTER attribute is declared. Target space is allocated to it at run-time, the amount being determined by two integer values read in. Later in the program, the space is recovered by use of the DEALLOCATE statement.

```
COMPLEX, POINTER :: hermitian (:, :)
READ *, m, n
ALLOCATE (hermitian (m, n))
DEALLOCATE (hermitian, STAT = ierr)
```

```
In the next example, a real allocatable array is
declared. The amount of space allocated to it depends
on how much is available.
REAL, ALLOCATABLE :: intense(:,:)
! Rank-2 allocatable array
CALL init_i_j(i, j)
DO
  ALLOCATE (intense(i, j), STAT = ierr4)
  ! ierr4 will be positive if there is not
  !  enough space to allocate this array.
  IF (ierr4 == 0) EXIT
  i = i/2; j = j/2
END DO
```

The derived type `node` in the next example is the basis of a binary tree structure. It consists of a real value component (`val`) and two pointer components, `left` and `right`, both of type `node`. The variable `top` (of type `node`) is declared, and space is allocated for targets for the pointers `top%left` and `top%right`.

The `ALLOCATE` and `DEALLOCATE` statements and pointer variables of type `NODE` make it possible to allocate space for nodes in such a tree structure, traverse it as required, and then recover the space when it is no longer needed.

```
TYPE node
  REAL val
  TYPE(node), POINTER :: left, right
  ! Pointer components.
END TYPE node
TYPE(node) top
ALLOCATE (top % left, top % right)
```

In the final example, two `CHARACTER` arrays, `para` and `key`, are declared with the `POINTER` attribute. `para` is allocated space; `key` is made to point at a section of `para`.

```
CHARACTER, POINTER :: para(:), key(:)
! Pointers to char arrays.
CALL init_k_m(k, m)
```

```
ALLOCATE (para(1000) )
key => para (k : k + m)
```

Related Statements

ALLOCATABLE (statement and attribute), DEALLOCATE, NULLIFY, and POINTER (statement and attribute)

Related Concepts

The intrinsic inquiry functions ALLOCATED and ASSOCIATED are described in the *Intel Fortran Compiler User's Guide*. See Chapter 3, Data Types and Data Objects for information about pointers.

ASSIGN

Assigns statement label to integer variable.

ASSIGN *stmt-label* TO *integer-variable*

stmt-label	is the statement label for an executable statement or a FORMAT statement in the same scoping unit as the ASSIGN statement.
integer-variable	is a scalar variable of the default integer type. It cannot be a field of a derived type or record, or an array element.

Description

Once a variable is defined by an ASSIGN statement, it can be used in an assigned GO TO statement or as a format specifier in an input/output statement. It should not be used in any other way.

A variable that has been assigned a statement label can be reassigned another label or an integer value. If *integer-variable* is subsequently assigned an integer value, it no longer refers to a label.

Examples

```
  ASSIGN 20 TO LAST1
  GO TO LAST1
  ! ASSIGN used with FORMAT statement
  ASSIGN 10 TO FORM1
10 FORMAT(F6.1,2X,I5/F6.1
  READ(5,FORM1)SUM,K1,AVE1
20 ...
```

Related Statements

`GO TO`, `READ`, and `WRITE`

Related Concepts

Statement labels are described in Chapter 2, Language Elements. The assigned `GO TO` statement is described later in this chapter as well as in Chapter 6, Execution Control.

AT

Identifies the beginning of a debug packet and indicates the point in the program where debugging is to begin.

`AT` *`stmt-label`*

`stmt-label`	is the number of an executable statement in the program at which debugging is to begin.

Description

The Intel Fortran compiler permits execution of debugging operations specified within a debug packet before the execution of the statement indicated by the *`stmt-label`*. However, you must adhere to the following guidelines:

- You cannot specify the *`stmt-label`* in another debug packet.

- You must have an AT statement for each debug packet, but you can have many debug packets for one program or subprogram.
- If you do not specify a stmt-label, the AT statement identifies the beginning of a debug packet and the end of a preceding packet if any exists. However, if the AT alone statement identifies the last debug packet, then you must specify the end of the debug packet with an END DEBUG statement.

AUTOMATIC

Makes procedure variables and arrays automatic.

AUTOMATIC *var-name-list*

var-name-list is a comma-separated list of names of variables and arrays to be declared as automatic. Array names may be followed by an optional explicit-shape-spec.

Description

The AUTOMATIC statement is provided as an Intel Fortran extension.

If a variable or array declared within a procedure is declared as automatic, then there is one copy of it for each invocation of the procedure. Space is allocated on entry to the procedure and deallocated on exit. This is also the default for variables that do not have the SAVE or STATIC attribute, unless the /Qsave option has been specified (see the *Intel Fortran Compiler User's Guide* for information about this option).

If it is required to have the same copy of a variable available to each invocation of the routine (for example, to keep a record of the depth of recursion), then the variable should have the SAVE attribute.

Note the following:

- The AUTOMATIC statement may only be used within a procedure.
- Local variables are AUTOMATIC by default.
- Arguments and function values are AUTOMATIC.
- Automatic variables may not appear in EQUIVALENCE, DATA or SAVE statements.

- The AUTOMATIC attribute is not the same as automatic arrays and automatic character strings.

Example

```
AUTOMATIC r, s, u, v, w(10)
```

Related Statements

SAVE and STATIC

Related Concepts

Automatic and static variables are described in Chapter 3, Data Types and Data Objects.

BACKSPACE

Positions file at preceding record.

The syntax of the BACKSPACE statement can take one of two forms:

- Short form:
  BACKSPACE *integer-expression*
- Long form:
  BACKSPACE *(io-specifier-list)*

integer-expression is the number of the unit connected to a sequential file.

io-specifier-list is a list of the following comma-separated I/O specifiers:

[UNIT=] *unit* specifies the unit connected to an external file opened for sequential access. *unit* must be an integer expression that evaluates to a number greater than 0. If the optional keyword UNIT= is omitted, *unit* must be the first item in *io-specifier-list*.

ERR=*stmt-label*	specifies the label of an executable statement to which control passes if an error occurs during statement execution.
IOSTAT=*integer-variable*	returns the I/O status after the statement executes. If the statement executes successfully, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.

Description

The BACKSPACE statement causes the external file connected to *unit* to be positioned just before the preceding record of the file. The file must be connected for sequential access.

Examples

The following statement causes the file connected to unit 10 to be positioned just before the preceding record:

```
BACKSPACE 10
```

The following statement causes the file connected to unit 17 to be positioned just before the preceding record. If an error occurs during the execution of the statement, control passes to the statement at label 99, and the error code is returned in ios:

```
BACKSPACE (17, ERR=99, IOSTAT=ios)
```

Related Statements

ENDFILE, OPEN, and REWIND

Related Concepts

For information about I/O concepts, see Chapter 8, I/O and File Handling, which also gives example programs that perform I/O. For information about I/O formatting, see Chapter 9, I/O Formatting.

BLOCK DATA

Introduces a BLOCK DATA program unit.

`BLOCK DATA` *`[block-data-name]`*

`block-data-name` is an optional name. If a name is given in the `END BLOCK DATA` statement terminating a block data program unit, it must be the same as the block-data-name given in the `BLOCK DATA` statement introducing the program unit.

Description

A block data program unit is used to give initial values to variables in a named common blocks by means of `DATA` statements and must start with a `BLOCK DATA` statement. The block data program unit is an obsolescent feature of Fortran 95 and is effectively superseded by the module facility (described in Chapter 7, Program Units and Procedures).

As an extension, Intel Fortran allows unnamed common blocks to be initialized.

Examples

The following block data program unit gives initial values to some variables in the common blocks `cb1` and `cb2`. All variables in each common block are specified completely.

```
BLOCK DATA
 REAL b(4) DOUBLE PRECISION z(3)
 COMPLEX c
 COMMON /cb1/c,a,b /cb2/z,y
 DATA b, z, c /1.0, 1.2 ,2*1.3, &
    3*7.654321D0, (2.4,3.76)/
END
```

Related Statements

COMMON, DATA, and END

Related Concepts

The initialization of variables is discussed in Chapter 3, Data Types and Data Objects.

BYTE

Declares entities of type integer.

BYTE [[, *attrib-list*] ::] *entity-list*

attrib-list is a comma-separated list of one or more of the following attributes:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

For information about each of the attributes, see the corresponding statement in this chapter.

entity-list is a list of entities, separated by commas. Each entity takes the form:

name [(array-spec)] [= initialization-expr]

where name is the name of a variable or function

array-spec is a comma-separated list of dimension bounds

initialization-expr is the initial value for the entity.

Description

The BYTE statement is an Intel Fortran extension that is used to declare the properties of entities. The entities can take values that are whole numbers and can be represented in one byte. It is equivalent to the INTEGER(KIND=1) statement. Note that the BYTE statement does not have a KIND parameter.

The BYTE statement is constrained by the rules for all type declaration statements, including the requirement that it precede all executable statements.

Explicitly declaring an entity with the BYTE statement overrides any implicit typing rules in effect.

An array specification included with an entity in *entity-list* overrides any specification made with the DIMENSION attribute.

If *attrib-list* or *initialization-expr* appear in the declaration, *entity-list* must be preceded by the double colon.

Initialization

initialization-expr must be a constant integer expression that can be evaluated at compile time.

The following entities may not be initialized:

- Dummy arguments
- Function results
- Allocatable arrays
- Pointers
- External names
- Intrinsic names
- Automatic objects

If *attrib-list* includes the PARAMETER attribute, every entity in *entity-list* must be accompanied by an initialization expression.

Initializing an entity implies the SAVE attribute.

To initialize an array in a BYTE statement, you may use an array constructor, as in the following example:

```
BYTE, DIMENSION(4) :: bvec=(/1,2,3,4/)
```

When initializing an array, all items in the array must be initialized. Implied-DO loops cannot be used to initialize an array in a type declaration statement.

As an extension, an initializer may appear between slashes in a type declaration statement, as in the following example:

```
BYTE b/12/, bb/27/
```

The double colon (`::`) may not be used with this initialization format.

Example

The following are valid declarations:

```
BYTE i, j
BYTE :: k
BYTE, PARAMETER :: limit=120
BYTE val /253/
```

Related Statements

`INTEGER`

Related Concepts

The following are discussed elsewhere in this manual:

- Implicit typing rules: Chapter 3, Data Types and Data Objects
- Data representation models: Chapter 3, Data Types and Data Objects
- Storage classes for variables: Chapter 3, Data Types and Data Objects
- Automatic objects: Chapter 3, Data Types and Data Objects
- Arrays: Chapter 4, Arrays
- Expressions: Chapter 5, Expressions and Assignment
- Initialization expressions: Chapter 5, Expressions and Assignment

CALL

Invokes a subroutine.

`CALL` *`subr-name`*`[([` *`subr-act-arg-spec-list`* `])]`

`subr-name`	is the name of the subroutine being invoked.
`subr-act-arg-spec-list`	is a comma-separated list of subr-act-arg-spec.

subr-act-arg-spec is *[keyword =]subr-act-arg.*

subr-act-arg is one of the following:

- expression
- variable
- procedure-name
- **label*

keyword is one of the dummy argument names of the subroutine being invoked. If any keyword is specified, the subroutine interface must be explicit.

Description

A CALL statement is used to invoke (call) a subroutine, and to specify actual arguments, if any. Execution of the subroutine begins with the first executable statement. The sequence of events when a CALL statement is executed is as follows:

1. Actual arguments that are expressions are evaluated.
2. The actual arguments are associated with the corresponding dummy arguments.
3. Control transfers to the subroutine being called, and the subroutine executes.
4. Control returns from the subroutine, normally to the statement following the CALL statement, or to a statement label indicated by an alternate return specifier argument (of the form ** label*).

The correspondence between actual and dummy arguments is primarily by position: the first actual argument corresponds to the first dummy argument, the second to the second, and so on. The positional correspondence may be overridden by argument keywords, where a keyword name attached to an actual argument specifies a correspondence to the dummy argument of the same name. The following conditions govern the use of argument keywords:

- If an argument keyword is used, all subsequent arguments in the CALL statement must also be accompanied by keywords.
- If an optional argument is omitted, the keyword form is required for any following arguments.

- If an argument keyword is used, 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 with an interface block accessible to the program unit making the call.

A subroutine can call itself, directly or indirectly; in this case the keyword RECURSIVE must be added to the SUBROUTINE statement of the subroutine definition.

The %VAL and %REF built-in functions are provided as Intel Fortran extensions. These allow cross-calling between languages by enabling arguments to be passed by value and by reference, respectively. %VAL causes its argument to be passed by value, as if to a C function; it is sign-extended to a 32-bit value if it is less than 32 bits. %REF causes its argument to be passed by reference, similar to the default Fortran 95 behavior, except that the hidden length parameter of a CHARACTER string is not passed.

The only subroutine invocation other than by the CALL statement in Fortran 95 is through "defined assignment", where a defined type assignment operator that has been defined by means of a subroutine is used. See the INTERFACE statement in this chapter for more information.

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, "DASHED")
! Arguments given by position.
! Optional argument scale omitted.
CALL draw (scale=.4, x_end=0., y_end=0., &
      x_start=.5, y_start=3.)
```

```
! Arguments given by keyword.
! Optional argument form omitted.
```

Related Statements

`INTERFACE` and `SUBROUTINE`

Related Concepts

The correspondence between the dummy arguments of a subroutine and the actual arguments specified in its invocation (“Argument association”) is discussed in detail in Chapter 7, Program Units and Procedures, as are the other methods of association between a program unit and a subroutine called by it.

CASE

Marks start of statement block in a CASE construct.

`CASE` *`( case-selector ) [ construct-name ]`*

`case-selector` is a comma-separated list of ranges of values that are candidates for matching against the case index specified by the `SELECT CASE` statement. Each item in the list can take one of the following forms:

- case-value
- *low*:
- :*high*
- *low*:*high*
- `DEFAULT`

where *`case-value`*, are scalar initialization expressions of type

low, and *high*	integer, character, or logical; and DEFAULT indicates the statement block to execute if none of the other CASE statements in the CASE construct produces a match.
construct-name	is the name given to the CASE construct.

Description

The CASE statement is used in a CASE construct to mark the start of a statement block. The CASE construct can consist of multiple blocks; at most, one is selected for execution. Selection is determined by comparing the case index produced by the SELECT CASE statement to the *case-selector* in each CASE statement. If a match is found, the statement block under the matching *case-selector* executes. A match between the case index (*c*) and *case-selector* is determined for each form of *case-selector*, as follows:

case-value	For integer and character types, a match occurs if c .EQ. *case-value*. For logical types, a match occurs if *c* .EQV. *case-value*.
low:	For integer and character types, a match occurs if c .GE. *low*.
:high	For integer and character types, a match occurs if c .LE. *high*.
low : high	For integer and character types, a match occurs if c .GE. *low* .AND. *c* .LE. *high*.
DEFAULT	For integer, character, and logical types, a match occurs if no match is found with any other *case-selector* and DEFAULT is specified as a *case-selector*.

If CASE DEFAULT is not present and no match is found with any of the other CASE statements, none of the statement blocks within the CASE construct executes and execution resumes with the first executable statement following the END SELECT statement.

At most only one DEFAULT selector can appear within a CASE construct.

Each CASE statement must specify a unique value or range of values within a particular CASE construct. Only one match can occur, and only one statement block can execute.

All *case-selectors* and the case index within a particular CASE construct must be of the same type: integer, character, or logical. However, the lengths of character types can differ.

The colon forms— *low:*, *:high*, or *low:high*—are not permitted for a logical type.

Although putting the CASE statements in order according to range may improve readability, it is not necessary for correct or optimal execution of the CASE construct. In particular, DEFAULT can appear anywhere among the CASE statements and need not be the last.

CASE statements inside a named CASE construct need not specify *construct-name*; but if they do, the name they specify must match that of the SELECT CASE.

A CASE statement can have an empty statement block.

Example

The following example considers a person's credits and debits and prints a message indicating whether a resulting account balance will be overdrawn, empty, uncomfortably small, or sufficient:

```
INTEGER :: credits, debits
SELECT CASE (credits - debits)
CASE (:-1)
 PRINT *, 'OVERDRAWN'
 CALL TRANSFERFUNDS
CASE (0)
 PRINT *, 'NO MONEY LEFT'
CASE (1:50)
 PRINT *, 'BALANCE LOW'
CASE (51:)
 PRINT *, 'BALANCE OKAY'
END SELECT
```

Related Statements

SELECT CASE and END (construct)

Related Concepts

The CASE construct is described in Chapter 6, Execution Control.

CHARACTER

Declares entities of type character.

```
CHARACTER [char-selector] [[, attrib-list] ::]
 entity-list
```

char-selector specifies the length and kind of the character variable. It takes one of the following forms:

- ([LEN=]*len-param*[, KIND=*kind-param*])
- (*len-param*, [KIND=]*kind-param*)
- (KIND=*kind-param*[, LEN=*len-param*])
- **char-len* [,]
- **(len-param*) [,]

where *kind-param* (if present) must be 1 (the default), *len-param* is either an asterisk (*) or a specification expression, and *char-len* is an integer constant. In the last form, *len-param* is enclosed in parentheses, and the optional comma may be included only if the double colon does not appear in the type declaration statement. If *len-param* evaluates to a negative value, a zero-length string is declared. If *len-param* is unspecified, the default is 1.

attrib-list is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

For information about each of the attributes, see the corresponding statement in this chapter.

entity-list is a list of entities, separated by commas. Each entity takes the form:

name [(*array-spec*)] [* *len-param*]
 [= *initialization-expr*]

where *name* is the name of a variable or function, *array-spec* is a comma-separated list of dimension bounds, *len-param* is either an asterisk (*) or a specification expression, and *initialization-expr* is the initial value for the entity.

Description

The CHARACTER statement is used to declare the length and properties of character data.

The CHARACTER statement is constrained by the rules for all type declaration statements, including the requirement that it precede all executable statements.

Explicitly declaring an entity with the CHARACTER statement overrides any implicit typing rules in effect.

An array specification included with each entity in *entity-list* overrides any specification made with the DIMENSION attribute.

Initializing an entity implies the SAVE attribute.

If *attrib-list* or *initialization-expr* appears in the declaration, *entity-list* must be preceded by the double colon.

Assumed Character Length Parameter

To indicate that the length of a character can vary, you may use an assumed character length parameter by specifying an asterisk (*) for *len-param*. The asterisk may only be used to do the following:

- Declare the type of a function. The function must not be an internal or module function, nor must it be array-valued, pointer-valued, or recursive.
- Declare a dummy argument of a procedure.
- Declare a named constant (see the PARAMETER statement).

Automatic Character Variables

Automatic character variables are allowed within procedures, but only as local objects, not dummy arguments. For example,

```
CHARACTER(LEN=arg) :: name
```

declares an automatic character variable of the nonconstant length `arg` within a procedure. The value of `arg` is known only at entry to the procedure. Such character variables and character dummy arguments specified with a length of `*` are the only character entities whose length may vary. Automatic character variables cannot be initialized in a type declaration statement or appear in a `DATA` statement.

Initialization

`initialization-expr` must be a constant character expression that can be evaluated at compile time.

The following entities may not be initialized:

- Dummy arguments
- Function results
- Allocatable arrays
- Pointers
- External names
- Intrinsic names
- Automatic objects

If *`attrib-list`* includes the `PARAMETER` attribute, each entity in *`entity-list`* must include an initialization expression; see Chapter 5, Expressions and Assignment for information about initialization expressions.

Initializing an entity implies the `SAVE` attribute.

The following is an example of character array initialization using array constructor syntax:

```
CHARACTER(4) :: response(3) = (/"Yes.", &
 "No!!", "Huh?"/)
```

As shown in the example, all items in the array must be initialized, and all of the character constants must be of the same length. Implied-DO loops cannot be used to initialize an array in a type declaration statement.

As an extension, an initializer may appear between slashes in a type declaration statement. However, the double colon separator (::) may not be used with this format.

Examples

The following are valid declarations:

```
CHARACTER c1, c2
CHARACTER(LEN=80) :: text(0:25)
CHARACTER(2, 1), PARAMETER :: limit='ZZ'
```

The following are valid uses of the assumed length parameter:

```
CHARACTER(*) dummy_arg_name
CHARACTER(*), PARAMETER :: hello="Hi Sam"
CHARACTER(LEN=*), PARAMETER :: hello="Hi Sam"
```

Assuming that c is an ordinary variable and not the dummy argument to a procedure, the following declaration is an illegal use of the assumed length parameter:

```
CHARACTER*(*) c ! illegal
```

Related Concepts

The following related concepts are discussed elsewhere in this manual:

- Implicit typing rules: Chapter 3, Data Types and Data Objects
- Data representation models: Chapter 3, Data Types and Data Objects
- Storage classes for variables: Chapter 3, Data Types and Data Objects
- Automatic objects: Chapter 3, Data Types and Data Objects
- Arrays: Chapter 4, Arrays
- Expressions: Chapter 5, Expressions and Assignment
- Initialization expressions: Chapter 5, Expressions and Assignment

CLOSE

Terminates file connection.

CLOSE *(io-specifier-list)*

io-specifier-list	is a list of the following comma-separated I/O specifiers:
[UNIT=]*unit*	specifies the unit connected to an external file. *unit* must be a positive integer-valued expression. If the optional keyword UNIT= is omitted, unit must be the first item in *io-specifier-list*.
ERR=*stmt-label*	specifies the label of the executable statement to which control passes if an error occurs during statement execution. If neither IOSTAT= or ERR= is specified and an error occurs, the program aborts and a system error message is issued. *stmt-label* must be in the same scoping unit as the CLOSE statement with the ERR= specifier.
IOSTAT= *integer-variable*	returns the I/O status after the statement executes. If the statement executes successfully, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred. If neither IOSTAT= or ERR= is specified and an error occurs, the program aborts and a system error message is issued.

STATUS=*character-expression* specifies the state of the file after it is closed. *character-expression* can be one of the following arguments:

'KEEP'	Preserve the file after it is closed (default).
'DELETE'	Do not preserve the file after it is closed.

The STATUS= specifier is ignored if the file was opened as a scratch file; see the OPEN statement in this chapter.

Description

The CLOSE statement closes the file whose unit number was obtained from an OPEN statement. A CLOSE statement must contain a unit number and at most one each of the other I/O specifiers.

A CLOSE statement need not be in the same program unit as the OPEN statement that connected the file to the specified unit. If a CLOSE statement specifies a unit that does not exist or has no file connected to it, no action occurs.

Examples

The following examples illustrate different uses of the CLOSE statement. In the first example, the CLOSE statement closes the file connected to unit 10; after it is closed, the file will continue to exist, unless it was opened with the STATUS='SCRATCH' specifier:

```
CLOSE (10)
```

In the next example, after the file connected to unit 9 is closed, it will cease to exist:

```
CLOSE(UNIT=9,STATUS='DELETE')
```

The following code produces the same results as the previous example:

```
CHARACTER(LEN=6) cstat
cstat='delete'
CLOSE(UNIT=9,STATUS=cstat)
```

The following example closes the file connected to unit 8. If an error occurs, control is transferred to the executable statement labeled 100, and the error code is stored in the variable `ios`:

```
CLOSE(8,IOSTAT=ios,ERR=100)
```

Related Statements

`OPEN`

Related Concepts

For information about I/O concepts, see Chapter 8, I/O and File Handling, which also lists example programs performing I/O.

COMMON

Specifies common blocks.

```
COMMON [/[[common-block-name]]/] object-list
    [[,]/ [common-block-name] / object-list]...
```

common-block-name is the name of a labeled common block.

object-list is a comma-separated list of scalar variables, arrays, records, and structures. If an array is specified, it may be followed by an explicit-shape specification expression.

Description

The `COMMON` statement defines one or more named or unnamed storage areas to be shared by different program units. It also identifies the objects—that is, variables, arrays, records, and structures—to be stored in those areas. Objects in common that are shared by different program units are made accessible by storage association.

Each object following a common-block name is declared to be in that common block. If */common-block-name/* is omitted, all objects in the corresponding *object-list* are specified to be in blank common. It is also possible to declare variables in blank common by specifying two slashes without *common-block-name*. Consider the following examples:

```
!Declare variables a, b, c in blank common.
COMMON a, b, c

!Declare pay and time in blank common,
! and red in the named common block color.
COMMON pay, time, /color/red

!Variables a1 and a2 are in common block a;
! array x and variable y are in blank common;
! and variable d is in common block c
COMMON/a/a1,a2,//x(10),y,/c/d
```

Any common block name or blank common specification can appear more than once in one or more COMMON statements within the same program unit. The variable list following each successive appearance of the same common block name is treated as a continuation of the list for that common block name. For example, the following COMMON statements:

```
COMMON a,b,c /x/y,x,d //w,r
COMMON /cap/hat,visor, //tax, /x/o,t
```

are equivalent to:

```
COMMON a,b,c,w,r,tax
COMMON /x/y,x,d,o,t
COMMON /cap/hat,visor
```

Unlike named common blocks, blank common can differ in size in different scoping units. However, blank common cannot be initialized.

Intel Fortran allows you to mix CHARACTER and numeric data types in COMMON. This may create undesirable alignment, however, and is not recommended. As an extension, Intel Fortran saves all common blocks in static memory unless you use the name of a named COMMON block in the dynamic common command-line switch; see the *Intel Fortran Compiler User's Guide*.

Restrictions on Common Block Usage

All common block names must be distinct from subprogram names.

The size of a named common block must be the same in all program units where it is declared. Note, however, that the size of blank common can differ.

The following data items must not appear in a `COMMON` statement:

- Dummy arguments in a subprogram
- Functions, subroutines, or intrinsic functions
- Pointees declared by Cray-style pointers
- Variables accessible by use association
- Automatic entities, including automatic character strings
- Allocatable arrays

A variable can only appear in one `COMMON` statement within a program unit.

Zero-sized common blocks are allowed. Zero-sized common blocks with the same name are storage associated.

Array bounds in a `COMMON` statement must be constant specification expressions.

Structures in common must be of sequence type.

A pointer may appear in a common block. When it does, it must have the same type, type parameter, and rank in every instance of that common block.

Initializing Common Blocks

As an extension to the Standard, Intel Fortran allows common blocks to be initialized outside of a block data program unit; for example, in a subroutine. However, note that all data initialization for a given common block must occur in the same compilation unit.

Intel Fortran also allows blank common to be initialized.

Common Block Size

The size of a common block is determined by the number and type of the variables it contains. In the following example, the common block `my_block` takes 20 bytes of storage: `b` uses 8 (2 bytes per element) and `arr` uses 12 (4 bytes per element):

```
INTEGER(2) b(4)
INTEGER(4) arr(3)
COMMON /cb/b, arr
```

Data space within the common area for arrays `b` and `arr` shown in this example is allocated as follows:

Bytes	Common Block Variables
0, 1, 2, 3	`b(1), b(2)`
4, 5, 6, 7	`b(3), b(4)`
8, 9, 10, 11	`arr(1)`
12, 13, 14, 15	`arr(2)`
16, 17, 18, 19	`arr(3)`

Allocation of Common Block Storage

Common block storage is allocated at link time. It is not local to any one program unit.

Each program unit that uses the common block must include a `COMMON` statement that contains the block name, if a name was specified. Variables assigned to the common block by the program unit need not correspond by name, type, or number of elements with those of any other program unit. The only consideration is the size of the common blocks referenced by the different program units. Correspondence between objects in different instances of the same common block is established by storage association.

Note the following Intel Fortran: when types with different alignment restrictions are mixed in a common block, the compiler may insert padding bytes as necessary. (For exact data type alignment rules, see Chapter 3, Data Types and Data Objects.)

Example

The following example illustrates how the same common block can be declared in different program units with different variables but the same size:

```
! common declaration for program unit 1
INTEGER i, j, k
COMMON /my_block/ i, j, k

! common declaration for program unit 2
INTEGER n(3)
COMMON /my_block/ n(3)
```

The variables `i`, `j`, and `k` in program unit 1 share the same storage with the array `n` in program unit 2: `i` in program unit 1 matches up with `n(1)` in program unit 2, `j` with `n(2)`, and `k` with `n(3)`.

Related Statements

`EQUIVALENCE`

Related Concepts

For additional information about storage association and alignment, see Chapter 3, Data Types and Data Objects.

COMPLEX

Declares entities of type complex.

`COMPLEX` *`[kind-spec] [[, attrib-list] ::] entity-list`*

kind-spec is the kind type parameter that specifies the range and precision of the entities in *entity-list*. *kind-spec* takes the form:

([KIND=] *kind-param*)

where *kind-param* represents the kind of both the real and imaginary parts of the complex number. It can be a named constant or a constant expression that has the integer value of 4 or 8. The size of the default type is 4.

As an extension, *kind-spec* can take the form:

** len-param*

where *len-param* is the integer 8 or 16 (default = 8), which represents the size of the whole complex entity.

attrib-list is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

For information about each of the attributes, see the corresponding statement in this chapter.

entity-list is a list of entities, separated by commas. Each entity takes the form:

name [(array-spec)] [= initialization-expr]

where *name* is the name of a variable or function, *array-spec* is a comma-separated list of dimension bounds, and *initialization-expr* is the initial value for the entity.

Description

The COMPLEX statement is used to declare the length and properties of data that are approximations to the mathematical complex numbers. A complex number consists of a real part and an imaginary part. A kind parameter (if specified) indicates the representation method.

The COMPLEX statement is constrained by the rules for type declaration statements, including the requirement that it precede all executable statements.

Explicitly declaring an entity with the COMPLEX statement overrides any implicit typing rules in effect.

If *attrib-list* or *initialization-expr* appear in the declaration, *entity-list* must be preceded by the double colon.

If *array-spec* is specified for an entity, it overrides any DIMENSION attribute.

Initialization

initialization-expr must be a constant complex expression that can be evaluated at compile time.

The following entities may not be initialized:

- Dummy arguments
- Function results
- Allocatable arrays
- Pointers
- External names
- Intrinsic names
- Automatic objects

If *attrib-list* includes the PARAMETER attribute, each entity in *entity-list* must include an initialization expression.

To initialize an array in a COMPLEX statement, you must use an array constructor, as in the following example:

```
COMPLEX, DIMENSION(2) :: &
  cvec=(/(2.294, 6.288E-2), (-1.0096E7, 0)/)
```

If an array is initialized, all items in the array must be initialized. Implied-DO loops cannot be used to initialize an array in a type declaration statement.

As an Intel Fortran extension, an initializer may appear between slashes in a type declaration statement, as follows:

```
COMPLEX cx/(2.294, 6.288E-2)/
```

The double colon (::) may not be used with this initialization format.

Length Specification Extension

As a portability extension, Intel Fortran allows the following syntax for specifying the length of an entity:

name [**len*] [(*array-spec*)] [= *initialization-expr*]

If *array-spec* is specified, **len* may appear on either side of *array-spec*.

If *name* appears with **len*, it overrides the length specified by *kind-spec*. For example, the following statements are equivalent declarations of x:

```
COMPLEX(KIND = 8) x
COMPLEX(8) x*16
```

Examples

The following are valid declarations:

```
COMPLEX x, y
COMPLEX(KIND=8) :: z
COMPLEX,PARAMETER :: t1(2)=(/(3.2, 0), &
  (.04, -1.1)/)
```

Related Statements

DOUBLE COMPLEX

Related Concepts

The following are discussed elsewhere in this manual:

- Implicit typing rules: Chapter 3, Data Types and Data Objects
- Data representation models: Chapter 3, Data Types and Data Objects
- Storage classes for variables: Chapter 3, Data Types and Data Objects
- Automatic objects: Chapter 3, Data Types and Data Objects
- Arrays: Chapter 4, Arrays
- Expressions: Chapter 5, Expressions and Assignment
- Initialization expressions: Chapter 5, Expressions and Assignment

CONTAINS

Introduces an internal procedure or a module procedure.

```
CONTAINS
```

Description

The CONTAINS statement introduces an internal procedure or a module procedure, separating it from the program unit that contains it. The statement can be used in:

- A main program, external subprogram, or module subprogram; in each case, it precedes one or more internal procedures.
- A module, where it precedes any module procedures.

When a CONTAINS statement is present, at least one subprogram must follow it.

Examples

The first example illustrates CONTAINS introducing an internal subroutine. It also illustrates how the internal subroutine mechanism can provide an alternative to the FORTRAN 77 statement function mechanism.

```
PRINT *, double_real(6.6)
CONTAINS
  FUNCTION double_real (x); REAL x
   double_real = 2.0 * x
  END FUNCTION
END
```

The next example illustrates a main program with an internal procedure part.

```
PROGRAM electric   ! Program header
  REAL current   ! Specification part
  current = 100.5  ! Execution part begins
```

```
  CALL compute_resistance( voltage, current, &
                resistance )
  CONTAINS     ! Internal procedure part
   SUBROUTINE compute_resistance( v, i, r )
     REAL i
     r = v / i
   END SUBROUTINE
END PROGRAM electric
```

The third example is of a module that contains a module subprogram, which in turn contains an internal subprogram.

```
MODULE one
  CONTAINS
  SUBROUTINE two(x)   ! Module subprogram
   CONTAINS
   LOGICAL FUNCTION three(y)
             !Internal subprogram
   END FUNCTION three
  END SUBROUTINE two
END MODULE one
```

Related Statements

`SUBROUTINE` and `FUNCTION`

Related Concepts

The following are discussed in Chapter 7, Program Units and Procedures:

- Program units
- Internal subprograms
- Module subprograms

CONTINUE

Establishes reference point within a program unit.

```
CONTINUE
```

Description

The `CONTINUE` statement has no effect on program execution. Control passes to the next executable statement. The `CONTINUE` statement is generally used to mark a place for a statement label, especially when it occurs as the terminal statement of a FORTRAN 77-style `DO` loop.

`CONTINUE` is obsolescent in Fortran 95.

Example

```
count = 0
DO 20 i = 1, 10
  count = count + i
20 CONTINUE
PRINT *, count
```

Related Statements

`DO`

Related Concepts

Flow control statements are described in Chapter 5, Expressions and Assignment.

CYCLE

Interrupts current iteration of a DO loop.

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

do-construct-name
is the name of a DO construct that must contain this CYCLE statement.

Description

The CYCLE statement is used to control the execution of a DO loop. When it executes, it interrupts a currently executing loop iteration and passes control to the next iteration, making the appropriate adjustments to the loop index. It may be used with either the DO construct or the FORTRAN 77-style DO loop.

A CYCLE statement belongs to a particular DO loop. If *do-construct-name* is not given, the CYCLE statement resumes the immediately enclosing DO loop. If *do-construct-name* is given, the CYCLE statement resumes an enclosing named DO loop with the same name.

Example

The following example uses the CYCLE statement to control a bubble sort:

```
LOGICAL :: swap
INTEGER :: i, j
outer: DO i = 1, n-1
 swap = .FALSE.

 inner: DO j = n, i+1, -1
  IF (a(j) >= a(j-1)) CYCLE inner

  swap = .TRUE.
  atmp = a(j)
  a(j) = a(j-1)
  a(j-1) = atmp
 END DO inner
```

```
 IF (.NOT. swap) EXIT outer
END DO outer
```

Related Statements

`DO` and `EXIT`

Related Concepts

The `DO` construct and flow control are discussed in Chapter 5, Expressions and Assignment.

DATA

Initializes program variables.

`DATA` *`var-list1 / value-list1 / [[,]var-list2 / value-list2 /]...`*

`var-list` is a comma-separated list of entities, including the following:

- A variable name
- An array name
- An array triplet section; for example:
  `points(1:10:2)`
- An array element reference; for example:
  `scores(0)`
- A substring name; for example:
  `name(1:10)`
- An implied-`DO` loop; for example:
  `((matrix(i,j),i=0,5),j=5,10)`
- For information about implied-`DO` loops, see Chapter 8, I/O and File Handling.
- An object of a derived type
- A component of a derived-type object

The following cannot appear in *var-list:*

- Pointer-based variables
- Records and record field references. However, you can initialize a record's fields in the record's structure definition; see the RECORD statement in this chapter.
- Automatic objects, including automatic character strings
- Dummy arguments
- Allocatable arrays: that is, arrays declared with a specified rank, but no specified bounds within each dimension
- The result variable of a function
- Objects made available by use or host association
- Procedure names

value-list is a list of constant values, separated by commas. Each constant in the list represents a value to be assigned to the corresponding variable in *var-list*. A constant value can be optionally repeated by preceding the constant with a repetition factor. The syntax of a repeated constant is:

```
r*val
```

where *r* is a positive integer specifying the number of times that *val*, the constant value, is to be specified.

Description

The DATA statement initializes variables local to a program unit before the program unit begins execution. Initialization occurs as follows:

The *var-list* is expanded to form a sequence of scalar variables, and the *value-list* is expanded to form a sequence of scalar constants. The number of items in each expanded sequence must be the same, and there must be a one-to-one correspondence between the items in the two expanded lists. The variables in the expanded sequence of *var-list* are initialized on the basis of the correspondence.

If *var-list* contains an array name, the expanded sequence of constants must contain a constant for every element in the array.

A zero-sized array or an implied-DO list with an iteration count of zero in *var-list* contributes no variables to the expanded sequence of variables. However, a zero-length character variable does contribute a variable to the list.

If a constant is of any numeric or logical type, the corresponding variable can be of any numeric type. If an object is of derived type, the corresponding constant must be of the same type. If the type of the constant does not agree with the type of the variable, type conversion is performed, according to the rules described in Chapter 5, Expressions and Assignment.

Variables can be initialized with binary, octal, or hexadecimal constants.

A variable or array element must not appear in a DATA statement more than once. If two variables share the same storage space through an EQUIVALENCE statement, only one can appear in a DATA statement. If a substring of a character variable or other array element appears in a DATA statement, no overlapping substring (including the entire variable or array element) can appear in any DATA statement.

The length of a character constant and the declared length of its corresponding character variable need not be the same. If the constant is shorter than the variable, blank characters are placed in the remaining positions. If the constant is longer than the variable, the constant is truncated from the right until it is the same length as the variable

If a subscripted array element appears in *var-list*, then the subscript must be a specification expression.

DATA statements can be interspersed among executable statements. However, they initialize prior to runtime and, therefore, cannot be used as executable assignment statements.

Extensions to Fortran 95

A variable of type other than integer may be initialized with a binary, octal, or hexadecimal constant. The data type for a constant is determined from the type of the corresponding variable. The size (in bytes) of the variable determines how many digits of the octal or hexadecimal constant are used. If the constant lacks enough digits, the value is padded on the left with zeros. If the constant has too many digits, it is truncated on the left.

An integer, binary, octal, or hexadecimal constant can initialize a character variable of length one, as long as the value of the constant is in the range 0 to 255.

Examples

The following DATA statement initializes integer, logical, and character variables:

```
INTEGER i
LOGICAL done
CHARACTER(LEN=5) prompt
DATA i, done, prompt/10, .FALSE., 'Next?'/
```

The next DATA statement specifies a repetition factor of 3 to assign the value of 2 to all three elements of array i:

```
INTEGER, DIMENSION(3) :: i
DATA i/3*2/
```

The next DATA statement uses two nested implied-DO loops to assign the literal value X to each element of an array of 50 elements, k(10,5); for detailed information about implied-DO loops, see Chapter 8, I/O and File Handling:

```
CHARACTER, DIMENSION(10,5) :: k
DATA ((k(i,j),i=1,10),j=1,5)/50*'X'/
```

Related Statements

BYTE, CHARACTER, COMPLEX, DOUBLE COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, and REAL

Related Concepts

The following are discussed elsewhere in this manual:

- Initialization: Chapter 3, Data Types and Data Objects
- Assignment: Chapter 5, Expressions and Assignment
- Implied-DO loops: Chapter 8, I/O and File Handling

DEALLOCATE

Deallocates allocatable arrays and pointer targets.

DEALLOCATE *(alloc-obj-list[, STAT=scalar-int-var])*

alloc-obj-list is a comma-separated list of pointers or allocatable arrays.

STAT=*scalar-int-var*
returns the error status after the statement executes. If given, it is set to a positive value if an error is detected, and to zero otherwise. If there is no status variable, the occurrence of an error causes the program to terminate.

Description

The DEALLOCATE statement deallocates allocatable arrays and pointer targets, making the memory available for reuse. A specified allocatable array then becomes not allocated (as reported by the ALLOCATED intrinsic), while a specified pointer becomes disassociated (as reported by the ASSOCIATED intrinsic).

An error occurs if an attempt is made to deallocate an allocatable array that is not currently allocated or a pointer that is not associated. Errors in the operation of DEALLOCATE can be reported by means of the optional STAT= specifier.

You can deallocate an allocatable array by specifying the name of the array with the DEALLOCATE statement. You cannot deallocate a pointer that points to an object that was not allocated.

Some or all of a target associated with a pointer by means of the ALLOCATE statement can also be associated subsequently with other pointers. However, it is not permitted to deallocate a pointer that is not currently associated with the whole of an allocated target object.

Deallocation of a pointer target causes the association status of any other pointer associated with all or part of the target to become undefined. When a pointer is deallocated, its association status becomes disassociated, as if a NULLIFY statement had been executed.

Examples

The following example declares a complex array with the POINTER attribute. The ALLOCATE statement allocates target space to the array at run-time; the amount is determined by the input values to the READ statement. Later in the program, the DEALLOCATE statement will recover the space.

```
COMPLEX, POINTER :: hermitian (:, :)
...
READ *, m, n
ALLOCATE (hermitian (m, n))
...
DEALLOCATE (hermitian, STAT=ierr)
```

Related Statements

ALLOCATABLE, ALLOCATE, NULLIFY, and POINTER

Related Concepts

Pointers are discussed in Chapter 3, Data Types and Data Objects, Chapter 4, Arrays, and Chapter 5, Expressions and Assignment. The intrinsic inquiry functions ALLOCATED and ASSOCIATED are described in the *Intel Fortran Compiler User's Guide*.

DEBUG

Sets the conditions for operation of the debugging tool.

`DEBUG` *`option1 [[,]option2 ...]`*

An *`option`* can be any of the following:

`UNIT(`*un*`)` un is an integer constant that specifies a unit number. This is the debug output file in which the system will place the debug output. If you do not specify this option, the system uses the default debug output file. You must use the same unit for all unit definitions within an executable program.

`SUBCHK` *`(a1, a2,..., an)`*

a specifies the array name. The system compares the subscript combination with the array size to validate the named arrays. If the subscript value exceeds the size of the array, you will get a message in the debug file. The program executes using the incorrect subscript If you omit the list of array names, the program checks for valid subscript usage. If you omit the option, no arrays are checked for valid subscripts.

`TRACE` Displays the program flow by statement label. You must specify this statement in the `DEBUG` statement of each program that you want to trace. You must also use the `TRACE ON` statement in the first debug packet that you want to trace.

`INIT` *`(i1, i2,..., in)`*

`i` specifies an array name or variable with an assigned value that you want displayed in the debug output file. If i is a variable name, the name and value are displayed whenever it is assigned a new value in an assignment or in a `READ`, or `ASSIGN` statement.

SUBTRACE	Specifies to display the name of the subprogram when it is entered. When the subprogram completes, the message RETURN is displayed.

Description

You can supply the options in the DEBUG statement in any order, providing that you separate them with commas. You must place all debugging statements before the first statement of the program that you are debugging.

In a subroutine, you must place the debug statements immediately before the SUBROUTINE statement. In a function subprogram, you must place the debug statements immediately before the FUNCTION statement.

The following is the required statement sequence:

1. DEBUG statement
2. Debug packets
3. END DEBUG statement
4. First of the source statements of a program to be debugged

You must specify a debug packet with an AT statement and end it with another AT statement or an END DEBUG statement. You can write debug statements in fixed or free form and follow the same rule as other Fortran 95 programs. You can also use TRACE ON, TRACE OFF, and DISPLAY in the debug statement.

Guidelines for Using DEBUG

Use the following guidelines when you set up a debug packet:

You must contain within the DEBUG packet all DO loops, IF, ELSE IF, and ELSE statements.

- You must use unique statement labels within a debug packet and within a program.
- Do not correct errors in debug packets because the error will remain in the program when you remove the debug packet.
- Do not include specification statements nor any of the following statements in a debug packet:

```
BLOCK DATA
ENTRY
FUNCTION
PROGRAM
statement function
SUBROUTINE
```

You cannot transfer control to any statement label in a debug packet. However, you can return control from a packet to any point in the program. In addition, a debug packet cannot refer to a label in another debug packet. A debug packet can contain a RETURN, STOP, or CALL statement.

The SUBCHK function of DEBUG checks array subscripts if, and only if, the array has one dimension with a lower bound of 1. If the lower bound is not 1 and an error is detected, the message defaults to assign the element a lower bound of 1. If you check multi-dimensional arrays for valid subscripts, the array defaults to a single-dimension array with the correct number of elements. The resulting check indicates whether you are referencing an element within the range of the array, but not whether the subscript is invalid. Individual subscripts are not checked for a valid range.

Therefore, if array A has the dimensions A(5,6) and a reference is made to A(K, 2), where K is 7, the SUBCHK function does not flag this because the subscript value yields an element within array A. The values of the first and second subscripts are not checked for having values of 1 to 5 or 1 to 6 respectively.

Examples

The following example program illustrates the DEBUG statement:

Example 1:

```
DEBUG UNIT(7),SUBCHK
END DEBUG
PROGRAM TEST
    .
    .
    .
END
```

This checks all arrays for valid subscripts.

Example 2:

```
   DEBUG UNIT(7),
   AT 13
   write(6, 21) W, Y, Z
21 FORMAT(1X, 'W=', I12, 'Y=', I12, 'Z=', I12)
   END DEBUG
       .
       .
       .
```

```
    INTEGER A, B, C
        .
        .
        .
12 Y=W* SQRT(FLOAT(C))
13 IF(Y) 35, 45, 55
        .
        .
        .
```

The values of W, Y, and Z are examined as they were at the completion of the arithmetic operation in statement 13. Therefore, the statement label specified in the AT statement is 13. The values of W, Y, and Z are written to the file connected to unit 7.

Example 3:

```
    DEBUG TRACE, UNIT(7)
    AT 13
    TRACE ON
    AT 35
    TRACE OFF
    AT 45
    DISPLAY z
    TRACE ON
    END DEBUG
        .
        .
        .
13 X=3.0
18 L=1
22 Y = X + 1.5
35 DO 30 I = 1,5
        .
        .
        .
40 CONTINUE
45 Z = Y + 3.415
50 W=Z**2
```

```
55 CALL SUB1(W, L, R)
   STOP
   END

   DEBUG SUBTRACE, TRACE
   AT 8
   TRACE ON
   END DEBUG
   SUBROUTINE SUB1(A, I, B)
       .
       .
       .
8  Y=FUNC1 (A-INT(A))
   WRITE(6, *) B
       .
       .
       .
   RETURN
   END

   DEBUG SUBTRACE, TRACE
   AT 100
   TRACE ON
   END DEBUG
   FUNCTION FUNC1(C)
       .
       .
       .
100FUNC1 = COS(C) + SIN(C)
       .
       .
       .
   RETURN.
   END
```

Tracing begins when statement 13 is encountered as specified by the TRACE ON statement in the first debug packet. Tracing stops at statement 35 as specified by TRACE OFF. Tracing begins again at statement 45 and the value of Z is written to the output file as specified in the third debug packet.

Related Concepts

For information about statements related to DEBUG see TRACE ON, TRACE OFF, AT, and END DEBUG.

DECODE

Inputs formatted data from internal storage.

```
DECODE (count, format, unit, io-specifier-list)
 [in-list]
```

count is an integer expression that specifies the number of characters (bytes) to translate from character format to internal (binary) format. *count* must precede *format*.

format specifies the format specification for formatting the data. format can be one of the following:

- The label of a FORMAT statement containing the format specification.
- An integer variable that has been assigned the label of a FORMAT statement.
- An embedded format specification. For information about embedded format specifications, see Chapter 9, I/O Formatting.

format must be the second of the parenthesized items, immediately following *count*. Note that the keyword FMT= is not used.

unit is the internal storage designator. It must be a scalar variable or array name. Assumed-size and adjustable-size arrays are not permitted. Note that char-var-name is not a unit number and that the keyword UNIT= is not used.

unit must be the third of the parenthesized items, immediately following *format*.

io-specifier-list is a comma-separated list of I/O specifiers. Note that the unit and format specifiers are required; the other I/O specifiers are optional. The arguments that can appear in io-specifier-list as I/O specifiers are:
ERR=stmt-label
IOSTAT=integer-variable
in-list

ERR=stmt-label specifies the label of the executable statement to which control passes if an error occurs during statement execution.

IOSTAT=integer-variable returns the I/O status after the statement executes. If the statement successfully executes, *integer-variable* is set to zero. If an end-of-file record is encountered without an error condition, it is set to a negative integer. If an error occurs, *integer-variable* is set to a positive integer that indicates which error occurred.

in-list is a comma-separated list of data items for input. The data items can include expressions and implied-DO lists (see Chapter 8, I/O and File Handling).

Description

The DECODE statement is a nonstandard feature of Intel Fortran and is provided for compatibility with other versions of Fortran. The internal-I/O capabilities of the standard READ statement provide similar functionality and should be used to ensure portability.

The DECODE statement translates formatted character data into its binary (internal) representation.

Examples

The following example program illustrates the DECODE statement:

```
PROGRAM decode_example
 CHARACTER(LEN=20) :: buf
 INTEGER i, j, k
 buf = 'XX1234 45 -12XXXXXX'
 DECODE (15,'(2X,3I4,1X)', buf) i, j, k
 ! The equivalent READ statement is:
 ! READ (buf, '(2X,3I4,1X)') i, j, k
 PRINT *, i, j, k
END
```

When compiled and executed, this program produces the following output:

```
 1234 45 -12
```

Related Statements

ENCODE and READ

Related Concepts

For information about I/O concepts, see Chapter 8, I/O and File Handling, which also presents example programs performing I/O. For information about I/O formatting, see Chapter 9, I/O Formatting.

DIMENSION (Statement and Attribute)

Declares a variable to be an array.

A type declaration statement with the DIMENSION attribute is:

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

type	is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE(*type-name*), etc.), as described in Chapter 3, Data Types and Data Objects.

array-spec is one of the following:

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

explicit-shape-spec

[lower-bound :] upper-bound

lower-bound

specification-expr

upper-bound

specification-expr

assumed-shape-spec

[lower-bound] :

deferred-shape-spec

:

assumed-size-spec

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

That is, *assumed-size-spec* is *explicit-shape-spec-list* with the final upper bound given as *.

attrib-list is a comma-separated list of attributes including DIMENSION and optionally those attributes compatible with it, namely:

ALLOCATABLE	PARAMETER	PUBLIC
INTENT	POINTER	SAVE
OPTIONAL	PRIVATE	TARGET

entity-list

object-name [(array-spec)]

If (*array-spec*) is present, it overrides the (*array-spec*) given with the DIMENSION keyword in *attribute-list*; see the example below.

The syntax of the DIMENSION statement is:

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

Description

An array consists of a set of objects called the array elements, all of the same type and type parameters, arranged in a pattern involving columns, and possibly rows, planes, and higher dimensioned configurations. The type of the array elements may be intrinsic or user-defined. In Intel Fortran, an array 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 that 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 said to be conformable.

It is not necessary for the keyword DIMENSION to appear in the declaration of a variable to give it the DIMENSION attribute. This attribute, as well as the rank, and possibly the extents and the bounds of an array, may be specified in the entity declaration part of any of the following statements:

- type declaration
- DIMENSION
- ALLOCATABLE
- COMMON
- POINTER
- TARGET

The *array-spec* (see Syntax, above) determines the category of the array being declared. As fully described in Chapter 4, Arrays, these categories are:

- Explicit-shape array
- Assumed-shape array
- Assumed-size array
- Deferred-shape array

Examples

```
REAL a (20,2), b (20,2), c (20,2)
REAL, DIMENSION (20,2) :: a, b, c
! These 2 declaration statements are equivalent.

DIMENSION x(100), y(100)
! x and y are 1-dimensional.
INTEGER jj (0:100, -1:1)
! Lower bounds are specified for jj.
!  (If not given, they default to 1.)

LOGICAL l
ALLOCATABLE l(:,:,:,:)
! l is a 4-dimensional, allocatable,
!  deferred shape logical array.

COMPLEX s     ! s has explicit shape and
TARGET :: s(10,2) !  has the target attribute.

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

SUBROUTINE calc(arr1, ib1, ib2)
REAL, DIMENSION (ib1, ib2) :: arr1, arr2
! arr1 is an adjustable array.
! arr2 is an automatic array.

REAL, POINTER, DIMENSION(:,:) :: arr3
! arr3 is a deferred-shape array with the
!  pointer attribute.
LOGICAL, DIMENSION(10,20) :: ta, tb(10,10), tc
! All three arrays have explicit shape.
!  The array specifier (10,10) overrides the
```

```
!  (10,20) specifier for the declaration of
!  tb only.
```

Related Statements

ALLOCATABLE, COMMON, POINTER, TARGET, TYPE, and the type declaration statements

Related Concepts

See Chapter 4, Arrays for a detailed description of Fortran 95 arrays.

The following intrinsic functions relate to array properties:

- LBOUND
- RESHAPE
- SHAPE
- SIZE
- UBOUND

DISPLAY

Displays data in an output format.

`DISPLAY` *`list`*

`list` a comma-separated list of array and variable names.

Description

Displays the results of a debugging operation and places it in the debug output file without the need for FORMAT, NAMELIST, or WRITE.

The effect of a DISPLAY list statement is the same as the following source language statements:

```
NAMELIST / name / list
WRITE (un, name)
```

where *name* is the same in both statements:

Array elements, dummy arguments, and substring references cannot appear in the list.

Related Concepts

See the example section "DEBUG" for a demonstration of how to use DISPLAY.

DO

Controls execution of DO loop.

[construct-name :] DO [label] [loop-control]

construct-name
: is the name given to the DO construct. If construct-name is specified, an END DO statement must appear at the end of the DO construct and have the same construct-name.

label
: is the label of an executable statement that terminates the DO loop. If you specify label, you can terminate the DO loop either with an END DO statement or with an executable statement; the terminating statement must include label. If you do not specify label, you must terminate the DO loop with the END DO statement.

loop-control
: is information used by the DO statement to control the loop. It can take one of the following forms:

- *index = init, limit [, step]*
- WHILE *(logical-expression)*
- *loop-control*

In the first form, *index* is a scalar variable of type integer or real; *init*, *limit*, and *step* are scalar expressions of type integer or real. In the second form, *logical-expression* is a scalar logical

expression. In the third form, *loop-control* is omitted. If you use the second or third form, you must terminate the DO loop with the END DO statement.

Description

The syntax of the DO statement allows for the following types of DO loops:

- Counter-controlled 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 (*logical-expression*) is tested before each execution of the block; when it is false, execution ceases. An exit may occur at any time.
- Infinite loop: there is no *loop-control*; repeated execution of the block ceases only when an exit from the loop occurs.

For more information about the different types of DO loops, see Chapter 6, Execution Control, "DO construct".

When *label* is present in the DO statement, it specifies the label of the terminating statement of the DO loop. The terminating statement cannot be any of the following statements:

- GO TO (unconditional)
- GO TO (assigned)
- IF (arithmetic)
- IF (block)
- ELSE or ELSE IF
- END, END IF, END SELECT, or END WHERE
- RETURN
- STOP
- DO

- Any nonexecutable statement

  Note, however, that the terminating statement can be an IF (logical) or an END DO statement.

  To maintain compatibility with some older versions of Fortran, you can use the +onetrip command-line option to ensure that every counter-controlled DO loop in the program executes at least once. For more information about this option, see the *Intel Fortran Compiler User's Guide*.

Extended-range DO Loops

Extended-range DO loops—a compatibility extension—allow a program to transfer control outside the DO loop's range and then back into the DO loop. Extended-range DO loops work as follows: if a control statement inside a DO loop transfers control to a statement outside the DO loop, then any subsequent statement can transfer control back into the body of the DO loop.

For example, in the following code, the range of the DO loop is extended to include the statement GOTO 20, which transfers control back to the body of the DO loop:

```
  DO 50 i = 1, 10
20  n = n + 1
   IF (n > 10) GOTO 60
50 CONTINUE  ! normally, the range ends here
60 n = n + 100 ! this is the extended range,
  GOTO 20  ! which extends down to this line
```

Examples

The following DO construct displays the integers 1 through 10:

```
DO i = 1, 10
 WRITE (*, *) i
END DO
```

The next example is a FORTRAN 77-style DO loop that does the same as the preceding example:

```
  DO 50 i = 1, 10
   WRITE (*, *) i
50 CONTINUE
```

The following DO construct iterates 5 times, decrementing the loop index from 10 to 2:

```
DO i = 10, 1, -2
END DO
```

The following is an example of a DO WHILE loop:

```
DO WHILE (sum < 100.0)
  sum = sum + get_num(unit)
END DO
```

The following example illustrates the use of the EXIT statement to exit from a nested DO loop. The loops are named to control which loop is exited. Note that *loop-control* is missing from both the inner and outer loops, which therefore can be exited only by the execution of the EXIT statements:

```
outer:DO
 READ *, val
 new_val = 0
 inner:DO
  new_val = new_val + proc_val(val)
  IF (new_val >= max_val) EXIT inner
  IF (new_val == 0) EXIT outer
 END DO inner
END DO outer
```

The next DO construct never executes:

```
DO i = 10, 1, -2
END DO
```

Related Statements

CONTINUE, CYCLE, END (construct), and EXIT

Related Concepts

For information about the DO construct (including examples), see Chapter 6, Execution Control.

DOUBLE COMPLEX

Declares entities of type double complex.

```
DOUBLE COMPLEX [ [, attrib-list ] ::] entity-list
```

attrib-list is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

NOTE. DOUBLE COMPLEX *is equivalent to* COMPLEX*16 *and* COMPLEX(KIND=16).

entity-list is a list of entities, separated by commas. Each entity takes the form:

```
name [( array-spec )] [= initialization-expr]
```

where

name is the name of a variable or function

array-spec is a comma-separated list of dimension bounds

initialization-expr is the initial value for the entity.

Description

The DOUBLE COMPLEX statement is an Intel Fortran extension that declares the properties of complex data that has greater precision than data of default type complex. The two parts of a double complex value are each a double precision value. Note that the DOUBLE COMPLEX statement does not have a kind parameter.

The DOUBLE COMPLEX statement is constrained by the rules for type declaration statements, including the requirement that it precede all executable statements.

Explicitly declaring an entity with the DOUBLE COMPLEX statement overrides any implicit typing rules in effect.

If *attrib-list or initialization-expr* appear in the declaration, *entity-list* must be preceded by the double colon.

If *array-spec* is specified for an entity, it overrides any DIMENSION attribute.

Initialization

initialization-expr must be a constant complex typed expression that can be evaluated at compile time.

The following entities may not be initialized:

- Dummy arguments
- Function results
- Allocatable arrays
- Pointers
- External names
- Intrinsic names
- Automatic objects

If *attrib-list* includes the PARAMETER attribute, each entity in *entity-list* must include an initialization expression.

To initialize an array in a DOUBLE COMPLEX statement, you must use an array constructor, as in the following example:

```
DOUBLE COMPLEX, DIMENSION(2) :: dc_vec = &
  (/(2.294D-8, 6.288D-4), (-4.817D4, 0)/)
```

If an array is initialized, all items in the array must be initialized. Implied-DO loops cannot be used to initialize an array in a type declaration statement.

As an extension, an initializer may appear between slashes in a type declaration statement, as follows:

```
DOUBLE COMPLEX dcx/(2.294D-8, 6.288D-4)/
```

The double colon (`::`) may not be used with this initialization format.

Example

The following are valid declarations:

```
DOUBLE COMPLEX x, y
DOUBLE COMPLEX, PARAMETER :: t1(2)=(/(1.2, 0), &
  (-1.01, 0.0009)/)
```

Related Statements

`COMPLEX`

Related Concepts

The following are discussed elsewhere in this manual:

- Implicit typing rules: Chapter 3, Data Types and Data Objects
- Data representation models: Chapter 3, Data Types and Data Objects
- Storage classes for variables: Chapter 3, Data Types and Data Objects
- Automatic objects: Chapter 3, Data Types and Data Objects
- Arrays: Chapter 4, Arrays
- Expressions: Chapter 5, Expressions and Assignment
- Initialization expressions: Chapter 5, Expressions and Assignment

DOUBLE PRECISION

Declares entities of type double precision.

```
DOUBLE PRECISION [ [, attrib-list] ::] entity-list
```

attrib-list is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

NOTE. DOUBLE PRECISION *is equivalent to* REAL*8 *and* REAL(KIND=8).

For information about each of the attributes, see the corresponding statement in this chapter.

entity-list is a list of entities, separated by commas. Each entity takes the form:

```
name [( array-spec )] [= initialization-expr]
```

where *name* is the name of a variable or function, *array-spec* is a comma-separated list of dimension bounds, and *initialization-expr* is the initial value for the entity.

Description

The DOUBLE PRECISION statement is used to declare the properties of real data that has greater precision than data of default type real. By default, the DOUBLE PRECISION statement is equivalent to the REAL(KIND=8) statement. Note that the DOUBLE PRECISION statement does not have a kind parameter.

The DOUBLE PRECISION statement is constrained by the rules for type declaration statements, including the requirement that it precede all executable statements.

Explicitly declaring an entity with the DOUBLE PRECISION statement overrides any implicit typing rules in effect.

If *attrib-list* or *initialization-expr* appears in the declaration, *entity-list* must be preceded by the double colon.

If *array-spec* is specified for an entity, it overrides any DIMENSION attribute.

Initialization

initialization-expr must be a constant expression that can be evaluated at compile time.

The following entities may not be initialized:

- Dummy arguments
- Function results
- Allocatable arrays
- Pointers
- External names
- Intrinsic names
- Automatic objects

If *attrib-list* includes the PARAMETER attribute, each entity in *entity-list* must include an initialization expression.

To initialize an array in a DOUBLE PRECISION statement, you must use an array constructor, as in the following example:

```
DOUBLE PRECISION, DIMENSION(4) :: dp_vec= &
  (/4.7D0, 5.2D0, 3.3D0, 2.9D0/)
```

If an array is initialized, all items in the array must be initialized. Implied-DO loops cannot be used to initialize an array in a type declaration statement.

As an extension, an initializer may appear between slashes in a type declaration statement, as follows:

```
DOUBLE PRECISION dp1/5.28D0/, dp2/72.3D0/
```

The double colon (::) may not be used with this initialization format.

Example

The following are valid declarations:

```
DOUBLE PRECISION x, y
DOUBLE PRECISION, PARAMETER :: pi=3.1415927D0
```

Related Statements

REAL

Related Concepts

The following are discussed elsewhere in this manual:

- Implicit typing rules: Chapter 3, Data Types and Data Objects
- Data representation models: Chapter 3, Data Types and Data Objects
- Storage classes for variables: Chapter 3, Data Types and Data Objects
- Automatic objects: Chapter 3, Data Types and Data Objects
- Arrays: Chapter 4, Arrays
- Expressions: Chapter 5, Expressions and Assignment
- Initialization expressions: Chapter 5, Expressions and Assignment

EJECT

Starts a new full page of the source listing.

```
EJECT
```

Description

The EJECT statement is a compiler directive that starts a new page of the source listing. You cannot CONTINUE an EJECT statement.

ELSE

Provides a default path of execution for IF construct.

```
ELSE [ construct-name ]
```

construct-name
is the name given to the IF construct. If construct-name is specified, the same name must also appear in the IF statement and in the END IF statement.

Description

The ELSE statement is used in an IF construct to provide a statement block for execution if none of the logical expressions in the IF and ELSE IF statements in the IF construct evaluates to true.

An IF construct may contain (at most) one ELSE statement. If present, it must follow all ELSE IF statements within the IF construct.

Example

```
IF (a > b) THEN
 max = a
ELSE IF (b > max) THEN
 max = b
ELSE
 PRINT *, 'The two numbers are equal.'
 STOP 'Done'
END IF
```

Related Statements

ELSE IF, END IF, and IF (construct)

Related Concepts

For information about the IF construct, see Chapter 6, Execution Control.

ELSE IF

Provides alternate path of execution for IF construct.

```
ELSE IF (logical-expression) THEN [construct-name]
```

logical-expression is a scalar logical expression.

construct-name
: is the name given to the IF construct. If construct-name is specified, the same name must also appear in the IF statement and in the END IF statement.

Description

The ELSE IF statement executes the immediately following statement block, if the following conditions are met:

- None of the logical expressions in the IF statement and any previous ELSE IF statements evaluates to true.
- *logical-expression* evaluates to true.

Branching to an ELSE IF statement is illegal.

Example

```
INTEGER temperature
INTEGER, PARAMETER :: hot=1, cold=2
IF (temperature == hot) THEN
 PRINT *, 'Turn down your thermostat.'
ELSE IF (temperature == cold) THEN
```

```
 PRINT *, 'Turn up your thermostat.'
ELSE
 PRINT *, 'Your thermostat is working OK.'
END IF
```

Related Statements

ELSE, END IF, and IF (construct)

Related Concepts

For information about the IF construct, see Chapter 6, Execution Control.

ELSEWHERE

Introduces optional ELSEWHERE block within a WHERE construct.

```
ELSEWHERE
```

Description

The ELSEWHERE statement introduces an ELSEWHERE block, which is an optional component of the WHERE construct. The ELSEWHERE statement executes on the complement of the WHERE condition. For additional information, see the WHERE statement in this chapter.

Example

```
WHERE( b .GE. 0.0 )
  sqrt_b = SQRT(b)
  ! Assign to sqrt_b only where logical array b
  !  is zero or positive.
ELSEWHERE
  sqrt_b = 0.0
  ! Assign sqrt_b where b is negative.
END WHERE
```

Related Statements

WHERE and END WHERE

Related Concepts

The WHERE construct is described in Chapter 5, Expressions and Assignment.

ENCODE

Outputs formatted data to internal storage.

```
ENCODE
(count, format, unit, io-specifier-list) [out-list]
```

count is an integer expression that specifies the number of characters (bytes) to translate from character format to internal (binary) format. count must precede format.

format specifies the format specification for formatting the data. *format* can be one of the following:

- The label of a FORMAT statement containing the format specification.
- An integer variable that has been assigned the label of a FORMAT statement.
- An embedded format specification. For information about embedded format specifications, see Chapter 9, I/O Formatting.

format must be the second of the parenthesized items, immediately following count. Note that the keyword FMT= is not used.

unit is the internal storage designator. It must be a scalar variable or array name. Assumed-size and adjustable-size arrays are not permitted. Note that *char-var-name* is not a unit number and that the keyword UNIT= is not used.

unit must be the third of the parenthesized items, immediately following format.

io-specifier-list is a comma-separated list of I/O specifiers. Note that the unit and format specifiers are required; the other I/O specifiers are optional. The following I/O specifiers can appear in io-specifier-list:

ERR=stmt-label
specifies the label of the executable statement to which control passes if an error occurs during statement execution.

IOSTAT=*integer-variable*
returns the I/O status after the statement executes. If the statement successfully executes, *integer-variable* is set to zero. If an end-of-file record is encountered without an error condition, it is set to a negative integer. If an error occurs, *integer-variable* is set to a positive integer that indicates which error occurred.

out-list is a comma-separated list of data items for output. The data items can include expressions and implied-DO lists (see Chapter 8, I/O and File Handling).

Description

The ENCODE statement is a nonstandard feature of Intel Fortran and is provided for compatibility with other versions of Fortran. The internal-I/O capabilities of the standard WRITE statement provide similar functionality and should be used to ensure portability.

The ENCODE statement translates data from its internal (binary) representation into formatted character data.

Examples

The following example program uses the ENCODE statement to write to an internal file:

```
PROGRAM encode_example
 CHARACTER(LEN=20) :: buf
 ENCODE (LEN(buf), '(2X, 3I4, 1X)', buf) &
```

```
  1234, 45, -12
 PRINT *, buf
END
```

When compiled and executed, this program outputs the following (where ~~b~~ represents the blank character):

~~bb~~1234~~bb~~45~~b~~-12~~bbbbbb~~

Related Statements

DECODE and WRITE

Related Concepts

For information about I/O concepts, see Chapter 8, I/O and File Handling, which also gives example programs using different kinds of I/O. For information about I/O formatting, see Chapter 9, I/O Formatting.

END

Marks the end of a program unit or procedure.

END [*keyword* [*name*]]

keyword	is one of the keywords BLOCK DATA, FUNCTION, MODULE, PROGRAM, or SUBROUTINE. When the END statement is used for an internal procedure or module procedure, the FUNCTION or SUBROUTINE keyword is required.
name	is the name given to the program unit. If name is specified, keyword must also be specified.

Description

The END statement is the last statement of a program unit (that is, a main program, function, subroutine, module, or block data subprogram), an internal procedure, or a module procedure. It is the only statement that is required within a program unit.

Examples

The following example illustrates the use of the END statement to indicate the end of a main program. Notice that, even though the main program unit is given a name, the END PROGRAM statement does not require it:

```
PROGRAM main_prog
END PROGRAM
```

In the next example, the END statement marks the end of an internal function and must therefore specify the keyword FUNCTION. However, it is not required that the name, get_args, be also specified:

```
FUNCTION get_args (arg1, arg2)
END FUNCTION get_args
```

The following example uses the END statement to indicate the end of a block data subprogram. Because the END statement specifies the program unit name, it must also specify the keyword BLOCK DATA:

```
BLOCK DATA main_data
END BLOCK DATA main_data
```

Related Statements

BLOCK DATA, FUNCTION, MODULE, PROGRAM, and SUBROUTINE

Related Concepts

For information about program units, see Chapter 7, Program Units and Procedures.

END (Construct)

Terminates a CASE, DO, IF, or WHERE construct.

END *construct-keyword* [*construct-name*]

construct-keyword is one of the keywords DO, IF, SELECT, or WHERE.

construct-name is the name given to the construct terminated by this statement.

Description

The END (construct) statement terminates a CASE, DO, IF, or WHERE construct. If *construct-name* appears in the statement that introduces the construct, the same name must also appear in the END statement. If no *construct-name* is given in the introducing statement, none must appear in the END statement.

Example

For examples of each of the END (construct) statement, see the descriptions of the DO, IF, SELECT, or WHERE statements in this chapter.

Related Statements

DO, IF, SELECT, and WHERE

Related Concepts

The CASE, DO, and IF constructs are discussed in Chapter 6, Execution Control; the WHERE construct is discussed in Chapter 5, Expressions and Assignment.

END (Structure Definition)

Terminates the definition of a structure or union.

```
END record-keyword
```

record-keyword
is one of the keywords MAP, STRUCTURE, or UNION.

Description

The END (record definition) statement is an Intel Fortran extension that is used to delimit the definition of a structure (END STRUCTURE) or a union within a structure (END UNION and END MAP). For more information, refer to the description of the STRUCTURE statement in this chapter.

END INTERFACE

Terminates a procedure interface block.

```
END INTERFACE
```

Description

In Fortran 95, external procedures may be given explicit interfaces by means of procedure interface blocks. Such a block is always terminated by the END INTERFACE statement.

Example

The following makes the interface of function r_ave explicit, giving it the generic name g_ave.

```
INTERFACE g_ave
  FUNCTION r_ave(x)
  ! Get the size of array x from
  !  module ave_stuff.
   USE ave_stuff, ONLY: n
   REAL r_ave, x(n)
  END FUNCTION r_ave
END INTERFACE
```

Related Statements

INTERFACE

Related Concepts

Interface blocks are described in Chapter 7, Program Units and Procedures.

END DEBUG

Terminates the last debug packet for the program.

```
END DEBUG
```

Description

Place the END DEBUG statement after the other debug statements and just before the first statement of the program being debugged. You can use only one END DEBUG statement per program unit.

An IF construct may contain (at most) one ELSE statement. If present, it must follow all ELSE IF statements within the IF construct.

Related Concepts

For details on debugging, see "DEBUG".

END TYPE

Terminates a derived type definition.

```
END TYPE [type-name]
```

type-name is the name of the derived type being defined. type-name is optional. If given, it must be the same as the type-name specified in the TYPE statement introducing the derived type definition.

Description

The END TYPE statement terminates the definition of a derived type.

Example

The following is a simple example of a derived type with two components, `high` and `low`:

```
TYPE temp_range
  INTEGER high, low
END TYPE temp_range
```

Related Statements

`TYPE` (definition)

Related Concepts

Derived types are described in Chapter 3, Data Types and Data Objects.

ENDFILE

Writes end-of-file record to file.

The syntax of the `ENDFILE` statement can take one of the following forms:

- Short form:
  `ENDFILE` *`integer-expression`*
- Long form:
  `ENDFILE` *`(io-specifier-list)`*

`integer-expression` is the number of the unit connected to a sequential file.

`io-specifier-list` is a list of the following comma-separated I/O specifiers:

`[UNIT=]`*`unit`* specifies the unit connected to a device or external file opened for sequential access. *`unit`* must be an integer expression that evaluates to a non-negative number. If the optional keyword `UNIT=` is omitted, *`unit`* must be the first item in *`io-specifier-list`*.

ERR=*stmt-label*	specifies the label of the executable statement to which control passes if an error occurs during statement execution.
IOSTAT=*integer-variable*	returns the I/O status after the statement executes. If the statement executes successfully, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.

Description

The ENDFILE statement writes an end-of-file record to the file or device connected to the specified unit at the current position and positions the file after the end-of-file record.

An end-of-file record can occur only as the last record of a disk file. After execution of an ENDFILE statement, the file is positioned beyond the end-of-file record; any records beyond the current position are lost—that is, the file is truncated.

Some devices (for example, magnetic tape units) can have multiple end-of-file records, with or without intervening data records.

An end-of-file record can be written to a sequential file only.

Examples

The following statement writes an end-of-file record to the file connected to unit 10:

```
ENDFILE 10
```

The following statement writes an end-of-file record to the file connected to unit 17. If an error occurs during the execution of the statement, control passes to the statement at label 99, and the error code is returned in ios:

```
INTEGER :: ios
...
ENDFILE (17, ERR=99, IOSTAT=ios)
```

Related Statements

BACKSPACE, OPEN, and REWIND

Related Concepts

For information about I/O concepts, see Chapter 8, I/O and File Handling, which also presents example programs performing I/O. For information about I/O formatting, see Chapter 9, I/O Formatting.

ENTRY

Provides an additional external or module subprogram entry point.

```
ENTRY entry-name [( [dummy-arg-list] )
   [ RESULT (result-name) ]]
```

entry-name is the name of the entry point (subroutine or function) defined by the ENTRY statement. It must differ from the original subroutine or function name, and from other ENTRY statement entry-names specified in the subprogram in which it appears.

dummy-arg-list is a comma-separated list of dummy arguments for the subroutine or function defined by the ENTRY statement. The same rules and restrictions apply as for subroutine dummy arguments or function dummy arguments, as appropriate.

result-name is the result variable for a function defined by an ENTRY statement. *result-name* is optional; if not specified, the result variable is entry-name.

The RESULT *(result-name)* clause can only be specified when the ENTRY statement is included in a function subprogram.

Description

When an ENTRY statement appears in a function subprogram, it effectively provides an additional FUNCTION statement in the subprogram: execution starts from the ENTRY statement when the *entry-name* is invoked (by

being used). Similarly, an ENTRY statement in a subroutine subprogram effectively provides an additional SUBROUTINE statement in the subprogram, and execution starts from the ENTRY statement when the *entry-name* is called.

The following restrictions apply to the ENTRY statement:

- The ENTRY statement can appear in an external subprogram or a module subprogram; it may not appear in an internal subprogram. If the ENTRY statement appears in a function subprogram, it defines an additional function; if it appears in a subroutine subprogram, it defines an additional subroutine. The entry points thus defined can be referenced in the same way as for a normal function name or subroutine name, as appropriate. Execution starts at the ENTRY statement, and continues in the normal manner, ignoring any ENTRY statements subsequently encountered, until a RETURN statement or the end of the procedure is reached.
- The RESULT *(result-name)* clause can only be specified when the ENTRY statement is included in a function subprogram. If specified, *result-name* must differ from *entry-name*, and *entry-name* must not appear in any specification statement in the scoping unit of the function subprogram; *entry-name* assumes all the attributes of *result-name*. The RESULT clause in an ENTRY statement has the same syntax and semantics as in a FUNCTION statement.
- If the ENTRY statement appears in a function, the result variable is that specified in the FUNCTION statement; if none is specified, the result variable is *entry-name*.
- If the characteristics of the result variable specified in the ENTRY statement are the same as those of the result variable specified in the FUNCTION statement, then the result variable is the same, even though the names are different. If the characteristics are different, then the result variables must be:
  - — Nonpointer scalars of intrinsic type
  - — Storage associated
  - — If any is of character type, they must all be of character type and must all have the same length. If any is of noncharacter type, they must all be of noncharacter type.

- The result variable may not appear in a COMMON, DATA, or EQUIVALENCE statement. Also, the result variable may not have the ALLOCATABLE, INTENT, OPTIONAL, PARAMETER, or SAVE attribute.
- If RECURSIVE is specified on the FUNCTION statement at the start of a function subprogram, and RESULT is specified on an ENTRY statement within the subprogram, then the interface of the function defined by the ENTRY statement is explicit within the function subprogram; the function can thus be invoked recursively. (Note that the keyword RECURSIVE is not given on the ENTRY statement, but only on the FUNCTION statement.)
- If RECURSIVE is specified on the SUBROUTINE statement at the start of a subroutine subprogram, the interface of the subroutine defined by an ENTRY statement within the subprogram is explicit within the subprogram; the subroutine can thus be called recursively.
- A dummy argument in an ENTRY statement must not appear in an executable statement preceding the ENTRY statement, unless it also appears in a FUNCTION, SUBROUTINE, or ENTRY statement preceding the executable statement.
- If a dummy argument in a subprogram—that is, as specified in a FUNCTION or SUBROUTINE statement at the start of the subprogram or in any ENTRY statements within the subprogram—is used in an executable statement, then the statement may only be executed if the dummy argument appears in the dummy argument list of the procedure name actually referenced in the current call. The same restrictions apply when you use a dummy argument in a specification expression to specify an array bound or character length.
- A procedure defined by an ENTRY statement may be given an explicit interface by use of an INTERFACE block. The procedure header in the interface body must be a FUNCTION statement for an entry to a function subprogram, and a SUBROUTINE statement for an entry to a subroutine subprogram.

The ENTRY statement was often used in FORTRAN 77 programs in situations where a set of subroutines or functions had slightly different dummy argument lists but entailed computations involving identical data and code. In Fortran 95 the use of the ENTRY statement in such situations can be replaced by the use of optional arguments.

Examples

The following example defines a subroutine subprogram with two dummy arguments. The subprogram also contains an ENTRY statement that takes only the first dummy argument specified in the SUBROUTINE statement.

```
SUBROUTINE Full_Name (First_Name, Surname)
CHARACTER(20) :: First_Name, Surname
...
ENTRY Part_Name (First_Name)
```

The following example creates a stack. It shows the use of ENTRY to group the definition of a data structure together with the code that accesses it, a technique known as encapsulation. (This example could alternatively be programmed as a module, which would be preferable in that it does not rely on storage association.)

```
SUBROUTINE manipulate_stack
 IMPLICIT NONE
 INTEGER size, top /0/, value
 PARAMETER (size = 100)
 INTEGER, DIMENSION(size) :: stack
 SAVE stack, top

C Push value onto the stack
 ENTRY push(value)
 IF (top == size) STOP 'Stack Overflow'
   top = top + 1
   stack(top) = value
 RETURN

C Pop the top of the stack and place in Value
 ENTRY pop(value)
   IF (top == 0) STOP 'Stack Underflow'
   value = stack(top)
   top = top - 1
   RETURN
 END
```

Here are examples of CALL statements associated with the preceding example:

```
CALL push(10)
CALL push(15)
CALL pop(I)
CALL pop(J)
```

Related Statements

FUNCTION, SUBROUTINE, and CALL

Related Concepts

Subprograms and entry points are discussed in Chapter 7, Program Units and Procedures, as are dummy arguments and recursion.

EQUIVALENCE

Associates different objects with same storage area.

```
EQUIVALENCE (equivalence-list1)
 [, (equivalence-list2)]...
```

equivalence-list is a comma-separated list of two or more object names to be storage associated. Objects can include simple variables, array elements, array names, and character substrings.

Description

All objects in each *equivalence-list* share the same storage area. Such objects become storage associated and are equivalenced to each other. Equivalencing may also cause other objects to become storage associated.

The following items must not appear in *equivalence-list*:

- Automatic objects, including character variables whose length is specified with a nonconstant

- Allocatable arrays
- Function names, result names, or entry names
- Dummy arguments
- Records or record field references
- Nonsequence derived type objects
- Structure components
- Pointers or structures containing pointers
- Named constants

The following restrictions apply to objects that can appear in an `EQUIVALENCE` statement:

- Objects in the same *`equivalence-list`* must be explicitly or implicitly declared in the same scoping unit.
- The name of an equivalenced object must not be made available by use association.

The Fortran 95 standard imposes the following type restrictions on equivalenced objects:

- If one of the objects in *`equivalence-list`* is of type default integer, default real, double precision real, default complex, double complex, default logical, or numeric sequence type, then all objects in *`equivalence-list`* must be one of these types.

  Intel Fortran relaxes this restriction and allows character and noncharacter items to be equivalenced. Note, however, that use of this extension can impact portability.
- If one of the objects in *`equivalence-list`* is of derived type that is not a numeric sequence or character sequence type, then all objects in *`equivalence-list`* must be of the same type.
- If one of the objects in *`equivalence-list`* is of intrinsic type other than default integer, default real, double precision real, default complex, double complex, default logical, or default character, then all objects in *`equivalence-list`* must be of the same type with the same kind type parameter value.

  Intel Fortran relaxes this restriction.

The EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If an array and a scalar share the same storage space through the EQUIVALENCE statement, the array does not have the characteristics of a scalar and the scalar does not have the characteristics of an array. They only share the same storage space.

Care should be taken when data types of different sizes share the same storage space, because the EQUIVALENCE statement specifies that each data item in *equivalence-list* has the same first storage unit. For example, if a 4-byte integer variable and a double-precision variable are equivalenced, the integer variable shares the same space as the 4 most significant bytes of the 8-byte double-precision variable.

Proper alignment of data types is always enforced. The compiler will issue a diagnostic if incorrect alignment is forced through an EQUIVALENCE statement. For data type alignment rules, see Chapter 3, Data Types and Data Objects.

The lengths of the equivalenced objects need not be the same.

Equivalencing Character Data

An EQUIVALENCE statement specifies that the storage sequences of character data items whose names are specified in *equivalence-list* have the same first character storage unit. This causes the association of the data items in *equivalence-list* and can cause association of other data items as well. Consider the following example:

```
CHARACTER(LEN=4) :: a, b
CHARACTER(LEN=3) :: c(2)
EQUIVALENCE (a, c(1)), (b, c(2))
```

As a result of this EQUIVALENCE statement, the fourth character in a, the first character in b, and the first character in c(2) share the same storage.

Strings of the same or different lengths can be equivalenced to start on the first element, and you can use substring notation to specify other associations, as in the following:

```
CHARACTER (10) :: s1, s2
EQUIVALENCE (s1(2:2), s2(3:3)
```

Substring subscripts must be integer initialization expressions, and the substring length must be nonzero.

Equivalencing Arrays

To determine equivalence between arrays with different dimensions, Intel Fortran views all elements of an array in linear sequence. Each array is stored as if it were a one-dimensional array. Array elements are stored in ascending sequential, column-major order; for information about how arrays are laid out in memory, see Chapter 4, Arrays.

Array elements can be equivalenced with elements of a different array or with scalars. No equivalence occurs outside the bounds of any of the equivalenced arrays.

If equivalenced arrays are not of the same type, they may not line up element by element.

If an array name appears without subscripts in an `EQUIVALENCE` statement, it has the same effect as specifying an array name with the subscript of its first element.

It is illegal to equivalence different elements of the same array to the same storage area. For example, the following is illegal:

```
INTEGER :: a(2), b
EQUIVALENCE (a(1), b), (a(2), b)
```

Likewise, it is illegal to use the `EQUIVALENCE` statement to force consecutive array elements to be noncontiguous, as in the following example:

```
REAL :: a(2), r(3)
EQUIVALENCE (a(1), r(1)), (a(2), r(3))
```

Array subscripts must be integer initialization expressions.

Equivalence in Common Blocks

An `EQUIVALENCE` statement must not cause two common blocks to be associated.

You can use the `EQUIVALENCE` statement to place objects in common by equivalencing them to objects already in common. If one element of an array is equivalenced to an object in common, the whole array is placed in

common with equivalence maintained for storage units preceding and following the data element in common. The common block is always extended when it is necessary to fit an array that shares storage space in the common block. It may be extended after the last entry, but not before the first.

Consider the following example, which puts array `i` in blank common and equivalences array element `j(2)` to `i(3)`:

```
INTEGER :: i(6), j(6)
COMMON i
EQUIVALENCE (i(3), j(2))
```

The effect of the `EQUIVALENCE` statement is to extend blank common to include element `j(6)`. This is entirely legal because the extension occurs at the end of the common block.

But if the `EQUIVALENCE` statement were changed as follows:

```
EQUIVALENCE (i(1), j(2)) ! illegal
```

it would result in an illegal equivalence, because storage would have to be inserted in front of the block in order to accommodate element `j(1)`.

Example

In the following example, the variables `a`, `b`, and `c` share the same storage space; array elements `d(2)` and `e(5)` share the same storage space; variables `f`, `g`, and `h` share the same storage:

```
INTEGER :: a, b, c, d(20), e(30), f, g, h
EQUIVALENCE (a, b, c), (d(2), e(5)), (f, g, h)
```

Related Statements

`COMMON`

NOTE. *You cannot equivalence items in dynamic* `COMMON`.

Related Concepts

The following are discussed elsewhere in this manual:

- Storage association: Chapter 3, Data Types and Data Objects
- Arrays: Chapter 4, Arrays

EXIT

Terminates a DO loop.

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

`do-construct-name` is the name given to the `DO` construct. If do-construct-name is specified, it must be the name of a `DO` construct that contains the `EXIT` statement.

Description

If you do not specify *`do-construct-name`*, the `EXIT` statement terminates the immediately enclosing `DO` loop. If you do specify it, the `EXIT` statement terminates the enclosing `DO` loop with the same name.

Example

```
DO i = 1, 20
 n(i) = 0
 READ *, j
 IF (j < 0) EXIT
 n(i) = j
END DO
```

Related Statements

`CYCLE` and `DO`

Related Concepts

For information about the DO construct and flow control statements, see Chapter 6, Execution Control.

EXTERNAL (Statement and Attribute)

Declares a name to be external.

A type declaration statement with the EXTERNAL attribute is:

type , attrib-list :: *function-name-list*

type	is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (name), etc.), as described in Chapter 3, Data Types and Data Objects.
attrib-list	is a comma-separated list of attributes including EXTERNAL and optionally those attributes compatible with it, namely:

OPTIONAL PRIVATE PUBLIC

function-name-list	is a comma-separated list of function names to be designated EXTERNAL.

The syntax of the EXTERNAL statement is:

EXTERNAL *external-name-list*

Note that the syntax of the EXTERNAL statement does not permit optional colons.

Description

An EXTERNAL attribute or statement specifies that a name may be used as an actual argument in subroutine calls and function references. The name is either an external procedure, a dummy procedure, or a block data program unit.

A name that appears in a type statement specifying the EXTERNAL attribute must be the name of an external procedure or of a dummy argument that is a procedure.

The following rules and restrictions apply:

- A name can appear once in an EXTERNAL statement, in a declaration statement with an EXTERNAL attribute, or in an interface body, but not in more than one of these.
- If the name is a dummy argument, an EXTERNAL statement declares it to be a dummy procedure.
- If a user intrinsic procedure has the same name as an external procedure, then it must either be declared to have the EXTERNAL attribute or have an explicit interface. The named intrinsic procedure is then no longer available in such program units.
- The INTRINSIC and EXTERNAL attributes are mutually exclusive.

Examples

```
SUBROUTINE sub (fourier)
! fourier is a dummy procedure. The actual
!  argument corresponding to fourier could be
!  an external, an intrinsic, or a module
!  procedure.
  REAL fourier
  EXTERNAL fourier
REAL, EXTERNAL :: SIN, COS, TAN
! The preceding statement means that SIN, COS, and
! TAN are no longer intrinsic procedures.
! Functions with these names must be defined in the
! program.
...
END SUBROUTINE sub
SUBROUTINE gratx (x, y)
EXTERNAL init_block_a
! Specify init_block_a as the block data sub-
!  program that initializes common block a.
COMMON /a/ temp, pressure
```

```
! Common block available in subroutine 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
```

Related Statements

`INTRINSIC`

Related Concepts

Module procedures, interfaces, and interface blocks are described in Chapter 7, Program Units and Procedures.

FORMAT

Describes how I/O data is to be formatted.

label `FORMAT (`*format-list*`)`

label	is a statement label.
format-list	is a comma-separated list of format items, where each item in the list can be either one of the edit descriptors described in Chapter 9, I/O Formatting or (format-list). If format-list is one of the list items, it may be optionally preceded by a repeat specification—a positive integer that specifies how may times format-list is to be repeated. Many of the edit descriptors may also be repeated; see Chapter 9, I/O Formatting for more information.

Description

The FORMAT statement holds the format specification that indicates how data in formatted I/O is to be translated between internal (binary) representation and formatted (ASCII) representation. The translation makes it possible to represent data in a humanly readable format.

Although a format specification can be embedded within a data transfer statement, the point to using a FORMAT statement is to make it available to any number of data transfer statements. Several data transfer statements can use the same format specification contained in a FORMAT statement by referencing *label*.

Another advantage of the FORMAT statement over the use of embedded format specifications is that it is "pre-compiled", reducing the runtime overhead of processing the format specification and providing compile-time error checking of the FMT= specifier.

Examples

```
  PROGRAM format_example

  WRITE (15,FMT=20) 1234, 45, -12
20 FORMAT (I6, 2I4)
  END
```

When compiled and executed, this program outputs the following (where ~~b~~ represents the blank character):

```
~~bb~~1234~~bb~~45~~b~~ -12
```

Related Statements

READ and WRITE

Related Concepts

Statement labels are described in Chapter 2, Language Elements. For information about I/O formatting, see Chapter 9, I/O Formatting.

FUNCTION

Introduces a function subprogram.

```
[RECURSIVE] [type-spec] FUNCTION
      function-name ([dummy-arg-name-list])
     [RESULT (result-name)]
```

RECURSIVE
: is a keyword that must be specified in the FUNCTION statement if the function is either directly or indirectly recursive. The RECURSIVE clause can appear at most once, either before or after *type-spec*. It is not an error to specify RECURSIVE for a nonrecursive function.

  A recursive function that calls itself directly must also have the RESULT clause specified (see *result-name*, below).

type-spec
: is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (name), etc.), as described in Chapter 3, Data Types and Data Objects. The type and type parameters of the function result can be specified by type-spec or by declaring the result variable within the function subprogram, but not by both. The implicit typing rules apply if the function is not typed explicitly.

  If the function result is array-valued or a pointer, the appropriate attributes for the result variable (which is *function-name*, or result-name if specified) must be specified within the function subprogram.

function-name
: is the name of the function subprogram being defined.

dummy-arg-name-list
: is a comma-separated list of dummy argument names for the function.

result-name is the result variable. If the RESULT clause is not specified, function-name becomes the result variable. If result-name is given, it must differ from function-name, and function-name must not then be declared within the function subprogram.

As noted above, a recursive function that calls itself directly must have the RESULT clause specified. Other functions may have a RESULT clause.

Description

A FUNCTION statement introduces an external, module, or internal function subprogram.

Example

```
PROGRAM p
CONTAINS
  ! f is an internal function. In FORTRAN 77
  !  this could have been a statement function
  !  (also valid in Fortran 95).
  FUNCTION f(x)
     f = 2*x + 3
  END FUNCTION f

  RECURSIVE INTEGER FUNCTION factorial (n) &
   RESULT (factorial_value)
  ! A recursive function, which must
  !  therefore specify a RESULT clause.
  IMPLICIT INTEGER (a-z)
   IF (n <= 0) THEN
     factorial_value = 1
   ELSE
     factorial_value = n * factorial (n-1)
   END IF
  END FUNCTION factorial
END PROGRAM p
```

Related Statements

END, INTENT, INTERFACE, OPTIONAL, and the type declaration statements

Related Concepts

The following are described elsewhere in this manual:

- Data types: Chapter 3, Data Types and Data Objects.
- Defined operators: Chapter 3, Data Types and Data Objects.
- Expressions (a function reference is a primary in an expression): Chapter 5, Expressions and Assignment.
- External functions, module function, internal functions, recursive functions, and function invocation: Chapter 7, Program Units and Procedures.

GO TO (Assigned)

Transfers control to a variable that was assigned a label.

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

integer-variable	is a scalar variable of default type integer.
label-list	is a list of statement labels, separated by commas.

Description

The assigned GO TO statement transfers control to the statement whose label was most recently assigned to a variable with the ASSIGN statement.

integer-variable must be given a label value of an executable statement through an ASSIGN statement prior to execution of the GO TO statement. When the assigned GO TO statement is executed, control is transferred to the statement whose label matches the label value of *integer-variable*.

label-list is a list of labels that *integer-variable* might assume.

integer-variable must not be an array element or an integer component of a structure.

The use of this statement can hinder the ability of the compiler to optimize the program in which it occurs.

Example

```
ASSIGN 10 TO out
GO TO out
```

Related Statements

ASSIGN, GO TO (computed), and GO TO (unconditional)

Related Concepts

For additional information about the assigned GO TO and other flow control statements, see Chapter 6, Execution Control.

GO TO (Computed)

Transfers control to one of several labels.

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

label-list	is a list of statement labels, separated by commas.
arithmetic-expression	is a scalar integer expression. As an extension, Intel Fortran also allows the expression to be of type real or double precision.

Description

The computed GO TO statement transfers control to one of several labeled statements, depending on the value of *arithmetic-expression*. After *arithmetic-expression* is evaluated (and, if necessary, truncated to

an integer value), control transfers to the statement label whose position in *label-list* corresponds to the truncated value of *arithmetic-expression*.

If the value of *arithmetic-expression* is less than 1 or greater than the total number of labels in *label-list*, control transfers to the executable statement immediately following the computed GO TO statement.

Example

```
index = 3
GO TO (10, 20, 30, 40) index
! Branch made to the statement labeled 30.
```

Related Statements

SELECT CASE, GO TO (assigned), and GO TO (unconditional)

Related Concepts

For more information about the computed GO TO statement and other flow control statements, see Chapter 6, Execution Control.

GO TO (Unconditional)

Transfers control to a specified label.

GO TO *label*

label is the label of an executable statement.

Description

The unconditional GO TO statement transfers control directly to the statement at the specified label. The executable statement with *label* can occur before or after the GO TO statement, but it must be within the same scoping unit.

Example

```
  GO TO 30
30 CONTINUE
```

Related Statements

`GO TO` (assigned) and `GO TO` (computed)

Related Concepts

For more information about the unconditional `GO TO` statement and other flow control statements, see Chapter 6, Execution Control.

IF (Arithmetic)

Transfers control to one of three labels.

`IF (`*`arithmetic-expression`*`)` *`labelN, labelZ, labelP`*

`arithmetic-expression`	is an arithmetic expression of any numeric type except complex and double complex.
`label`	is a label of an executable statement.

Description

The arithmetic `IF` statement transfers control to the statement whose label is determined by *`arithmetic-expression`*. If *`arithmetic-expression`* evaluates to a negative value, control transfers to *`labelN`*; if it evaluates to 0, control transfers to *`labelZ`*; and if it evaluates to a positive value, control transfers to *`labelP`*.

The same label may appear more than once in the same arithmetic `IF` statement.

Each label must be that of an executable statement in the same scoping unit as the arithmetic `IF`.

Example

```
i = -1
IF (i) 10, 20, 30
! Branch made to the statement labeled 10.
```

Related Statements

`IF` (construct) and `IF` (logical)

Related Concepts

For more information about the arithmetic `IF` statement and other flow control statements, see Chapter 6, Execution Control.

IF (Block)

Begins an IF construct.

`[`*`construct-name`* `:] IF (`*`logical-expression`*`) THEN`

`construct-name` is the name given to the `IF` construct. If construct-name is specified, the same name must also appear in the `END IF` statement.

`logical-expression` is a scalar logical expression.

Description

The `IF` statement executes the immediately following statement block if *`logical-expression`* evaluates to true.

The `IF` construct, which the `IF` statement begins, may include `ELSE IF` statements and an `ELSE` statement to provide alternate statement blocks for execution.

The block following the `IF` statement may be empty.

As an extension, Intel Fortran allows the transfer of control into an `IF` construct from outside the construct.

Example

```
IF (x <= 0.0 .AND. y > 1.0) THEN
 CALL fix_coord(x, y)
END IF
```

Related Statements

`ELSE`, `ELSE IF`, `IF` (arithmetic), `IF` (logical), and `END` (construct)

Related Concepts

For more information about the `IF` construct, see Chapter 6, Execution Control.

IF (Logical)

Conditionally executes a statement.

`IF (`*`logical-expression`*`)` *`statement`*

`logical-expression` is a logical expression.

`statement` is any executable statement other than the following:

- A statement used to begin a construct
- Any `END` statement
- Any `IF` statement

Description

The logical `IF` statement is a two-way decision maker. If *`logical-expression`* evaluates to is true, *`statement`* executes and control passes to the next statement. If *`logical-expression`* evaluates to false, *`statement`* does not execute and control passes to the next statement in the program.

Example

```
IF (a .EQ. b) PRINT *, 'They are equal.'
```

Related Statements

`IF` (arithmetic) and `IF` (construct)

Related Concepts

For more information about the logical `IF` statement and other flow control statements, see Chapter 6, Execution Control.

IMPLICIT

Changes or voids default typing rules.

The `IMPLICIT` statement can take either of the following forms:

```
IMPLICIT type (range-list) [, type (range-list) ,]...
IMPLICIT NONE
```

`type`	is the data type to be associated with the corresponding letters in range-list.
`range-list`	is a comma-separated list of letters or ranges of letters (for example, `A-Z` or `I-N`) to be associated with *`type`*. Writing a range of letters has the same effect as writing a list of single letters.

Description

The `IMPLICIT` statement can be used either to change or void the default typing rules, depending on which of the two forms the statement takes.

First Form

This form of the `IMPLICIT` statement specifies *`type`* as the data type for all variables, arrays, named constants, function subprograms, `ENTRY` names in function subprograms, and statement functions that begin with any letter in *`range-list`* and that are not explicitly given a type.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements, except possibly the DATA and PARAMETER statements.

The same letter must not appear as a single letter or be included in a range of letters, more than once in all of the IMPLICIT statements in a scoping unit.

For information on how the IMPLICIT and PARAMETER statements interact, refer to the description of the PARAMETER in this chapter.

Second Form

The IMPLICIT NONE statement disables the default typing rules for all variables, arrays, named constants, function subprograms, ENTRY names, and statement functions (but not intrinsic functions). All such objects must be explicitly typed. The IMPLICIT NONE statement must be the only IMPLICIT statement in the scoping unit, and it must precede any PARAMETER statement. Types of intrinsic functions are not affected.

You can also use the -implicit_none command-line option to void the default typing rules. A program compiled with this option may include IMPLICIT statements, which the compiler will honor. For additional information about the -implicit_none option, see the *Intel Fortran Compiler User's Guide*.

Example

The following statement causes all variables and function names beginning with I, J, or K to be of type complex, and all data items beginning with A, B, or C to be of type integer:

```
IMPLICIT COMPLEX (I, J, K), INTEGER (A-C)
```

Related Concepts

The default typing rules and the behavior of the IMPLICIT and IMPLICIT NONE statements are discussed in Chapter 3, Data Types and Data Objects. The -implicit_none command-line option is described in the *Intel Fortran Compiler User's Guide*.

IMPLICIT AUTOMATIC

Defaults typing to automatic variable.

The IMPLICIT AUTOMATIC statement takes the following form:

```
IMPLICIT AUTOMATIC (range-list) [, type (range-list)
,]...
```

range-list is a comma-separated list of letters or ranges of letters (for example, A-Z or I-N) to be associated with *type*. Writing a range of letters has the same effect as writing a list of single letters.

Description

The IMPLICIT AUTOMATIC statement is used to make the type variable automatic, that is, a copy is generated each time you invoke the procedure.

Related Concepts

The default typing rules and the behavior of the IMPLICIT are discussed in Chapter 3, Data Types and Data Objects. The -implicit_none command-line option is described in the *Intel Fortran Compiler User's Guide*. Also, for a general description, see the "IMPLICIT". Also see "AUTOMATIC".

IMPLICIT STATIC

Defaults typing to a static variable.

The IMPLICIT STATIC statement takes the following form:

```
IMPLICIT STATIC (range-list) [, type (range-list)
,]...
```

range-list is a comma-separated list of letters or ranges of letters (for example, A-Z or I-N) to be associated with type. Writing a range of letters has the same effect as writing a list of single letters.

Description

The IMPLICIT STATIC statement is used to make the type variable static, that is, a one and only one copy of the data is kept regardless of the number of times a procedure is called.

Related Concepts

The default typing rules and the behavior of the IMPLICIT statements are discussed in Chapter 3, Data Types and Data Objects. The -implicit_none command-line option is described in the *Intel Fortran Compiler User's Guide*. Also, for a general description, see the "IMPLICIT". Also see "STATIC (Statement and Attribute)".

INCLUDE

Imports text from a specified file.

INCLUDE *character-literal-constant*

character-literal-constant is the name of the file to include.

Description

The keyword INCLUDE and *character-literal-constant* form an INCLUDE line, which is used to insert text into a program prior to compilation. The inserted text replaces the INCLUDE line; the INCLUDE line should therefore appear in your program where you want the inserted text. When the end of an included file is reached, the compiler continues processing with the line following the INCLUDE line.

character-literal-constant can be either a file name or a device name. It must not have a kind parameter that is a named constant.

The INCLUDE line must appear on one line with no other text except possibly a trailing comment. It must not have a statement label. Thus, you cannot branch to it, and it cannot be an action statement that is part of a Fortran 95 IF statement. You cannot use the ";" operator to add a second INCLUDE line, nor can you use the "&" operator to continue it over another line.

The compiler searches directories for the named include files in the following order:

1. The current directory
2. Directories specified by the /I command-line option, in the order specified
3. The directories specified with the INCLUDE environment variable

See the *Intel Fortran Compiler User's Guide* for information about the /I option.

INCLUDE lines can be nested to a maximum of ten levels. However, they must be nested nonrecursively. That is, inserted text must not specify an INCLUDE line that was encountered at an earlier level of nesting.

Line numbering within the listing of an included file begins at 1. When the included file listing ends, the include level decreases appropriately, and the previous line numbering resumes.

Example

```
INCLUDE 'my_common_blocks'
INCLUDE "/my_stuff/declarations.h"
```

INQUIRE

Returns information about file properties.

The syntax of the INQUIRE statement has two forms:

- Inquiry by output list:
  INQUIRE (IOLENGTH= *integer-variable*) *output-list*

- Inquiry by unit or file:
  INQUIRE (*io-specifier-list*)

integer-variable is the length of the unformatted record that would result from writing output-list to a direct-access file. The value returned in integer-variable can be used with the RECL= specifier in an OPEN statement to specify the length of each record in an unformatted direct-access file that will hold the data in output-list.

output-list is a comma-separated list of data items, similar to what would be included with the WRITE or PRINT statement. The data items can include variables and implied-DO lists; see Chapter 8, I/O and File Handling for more information.

io-specifier-list is a list of comma-separated I/O specifiers. As noted in the following descriptions, most of the specifiers return information about the specified unit or file. *io-specifier-list* must include either the UNIT= or FILE= specifier, but not both. The following paragraphs describe all the I/O specifiers that can appear in io-specifier-list:

[UNIT=]*unit* specifies the unit connected to an external file. *unit* must be an integer expression that evaluates to a number greater than 0. If the optional keyword UNIT= is omitted, *unit* must be the first item in *io-specifier-list*. If *unit* appears in *io-specifier-list*, the FILE= specifier must not be used.

ACCESS=*character-variable* returns the following values, indicating the method of access:

'SEQUENTIAL'	File is connected for sequential access.
'DIRECT'	File is connected for direct access.
'UNDEFINED'	File is not connected.

ACTION=*character-variable* returns the following values, indicating the direction of the transfer:

'READ'	File is connected for reading only.
'WRITE'	File is connected for writing only.
'READWRITE'	File is connected for reading and writing.
'UNDEFINED'	File is not connected.

BINARY=*bin* *bin* is a scalar default CHARACTER variable that is assigned one of the following values:

'YES'	File is connected to a binary file.
'NO'	File is not connected to a binary file.
'UNKNOWN'	It cannot be determined whether or not file is connected to a binary file.

BLANK=*character-variable* returns the type of blank control that is in effect. For information about blank control, see the BLANK= specifier for the OPEN statement. The values returned by the BLANK= specifier are:

'NULL'	Null blank control is in effect.
'ZERO'	Zero blank control is in effect.
'UNDEFINED'	File is not connected for formatted I/O.

BLOCKSIZE=*integer-expression* indicates the physical I/O transfer size for the file. If the value is non-zero, it should be rounded up to a multiple of 512. If it is zero or not specified, it defaults to system default, generally 512.

CARRIAGECONTROL= indicates the type of carriage control used when a

string file is displayed on a terminal device. The string values are:

'FORTRAN'	Default for Fortran interpretation of the first character.
'LIST'	Default for formatted file.
'NONE'	Default for binary and unformatted file.

DELIM=*character-expression* returns the following values, indicating the character to use (if any) to delimit character values in list-directed and namelist formatting:

'APOSTROPHE'	An apostrophe is used as the delimiter.
'QUOTE'	The double quotation mark is used as the delimiter.
'NONE'	There is no delimiting character.
'UNDEFINED'	File is not connected for formatted I/O.

DIRECT=*character-variable* returns the following values, indicating whether or not the file is connected for direct access:

'YES'	File is connected for direct access.
'NO'	File is not connected for direct access.
'UNKNOWN'	It cannot be determined whether or not file is connected for direct access.

ERR=*stmt-label* specifies the label of the executable statement to which control passes if an error occurs during statement execution.

EXIST=*logical-variable* returns the following values, indicating whether or not the file or unit exists:

.TRUE.	File exists or unit is connected.
.FALSE.	File does not exist or unit is not connected.

FILE=*character-expression*
: specifies the name of a file for inquiry. The file does not have to be connected or even exist. If the FILE= specifier appears in *io-specifier-list*, the UNIT= specifier must not be used.

FORM=*character-variable*
: returns the following values, indicating whether the file is connected for formatted or unformatted I/O:

'FORMATTED'	File is connected for formatted I/O.
'UNFORMATTED'	File is connected for unformatted I/O.
'UNDEFINED'	File is not connected.
'BINARY'	File is connected for binary transfer.

FORMATTED=*character-variable*
: returns the following values, indicating whether or not the file is connected for formatted I/O:

'YES'	File is connected for formatted I/O.
'NO'	File is not connected for formatted I/O.
'UNKNOWN'	It cannot be determined whether or not file is connected for formatted I/O.

IOFOCUS=*logical-variable*
: returns the following values indicating whether the specified UNIT is the current active window:

.TRUE.	Specified UNIT is the current active window in a QuickWin application.
.FALSE.	Specified UNIT is not the current active window.

IOSTAT=*integer-variable* returns the I/O status after the statement executes. If the statement successfully executes, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.

NAME=*character-variable* returns the name of file connected to the specified unit. If the file has no name or is not connected, NAME= returns the string UNDEFINED.

NAMED=*logical-variable* returns the following values, indicating whether or not the file has a name:

.TRUE.	File has a name.
.FALSE.	File does not have a name.

NEXTREC=*integer-variable* returns the number of the next record to be read or written in a file connected for direct access. The value is the last record read or written +1. A value of 1 indicates that no records have been processed. If the file is not connected or it is a device file or its status cannot be determined, *integer-variable* is undefined.

NUMBER=*integer-variable* returns the unit number that is connected to the specified file. If no unit is connected to the named file, *integer-variable* is undefined.

OPENED=*logical-variable* returns the following values, indicating whether or not the file has been opened (that is, is connected):

.TRUE.	File is connected.
.FALSE.	File is not connected.

ORGANIZATION=*scalar-characer*
returns a scalar character variable indicating the following record access types:

'SEQUENTIAL'	File is connected for a sequential access, records are accessed in order.
'RELATIVE'	File is connected for a direct access, records can be accessed in any order.
'UNKNOWN'	It cannot be determined whether the file is connected for sequential or relative access.

PAD=*character-variable*
returns a value indicating whether or not input records are padded with blanks. For more information about padding, see the PAD= specifier for the OPEN statement. The return values are:

'YES'	File or unit is connected with PAD='YES' in OPEN statement.
'NO'	File or unit is connected with PAD='NO' in OPEN statement.

POSITION=*character-variable*
returns the following values, indicating the file position:

'REWIND'	File is connected with its position at the start of the first record.
'APPEND'	File is connected with its position at the end-of-file record.
'ASIS'	File is connected without changing its position.
'UNDEFINED'	File is not connected or is connected for direct access.

READ=*character-variable* returns the following values, indicating whether or not reading is an allowed action for the file:

'YES'	Reading is allowed for file.
'NO'	Reading is not allowed for file.
'UNKNOWN'	It cannot be determined whether or not reading is allowed for file.

READWRITE=*character-variable*
returns the following values, indicating whether or not reading and writing are allowed actions for the file:

'YES'	Both reading and writing are allowed for file.
'NO'	Reading and writing are not both allowed for file.
'UNKNOWN'	It cannot be determined whether or not reading and writing are both allowed for file.

RECL=*integer-variable* returns the record length of the specified unit or file, measured in bytes. The file must be a direct-access file. If the file is not a direct-access file or does not exist, *integer-variable* is undefined.

RECORDTYPE=*scalar-character-expression*
returns a scalar default variable *rtype* of default CHARACTER type with one of the following values:

'FIXED'	File is connected for a fixed-length record.
'VARIABLE'	File is connected for a variable-length record.
'SEGMENTED'	File is connected for unformatted sequential access with segmented records.

'STREAM'	File is connected without record termination.
'STREAM_CR'	File is connected with its records terminated with carriage return.
'STREAM_LF'	File is connected with its records terminated with line feed.
'UNKNOWN'	File is not connected.

SEQUENTIAL=*character-variable*
returns the following values, indicating whether or not the file is connected for direct access:

'YES	File is connected for sequential access.
'NO'	File is not connected for sequential access.
'UNKNOWN'	It cannot be determined whether or not file is connected for sequential access.

SHARE=*character-variable* indicates whether the file locking is applied while the unit is open. The following values are used:

'DENYRW'	Deny-read/write mode. No process can open this file.
'DENYWR'	Deny-write mode. No process can open the file with write access.
'DENYRD'	Deny-read mode. No process can open the file with read access.
'DENYNONE'	Deny-none mode. Any process can open the file in any mode. This is the default value.
'UNDEFINED'	The access mode is undefined.

UNFORMATTED=*character variable*
: returns the following values, indicating whether or not the file is connected for formatted I/O:

'YES'	File is connected for unformatted I/O.
'NO'	File is not connected for unformatted I/O.
'UNKNOWN'	It cannot be determined whether or not file is connected for unformatted I/O.

WRITE=*character-variable*
: returns the following values, indicating whether or not writing is an allowed action for the file:

'YES'	Writing is allowed for file.
'NO'	Writing is not allowed for file.
'UNKNOWN'	It cannot be determined whether or not writing is allowed for file.

Description

The INQUIRE statement returns selected properties of a specified file or unit number. (It is illegal to include both the UNIT= specifier and the FILE= specifier in the same INQUIRE statement.) Inquiring by unit number should be used on connected files; inquiring by filename is typically used on unconnected files.

In addition, the INQUIRE statement can also be used to determine the record length of a new or existing file. That is, you can use INQUIRE to obtain the record length before creating the file and then use the return value as the argument to the RECL= specifier in an OPEN statement.

Examples

The examples in this section illustrate different uses of the INQUIRE statement.

Inquiry by File

The following statement returns the following information about the file named my_file:

- Is it connected?
- Is it connected for direct access?
- Can it be read and written?

```
LOGICAL :: exist
CHARACTER(LEN=9) :: dir_acc, rw_sts
INQUIRE (FILE='my_file', EXIST=exist, &
     DIRECT=dir_acc, READWRITE=rw_sts)
```

Inquiry by Unit

The following INQUIRE statement returns the following information about the file connected to the unit in u_num:

- Is there a file connected to u_num?
- Is it named file or a scratch file?
- What is the name?

```
LOGICAL :: opened, named
INTEGER :: u_num
CHARACTER(LEN=80) :: fname
.
.
.
INQUIRE (UNIT=u_num, NAMED=named, &
     OPENED=opened, NAME=fname)
```

Inquiry by Output List

When using the OPEN statement to create a direct-access file, you must specify the record length for the file with the RECL= specifier. Previous to Fortran 95, you had to resort to a nonportable strategy to determine record length. The Fortran 95 INQUIRE statement provides a portable solution: use the INQUIRE statement to inquire by output list, and specify the return value from the INQUIRE statement as the argument to the OPEN statement.

The following is an example:

```
INTEGER :: rec_len, ios
INQUIRE (IOLENGTH=rec_len) x, y, i, j
OPEN (UNIT=32, FILE='new_file', IOSTAT=ios, &
   ACCESS='DIRECT', RECL=rec_len)
```

Related Statements

OPEN

Related Concepts

For information about I/O concepts, see Chapter 8, I/O and File Handling.

INTEGER

Declares entities of type integer.

`INTEGER [`*`kind-spec`*`] [[, `*`attrib-list`*`] ::] `*`entity-list`*

kind-spec	is the kind type parameter that specifies the range of the entities in entity-list. *kind-spec* takes the form: ([KIND=] *kind-param*)
where *kind-param*	can be a named constant or a constant expression that has the integer value of 1, 2, 4, or 8. The size of the default type is 4.

As an extension, *kind-spec* can take the form:
**len-param*

where *len-param*	is the integer 1, 2, 4, or 8 (default = 4).
attrib-list	is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC

EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

For information about each of the attributes, see the corresponding statement in this chapter.

entity-list is a list of entities, separated by commas. Each entity takes the form:

name [(array-spec)] [= initialization-expr]

where *name* is the name of a variable or function, *array-spec* is a comma-separated list of dimension bounds, and *initialization-expr* is the initial value for the entity.

Description

The INTEGER statement is used to declare the length and properties of data that are whole numbers. A kind parameter (if present) indicates the representation method.

The INTEGER statement is constrained by the rules for all type declaration statements, including the requirement that it precede all executable statements.

Explicitly declaring an entity with the INTEGER statement overrides any implicit typing rules in effect.

If *attrib-list* or *initialization-expr* appear in the declaration, *entity-list* must be preceded by the double colon.

If *array-spec* is specified for an entity, it overrides any DIMENSION attribute.

Initialization

initialization-expr must be a constant integer expression that can be evaluated at compile time.

The following entities may not be initialized:

- Dummy arguments
- Function results

- Allocatable arrays
- Pointers
- External names
- Intrinsic names
- Automatic objects

If *attrib-list* includes the PARAMETER attribute, each entity in *entity-list* must include an initialization expression.

To initialize an array in an INTEGER statement, you must use an array constructor, as in the following example:

```
INTEGER, DIMENSION(4) :: ivec=(/1,2,3,4/)
```

If an array is initialized, all items in the array must be initialized. Implied-DO loops cannot be used to initialize an array in a type declaration statement.

As an extension, an initializer may appear between slashes in a type declaration statement, as follows:

```
INTEGER i/-1/, j/-2/, k/-7/
```

The double colon (::) may not be used with this initialization format.

Length Specification Extension

As a portability extension, Intel Fortran allows the following syntax for specifying the length of an entity:

name[**len*] [(*array-spec*)] [= *initialization-expr*]

If (*array-spec*) is specified, **len* may appear on either side of (*array-spec*).

If *name* appears with **len*, it overrides the length specified by INTEGER**size*. For example, the following statements are equivalent declarations of int1:

```
INTEGER (KIND = 8) int1
INTEGER*4 int1*8
```

Example

The following are valid declarations:

```
INTEGER i, j
INTEGER(KIND=2) :: k
INTEGER(2), PARAMETER :: limit=420
```

Related Statements

BYTE

Related Concepts

The following are discussed elsewhere in this manual:

- Implicit typing rules: Chapter 3, Data Types and Data Objects
- Data representation models: Chapter 3, Data Types and Data Objects
- Storage classes for variables: Chapter 3, Data Types and Data Objects
- Automatic objects: Chapter 3, Data Types and Data Objects
- Arrays: Chapter 4, Arrays
- Expressions: Chapter 5, Expressions and Assignment
- Initialization expressions: Chapter 5, Expressions and Assignment

INTENT (Statement and Attribute)

Specifies the intended use of dummy arguments.

A type declaration statement with the INTENT attribute is:

type , *attrib-list* :: *dummy-arg-name-list*

type	is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.), as described in Chapter 3, Data Types and Data Objects.
attrib-list	is a comma-separated list of attributes including INTENT (*intent-spec*) and optionally those attributes compatible with it, namely: DIMENSION OPTIONAL TARGET
intent-spec	is one of IN, OUT, or INOUT. (The form IN OUT is valid.)

dummy-arg-name-list
is a comma-separated list of subprogram dummy arguments to which intent-spec is to apply.

The syntax of the INTENT statement is:

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

Description

The INTENT attribute declares whether a dummy argument is intended for transferring a value into a procedure, or out of it, or both. The INTENT attribute helps detect the use of arguments inconsistent with their intended use, and may also assist the compiler in generating more efficient code.

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.

If a dummy argument has intent OUT, the corresponding actual argument must be definable; that is, it cannot be a constant. When execution of the procedure begins, the dummy argument is undefined; thus it must be given a value before it is referenced. The dummy argument need not be given a value by the procedure.

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. The dummy argument need not be given a value by the procedure.

The following points should also be noted:

- Intent specifications apply only to dummy arguments and may only appear in the specification part of a subprogram or interface body.
- 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.
- The intent of a pointer dummy argument must not be specified.

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 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)
  ! First subroutine defined above.
  CALL lab_ten (dg, e, f, g+1.0)
END PROGRAM pxx
```

Related Statements

`FUNCTION` and `SUBROUTINE`

Related Concepts

Procedure arguments—including argument association and argument keywords—are discussed in Chapter 7, Program Units and Procedures.

INTERFACE

Introduces an interface block.

`INTERFACE [`*`generic-spec`*`]`

`generic-spec`	is one of: • *`generic-name`* • `OPERATOR(`*`defined-operator`*`)` • `ASSIGNMENT(=)`
`generic-name`	is the name of a generic procedure.
`defined-operator`	is one of: • An intrinsic operator • *`.operator.`*, where *`operator`* is a user-defined name

Description

The `INTERFACE` statement is the first statement of an interface block. Interface blocks constitute the mechanism by which external procedures may be given explicit interfaces and also provide additional functionality, as described below.

The `INTERFACE` *`generic-name`* form defines a generic interface for the procedures in the interface block.

The `INTERFACE OPERATOR (`*`defined-operator`*`)` form is used to define a new operator or to extend the meaning of an existing operator.

The `INTERFACE ASSIGNMENT(=)` form is used to extend the definition of the assignment operator to new combinations of data types, or to redefine the assignment operator for user-defined types.

Examples

The following examples illustrate various forms of interface block:

```
! Make explicit the interfaces of
! external function spline and external
! subroutine sp2.
```

```
INTERFACE
  REAL FUNCTION spline(x,y,z)
  END FUNCTION spline
  SUBROUTINE sp2(x,z)
  END SUBROUTINE sp2
END INTERFACE

! Make the interface of function r_ave
! explicit, and give it the generic name
! g_ave.
INTERFACE g_ave
  FUNCTION r_ave(x)
  ! Get the size of x from the module
  ! ave_stuff.
  USE ave_stuff, ONLY: n
  REAL r_ave, x(n)
  END FUNCTION r_ave
END INTERFACE
! Make the interface of external function b_or
!  explicit, and use it to extend the +
!  operator.
INTERFACE OPERATOR ( + )
  FUNCTION b_or(p, q)
  LOGICAL b_or, p, q
  INTENT (IN) p, q
  END FUNCTION b_or
END INTERFACE
```

Related Statements

END INTERFACE, FUNCTION, and SUBROUTINE

Related Concepts

The following are discussed elsewhere in this manual:

- Derived types: Chapter 3, Data Types and Data Objects
- Assignment: Chapter 5, Expressions and Assignment
- Procedures, generic procedures, procedure interfaces, and user defined operators: Chapter 7, Program Units and Procedures

INTERFACE TO

A block to identify a subprogram before it is actually referenced.

```
INTERFACE TO routine-declaration
( formal-argument-declaration(s) )
```

formal-argument-declaration(s)	Fortran type argument declarations. Optionally, each argument can contain attributes.

Description

This block identifies a subprogram and its actual arguments before it is actually referenced or called.

The *routine-declaration* defines a function or subroutine depending on whether one of those needs to be identified

Example

Consider calling a C function that has this prototype:

```
extern void Foo (int i);
```

The INTERFACE TO block to declare the Fortran call to this function is as follows:

```
INTERFACE TO SUBROUTINE Foo [C.ALIAS: '_Foo'] (I)
INTEGER*4 I
END
```

INTRINSIC (Statement and Attribute)

Identifies an intrinsic procedure.

The syntax of the type declaration statement with the INTRINSIC attribute is:

type , *attrib-list* :: *intrinsic-function-name-list*

type is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE(*name*), etc.), as described in Chapter 3, Data Types and Data Objects.

attrib-list is a comma-separated list of attributes including INTRINSIC and optionally those attributes compatible with it, namely:

PRIVATE PUBLIC

intrinsic-function-name-list is a comma-separated list of intrinsic-function-names. (Note that subroutine names cannot appear in type statements, so that intrinsic subroutine names can only be identified as such by use of the INTRINSIC statement, described below.)

The syntax of the INTRINSIC statement is:

INTRINSIC *intrinsic-procedure-name-list*

intrinsic-procedure-name-list is a comma-separated list of procedure names.

NOTE. *Like the* EXTERNAL *statement, the* INTRINSIC *statement does not have optional colons.*

Description

The INTRINSIC statement and attribute identifies a specific or generic name as that of an intrinsic procedure, enabling it to be used as an actual argument. The INTRINSIC statement is necessary to inform the compiler that a name is intrinsic and is not the name of a variable. Whenever an intrinsic name is passed as an actual argument and no other appearance of the name in the same scoping unit indicates that it is a procedure, it must be specified by the calling program in an INTRINSIC statement, or (if a function name) in a type declaration statement that includes the INTRINSIC attribute.

Each name can appear only once in an INTRINSIC statement and in at most one INTRINSIC statement within the same scoping unit. Also, a name cannot appear in both an EXTERNAL and an INTRINSIC statement within the same scoping unit.

Examples

The following INTRINSIC statement informs the compiler that sin and tan are intrinsics, enabling them to be passed to the subroutine MATH:

```
INTRINSIC sin, tan
CALL math(sin, tan)
```

The following REAL statement does the same thing, using the INTRINSIC attribute to inform the compiler that sin and tan are intrinsics:

```
REAL, INTRINSIC :: sin, tan
```

Related Statements

EXTERNAL

Related Concepts

LOGICAL

Declares entities of type logical.

LOGICAL [*kind-spec*] [[, *attrib-list*] ::] *entity-list*

kind-spec specifies the size of the logical entity in bytes. kind-spec takes the form:

([KIND=] *kind-param*)

where *kind-param* can be a named constant or a constant expression that has the integer value of 1, 2, 4, or 8. The size of the default type is 4.

As an extension, *kind-spec* can take the form:

**len-param*

where *len-param* is the integer 1, 2, 4, or 8 (default = 4).

attrib-list is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

For information about each of the attributes, see the corresponding statement in this chapter.

entity-list is a list of entities, separated by commas. Each entity takes the form:

name [(*array-spec*)] [= *initialization-expr*]

where *name* is the name of a variable or function, *array-spec* is a comma-separated list of dimension bounds, and *initialization-expr* is the initial value for the entity.

Description

The LOGICAL statement is constrained by the rules for type declaration statements, including the requirement that it precede all executable statements.

Explicitly declaring an entity with the LOGICAL statement overrides any implicit typing rules in effect.

If *attrib-list* or *initialization-expr* appears in the declaration, *entity-list* must be preceded by the double colon.

If *array-spec* is specified for an entity, it overrides any DIMENSION attribute.

Initialization

initialization-expr must be a constant logical expression that can be evaluated at compile time.

The following entities may not be initialized:

- Dummy arguments
- Function results
- Allocatable arrays
- Pointers
- External names
- Intrinsic names
- Automatic objects

If *attrib-list* includes the PARAMETER attribute, each entity in *entity-list* must include an initialization expression.

To initialize an array in a LOGICAL statement, you must use an array constructor, as in the following example:

```
LOGICAL, DIMENSION(2) :: lvec=(/.TRUE.,.FALSE./)
```

If an array is initialized, all items in the array must be initialized. Implied-DO loops cannot be used to initialize an array in a type declaration statement.

As an extension, an initializer may appear between slashes in a type declaration statement, as follows:

```
LOGICAL log1/.TRUE./, log2/.FALSE./
```

The double colon (::) may not be used with this initialization format.

Length Specification Extension

As a portability extension, Intel Fortran allows the following syntax for specifying the length of an entity:

name [**len*] [(*array-spec*)] [= *initialization-expr*]

If (*array-spec*) is specified, **len* may appear on either side of (*array-spec*).

If *name* appears with **len*, it overrides the length specified by LOGICAL**size*. For example, the following statements are equivalent declarations of log:

```
LOGICAL (KIND = 8) log8
LOGICAL*4 log8*8
```

Example

The following are valid declarations:

```
LOGICAL log1, log2
LOGICAL(KIND=2) :: log3
LOGICAL(2), PARAMETER :: test=.TRUE.
```

Related Statements

See the statement form of each of the attributes that can be specified for the LOGICAL statement.

Related Concepts

The following are discussed elsewhere in this manual:

- Implicit typing rules: Chapter 3, Data Types and Data Objects
- Expressions: Chapter 5, Expressions and Assignment
- Initialization expressions: Chapter 5, Expressions and Assignment
- Data representation models: Chapter 3, Data Types and Data Objects
- Storage classes for variables: Chapter 3, Data Types and Data Objects
- Automatic objects: Chapter 3, Data Types and Data Objects
- Arrays: Chapter 4, Arrays

MAP

Defines a union within a structure.

```
MAP
  field-def
  .
  .
  .
END MAP
```

field-def

is one of the following:

- A type declaration statement
- Another nested structure
- A nested record
- A union definition

Description

The MAP statement is an Intel Fortran extension that is used with the UNION statement to define a union within a structure. For detailed information about the MAP and UNION statements, see the description of the STRUCTURE statement in this chapter.

MODULE

Introduces a module.

```
MODULE module-name
```

module-name is a unique module name.

Description

Modules are nonexecutable program units that can contain type definitions, object declarations, procedure definitions (module procedures), external procedure interfaces, user-defined generic names, and user-defined operators and assignments. Any such definitions not specified to be private to the module containing them are available to those program units that specify the module in a USE statement. Modules provide a convenient sharing and encapsulation mechanism for data, types, procedures, and procedure interfaces.

Examples

```
! Make data objects and a data type
!  shareable via a module.
MODULE shared
  COMPLEX gtx (100, 6)
  REAL, ALLOCATABLE :: y(:), z(:,:)
  TYPE peak_item
   REAL peak_val, energy
   TYPE(peak_item), POINTER :: next
  END TYPE peak_item
END MODULE shared

! Define a data abstraction for rational
!  arithmetic via a module.
MODULE rational_arithmetic
  TYPE rational
   PRIVATE
   INTEGER numerator, denominator
  END TYPE rational  ! Generic extension of =
  INTERFACE ASSIGNMENT (=)
   MODULE PROCEDURE eqrr, eqri, eqir
  END INTERFACE
  ! Generic extension of +
  INTERFACE OPERATOR (+)
   MODULE PROCEDURE addrr, addri, addir
  END INTERFACE
```

```
  ...
  CONTAINS
  ! A specific definition of =
  FUNCTION eqrr (...)
  ...
  ! A specific definition of +
  FUNCTION addrr (...)
  ...
END MODULE rational_arithmetic
```

Related Statements

CONTAINS, END, PRIVATE, PUBLIC, and USE

Related Concepts

Use association, module procedures, program units, and encapsulation are discussed in Chapter 7, Program Units and Procedures.

MODULE PROCEDURE

Specifies module procedures in a generic interface.

MODULE PROCEDURE *module-procedure-name-list*

module-procedure-name-list is a comma-separated list of *module-procedure-names*.

Description

A MODULE PROCEDURE statement appears within an interface block. It is used when the specification is generic and a specific procedure is defined within the module rather than as an external procedure. The MODULE PROCEDURE statement only names the subprograms; it does not contain the

definition of the interface. The named subprograms must be defined within the current module or within another module that is accessible by use association.

Examples

```
MODULE path
! Module data environment.
! Module procedures contained in this module
!  have access to this data environment.
REAL x, y, z

! Generic name substance for procedures
!  air and water.

INTERFACE substance
  MODULE PROCEDURE air, water
END INTERFACE

INTERFACE OPERATOR (*)
  MODULE PROCEDURE rational_multiply
END INTERFACE
...
! Module procedures are preceded by CONTAINS.
CONTAINS
  SUBROUTINE air (contents)
  ...
  END SUBROUTINE air

  SUBROUTINE water (x, a, z)
   a = x + y
   ! x is a dummy argument.
   ! y is from the module data environment.
   ...
  END SUBROUTINE 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 Statements

`FUNCTION`, `SUBROUTINE`, and `INTERFACE`

Related Concepts

Modules and procedure interfaces are discussed in Chapter 7, Program Units and Procedures.

NAMELIST

Names a group of variables for I/O processing.

```
NAMELIST /grp-name/var-list [ [ ,]/grp-name/
 var-list ]...
```

grp-name	is a unique namelist group name.
var-list	is a comma-separated list of scalar and array variable names.

Description

The `NAMELIST` statement declares *var-list* as a namelist group and associates the group with *grp-name*.

Variables appearing in *var-list* may be of any type, including objects of derived types or their components, saved variables, variables on the local stack, and subroutine parameters. The following, however, are not allowed:

- Record or composite references
- Pointers or their targets
- Automatic objects
- Allocatable array
- Character substrings
- Assumed-size array parameters
- Adjustable-size array parameters
- Assumed-size character parameters
- Individual components of a derived type object

The *var-list* explicitly defines which items may be read or written in a namelist-directed I/O statement. It is not necessary for every item in *var-list* to be defined in namelist-directed input, but every input item must belong to the namelist group. The order of items in *var-list* determines the order of the values written in namelist-directed output.

More than one NAMELIST statement with the same *grp-name* may appear within the same scoping unit. Each successive *var-list* in multiple NAMELIST statements with the same *grp-name* is treated as a continuation of the list for *grp-name*.

The same variable name may appear in different NAMELIST statements within the same scoping unit.

Examples

```
PROGRAM
 INTEGER i, j(10)
 CHARACTER*10 c
 NAMELIST /n1/ i, j, c
 ! Define the namelist group n1.
 READ (UNIT=5,NML=n1)
 WRITE (6, n1)
END
```

When this program is compiled and executed with the following input record:

```
&n1
j(8) = 6, 7, 8
i = 5
c = 'xxxxxxxxx''
j = 5*0, -1, 2
c(2:6) = 'abcde'
/
its output is:
 &n1
 I   = 5
 J   = 0 0 0 0 0 -1 2 6 7 8
 C   ='xabcdexxx'
 /
```

Related Statements

ACCEPT, OPEN, INQUIRE, PRINT, READ, and WRITE

Related Concepts

Namelist-directed I/O is described in Chapter 8, I/O and File Handling.

NULLIFY

Disassociates a pointer from a target.

`NULLIFY (`*`pointer-object-list`*`)`

`pointer-object-list`
is a comma-separated list of variable names and derived-type components.

Description

The NULLIFY statement disassociates a pointer from any target. A NULLIFY statement is also used to change the status of a pointer from undefined to disassociated.

Examples

The following example shows the declaration and use of a variable with the pointer attribute:

```
REAL, TARGET :: value ! value can be a target
REAL, POINTER :: pt  !  for the pointer pt.pt => value
! Associate pt with value.
NULLIFY (pt)   ! Disassociate pt.
IF (.NOT.ASSOCIATED(pt)) pt => x
! The ASSOCIATED intrinsic is valid here if (and
!  only if) pt has been previously allocated,
!  assigned (as above) or nullified (as above).
```

The next example shows how a derived type can be used in list processing applications:

```
TYPE list_node
  INTEGER value
  TYPE (list_node), POINTER :: next
END TYPE list_node
TYPE (list_node), POINTER :: list
ALLOCATE (list)    ! Create new list node.
list % value = 28   ! Initialize data field.
nullify (list % next) ! Nullify pointer to the
           !  next node.
```

Related Statements

ALLOCATE, DEALLOCATE, POINTER, and TARGET

Related Concepts

Pointers and pointer association are discussed in Chapter 3, Data Types and Data Objects .

OPEN

Connects file to a unit.

`OPEN (`*`io-specifier-list`*`)`

`io-specifier-list` is a list of the following comma-separated I/O specifiers:

`[UNIT=]`*`unit`* specifies the unit to connect to an external file. *`unit`* must be an integer expression that evaluates to a number greater than 0. If the optional keyword `UNIT=` is omitted, *`unit`* must be the first item in *`io-specifier-list`*.

`ACCESS=`*`character-expression`* specifies the method of file access. *`character-expression`* can be one of the following arguments:

`'DIRECT'`	Open file for direct access.
`'SEQUENTIAL'`	Open file for sequential access (default).
`'APPEND'`	Open existing file to append to end of file.

To open a file for append (that is, to position the file just before the end-of-file record), use `POSITION=APPEND`. For information about file access methods, see Chapter 8, I/O and File Handling.

`ACTION=`*`character-expression`* specifies the allowed data-transfer operations. *`character-expression`* can be one of the following arguments:

`'READ'`	Do not allow `WRITE` and `ENDFILE` statements.
`'WRITE'`	Do not allow `READ` statements.
`'READWRITE'`	Allow any data transfer statement (default).

ASSOCIATEVARIABLE=*integer-variable*
: indicates a variable that is updated after each direct access I/O operation; this variable contains the record number of the next sequential record in the file. The restriction is that this variable cannot be a dummy argument to the routine in which the OPEN statement appears. It is valid only for direct access, in all other access modes, it is ignored.

For information about data transfer operations, see the READ, PRINT, and WRITE statements in this chapter; see also Chapter 8, I/O and File Handling.

BLANK=*character-expression*
: specifies treatment of blanks within numeric data on input. This specifier is applicable to formatted input only. *character-expression* can be one of the following arguments:

'NULL'	Ignore blanks (default).
'ZERO'	Substitute zeroes for blanks.

BLOCKSIZE=*integer-expression*
: indicates the physical I/O transfer size for the file. If the value is non-zero, it should be rounded up to a multiple of 512. If it is zero or not specified, it defaults to system default, generally 512.

CARRIAGECONTROL=*string*
: indicates the type of carriage control used when a file is displayed on a terminal device. The string values are:

'FORTRAN'	Default for code files.
'LIST'	Default for formatted file.
'NONE'	Default for binary and unformatted file.

BUFFERCOUNT=*integer expression*
: specifies the number of buffers associated with the unit for multibuffered I/O. If zero or not specified or, assumes system default.

DEFAULTFILE=*character*-expression
: indicates a default file specification string.

  Supplies the missing components of a file specification. If not specified, Fortran uses the default value FORT.n, where n is the UNIT number with leading zeros. You can indicate the following file specification components: a device, a directory, a file name, and a file type.

DELIM=*character-expression*
: specifies the delimiter to use (if any) when delimiting character constants in list-directed and namelist-directed formatting. This specifier is applicable to formatted output only. *character-expression* can be one of the following arguments:

'APOSTROPHE'	Use the apostrophe to delimit character constants in list-directed and namelist-directed formatting.
'QUOTE'	Use double-quotation marks to delimit character constants in list-directed and namelist-directed formatting.
'NONE'	Use no delimiter to delimit character constants in list-directed and namelist-directed formatting (default).

ERR=*stmt-label*
: specifies the label of the executable statement to which control passes if an error occurs during statement execution.

FILE=*character-expression* or NAME

specifies the name of the file to be connected to *unit*. *character-expression* can also be the ASCII representation of a device file. If this specifier does not appear in the OPEN statement, a temporary scratch file is created.

DISPOSE or DISP=*integer-expression*

returns the status of the file after the unit is closed. The default is 'KEEP' except for a scratch file which cannot be saved, printed or submitted. The default for scratch file is 'DELETE'.

integer-expression can be one of the following arguments:

'KEEP or SAVE'	Keeps the file after the unit closes.
'DELETE'	Deletes the file after the unit closes
'PRINT'	Submits the file to the system line printer spooler and keeps it. Note that this specifier can be used on sequential files only.
'PRINT/DELETE'	Submits the file to the system line printer spooler and then deletes it
'SUBMIT'	Submits the file to the batch job queue and keeps it.
'SUBMIT/DELETE'	Submits the file to the batch job queue and then deletes it.

FORM=*character-expression*

specifies whether the file is connected for formatted or unformatted I/O. *character-expression* can be one of the following arguments:

'FORMATTED'	Specify formatted I/O. If the file is to be opened for sequential access, this is the default.
'UNFORMATTED'	Specify unformatted I/O. If the file is to be opened for direct access, this is the default.

IOFOCUS=*logical-variable*
: returns the following values indicating whether the specified UNIT is the current active window:

.TRUE.	Specified UNIT is the current active window in a Quickwin application.
.FALSE.	Specified UNIT is not the current active window.

IOSTAT=*integer-variable*
: returns the I/O status after the statement executes. If the statement successfully executes, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.

MAXREC=*integer-expression*
: specifies the maximum number of records that can be transferred from or to a direct access file while the file is connected. The default is an unlimited number of records.

ORGANIZATION=*scalar-charatcer*
: returns a scalar character variable indicating the following record access types:

'SEQUENTIAL'	File is connected for a sequential access, records are accessed in order.
'RELATIVE'	File is connected for a direct access, records can be accessed in any order.
'UNKNOWN'	It cannot be determined whether the file is connected for sequential or relative access.

PAD=*character-expression*
: specifies whether or not to pad the input record with blanks if the record contains fewer characters than required by the format specification. This specifier is applicable to formatted input only. *character-expression* can be one of the following arguments:

  'YES'
  : Pad input records with blanks (if necessary) to fill it out to length required by format specification (default).

  'NO'
  : Do not pad input record with blanks if it is not as long as record specified by format specification.

POSITION=*character-expression*
: specifies the position of an existing file to be opened for sequential access. *character-expression* can be one of the following arguments:

  'ASIS'
  : Leave file position unchanged (default).

  'REWIND'
  : Position the file at its start.

  'APPEND'
  : Position the file just before the end-of-file record.

If the file to be opened does not exist, this specifier is ignored. New files are always positioned at their start.

READONLY
: indicates that only READ statements can refer to this connection. This specifier is similar to ACTION='READ' but READONLY prevents deletion of the file if it is closed with STATUS='DELETE' in effect. The Fortran I/O system's default file access is READWRITE. If access is denied, the I/O system automatically retries accessing the file for READ access.

RECL=*integer-expression* or RECORDSIZE
: specifies the length of each record in a file to be opened for direct access. The length is measured in characters (bytes). This specifier must be present when a file is opened for direct access and is ignored if file is opened for sequential access.

RECORDTYPE=*character-expression*
: specifies a scalar default variable *rtype* of default CHARACTER type with one of the following values:

'FIXED'	Indicates fixed-length records.
'VARIABLE'	Indicates variable-length records.
'SEGMENTED'	Indicates segmented records.
'STREAM'	Indicates stream-type variable-length records.
'STREAM_CR'	Indicates stream-type variable-length records terminated with a carriage return.
'STREAM_LF'	Indicates stream-type variable-length records terminated with a line feed.
'UNKNOWN'	File is not connected.

When you open a file, the default record types are as follows:

'FIXED'	For relative files
'FIXED'	For direct access sequential files
'STREAM_LF'	For formatted sequential access files
'VARIABLE'	For unformatted sequential access files

A segmented record is a logical record consisting of segments that are physical records. Use segmented records only for unformatted

sequential access to disk or raw magnetic tape files because the length of a segmented record can be greater than 65,535 bytes.

If an output statement does not specify a full record for a file containing fixed-length records, the following occurs:

- In formatted files, the record is filled with blanks.
- In unformatted files the record is filled with zeros.

SHARE=*character-variable* indicates whether the file locking is applied while the unit is open. The following values are used:

`'DENYRW'`	Deny-read/write mode. No process can open this file.
`'DENYWR'`	Deny-write mode. No process can open the file with write access.
`'DENYRD'`	Deny-read mode. No process can open the file with read access.
`'DENYNONE'`	Deny-none mode. Any process can open the file in any mode. This is the default value.
`'UNDEFINED'`	The access mode is undefined.

SHARED indicates that the file is connected for shared access by more than one program executing simultaneously. SHARED access is the default for Fortran I/O system.

STATUS=*character-expression* specifies the state of the file when it is opened. *character-expression* can be one of the following arguments:

`'OLD'`	Open an existing file. `FILE=` must also be specified and the named file must exist.
`'NEW'`	Create a new file. `FILE=` must also be specified and the named file must not exist.

'UNKNOWN'	If the file named in FILE= exists, open it with the status of OLD; if it does not exist, open it with the status of NEW. This is the default status.
'REPLACE'	If the file does not exist, create it with a status of OLD; if it does exist, delete it and open it with a status of NEW. If STATUS='REPLACE' is specified, FILE= must also be specified.
'SCRATCH'	Create a scratch file. FILE= specifier must *not* be specified. For information about scratch files, see Chapter 8, I/O and File Handling.

TITLE=*character expression*	indicates the name of a child window in a QuickWin application. Specifying TITLE in a non-QuickWin application causes a run-time error.
USEROPEN=*function-name*	indicates a user-written external function name that controls the opening of the file. The function must be declared in a previous EXTERNAL statement; if it has a type, the type should be INTEGER(4) (INTEGER*4). This specifier allows you to use features of the operating system additionally to Fortran.

Description

The OPEN statement connects a unit to a file so that data can be read from or written to that file. Once a file is connected to a unit, the unit can be referenced by any program unit in the program.

I/O specifiers do not have to appear in any specific order in the OPEN statement. However, if the optional keyword UNIT= is omitted, *unit* must be the first item in the list.

Only one unit can be connected to a file at a time unless the file is opened SHARED. That is, the same file cannot be connected to two different units. Attempting to open a file that is connected to a different unit will produce undefined results.

However, multiple OPENs can be performed on the same unit. In other words, if a unit is connected to a file that exists, it is permissible to execute another OPEN statement for the same unit. If FILE= specifies a different file, the previously opened file is automatically closed before the second file is connected to the unit. If FILE= specifies the same file, the file remains connected in the same position; the values of the BLANK=, DELIM=, PAD=, ERR=, and IOSTAT= specifiers can be changed, but attempts to change the values of any of the other specifiers will be ignored.

Examples

The examples in this section illustrate different uses of the OPEN statement.

The following OPEN statement connects the existing file inv to unit 10 and opens it (by default) for sequential access. Only READ statements are permitted to perform data transfers. If an error occurs, control passes to the executable statement labeled 100 and the error code is placed in the variable ios:

```
OPEN(10, FILE='inv', ERR=100, I0STAT=ios, &
   ACTION='READ', STATUS='OLD')
```

The following OPEN statement opens the file whose name is contained in the variable next1, connecting it to unit 4 as a formatted, direct-access file with a record length of 50 characters:

```
OPEN(ACCESS="DIRECT", UNIT=4, RECL=50, &
   FORM="FORMATTED", FILE=next1)
```

The following two OPEN statements produce the same results. Both open a scratch file that is connected to unit 19 (if the FILE=*name* specifier had appeared in the first statement, the named file would have been opened instead):

```
OPEN (UNIT=19)
OPEN (UNIT=19, STATUS="SCRATCH")
```

Because the I/O specifiers that can be used in an OPEN statement do not have to appear in any specific order, the following three OPEN statements are all equivalent:

```
OPEN(UNIT=3, STATUS='NEW', FILE='OUT.DAT')
OPEN(3, STATUS='NEW', FILE='OUT.DAT')
OPEN(STATUS='NEW', FILE='OUT.DAT', UNIT=3)
```

Note, however, that in the second OPEN statement 3 must appear first because of the omission of the optional keyword UNIT=. Thus, the following OPEN statement is illegal:

```
OPEN(STATUS='NEW', 3, FILE='OUT.DAT') ! illegal
```

Related Statements

CLOSE, INQUIRE, READ, and WRITE

Related Concepts

For information about I/O concepts and examples of programs that perform I/O, see Chapter 8, I/O and File Handling. For information about I/O formatting, see Chapter 9, I/O Formatting.

OPTIONAL (Statement and Attribute)

Identifies optional arguments for procedures.

The syntax of the type declaration statement with the OPTIONAL attribute is:

type , *attrib-list* :: *dummy-argument-name-list*

type is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.), as described in Chapter 3, Data Types and Data Objects.

attrib-list is a comma-separated list of attributes including OPTIONAL and optionally those attributes compatible with it, namely:

DIMENSION	INTENT	TARGET
EXTERNAL	POINTER	VOLATILE

dummy-argument-name-list
is a comma-separated list of

dummy-argument-names.

The syntax of the OPTIONAL statement is:

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

Description

If a dummy argument has the OPTIONAL attribute, the corresponding actual argument need not appear 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 presence of an optional argument in a procedure may be determined by using the PRESENT intrinsic function.

Many uses of the ENTRY statement in FORTRAN 77 programs can be replaced by the use of optional arguments.

Rules and Restrictions

- The OPTIONAL attribute may be specified only for dummy arguments. It may occur in a subprogram and in any corresponding interface body.
- 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.
- 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.

Examples

The following are two examples of the OPTIONAL statement. In the first example, the call to the subroutine trip can legally omit the path argument because it has the OPTIONAL attribute:

```
CALL TRIP ( distance = 17.0 ) ! path is omitted.
SUBROUTINE trip ( distance, path )
  OPTIONAL distance, path
```

In the next example, the subroutine `plot` uses the `PRESENT` function to determine whether or not to execute code that depends on the presence of arguments that have the `OPTIONAL` attribute:

```
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.
  ! Note that the o_xaxis dummy argument
  !  cannot be referenced if the actual
  !  argument is not present. The same
  !  applies to o_yaxis (below).
  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
! Some valid calls to plot.
CALL plot (points)
CALL plot (observed, o_xaxis = 100., &
           o_yaxis = 1000.)
CALL plot (random_pts, smooth = .TRUE.)
```

Related Statements

SUBROUTINE and FUNCTION

Related Concepts

Procedures, argument association, argument keywords are discussed in Chapter 7, Program Units and Procedures.

OPTIONS

Overrides or confirms the compiler options for a unit.

OPTIONS *option [option...]*

option can be one of the following:

/CHECK=ALL, [NO]BOUNDS, [NO]OVERFLOW, NONE
/NOCHECK
/[NO]EXTEND_SOURCE
/[NO]F77
/[NO]I4
/[NO]RECURSIVE

Some options of the OPTIONS statement are equivalent to the compiler options.

NOTE. *An option must be always preceded by a slash (/).*

Description

The OPTIONS statement overrides or confirms the compiler options in effect for a program unit.

When using the OPTIONS statement, follow these rules:

- The OPTIONS statement must be the first statement in a program unit, preceding the PROGRAM, SUBROUTINE, FUNCTION, MODULE, and BLOCK DATA statements.
- The options of the OPTIONS statement override compiler options, but only until the end of the program unit for which they are defined.

Examples

```
OPTIONS /CHECK=ALL/F77
OPTIONS /I4
```

PARAMETER (Statement and Attribute)

Specifies a named constant.

A type declaration statement with the PARAMETER attribute is:

```
type , attrib-list :: cname1 = cexpr
 [, cname2 = cexpr] ...
```

type is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.), as described in Chapter 3, Data Types and Data Objects.

attrib-list is a comma-separated list of attributes including PARAMETER and optionally those attributes compatible with it, namely:

DIMENSION	PUBLIC
PRIVATE	SAVE

Specifying the SAVE attribute in a PARAMETER statement has no effect.

cname is the name that will represent the constant.

cexpr is an initialization expression that evaluates to the constant represented by cname. In the case of an array constant, *cexpr* must be an array constructor. In the case of a derived type constant, *cexpr* must be a structure constructor.

The syntax of the PARAMETER statement is:

```
PARAMETER ( cname1 = cexpr1 [, cname2 = cexpr2]...)
```

Description

The PARAMETER statement associates a symbolic name with a constant. A symbolic name defined in a PARAMETER statement is known as a named constant. A named constant must not become defined more than once in a program unit. Once defined, it can be used only as a named constant. This means that a named constant cannot be assigned a value like a variable.

When the PARAMETER attribute is used, the value of the named constant must be provided by the initialization part of the statement in which the PARAMETER attribute appears.

The type of a named constant is determined by the implicit typing rules, unless its type is specified by a type declaration statement prior to its first appearance in a PARAMETER statement or by a type declaration statement that includes PARAMETER as one of its attributes. If a PARAMETER statement declares and implicitly types a named constant, the named constant may appear in a subsequent type declaration or IMPLICIT statement, but only to confirm the type of the named constant.

When the type of the symbolic name and the constant do not agree, the value of the named constant is assigned in accordance with assignment statement type-conversion rules, as given in Chapter 5, Expressions and Assignment.

The following rules apply to type agreement between the constant and the symbolic name:

- If *cname* is of numeric type, *cexpr* must be an arithmetic constant expression.
- If *cname* is of type character, the corresponding *cexpr* must be a character constant expression.

- If *cname* is of type logical, the corresponding *cexpr* may be either an arithmetic or logical constant expression.

Any symbolic name of a constant that appears in *cexpr* must have been defined previously in the same or a different PARAMETER statement in the same program unit. For example, the expression in the second PARAMETER statement below is built from the expression in the first PARAMETER statement, and is legal:

```
PARAMETER (limit = 1000)
PARAMETER (limit_plus_1 = limit + 1)
```

The logical operators (.EQ., .NE., .LT., .LE., .GT., and .GE.), as well as the following intrinsic functions, can appear in the PARAMETER statement:

ABS	IAND	IXOR	MAX
CHAR	ICHAR	LEN	MIN
CMPLX	IEOR	LGE	MOD
CONJB	IMAG	LGT	NINT
DIM	IOR	LLE	NOT
DPROD	ISHFT	LLT	

If these intrinsic functions are used in a PARAMETER statement, their arguments must be constants.

If the named constant is of type character and its length is not specified, the length must be specified in a type declaration statement or IMPLICIT statement prior to the first appearance of the named constant. Its type and/or length must not be changed by subsequent statements, including IMPLICIT statements. If a symbolic name of type CHARACTER*(*) is defined in a PARAMETER statement, its length becomes the length of the expression assigned to it.

If the named constant is an array, the bounds must be explicit and determined by an initialization expression.

Once such a symbolic name is defined, that name can appear in any subsequent statement of the defining program unit as a constant in an expression or DATA statement.

Example

```
! PARAMETER used in a type declaration
!  statement as an attribute.
REAL, DIMENSION(4), PARAMETER :: const = &
  (/1.2, 1.45, 0.9, 24.3/)

! PARAMETER used as a statement.
INTEGER year
PARAMETER year = 1996

! Type declaration statement declaring a
!  derived-type constant.
TYPE (postal_info), PARAMETER :: package = &
   postal_info (9.5, (/10.0, 5.5, 2.25/) )
```

Related Concepts

For information about data types and objects, see Chapter 3, Data Types and Data Objects.

PAUSE

Temporarily stops program execution.

PAUSE *pause-code*

pause-code is a character constant or a list of up to 5 digits.

Description

The PAUSE statement suspends program execution and prints a message, depending on whether digits, characters, or nothing has been specified in the PAUSE statement:

- If digits, the message "PAUSE *digits*" is written to standard error.

- If a character expression, the message "PAUSE *character-expression*" is written to standard error.
- If nothing appears after PAUSE, the word "PAUSE" is written to standard error.

After displaying the appropriate message, the PAUSE statement writes to standard output one of two messages that give information on resuming the program. If the standard input device is a terminal, the message is:

To resume program execution, type GO.

At this point the program is suspended and remains so until the operator types the word GO and presses the Return key. The program will terminate if anything other than GO is entered.

If the standard input device is other than a terminal, the message is:

To resume execution, execute a kill -15 *pid* &

command where *pid* is the unique process identification number of the suspended program. This command can be issued at any terminal at which the user is logged in.

Examples

```
! Write "PAUSE 7777" to standard error
PAUSE 7777
! Write "PAUSE MOUNT TAPE" to standard error
PAUSE 'MOUNT TAPE'
! Write "PAUSE" to standard error
PAUSE
```

Related Statements

STOP

Related Concepts

For information about the PAUSE statement and other flow control statements, see Chapter 6, Execution Control.

NOTE. *PAUSE is obsolescent in Fortran 95 and later.*

POINTER (Cray-style)

Declares Cray-style pointers and their objects. (Extension)

`POINTER` *`(pointer1, pointee1) [, (pointer2, pointee2)]...`*

`pointer`	is a pointer.
`pointee`	is a variable name or array declarator.

Description

Intel Fortran supports both the standard Fortran 95 `POINTER` statement as well as the Cray-style `POINTER` statement. The Cray-style `POINTER` statement is supported for compatibility with older, FORTRAN 77 programs. The following information applies only to the Cray-style `POINTER` statement; the Fortran 95 `POINTER` statement is described elsewhere in this chapter.

The following restrictions apply to *`pointer`*:

- It should be of type `INTEGER(4)`. If it is not, the compiler interprets its type as `INTEGER(4)` regardless of other implicit or explicit type declarations.
- It cannot be declared of any other data type.
- Another pointer cannot point to it.
- It cannot appear in a `PARAMETER` or `DATA` statement.
- It cannot be in a derived type object.

`pointee` may be of any type, including an array, a derived type, a structure, or a character string.

The following restrictions apply to the pointee:

- It cannot be a dummy argument, function name, function value, common block element, automatic object, generic interface block name, or derived type.
- It cannot be used in a COMMON, DATA, EQUIVALENCE, or NAMELIST statement.
- It cannot have any of the following attributes: ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, OPTIONAL, PARAMETER, POINTER, SAVE, and TARGET.
- Pointees that are arrays with nonconstant bounds can be used only in subroutines and functions, not in main programs.
- Variables used in an array-bound expression that appears in a POINTER statement must be either subprogram formal arguments or common block variables. The value of the expression cannot change after subprogram entry.

You associate memory with a pointer by assigning it the address of an object. Typically, this is done with the function, LOC. The LOC function returns the address of its argument, which can be assigned to a pointer. The following example assigns 0 to the pointee i:

```
INTEGER i, j
POINTER (p, i)

p = LOC(j)
j = 0
```

You can also use the MALLOC intrinsic to allocate memory from the heap and assign its return value to a pointer. Once you are done with the allocated memory, you should use the FREE intrinsic to release the memory so that it is available for reuse.

If you are using the pointer to manipulate a device that resides at a fixed address, you can assign the address to the pointer, using either an integer constant or integer expression.

Under certain circumstances, Cray-style pointers can cause erratic program behavior—especially if the program has been optimized. To ensure correct behavior, observe the following:

- Subroutines and functions must not save the address of any of their arguments between calls.
- A function must not return the address of any of its arguments.
- Only those variables whose addresses are explicitly taken with the LOC function must be referenced through a pointer.

Example

In the following example, the function MALLOC returns either the address of the block of memory it allocated or 0 if MALLOC was unable to allocate enough memory. The formal argument nelem contains the number of array elements and is multiplied by 4 to obtain the number of bytes that MALLOC is to allocate. The FREE intrinsic returns memory to the heap for reuse.

```
SUBROUTINE print_iarr(nelem)
POINTER (p, iarr(nelem))

p = MALLOC( 4*nelem )

IF (p.EQ.0) THEN
 PRINT *, 'MALLOC failed.'
ELSE
 DO i = 1,nelem
  iarr(i) = i
 END DO

 PRINT *, (iarr(i),i=1,nelem)
 CALL FREE( p )ENDIF
ENDIF
RETURN
END
```

NOTE. *Pointers can be different sizes on different architectures.* `EQUIVALENCE` *of pointers may have unpredictable results.*

NOTE. *Using Cray pointers with* `LOC` *or* `IADDR` *to initialize them is discouraged. It has detrimental effects on performance. Use Fortran 95 style pointers instead.*

Related Statements

`POINTER` (standard Fortran 95)

Related Concepts

For more information about pointers, see Chapter 3, Data Types and Data Objects.

POINTER (Statement and Attribute)

Specifies variables with the POINTER attribute.

The syntax of a type declaration statement with the `POINTER` attribute is:

`type, attrib-list :: dummy-argument-name-list`

`type` is a valid type specification (`INTEGER`, `REAL`, `LOGICAL`, `CHARACTER`, `TYPE (` *`name`*`)`, etc.), as described in Chapter 3, Data Types and Data Objects.

attrib-list is a comma-separated list of attributes including POINTER and optionally those attributes compatible with it, namely:

DIMENSION	PRIVATE	SAVE
OPTIONAL	PUBLIC	

dummy-argument-name-list is a comma-separated list of *dummy-argument-names*.

The syntax of the POINTER statement is:

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

object-name is a data object or function result.

deferred-shape-spec-list is a comma-separated list of colons.

Description

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 when it is assigned to point to a target using a pointer assignment statement.

Examples

In the first example, two array pointers are declared and used.

```
REAL, POINTER :: weight (:,:,:)
REAL, POINTER :: w_reg (:,:,:)
! Extents are not specified; they are
!  determined during execution.
READ *, i, j, k
ALLOCATE (weight (i, j, k))
! weight is created.
w_reg => weight (3:i-2, 3:j-2, 3:k-2)
```

```
! w_reg is an alias for an array section.
avg_w = sum (w_reg) / ((i-4) * (j-4) * (k-4))
DEALLOCATE (weight)
! weight is no longer needed.
```

The next example illustrates the use of pointers in a list-processing application.

```
TYPE link
  REAL value
  TYPE (link), POINTER :: next
END TYPE link

TYPE(link), POINTER :: list, save_list
NULLIFY (list)    ! Initialize 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
! Linked list removed when no longer needed.
DO
  IF (.NOT.ASSOCIATED (list) ) EXIT
  save_list => list % next
  DEALLOCATE (list)
  list => save_list
END DO
```

Related Statements

ALLOCATE, DEALLOCATE, NULLIFY and TARGET

Related Concepts

The elements of the Fortran 95 pointer facility are:

- The `POINTER` and `TARGET` attributes: see "POINTER (Statement and Attribute)"
- The `ALLOCATE`, `DEALLOCATE`, and `NULLIFY` statements: see "ALLOCATE"

The following topics related to pointers are discussed elsewhere in this manual:

- Declaring pointers: Chapter 3, Data Types and Data Objects
- Pointer arrays: Chapter 4, Arrays
- Pointer assignment: Chapter 5, Expressions and Assignment

PRINT

Writes to standard output.

The syntax of the `PRINT` statement can take one of two forms:

- Formatted and list-directed syntax:
  `PRINT` *`format [, output-list ]`*
- Namelist-directed syntax:
  `PRINT` *`name`*

`format` is one of the following:

- An asterisk (`*`), specifying list-directed I/O. For information about list-directed I/O, see Chapter 8, I/O and File Handling.
- The label of a `FORMAT` statement containing the format specification.
- An integer variable that has been assigned the label of a `FORMAT` statement.
- An embedded format specification. For information about format specifications, see Chapter 9, I/O Formatting.

name	is the name of a namelist group, as previously defined by a NAMELIST statement. Using the namelist-directed syntax, the PRINT statement sends data in the namelist group to standard output. To direct output to a connected file, you must use the WRITE statement and include the NML= specifier.
output-list	is a comma-separated list of data items for output. The data items can include expressions and implied-DO lists; see Chapter 8, I/O and File Handling for more detailed information.

Description

The PRINT statement transfers data from memory to standard output. The PRINT statement can be used to perform formatted, list-directed, and namelist-directed I/O only.

To direct output to a connected file, use the WRITE statement.

Examples

The examples in this section illustrate different uses of the PRINT statement.

Formatted Output

The following statement writes the contents of the variables num and des to standard output, using the format specification in the FORMAT statement at label 10:

```
PRINT 10, num, des
```

List-directed Output

The following statement uses list-directed formatting to print the literal string x= and the value of the variable x:

```
PRINT *, 'x=', x
```

Embedded Format Specification

The following statement uses an embedded format specification to print the same output:

```
PRINT '(A2, F8.2)', 'x=', x
```

Namelist-directed Output

The following statement prints all variables in the namelist group `coord`, using namelist-directed formatting:

```
PRINT coord
```

Related Statements

`WRITE`

Related Concepts

For information about I/O concepts, see Chapter 8, I/O and File Handling, which also gives example programs that perform I/O. For information about I/O formatting, see Chapter 9, I/O Formatting.

PRIVATE (Statement and Attribute)

Prevents access to module entities by use association.

The syntax of a type declaration statement with the `PRIVATE` attribute is:

type`, `*attrib-list*` :: `*access-id-list*

type is a valid type specification (`INTEGER`, `REAL`, `LOGICAL`, `CHARACTER`, `TYPE (` *name* `)`, etc.), as described in Chapter 3, Data Types and Data Objects.

attrib-list is a comma-separated list of attributes including PRIVATE and optionally those attributes compatible with it, namely:

ALLOCATABLE	INTRINSIC	SAVE
DIMENSION	PARAMETER	TARGET
EXTERNAL	POINTER	

access-id-list is a comma-separated list of one or more of the following:

- constant-name
- variable-name
- procedure-name
- defined-type-name
- namelist-group-name
- OPERATOR (*operator*)
- ASSIGNMENT (=)

The syntax of the PRIVATE statement is:

PRIVATE [[::] *access-id-list*]

Description

The PRIVATE attribute may appear only in the specification part of a module. The default accessibility in a module is PUBLIC; it can be changed to PRIVATE using a statement without a list. However, only one PRIVATE accessibility statement without a list is permitted in a module.

The PRIVATE attribute in a type statement or in an accessibility statement restricts the accessibility of entities such as module variables, type definitions, functions, and named constants. USE statements may restrict accessibility further.

A derived type may contain a PRIVATE attribute or an internal PRIVATE statement, if it is defined in a module. The internal PRIVATE statement in a type definition makes the components unavailable outside the module even though the type itself might be available.

The PRIVATE statement may also be used to restrict access to subroutines, generic specifiers, and namelist groups.

The PRIVATE specification for a generic name, operator, or assignment does not apply to any specific name unless the specific name is the same as the generic name.

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 within
  !  this module.
  PUBLIC  r, s, t
  ! r, s, and t are accessible outside the
  !  module.
END MODULE fourier

MODULE place
  PRIVATE
  ! Change default accessibility to PRIVATE.
  INTERFACE OPERATOR (.st.)
   MODULE PROCEDURE xst
  END INTERFACE

  PUBLIC OPERATOR (.st.)
  ! This makes .st. public; everything else is
  !  private.
  LOGICAL, DIMENSION (100) :: lt
  CHARACTER(20) :: name
  INTEGER ix, iy
END MODULE place
```

Related Statements

PUBLIC and USE

Related Concepts

The following are discussed elsewhere in this manual:

- Derived types: Chapter 3, Data Types and Data Objects
- Modules: Chapter 7, Program Units and Procedures
- Use association: Chapter 7, Program Units and Procedures
- Interface blocks: Chapter 7, Program Units and Procedures
- OPERATOR and ASSIGNMENT clauses: Chapter 7, Program Units and Procedures

PROGRAM

Identifies the main program unit.

PROGRAM *name*

name is the name of the program.

Description

The optional PROGRAM statement assigns a name to the main program unit. *name* does not have to match the main program's filename. However, if the corresponding END PROGRAM statement specifies a name, it must match *name*.

If the PROGRAM statement is specified, it must be the first statement in the main program unit.

Example

```
! A program with a name.
PROGRAM main_program

PRINT *, 'This program doesn''t do much.'
END PROGRAM main_program
```

Related Statements

`END`

Related Concepts

For information about the main program unit, see Chapter 7, Program Units and Procedures.

PUBLIC (Statement and Attribute)

Enables access to module entities by use association.

The syntax of a type declaration statement with the `PUBLIC` attribute is:

`type, attrib-list :: access-id-list`

`type` is a valid type specification (`INTEGER`, `REAL`, `LOGICAL`, `CHARACTER`, `TYPE ( name )`, etc.), as described in Chapter 3, Data Types and Data Objects.

`attrib-list` is a comma-separated list of attributes including `PUBLIC` and optionally those attributes compatible with it, namely:

`ALLOCATABLE`	`INTRINSIC`	`SAVE`
`DIMENSION`	`PARAMETER`	`TARGET`
`EXTERNAL`	`POINTER`	`VOLATILE`

`access-id-list` is a comma-separated list of one or more of the following:

- constant-name
- variable-name
- procedure-name
- defined-type-name
- namelist-group-name
- `OPERATOR ( operator )`
- `ASSIGNMENT ( = )`

The syntax of the PUBLIC statement is:

PUBLIC [[::] *access-id-list*]

Description

The PUBLIC attribute may appear only in the specification part of a module. The default accessibility in a module is PUBLIC; it can be reaffirmed using a PUBLIC statement without a list. However, only one PUBLIC accessibility statement without a list is permitted in a module.

The PUBLIC attribute in a type statement or in an accessibility statement permits access to entities such as module variables, type definitions, functions, and named constants. USE statements may control accessibility further.

A derived type may contain a PUBLIC attribute or an internal PUBLIC statement, if it is defined in a module.

The PUBLIC statement may also be used to permit access to subroutines, generic specifiers, and namelist groups.

The PUBLIC specification for a generic name, operator, or assignment does not apply to any specific name unless the specific name is the same as the generic name.

Examples

```
MODULE fourier
  PUBLIC
  ! PUBLIC unless explicitly PRIVATE.
  COMPLEX, PRIVATE :: fft
  ! fft is accessible only in the module.
  PRIVATE a, b, c
  PUBLIC  r, s, t
  ! a, b, and c are accessible only in the
  !  module. r, s, and t are accessible
  !  outside the module.
END MODULE fourier
```

```
MODULE place
  PRIVATE
  ! Change default accessibility to PRIVATE.
  INTERFACE OPERATOR (.st.)
   MODULE PROCEDURE xst
  END INTERFACE

  PUBLIC OPERATOR (.st.)
  ! This makes .st. public; everything else is
  !  private.
  LOGICAL, DIMENSION (100) :: lt
  CHARACTER(20) :: name
  INTEGER ix, iy
END MODULE PLACE
```

Related Statements

PRIVATE and USE

Related Concepts

The following are discussed elsewhere in this manual:

- Derived types: Chapter 3, Data Types and Data Objects
- Modules: Chapter 7, Program Units and Procedures
- Use association: Chapter 7, Program Units and Procedures
- Interface blocks: Chapter 7, Program Units and Procedures
- OPERATOR and ASSIGNMENT clauses: Chapter 7, Program Units and Procedures

READ

Inputs data from external and internal files.

The syntax of the READ statement can take one of the following forms:

- Long form (for use when reading from a connected file):
- READ *(io-specifier-list) [input-list]*
- Short form (for use when reading from standard input):
- READ *format [, input-list]*
- Short namelist-directed form (for use when reading from standard input into a namelist group):
- READ *name*

format is one of the following:

- An asterisk (*), specifying list-directed I/O. For information about list-directed I/O, see Chapter 8, I/O and File Handling.
- The label of a FORMAT statement containing the format specification.
- An integer variable that has been assigned the label of a FORMAT statement.
- An embedded format specification. For information about embedded format specifications, see Chapter 9, I/O Formatting.

name is the name of a namelist group, as previously defined by a NAMELIST statement. Using the namelist-directed syntax, the READ statement takes its input from standard input. To read from a connected file, you must use the NML= specifier with the full syntax form, as described below.

input-list is a comma-separated list of data items for input. The data items can include variables and implied-DO lists; see Chapter 8, I/O and File Handling for more information.

io-specifier-list is a list of the following comma-separated I/O specifiers:

[UNIT=]*unit* specifies the unit connected to the input file. *unit* can be one of the following:

- The name of a character variable, indicating an internal file
- An integer expression that evaluates to the unit connected to an external file
- An asterisk, indicating a pre-connection to unit 5 (standard input)

If the optional keyword UNIT= is omitted, *unit* must be the first item in *io-specifier-list*.

[FMT=]*format* specifies the format specification for formatting the data. *format* can be one of the following:

- An asterisk (*), specifying list-directed I/O. For detailed information about list-directed I/O, see Chapter 8, I/O and File Handling.
- The label of a FORMAT statement containing the format specification.
- An integer variable that has been assigned the label of a FORMAT statement.
- A character expression that provides the format specification. For detailed information about format specifications, see Chapter 9, I/O Formatting.

If the optional keyword FMT= is omitted, *format* must be the second item in *io-specifier-list*.

NOTE. *The* NML= *and* FMT= *specifier may not appear in the same* io-specifier-list.

[NML=]*name* specifies the name of a namelist group for namelist-directed input. *name* must have been defined in a NAMELIST statement. If the optional keyword NML= is omitted, *name* must be the

second item in the list. The first item must be the unit specifier without the optional keyword UNIT=.

The NML= and FMT= specifier may not both appear in the same *io-specifier-list.*

ADVANCE=*character-expression* specifies whether to use advancing I/O for this statement. *character-expression* can be one of the following arguments:

'YES'	Use advancing formatted sequential I/O (default).
'NO'	Use nonadvancing formatted sequential I/O.

If the ADVANCE= specifier appears in *io-specifier-list*, *unit* must be connected to an external file opened for formatted sequential I/O. Also, ADVANCE='NO' must be specified if the EOR= or SIZE= specifier appear in the list. Nonadvancing I/O is incompatible with list-directed and namelist I/O.

For more information about nonadvancing I/O, see Chapter 8, I/O and File Handling.

END=*stmt-label*

specifies the label of the executable statement to which control passes if an end-of-file record is encountered. This specifier is only valid for reading files opened for sequential access.

EOR=*stmt-label*

specifies the label of the executable statement to which control passes if an end-of-record condition is encountered. This specifier may appear in *io-specifier-list* only if ADVANCE='NO' also appears in the list.

IOSTAT=*integer-variable*

returns the I/O status after the statement executes. If the statement successfully executes, *integer-variable* is set to zero. If an end-of-file record is encountered without an error condition, it is set

	to a negative integer. If an error occurs, *integer-variable* is set to a positive integer that indicates which error occurred.
REC=*integer-expression*	specifies the number of the record to be read from a file connected for direct access. This specifier cannot appear in *io-specifier-list* with the NML=, ADVANCE=, SIZE=, and EOR= specifiers, nor with FMT=* (for list-directed I/O).
SIZE=*integer-variable*	returns the number of characters that have been read by this READ statement. This specifier may appear in *io-specifier-list* only if ADVANCE='NO' also appears in the list.

Description

The READ statement transfers data from an external or internal file to internal storage. An external file can be opened for sequential access or direct access. If it is opened for sequential access, the READ statement can perform the following types of I/O:

- Formatted
- Unformatted
- List-directed
- Namelist-directed

If the file is opened for direct access, the READ statement can perform formatted or unformatted I/O.

READ statements operating on internal files can perform formatted or list-directed I/O.

Examples

The examples in this section illustrate different uses of the READ statement.

Formatted Sequential I/O

The following READ statement reads 10 formatted records from a file opened for sequential access, using an implied-DO list to read the data into the array x_array. If the end-of-file record is encountered before the array is filled, execution control passes to the statement at label 99.

```
READ (41, '(F10.2)', END=99) (x_array(i),i=1,10)
```

Nonadvancing I/O

The following READ statement takes its input from a file that was opened for sequential access and is connected to unit 9. It uses nonadvancing I/O to read an integer into the variable key. If the statement encounters the end-of-record condition before it can complete execution, control will pass to the executable statement at label 100. After the statement executes, the number of characters that have been read will be stored in cnt.

```
INTEGER :: key
READ (UNIT=9, '(I4)', ADVANCE='NO', SIZE=cnt, &
   EOR=100) key
```

Internal File

The following statement inputs a string of characters from the internal file cfile, uses an embedded format specification to perform format conversion, and stores the results in the variables i and x:

```
READ (cfile, FMT='(I5, F10.5)') i, x
```

Namelist-directed I/O

Each of the four READ statements in the next example uses a different style of syntax to do exactly the same thing:

```
NAMELIST /nl/ a, b, c
READ (UNIT=5, NML=nl) ! 5 = standard input
READ (5, nl)
READ (*, NML=nl) ! * = standard input
READ nl ! assume standard input
```

List-directed I/O

The following statement takes its data from standard input, storing the converted value in `int_var`. The format conversion is based on the type of `int_var`.

```
READ *, int_var
```

If you knew the format, you could substitute for the asterisk one of the following:

- The label of the `FORMAT` statement with the format specification, as in the following:
  - `READ 100, int_var`
  - `100 FORMAT(I4)`
- An embedded format specification, as in the following:
  - `READ '(I4)', int_var`

Unformatted Direct-access I/O

The following statement takes its input from the file connected to unit 31. The `REC=` specifier indicates that the file has been opened for direct access and that this statement will read the record whose number is stored in the variable `rec_num`. If an I/O error occurs during the execution of the statement, an error number will be stored in `ios`, and execution control will branch to the executable statement at label `99`.

```
READ (31, REC=rec_num, ERR=99, IOSTAT=ios) a, b
```

Related Statements

`CLOSE`, `OPEN`, and `WRITE`.

Related Concepts

For more about I/O concepts, including information about files and different types of I/O, see Chapter 8, I/O and File Handling. This chapter also gives example programs using different types of I/O. For information about I/O formatting, see Chapter 9, I/O Formatting.

REAL

Declares entities of type real.

REAL [*kind-spec*] [[, *attrib-list*] ::] *entity-list*

kind-spec is the kind type parameter that specifies the range and precision of the entities in entity-list. kind-spec takes the form:

([KIND=]*kind-param*)

where *kind-param* can be a named constant or a constant expression that has the integer value of 4, 8, or 16. The size of the default type is 4.

As an extension, kind-spec can take the form:

** len-param*

where len-param is the integer 4, 8, or 16 (default = 4).

attrib-list is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

For information about each of the attributes, see the corresponding statement in this chapter.

entity-list is a list of entities, separated by commas. Each entity takes the form:

name [(array-spec)] [= initialization-expr]

where *name* is the name of a variable or function, *array-spec* is a comma-separated list of dimension bounds, and *initialization-expr* is the initial value for the entity.

Description

The REAL statement is used to declare the length and properties of data that approximate the mathematical real numbers. A kind parameter (if present) indicates the representation method.

The REAL statement is constrained by the rules for all type declaration statements, including the requirement that it precede all executable statements.

Explicitly declaring an entity with the REAL statement overrides any implicit typing rules in effect.

If *attrib-list* or *initialization-expr* appear in the declaration, *entity-list* must be preceded by the double colon.

If *array-spec* is specified for an entity, it overrides any DIMENSION attribute.

Initialization

initialization-expr must be a constant expression that can be evaluated at compile time.

The following entities may not be initialized:

- Dummy arguments
- Function results
- Allocatable arrays
- Pointers
- External names
- Intrinsic names
- Automatic objects

If *attrib-list* includes the PARAMETER attribute, each entity in *entity-list* must include an initialization expression.

To initialize an array in a REAL statement, you must use an array constructor, as in the following example:

```
REAL, DIMENSION(4) :: rvec=(/ 1.1,2.2,3.3,4.4 /)
```

If an array is initialized, all items in the array must be initialized. Implied-DO loops cannot be used to initialize an array in a type declaration statement.

As an extension, an initializer may appear between slashes in a type declaration statement, as follows:

```
REAL x/2.87/, y/93.34/, z/13.99/
```

The double colon (`::`) may not be used with this initialization format.

Length Specification Extension

As a portability extension, Intel Fortran allows the following syntax for specifying the length of an entity:

name [**len*] [(*array-spec*)] [= *initialization-expr*]

If (*array-spec*) is specified, **len* may appear on either side of (*array-spec*).

If name appears with **len*, it overrides the length specified by REAL**size*. For example, the following statements are equivalent declarations of x:

```
REAL (KIND = 8)::x
REAL*4 x*8
```

Example

The following are valid declarations:

```
REAL, TARGET :: x, y
REAL(KIND=16) :: z
REAL(4), PARAMETER :: pi=3.14
```

Related Statements

`DOUBLE PRECISION`

Related Concepts

The following are discussed elsewhere in this manual:

- Implicit typing rules: Chapter 3, Data Types and Data Objects
- Data representation models: Chapter 3, Data Types and Data Objects
- Storage classes for variables: Chapter 3, Data Types and Data Objects
- Automatic objects: Chapter 3, Data Types and Data Objects
- Arrays: Chapter 4, Arrays
- Expressions: Chapter 5, Expressions and Assignment
- Initialization expressions: Chapter 5, Expressions and Assignment

RECORD

Declares a record of a previously defined structure.

```
RECORD /struct-name/rec-name [, rec-name]...
  [/struct-name/rec-name [, rec-name ] ]...
```

struct-name is the name of a structure declared in a previous structure definition.

rec-name is a record name.

Description

Intel Fortran supports the RECORD statement as a compatibility extension. New programs should use the derived type, a standard feature of Fortran 95. For more information about derived types, see Chapter 3, Data Types and Data Objects and the TYPE statement in this chapter.

The RECORD statement declares a record variable of a structure that has been previously defined by a STRUCTURE statement. A record variable can consist of multiple data items, called *fields*. The STRUCTURE statement is described separately in this chapter.

Referencing Record Fields

The syntax for referencing a field in a record depends on whether the field itself is another record (a composite reference) or not (a simple reference). Composite references have the following syntax:

```
rec-name [. substruct-fieldname]...
```

Simple references have the following syntax:

```
rec-name [. substruct-fieldname]... simple-fieldname
```

rec-name is the name of the record in which a composite or simple field is being referenced.

substruct-field-name is the name of a nested structure or nested record field name, if applicable.

simple-field-name is the name of a lowest-level field, defined with a type declaration statement. As indicated by the syntax, the field could be part of a nested structure or nested record.

Given the following structure definition and record declarations:

```
STRUCTURE /abc/
 REAL a, b, c(5)
 STRUCTURE /xyz/ xyz, xyzs(5)
  INTEGER   x, y, z(3)
 END STRUCTURE
END STRUCTURE

RECORD /abc/ abc, abcs(100)
RECORD /xyz/ xyz
```

the following are composite references:

```
abc   !composite record references
abcs(1)
xyz
abcs(idx)

abc.xyz !composite field references
abc.xyzs(3)
```

and the following are simple references:

```
abc.a
abc.c(1)
xyz.x
xyz.z(1)
abc.xyz.x
abcs(idx).xyz.y(1)
abcs(2).xyzs(3).z(1)
```

Composite references can be either to an entire record or to a record field that is itself a structure or record.

Rules for Record Field

Arrays of records can be created as follows:

```
RECORD /student/ students(1000)
```

or

```
RECORD /student/ students
DIMENSION students (1000)
```

In either case a 1000-record array called `students` of structure `student` is declared.

Records can be placed in common blocks. The following code places the `students` array (declared above) in the common block `frosh`, along with variables `a`, `b`, and `c`:

```
COMMON /frosh/ a, b, c, students
```

Simple field references can appear wherever a variable can appear. The following assigns values to the fields of record `r` of structure `struct`:

```
STRUCTURE /struct/
 INTEGER    i
 REAL     a
END STRUCTURE

RECORD /struct/ r
r.i = r.i + 1
r.a = FLOAT(r.i) - 2.7
```

Composite assignment is allowed for two records or two composite fields of the same structure—that is, the record declaration statements for both records must have specified the same *`struct-name`*. For example, the following is legal:

```
STRUCTURE /string/ BYTE len
 CHARACTER*1 str(254)
END STRUCTURE
RECORD /string/ str1, str2
str1 = str2
```

The following example is also valid and uses composite assignment to assign the value of the record `edate` of structure `date` to a field of the same structure (when) in the record event:

```
STRUCTURE /event/
 CHARACTER*20 desc
 STRUCTURE /date/ when
  BYTE month, day
  INTEGER*2 year
 END STRUCTURE
END STRUCTURE
```

```
RECORD /date/ edate
RECORD /event/ event
edate.month = 1
edate.day  = 6edate.year = 62
event.desc = 'Party for Joanne'

! composite assignment of record to field
!  of record--both have same structure
event.when = edate
```

Even though the following records are of identical structures—that is, the fields of both structures have the same type, size, and format—the code is invalid because the structures have a different name:

```
STRUCTURE /intarray/
 BYTE     elem_count
 INTEGER    arr(100)
END STRUCTURE

STRUCTURE /iarray/
 BYTE     elem_count
 INTEGER    arr(100)
END STRUCTURE

RECORD /intarray/ iarray1
RECORD /iarray/ iarray2

! The next assignment won't work. The two
!  records are not of the same structure.
iarray1 = iarray2   ! Invalid
```

When performing I/O on structures and records, composite record and field references can appear only in unformatted I/O statements. They are not allowed in formatted, list-directed, or namelist-directed I/O statements. However, simple field references can appear in all types of I/O statements. (For information about formatted and unformatted I/O, see Chapter 9, I/O Formatting.)

A record name or composite field reference can appear as either a formal or an actual argument to a subroutine or function. Formal and actual arguments must have the same size as well as the same number, type, and order of fields. (For information about procedure arguments, see Chapter 7, Program Units and Procedures.)

Composite record and field arguments to subroutines and functions are passed by reference, just like other Intel Fortran arguments.

Adjustable arrays are allowed in RECORD statements that declare formal arguments.

Do not name a field with any of the following:

- Logical constants, .TRUE. and .FALSE.
- Logical operators, such as .OR., .AND., and .NOT.
- Relational operators, such as .EQ., .LT., and .NEQV.
- The name of a defined operator

Related Statements

STRUCTURE and *TYPE*

Related Concepts

For information about derived types, see Chapter 3, Data Types and Data Objects.

RETURN

Returns control from a subprogram.

RETURN [*scalar-integer-expression*]

scalar-integer-expression	is an optional scalar integer expression that is evaluated when the RETURN statement is executed. It determines which alternate return is used.

Description

A RETURN statement can appear only in a subprogram.

An expression may appear in a RETURN statement only if alternate returns (one or more asterisks) are specified as dummy arguments in the relevant FUNCTION, SUBROUTINE, or ENTRY statement of the subprogram. An expression with a value *i* in the range will return to the *i*th asterisk argument (specified as **label*) in the actual argument list. A normal return is executed if *i* is not in the range 1 to *n*, where *n* is the number of dummy argument alternate returns specified.

Example

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

Related Statements

SUBROUTINE and FUNCTION

Related Concepts

Procedures are described in Chapter 7, Program Units and Procedures.

REWIND

Positions file at its initial point.

The syntax of the REWIND statement can take one of the following forms:

- Short form:
  REWIND *integer-expression*
- Long form:
  REWIND *(io-specifier-list)*

integer-expression is the unit connected to a sequential file or device.

io-specifier-list is a list of the following comma-separated I/O specifiers:

[UNIT=]*unit*	specifies the unit connected to an external file opened for sequential access. *unit* must be an integer expression that evaluates to a number greater than 0. If the optional keyword UNIT= is omitted, *unit* must be the first item in *io-specifier-list*.
ERR=*stmt-label*	specifies the label of the executable statement to which control passes if an error occurs during statement execution.
IOSTAT=*integer-variable*	returns the I/O status after the statement executes. If the statement executes successfully, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.

Description

The REWIND statement repositions the file connected to the specified unit at the start of the first record. If the file is already at its starting point or if the unit is not connected to a file, the REWIND statement has no effect.

Examples

The following example of the REWIND statement repositions the file connected to unit 10 to its initial point:

```
REWIND 10
```

The next example repositions to its initial point the file connected to unit 21. If an error occurs during the execution of the statement, control passes to the statement at label 99, and the error code is returned in ios:

```
REWIND (21, ERR=99, IOSTAT=ios)
```

Related Statements

BACKSPACE, ENDFILE, and OPEN

Related Concepts

For information about I/O concepts, see Chapter 8, I/O and File Handling. This chapter also gives example programs performing I/O.

SAVE (Statement and Attribute)

Stores variables in static memory.

A type declaration statement with the SAVE attribute is:

type , attrib-list :: save-list

type is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.), as described in Chapter 3, Data Types and Data Objects.

attrib-list A comma-separated list of attributes including SAVE and optionally those attributes compatible with it, namely:

ALLOCATABLE	PRIVATE	TARGET
DIMENSION	PUBLIC	
POINTER	STATIC	

save-list is a comma-separated list of names of objects to save.

The syntax of the SAVE statement is:

SAVE [[::] *save-list*]

Description

The SAVE statement and attribute cause objects in a subroutine or function to be stored in static memory, instead of being dynamically allocated whenever the procedure is invoked (the default case). A saved object retains its value and definition, association, and allocation status between invocations of the program unit in which the saved object is declared.

If *save-list* is omitted, everything in the scoping unit that can be saved is saved. No other explicit occurrences of the SAVE attribute or the SAVE statement are allowed.

The names of the following may appear in *save-list*:

- Scalar variables
- Arrays
- Named common blocks
- Derived type objects
- Records

If the name of a common block appears in *save-list*, it must be delimited by slashes (for example, /my_block/); all variables in the named common block are saved. If a common block is saved in one program unit, it must be saved in all program units (except main) where it appears.

Intel Fortran always saves all common blocks unless they appear in a dynamic COMMON command-liner switch.

The following must not appear in *save-list*:

- Formal argument names
- Procedure names
- Selected items in a common block
- Variables declared with the AUTOMATIC statement or attribute
- Function results
- Automatic data objects (such as automatic arrays, allocatable arrays, automatic character strings, and Fortran 95 pointers)

Initializing a variable in a DATA statement or in a type declaration statement implies that the variable has the SAVE attribute, unless the variable is in a named common block in a block data subprogram.

A SAVE statement in a main program unit has no effect.

NOTE. SAVE, *used on variables that are often used in your program, is likely to have a negative impact on your program's performance. It is better to make sure early in the routine that you assign a value to often used variables or declare them in* COMMON *and use a temporary copy of the* COMMON *variable for* USE*s in a routine.*

Example

The SAVE statement in the following example saves the variables a, b, and c, as well as the variables in the common block dot:

```
SUBROUTINE matrix
SAVE a, b, c, /dot/
RETURN
```

The SAVE statement in the next example saves the values of all of the variables in the subroutine fixit:

```
SUBROUTINE fixit
SAVE
RETURN
```

Related Statements

AUTOMATIC and STATIC

Related Concepts

Storage classes are described in Chapter 3, Data Types and Data Objects, and recursion is described in Chapter 7, Program Units and Procedures.

SELECT CASE

Begins CASE construct.

`[`*`construct-name`* `:] SELECT CASE (`*`case-expr`*`)`

`construct-name` is the name given to the CASE construct.

`case-expr` is a scalar expression of type integer, character, or logical.

Description

The `SELECT CASE` statement, the first statement of a `CASE` construct, causes *`case-expr`* to be evaluated, resulting in the case index. The `CASE` construct uses the case index to determine which of its statement blocks to execute.

If *`construct-name`* is specified, it must also appear in the `END SELECT` statement.

Example

For an example of the `SELECT CASE` statement, see the `CASE` statement in this chapter.

Related Statements

`CASE` and `END` (construct)

Related Concepts

For information about the `CASE` construct, see Chapter 6, Execution Control.

SEQUENCE

Imposes storage sequence on components of derived type object.

```
SEQUENCE
```

Description

The SEQUENCE statement can appear once within any derived type definition; its presence specifies that the storage sequence of the elements is the same as their definition order. The derived type then becomes a *sequence derived type*. The SEQUENCE statement is used:

- To allow objects of this type to be storage associated, or
- To allow actual and dummy arguments to have the same type without use or host association.

Points to note:

- If a component of a sequence derived type is a derived type, then it must also be a sequence derived type.
- The storage association statements COMMON and EQUIVALENCE can be applied to structures when sequencing is imposed on their type definitions.
- The corresponding actual and dummy arguments of derived types are of the same derived type if the structures refer to the same type definition. Alternatively, they are of the same type if all of the following conditions are true:
  - — They refer to different type definitions with the same name.
  - — They have the SEQUENCE statement in their definitions.
  - — The components have the same names and types and are in the same order.
  - — None of the components is of a private type or of a type that has private access.

Examples

```
TYPE weather
! weather is a sequence derived type with two
! character components & two integer components.
  SEQUENCE
  CHARACTER(LEN=32) place
  INTEGER high_temp, low_temp
  CHARACTER(LEN=16) conditions
END TYPE weather
```

Related Statements

TYPE, COMMON, and EQUIVALENCE

Related Concepts

Storage association is discussed in Chapter 3, Data Types and Data Objects, and argument association in Chapter 7, Program Units and Procedures.

STATIC (Statement and Attribute)

Gives variables and arrays static storage. (Extension)

The syntax of a type declaration statement with the STATIC attribute is:

type, attribute-list :: entity-list

type is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.), as described in Chapter 3, Data Types and Data Objects.

attribute-list is a comma-separated list of attributes including STATIC and optionally those attributes compatible with it, namely:

ALLOCATABLE	PRIVATE	VOLATILE
DIMENSION	SAVE	
POINTER	TARGET	

entity-list is a comma-separated list of variables and arrays.

The syntax of the STATIC statement is:

STATIC [::] *entity-list*

Description

The STATIC statement and attribute is a Intel Fortran extension. Variables possessing the STATIC attribute retain their storage location for the duration of the program. A STATIC variable declared within a procedure will therefore retain its value between calls of the procedure.

Examples

```
SUBROUTINE work_out(first_call)
 LOGICAL first_call
 INTEGER, STATIC :: ncalls
 IF (first_call) ncalls = 0
 ncalls = ncalls + 1
 ! Record the number of times the subroutine
 !  has been called.
```

Related Statements

AUTOMATIC and SAVE

Related Concepts

Storage classes for variables are discussed in Chapter 3, Data Types and Data Objects.

STOP

Terminates program execution.

```
STOP [stop-code]
```

stop-code is a character constant, a named constant, or a list of up to 5 digits.

Description

The STOP statement terminates program execution and optionally prints a message to standard error or standard list.

STOP also sends a message to standard error, dependent on whether digits, characters, or nothing was specified with the STOP statement:

- If digits are specified, the message "STOP *digits*" is written to standard error.
- If a character expression is specified, the message "STOP *character-expression*" is written.
- If nothing appears after STOP, nothing is written.

Example

```
IF (b .LT. c) STOP 'BAD VALUE!'
```

Related Statements

PAUSE

Related Concepts

For information about the STOP statement and other flow control statements, see Chapter 5, Expressions and Assignment.

STRUCTURE

Defines a named structure.

```
STRUCTURE /struct-name/
field-def
.
.
.
END STRUCTURE
```

struct-name is the structure's name, delimited by slashes. *struct-name* can be used later to declare a record.

field-def is a field definition.

Description

Intel Fortran supports the STRUCTURE statement as a compatibility extension. New programs should use the derived type, a standard feature of Fortran 95; derived types provide the same functionality as named structures. For more information about derived types, see Chapter 3, Data Types and Data Objects and the TYPE (declaration) statement in this chapter.

The STRUCTURE statement defines the type, size, and layout of a structure's fields, and assigns a name to the structure. Once a structure is defined, you can declare records of that structure using the RECORD statement and can manipulate the record's fields.

A structure definition pertains only to the program unit in which it is defined. For example, you cannot define a structure in the main program unit and then declare a record of that structure in a subprogram unit. Instead, the structure must be explicitly defined again in the subprogram unit.

field-def can be any of the following:

- A type declaration statement
- A nested structure definition
- A nested record declaration
- A union definition

Each type of field definition is described in the remaining sections.

Field Definition as Type Declaration

At the simplest level, *field-def* can be a type declaration statement. As such, *field-def* has the same syntax as a standard Fortran 95 type declaration statement, except that the only attribute that can be specified is the DIMENSION attribute. A variable defined with a type declaration statement is called a *field*.

The following code uses simple type declaration statements to define a structure named date with three fields: month and day of type BYTE, and year of type INTEGER(KIND=2):

```
STRUCTURE /date/
 BYTE :: month, day
 INTEGER(KIND=2) :: year
END STRUCTURE
```

A type declaration statement in a structure definition can optionally define initial values for the fields. For example:

```
STRUCTURE /xyz/
 REAL :: x = 1.0, y = 2.0, z = 3.0
END STRUCTURE
```

Thereafter, any record declared of structure xyz will have its x, y, and z fields initially set to 1.0, 2.0, and 3.0 respectively. Consider the following:

```
RECORD /xyz/ xyz
PRINT *, xyz.x, xyz.y, xyz.z
```

Even though no values have been assigned to the fields of xyz with an assignment statement, the above code will display:

```
 1.0 2.0 3.0
```

Implicit typing is not allowed in a structure definition. For example, the following code would cause a compile error:

```
STRUCTURE /dimensions/
 x, y, z  ! illegal
END STRUCTURE
A correct way to code this would be:
STRUCTURE /dimensions/
 REAL(KIND=8) :: x, y, z  ! legal
END STRUCTURE
A field type declaration statement can also define an
array, as in the following:
```

```
STRUCTURE /foo_bar/
 INTEGER foo(10)
END STRUCTURE
or, using Fortran 95 syntax:
STRUCTURE /foo_bar/
 REAL, DIMENSION(30, 50) :: bar
END STRUCTURE
```

The array's dimensions must in any case appear in the type statement. The DIMENSION statement (but not the DIMENSION attribute) is illegal in a structure definition. The following code defines the structure, string, which uses a type declaration statement to define an array field str of type CHARACTER(LEN=1), containing 254 elements:

```
STRUCTURE /string/
 CHARACTER(LEN=1) :: str(254)! Contains string
 INTEGER :: length     ! string's length
END STRUCTURE
```

As mentioned, the DIMENSION statement cannot be used in a structure definition. For example, the following code would cause a compile error:

```
STRUCTURE /real_array/
 REAL :: rarray
 DIMENSION arr(100)   ! illegal example
END STRUCTURE
```

A correct way to code this would be:

```
STRUCTURE /real_array/
 REAL :: rarray(100)
END STRUCTURE
```

or

```
STRUCTURE /real_array/
 REAL, DIMENSION(100) :: arr
END STRUCTURE
```

Assumed-size and adjustable arrays are also illegal in structure definitions. For example, the following is illegal:

```
STRUCTURE /assumed_size/ ! illegal example
 CHARACTER*(*) :: carray
END STRUCTURE
```

The following is also illegal:

```
STRUCTURE /adj_array/  ! illegal example
 INTEGER :: size
 REAL :: iarray(size)
END STRUCTURE
```

For alignment purposes, Intel Fortran provides the %FILL field name. It enables the programmer to pad a record to ensure proper alignment. The padding does not have a name and is therefore not accessible. For example, the following structure, sixbytes, creates a 6-byte structure, of which 4 bytes are inaccessible filler bytes:

```
STRUCTURE /sixbytes/
 INTEGER(KIND=2) :: twobytes
 CHARACTER(LEN=4) :: %FILL
END STRUCTURE
```

%FILL can be of any type and may appear more than once in a structure.

%FILL should not be needed in normal usage. The compiler automatically adds padding to ensure proper alignment.

Nested Structures

A *field-def* can itself be a structure definition, known as a nested structure. The syntax of a nested structure definition is:

```
STRUCTURE /struct-name/struct-field-list
field-def
.
.
.
END STRUCTURE
```

struct-name	is the structure's name (delimited by slashes), which can be used later to declare a record.
struct-field-list	is a list of one or more names of nested structure field names separated by commas.
field-def	can be one of the following regular field definitions (defined in the same way as an unnested structure field): • A type declaration statement • Another nested structure

- A nested record
- A union definition

Note that a structure definition allows multiple levels of nesting.

A nested structure definition is the same as an unnested structure definition, with two exceptions:

- */struct-name/* is optional in a nested structure.
- A nested structure definition must include a list of one or more structure field names (*struct-field-list*).

If */struct-name/* is present in a nested structure definition, the structure *struct-name* can also be used in subsequent record declarations. For example, the following code defines a structure named `person`, which contains a nested structure named `name`. The structure's field name is `nm` and contains three `CHARACTER*10` fields: `last`, `first`, and `mid`.

```
STRUCTURE /person/
 INTEGER :: person_id
 ! Define the nested structure 'name' with the
 !  field name 'nm'.
 STRUCTURE /name/ nm
  CHARACTER(LEN=10) :: last, first, mid
 END STRUCTURE
END STRUCTURE
```

Given this definition, the following code defines the record `p` of structure `person` and the record `n` of structure `name`:

```
RECORD /person/p
RECORD /name/n
```

If */struct-name/* is not present, then the structure can be used only in this declaration. For example, we could redefine the `person` structure so that the nested structure no longer has a name:

```
STRUCTURE /person/
 INTEGER :: person_id
 STRUCTURE nm
  CHARACTER(LEN=10) :: last, first, mid
 END STRUCTURE
END STRUCTURE
```

There is no way to declare a separate record of the nested structure because it has no name. Note, however, that the nested structure still has a field name, `nm`. The field name is required.

To declare an array of nested structures, simply specify a dimension declarator with the structure's field name. For example, the following structure definition contains a nested, 3-element array of structures with field name phones of structure phone:

```
STRUCTURE /person/
 INTEGER :: person_id
 ! Define the nested structure 'name' with the
 !  field name 'nm'.
 STRUCTURE /name/ nm
  CHARACTER(LEN=10) :: last, first, mid
 END STRUCTURE
 ! Nested array of structures.
 STRUCTURE /phone/ phones(3)
  INTEGER(KIND=2) :: area_code
  INTEGER :: number
 END STRUCTURE
END STRUCTURE
```

Nested Records

A *field-def* can be a record declaration, known as a nested record. (See the RECORD statement in this chapter for information about record declarations.) A nested record declaration must use a structure that has already been defined. The following code first defines the structure date. It then declares the structure event, which contains the nested record when of structure date:

```
STRUCTURE /date/
 BYTE :: month, day
 INTEGER :: year
END STRUCTURE
STRUCTURE /event/
 CHARACTER :: what, where
 RECORD /date/ when
END STRUCTURE
```

A structure definition can also declare an array of nested records. For example, the following code defines the structure `calendar`, which contains a 100-element array of records of structure `event`:

```
STRUCTURE /calendar/
 ! number of events
 INTEGER(KIND=2) :: event_count
 ! array of event records
 RECORD /event/ events(100)
END STRUCTURE
```

Unions

A *field-def* can be a union—a form of nested structure in which two or more map blocks share memory space. The `UNION` and `MAP` statements together define a union. The syntax of a union definition is:

```
UNION
  map-block
  map-block
  .
  .
  .
END UNION
```

where *map-block* is defined by a `MAP` statement and one or more field definitions. All map blocks within the enclosing `UNION` statement share the same memory space in a record. The syntax for defining a map block is:

```
MAP
  field-def
  .
  .
  .
END MAP
```

where *field-def* can be one of the following:

- A type declaration statement
- Another nested structure

- A nested record
- A union definition

Note that a structure definition allows multiple levels of nesting.

For programmers who are familiar with C or Pascal, Intel Fortran unions are similar to unions in C and variant records in Pascal. Intel Fortran unions differ from C unions in that they must be defined inside a structure definition.

The structure below contains a union with two map blocks. The first contains the integer field `int`; the second contains the real field `float`.

```
STRUCTURE /var/
 INTEGER :: type  ! 1=INTEGER, 2=REAL
 UNION
  MAP
   INTEGER :: int
  END MAP
  MAP
   REAL :: float
  END MAP
 END UNION
END STRUCTURE
```

To declare a record of this structure named v, use the following `RECORD` statement:

```
RECORD /var/ v
```

The declaration of the record v reserves 8 bytes of storage: 4 bytes for the `type` field and 4 bytes to be shared by `int` and `float`. If you use the `int` field to access the 4 bytes, they will be interpreted as an integer; if you use the `float` field, they will be interpreted as a real.

It is the programmer's responsibility to ensure that appropriate values are assigned to each field in a union. For instance, given the previous declaration of v, the following assignments make sense:

```
v.type =1 ! set the type to integer
! access the storage shared by 'int' and 'float'
!  as an integer
v.int = 3
```

In contrast, the following code would yield unexpected results, although it would compile without errors:

```
v.type = 1 ! set the type to integer
! the next statement contradicts the previous
!  statement
v.float = 3.14
```

Once a value is assigned to a map block, all other map blocks become undefined. The reason is that all map blocks share memory space within a union; therefore, the values of one map block may become altered if you assign a value to a field in another map block. Consider the following definition of a structure called `struct` and the declaration of a record called `rec`:

```
STRUCTURE /struct/
 UNION
  MAP
   CHARACTER*8 :: s
  END MAP
  MAP
   CHARACTER*1 :: c(8)
  END MAP
 END UNION
END STRUCTURE

RECORD /struct/ rec
```

If we made the following assignment to the s field:

```
rec.s = 'ABCDEFGH'
```

and then executed the next two PRINT statements:

```
PRINT *, rec.s
PRINT *, rec.c
```

the output would be:

```
ABCDEFGH
ABCDEFGH
```

Now, if we set values in the c field and display both fields again

```
rec.c(1) = '1'
rec.c(8) = '8'
PRINT *, rec.s
PRINT *, rec.c
```

the output would be:

```
1BCDEFG8
1BCDEFG8
```

Note how the s field has changed, even though it was not directly assigned any new values. This is a result of the `s` and `c` field sharing the same storage space in the union. Although this is valid coding—that is, it will not cause a compiler or runtime error—it may cause unexpected results.

However, you can also use shared memory mapping to your benefit. The fact that map blocks share space within a union makes unions useful for equivalencing data within a record. For example, the following structure could be used to mask off individual bytes in a 4-byte word:

```
STRUCTURE /wordmask/
 UNION
  MAP
   INTEGER(KIND=4) :: word
  END MAP
  MAP
   BYTE :: byte0, byte1, byte2, byte3
  END MAP END UNION
END STRUCTURE

RECORD /wordmask/ maskrec
```

If we assign a value to the `word` field of `maskrec`, we can then get the individual values of all four bytes in `maskrec` by looking at the fields `byte0`, `byte1`, `byte2`, and `byte3`. To see how the integer variable `word` maps onto the byte variables `byte0`, `byte1`, `byte2`, and `byte3`, use the following statements:

```
  maskrec.word = 32767

  WRITE(*, fmt=100) 'word = ', maskrec.word
  WRITE(*, 200) 'byte 0 = ', maskrec.byte0
  WRITE(*, 200) 'byte 1 = ', maskrec.byte1
  WRITE(*, 200) 'byte 2 = ', maskrec.byte2
  WRITE(*, 200) "byte 3 = ', maskrec.byte3
100 FORMAT(A, Z8.8)
200 FORMAT(A, Z2.2)
```

This code displays the following output:

```
word = 00007FFF
byte 0 = 00
byte 1 = 00
byte 2 = 7F
byte 3 = FF
```

Such code, depending as it does on a specific word size, is inherently nonportable.

Related Statements

RECORD and TYPE

Related Concepts

Derived Types are described in Chapter 3, Data Types and Data Objects.

SUBROUTINE

Begins the definition of a subroutine subprogram.

```
[RECURSIVE] SUBROUTINE subr-name [([dummy-arg-list])]
```

dummy-arg-list is a comma-separated list of zero or more of the following:

- *dummy-arg-name*
- *

As indicated by the syntax, the parentheses surrounding the dummy arguments may be omitted if there are no dummy arguments.

Description

The SUBROUTINE statement is the first statement of a subroutine subprogram.

Points to note:

- A subroutine subprogram is either an external, module, or internal subprogram.
- If a subroutine calls itself directly or indirectly, the word RECURSIVE must appear in the SUBROUTINE statement. If the keyword RECURSIVE is specified, the subroutine interface is explicit within the subprogram.
- The keyword SUBROUTINE must appear on the END statement if the subroutine is a module or internal procedure.
- An asterisk in a subroutine dummy argument list designates an alternate return.
- 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.

Examples

Consider the following subroutines:

```
SUBROUTINE exchange (x, y)
  ! A subroutine definition with two arguments.
  temp = x; x = y; y = temp
END SUBROUTINE exchange
SUBROUTINE altitude (*, long, lat)
  ! An alternate return
  IMPLICIT NONE
  INTEGER, OPTIONAL :: long, lat
  RETURN 1
END SUBROUTINE altitude
```

The preceding subroutines may be referenced with the CALL statement, as in the following program:

```
PROGRAM reject
  ! A subroutine reference.
  CALL exchange (a,t)
  !
```

```
  !A subroutine reference, including an
  ! alternate return label, missing optional
  ! argument, and an argument keyword
  CALL altitude (*90, lat = 49)
END PROGRAM reject
```

Following are some other examples of subroutine statements:

```
SUBROUTINE PRESSURE_SURFACE ! No arguments
SUBROUTINE TAFFY ()     ! Also no arguments
RECURSIVE SUBROUTINE FACT (N,X)
```

Related Statements

`CALL`, `END`, `ENTRY`, `FUNCTION`, and `RETURN`

Related Concepts

Module procedure, internal procedure, generic procedure, defined assignment, recursion, argument association, and scope are all covered in Chapter 7, Program Units and Procedures.

TARGET (Statement and Attribute)

Allows variables and arrays to be pointer targets.

The syntax of a type declaration statement with the `TARGET` attribute is:

`type, attrib-list :: entity-list`

`type`	is a valid type specification (`INTEGER`, `REAL`, `LOGICAL`, `CHARACTER`, `TYPE (`*`name`*`)`, etc.), as described in Chapter 3, Data Types and Data Objects.

attrib-list is a comma-separated list of attributes including TARGET and optionally those attributes compatible with it, namely:

ALLOCATABLE	OPTIONAL	SAVE
DIMENSION	PRIVATE	
INTENT	PUBLIC	

entity-list is a comma-separated list of entities. Each entity is of the form:

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

If (*deferred-shape-spec-list*) is omitted, it must be specified in another declaration statement.

array-name is the name of an array being given the attribute ALLOCATABLE.

deferred-shape-spec-list is a comma-separated list of colons, each colon representing one dimension. Thus the rank of the array is equal to the number of colons specified.

The syntax of the TARGET statement is:

```
TARGET [::] object-name [( array-spec )]
      [, object-name [( array-spec )]]...
```

array-spec is one of the following:

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

explicit-shape-spec is *[lower-bound :] upper-bound*

assumed-shape-spec is *[lower-bound] :*

deferred-shape-spec is :

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

That is, an *assumed-size-spec* is an *explicit-shape-spec-list* with the final upper bound given as *.

Description

The TARGET attribute or statement specifies that *name* is a target that may be pointed at 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.

If the target in a pointer assignment is a variable, then one of the following must be true:

- It must have the TARGET attribute.
- 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.
- It must have the POINTER attribute.

If the target of a pointer assignment is an array section, the array must have either the TARGET or the POINTER attribute.

Examples

```
INTEGER, POINTER, DIMENSION(:,:) :: p
! p is a pointer array.
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 = 9.4
```

Related Statements

POINTER, ALLOCATE, DEALLOCATE, and NULLIFY

Related Concepts

For more information about pointer association and pointer assignment, see Chapter 3, Data Types and Data Objects.

TRACE OFF

Stops the display of program flow by statement label.

```
TRACE OFF
```

Description

TRACE OFF can appear anywhere within a debug packet. After a TRACE ON statement, tracing continues until a TRACE OFF statement is encountered.

Related Concepts

See "DEBUG" for a detailed description of using TRACE OFF in debugging.

TRACE ON

Initiates the display of program flow by statement label.

```
TRACE ON
```

Description

TRACE ON is active only when the TRACE option appears in a DEBUG packet. Tracing continues until a TRACE OFF statement is encountered. TRACE ON remains active through any level of subprogram CALL or RETURN statement. However, if a TRACE ON statement is active and control is given to a program in which the TRACE option is not specified, the statement labels in that program are not traced.

TRACE ON makes a record of the statement label on the debug output file each time that it encounters a statement with an external statement label.

The TRACE ON statement takes effect immediately before the execution of the statement specified in the AT statement.

Related Concepts

See "DEBUG" for a detailed description of using TRACE ON in debugging

TYPE (Declaration)

Declares a variable of derived type.

```
TYPE (type-name) [[, attrib-list] ::] entity-list
```

type-name is the name of a previously defined derived type.

attrib-list is a comma-separated list of one or more of the following attributes:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

For information about the attributes, see the corresponding statements in this chapter.

Description

The TYPE declaration statement specifies the type and attributes of derived-type objects, sometimes called structured objects or simply structures. (Note that, as used here, structures is not to be confused with the structure defined by the Intel Fortran STRUCTURE statement.) A derived-type object may be an array, which may be deferred shape (pointer or allocatable), assumed shape (dummy argument), or assumed size (dummy argument).

Assignment is intrinsically defined for each derived type but may be redefined by the user. Operators appropriate to a derived type may be defined by procedures with the appropriate interfaces.

When a derived-type object is used as a procedure argument, the types of the associated actual and dummy arguments must be the same. For sequence derived types 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 nonsequenced types the same physical type definition must be used, typically accessed via host or use association, for both the actual and dummy arguments.

Examples

```
! Weather is a simple derived type with two
!  character components and two integer
!  components.
TYPE Weather
```

```
  CHARACTER(LEN=32) Place
  INTEGER High_temp, Low_temp
  CHARACTER(LEN=16) Conditions
END TYPE Weather

TYPE (Weather) July(num_ws, 31)
! A two-dimensional Weather array for July.
July(:,:) % Low_temp = -40
! Initialize all low temps in July.
TYPE Polar
! Polar is a derived type with two real
!  components that cannot be directly accessed
!  in Polar objects outside the module.
  PRIVATE
  REAL rho, theta
END TYPE Polar

! Point is a derived type with three
!  components, one of which is itself
!  of derived type.
TYPE Point
  REAL x, y
  TYPE (Polar) p
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 nested structure constructors.
```

Related Statements

INTERFACE, PRIVATE, PUBLIC, SEQUENCE, and TYPE (definition)

Related Concepts

See Chapter 3, Data Types and Data Objects for information about derived types.

TYPE (Definition)

The first statement of a derived type definition.

`TYPE [[, access-spec ] ::] derived-type-name`

access-spec is the keyword PUBLIC or PRIVATE.

derived-type-name is a legal Fortran 95 name.

Description

This statement introduces the definition of a derived type. A derived type name may be any legal Fortran 95 name, as long as it is not the same as an intrinsic type name or another local name (except component names and actual argument keyword names) in that scoping unit.

A derived type may contain an access specification (PUBLIC or PRIVATE attribute) or an internal PRIVATE statement only if it is in a module.

Examples

```
! This is a simple example of a derived type
!  with two components, high and low.
TYPE temp_range
  INTEGER high, low
END TYPE temp_range
! This type uses the previous definition for one
!  of its components.
TYPE temp_record
  CHARACTER(LEN=40) city
  TYPE (temp_range) extremes(1950:2050)
```

```
END TYPE temp_record
! This type has a pointer component to provide
!  links to other objects of the same type,
!  thus providing linked lists.
TYPE linked_list
  REAL value
  TYPE(linked_list),POINTER :: next
END TYPE linked_list
! This is a public type whose components
!  are private; defined operations
!  provide all functionality.
TYPE, PUBLIC :: set; PRIVATE
  INTEGER cardinality
  INTEGER element ( max_set_size )
END TYPE set
! Declare scalar and array structures of type
!  set.
TYPE (set) :: baker, fox(1:size(hh))
```

Related Statements

INTERFACE, PRIVATE, PUBLIC, SEQUENCE, and TYPE (declaration)

Related Concepts

See Chapter 3, Data Types and Data Objects for information about derived types.

TYPE (I/O)

Writes to standard output.

Description

The TYPE statement is a synonym for the PRINT statement and has the same functionality and syntax. It is provided as an Intel Fortran extension for compatibility with earlier versions of Fortran. For more information, see the description of "PRINT".

NOTE. *The* TYPE *statement as an* I/O *statement cannot appear as the first executable statement in a program unit. It must be preceded by at least one other executable statement. There are cases where the syntax permitted for* TYPE *as an executable statement conflicts with the* TYPE *statement used to declare a derived type. Placing some other executable statement first in the program unit after all declarations causes the compiler to be able to determine what kind of statement* TYPE *is.*

UNION

Defines a union within a structure.

```
UNION
  map-block
  map-block
  .
  .
  .
END UNION
```

map-block is one or more of the following:

- A TYPE(I/O) declaration statement
- Another nested STRUCTURE
- A nested RECORD
- A UNION definition

Description

The UNION statement is an Intel Fortran extension that is used with the MAP statement to define a union within a structure. For detailed information about the MAP and UNION statements, see the description of the STRUCTURE statement in this chapter.

USE

Provides controlled access to module entities.

A USE statement has one of the following forms:

- USE *module-name*
- USE *module-name*, *rename-list*
- USE *module-name*, ONLY : *access-list*

rename-list is a comma-separated list of *rename*

rename is *local-name* => *module-entity-name*

access-list is a comma-separated list of the following:

- [*local-name* =>] *module-entity-name*
- OPERATOR (*operator*)
- ASSIGNMENT (=)

Description

The USE statement provides access to a module's public specifications and definitions. These include declared variables, named constants, derived-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.

All USE statements must appear after the program unit header statement and before any other statements. More than one USE statement may be present, including more than one referring to the same module.

Modules may contain USE statements referring to other modules; however, references must not directly or indirectly be recursive.

The local-name in a renaming operation is not declared: it assumes the attributes of the module entity being renamed.

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 *access-list*, with possible renaming of some module entities.

Entities made accessible by a USE statement include public entities from other modules referenced by USE statements within the referenced module.

The same name or specifier may be made accessible by means of two or more USE statements. Such an entity must not be referenced in the scoping unit containing the USE statements, except where specific procedures can be distinguished by the overload rules. A rename or ONLY clause may be used to restrict access to one name or to rename one entity so that both are accessible.

Examples

```
MODULE rat_arith
  TYPE rat
   INTEGER n, d
  END TYPE
  TYPE(rat), PRIVATE, PARAMETER :: &
                zero = rat(0,1)
  ! All entities are public except zero.
  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 Statements

MODULE

Related Concepts

Modules, scope, and association are discussed in Chapter 7, Program Units and Procedures.

VIRTUAL

Declares an array.

`VIRTUAL` *`array-declarator-list`*

`array-declarator-list` is a comma-separated list of array declarators.

Description

The `VIRTUAL` statement is provided as an extension in Intel Fortran for compatibility with earlier versions of Fortran. It is an alternative to the `DIMENSION` statement. `VIRTUAL` cannot be used as an attribute in type declaration statements.

Example

```
VIRTUAL A(10), B(1:5,2:6)
```

Related Statements

`DIMENSION`

Related Concepts

Arrays are discussed in Chapter 4, Arrays.

VOLATILE

Provides for data sharing between asynchronous processes.

`VOLATILE [::]` *`object-name-list`*

`object-name-list` is a comma-separated list of the following:

- variable-name
- array-name
- common-block-name

Description

It is only necessary to declare an object as VOLATILE when its value may be altered by an independent asynchronous process or event (for example, a signal handler). All optimization processes are inhibited for objects with the VOLATILE attribute. Data objects declared as VOLATILE will be addressable by otherwise independent processes.

If an array or common block is declared as VOLATILE then all of the array elements or common block variables are considered VOLATILE. Similarly, use of EQUIVALENCE with a VOLATILE object implies that any associated object is also volatile.

Examples

```
INTEGER alarm, trem
EXTERNAL wakeup
COMMON/FLAGS/ialarm
VOLATILE ialarm
! Set an alarm to execute in 60 seconds.
trem = ALARM(60,wakeup)
wakeup
IALARM = 0
DO
  IF (ialarm.NE.0) EXIT
END DO
SUBROUTINE wakeup
 COMMON/flags/ialarm
 VOLATILE ialarm
 ialarm=1
END
```

WHERE (Statement and Construct)

Performs masked array assignments.

```
WHERE (array-logical-expr)
[array-assignment-statement]
```

If the optional array-assignment clause is present, the WHERE statement is syntactically complete and does not require the END WHERE statement.

If the array-assignment clause is not present, the WHERE statement is the first statement of a WHERE construct. The syntax of the WHERE construct is:

```
WHERE (array-logical-expr)
  array-assignment-statement
  ...
[ ELSEWHERE
  array-assignment-statement
  ... ]
END WHERE
```

array-logical-expr is a logical array expression.

array-assignment-statement is an array assignment statement.

Description

Certain array elements can be selected by a mask and assigned in array-assignment statements using the WHERE statement or WHERE construct. *array-logical-expr* establishes the mask.

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.

The following rules and restrictions apply:

- The shape of the result of *array-logical-expr* and the arrays in each *array-assignment-statement* must be the same; they may be of size zero.
- *array-assignment-statement* must be an intrinsic array assignment statement; no defined assignment statements are permitted.
- Each elemental operation in *array-assignment-statement* is masked by the array logical expression.
- The elements of the arrays that are used in the WHERE part (the assignments after the WHERE keyword) are those corresponding to the true elements of the array logical expression. The elements of the arrays that are used in the ELSEWHERE part (the assignments after the ELSEWHERE keyword and before the END WHERE keywords) are those corresponding to the false elements of the array logical expression.
- Each *array-assignment-statement* executes in the order in which it appears in both the WHERE and ELSEWHERE part of the WHERE construct.
- In a WHERE construct, only the WHERE statement may be a branch target statement.

Examples

```
REAL, DIMENSION(150) :: a, recip_a
REAL(DOUBLE), DIMENSION(10,20,30) :: b, sqrt_b
! Assign 1.0/a to recip_a only where a is
!  nonzero.
WHERE( a /= 0.0 ) recip_a = 1.0 / a
WHERE( b .GE. 0.0 ) ! Assign to sqrt_b only
          !  where b is nonnegative.
  sqrt_b = SQRT(b)
ELSEWHERE  ! Set sqrt_b to 0.0 where b is -ve.
  sqrt_b = 0.0
END WHERE
INTEGER, DIMENSION(no_of_tests, student):: score
CHARACTER, DIMENSION(no_of_tests, student) &
                 :: letter_grade
! Assign letter grades for 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 = 'E'
```

In the next example, the arrays `values`, `delta`, and `count` must all be of the same shape:

```
WHERE (ABS(values) .LT. 10.0)
  values = ABS(values) + delta
  count = count + 1
ELSEWHERE
  values = 0
  count = count + 1
ENDWHERE
```

The first two assignment statements are processed for elements corresponding to true elements of the mask. The second two assignment statements are processed for elements corresponding to false elements of the mask. Unlike the `ELSE` clause of an `IF` statement, the assignment statements in both the `WHERE` and `ELSEWHERE` parts are processed.

Note the different behavior of the calls to `ABS`. In evaluating the mask expression, the entire `VALUES` array is passed to `ABS`, producing an array result whose elements are then compared to `10`. In the assignment statement, however, `ABS` is only invoked for those particular elements of `VALUES` corresponding to true elements of the mask. Also, note the mixed use of arrays and scalars in the assignment statement expressions.

The mask expression must have the same shape as the arrays in the assignment statements, but it might involve completely separate arrays. In the following example, `A`, `B`, and `C` can be independent of `D` and `E`, as long as they are all conformable:

```
WHERE (a+b .EQ. c) d = SIN(e)
```

The following example illustrates why the order of processing is important for dependency reasons:

```
REAL a(100)
REAL b(100)
EQUIVALENCE b,a
WHERE(a(1:20:1) .GT. 0) a(20:1:-1) = -1.0
WHERE(a(61:100:2) .LT. 1) &
        b(20:1:-1) = a(1:20:1) * 100.0
```

In the first WHERE statement, changing elements of a in the assignment might be thought to affect the mask expression. However, because the mask is evaluated before the assignment is processed, the behavior of WHERE statement is well defined. A similar situation arises in the second WHERE statement. Assignment values to elements of the assignment variable b alter the elements of the assignment expression a * 100.0. Because the assignment expression is evaluated for all true elements of the mask before any transfer of values to B, the behavior is again well defined.

It is important to note that assignment statements in a WHERE construct are processed sequentially. In the next example, the second assignment is not processed until the first is completely finished. This means that the values of b used in the second assignment have been modified by the first statement:

```
WHERE (SQRT(ABS(a)) .gt. 3.0)
  b = SIN(a)
  c = SQRT(b)
ENDWHERE
```

Related Statements

END WHERE and ELSEWHERE

A WHERE statement may be nested within a FORALL construct, or a FORALL construct may be nested within a WHERE construct.

Related Concepts

Elemental intrinsic functions, conformable arrays, and array language are described in Chapter 4, Arrays.

WRITE

Outputs data to external and internal files.

`WRITE (`*`io-specifier-list`*`) [`*`output-list`*`]`

`output-list` is a list of comma-separated data items for output. The data items can include expressions and implied-`DO` lists; see Chapter 8, I/O and File Handling for more detailed information.

`io-specifier-list` is a list of the following comma-separated I/O specifiers:

`[UNIT=]`*`unit`*

specifies the unit connected to the output file. *`unit`* can be one of the following:

- The name of a character variable, indicating an internal file
- An integer expression that evaluates to the unit connected to an external file
- An asterisk, indicating the preconnected unit 6 (standard output)

If the optional keyword `UNIT=` is omitted, *`unit`* must be the first item in *`io-specifier-list`*. This is the only specifier required in *`io-specifier-list`*.

`[FMT=]` *`format`* specifies the format specification for formatting the data. *`format`* can be one of the following:

- An asterisk (`*`), specifying list-directed I/O. For detailed information about list-directed I/O, see Chapter 8, I/O and File Handling.
- The label of a `FORMAT` statement containing the format specification.
- An integer variable that has been assigned the label of a `FORMAT` statement.

- An embedded format specification. For information about embedded format specifications, see Chapter 9, I/O Formatting.

If the optional keyword FMT= is omitted, *format* must be the second item in *io-specifier-list*.

NOTE. *The* NML= *and* FMT= *specifier may not appear in the same* *io-specifier-list*.

[NML=]*name* specifies the name of a namelist group for namelist-directed output. *name* must have been defined in a NAMELIST statement. If the optional keyword NML= is omitted, *name* must be the second item in the list. The first item must be the unit specifier without the optional keyword UNIT=.

The NML= and FMT= specifier may not both appear in the same *io-specifier-list*.

ADVANCE=*character-expression* specifies whether to use advancing I/O for this statement. *character-expression* can be one of the following arguments:

'YES'	Use advancing formatted sequential I/O default.
'NO'	Use nonadvancing formatted sequential I/O.

If the ADVANCE= specifier appears in *io-specifier-list*, *unit* must be connected to an external file opened for formatted sequential I/O. Nonadvancing I/O is incompatible with list-directed and namelist I/O.

For more information about nonadvancing I/O, see Chapter 8, I/O and File Handling.

ERR=*stmt-label*	specifies the label of the executable statement to which control passes if an error occurs during statement execution.
IOSTAT=*integer-variable*	returns the I/O status after the statement executes. If the statement executes successfully, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.
REC=*integer-expression*	specifies the number of the record to be written to the file connected for direct access. This specifier cannot appear in *io-specifier-list* with the NML= and ADVANCE= specifiers, nor with FMT=* (for list-directed I/O).

Description

The WRITE statement transfers data from internal storage to an external or internal file. An external file can be opened for sequential access or direct access I/O. If it is opened for sequential access, the WRITE statement can perform the following types of I/O:

- Formatted
- Unformatted
- List-directed
- Namelist-directed

If the file is opened for direct access, the WRITE statement can perform formatted or unformatted I/O.

WRITE statements operating on internal files can perform formatted or list-directed I/O.

For detailed information about files and different types of I/O, see Chapter 8, I/O and File Handling.

Examples

The examples in this section illustrate different uses of the WRITE statement.

Nonadvancing I/O

```
CHARACTER(LEN=80) :: prompt
WRITE (6, '(I4)', ADVANCE='NO') prompt
```

The WRITE statement outputs to the file connected to unit 6, which is preconnected to standard output. The ADVANCE='NO' specifier indicates the following:

The file has been opened for formatted sequential I/O.

The statement uses nonadvancing I/O to write an integer formatted as four characters from the variable prompt.

The effect of the nonadvancing WRITE is to output the character string in prompt to standard output without a terminating newline. This means that anything subsequently entered by the user will appear on the same line.

Internal File

```
CHARACTER(LEN=80) :: cfile
WRITE (cfile, '(I5, F10.5)') i, x
```

The statement writes a string of characters into the internal file cfile, using the embedded format specification to perform the format conversion.

Namelist-directed I/O

In the next example, each of the four WRITE statements following the NAMELIST statement uses a different style of syntax to do exactly the same thing:

```
NAMELIST /nl/ a, b, c
WRITE (UNIT=6, NML=nl) ! 6 = standard output
WRITE (6, nl)
WRITE (*, NML=nl)  ! * = standard output
WRITE nl       ! assume standard output
```

List-directed I/O

```
WRITE (6, *) int_var
```

This statement converts the value of `int_var` to character format and outputs the character string to standard output. The format conversion is based on the type of `int_var`. If you knew the format, you could substitute for the asterisk one of the following:

- The label of the `FORMAT` statement with the format specification, as in:

```
WRITE (6, 100) int_var
100 FORMAT(I4)
```

- An embedded format specification itself, as in:

```
WRITE (6, '(I4)') int_var
```

Unformatted Direct-access I/O

```
WRITE (31, REC=rec_num, ERR=99, IOSTAT=ios) a, b
```

This statement outputs to the file connected to unit 31. The `REC=` specifier indicates that the file has been opened for direct access and that this statement will output to the record whose number is stored in the variable `rec_num`. If an I/O error occurs during the execution of the statement, an error number will be stored in `ios`, and execution control will branch to the executable statement at label `99`.

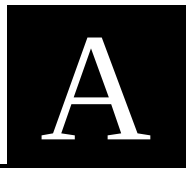
Related Statements

`CLOSE`, `OPEN`, `PRINT`, and `READ`

Related Concepts

For information about I/O concepts, see Chapter 8, I/O and File Handling, which also gives example programs that perform I/O. For information about I/O formatting, see Chapter 9, I/O Formatting.

Intel Fortran Extensions



This appendix lists all of the Intel Fortran extensions to the Fortran 95 Standard. It does not include nonstandard features that are enabled by command-line options (see the *Intel Fortran Compiler User's Guide* for the command-line options).

The following sections are organized according to the chapters in which each extension is described. If an extension is described in more than one place, additional references are included.

NOTE. *Most of the extensions provide compatibility with features found in other implementations of Fortran. If it is important that your program have maximum portability, you should avoid using the extensions. By default, the compiler will issue warnings for all non-standard features in your program. If you want to suppress these warnings, use /w90 and /cm options.*

Language Elements

The following extensions are described in Chapter 2, Language Elements:

- Symbolic names exceeding 31 characters in length
- Up to 99 continuation lines
- Extended Unix Character (EUC) encoding for multibyte characters in string constants, comments, and file names
- Dollar sign ($) accepted as alphabetical character in names
- Tab formatting in fixed format source files

- D debug lines in fixed format source files
- Use of pound (#) character in column 1 to denote comment
- Alternative statement continuation, using an ampersand (&) in fixed format

Data Types and Objects

The following are the extensions presented in Chapter 3, Data Types and Data Objects, and detailed in Chapter 10, Intel Fortran Statements:

- BYTE statement
- DOUBLE COMPLEX statement
- REAL*16 statement
- POINTER (Cray-style) statement
- VAX structures (see also the STRUCTURE statement in Chapter 10)
- Alternate form for initializing data: (*variable-declaration* / *constant-list* /
- Byte length notation (**n*) in type declaration; for example, INTEGER*8
- AUTOMATIC, STATIC (Statement and Attribute), VIRTUAL, and VOLATILE statements
- Use of Q exponent for quadruple-precision real constants
- CHARACTER and noncharacter items in the same common block
- COMMON always saved
- Alignment of variables in common blocks
- Equivalencing of character and noncharacter data (see also the EQUIVALENCE statement in Chapter 10)
- Alternate syntax for binary, octal, and hexadecimal constants
- Initialization of character variables with unsigned integers
- Structures and records (see also the STRUCTURE and RECORD statements in Chapter 10)

Array Concepts

The following extensions are described in Chapter 4, Arrays:

- Alternate array constructor syntax (/ /)
- Array subscripts of type real

Expressions

The following extensions are described in Chapter 5, Expressions and Assignment:

- Allow *a* * * - *b*
- Use of binary, octal and hexadecimal constants in expressions; type can be other than integer, as determined by context (see also typeless constants in Chapter 3, Data Types and Data Objects)
- Use of Holleriths in expressions with type determined by context (see also Hollerith constants in Chapter 3, Data Types and Data Objects)
- Use of noninteger expressions as array subscripts
- Logical operations on integers
- Integer operations on logicals
- The `.XOR.` operator (equivalent to `.NEQV.`)

Execution Control

The following extensions are described in Chapter 6, Execution Control:

- Branches into statement block of `DO`, `CASE`, or `IF` constructs
- Extended-range `DO` loops (see also the `DO` statement in Chapter 10, Intel Fortran Statements)

Scope, Program Units, and Procedures

The following extensions are described in Chapter 7, Program Units and Procedures:

- `%VAL`, `%LOC`, and `%REF` built-in functions (see also the `CALL` statement in Chapter 10, Intel Fortran Statements.
- Alternate return labels preceded by ampersand `(&)` character
- Initialization of blank common in a `BLOCK DATA` subprogram (see also `BLOCK DATA` statement in Chapter 10, Intel Fortran Statements).

Attributes

To specify additional information about a variable, variable types, and subprograms and their formal arguments, you can use a set of directives called attributes. Attributes allow you to do the following:

- Pass arguments by reference or value
- Use segmented or unsegmented addresses
- Use calling conventions of Microsoft C or Pascal
- Specify the span of a formal argument beyond one segment
- Specify and external name for a subprogram or common block.

You can use attributes in subroutine function definitions, type declarations, and with the `INTERFACE`, `INTERFACE TO` and `ENTRY` statements. The syntax of an attribute is as follows:

`!MS$ATTRIBUTES [`*`attribute-option`*` ] :: `*`variable-list`*

For example:

```
INTERFACE
   SUBROUTINE FOO(X)
      !MS$ATTRIBUTES C :: FOO
   END SUBROUTINE FOO
END INTERFACE
```

In the preceding example, the subroutine FOO has been given the C attribute. This ensures that calling procedure will use the Microsoft C calling convention to call FOO.

A synonym to `!MS$ATTRIBUTES [`*`attribute-option`*` ]` is `!DEC$ATTRIBUTES`. The functionality of both attributes is the same.

Restriction

When using the `!MS$ATTRIBUTES` directives, some of them affect the declaration of a variable, array or procedure name used in the directive. For this type of the compiler directive, Intel Fortran Compiler enforces a restriction that the directive must appear before the first executable statement in the procedure that contains the directive.

Correct usage:

```
FUNCTION MYFUNC ()
   IMPLICIT NONE
   !MS$ATTRIBUTES DLLEXPORT:: MYFUNC
   CHARACTER*8 MYFUNC
   MYFUNC='XXXXXXXX'
RETURN
END
```

Incorrect usage:

```
FUNCTION MYFUNC ()
    IMPLICIT NONE
    CHARACTER*8 MYFUNC
    MYFUNC='XXXXXXXX'
    RETURN
    !MS$ATTRIBUTES DLLEXPORT:: MYFUNC
END
```

The first executable statement in the example is `MYFUNC='XXXXXXXX'`. Placing the `DLLEXPORT` declarative directive before the executable statement permits successful compilation.

Compatibility with Microsoft Attributes

The following attributes are Intel Fortran extensions that are provided for Compatibility with Microsoft Attributes:

Table A-1 Attributes and Associated Objects

Attribute	Declarations in Variables and Arrays	EXTERNAL Statements and Subprogram Specifications	Common Block Names
ALIAS	Yes	Yes	Yes
ALLOCATABLE	Yes (arrays only)	No	No
C	Yes	Yes	Yes
DLLEXPORT	Yes	Yes	Yes
DLLIMPORT	Yes	Yes	Yes
EXTERN	Yes	Yes	Yes
FAR	Yes	Yes	Yes
HUGE	Yes	No	No
IVDEP	Yes	Yes	No
LOADDS	No	Yes	No
NEAR	Yes	Yes	Yes
PASCAL	Yes	Yes	Yes
REFERENCE	Yes	Yes	Yes
STDCALL	No	No	No

Table A-1 Attributes and Associated Objects (continued)

Attribute	Declarations in Variables and Arrays	EXTERNAL Statements and Subprogram Specifications	Common Block Names
VALUE	Yes	Yes	Yes
VARYING	No	No	Yes

Each of these attributes is described in detail in the following sections.

NOTE. *In all cases, the* `DLLEXPORT` *and* `DLLIMPORT` *attributes must appear in an* `INTERFACE` *block.*

ALIAS

Specifies an external name for a subroutine.

Syntax

`ALIAS : string`

where string is a character constant.

Example

```
SUBROUTINE OldName [ALIAS:'NewName']
```

In the preceding example, the `NewName` is given to the subroutine `OldName`.

Description

Within the source file you can only refer to a subprogram by its name given in the declaration. `ALIAS` overrides the `C` attribute. If you use the `C` attribute on a subprogram along with the `ALIAS` attribute, the subprogram will be given the `C` calling convention but not the `C` naming convention. Consequently, it will receive the `ALIAS` name with no modifications. This means that the new name is case sensitive, which is useful when you interface with case-sensitive language like `C`.

ALLOCATABLE

The Microsoft attribute `ALLOCATABLE` is provided for compatability with older programs that use this attribute. It permits you to delay allocation of storage for a particular declared entity until some point at run time when you explicitly call a storage allocation routine. The routine ensures that storage for that entity is dynamically allocated. In general, you should use the standard `FORTRAN ALLOCATABLE` statement and attribute instead (see Chapter 3, Attributes). The Microsoft attribute is declared as:

```
DIMENSION A[allocatable](:)
END
```

An entity declared with the `ALLOCATABLE` attribute must have a deferred-shape.

You can allocate storage for the array as follows:

```
DIMENSION A[allocatable](:)
ALLOCATE(A(10))
END
```

C Attribute

Defines the subprogram as having the same calling conventions as a Microsoft C procedure.

Syntax

`INTEGER[c] : argument`

where `INTEGER[c]` is a character constant.

Description

Arguments to subprograms with the C attribute are passed by value unless you specify the formal argument with the REFERENCE attribute. Subprograms that use the C attribute are modified automatically so that you can more easily match the naming conventions used in C. External names are matched to lowercase and start with an underscore (_).

When you assign an integer variable the C attribute, it becomes a C variable and assumes the default size according to the microprocessor on the system.

DLLEXPORT, DLLIMPORT

These Microsoft attributes allow you to import symbols to dynamic link libraries (DLLs) and to create FORTRAN DLLs that export symbols to other programs. You can use the DLLIMPORT and DLLEXPORT attributes on both data objects and routines.

When you use the DLLEXPORT attribute, you are asking for the identifier associated with the attribute to be exported to other programs or DLLs. The advantage to allowing other programs to access a routine in your DLL is that the routine doesn't have to actually be linked into the program, allowing the executable to be smaller.

To create a FORTRAN DLL, follow these steps as an example:

```
ifl /c test_dll.f90
ifl /LD add_dll.f90
ifl /Fetest_dll.exe test_dll.obj add_dll.lib
where test_dll.f90 is:
PROGRAM TEST_DLL

INTERFACE
   SUBROUTINE ADD (A,B)
     !MS$ATTRIBUTES DLLIMPORT :: ADD
     INTEGER A, B
   END SUBROUTINE ADD
END INTERFACE
!INTEGER P, Q
!P = 1
!Q = 2
CALL ADD (1, 2)
END PROGRAM TEST_DLL
And add_dll.f90 is:
   SUBROUTINE ADD (A,B)
  !MS$ATTRIBUTES DLLEXPORT :: ADD
  INTEGER A, B
  PRINT *, "Value of A = ", A
  PRINT *, "Value of B = ", B
```

```
  PRINT *, "Sum of A + B = ", A+B
END SUBROUTINE ADD
```

The /LD switch to ifl is a linker switch that tells the compiler driver to execute a link step to create the DLL.

EXTERN

Indicates that a variable is allocated in another source file.You must use EXTERN when you are accessing variables used in other languages. You cannot apply EXTERN in formal arguments.

FAR

Use the FAR attribute to specify that the argument is to be passed using a segment address. When used with variables, it specifies that the variable is allocated in far data areas.

HUGE

Use HUGE to specify that a formal argument or an allocatable array can span more than one segment. You can also use the $LARGE metacommand to specify the same thing. For example, the following two statements provide the same result:

```
FUNCTION Func (a(HUGE))
DIMENSION a(300)

$Large: a
FUNCTION Func (a)
DIMENSION A(300)
```

NOTE. *The compiler does not ensure that* HUGE *is specified for all arguments that span more than one segment.*

IVDEP

Use this directive immediately preceding a counted loop to instruct the compiler to vectorize the loop, regardless of any apparent dependences that would otherwise prevent vectorization.

The format of this directive is:

```
!MS$ IVDEP
```

The synonyms to `!MS$ IVDEP` are:

```
!DIR$ IVDEP
!DEC$ IVDEP
```

Use this directive when you know that the assumed loop dependences are safe to ignore. For example, the following loop:

```
DO WHILE (I .LE. N)
  A(I) = A(I+K) * C
......
```

will not vectorize with the `IVDEP` directive, since the value of K is not known (vectorization would be illegal if K<0).

For more information on compiler vectorization, refer to the *Intel¨ Fortran Compiler User's Guide*.

LOADDS

Use the `LOADDS` attribute to direct the compiler to create a separate data segment for the data within that procedure. The base address (`DGROUP`) of this new segment is automatically loaded into DS when the procedure is called. The separate data segment allows the procedureÕs data to be called with 16-bit NEAR references rather than 32-bit FAR references in order to speed up data references.You can only apply this attribute to separately compiled subprograms or functions. The default data segment for the program is automatically reloaded when execution of the procedure terminates.

Use the `LOADDS` attribute for user-written routines that are to be included in an OS/2 dynamic linking library (DLL). You do not need it for procedures that run in DOS programs because the command-line option /ND (name data segment) automatically ensures that the new data segmentÕs base address is loaded. The following is an example of the `LOADDS` attribute:

```
REAL*8 FUNCTION [LOADDS] GetNewData
```

NEAR

Use the NEAR attribute to specify that the actual argument is in the default data segment and that only its offset is passed to the subprogram. This attribute can also be used with common blocks. Common blocks that have the near attribute are mapped into the default data segment.

The syntax for the NEAR statement is as follows:

```
COMMON [/[name][NEAR]/]...
```

The parameter name is the name of the common block. When no *name* is specified, all blank common blocks are put in the default data segment. You must specify NEAR for at least the first definition of the common block in the source file. You can also specify NEAR for any COMMON statement in a subprogram.

To make a common block near, try to specify NEAR for all definitions of the common block. However, if you are modifying an existing program, it is easier to add a subroutine at the beginning of your source file to make common blocks near in the remainder of the program.

The advantage to having common blocks at the beginning of your program is that you can specify addresses with offsets only. This generates smaller, more efficient, code. If you do not specify NEAR, the compiler uses segmented addresses to refer to everything in common blocks.

If you specify a common block near one compiland but not in another, it will be mapped into the default data segment. The compiland that recognizes it as near will use short addresses, and the other will use long addresses.

Actual arguments passed to a near formal argument must be in the default data segment. You cannot pass any of the following to a near argument.

- Data in common blocks that are not specified using the NEAR attribute
- Arrays specified using the HUGE attribute
- Arrays defined while the $LARGE metacommand is active
- Variable named in a $LARGE metacommand

PASCAL

Use the PASCAL attribute to identify a subprogram as having the following characteristics of Microsoft PASCAL:

- The argument or the subprogram arguments are passed by value (unless the REFERENCE attribute is specified).
- Microsoft FORTRANÕs calling conventions are still used

- You can only use the PASCAL attributes with subprograms, common blocks, and formal argument type declarations (but not on formal arguments in the formal arguments list).

REFERENCE

Passes an argument by ÒreferenceÓ rather than by value. This means that the address (memory location) is passed to the subroutine rather than the argumentÕs actual value. This is similar to obtaining values using a pointer instead of copying the value and then passing it.

STDCALL

The `STDCALL` attribute allows you to call external routines written in C that have been compiled with the `STDCALL` option. To determine how to compile your C routines with `STDCALL`, consult your C compiler documentation. `STDCALL` changes the name of the external symbol that the Fortran compiler emits for your routine, unless you also have an `ALIAS` attribute specified. Most Win32* API functions assume that you will call them using `STDCALL`.

When you compile with the `/Gz` compiler command line option, all of the routines in the source file compiled with that option are compiled for `STDCALL`. You can also declare that a given routine should be compiled for `STDCALL` using Microsoft attributes.

Example

```
INTERFACE
INTEGER(4) FUNCTION CREATEEVENT(lpEventAttributes,
bManualReset, bInitialState,& lpName)
TYPE SECURITY_ATTRIBUTES
  SEQUENCE
    INTEGER(4) NLENGTH
    INTEGER(4), POINTER :: LPSECURITYDESCRIPTOR
    LOGICAL(4) BINHERITHANDLE
  END TYPE
TYPE (SECURITY_ATTRIBUTES), POINTER ::
lpEventAttributes
```

```
    LOGICAL(4) bManualReset
    LOGICAL(4) bInitialState
    INTEGER(1), POINTER :: lpName
   !MS$ATTRIBUTES STDCALL, ALIAS:'_CreateEventA@16'
::  CREATEEVENT
  END FUNCTION
END INTERFACE
```

Usage

When you compile a routine for STDCALL, parameters are passed to the routine by value. Normally, Fortran routines have their parameters passed by reference, that is, the address of each parameter is passed, rather than its value. For this reason, it usually is not advisable to try to compile a Fortran routine for STDCALL, and attempt to call it from Fortran. Arrays and CHARACTER variables are still passed by reference. You must exercise some care when calling a routine that is compiled for STDCALL from Fortran and passing a CHARACTER argument.

STDCALL will pass only the address of the CHARACTER argument, just as if your routine were written in C, and you were passing a `char *v` argument. Usually Fortran passes both the address and the length of a CHARACTER variable. The C routine that you are calling will expect that it is receiving a null-terminated string. Fortran normally pads strings with blanks.

You can declare a null-terminated CHARACTER constant in Fortran by placing a C after the constant. For example:

```
"This is a null terminated string" C
```

This facility, which can be used in assignment or data statements, allows you to set up the string correctly for passing it to C. You must be careful to allow for the extra unseen null character when you declare a length for a CHARACTER variable to hold the constant. For example, the string above is 32-characters long. To place it in a CHARACTER variable, the CHARACTER variable must be declared CHARACTER*33. You can also add null characters to Fortran CHARACTER strings by using explicit escape characters:

```
"This is a null terminated string, too\0"
```

Compiling a routine for STDCALL changes the name of the external symbol that the compiler emits for the call. An underscore (_) is added as a prefix of the name, and a suffix of @n is added to the end of the name, where n is the total length in bytes of parameters passed to the routine. The external symbol name is also changed to all lower-case letters. Note that in the example above, the ALIAS attribute is also used, so the prefix and suffix must be made explicit in the ALIAS – the compiler will not add them automatically when the ALIAS attribute is used.

Some other compilers, such as Compaq* Visual Fortran*, use a calling convention as their default which causes the names of external symbols to be changed in a manner *similar* to the STDCALL calling convention, but still passes parameters by reference. This is not the same as STDCALL, and is not supported in Intel Fortran Compiler.

NOTE. *The* STDCALL *attribute can only be used on Win32* with IA-32 systems.*

You can only use the STDCALL on routine names. You can use STDCALL with FORTRAN-77-style syntax as follows:

```
EXTERNAL BAR
!MS$ATTRIBUTES STDCALL :: BAR
CALL BAR
END
```

This results in a call to the external symbol _bar@0.

VALUE

Specifies that the argumentÕs value is to be passed rather than a reference to its memory location address. In either C or Pascal, the attribute is specified on the subprogram definition, and all arguments are assumed to be passed by value because this is the default.

If you specify VALUE and your argument is of a different type, a type conversion is required and it you must perform it before the call.

In C, arrays always pass by a reference to the memory location rather than by value. If you specify the C attribute and your subprogram has an array argument, the array is passed as a Struct (a C aggregate type). To pass an array so that it is handled as an array instead of a Struct you must do one of the following:

- Use the REFERENCE attribute on the formal argument.
- Pass the address returned by the LOC, LOCNEAR, or LOCFAR functions by value.

Example

```
SUBROUTINE SubFoo (x[VALUE])
INTEGER x[VALUE]
```

Integer X is passed by value from SubFoo.

VARYING

In FORTRAN, you must define a formal argument for each actual argument. Languages like C let you have arguments without having you specify formal arguments for them. The actual arguments are implicitly passed by value without automatic data-type conversion. You can specify VARYING when you specify the C attribute and this lets the actual number of arguments differ from the formal number.

However, actual arguments must still follow the type rules of the formal arguments. When you are writing a Fortran procedure with VARYING, use only arguments that you actually passed or your results will be undefined.

The VARYING attribute has no effect unless you have also specified the C attribute on the subprogram.

I/O and File Handling

The following extensions are described in Chapter 8, I/O and File Handling:

- Unit numbers can exceed 99
- ACCEPT, TYPE (as synonym for PRINT), ENCODE, and DECODE statements (see also Chapter 10, Intel Fortran Statements)
- Auto-opening of files
- Alternate form for namelist-directed input records
- Use of integer or real array as internal file

I/O Formatting

The following extensions are described in Chapter 9, I/O Formatting:

- R and Q edit descriptors
- Use of $ edit descriptor to suppress newline
- Default field widths for data edit descriptors
- Omission of comma between edit descriptors
- Use of A edit descriptor for any type
- Relaxation of rules governing data types that may be edited by certain repeatable edit descriptors
- Use of integer array to contain format specification

Statements

The following statements and attributes are extensions and are described in Chapter 10, Intel Fortran Statements:

- ACCEPT
- AUTOMATIC
- BYTE
- DECODE
- DOUBLE COMPLEX
- ENCODE
- MAP
- POINTER (Cray-style)
- RECORD
- STATIC
- STRUCTURE
- TYPE (I/O)
- UNION
- VIRTUAL
- VOLATILE

Intrinsic Procedures

Nonstandard intrinsic procedures provided in Intel Fortran are listed and described in *Intel Fortran Compiler User's Guide*.

Miscellaneous

The following extensions are described in the referenced chapters:

- Output from `fpp` is accepted (see the description of the `/fpp` option in the *Intel Fortran Compiler User's Guide*).
- Compiler directives (see the *Intel Fortran Compiler User's Guide*).
- Shift-JIS encoding for multibyte characters.
- Use of the `DATA` statement for variables in common outside `BLOCK DATA` subprogram (see `DATA` statement in Chapter 10, Intel Fortran Statements).

Glossary

actual argument	A value, variable, or procedure that is passed by a call to a procedure (function or subroutine). The actual argument appears in the source of the calling procedure. See also dummy argument.
allocatable array	A named array with the `ALLOCATABLE` attribute whose rank is specified at compile time, but whose bounds are determined at run time. Storage for the array must be explicitly allocated before the array may be referenced.
argument	(1) A variable, declared in the argument list of a procedure or `ENTRY` statement, that receives a value when the procedure is called (a dummy argument). (2) The variable, expression, or procedure that is passed by a call to a procedure (an actual argument).
argument association	The correspondence between an actual argument and a dummy argument during execution of a procedure reference.
argument keyword	A dummy argument name. Argument keywords can be used to pass actual arguments to a procedure in any order if the procedure has an explicit interface.

array	A rectangular pattern of elements of the same data type. The properties of an array include its rank, shape, extent, and data type. See also bounds and dimension.
array constructor	A rank-one array represented as a sequence of scalar or array values that may be constant or variable.
array element	An individual, scalar component of an array that is specified by the array name and, in parenthesis, one or more subscripts that identify the element's position in the array.
array section	A subset of an array specified by a subscript triplet or vector subscript in one or more dimensions. For an array `a(4,4)`, `a(2:4:2,2:4:2)` is an array section containing only the evenly indexed elements `a(2,2)`, `a(4,2)`, `a(2,4)`, and `a(4,4)`.
array-valued	Having the property of being an array.
assumed-shape array	An array that is a dummy argument to a procedure and whose shape is assumed (taken) from that of the associated actual argument. An assumed-shape array's upper bound in each dimension is represented by a colon `(:)`. See also assumed-size array.
assumed-size array	An older FORTRAN 77 feature. An array that is a dummy argument to a procedure and whose size (but not necessarily its shape) is assumed (taken) from that of the associated actual argument. The upper bound of an assumed-size array's last dimension is specified by an asterisk `(*)`. See also assumed-shape array.
attribute	A property of a constant or variable that may be specified in a type declaration statement. Most attributes may alternately be specified in a separate statement. For instance, the

	ALLOCATABLE statement has the same meaning as the ALLOCATABLE attribute, which appears in a type declaration statement.
automatic array	An explicit-shape array that is local to a procedure and is not a dummy argument. One or more of an automatic array's bounds is determined upon entry to the procedure, allowing automatic arrays to have a different size and shape each time the procedure is invoked.
automatic data object	A data object declared in a subprogram whose storage space is dynamically allocated when the subprogram is invoked; its storage is released on return from the subprogram.
bit	A binary digit, either 1 or 0. See also byte.
blank common	A common block that is not associated with a name.
block	A series of consecutive statements that are treated as a complete unit and are within a SELECT CASE, DO, IF, or WHERE construct.
block data subprogram	A procedure that establishes initial values for variables in named common blocks and contains no executable statements. A block data subprogram begins with a BLOCK DATA statement.
bounds	The minimum and maximum values permitted as a subscript of an array for each dimension.
byte	A group of 8 contiguous bits starting on an addressable boundary. See also gigabyte, kilobyte, megabyte, and terabyte.
character	A digit, letter, or other symbol in the character set.
character string	A sequence of zero or more consecutive characters.

column-major order	The default storage method for arrays in Fortran 95. Memory representation of an array is such that the columns are stored contiguously. For example, in the array `a(3,4)` element `a(1,2)` follows `a(3,1)`, which follows `a(2,1)` in memory. See also row-major order.
common block	A block of memory for storing variables. A common block is a global entity that may be referenced by one or more program units.
command-line option	A flag that can be specified with the `f90` command line to override the default actions of the Intel Fortran compiler.
compiler directive	A specially-formatted comment within a source program that affects how the program is compiled. Compiler directives are not part of the Fortran 95 Standard. In Intel Fortran, compiler directives provide control over source listing, optimization, and other features.
complete executable	An executable program that is created using only archive libraries and thus contains its own copy of the library routines referenced in the program. See also incomplete executable.
component	A constituent that is part of a derived type. A derived type may consist of one or more components. For example, `time%hour` refers to the `hour` component of `time` (and `time` is a variable whose data type is a derived type defined in the program).
conformable	Two arrays are conformable if both arrays have the same rank (number of dimensions) and the same extent (number of elements for each dimension). A scalar is conformable with any array.

connected	(1) A unit is connected if it refers to an external file. (2) An external file is connected if a unit refers to it. In both cases, connection is established either by the `OPEN` statement or by preconnection. See also preconnected.
constant	A data object that retains the same value during a program's execution. A constant's value is established when a program is compiled. A constant is either a literal constant or a named constant.
constant expression	An expression whose value does not vary during the program's execution. A constant expression's operands are all constants.
construct	A series of statements that begins with a `SELECT CASE`, `DO`, `IF`, or `WHERE` statement and ends with a corresponding `END SELECT`, `END DO`, `END IF`, or `ENDWHERE` statement.
data type	A named category of data that has a set of values, a way to denote its values, and a set of operations for interpreting and manipulating the values. Fortran 95 intrinsic data types include character, complex, double precision, integer, logical, and real. Intel Fortran also provides the byte and double complex data types as extensions. See also derived type.
deferred-shape array	An allocatable array or a pointer array (an array with the `ALLOCATABLE` or `POINTER` attribute).
defined assignment	A non-intrinsic assignment statement that is defined by an `ASSIGNMENT(=)` interface block and a subroutine.

defined operator	An operator that is present in an `INTERFACE` statement and has its operation implemented by one or more user-defined functions.
definable	A variable is definable if its value may be changed by its name or designator appearing in an assignment context (for example, in a `READ` statement or on the left-hand side of an assignment statement).
demand-loadable	A process is demand-loadable if its pages are brought into physical memory only when they are accessed.
derived type	A user-defined (non-intrinsic) data type that consists of one or more components. Each component of a derived type is either an intrinsic data type or another derived type.
designator	A name that references a part of a data object that can be defined and referenced separately from other parts of the data object. A designator may be a derived type component, array section, array element, substring, or actual argument with `INTENT(INOUT)` or `INTENT(OUT)`.
dimension	Each subscript of an array corresponds to a dimension of the array; arrays may have from one to seven dimensions. The number of dimensions is an array's rank. See also extent.
directive	See compiler directive.
disassociated	A pointer that is disassociated points to no target. A pointer becomes disassociated following a `DEALLOCATE` or `NULLIFY` statement involving the pointer or by the pointer being associated with (pointing to) a disassociated pointer.

dummy argument	An entity whose name appears in the argument list of a procedure or `ENTRY` statement. It is associated with an actual argument when the procedure is called. The dummy argument appears in the source of the called procedure.
dusty deck program	An older, pre-FORTRAN 77 program. Presumably called a "dusty deck" program because it was stored on punched cards and has not been changed since. Such programs generally rely on nonstructured programming techniques such as the `GOTO` statement.
element	See array element.
elemental	To be elemental, an intrinsic operation, procedure, or assignment must apply independently to every element of an array or apply independently to the corresponding elements of a set of conformable arrays and scalars
equivalencing	The process of sharing storage units among two or more data objects by means of the `EQUIVALENCE` statement.
executable program	A set of program units, including one main program, that can be run as a self-contained program.
executable statement	An instruction that causes the program to perform one or more computational or branching actions.
explicit-shape array	An array with explicitly-declared bounds for each dimension.
explicit interface	A procedure interface whose properties (including the name and attributes of the procedure and the order and attributes of its arguments) are known by the calling program unit. A procedure may have an explicit interface in a scoping unit if it: • is described by an interface block, or • is an `INTERNAL` procedure, or • is a `MODULE` procedure

expression	A series of operands and (optionally) operators and parentheses that forms either a data reference or a computation.
extended operator	See defined operator.
extent	The number of elements in one dimension of an array.
external file	A file that is stored on a medium external to the executing program.
external procedure	A procedure that is not contained in a main program, module, or another subprogram.
file	A sequence of records (characters or values processed as a unit). See also external file and internal file.
function	A procedure that returns a value (the function result) and that can be referenced in an expression.
function result	The data object returned from a call to a function.
generic procedure	A procedure in which at least one actual argument may have more than one data type. Generic procedures may be intrinsic or user-defined.
gigabyte	1073741824 bytes (2^{30} bytes). See also byte.
global entity	A program unit, common block, or external procedure whose scope is the entire executable program.
host	A program unit or subprogram that contains an internal procedure or module.
host association	The process by which an internal procedure, module procedure, or derived type definition accesses the entities of its host.
incomplete executable	An executable program that is created using at least one shared library. Copies of shared library routines are not present in an incomplete

	executable; instead, the executable has a linkage table that lists the routines' addresses in the shared library. See also complete executable.
inquiry function	An intrinsic function whose return value provides information based on the principal arguments' properties and not the arguments' values.
intent	An attribute of a dummy argument that indicates whether the argument is used for transferring data into the procedure, out of the procedure, or both.
internal file	A variable that is used as a file storage medium for formatted I/O. Internal files are stored in memory and typically are used to convert data from a machine representation to a character representation by use of edit descriptors.
internal procedure	A procedure contained in a main program or another subprogram.
intrinsic	Assignment statements, data types, operations, and procedures are intrinsic if they are defined in the Fortran 95 Standard and may be used, without being defined, in any scoping unit.
keyword	See argument keyword and statement keyword.
kilobyte	1024 bytes (2^{10} bytes). See also byte.
kind type parameter	An integer parameter whose value determines the range for an intrinsic data type; for example `INTEGER(KIND=2)`. The kind type parameter also determines the precision for complex and real data types.
label	An integer, one to five digits long, that precedes a statement and identifies it with a unique number. A statement's label provides a way to transfer control to the statement or to reference it as a `FORMAT` statement.

library	A file that contains object code for subroutines and data that can be used by programs written in Fortran 95, among other languages. See also linker.
literal constant	A constant that does not have a name. A literal constant's value is written directly into a program. See also named constant.
linker	The linker resolves references in a program's source to routines that are not in the source file being compiled. The linker matches each reference, if possible, to the corresponding library routine.
loader	A loader takes an executable file, the output of a linker, and loads it into physical memory. While doing so, it changes the virtual address to the physical address and prepares the executable file for running by the operating system.
main program	The first program unit that starts executing when a program is run. The first statement of a main program usually is the PROGRAM statement.
megabyte	1048576 bytes (2^{20} bytes). See also byte.
module	A program unit that contains definitions of derived types, procedures, name lists, and variables that are made accessible to other program units. A module begins with the MODULE statement and its public definitions are made available to other program units by means of the USE statement.
module procedure	A procedure that is contained in a module and is not an internal procedure.
name	A letter followed by up to 254 alphanumeric characters (letters, digits, underscores, and $) that identifies an entity in an Intel Fortran program unit, such as a common block, dummy argument, procedure, program unit, or variable.

named constant	A constant that has a name. See also literal constant.
numeric type	A complex, double precision, integer, or real data type.
obsolescent feature	A feature defined in the FORTRAN 77 Standard that still is in common use but is considered to be redundant, such as the arithmetic `IF` statement. The use of obsolescent features is discouraged. The Fortran 95 Standard summarizes the obsolescent features.
operand	An expression that precedes or follows an operator. For example, in `a + b`, both `a` and `b` are operands.
operation	A computation performed on one or two operands.
operator	A sequence of one or more characters in an expression that specifies an operation. For example, in `a + b`, `+` is an operator.
option	See command-line option.
optional argument	A dummy argument that does not require a corresponding actual argument to be supplied when its procedure is invoked.
pointer	A variable that has the `POINTER` attribute, which enables it to reference (point to) variables of a specified data type (rather than storing the data itself).
pointer association	The process by which a pointer becomes associated with the storage space of its target. Pointer association occurs during pointer assignment or a valid `ALLOCATE` statement.

preconnected	Three input/output units are preconnected to files by the operating system and need not be connected by the `OPEN` statement. The preconnected units are: • Unit 5 (standard input) • Unit 6 (standard output) • Unit 0 (standard error)
present	An optional dummy argument is present in an instance of a procedure if it is associated with an actual argument passed by the invoking procedure.
procedure	A unit of program code that may be invoked. A procedure can be either a function or a subroutine.
program	A sequence of instructions for execution by a computer to perform a specific task. See also executable program.
program unit	A main program, a module, an external procedure, or a block data subprogram.
rank	The number of dimensions of an array. Scalars have a rank of zero.
record	A sequence of values treated as a whole within a file.
return value	See function result.
row-major order	The default storage method for arrays in C. Memory representation is such that the rows of an array are stored contiguously. For example, for the array `a[3][4]`, the element `a[1][0]` immediately follows `a[0][3]`. See also column-major order.
scalar	A data item that has a rank of zero and therefore is not an array.
scope	The part of an executable program in which a name or declaration has a single interpretation.

scoping unit	A derived-type definition, an interface body (excluding derived-type definitions or interface bodies it contains), or a program unit or subprogram (excluding any derived-type definitions, interface bodies, or subprograms it contains).
shape	An array's extent (number of elements) in each dimension and rank (number of dimensions).
size	The total number of elements in an array; the product of all its extents.
specific procedure	A procedure for which each actual argument must be of a specific data type. See also generic procedure.
statement	A sequence of characters that represents an instruction or step in a program. A single statement usually, but not always, occupies one line of a program. A statement may consist of multiple lines by using the ampersand (`&`) continuation character. Similarly, multiple statements may appear on a single line separated by semicolons (`;`).
statement function	A function that returns a scalar value and is defined by a single scalar expression.
statement keyword	A word that is part of a statement's syntax, such as `CHARACTER`, `DO`, `ELSE`, or `FORMAT`.
statement label	See label.
stride	The increment that may optionally be specified in a subscript triplet. If it is not specified, the stride has a value of one.
structure	A data object that is scalar and is of derived type.
structure component	See component.
subprogram	See procedure.

subroutine	A procedure that is referenced by a CALL statement; values returned by a subroutine are usually provided through the subroutine's arguments.
subscript	A scalar value within the bounds of one dimension of an array. To specify a single array element, a subscript must be specified for each of the array's dimensions.
subscript triplet	An array section specification that consists of a starting element, an ending element, and (optionally) a stride separated by colons (:).
substring	A contiguous segment of a scalar character string. Note that a substring is not an array section.
target	A named data object that may be associated with a pointer. A target is specified in a TARGET statement or in a type declaration statement that has the TARGET attribute.
terabyte	1099511627776 bytes (2^{40} bytes). See also byte.
type	See data type.
type declaration statement	A statement that specifies the data type and, optionally, attributes for one or more constants, functions, or variables.
unit number	A logical number that can be connected to a file to provide a means for referring to the file in input/output statements.
use association	The association of names among different scoping units as specified by a USE statement. See also module.
user-defined operator	See defined operator.
user-defined assignment	See defined assignment.

variable	A data object whose value may be defined and redefined during a program's execution. For example, array elements or array sections, named data objects, structure components, and substrings all can be variables.
vector subscript	A method of referencing multiple, possibly discontinuous elements of an array by using a rank-one array of integer values as a subscript.
zero-size array	An array with at least one dimension that has at least one extent of zero. A zero-sized array has a size of zero and contains no elements.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X - Z