Fortran 90/95

Concise Reference

Jerrold L. Wagener

Published by Absoft Corporation

Rochester Hills, Michigan

Concise Fortran 90/95 Reference

Jerrold L. Wagener, author

Absoft Corporation, Publisher
2781 Bond Street
Rochester Hills, MI 48309

This book was set by the author and published, in both printed and electronic form, in the United States of America by Absoft Corporation.

ISBN

Fortran 90/95 Concise Reference

Contents

Preface

Fortran has been the principal programming language of the scientific community since the mid-1950s and has evolved over that time. The 1978 standard version, called Fortran 77, saw extremely wide use. Fortran 90 is the next step in the evolution of Fortran; it supports all of the features of Fortran 77 as well as many new ones which can make programming easier and more efficient. The purpose of this book is to provide a concise yet complete reference for Fortran 90.

Though there are numerous examples, this reference is neither a programming nor Fortran 90 tutorial; it assumes some familiarity with Fortran 77 programming. Though focusing mainly on standard Fortran 90, a separate chapter on the Absoft implementation of Fortran 90 describes implementation-specific features, including extensions to support interoperability with other languages and to facilitate porting of extended Fortran 77 legacy code.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Chapter 1 summarizes various “structural” aspects of a Fortran 90 program: source form, program units (e.g., procedures, modules) “parts” (e.g., specification part, execution part), etc. it also indicated how to read the formal syntax rules (BNF), upon which heavy reliance is used throughout.

Chapter 2 describes the Fortran 90 intrinsic data types. Those familiar with Fortran 77 will recognize them all. The one new feature here is the type “kinds” and kind parameters. All implementations must provide at least two kinds of real (single and double precision), and may provide more; accordingly the Fortran 77 data type double precision is deprecated.

Chapter 3 describes the Fortran 90 features for user-defined data types, formally called derived types (as they are derived from the intrinsic types). This provides for the first time in standard Fortran, data structures, including dynamic (linked) structures, and most of the data abstraction capabilities of a modern object-oriented language. The generic procedure and data abstraction features are superb (but lacking are inheritance and polymorphism).

Chapter 4 describes the Fortran 90 array-processing language. this is one of the major features of Fortran 90, and gives the language much of the array power (albeit with Fortran syntax and efficiency) of APL. In addition to array-level operations, there are two new facilities for dynamic arrays, including a couple of flavors of allocatable arrays and.

Chapter 5 is an identification of many of the Fortran 90 features that are redundant with other (usually more modern) features. This includes the features “deprecated” in the Fortran 90 standard and which may therefore be removed in future versions of the Fortran standard.

Chapter 6 describes the Fortran 90 Input/Output facilities. There's not much new here beyond Fortran 77, additional file connection specifiers and name-directed (namelist) I/O being the main ones. These features require two chapters and 47 (large) pages to fully describe in the Fortran standard; they consume 127 (smaller) pages in the exhaustive Fortran 90 Handbook (see below).

Chapter 7 describes the Fortran 90 control structures. To the Fortran 77 **if** construct are added: a case construct and three modern loop constructs (do while, do forever with exit, a modern form of the indexed **do**).

Chapter 8 describes the Fortran 90 module program unit and its uses. This is a major addition to Fortran and provides flexible software "packaging", complete with information hiding (public, private) capabilities.

Chapter 9 describes the Fortran 90 procedure facilities. Much of this will be familiar to Fortran programmers, but there are some significant features new in Fortran 90. These include explicit procedure interfaces, user-defined generic procedures, module and internal procedures, and operator definition.

Chapter 10 summarizes Fortran 90's 113 intrinsic procedures. These include, in addition to the traditional numeric and character computational functions, numeric environmental inquiry functions, array-processing and inquiry functions, bit-processing procedures, a few intrinsic subroutines.

Chapter 11 is the complete BNF for Fortran 90, extracted from the Fortran standard. This is *the* rigorous description of the Fortran syntax, and this reference relies very heavily on it; in many cases reference to the syntax rules in chapter 11 is the extent to which the syntax is described, with the text devoted primarily to describing the semantics and constraints related to the syntax.

Chapter 12 describes various implementation-specific values (e.g., kind values), options (e.g., compiler options), and extensions in the Absoft implementation of Fortran 90.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

This reference is intended to completely describe all of Fortran 90, but being concise it may give short shrift to some of the more subtle details of the language. The recommended sources to pursue these details are the following:

1. The Fortran standard. A committee draft of the version of the Fortran standard currently under development is available on the Fortran standard committee's web site, http://www.ionet.net/~jwagener/j3. As of this writing standing document J3/007 is the working draft of the Fortran 2000 standard, a very substantial subset of which is the official definition of Fortran 90 (less five of the deprecated Fortran 90 features); because of copyright restrictions the Fortran 90 standard cannot be posted publically.
2. The Fortran 90 Handbook, by Adams, Brainerd, Martin, Smith, and Wagener. This book was published by McGraw-Hill in 1992, and is an exhaustive, 740-page reference on Fortran 90; it's chapter organization is the same as the Fortran 90 standard and it was intended as a "readable version" of that standard.

Jerry Wagener
June 1998
Jerry@Wagener.com

1 Program Structure

This concise reference to Fortran 90 summarizes all aspects of the language in a crisp, concise manner. This reference is neither a programming nor Fortran 90 tutorial; it assumes some (but not extensive) familiarity with Fortran 77 programming concepts.

This chapter summarizes various "structural" aspects of a Fortran 90 program: source form, program units (e.g., procedures, modules), "parts" (e.g., specification part, execution part), etc. It also indicates how to read the formal syntax rules (BNF) used throughout.

syntax

Chapters 1-9 rely heavily on, and make frequent reference to, chapter 11, which is a complete BNF description of Fortran 90; while there are numerous examples of syntax throughout, this reference primarily lets syntax rules (each with an "R" number) of chapter 11 "do the talking" for a complete and rigorous account of the Fortran 90 syntax. The various chapters describe the associated semantics; no semantics are omitted, though some (especially those related to the "deprecated" features) are indeed brief.

The characteristics of the formal BNF of chapter 11 are summarized at the beginning of that chapter. A somewhat more informal form often will be used in chapters 1-9 in describing a feature. For example: a Fortran 90 programmer-defined *name* is (R304):

> *letter* [*name-character*] ...
>
> where a *name-character* is a letter, (decimal) digit, or underscore(_) character. The maximum length of a name is 31 characters; letters may be either upper or lower case (the case is not significant in names and keywords).

Square brackets ([ ]) indicate optionality and square brackets followed by three dots ([ ] ...) indicate optionally repeated any number of times. Constraints (like the 31 limit for name lengths) will be mentioned. In those cases where the language imposes no constraints, an implementation might. To take an extreme example, the internal subprogram part of a function (say) may contain, according to the BNF, [*internal-subprogram*] ... without limit; clearly no contemporary implementation could cope with a function having ten trillion internal functions, even if the programmer could (!?).

Chapters 1-9 will contain a number of short, illustrative examples. The predominant style in these examples will be: keywords in all lower case (**if**, **end**), procedure and variable names starting with a lower-case letter (**x**, **midPoint**), main program, module, and derived-type names starting with an upper-case letter (**HelloWorld**, **RangeArithmetic**), constants in all upper case (**N**, **QUAD**), "free-form" source (described below), and modest indentation of internal structure. A (nonsense) example is:

```
program HelloWorld                  ! nonsense example
  use RangeArithmetic               ! this program uses a module
  real(QUAD) :: X(N)                ! a "quad-precision" array
  type(Range) :: m1=Range(0,6)      ! variable of user-defined type

  x(1) = midPoint(m1)
```

```
  if (x(1) > 0) then
    print *, "Hello, world; the midpoint of m1 is: ", x(1)
  end if
end program
```

There are no column restrictions in free-form source, except that lines have a 132 character limit, and an exclamation point (!) initiates an *end-of-line comment* (from where it begins to the end of that line). Blanks must be used to separate keywords and names (e.g., the blanks are significant in **program HelloWorld** and **use RangeArithmetic** above); the other blanks in the above example are used for purposes of readability, not because they are syntactically required by the language)[1]. The above program is syntactically legal Fortran 90 if (and only if) a module named **RangeArithmetic** exists and it defines integer constants **N** and **QUAD**, a derived-type named **Range**, and a real (or integer) function named **midPoint** that takes an argument of type **Range**.

Every Fortran program is a set of *program units*, each of which is a sequence of *statements* - the above example has nine statements, each on a separate line. This is the normal pattern, but there are two other aspects of free-form source, both used occasionally, but sparingly, in subsequent examples: (1) two (or more) statements can appear on the same line, separated by a semicolon (the second statement can be null if you want to simulate the C style of ending a statement with a semicolon); (2) a statement can be continued on the next line by ending the (first) line (before any end-of-line comment) with an ampersand (&).

program units

A Fortran program is a set of (related) individual program units (R201), consisting of one *main program* (R1101), zero or more *external subprograms* (R203), zero or more *modules* (R1104), and zero or more *block data* units (R1110). The Fortran standard does not specify "where" these parts reside on a computing system, but typically one or more complete units are placed in a file; each such file is a separately-compiled *compilation unit*. The order of compiling such compilation units is normally immaterial, except that a module must be compiled before any units using that module, and the main program should be compiled last if an executable program is to be prepared therefrom. (For small programs everything can be placed in one file, with the modules first and the main program last.)

---

1. In Fortran's original fixed-form source (see chapter 5) blanks are never significant (e.g., never required) except in character strings, but blanks could be used freely for improving readability. Blanks could even be used within keywords and names; for example: `integer` could be written `int eger`. In free-form source blanks are significant (token delimiters) and must not be used within a keyword, name, or constant (except character constants), and (one or more) must be used to separate (delimit) consecutive keywords and names. The exceptions to this rule are that the separating blanks are optional in the following keyword sequences: **block data**, **double precision**, **else if**, **go to**, **in out**, **select case**, and all that start with the word **end**. The nonletter nondigit characters serve as delimiters and blanks may be freely used with (on either side of) them.

A large program unit can be spread across multiple files and "put back together" with *include lines* in a compilation unit. The include line, which is not a Fortran statement (but rather a compiler directive), has the form **include** *file-name* and cannot have an end-of-line comment; during compilation the compiler replaces the include line with the contents (e.g., common block definitions) of the specified file. With the advent of modules in Fortran, there are not many situations in which the include line is needed.

Execution of a program begins with its main program, which may *call* procedures defined in external subprogram units and modules to perform computations. In addition, modules may provide the main program (or any of the procedures it calls) with various "global" entities, such as global constants and variables, type definitions, and procedure interfaces. Ignoring the (unnecessary and not recommended) "attribute" statements (R519-538) - see chapter 5 - the eight sections of these program units are as depicted in the following diagrams.

	main programs	***external subprograms***	***module program units***
1	**program** statement (R1102)	**function** statement (R1216) or **subroutine** statement (R1219)	**module** statement (R1105)
2	**use** statements (R1107) →	→	→
3	**implicit** statements (R540) →	→	→
4	specifications [a] (R422, R1201, R501) →	→	→
5	executable constructs (R215) →		see note[b]
6	**contains** statement (R1225) →	→	→
7	internal subprograms[c] (R211) →	→	module subprograms (R213)
8	**end** statement (R1103)	**end** statement (R1218, R1222)	**end** statement (R1106)

a. Section 4 may also contain **common** (R548), **data** (R529), **equivalence** (R545), **namelist** (R543), and **save** (R523) statements; **format** statements (R1001) may be included in sections 3-5 of main programs and external subprograms, and **entry** statements (R1223) may be included in parts 3-5 of external subprograms and part 7 of modules (but not in part 7 of main programs and external subprograms).

b. Module program units do not have section 5.

c. Internal subprograms must not contain internal subprograms (but module subprograms may).

Sections 1 and 8 define the beginning and end of the program unit, and are the only sections that are not optional. Section 2 provides the program unit access to any modules it uses. Section 3 modifies the implicit environment (or replaces it with **implicit none**). Section 4 is the heart of the *specification part* of the program unit; it is here that all the local constants and variables are declared and any derived types and interface blocks not pro-

vided by modules are defined. Section 5 is the *execution part*, which represents the primary computation part of the program unit; note that modules do not have an execution part (they are intended as a source of definitions used by other program units). Section 6 is a keyword that marks the beginning of any internal or module subprograms (section 7); sections 6 and 7 go together - if one is present the other must be also. Each internal or module subprogram has the same structure as an external subprogram, except that internal subprograms cannot themselves contain internal subprograms. Note that a module may access other modules (and pass their definitions on to other program units using that module).

statements

A Fortran statement is made up of keywords, names, expressions, and delimiters. Keywords are predefined words that have some special meaning; most statements start with a keyword (e.g., **print**, **end**, **integer**, **function**, etc.). Names are programmer-defined, as described above, and are used to give unique identifiers to variables, constants, procedures, modules, etc. Expressions are the "computational engines" that generate computed results; they are formed from operands and operators. Operands include constants, variables, and function calls, and are described in the next chapter and in other appropriate places throughout; operators are either special characters (**+**, **-**, *****, ******, **/**, **//**, **==**, **/=**, **<**, **<=**, **>**, **>=**) or letters enclosed in periods (**.eq.**, **.ne.**, **.lt.**, **.le.**, **.gt.**, **.ge.**, **.and.**, **.or.**, **.eqv.**, **.neqv.**, **.not.**, and user-defined operators, R311).

Delimiters include blanks (as described above), **=** (for assignment - see below), **%** (structure component selector, R612 and chapter 3), left and right parentheses (multiple "enclosing" uses), single and double quotes (character constant delimiters - see chapter 2), the comma (list separator), **:** and **::** (other separators), ! (comment initiator), & (statement continuation), and the semicolon (statement separation).

The only statements that do not begin with a keyword are the *assignment statement* (R735) and the *statement function* (R1226), which have similar forms; the statement function is described in chapter 9. The assignment statement has the form

variable **=** *expression*

and has the purpose of saving the value of a computation (expression) so that the value can be used in subsequent computations. The *variable* (R601-602) is where the value of the *expression* (R723) is saved (the previous value of the variable is replaced); the *variable* may be a scalar variable name, an array variable name (for results of an array expression - see chapter 4), an array element, an array section (see chapter 4), a structure component (see chapter 3), or a substring (see chapter 2). Normally the type and kind of the variable must be the same as the type and kind of the expression value, but this rule is relaxed when numeric values are involved (see chapter 2). The expression can represent an arbitrarily complex computation involving operators and operands; the main features of expressions are described in chapter 2, with expressions involving user-defined types in chapter 3 and whole arrays (and array sections) in chapter 4.

2 Intrinsic Data Types

Fortran 90 has “built-in” (*intrinsic*) data types and user-defined (*derived*) data types; the latter are described in the next chapter. The intrinsic data types may be categorized as the numeric types (integer, real, and complex) and the nonnumeric types (character and logical). Each of these may have any number of *kinds*, though an implementation need provide only one kind of integer, character, and logical, and two kinds (“single” and “double” precision) of real and complex. Typical additional kinds that an implementation may provide are long and short integers, “quad” precision for real and complex, additional national character sets (the default character kind is essentially ASCII), and one-bit (or one-byte) logically. Arrays of any of these intrinsic type/kinds are permitted, with full array operations appropriate for that type.

The kinds are specified by integer constants called *kind type parameters*, and allow types to be “parameterized”. Thus a given program can, for example, be run with some set (or all) of the real variables having single precision; on another run these variables could be double precision. Across procedure boundaries both the type and type kind parameters of associated actual and dummy arguments must match, and thus kind mechanism can be used to provide families of generic procedures in procedure libraries. Each type has a “default” kind that is used for variables for which a kind type parameter is not explicitly declared; for example, the default real (and complex) kind is “single precision” and the default character kind is (essentially) ASCII.

integer data type

The integer type is intended to represent integer numeric values, and as such is modeled by $v = s\sum d_i r^i$, $0 \le i < n$, where for integer value v, s is the sign (+1 or -1), r is the radix (see intrinsic function **radix**), n is the number of digits (see intrinsic function **digits**), and the d’s are digits in the base-r system.

An integer constant is specified with decimal digits, with an optional sign (+ or -) and an optional kind parameter (R403). Examples of integer constants are:

```
42

135843_LONG

-687
```

The second of these examples illustrates how the kind parameter is specified for constants (“attached” to the value by the underscore); in this case **LONG** is a named integer constant having a valid kind value provided by the implementation. (See the intrinsic functions **kind**, and **selected-int-kind** to determine the implementation’s kind values for integers and see the intrinsic functions **range**, **huge**, and **tiny** to determine the range for any integer kind.)

The usual operations are provided intrinsically for two integer operands: addition (**+**), subtraction (**-**), multiplication (*****), division (*/*), and exponentiation (******). Such operations all produce (the usual) integer results, the only potentially surprising one being integer division, which results in the truncated arithmetical result. Plus (**+**) and minus (**-**) may also be used as unary operators with integer operands with the usual results. The other intrinsic operations on integers are the relational operators **==**, **/=**, **<**, **<=**, **>**, **>=** (or, equivalently, **.**eq., **.**ne., .lt., .le., .gt., .ge.) for comparing two integer values. Respectively, these operations result in the logical value **.true.** if the first integer operand value is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to the second integer operand value, and **.false.** otherwise. Examples of operations involving integer operands are:

```
j + k
j - k * n              ! evaluated as j - (k * n)
(j - k) * n
j - k + n              ! evaluated as (j - k) + n
- j + k               ! evaluated as (-j) + k
-(j + k)
- j * k               ! evaluated as -(j * k)
(-j) * k
j / k * n              ! evaluated as (j / k) * n
j / (k * n)
k ** n
j < k
j + k > n - m          ! evaluated as (j + k) > (n - m)
```

Note that parentheses may be used in the usual way to specify evaluation order in expressions containing multiple operations. The usual default precedence (e.g., left-to-right and multiplication-before-addition) is used in the absence of parentheses, and the numeric operations take precedence over the relational operations. See R723 for a rigorous description of the precise rules for evaluation of expressions involving integer operands. Additional (user-defined) operations may be provided for integers - see chapter 9.

Though (arguably) Fortran does not provide an intrinsic bit data type (but see the logical type) a complete bit processing facility is provided via integer objects and intrinsic procedures **btest**, **iand**, **ibclr**, **ibits**, **ibset**, **ieor**, **ior**, **ishft**, **icshft**, **mvbits**, and **not**. These provide bit-by-bit operations on non-negative scalar integer values represented by binary digits (bits) according to the model $v = \sum b_i 2^i$, $0 \le i < m$. Integer constants may, alternatively, be specified or printed as a set of binary digits, octal digits, or hexadecimal digits, in accordance with R407, and R1005. The rightmost bit in the integer object is b_0 and, for example, the integer value **13** (or **B"1101"** or **O"15"** or **Z"D"**) represents the bit string **. . . 0001101**.

real data type

The real type is intended to represent the real, or floating-point, numeric values, and as such is modeled by $v = s\, r^e \sum d_i r^{-i}$, $0<i<q$. where for real value v, s is the sign (+1 or -1), r is the radix (see intrinsic function **radix**), q is the number of mantissa digits (see intrinsic function **digits**), the d's are the mantissa digits in base-r system and e is the exponent (an integer - see intrinsic function **exponent**). Intrinsic functions **epsilon**, **huge**, **minexponent**, **maxexponent**, **nearest**, **precision**, **range**, **rrspacing**, **scale**, **set-exponent**, **spacing**, and **tiny** provide access to the numeric properties of any real kind. A common implementation for the default real kind is the single precision IEEE floating point standard, for which r=2, q=24, and $-126 \le e \le 127$.

A real constant is specified with decimal digits, and optional sign (+ or -), an optional kind parameter, and either a fractional part, an exponent, or both; see R412 for the syntactic details of the various forms that a real constant may take. Examples of real constants are:

```
-15.24
2.6E7              ! value is 26,000,000.0
4.3
-22.9E22_QUAD
.123
123.
7E4                ! value is 70,000 in single precision
7D4                ! value is 70,000 in double precision
-1.1D-3            ! value is -0.0011 (double precision)
```

The fourth of these examples illustrates the use of a kind parameter in a real constant - **QUAD** is a named integer constant having a valid kind value provided by the implementation for reals. As mentioned above, an implementation must provide at least two kinds of real, informally known as single and double precision. (See the intrinsic functions **kind** and **selected-real-kind** to determine the implementation's kind values for reals.) Note that, unlike for integers, a real constant may not be representable exactly; for example the real constant 0.1, though within range for (most) real types, is not representable exactly in a binary implementation (that is, one for which r=2 - e.g., the IEEE floating point standard).

Exactly the same intrinsic operations as for integers are provided for real operands, with the same meanings, except that division works "normally". Because of the potential noted above for inexact representation of real values, the equality relational operators (**==** and ***/=***) may at times yield "unreliable" results and thus should be used with care (if at all) with real operands. As for integers, see R723 for a detailed descriptions of the syntax and semantics (precedence) of expressions involving intrinsic operators with real operands, and see chapter 9 for expressions involving user-defined expressions with real operands.

The last two examples above are *double precision constants*; using the **D** (instead of **E** or neither) makes the constant the double precision kind of real rather than the (default) single precision kind. The data type **double precision** (R502), before the advent of the *kind* concept in Fortran, was the only way that double precision objects could be declared. The **double precision** type is still available, but is now deprecated in favor of **real(kind(1D0))**; **double precision** and **real(kind(1D0))** represent identical type/kind combinations.

complex data type

The complex type is intended to represent complex numeric values, and as such is an ordered (real part, imaginary part) pair of reals; there is a complex kind for every real kind. A complex constant comprises values for the real part and imaginary part, separated by a comma and enclosed in parentheses, as described in R417. Note that integers may be used for these values, in which case they are converted to the equivalent real value; different kinds of reals may be used for these two values, in which case the one with the lower kind value is converted to the equivalent value of the other kind; the kind of the complex constant is the kind of the parts (after any conversion). Examples of complex constants are:

```
( 1.0, -1.0 )

( 5, 5.5E5 )

( -3.3_QUAD, 4.4_QUAD )
```

The complex type has the same intrinsic numeric operators (**+**, **-**, *****, ***/***, ******) as the other numeric types, with the usual meanings. However, since complex values do not have a natural ordering, the equality operators (**==** and ***/=***) are the only intrinsically defined relational operators for complex. Because comparison of complex values involves comparison of real values, the same comparison caveat applies to complex values as to real values. Though intrinsic meanings for **<**, **>**, etc., are not provided for complex, an application may provide such meanings as user-defined operators - see chapter 9.

Intrinsic functions **real** and **aimag** return the real and imaginary parts, respectively, of a complex argument. Function **conjg** returns the complex conjugate of a complex argument, and function **cmplx** allows construction of a complex value of any kind from any integer or real values (e.g., any integer or real expressions, not just constants) and conversion of complex values to different complex kinds. Intrinsic function **int** converts any integer, real or complex argument to any kind of integer value, and function **real** converts any integer, real or complex argument to any kind of real value.

logical data type

The logical type is intended to represent boolean (true, false) values. A logical constant is either **.true.** or **.false.**, optionally appended with a kind parameter, as described in R421. A default logical value is defined to require the same storage (most often 32 bits) as a default integer value and a default real value (both of which, by definition, require the same amount of storage - this unit of storage is called a *numeric storage unit*). The role of different kinds of logical is to allow for different amounts of storage (usually less) for logical

values than is required for default logical values. One bit is adequate for representing a true/false (1/0) value, though one byte (8 bits) is often used as well. A logical kind in which exactly one bit is used to represent a logical value has many of the properties of a bit data type (especially in the form of logical operations on logical arrays).

There are four intrinsic binary operators that take logical operands (**.and.**, **.or.**, **.eqv.**, **.neqv.**) and one unary operator (**.not.**). The results of these operations are shown in the following table, where T stands for **.true.** and F stands for **.false.**

	x=T	x=F	x=T	x=F	x=T	x=F	x=T	x=F	
y=T	T	F	T	T	T	F	F	T	F
y=F	F	F	T	F	F	T	T	F	T
	x .and. y		**x .or. y**		**x .eqv. y**		**x .neqv. y**		**.not.y**

Note that relational operations on numeric values result in logical values, which can then be used as operands in logical operations; for example: **x < y .and. y < z** (meaning x<y<z).

character data type

The character type is used to represent and process character strings. Character values may be any length, from one character to an implementation-defined (usually pretty large) number of characters. Arrays of character strings are allowed, but each element of such an array has the same length (same number of characters). Each character may be any one of the characters in the implementation-defined character set (the default character kind), which is often a subset or superset of ASCII. An implementation may provide additional character sets, with different character kind parameters; this is often used to support national character sets (e.g., kanji, greek, cyrillic, etc.).

A character constant is simply a character sequence delimited by (single or double) quotes and optionally *preceded* by a kind parameter (R420). A (single or double) quote delimiter may be included in a character constant by repeating it once. Examples of character constants are:

```
"now is the time"

'x'

GREEK_"μøπß"

"He said ""OK"", and left."          ! "OK"  in quotes
```

There is just one intrinsic character operator, the concatenation operator (**//**); it appends the second operand to the first, forming a longer character string:

```
"blue" // "whale"                    !  results in "bluewhale"
```

There are a number of intrinsic procedures that are very useful for a wide range of character processing, including the following functions: **len**, **index**, **trim**, **adjustl**, **scan**, etc. Two such functions, **char** and **ichar**, provide conversion between characters and integers. The characters have a *collating sequence*, which is a one-to-one mapping between the characters and (a subset of) the integers. Function **char** takes an integer argument and returns the character associated with that integer value in the collating sequence; function **ichar** does the reverse, returning the collating-sequence integer associated with the character argument.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

declarations

Making a specific variable name have a specific type is known as a data object declaration, *type declaration*, or just *declaration* for short. Declarations are used as well to establish named constants and to type function results. The simplest form of a type declaration (R501) is:

type-spec [**::**] *object-list*

where *type-spec* (R502) is, for intrinsic types, the type name (**integer**, **real**, **complex**, **logical**, **character**) with an optional *kind* specification and, for **character**, an optional *length* specification. Examples of declarations of simple, uninitialized scalar variables are:

```
integer :: i, j

integer(SHORT) :: k

real :: ab, cd

real(QUAD) :: eps

real(DOUBLE) :: p, q2

complex :: z1, z2

complex(QUAD) :: tp3

logical :: done

character(20) :: s1, s2, s3*30

character(10, GREEK) :: g, h5
```

The **::** is optional, but it makes a nice visual barrier for the more complicated declarations (to be described below) and so will be used consistently in examples henceforth. The kind parameters (**SHORT**, **QUAD**, **GREEK**) are named integer constants; the actual integer values could be used in place of the named constants, but since kind parameter values are not standardized, and differ from implementation to implementation, the better practice is to have the actual value appear only once - where the named constant is defined. If the kind value **DOUBLE** has the appropriate integer value, the fifth example above is equivalent to using (the deprecated) type name **double precision** (without a kind specification). The

same declaration is obtained by using **real(kind(1D0))** instead of **real (DOUBLE)**, and indeed the value of **DOUBLE** could have been specified, in its definition, as **kind(1D0)**.

All of the objects in a character type declaration have the length specified in the type-spec, unless overridden as illustrated by **s3*30** in the next-to-last example above; s1 and s2 both have length 20, while s3 has length 30. If the length specification is omitted, the value 1 is assumed; if the length is omitted and the kind is not, then the **kind=** keyword must be used; if both the length and kind are specified, with kind first, both the **kind=** and **len=** keywords must be used; otherwise the **kind=** and **len=** keywords are optional (and the **kind=** keyword is always optional in declarations of the other intrinsic types).

The above examples declared uninitialized scalar variables. The following examples illustrate how array objects can be declared and how objects (scalars or arrays) can be given initial values.

```
integer :: i, j(10)                    ! j is an array of 10 elements
integer(SHORT) :: k=0                  ! k has initial value zero
real :: ab(100), cd(100,100)           ! a one dimensional and a two dimensional array
real(QUAD) :: eps=0.000001
real(DOUBLE) :: p=0D0, q2              ! p initialized to (a double precision) zero
complex :: z1, z2=(1.0,-1.0)
complex(QUAD) :: tp3(1000,1000)
logical :: done=.false.
character(20) :: s1, s2(250), s3*30="the third degree ....."
character(10, GREEK) :: g, h5="μøπß"
```

Arrays can have up to seven dimensions, the specification of which are enclosed in parentheses, and separated by commas; each has the form [*lower* **:**] *upper*, where lower and upper are integer values which specify the lower and upper bounds of the subscript range for that dimension; if the optional part is omitted, the lower bound is 1. Arrays are described in detail in chapter 4, together with additional forms of array declaration (for dynamic and dummy argument arrays).

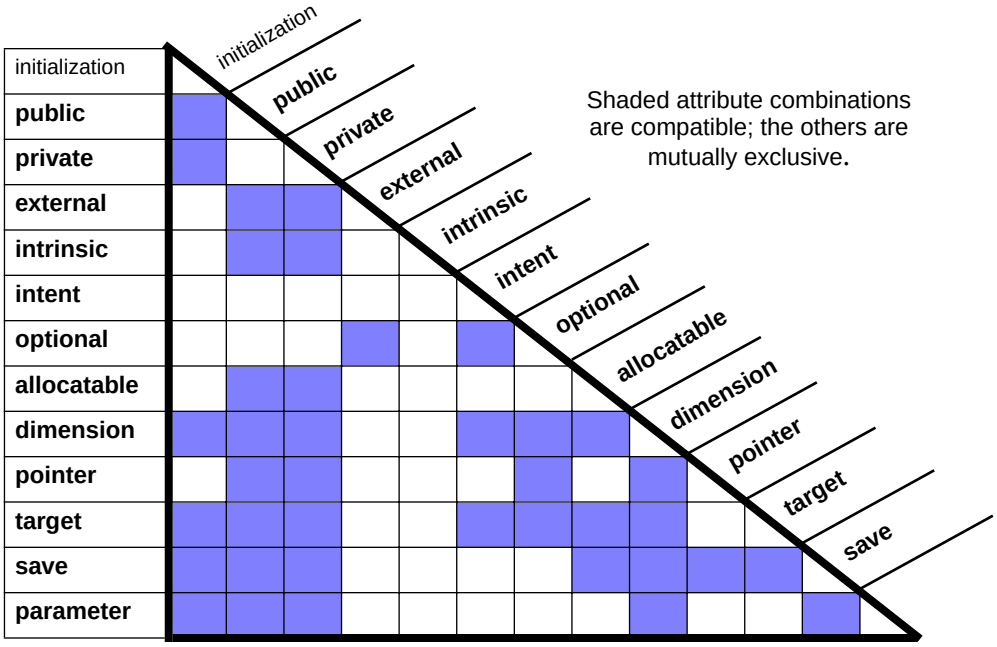
An initial value is specified by an *initialization expression*, which can by any expression (of an appropriate type) involving only constant values, with the following two exceptions: (1) operators must be intrinsic operators and any exponentiation operator (******) must have a second operand of type integer; (2) a procedure call must be to an intrinsic function that (a) can be evaluated at compile type and (b) except for the **reshape**, **transfer**, and inquiry functions, have arguments only of type integer and character and deliver a result only of type integer or character. A variable local to a procedure may be initialized, but that initialization is effective only at the first execution of the procedure and the variable, though saved (see below) is not reinitialized at the beginning of subsequent executions.

attributes

The properties of data objects are called *attributes*; the most basic attribute, one that every data object has, is type (and kind). The other attributes are optional and the two illustrated above are the array (**dimension**) attribute and the initial value (**data**) attribute. There are 11 more attributes, the specification of which requires either the attribute form of the type declaration statement (R501, R503) or a separate "attribute-oriented" statement for each such attribute (R519-539). The attribute form of the type declaration statement is

type-spec **,** *attribute-list* **::** *object-list*

The attribute-list can contain any of the attributes listed in R503, in any order, separated by commas; but any given attribute can appear at most once in the list, and over half of them are mutually exclusive (see the following table).



Attributes **public** and **private** control the visibility of module objects and are described in Chapter 8. Attributes **external** and **intrinsic** specify objects that are either external or intrinsic functions, respectively. Attributes **intent** and **optional** apply only to procedure dummy arguments (see chapter 9). Attributes **allocatable** and **dimension** apply to arrays and are discussed in chapter 4. Attributes **pointer** and **target** are used for dynamic structures, including dynamic arrays, and are discussed in chapters 3 and 4.

save

The **save** attribute is used to retain an object's value when the object "goes out of scope" and then comes back in. One example of this is local variables in a procedure; upon return from the procedure, its local variables normally cease to exist, the space may be reallocated for another purpose, and the variables are reinstantiated when the procedure is next executed. However, any such variable with the **save** attribute remains intact with its value unchanged between executions of the procedure, ready to "take up where it left off" when the procedure is next executed. Initialized variables (and named constants) automatically have the **save** attribute. A module variable goes "out of scope" and, without the **save** attribute, becomes undefined whenever all program units using that module are inactive (i.e., not executing). Similarly the **save** attribute is used to retain a common block and its variable values when all program units using that common block are inactive.

parameter

A data object with the **parameter** attribute must also be initialized, and that value cannot be changed - the object is a constant; it has a name and "looks" like a variable, and thus it is called a *named constant*. Named constants are declared identically to initialized variables, but with the **parameter** attribute. Named constants may be scalars or arrays, of any type, and are extremely important to reliable programming. A constant that appears more than once in a program, such as an array bound value or a kind parameter, should be a named constant, and then when a change is needed in that value the change can be made in only one place.

Examples of declarations with specified attributes are:

```
integer, parameter :: DOUBLE=kind(1D0)

real(DOUBLE), parameter :: PI=3.141592653589793

integer, private :: maxCount

complex, save :: z0=(1.0,-1.0)

real, allocatable, save :: workArray(:,:)

character(40),dimension(100) :: firstName, lastName, address1, address2

real, intent(in) :: length, width

logical, optional :: codeFlag

type(Tree), pointer :: left, right

real(DOUBLE), target, dimension(400,500) :: rho1, rho2, vel_x, vel_y, vel_z

complex, external :: f1, f2

complex, pointer :: temp(:)

logical(BIT), target, intent(out) :: digitalRadar
```

numeric computations

As mentioned in chapter 1, computations are formulated as the results of expressions; these results can then be assigned to variables, written (to the screen, the printer, or a file), used as actual arguments in procedure calls, or used for control purposes (a logical expression as an **if** condition, ann array subscript, an integer expression as the final index value in an indexed loop, etc.). In many respects, numerical expressions are the heart of Fortran and, in addition to the four numerical types (counting both of the required real kinds) and the numerical intrinsic operators, Fortran provides most of the elementary numerical functions and the basic vector and matrix computations as intrinsic functions. In addition, the facilities for providing user-defined computational libraries is very powerful.

Individual expression operands are defined in R701 and include constants (including named constants), variables (scalars and array elements), function calls, and subexpressions (expressions in parentheses). R723 defines expression structure in detail, including operator precedence. Without the operator precedence, R723 boils down to this:

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

The *binary-operator* precedence determines the execution order, but this can be controlled with parentheses - that is, by making each operand a (parenthesized) subexpression, leaving only one *binary-operator* at the top level; unparenthesized expressions are evaluated as if the parentheses needed to represent the operator hierarchy had been added. For numeric expressions the operators are *power-op* (R708), *mult-op* (R709), *add-op* (R710), and unary and binary user-defined operators (R704, R724) delivering numeric results. Note that R705-707 indicates that exponentiation, ******, (power-op) has precedence higher than any other numeric operator and works right-to-left for consecutive (unparenthesized) power-ops (R705); multiplication and division (*****, */*), both mult-ops, have the same precedence, which is higher than addition (**+**), and subtraction (**-**), both add-ops, and work left-to-right (R706); additional and subtraction also work left-to-right (R707).

If a numeric operation has two operands with the same type and kind, it delivers a result of that type and kind. Otherwise, the operation is “mixed-mode”, and one of the two operands (x_w) will have a type/kind combination that is “wider” than that of the other operand (x_n); in this case the value of x_n will be converted to the equivalent value in the x_w number system, the computation will be made at the x_w type/kind level, and the result will have the type and kind of x_w; operand value conversion is performed as described below for mixed-mode assignment. Any operand of type real is wider than any operand of type integer, and any operand of type complex is wider than any operand of type real. Of two integer kinds, the one having the greater integer range is the wider; normally this one will have the greater kind value as well. Of two real (or complex) kinds, the one having the greatest precision is the wider; normally this one will have the greater kind value as well. By this rule, double precision real/complex is wider than default (single precision) real/complex.

In numeric assignment, **v=e**, the *variable* **v** and the *expression* **e** can be any combination of the numeric types (integer, real, complex). The value assigned is *cf* **(e,kind(v))** where *cf* is the intrinsic conversion function **int**, **real**, or **cmplx**, depending on whether **v** is type integer, real, or complex, respectively.

character computations

Character expressions (with intrinsic operators) are all of the form:

character-operand [**//** *character-operand*] ...

There are no intrinsic character unary operators, and concatenation (**//**) is the only intrinsic character binary operator. Concatenation "works" left-to-right, but it doesn't matter since concatenation is associative (**A//(B//C)** has the same result as **(A//B)//C**). The length of a concatenation result is sum of lengths of the two operands.

Character operands can be character constants, character variables (including character array elements), functions delivering character results, character expressions in parenthesis, and substrings (R609-611). The form of a substring is:

parent-string **(** [*lower*] **:** [*upper*] **)**

where a *parent-string* can be a scalar character variable, a character array element, a character component of a structure, or a character constant; it cannot be a general character expression, a character functional call, or substring itself. *Lower* and *upper*, which may be arbitrary integer expressions, define the portion of the parent string that comprises the substring; both lower and upper must have values greater than zero and less than or equal to the length of the parent string. The length of the substring is upper-lower+1 (actually, max(0,upper-lower+1)) and comprises the characters from index *lower* in the parent string up to and including index *upper*. If lower==upper, then the substring is just one character - that at index *lower* (or *upper*); if *upper* is less than *lower* then the substring is empty (has length zero). Examples of substrings are:

```
s2(2:5)

lastName(k:)

address(i-1:index(address,'-'))

account % name(i:j)
```

The result of a character expression may, among other uses, be assigned to a character variable (including an array element) of the same kind or to a substring of such a variable. In either event the expression has a length (number of characters it produces) and the receiving variable has a length (number of characters it receives). If these lengths are the same, then the expression result becomes the value of the receiving variable. If the expression length is less than the receiving length, the expression result is padded on the right with the requisite number of blank characters so that the two lengths are the same and then the assignment is made; if the expression length is greater than the receiving length, the expression is truncated on the right to the receiving length, and then the assignment is made. Examples of character assignment are:

```
s1 = s1(1:4) // s2(5:)                  ! concatenate two substrings, then assign

s2(2:5) = "."                           ! replace substring with "...."

firstName(k+1) = firstName(n)           ! character array elements

firstName(k+1)(i:j) = firstName(n)(m:m+2)! substrings of array elements
```

implicit declaration

Unless **implicit none** is specified at the beginning of the program-unit (R205, R540), it is possible to use variables in the program that have not been explicitly typed in type declaration statements. Such variables will have the types specified by the implicit type environment in effect and are said to be *implicitly typed*. Such variables may be given other attributes with "attribute-oriented" statements (R519-539).

The implicit type environment associates a type/kind combination to each letter. A name not explicitly typed is implicitly typed according to its first letter. The default implicit type environment is that letter I-N are associated with type default integer and all others are associated with type default real. **implicit** statements (R540-542) may be used to change these associations for some or all of the letters, and this may include implicitly typing for the nondefault kinds and for user-defined types. **implicit none** is a special form of **implicit** statement that "turns off" all implicit typing and requires that each data object be explicitly typed. **implicit none** is not the default implicit environment (unfortunately); unless **implicit none** is explicitly specified in the program unit, each letter in that program unit has an associated implicit type/kind combination.[1]

1. An exception to the rule that "**implicit none** turns off all implicit typing" is in internal and module procedures having **implicit none** in their hosts; such procedures may have **implicit** statements defining implicit typing for part of the letters, leaving **implicit none** (and hence explicit typing required) for the other letters - see chapter 9.

3 User-defined Data Types

The intrinsic types are the numeric types (integer, real, complex), the logical type, and the character type. Each of these may be parameterized (type kinds), have intrinsically-defined forms for constant values, and have intrinsically-defined operators and operations. A number of intrinsic functions are defined for the intrinsic types, and user-defined functions may return values of intrinsic type. Fortran 90 has user-defined types as well, called *derived types* because they are derived from the intrinsic types (and other derived types); all aspects of an intrinsic type, except type kind parameterization, may be provided for a derived type - name, constants, operators, and functions returning derived type values. Derived types, once defined, essentially augment the intrinsic types as the data types available for use in a program. Two other uses of derived types are common: *record structures* and *dynamic structures*. Record structures are convenient for organizing data into logical groups (records) for input and output and corresponding processing. Dynamic structures, such as linked lists and binary trees, are indispensable for certain application areas.

derived-type definitions

Unlike intrinsic types, whose definitions are intrinsic (built into the language), a derived type must be defined. This is done by specifying its *components*, which are objects of intrinsic or derived type (R422). The simplest form for such a definition is:

```
type type-name
  component-definition                ! a derived type has one or
  [ component-definition ] ...        ! more components - see R426
end type [ type-name ]
```

Each component definition is an ordinary type declaration statement, except that the only attributes permitted are **dimension** and **pointer**. A component can be either a scalar or an array, and derived types for recursive dynamic structures (e.g., linked lists, binary trees) can have pointer components that have the same type as that being defined. More than one component may be declared in the same component definition statement. Simple examples illustrating all of these possibilities:

```
type Point                            ! a type to represent points
  real :: x, y                        ! in a two-dimensional
end type                              ! cartesian coordinate system

type Student
  character(30) :: ID
  integer :: homework(15)             ! can represent up to 15 homework grades
  integer :: hour_exam(3)             !              and 3 hour_exam grades
  integer :: final_exam
end type
```

```
type Tree                              ! the nature of the node data
  type (TreeData) :: node_data         !    is defined in another type;
  type (Tree), pointer :: left, right  ! two “links”, for left and right subtrees
end type

type Pixel
  type (Point) :: p                    ! the position on the screen;
  integer :: R, G, B                   ! primary color values for this point
end type
```

The first (and last) of these four examples do indeed suggest new data types, and the expectation would be that appropriate operations would be defined on objects of these types. The second example suggests a typical record structure, with an expectation that various computations would be made with individual fields (components) but not on “a student as a whole”. The third example clearly is intended to be used as a dynamic structure, in this case a binary tree with **left** and **right** links to subtrees. The data for such a tree can, of course, be of any nature, and in this case the details of that nature have been encapsulated in yet another derived type called **TreeData**.The fourth example also illustrates the use of a previously-defined derived type as a type component. Though not having the properties of true object-oriented inheritance, the Point component of Pixel does “inherit” all of the properties (e.g., operations) defined for type Point. (In this manner, and through use of type definitions packaged in modules, Fortran 90 provides much of the inheritance and data abstraction benefits of object-oriented programming; typical O-O features not accommodated are polymorphism and object invocation of procedures.)

The name of a derived type must not be the same as any intrinsic type name, nor the same as any other derived type accessible in that scope; it also must not be the same as any variable or procedure name accessible in that scope. Derived-type definitions are local to the scope in which they are defined, but may be accessible to other scopes via use and host association.

Component names have a scope of their derived-type definition, though they are accessible outside the type definition when (and only when) selected with the **%** component selection operator (see below). Within a given type definition, each component name must be unique, but otherwise there is no restriction on component names; in particular, they can be the same as the names of entities defined in or accessible to the surrounding scope of the type definition.

derived-type objects

Derived-type definitions do not create data objects; a type definition only specifies the name of the type (analogous to, say, the type name **integer**) and the component structure for objects of that type. Actual objects must be created, again in analogy with intrinsic types, in type declaration statements (R501); derived type objects are created with a *type-spec* of

type (*type-name* **)**

As with objects of intrinsic type, a derived type object, also called a *structure* or *structured object*, may be: declared as a constant (have the **parameter** attribute), an array (have the **dimension** attribute and, optionally, be an **allocatable** or **pointer** array), a dummy argument (and, optionally, have the **intent** and/or the **optional** attributes), a module object (and, optionally, have the **private** or **public** attribute), a pointer or a pointee (have the **pointer** or **target** attribute), a saved local variable (have the **save** attribute). Function results may be declared to be of derived type, in which case either the **type (** *type-name* **)** type specifier may appear on the function statement or the function may be typed in the function's specification part.

Some example declarations of derived-type objects:

```
type (Point) :: p1, p2

type (Point), allocatable :: property_corner(:)

type (Student), intent(in) :: valedictorian

type (Student), private :: onProbation (100)

type (Tree) :: dictionary

type (Tree), pointer :: root

type (Pixel), save :: screen (480,640)

type (Pixel) function summit (topo_map)
```

A structure reference may appear in an expression, on the left hand side of an assignment statement, and as an actual argument - as may a *component reference*. A component of a structure may be referenced by using the *component access operator*, **%**, in the following manner (R612):

structure-reference **%** *component*

Specific examples of component references (using the above example type definitions and structure object declarations) are:

```
p1 % x                          ! the x component of p1, a scalar real object
property_corner(i) % x          ! x component of the ith element of property_corner
property_corner % y             ! an array of real objects: all the y components
valedictorian % ID              ! a scalar character object
valedictorian % homework        ! an integer array - all valedictorian homework grades
onProbation(n) % hour_exam(m)   ! a scalar integer object
dictionary % node_data          ! scalar object of type TreeData
dictionary % left               ! the left subtree
screen(i,j) % R                 ! a scalar object
```

```
screen % B                  ! 2D array of integer objects
screen % p                  ! 2D array of Point objects
screen(i,j) % p % x         ! a scalar real object
```

The last of these examples illustrates that if a structure component is itself a structured object, then its components may be accessed as well. Such a sequence of **%** operators may be of any length, as long the entity on the left of each **%** is of derived type and the entity on the right is a component of that derived type; the "parsing" of such a sequence is from left to right. The very left one must be a structured object, and is either its declared name or (if the structure is an array) an element or section of that array. Each entity on the right of a **%** operator must be either a component name or (if the component is an array) an element or section of that array. Any array name (whole array) or array section in a component reference makes the result of the reference array valued, with the same shape; there can be at most one array-valued entity in a given component reference.

structure constructors

A structured object may be assigned a value by individually assigning a value to each of its components. For example:

```
p2 % x = 112.2              ! these two assignments
p2 % y = 35.7               ! completely define p2
```

A complete list of component values may be gathered together in a *structure constructor* to define a structure value; these may be used in any expression legitimate for a value of that type. A structure constructor has the general form (R430)

type-name **(** *component-value* [**,** *component-value*] ... **)**

and a specific example is

```
p2 = Point ( 112.2, 35.7 )
```

The effect of this last assignment is the same as the preceding individual component assignments; a structure value of a given type may be assigned to a derived-type variable of that type. (But this intrinsically-defined assignment may be overridden by a defined assignment - see chapter 20).

The structure value **Point (112.2, 35.7)** is a constant because all of the component values are constants. Such a value may be used to establish a named constant of derived type:

```
type (Point), parameter :: origin = Point (0.0, 0.0)
```

In general, however, any expressions may be used for the component values in a structure constructor, so long as the constructor contains an assignment-compatible value for each component, in the order in which the components are declared in the type definition.

An associated structure constructor is available for any defined or accessible type, (except for types with private components - see below). If a type has a pointer component (either

scalar or array) an allowable target object must appear in the corresponding position in a constructor for that type. If a component is an explicit-shape array (the only type allowed as a component other than a pointer array), an array value with that shape must appear in the corresponding position in a structure constructor.

Additional examples of structure constructors are:

```
Point (z+1, w-2)

Point ( r*sin(phi), r*cos(phi) )

Point ( 2*p2%y, 0.0)

Student ( "Joe", hw(i,:), (/e1,e2,e3/), fe )
```

In the last example **hw** must be a two-dimensional integer array with a size of 15 in its second dimension, and **e1**, **e2**, **e3** and **fe** must all be scalar integer variables (or named constants). The first three of these examples illustrate the use of arbitrary expressions to specify component values in a structure constructor.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Structure values may also be obtained from input. A structure variable name may appear in the input list of a **read** statement, in which case a value is input for each component. In list directed input the effect is the same as if the individual components had been placed (with the **%** operator) in the input list instead. For formatted input, a format is specified for each component, in component-declaration order, in exactly the same manner as for the intrinsic type **complex**.

```
read *, p1                          ! read two real values, one for p1%x and one for p1%y

read "(2F10.2)", p1

read *, valedictorian               ! consumes one character and 19 integer values

read "(A30,19I4)", valedictorian
```

Similarly, structure values may be placed in the output list of a **print** or **write** statement, and the component values are output in component-declaration order. Structures with pointer components can be neither input nor output, though individual nonpointer components of such structures can appear in I/O lists.

derived-type operators

Since dummy arguments may be of derived type, and functions may return structure values, functions may be used to define virtually any operation on and among structured objects. For example, the midpoint between two Point objects may be defined as follows:

```
type (Point) function midpoint(a,b); type (Point) :: a, b
  midpoint = Point ( (a%x+b%x)/2, (a%y+b%y)/2) )
end function
```

Note that the midpoint is, of course, a **Point** value. An operator, say **.mid.**, can be associated with this operation:

```
interface operator (.mid.)
  module procedure midpoint          ! assuming midpoint is an accessible module procedure
end interface
```

Then the midpoint of two points, say **p1** and **p2**, can be computed with the expression

```
p1.mid.p2                            ! as well as with the function call: midpoint(p1,p2)
```

Chapter 20 contains more details on defining operators. Chapter 19 illustrates how a module can be used to encapsulate a derived-type definition, its constants, and any operations, so that the type could be used almost as naturally as an intrinsic type.

private types

Sometimes it is desirable to hide the internal structure of objects and access components only via explicitly provided procedures. This is possible only for derived types defined in modules. A derived-type's components are accessible anywhere inside the module in which the type is defined, but the internal structure of a private type can be hidden from users of the module. This is accomplished with the derived-type **private** statement, an example of which is:

```
type Point; private
  real :: x, y
end type
```

In this case users of the module which contains this type definition can declare objects of type **Point**, but cannot use the **%** operator to access either the **x** component or the **y** component of those objects, nor is the **Point** structure constructor accessible to the user of the module. The module may contain functions to access **x** and **y**, but need not if the intended use of **Point** does not involve the user accessing the components. In this latter case the module presumably includes procedures that provide the intended computations on and with **Point** objects.

The **private** statement, if used, immediately follows the **type** statement (R422, R423) and hides all of the components from users of the module containing the derived-type definition. Note that

type *type-name*; **private**

is really two statements, and many programmers prefer to put them on separate lines, whereas

type, private :: *type-name*

is a single statement that makes the entire type, name as well as the components, private to the module and not accessible to users of the module.

sequence types

A derived-type actual argument must be exactly the same type as the associated dummy argument or an *equivalent type*. "Same type" means that both a dummy and its associated actual argument derive from the same type definition. For all practical purposes this means that the derived type must be defined in a module that is used by, or host to, (a) any procedure definition with arguments of that type and (b) any program unit making calls to such procedures. (The only way the requisite conditions can be met without a module is if all of the procedures having an argument of this type are internal procedures in a host procedure containing the type definition.)

"Equivalent types" are *sequenced* type definitions with the same type and the same number of components; corresponding components in each list must have the same name, the same or equivalent type, the same attributes (dimension and/or pointer), and neither can be private. An example of a *sequence* type is

```
type Student; sequence
  character (30) :: ID
  integer :: homework(15), hour_exam(3), final_exam
end type
```

This type definition need not be used from a module and can be repeated in any scope which has actual or dummy arguments of type Student. The main problem with such duplication is, of course, the maintenance one of needing to make a change in more than one place. This problem can be circumvented by putting the sequenced type definition in its own file and including that file wherever the type definition is needed; this, however, is not much different from putting the type definition in a module and using the module wherever the type definition is needed.

A derived type is made into a sequenced type by inserting a **sequence** statement between the **type** statement and the list of component definitions (R422, 423). There is one use of sequence types other than for equivalent type argument association. This is to allow derived-type objects to be associated (e.g., in common blocks) with numeric or character storage sequences. If all of the components of a sequence derived type have numeric storage units (i.e., they comprise only intrinsic kinds of integer, real, complex, and logical components), the type is a *numeric sequence type* and can be storage associated with any numeric storage sequence. Similarly, if all of the components of a sequence derived type are of type default character, the derived type is a *character sequence type* and can be storage associated with any character storage sequence.

4 Arrays

The array features of Fortran 90 represent one of the most significant aspects of the language. A computation can be specified on a whole array (or any portion of an array); such a computation is performed on each element of the array, conceptually concurrently. The corresponding potential for actual process parallelism is enormous - namely the number of elements in the array. Roughly speaking, versions of Fortran prior to Fortran 90 allowed the programmer to specify computations only on scalar entities, such as individual array elements, with an entire array processed by sequencing (looping) through its elements.

array-valued expressions

Suppose that **A**, **B**, and **C** are two-dimensional real arrays, all dimensioned at 200x300 elements. Fortran 90 allows the following statement, involving the addition of two of these arrays and the assignment of the result:

```
C = A + B
```

The meaning of this operation is **C(i,j) = A(i,j) + B(i,j)** for all 200 values of **i** and all 300 values of **j**, for a total of 60,000 individual (scalar) computations involving the array elements. The array computational model is *concurrent element-by-corresponding-element computation* for all elements of the arrays.

In addition to extending all of Fortran's scalar operations to arrays in this manner, other useful whole array operations are provided. These include *reduction operations* (e.g., **product(A)** returns the product of all the elements of array **A**), *construction operations* (e.g., **(/ (i, i=1,n) /)** constructs the vector [1, 2, 3, ..., n], and *inquiry operations* (e.g., **shape(B)** returns the shape of array **B** - a vector comprising the size of each dimension of array **B**, or [200, 300] for the array **B** in the above example). All such operations can be combined into more complex expressions; for example, **product(shape(B))** has the value 60,000, the total number of elements in **B** (but Fortran 90's rich supply of array functions also includes the **size** intrinsic function, which gives the same result more directly).

Generally speaking, except in a few contexts in which an expression is restricted to be scalar, any Fortran 90 expression may have array operands and the result is array valued. (Scalar expressions are required in control contexts such as **if** construct control (scalar logical expressions required), **do** loop indexing expressions, and I/O specifiers such as unit number, file names, open statement specifiers, etc.) In most cases the arrays in an array-valued expression must have the same shape (must be ***conformable***) and the expression value is an array of the same shape.

Note that in many cases, such as in the above example (**C = A + B**), array expressions appear indistinquishable from scalar expressions and one needs to know from other contexts (e.g., the specifications) that the variables have been declared as arrays. However, Fortran 90 allows such expressions to be written with explicit dimensionality, which clearly identifies array operations. For example, the above assignment can be written as:

```
C(:,:) = A(:,:) + B(:,:)          ! makes the "arrayness" explicit
```

Functions may be defined to return array values (*array-valued functions*) and calls to such functions may be operands in array-valued expressions. Array-valued functions, including both user-defined and intrinsic ones, make array-valued expressions a complete, natural extension/generalization of scalar expressions, with arrays replacing scalars as operands and results.

conformability and element-by-element computation

The principal requirement in forming an array expression is conformability of the operands. Each operand of an array operation must have the same rank and the same number of elements along each dimension as the other operands - that is, conformable arrays are arrays with exactly the same shape. The result of such an operation is, of course, conformable with the operands, and the value of each element of the array result is the scalar computation involving the corresponding elements of the array operands.

Thus if A and B are the following 2x3 arrays: $A = \begin{bmatrix} 2 & 3 & 5 \\ 1 & 7 & 4 \end{bmatrix}$ $B = \begin{bmatrix} 5 & 4 & 1 \\ 2 & 2 & 3 \end{bmatrix}$

the result of A + B is $\begin{bmatrix} 7 & 7 & 6 \\ 3 & 9 & 7 \end{bmatrix}$ and the result of A* B is $\begin{bmatrix} 10 & 12 & 5 \\ 2 & 14 & 12 \end{bmatrix}$

If there is more than one operation in an expression, the (array-valued) result of the first subexpression is an operand for the second operation, and so on, just as in scalar operations. For example, for A and B as given above, in the expression A + B * A, A is added to the result of B * A; thus the result of A + B * A is

$$\begin{bmatrix} 2 & 3 & 5 \\ 1 & 7 & 4 \end{bmatrix} + \begin{bmatrix} 10 & 12 & 5 \\ 2 & 14 & 12 \end{bmatrix} = \begin{bmatrix} 12 & 15 & 10 \\ 3 & 21 & 16 \end{bmatrix}$$

Note that, for example, a 3x2 array is not conformable with a 2x3 array - they have the same rank and total number of elements, but corresponding dimensions don't have the same size - and thus two such arrays cannot be the operands in the same array operation. The only exception to this basic conformability rule is in the event that one of the operands is a scalar. In this case the scalar is *broadcast* into an array conformable with the other operand, the value of each element of this broadcast array being that of the scalar. For example, B + 2 is a valid array operation and (assuming B is as given above)

the result of B+2 is $\begin{bmatrix} 5 & 4 & 1 \\ 2 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 7 & 6 & 3 \\ 4 & 4 & 5 \end{bmatrix}$

Common uses of (broadcast) scalars in array operations are to initialize and scale arrays:

```
A = 0                  ! sets each element of A to zero
B = (B + 1)/2          ! add 1 to each element of B then take half the result
```

This last example illustrates a key aspect of the Fortran array operations: in an array-valued assignment the effect is as if the right-hand side array value is fully evaluated before any assignment takes place. Otherwise it is possible (though not in this simple example) for the right-hand-side array value to be affected before its evaluation is complete. Thus the conceptual model is that all elements of the right-hand-side array value are computed concurrently (or in any order) before any assignment takes place, and any implementation is allowed that guarantees this behavior.

An example where this rule is important is in the pivoting step in Gauss elimination (see the last example in this chapter). There the pivot row is normalized with the array operation

```
G(P,:) = G(P,:)/G(P,K)
```

G(P,:) is the Pth row of the two-dimensional array **G**, and **G(P,K)** is the pivot element; the normalization scales the row so that the pivot element value is 1.0. Note that if the value of this element is changed to 1.0 before the evaluation of the right-hand side is complete, then the row is not properly normalized (a typical error in sequential scalar code). Therefore, array operations should not be thought of as "loops" over the array elements, as a loop implies a sequentially of the operations; in general, thinking of array operations as loops gives incorrect results when assignment is involved. Array operations should be thought of as integral/parallel computations.

array constants - array constructors

Array values may be explicitly constructed using the *array constructor* (R431) and, if the desired resultant array has dimension higher than one, the **reshape** intrinsic function; an array constructor forms a one-dimensional array. An array constructor is simply a list of the element values of the result, separated by commas and enclosed in **(/**... **/)** delimiters. These values can be any scalar expressions, as long as they all have the same type and type parameters. If they are all constants, however, then such a constructor (possibly combined with the **reshape** function) is an array constant and may appear in a **parameter** declaration.

Because lists of individual scalar values are not very practical for constructing large arrays, two forms for array constructor list items are provided in addition to scalar expressions. These are implied-do constructs (R433) and array expressions. The first of these has the form

(*expression-list* **,** *index-variable* **=** *first-value* **,** *last-value* [**,** *increment*] **)**

The index-variable is a scalar integer variable serving as an iterative index in exactly the same manner as in a **do** loop. The example above, **(/ (i, i=1,n) /)**, employs an implied-do construct in an array constructor. In general, a list of expressions can precede the indexing in an implied-do construct; a simple example: 100 million alternating ones and zeros, (/1,0,1,0,1,0,1,... /) can be constructed with the array constructor **(/ (1,0, j=1,50000000) /)**. The implied-do simply replicates the list the specified number of times, and if the index-variable is an operand in an expression in the expression-list, each replication of that item uses the corresponding value of the index-variable. The items in the implied-do expres-

sion-list may be any of the three forms allowed in the array constructor itself - scalar expressions, implied-do constructs, and array expressions. The two examples above have only simple scalar expressions in the implied-do lists.

An array expression of any dimension may appear in an array constructor. For example, if A is a 1000*1000 array then **(/ A+1.3 /)** is an array constructor of one million elements, each having a value of 1.3 more than the corresponding element value of A. The elements of **A+1.3** are placed in the array constructor in *array element order*; array element order is obtained by varying the first dimension first, the second dimension next, and so on. Thus **(/ A+1.3 /)** is equivalent to **(/ ((A(j,k)+1.3, j=1,1000), k=1,1000) /)**. Implied-do constructs may be used to specify a different order of the array elements in the constructor. For example, if a row by row vector of the elements of **A+1.3** is desired, rather than the column by column of array element order, **(/ ((A(j,k)+1.3, k=1,1000), j=1,1000) /)** would do the job.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Finally, a simple form of the **reshape** intrinsic function can be used to reshape the (one-dimensional) result of an array constructor into the desired array shape:

reshape (*array-constructor* **,** *shape-vector* **)**

where the shape-vector (which may itself be an array constructor) has one element for each dimension of the desired array shape; the value of each shape-vector element is the number of elements in that dimension in the target array. For example, a 1000*1000 *identity matrix* of can be created as the array constant named **ident** by the declaration

```
real, parameter :: ident(1000,1000) =                                    &
        reshape ( (/ (1.0, (0.0, k=1,1000), j=1,999), 1.0 /), (/ 1000,1000 /) )
```

Thus the array constructor, coupled with the **reshape** intrinsic, is an extremely powerful tool for constructing array values, including array constants.

masked array assignment

A *mask* is an array of type logical. A masked array operation is one in which a mask conformable to the result of the operation is used to specify that only a subset of the concurrent element operations are to be performed. This functionality is available in some of the intrinsic functions, those with a mask argument, and for array assignment. An array assignment is placed under mask control in a **where** statement (R738):

where (*mask* **)** *array-assignment-statement*

The **where** mask must be conformable with the array on the left of the assignment, which must be conformable with the expression on the right of the assignment. For mask elements that have the value *true* the corresponding element assignments take place; where the mask is *false* the assignment is not made. A example of masked array assignment is

```
where (C.gt.0)  A = B/C
```

in which the assignment is made only for those elements of **C** that have a positive value.

Arrays **A**, **B**, and **C** must all be conformable and the (array-valued) logical expression **C.gt.0** is therefore a mask conformable with these arrays.

Another simple example of the use of masked array assignment can be found in the picture refinement program near the end of this chapter. In this case the elements of a character array are set to # where all corresponding elements in another (conformable) array are 1:

```
where (picData.eq.1)  picDisplay = "#"
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Any number of array assignments that are conformable with the mask, can be placed under the control of a single mask; in this case the **where** takes a block form (R739):

```
where ( mask )
   array-assignment-1
   array-assignment-2
      ...
end where
```

The forms of **where** described above leave unassigned some elements of the array on the left hand side of the assignment statement. An extension of the block form of **where**, the **elsewhere** option, specifies a value to be given to the left-hand-side array elements where the mask is *false*. This takes the form (R739):

```
where ( mask )
   array-assignment-1
   array-assignment-2
      ...
elsewhere
   array-assignment-n+1
      ...
end where
```

The picture refinement example uses this last form of **where**:

```
where (picData.eq.1)
   picDisplay = "#"
elsewhere
   picDisplay = " "
end where
```

In this case those elements of **picDisplay** for which **picData** has a value other than one are assigned a blank rather than the # character. This is an important form of **where**, because it results in a fully defined array **picDisplay** that can be used in subsequent array operations. Without the **elsewhere** option the array **picDisplay** might end up not being fully defined, in which case it cannot be used in other array expressions.

assumed-shape dummy arguments

Fortran has always allowed array arguments, but before array-valued expressions were possible, array actual arguments were limited to array variable names; now such actual arguments may be array expressions as well. No new mechanism would be required to handle array exprssions as actual arguments, except for the fact an array expression may be an array section and hence not a *contiguous* array object. In such cases either *stride* information (how the actual array is distributed in memory) must be supplied in the call or the actual argument must be "repacked" so that the old argument association mechanisms, which assume "compacted" arrays, will still work; generally it is more efficient to pass the extra information. *Assumed-shape dummy arguments* are used for this purpose and accommodate the passing of complete *array descriptor* information.

An assumed-shape dummy argument (R516) is declared with a colon for each dimension:

```
subroutine calc3(T,U,V)
 real :: T, U(:,:), V(:)          ! U is a two-dimensional assumed-shape array
  ...                             ! V is a one-dimensional assumed-shape array
end subroutine
```

In a call to **calc3**, any two-dimensional array expression (of type real) may be passed to **U** and any one-dimensional array expression may be passed to **V**; conversely, a two-dimensional real array *must* be passed to **U** and a one-dimensional real array *must* be passed to **V**. In effect the colons in the declarations for **U** and **V** instruct **calc3** to accept the descriptor information supplied by the calling program. **U** and **V** then exactly represent the corresponding array objects in the actual argument list and may be used in array operations in the body of **calc3**.

Assumed-shape dummy arguments require explicit interfaces (see chapter 9). This requirement is automatically met for internal and module procedures; an interface block must be supplied for an external procedure, however. When the procedure's interface is explicit, the calling program knows when an assumed-shape dummy argument is the receiver and can then pass an efficient descriptor; otherwise the calling program cannot assume the dummy arguments are assumed-shape and must therefore provide a contiguous actual argument, packing and unpacking the actual argument array(s) as necessary.

array elements and sections

A portion of an array containing more than one element is called an *array section*. Often an operation is needed on an array section, not on the entire array. The earlier example of normalizing the pivot row of a matrix is a case in point. In this example exactly one row of the matrix was of interest in the computation, not the whole array, and the array section was one row of a two-dimensional array. In general virtually any subset of an array can be an array section. Array sections have array values and may be used in array-valued expressions; they may be assigned array values.

An *array element* is specified by the array name and a subscript value for each dimension:

array-name **(** *scalar-subscript-1* **,** *scalar-subscript-2* **,** *scalar-subscript-3* **,** *...* **)**

where the number of subscripts is the rank of the array and each subscript is a scalar integer expression (*scalar subscript*). An array section is specified by replacing at least one scalar subscript by a *vector subscript* (R618). A vector subscript is a one-dimensional array of scalar subscript values for that dimension; a vector subscript may be constructed with an array constructor. If (only) one scalar subscript is replaced by a vector subscript the resulting array section is a one-dimensional array; if two scalar subscripts are replaced by vector subscripts the result is a two-dimensional array, and so on. An array section has a rank equal to the number of vector subscripts used to specify it.

As an example, consider the following 5x6 array, Q. Three sections of Q are shown in bold: the entire second column (a one-dimensional section), the 2x2 upper right hand corner of Q (a two-dimensional section), and the last half of the fifth row of Q (a one-dimensional section).

$$Q = \begin{bmatrix} 13 & \mathbf{11} & 25 & 2 & \mathbf{1} & \mathbf{9} \\ 9 & \mathbf{3} & 31 & 14 & \mathbf{52} & \mathbf{27} \\ 16 & \mathbf{45} & 54 & 36 & 15 & 20 \\ 7 & \mathbf{20} & 18 & 19 & 8 & 19 \\ 37 & \mathbf{56} & 54 & \mathbf{66} & \mathbf{77} & \mathbf{90} \end{bmatrix}$$

```
Q( :, 2 ) == (/11,3,45,20,56/)                          ! the second column
Q( (/1,2/), (/5,6/) ) == reshape( (/1,52,9,27/), (/2,2/) )   ! upper right corner
Q( 5, (/4,5,6/) ) == (/66,77,90/)                       ! last part of 5th row
```

Note that all of these vector subscripts could be written with implied-do constructs:

```
Q( (/(k, k=1,5)/), 2 )                                  ! the second column
Q( (/(k, k=1,2)/), (/(k, k=5,6)/) )                     ! the upper right corner
Q( 5, (/(k, k=4,6)/) )                                  ! last part of 5th row
```

The implied-do form is more extensible and, for large sections, considerably more compact than explicit lists. Implied-do constructs are also useful for regularly-spaced but non-contiguous vector subscripts. For example,

```
Q( (/(k, k=1,5,2)/), 2) == Q( (/1,3,5/), 2 ) == (/11,45,56/)
```

The implied-do form is common enough that a more readable shorthand notation, called a *triplet subscript* (R619), is also provided for the indexed-do control triplet.

A triplet subscript is just the indexed-do control values, separated by colons rather than commas, with the last one (the increment or stride value) optional. Thus using triplet notation the above four examples may be written (much more clearly!) as:

```
Q( 1:5, 2 )       or   Q( :, 2 )          ! the second column
Q( 1:2, 5:6 )     or   Q( :2, 5: )        ! the upper right corner
Q( 5, 4:6 )       or   Q( 5, 4: )         ! last part of 5th row
Q( 1:5:2, 2 )     or   Q( : :2, 2 )       ! every other element of 2nd col.
```

(The form **Q(:,:)** is a section that comprises the entire array **Q**. This form in a dummy argument declaration, rather than in an array expression, specifies an assumed shape dummy.)

Though the above examples employ array constructors, any one-dimensional integer array expression is permitted as a vector subscript. The only requirement is that the value of each element of the vector subscript be a valid subscript value for that dimension of the array. A common form for vector subscripts is a one-dimensional integer array name (or section), whose element values have been previously established. This form is useful for indirect access, such as indexing into a table; e.g., table elements may be retrieved (or set) by subscripting the table array with an array containing the desired table index values.

For the array **Q** defined above, for example, consider **Q((/2,5,3/), (/6,4/))**

This represents the array section $\begin{bmatrix} Q_{2,6} & Q_{2,4} \\ Q_{5,6} & Q_{5,4} \\ Q_{3,6} & Q_{3,4} \end{bmatrix} = \begin{bmatrix} 27 & 14 \\ 90 & 66 \\ 20 & 36 \end{bmatrix}$

This section can be used in any array expression in which a 3x2 array object is valid. It may also appear on the left hand side of an array assignment, in which case the (1,1) element of the right hand side expression value gets assigned to $Q_{2,6}$, the (3,2) value of the right hand side gets assigned to $Q_{3,4}$, and so on.

A vector subscript may contain more elements than the size of that array dimension. In this case there are duplicate values, since all of the values must be within the array dimension range. Indeed, subscript values may be duplicated in a vector subscript even if the size of the vector is less than the array dimension (the only requirement is that the subscript values must be within range). Both of these cases are illustrated in the following example, which specifies a 7x4 section from the elements of **Q**.

$$\mathbf{Q}((/4,1,2,3,4,2,5/),(/1,4,4,3/)) = \begin{bmatrix} Q_{4,1} & Q_{4,4} & Q_{4,4} & Q_{4,3} \\ Q_{1,1} & Q_{1,4} & Q_{1,4} & Q_{1,3} \\ Q_{2,1} & Q_{2,4} & Q_{2,4} & Q_{2,3} \\ Q_{3,1} & Q_{3,4} & Q_{3,4} & Q_{3,3} \\ Q_{4,1} & Q_{4,4} & Q_{4,4} & Q_{4,3} \\ Q_{2,1} & Q_{2,4} & Q_{2,4} & Q_{2,3} \\ Q_{5,1} & Q_{5,4} & Q_{5,4} & Q_{5,3} \end{bmatrix}$$

Note that rows one and five of this section are identical, as are rows three and six and columns two and three. Many elements of **Q** therefore appear twice in this array section and two elements, $Q_{2,4}$ and $Q_{4,4}$, each appear four times. Array sections with multiple appearances of a given parent array element are perfectly legitimate array operands in array expressions, but such sections must not appear on the left hand side of array assignments (or be actual arguments associated with **intent(out)** dummy arguments - see chapter 9).

dynamic arrays

Fortran 90 has three varieties of dynamic arrays. All three allow array creation at run time with sizes determined by computed (or input) values. The three varieties are: *automatic arrays*, *allocatable arrays*, and *pointer arrays*.

automatic arrays: Automatic arrays are local arrays whose sizes depend upon values associated with dummy arguments. Automatic arrays are automatically created (allocated) upon entry to the procedure and automatically deallocated upon exit from the procedure. The size of an automatic array typically is different in different calls to the procedure. Examples of automatic arrays are:

```
function F18(A,N)
  integer   N                          ! a scalar
  real  A(:,:)                         ! an assumed shape array
  real  F18(size(A,1))                 ! the function result itself is an automatic array

  complex   Local_1(N,2*N+3)           ! Local_1 is an automatic array whose size
                                       !                 is based on N
  real  Local_2(size(A,1),size(A,2))   ! Local_2 is an automatic array
                                       !           exactly the same size as A
  real  Local_3(4*size(A,2))           ! Local_3 is a one-dimensional array 4 times
    ...                                !    the size of the second dimension of A
end function
```

Note the importance of the intrinsic inquiry functions, such as **size** in declaring automatic arrays; a number of inquiry functions are provided that are allowed to appear in declarations. Array bounds and sizes, character lengths, and type kinds may all be specified with expressions involving these inquiry functions. Roughly, a *specification expression*, as such expressions are called, is a scalar integer expression that has operands whose values are determinable upon entry to the procedure. Such operands include constants, references to certain intrinsic procedures, and variables accessible through dummy arguments, modules, common, and (in the case of module and internal procedures) the host procedure.

allocatable arrays: Allocatable arrays are those explicitly declared **allocatable**. An allocatable array may be local to a procedure or may be placed in a module and effectively be global to all procedures of the application. An allocatable array is explicitly allocated with the **allocate** statement, and deallocated either explicitly with the **deallocate** statement or, if it is a local array for which **save** has not been specified, automatically upon exit from the procedure. (If **save** has been specified, local allocatable arrays can persist from one execution of the procedure to the next - they must be explicitly deallocated with a **deallocate** statement.) A global allocatable array persists until it is explicitly deallocated, which may occur in a procedure different from the one in which it was allocated. An allocatable (or pointer) array is indicated if its size depends on a value to be computed after its declaration. The *allocation status* (allocated or not yet allocated) of an allocatable array may be tested with the **allocated** intrinsic function. Examples of allocatable arrays are:

```
subroutine Peach
 use Recipe                          ! accesses global allocatable array, Jam
 real, allocable :: Pie(:,:)         ! Pie is a 2-dimensional allocatable array
   ...
 allocate ( Pie(N,2*N) )             ! allocate a local allocatable array
 if (.not.allocated(Jam))  allocate ( Jam(4*M) )      ! allocate a global allocable array
   ...                                                !    if it is not already allocated
 deallocate ( Pie )
   ...
end subroutine Peach

module Recipe                        ! Jam is a global allocatable array,
 real, allocable :: Jam(:)           !       and can be allocated and deallocated in
   ...                               !         any procedure(s) using this module
end module Recipe
```

Note that the declared bounds for allocatable arrays are simply colons, indicating that these will be provided later, at the time of allocation. This makes allocatable array declaration appear similar to assumed-shape dummy argument declaration, appropriate because the "deferred" nature of the sizes of the dimensions is conceptually similar.

pointer arrays: Pointer arrays are similar to allocatable arrays in that they are explicitly allocated with the **allocate** statement to have computed sizes and are explicitly deallocated with the **deallocate** statement. Simple examples of pointer arrays result by replacing **allocatable** with **pointer** in the preceding examples of allocatable arrays.

In addition, pointer arrays can be used as *aliases* for (may *point to*) other arrays and array sections; the pointer assignment statement is used to establish such aliases. The target for pointer associations (as such aliasing is called) may be other explicitly allocated arrays, or static or automatic arrays that have been explicitly identified as allowable targets for pointers. The association status of a pointer array may be tested with the **associated** intrinsic function. Pointer arrays may be dummy arguments and structure components, neither of which are allowed for allocatable arrays.

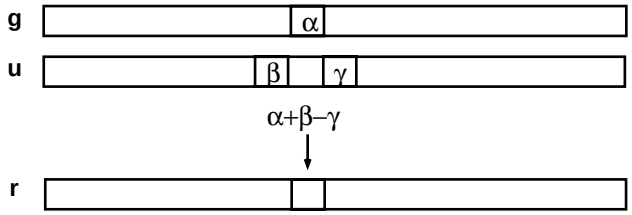
Given this apparent similarity between allocatable arrays and pointer arrays, what is the fundamental distinction between these two forms of dynamic arrays, and when should allocatable arrays be used rather than pointer arrays? Pointer arrays subsume all of the functionality of allocatable arrays, and in this sense allocatable arrays are never needed - pointer arrays could always suffice. The problem with pointer arrays is efficiency. Though pointer arrays must always point to explicit *targets*, which makes some optimization practical that would otherwise be infeasible, pointer assignment makes optimization of pointer arrays much more difficult than for allocatable arrays. Because of their more limited nature and functionality, allocatable arrays are just "simpler" and can be expected to be more efficient than pointer arrays.

Therefore, when all that is needed is simple dynamic allocation and deallocation of arrays, and automatic arrays are not sufficient, use allocatable arrays. A common example of this is if a "work array" is needed of a size dependent upon the results of a local computation. If, on the other hand, the algorithm calls for a dynamic alias, of for example a "moving" section of a host array, then a pointer array is probably indicated.

array-valued functions

As noted above, functions can be defined that return array-valued results. In addition, most intrinsic functions can return array values (and some always do). Array-valued functions may be used as operands in array-valued expressions, allowing more forms of concurrent computation expression.

An example using an array-valued intrinsic function is common in applications of finite difference modeling. Here each element of a large two-dimensional array **g** is to be added to the "next" element in the same row of a conformable array **u**, and subtracted from this is the "previous" row element of **u**; this is to be done with all of the elements of **g**, resulting in a conformable array **r**. The computation, for each element, is sketched as follows:



Using the **cshift** intrinsic function, which returns a given array "shifted" a specified amount, this computation is nicely expressed as follows:

```
r = g + cshift(u,1, 2) - cshift(u,-1, 2)
```

This illustrates the power of array-valued functions, especially if **g** were replaced with an array-valued function reference rather than an array variable reference.

The array-valued intrinsic functions are all summarized at the end of chapter 9 and described in detail in chapter 10. Function **F18** in the previous section and the functional form of **g** in the preceding example are examples of user-defined array-valued functions. The shape of the result returned by user-defined array-valued functions normally is determined (dynamically) from arguments, as illustrated in the **F18** example (and see the **Refine** and **Solve** examples below). Note that function results are declared to be array-valued with ordinary declaration statements, as if the function name is an ordinary variable (as indeed it is within the body of the function). Though automatic arrays may be the most useful form for user-defined array-valued functions, any other form is also valid: explicit-shape array, allocatable array, pointer array.

The main additional requirement for user-defined array-valued functions is that the array value must be fully defined before completeion of execution of the function. On the caller side, the interface of an array-valued function must be explicit so that the caller knows that it is dealing with a function that is array-valued; otherwise the caller has to assume the function returns a scalar value, which is then broadcast in the array-valued expression from which the function is called.

example - picture refinement

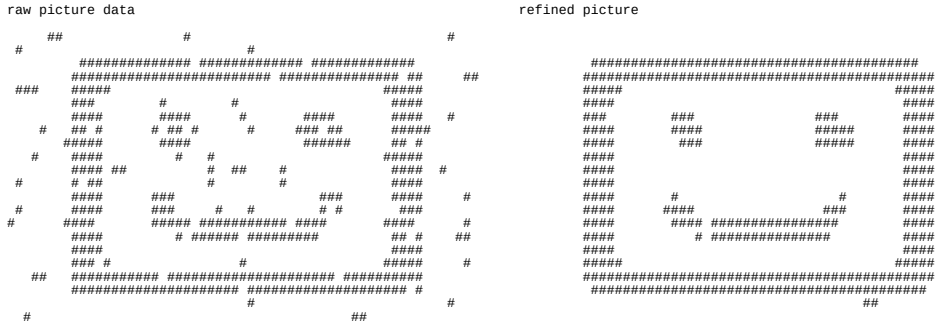
This simple example illustrates a number of the Fortran 90 array features. A two dimensional array of ones and zeros is received, perhaps from space. This data represents the bits of a black and white "picture", but some of the bits have been "corrupted" in transmission. Program **Refine** applies a simple algorithm to this picture to "correct" the corrupted data. This simple algorithm, which is similar to finite-difference algorithms, replaces each element of the array with the average (in this case truncated to an integer) of the values of the 3x3 neighborhood of which that element is the center.

```
program Refine
  integer :: picData(60,24)                    ! the two-dimensional picture array
  integer :: picFile=11
  open (picFile,file="refine.data")            ! open and read the picture data file,
  read(picFile,"(60I1)") picData               ! then display the raw and refined data

  print "(A/(60A1))", "raw picture data", display(picData)
  print "(/A/(60A1))", "refined picture", display(refined(picData) )

contains
!-------------------------------------------------------
function display(pic); integer :: pic(:,:)     ! assumed shape argument
  character :: display(size(pic,1),size(pic,2))  ! returned value same size

  where (pic==1);  display = "#"               ! turn all ones and zeros
  elsewhere;       display = " "               ! into display characters
  end where
end function
!-------------------------------------------------------
function refined(pic); integer :: pic(:,:)     ! assumed-shape argument
  integer :: refined(size(pic,1),size(pic,2))  ! returned value same size
  integer :: r, c
  r = size(pic,1); c = size(pic,2)             ! use of array intrinsics
  refined = 0
  refined(2:r-1,2:c-1) = (  pic(1:r-2,1:c-2) +  &! entire array refined
                            pic(1:r-2,2:c-1) +  &! in one array operation
                            pic(1:r-2,3:c  ) +  &
                            pic(2:r-1,1:c-2) +  &! the sum of the 3x3
                            pic(2:r-1,2:c-1) +  &! neighborhood is divided
                            pic(2:r-1,3:c  ) +  &! by 5 to give the refined
                            pic(3:r  ,1:c-2) +  &! one or zero for each
                            pic(3:r  ,2:c-1) +  &! element in the interior
                            pic(3:r  ,3:c  ))/5   ! of the array
end function
!-------------------------------------------------------
end program
```



Program **Refine** defines two array-valued functions, both with assumed-shape dummy arguments whose size determines the returned array size. Function **display** illustrates the use of the **where** construct. Function **refined** illustrates (a) the use of an array intrinsic (**size**) in the execution part as well as in the specification part, (b) assigning a value to an array section, and (c) a nontrivial array-valued expression which exhibits considerable conceptual concurrent computations.

example - Gaussian elimination

The classic Gauss elimination algorithm, with maximum pivot strategy, for solving systems of linear equations illustrates additional array features. Function **Gauss** is an array-valued function that returns the solution vector for the supplied matrix; it employs several automatic arrays; it has just one loop (to sequence through the pivot elements). The typical nonarray sequential version of this algorithm has loops nested up to four deep; in function **Gauss**, on the other hand, searching for the next pivot element, normalizing the pivot row, and using the pivot row in the next elimination step are all done as array operations.

```
program Solve
  print *,  Gauss( reshape( (/1.,1.,3.,2.,2.,2.,3.,1.,1.,5.,1.,3./), shape=(/3,4/) ) )
  print *,  Gauss( reshape( (/2.,1.,3.,4.,8.,-1./), shape=(/2,3/) ) )
contains
function Gauss(Grid);  real :: Grid(:,:)
  real  ::   Gauss(size(Grid,1)    )                ! returns the solution vector
  real  ::   G(size(Grid,1),size(Grid,2))           ! a local work array (copy of Grid)
  integer  ::   p(size(Grid,1),2)                   ! array for the pivot rows and columns
  logical  ::  not_pivot_row(size(Grid,1),size(Grid,2))          ! mask current pivot row
  logical  ::  not_pivot_rows_or_cols(size(Grid,1),size(Grid,1))  ! mask out all pivots
  integer :: n, pn

  if (size(Grid,2).ne.size(Grid,1)+1) stop 'bad Grid shape'      ! check for valid shape
  n = size(Grid,1)
  G = Grid                                                       ! work on G, not Grid
```

```
    do pn=1,n                                                    ! pn is next pivot number
      not_pivot_rows_or_cols = .true.
      not_pivot_rows_or_cols(p(1:pn-1,1),:) = .false.            ! mask all pivot rows
      not_pivot_rows_or_cols(:,p(1:pn-1,2)) = .false.            ! mask all pivot columns
      P(pn,:) = maxloc(abs(G(:,1:n)),mask=not_pivot_rows_or_cols)      ! find next pivot
      if (abs(G(p(pn,1),p(pn,2))).lt.1E-4)  stop 'ill-conditioned matrix'      ! check stability

      G(p(pn,1),:) = G(p(pn,1),:)/G(p(pn,1),p(pn,2))             ! normalize pivot row

      not_pivot_row = .true.; Not_pivot_row(p(pn,1),:) = .false.  ! mask pivot row
      where ( not_pivot_row )                                    &! reduce matrix
           G = G - G(:,spread(p(pn,2),1,n+1))*G(spread(p(pn,1),1,n),:)
    end do
                                                 ! repeat for all pivots, then
    Gauss(p(:,2)) = G(p(:,1),n+1)                ! unscramble the solution vector

  end function
end program
```

When executed with the simple data sets shown above, this program returns (1.0, -1.0, 2.0) and (7.0, -2.0) for the respective solution vectors, demonstrating that function **Gauss** will correctly solve any size linear system. The entire matrix is reduced for each pivot, rather than just those columns needing reduction, so that about twice as many (scalar element) operations are performed as are really necessary; further tailoring of the **not_pivot_row** mask could decrease the number of (scalar) operations. Note that, in terms of the number of array operations, more attention is devoted in **Gauss** to preparing the masks than to the numerical computations themselves - in array algorithms logical masks take the place of conditional statements in sequential scalar algorithms.

The **Gauss** code is pretty straightforward (though reading and writing such compact, highly concurrent array operations code takes some getting used to); perhaps the least obvious aspect of **Gauss** is its use if the **spread** intrinsic function. **spread** replicates (spreads) a scalar into a one-dimensional array, or replicates an n-dimensional array into an n+1-dimensional array. The scalar-to-one-dimensional array form is used in **Gauss** to convert the scalar operation G(i,j)=G(i,k)*G(k,j), where k is a constant, into a whole-array operation (over all i and j) on G. **spread** has three arguments: the first is the scalar or array value to be spread, the second is the dimension over which the spreading occurs (and must be one for spreading a scalar), and the third is the number of replications.

Function **Gauss** has, for any given system size n, of the order of: 7n array masking operations and 7n array numerical computations, corresponding to about n^4 scalar logical operations and $7n^3$ scalar numerical operations. As the “cost” (execution time) of an array operation, continues its inexorable march toward that of a scalar operation, array codes such as **Gauss** become increasingly attractive in terms of performance.

5 Redundancy

Some of the newer features of Fortran, motivated by modern common programming practices, have made some of the earlier features redundant. The purpose of this chapter is to identify and summarize these redundancies.

The current Fortran standard officially identifies “two categories of outmoded features”: (1) those for which “better methods existed in Fortran 77” and (2) those for which “better methods exist in Fortran 90”. The standard goes on to say “programmers should use these better methods ...”. In this reference these two categories are called “deprecated” (out of favor) features.

In addition to those officially deprecated, there are several features of Fortran 90 that are redundant and for which many believe that “better methods exist”: **common**, **equivalence**, and attribute specification statements.

common blocks

Before Fortran 90, the only practical way to provide for *global* objects (variables and constants) was via *common blocks*; because global objects are important to many Fortran application areas, much practical pre-Fortran-90 code uses common. Common blocks are contiguous blocks of storage, and an object may be associated with (occupy) certain storage units in a common block. Any program unit can access a given common block and thereby access an object by virtue of its known location in the common block. Such objects are said to be *storage associated*, and common blocks share objects among program units through *storage association*.

Common blocks are distinguished by (programmer-specified) names, and the **common** statement (R548) allows the programmer to declare a common block having a given name. The common statement also specifies a sequence of objects that are associated with the successive storage units of that common block; the common block can contain any mix of scalar and array variables (R549), but cannot contain an allocatable array, a dummy argument, a nonsequence structure, or a function result name; arrays dimensioned in common block arrays must have constant bounds. In a given program unit, a object cannot be assigned to two (or more) common blocks.

The common block names are themselves global, known to all program units. Two (or more) program units accessing the same common block access the same sequence of storage units; the object names and mix may be different in the two program units - the association is by storage sequence. For example, suppose that two program units specify the same common block as follows (where all the variables are of type default real):

```
common / Omega / x(100), y, z(200)

common / Omega / a(200), b(100), c
```

The common block name is **Omega** and it contains 301 storage numeric units. In one of the program units the first 100 storage units of **Omega** are known as the array **x**, the 101st

is the scalar **y**, and the last 200 are the array **z**; in the second program the first 200 storage units are known as the array **a**, the next 100 storage units are known as the array **b**, and the last storage unit is know as the scalar variable **c**; **y** and **a(101)**, for example, represent the same storage location and hence are the same "thing".

Objects of type default integer, default real, and default logical have *numeric storage units*; objects of type default complex and double precision real each require two numeric storage units and double precision real requires four numeric storage units. Objects of type default character have *character storage units*. Every other type/kind combination (or type/kind/rank combination for pointers) has a different, unique (but unspecified) storage unit. Objects with different storage units maybe be placed in a given common block, but each program unit in a program using that common block must specify *exactly* the same sequence of storage units. If a sequenced structure appears in a common block, the effect is as if the individual components, in order, had been placed in the common block.

Because of the strict requirement of the preceding paragraph, a popular use of include lines (before the advent of modules in Fortran) was to make one copy of a common block, place it in a file, and include that file in every program unit that used that common block. With modules, the same effect can be achieved by placing the common block in a module and using that module. Even better (and simpler), place the variable definitions directly in the module, without putting them in common blocks; those variables are then global to all programs units using that module (see chapter 8).

Two or more **common** statements naming the same common block can appear in the same program unit; the effect of the second (and subsequent) statements is to extend the common block defined in the first such statement. A named common block must have the same size in all program units. A common block name may appear in a **save** statement, in which case the entire block is saved; individual variables in a common block must not have the save attribute. Variables in a named common block may be initialized in a **block data** program unit.

The common block name may be omitted in a **common** statement, in which case the common block is known as *blank common*. There may be any number of named common blocks, but there is only one blank common; multiple blank common statements in a program unit simply extend the one blank common. The rules for blank common are the same as for named common, except that blank common is always saved, blank common variables cannot be initialized, and different program units can specify blank commons of different sizes (but storage units must still be associated with like storage units).

equivalence

To save space, two or more variables may share the same storage; the **equivalence** statement is how such sharing is specified. This was important when computing systems had quite limited storage, but equivalence is largely redundant these days because there is not normally now the overpowering need to "save space".

Two (or more) variables are *equivalence objects* (share the same space) if they appear in an *equivalence-set* of an **equivalence** statement (R545-547). An equivalence object may be

a variable name (scalar or array), an array element, or a substring; it may not be a named constant, an allocatable array, a dummy argument, a nonsequence structure, a function result name, a pointer (or structure containing a pointer), or any part thereof. There must be at least two equivalence objects in an equivalence set and all equivalence objects in an equivalence set must be of like storage unit. An example of an equivalence statement is:

```
equivalence  ( x, b(10,20) ), ( first, name )
```

In this case the variable **x** is the same as the array element **a(10,20)**, assuming that **x** is a scalar real and **b** is and array of reals, and changing one changes the other; if **first** is a single-character variable and **name** is a longer character string, this equivalence statement causes **first** to be the same character (share the same character storage unit) as the first character of **name**.

Equivalence superimposes (makes the same) two or more storage unit sequences; thus unlike storage units cannot be equivalenced. Moreover since arrays and array element may be equivalence objects, and (whole) arrays occupy contiguous storage units, care must be taken to not specify inconsistent pairings. For example, if x and y are both one-dimensional real arrays

```
equivalence  ( x(10), y(20) )
```

assigns **x** and **y** overlapping storage units, offset by 10, but

```
equivalence  ( x(10), y(20) )  ( x(20), y(10) )
```

specifies inconsistent offsetting and hence is illegal. As illustrated, any array elements or substrings specified as equivalence objects must use constants as the subscripts or substring ranges. An unsubscripted array name (or a character variable name) as an equivalence object, storage associates the first element of the array (or first character of the string) to the other equivalence objects in that equivalence set.

The objects in an equivalence set must be local to that program unit. An equivalence object may be a variable in a common block, but equivalence must not cause two different common blocks to become storage associated, nor add storage units that precede the front of a common block.

attribute statements

Prior to Fortran 90 Fortran did not have the attribute form of the type declaration statement (see chapter 2); separate statements, now called *attribute statements*, were used to convey attributes to objects; each such statement conveys exactly one such attribute. Though the attribute form of the type declaration makes the attribute statements essentially redundant, attribute statements are the only way attributes can be given to implicitly typed objects. Constraints pertaining to attributes are summarized in chapter 2.

The typical (but not only) form for attribute statements is

attribute-name [**::**] *object-list*

The effect is that the named attribute is given to each of the object listed in the object list.

Some examples are:

```
parameter ( MAX=100 )                ! with type declaration: integer, parameter :: MAX=100

real :: x, y                         ! using type declarations:
dimension :: x(100), y(200,200)      !       real :: x(100)
save :: y                            !       real, save :: y(200,200)
```

In the first of these, which is not of the general form given above, **MAX** may be implicitly typed; in the second, the (two) variables are explicitly typed, but the other attributes are conveyed with separate attribute statements. The attribute statements having precisely the form shown above are: the **optional** statement (R520), the **dimension** statement (R525), the **allocatable** statement (R526), the **pointer** statement (R527), the **target** statement (R528), the **external** statement (R1207), and the **intrinsic** statement (R1208).

The attribute statements having almost the form shown above are: the **intent** statement (R519), the **public** and **private** statements (R521), and the **save** statement (R523). In the **intent** statement, an *intent-spec* (R511) must follow the **intent** keyword. In the **public**, **private**, and **save** statements, the object-list is optional; if it is missing then the attribute applies to all of the local objects with which it is compatible. Note that the **save** attribute statement is the only mechanism for saving a named common block.

The form of the **parameter** statement is given in R538-539; such a statement can contain any number of named constant value definitions, separated by commas and enclosed in parentheses. Another example of a parameter statement is:

```
parameter ( MAX=100, DOUBLE=kind(1D0) ) )        ! assuming DOUBLE is of type integer
```

The **data** statement (R529-537), which initializes variables, not constants, is the attribute statement that differs the most from the general form; it also is the one that is almost not redundant, as it can be used to initialize part of an array, a structure, or a substring (the initialization provision of the type declaration, when applied to an array, structured object, or a character string, must initialize the entire array, structure, or string).

The simplest form of the **data** statement is:

data *variable-list* **/** *value-list* **/**

The variables in the variable list are “paired”, left to right, with the values in the value list; each value has to be assignment-compatible with its associated variable. Any substring or array section ranges, or array element subscripts, in the variable list must be constants, and all values in the value list must be constants and any repeat factors must be positive integer constants. An example of a data statement is:

```
data  count, n, name(1:3), (x(i), i=4,19,3)  /  0, 0, "Dru", 6*3.5  /
```

After the **data** keyword comes the list of variables to be initialized; between the slashes (*/*) are the initial values. In this case the variables are scalar integers **count** and **n**, both initialized to zero, a substring (first three characters) of character variable **name,** initialized to "Dru", and six elements (4,7,10,13,16,19) of real array **x** all initialized to the value 3.5. The 6 in 6*3.5 is a *repeat factor* and 6*3.5 is equivalent to 3.5,3.5,3.5,3.5,3.5,3.5.

Recall that an object with the **data** attribute (i.e., has been initialized) also automatically has the **save** attribute. The following objects must not appear in a data statement: a named constant, an object in common (unless the data statement is in a block data program unit) an allocatable array, a dummy argument, a function result name, a pointer (or structure containing a pointer), or any variable imported by host or use association.

block data program unit

The sole purpose of the **block data** program unit (R1110) is to initialize objects in named common blocks. A block data program unit has only a (limited) specification part and no execution part or internal procedure part. It has **common** statements to specify the common blocks it initializes, and any declaration and specification statements needed to fully specify the attributes of the common block variables to be initialized. Thus the only statements that block data program units can have are: derived-type definitions, type declarations, and **use**, **common**, **equivalence**, **dimension**, **pointer**, **target**, **intrinsic**, **save**, **parameter**, and **data** statements; common statements must specify named common blocks. The role of use statements in block data program units is quite limited; only named constants can be imported in this manner.

Common block variables cannot be initialized in type declaration statements, and so data statements must be used for this purpose; therefore, unlike in other program units, in block data program units common block variables may appear in the variable lists of data statements.

A block data program unit can initialize more than one named common block, and only a part of a common block need be initialized - it is not necessary to initialize the entire common block. Though a common block may be only partially initialized, it must be entirely specified in that block data program unit. A program may contain any number of block data program units (at most one of which can be unnamed), but a given common block may be initialized in at most one of the block data program units.

deprecated features

The Fortran standard says that deprecated features may be removed from subsequent versions of the standard. Even should this happen, standard-conforming implementations are still allowed to support these features (as “extensions” to the language); many will do so.

The five deprecated features for which the standard proclaims there are “better methods in Fortran 77” are:

1.1 **real** (and double precision) **do** control variables (R822)
This tends to be error-prone because of accumulated round-off error associated with repeated arithmetical operations. The better method is to use an integer control variable and to convert it to the requisite real value prior to using it in the computations of the loop.

1.2 branching to an **end if** statement (from outside that **if** construct)
Better method - branch to the statement following the **end if**.

1.3 the **pause** statement (R844)

Execution of this statement requires subsequent "operator intervention" to resume execution. Operator intervention is an archaic notion in most modern computing; a better method is to use a **read** statement (R909) without an input list (e.g., **read ***) to pause execution; the user can resume execution by pressing the "return" key.

1.4 **assign** and **assigned goto** statements (R838-839) and assigned **format** specifiers

These statements involve using statement labels as integer values for controlling selective execution; a major use was to simulate internal procedures. Better methods are either internal procedures or equivalent **if** constructs. A better method for assigned format specifiers is to use character formats (R913, first alternative).

1.5 **H** edit descriptor (R1016, second alternative)

These were used to provide character output before the advent of the character type. Better methods are to use character constant edit descriptors (R1016, first alternative) or, better yet, to place character constants in the output list, associated with **A** edit descriptors.

The ten deprecated features for which the standard proclaims there are "better methods in Fortran 90" are:

2.1 **arithmetic if** statement (R840)

Redundant ever since the introduction of the logical **if** statement. Better methods are the logical **if** statement (R807) and the **if** construct (R802).

2.2 **shared do** termination (R826)

This allowed nested do loops to share the same terminal statement, which is now considered to be poor software engineering practice. A better method is to have a separate **end do** statement for every **do** statement (R817).

2.3 alternate **return** (R1214, R1221, second alternative)

This feature introduces labels into argument lists; upon return from the procedure a branch may be made to such a specified label. A better method is to return an integer or character code, which can then be used as the controlling case expression in a case construct (R809) to achieve the desired processing.

2.4 **computed goto** statement (R837)

This is another instance in which the case construct (R809) is a better method.

2.5 statement functions (R1226)

Statement functions look like assignment statements and have a number of error-prone non-intuitive restrictions. Statement functions are completely superseded by internal procedures.

2.6 **data** statements in the execution part

Data statement initialization is done at compile time, not execution time, so this capability is at best misleading. A better method is to place all data statements before the execution part; within the execution part if the value of a variable is to be changed that must be done with an assignment statement - it can't be done with a data statement.

2.7 fixed source form

Fixed source form requires close attention to columns 6, 7 and 72 on a line and does not use blanks as delimiters. This is error-prone for several reasons, but especially on modern screen equipment using proportional fonts. Free-form source is a better method.

In fixed-form source, columns 1-6 are reserved for comment initiators, labels, and statement continuation; a **C** or **#** in column 1, or a **!** in any column (except column 6), makes that line into a comment line; (optional) statement labels must be put in coulmns 1-5, and any nonblank nonzero character in column 6 makes (columns 7-72 of) that line a continuation of the preceding (noncomment) line. Statement text must go in columns 7-72.

2.8 assumed-size arrays (R518)

Assumed-size arrays are "open ended" and not consistent with the conformability requirements of the Fortran 90 array operations and assumed-shape arguments. Better methods are to use automatic arrays, assumed-shape arrays, and deferred shape arrays, as appropriate in specific contexts.

2.9 **character(*)** function results

A character function may be defined with an asterisk (*****) length. There is no way the function can determine the value of the length for a given invocation of the function, however, say from argument values; rather the calling program must declare this function with a specified length. This is not very useful functionality and is inconsistent with other function result concepts (e.g., deferred shapes for array-valued functions). In most cases a better method for the (likely) intended functionality is to use a subroutine with an extra character argument that can be used to return the desired character value to the calling program.

2.10 **character*** type specifier (R507, second alternative)

This original form of character length declaration, introduced in Fortran 77, is clearly redundant with the comprehensive and consistent type parameter declaration model in Fortran 90. A better method is to use the **character(*)** form (R507, first alternative).

6 Input/Output

Just as Fortran has an entire “sublanguage” for array processing (see chapter 4), so too does it have a comprehensive sublanguage for performing *data input* and *data output*.

The **read** statement performs data input. The sources for data input are the user’s keyboard and/or one or more data files on the computing system; the input process transfers a copy of the data from the external source(s) into specified variables of the program, replacing the previous values of those variables. The **write** statement performs data output. The destinations for data output are the user’s screen and/or one or more data files; the output process transfers a copy of the values of specified variables and expressions to the external destinations, either appending the data to previously written data or replacing existing data on the external destination(s), depending on the nature of that particular output operation.

Fortran I/O is *record oriented*. A data file is a sequence of records, each record being a sequence of values terminated by a special end-of-record (EOR) character (or character combination). EOR is system-dependent but often is equivalent to end-of-line - when displayed on a screen, each line represents a record. A read statement normally “consumes” an entire record (line), regardless of how much data is then actually used; a write statement normally produces an entire record (including the EOR). When reading from the keyboard each typed line, ending with the *return* key, is an input record; when writing to the screen each write operation produces one line of output. Fortran 90 introduced *nonadvancing I/O*, providing Fortran, for the first time, with partial-record I/O capability.

The basic read and write statements are quite simple. The bulk of the I/O sublanguage involves the many data formats that the input/output processes must accommodate, as well as tools for effectively utilizing the data file system. The first two sections of this chapter illustrate basic reading and writing of data; though relatively simple, these illustrations include a great many practical uses of the read and write statements. The remaining sections are devoted to the more specialized, or more subtle, aspects of formats and files.

inputting data (read)

A simplified general form of the **read** statement (R909) is:

read ([**unit=**] *unit* [**,** [**fmt=**] *format*] [**,** [**iostat=** *ios-variable*] **)** [*input-list*]

The *input-list* specifies the variables into which the data is to be read; the items in parentheses specify the data source (*unit*), the data *format*, and a *status variable* (to detect input errors, end of file, etc.) - these are called the *input control specifiers* (R912). Note that the only control specifier required is the unit and that the only specifier keyword required, when that specifier is used, is **iostat=**. If the input list is omitted, no values are input, but a record is consumed nonetheless.

Actual uses of the read statement tend to appear quite a bit simpler than the above general form:

```
read (dataFile, fmt=* )  x, y, z          ! read 3 values from a data file, with "free form" input
```

```
read (*, fmt="(I4,A)", iostat=k2)  number, name     ! read two values from the keyboard
read (expenses, fmt=*)  balance, amounts            ! read 2 values from a file; "free form" input
read (*,*, iostat=ios)  next                        ! read one value from the keyboard
read (labData, "(A,I4,5F10.3)")  specimen, n, weight(1:5)          ! formatted read from a file
```

An asterisk (*****) for the unit specifies keyboard input rather from a file. An asterisk for the format specifies default formatting (also called *list-directed* formatting); list-directed input formatting assumes the individual values requested (by the *input-list*) are separated by (any combination of) spaces, commas, and end-of-lines. The iostat option is used if and only if the programmer wants to detect input errors or the end of the file.

If the input unit is to be a data file rather than the keyboard, the unit is an integer value (this is a good place to use a named constant); this value must have been connected to a specific file, with the **open** statement, prior to executing the read statement. For example:

```
open(inputData, file="lab/test-16.data")          ! inputData is a previously defined integer
```

After execution of this open statement the appearance of **inputData** as the unit in a read statement will cause the input to be taken from the next record in the file identified as “lab/test-16.data”, which on most systems is the file named “test-16.data” in the directory named “lab”. The simplest form of the open statement is

open ([**unit=**] *unit* **,** **file=** *file-name* **)**

where *unit* is as defined above for the read statement and *file-name* is any character expression; of course if the file specified by *file-name* does not exist, an I/O error occurs. See the section below on opening files for other features of the open statement and how to prevent, detect, and recover from I/O errors encountered while opening files.

The format specifier in the read statement may be: omitted (in which case this is a *unformatted* read), an asterisk (listed-directed formatting, see above and the relevant section below), a character expression (whose value must be a valid format specification, R1002), or a label (which must be the label of a **format** statement, R1001). (See chapter 5 for a deprecated option not listed here.) The format specifies how the input data is converted and assigned as the values of the variables in the input list. An unformatted read must specify a unit that is connected to a file previously created with unformatted write statements.

If an iostat variable is specified in a read statement, it must be an integer variable, and after execution of the read statement it is defined as follows: with a negative value if end-of-file is detected (in which case no data input occurs, and the variables in the input list are undefined), with a positive integer value if an I/O error occurs (also in which case the variables are undefined), or zero (in which case no error or end-of-file occurred and the variables are defined with the input values). The non-zero values for the iostat variable are implementation-dependent, but in principle can be used to determine the exact nature of the error.

Alternatives to the **iostat=** specifier are the **end=** specifier and the **err=** specifier (R912). **end=** applies only to input and specifies the label to which the program branches if the end-of-file is encountered during execution of the read statement in which the end=

appears. **err=** can be used with both input and output and specifies a label to which the program branches if any I/O error occurs during execution of the read or write statement in which the err= appears. In addition, the err= option is available in the other I/O contexts in which iostat= can appear: the open and inquire statements. end=/err= can be used together with iostat=, in the same statement, or they can be used separately.

The input list (R914) can contain any variables, in any order, that can appear on the left-hand side of an assignment statement, including scalar variable names, array variable names, array elements, array sections, substrings, and structure components; in addition, the input list can include io-implied-do constructs (R916).

outputting data (write)

A simplified general form of the **write** statement (R910) is similar to the read statement:

write ([**unit=**] *unit* [**,** [**fmt=**] *format*] [**,** [**iostat=** *ios-variable*] **)** [*output-list*]

The *output-list* specifies the values to be copied to the output destination; the only difference between the input list and the output list is that, whereas the input list must specify assignment-capable variables, the output list can comprise any set of expressions (including stand-alone variables and expressions formed in io-implied do constructs). If the output list is omitted, an empty record is written; on the screen this appears as a blank line.

The unit, format, and iostat specifiers in the write statement are the same as in the read statement. The unit is an integer value that identifies the file that is to receive the output, or it is an asterisk; if the latter, the output is displayed on the user's screen. As for input, if the output is to a file, the unit (integer value) must have been connected, by an open statement, to the desired file before execution of the write statement. The simplest form of opening a file for output appears exactly like that for input.

Also as in the read statement, the format specifier in the write statement may be: omitted (in which case this is a unformatted write, and any subsequent reads on this file must be unformatted), an asterisk (listed-directed, system-defined default output formats are used), a character expression, or a label. The format specifies how the values of the output list are to appear in the destination record. Unformatted output should not be sent to the user's screen.

The only role of the iostat variable in a write statement is to detect, and take corresponding action, if an error occurs during the execution of the write statement. As in the read statement, if an error occurs the iostat variable is defined with an implementation-determined positive value; if no error occurs the iostat value is zero.

Examples of the write statement are:

```
write (dataFile, fmt=* )  x, y, z              ! write 3 values to a data file, with default formatting
write (*, *, iostat=ios) number, name                    ! write two values to the screen
write (*, fmt="(T20,I5)", iostat=ios)  next               ! write one value to the screen
write (labData, "(A,I4,5F10.3)") specimen, n, weight(1:5)     ! formatted write to a file
```

A redundant form of **write(*,** *format* **) ...** is provided as the **print** statement (R911):

print *format* [**,** *output-list*]

Similarly, a redundant form of **read(*,** *format* **) ...** is provided (R909, second alternative):

read *format* [**,** *input-list*]

where *format*, *input-list*, and *output-list* are exactly as described above.

data formats

Data read from the keyboard or written to the screen is always *formatted*; data read from or written to a file may be either formatted or *unformatted*. Unformatted I/O is specified, as outlined above, by omitting the *format* specifier from the *io-control-list*. The purpose of unformatted I/O is to provide efficient data transfer, without conversion between the file and the internal representation in the programs variables; in an unformatted write the bit patterns of the data as represented in the program variables is written, unchanged to the file. Such data is subsequently readable only by the corresponding unformatted read statement ("corresponding" meaning the same type/kind pattern of variables in the input list as values in the written by the output list), and again the bit patterns are simply transferred - in this case from the file to the variables - without conversion. To be readable by humans, data must be converted from internal form to appropriate character strings, and vice versa - that is, formatted.

Default numeric types, for example, are typically groups of 32 binary bits; to be readable by humans this the value of an real variable must be converted to the familiar decimal digit representation of numerical values (complete with decimal points, minus signs, etc.). Similarly, when one types a **-14** as keyboard input, this must be converted to the internal (usually binary) representation used by the variable receiving this value. Formatted I/O first specifies that such conversion is to take place and second allows the programmer to specify the exact form of the output (number of decimal places, for example) and, for input, the exact form in which the data exists (and from which conversion must be made).

Such conversion is specified by format *data edit descriptors* (R1005). A *format-specification* (R1002) is a sequence of such edit descriptors, delimited by commas, possibly interspersed with *control edit descriptors* (R1010), and enclosed in parentheses. In a formatted I/O statement each value in the input-list or output-list is "paired with" a data edit descriptor that specifies how that value appears in the source (input) or how that value is to appear in the destination (output). The association is positional, with each value in the input/output list associated with the data edit descriptor in the same position (ignoring any control edit descriptor) in the format specification - thus, in left-to-right fashion, the first value in the input/output list is associated with the first data edit descriptor in the format specification, the second with the second, etc. The descriptor list may be longer (have more data edit descriptors) than the input/output list (has values), in which case the extra descriptors are unused; if the descriptor list is shorter, then it is "reused" as often as needed. Some examples appear above - others are:

```
(A8, E16.6, I10)                ! a character, real, and integer, in 34 spaces
```

```
(I5, I5, 2F10.2, A42)          ! five values in record, in 72 spaces
(I5, 3(4F5.1, Z5), E20.4)      ! 14 values in record, in 100 spaces
(A10, 2L5, A20, A30, F10.4)    !      ( note that these five examples are
(A, A, G10.2E4, EN8.4)         !        not complete Fortran statements )
```

Each intrinsic data type has a set of data edit descriptors. The I, B, O, and Z data edit descriptors are for integer values. The F, D, G E, EN, and ES data edit descriptors are for real values (and complex values - each complex values takes two real data edit descriptors - one for the real part and one for the imaginary part). The L edit descriptor is for logical values, and the A edit descriptor is for character values. A derived type (structure) value requires a set of data edit descriptors corresponding to its components, an appropriate one for each of its (intrinsic) components, similar to complex (thinking of a complex value as a structure with two real components). The following table summarizes these 12 data edit descriptors:

data edit descriptor	data type	effect	more examples
Iw [.m]	integer	optional ± followed by decimal digits; on output write a minimum of m digits and right-justify value in field width w; on input value must be an integer constant (not necessarily right-justified in the field); m has no effect on input; m≤w; default value of m is 1	I8 I4 I9.5
Bw [.m]	integer	same as I format, except binary digits (0,1) instead of decimal digits and no sign	B16
Ow [.m]	integer	same as B format, except octal digits (0-7) instead of binary digits	O3
Zw [.m]	integer	same as B format, except hexadecimal digits (0-9,A,B,C,D,E,F) instead of binary digits	Z2.2
Fw.d	real, complex	output has optional ± followed by decimal digits with d digits to right of decimal point, right-justified in field width w; input may be integer, decimal digits with decimal point anywhere, or either followed by ± followed by an integer exponent; d+1<w; need two for complex values (and for the E, D, G, EN, and ES data edit descriptors)	F7.2 F12.8 F5.1 F6.0
Ew.d [Ee] Dw.d	real, complex	Ew.d output has optional ± and 0 preceding decimal point, d digits after decimal point, followed by a base-10 exponent of the form E±uu or ±uuu (u being a decimal digit); Ew.dEe is the same but with e u's in the exponent part; Dw.d is the same as Ew.d but with a D instead of E in the exponent; d+6<w; same as Fw.d on input, except input value may have an E or D exponent	E9.2 D9.2 E14.4E4 E30.6
Gw.d [Ee]	real, complex	same as Fw.d on input; a generalized edit descriptor that for output value v acts approximately as Fw.d for 0.1<v<10**d and approximately as Ew.d [Ee] otherwise	G10.3 G10.3E3
ENw.d [Ee]	real, complex	output is in "engineering notation", which is like Ew.d [Ee] but with an exponent divisible by 3 and 1-3 digits preceding the decimal point; same as Fw.d on input	EN15.5
ESw.d [Ee]	real, complex	output is in "scientific notation", which is Ew.d [Ee] with the exponent one smaller so that there is a single nonzero digit preceding the decimal point; same as Fw.d on input	ES15.5
Lw	logical	output consists of w-1 blanks followed by a T or an F; on input, in field width w, any number of blanks followed by an optional period followed by a T or F, followed by anything	L2 L14
A [w]	character	if w is omitted the field width is the length, n, of the character value/variable; on input, if w>n then the n rightmost characters in the field are read and if w<n then the w characters are character-assigned to the variable; on output, if w>n then the characters are left-justified in the field and if w<n then the leftmost w character of the value are output	A A10 A40

In these data edit descriptors w, d, e, and m all must be unsigned integer constants (but not named constants); in addition all may be optionally preceded by an unsigned integer constant (but not named constant) *repeat factor*; the letters I, B, O, Z, F, E, D, G EN, ES, L and A must all be capital (cannot be lowercase). All involve a field width, w, which is the total number of characters "reserved" for this value.

The control edit descriptors, which can be inserted among the data edit descriptors as desired, and also comma delimited, are summarized in the following table:

control edit descriptor	effect	examples
/	at this point complete the current record and start a new one; need not be comma delimited	/,/
T n	tab to the character column n of the record (tabbing may be either forward or backward)	T40
TL n	tab left (backward) n character columns in the record	TL2
TR n	tab right (forward) n character columns in the record	TR12
n X	same as TR n	12X
S	processor choice as to whether or not to output the optional plus sign (this is the default)	S
SP	from this point, the optional plus sign must be output; no effect on input	SP
SS	from this point, the optional plus sign must not be output; no effect on input	SS
BN	from this point, nonleading blanks in numeric input fields treated as nulls; no effect on output	BN
BZ	from this point, nonleading blanks in numeric input fields treated as zeros; no effect on output	BZ
k P	"scales" subsequent numerical values; on input, no effect if the input field has an exponent, and otherwise divides the input value by 10**k during conversion; on output, no effect for the F, (F part of) G, EN, and ES edit descriptors, and for the E and D (and E part of G) descriptors reduces the exponent value by k and multiplies the nonexponent part by 10**k	3P 8P
ch-ed	a character constant (but not a named constant) that is written to output; no effect on input	"x= "
:	stops output format processing if the output list has been finished (suppresses subsequent ch-ed)	:

In these control edit descriptors k and n must be unsigned integer constants (but not named constants); the letters P, T, L, R, X, S, N, and Z must all be capital (cannot be lowercase). Some examples of format-specifications containing both data edit descriptors and control edit descriptors are:

```
(T5, I5, I5, 2F10.2, A42)              ! tab to column 5 first
(I5, /, 3(4F5.1, Z5, /), E20.4)        ! a total of five records involved
(A10, 2L5, TR10, A20, A30, 2PF10.4)    ! tab right after logicals, and scale the real
(A, BZ, G10.2E4, EN8.4)                ! treat blanks as zeros in the two numeric fields
```

The discussion, tables, and examples in this section summarize most of the important concepts and techniques of I/O formatting. But there are many other combinations and subtlies: treatment of formatting takes an entire chapter and 21 large pages in the Fortran standard, and 53 (smaller) pages in the exhaustive Fortran 90 Handbook. Consult the references listed in the preface for additional details regarding formatted I/O.

opening and closing files

A file read or write statement specifies the file via a file *unit* number, which is an integer value; prior to executing the read or write statement, this unit must be associated with an actual file on the computing system. Making this association is called *connecting* the file to the unit or *opening* the file; breaking this connection is called *closing* the file. During program execution a file may be opened on a unit, subsequently closed, and then reopened again or a different file opened on that unit. The open and close statements control file connection.

The simplest form of the open statement was illustrated above, with the two required "connect specifications" (unit and file). There are many more connection properties that can be specified when making a file connection, however, and the general form of the **open** statement (R904-905) has 11 additional connect specifications, all optional; all have required keywords (e.g., **access=**). Only the specifier in the open statement that has an optional keyword is the unit specifier, and **unit=** may be omitted only when the unit specifier is the first in any open statement specifier list - the specifiers may be in any order an none may appear more than once. Open statement specifiers are summarized in the following table, with the unit and file specifiers first followed by the others in alphabetical order.

specifier	value	meaning
unit=	integer expr.	the unit number to be connected by this open statement
file=	character expr.	the name of the file to be connected to this unit
access=	"direct"	file to be connected for direct, or "random", access to its records
	"sequential"	file to be connected for sequential access to its records; the default
action=	"read"	file to be connected for reading (input) only
	"write"	file to be connected for writing (output) only
	"readwrite"	file to be connected for both reading and writing
blank=	"null"	ignore blanks in numeric input fields (can be overridden by BZ); the default
	"zero"	treat blanks as zeros in numeric input fields (can be overridden by BN)
delim=	"apostrophe"	this specifies the delimiter to be used in writing character data
	"quote"	by list-directed or name-directed (namelist) output statements;
	"none"	"none" is the default
err=	label	branch point if an error occurs in executing this statement
form=	"formatted"	the records in the file are formatted; the default for sequential access
	"unformatted"	the records in the file are unformatted; the default for direct access
iostat=	integer variable	same as in read/write statements; gets positive value if error occurs
pad=	"yes"	use blank padding for character input, when needed; the default

specifier	value	meaning
	"no"	don't pad - require that input data has the requested number of characters
position=	"asis"	do not change file position upon connection; the default
	"rewind"	upon connection, insure that file is positioned at its first record
	"append"	upon connection, insure that file is positioned after its last record
recl=	integer expr.	record length for direct files; number of characters for formatted files; processor-dependent units for unformatted files, typically bytes or words
status=	"old"	the file must exist prior to making the connection
	"new"	the file must not exist prior to the connection - created by connection
	"unknown"	processor-dependent status; this is the default
	"replace"	creates or replaces file; in either case, the exist= inquiry returns .true.
	"scratch"	a temporary file is created, for the duration of the connection; this is the one case in which the file= specification is not (must not be) used

For those open specifiers for which character values are listed in the above table (e.g., **action= "read"**), the value can be specified as a character expression, but such expressions must evaluate to one of the listed options, either in uppercase or lowercase (all lowercase shown above). For those specifiers having a specific default value, the default is identified; in the other cases the default is processor dependent. Note the one case that is incompatible with the **file=** specifier: when **status="scratch"** is specified.

Some of the options in the above table involve concepts to be discussed in subsequent sections of this chapter; for example, **access= "direct"** specifies a "random" file rather than a sequential file, and random files are described in section below entitled "sequential and random files".

The close statement (R907-908) disconnects the file currently connected to the specified unit, allowing the unit to be reconnected later to another (or even the same) file. Any connections not explicitly terminated by close statements are automatically close at the termination of the program. As with the open statement, the close statement has a list of specifiers, only one of which is required (the unit specifier, which is the same as in the open statement). The close **err=** and **iostat=** specifiers play the same error-handling role in the close statement as they do in the open statement. The only other close specifier is the **status=** option, which has two possible values: "keep" specifies that the file remain on the system after being closed, and "delete" specifies that the file be deleted from the system; "delete" is the default for scratch files and "keep" is the default for all other files.

file inquiry

The Fortran I/O sublanguage has an extensive file inquiry mechanism, which allows information to be obtained about a file before opening it; such information can be used in subsequent connection specifiers. The form of the **inquire** statement (R923-924) is similar to that of the open statement in that in has a statement name and a list of specifiers; each specifier specifies a variable to hold the returned information (except **err=**, which specifies a label). As in the open statement, the only specifier for which the keyword can be omitted is the unit specifier, and then only if this is the first specifier in the list; any given specifier can appear at most once in a given inquire statement.

Each inquire statement must have either a unit specifier or a file= specifier but, unlike the open statement, not both. If it has a unit specifier then the inquiry is “by unit”, and the information returned pertains to the unit and the file connected thereto (if any). If the inquire statement has a file= specifier then the inquiry is “by file” and the information returned pertains to the file on the system with the name specified in the file= specifier. In inquiry by unit the specified unit may or may not be connected; in inquiry by file the file may or may not exist and, if it exists, may or may not be connected to a unit. the various inquiry specifiers are summarized in the following table.

specifier	returned value for *file inquiry*	returned value for *unit inquiry*
unit=	not allowed	the unit number about which to inquire
file=	name of the file about which to inquire	not allowed
number=	the unit number, if currently connected; otherwise the integer value -1	
named=	.true.	.true. iff connected to a named file
name=	file name	file name if connected to a named file[a]
exist=	.true. if file exists, .false. otherwise	.true. if unit exists, .false. otherwise
opened=	.true. if file is currently connected	.true. if unit is currently connected
access=	"sequential" or "direct", if connected; otherwise undefined	
sequential=	"yes", "no", or "unknown", if connected; otherwise "unknown"	
direct=	"yes", "no", or "unknown", if connected; otherwise "unknown"	
action=	"read", "write", or "readwrite", if connected; otherwise undefined	
read=	"yes", "no", or "unknown", if connected; otherwise "unknown"	
write=	"yes", "no", or "unknown", if connected; otherwise "unknown"	
readwrite=	"yes", "no", or "unknown", if connected; otherwise "unknown"	
form=	"formatted" or "unformated", if connected; otherwise undefined	
formatted=	"yes", "no", or "unknown", if connected; otherwise "unknown"	
unformatted=	"yes", "no", or "unknown", if connected; otherwise "unknown"	

specifier	returned value for *file inquiry*	returned value for *unit inquiry*
blank=	"null", "zero", or undefined, if connected; otherwise undefined	
delim=	"apostrophe", "quote", "none", or undefined, if connected; otherwise undefined	
err=	label of statement to which to branch if an error occurs	
iostat=	0 for no error; a positive integer value if an error occurs	
pad=	"yes" or "no", if connected; "yes" if not connected	
position=	"asis", "rewind", "append", or undefined, if connected; otherwise undefined	
recl=	record length, if connected; otherwise undefined[b]	
nextrec=	next record number, if connected for direct access; otherwise undefined	
iolength=	**recl=** value for the *output-item-list* (a special form of the inquiry statement)	

a. The value is undefined if the unit is not connected, or is connected to a scratch file.

b. If the connection is for direct access, all records have the same length; if the connection is for sequential access, the maximum record length is returned.

Three of the inquire specifiers (**unit=**, **file=**, and **err=**) serve as input to the inquire statement; the others all return values to the program. Three of these specifiers (**named=**, **exist=**, and **opened=**) return logical values, five (**number=**, **iostat=**, **recl=**, **nextrec=**, and **iolength=**) return (default) integer values, and the rest return (default) character values. Note that in a great many cases the value return is undefined if the file or unit in not currently connected, which means that normally an **opened=** inquiry should be made first.

sequential and random files

Fortran data files come in two flavors, *sequential* and *direct*. The records of a sequential file are processed in sequence, starting from the first record of the file. The read and write statements illustrated above are sequential reads and writes. (Note that the keyboard is a sequential input “file” and the screen is a sequential output “file”.) Opening a file for sequential access *positions* the file at its first record; a sequential file may be repositioned at its first record by closing the file and reopening it; it may also be repositioned at the first record, without closing and reopening, by executing a **rewind** statement (R921-922) on the unit connected to that file. A sequential file may also be “backed up” one record by issuing a **backspace** (R919) on the unit; this is handy if there is a need to reread (or rewrite) the previous record. The **endfile** statement (R920) causes an end-of-file marker to be written to a sequential file opened for write or readwrite action; closing such a file also writes an end-of-file marker.

A direct file is so called because a one can “go directly” to any record number in the file; direct files are also called ‘random” files, because one can specify processing any record at “random”. If there are **n** records in the file, they are numbered, 1, 2, 3, ..., **n**, and the read or write statement can specify, with the **rec=** specifier, which record is to be involved:

read ([**unit=**] *unit* [**,** [**fmt=**] *format*] **, rec=** *record-number* [**, iostat=** *ios-variable*] **)** *input-list*

write ([**unit=**] *unit* [**,** [**fmt=**] *format*] **, rec=** *record-number* [**, iostat=** *ios-variable*] **)** *output-list*

Other than the addition of the **rec=** specifier, the read and write statements for direct files are the same as for sequential files. The record-number is any (default) integer expression, the value of which specifies the record to be processed.

All of the records of a direct file are the same length (this does not have to be the case for sequential files). **access="direct"** and the **recl=** specifier must be included when opening direct files; note that the default formatting for direct files is **"unformatted"**, and thus **form="formatted"** must also be specified if the direct file is to be formatted. The inquire statement with the **recl=** and **nextrec=** specifiers can be used to, respectively, determine the record length of a direct file and the record following the last record processed with a direct file read or write statement.

partial-record (nonadvancing) I/O

As mentioned above, Fortran I/O is fundamentally record oriented, and explicit specification is needed for a read (write) statement to consume (produce) only part of a record. In (the default) whole-record I/O, the position of the file is said to *advance* to the next record after a read or a write statement. Thus partial-record I/O is called *nonadvancing* I/O, as the file position is "left where it is" rather than advancing to the beginning of the next record. In nonadvancing input the position of the file is left at the beginning of the next datum within the record that has not yet been read, and the next read statement continues reading from that point; in nonadvancing output an end-of-record is not written by the write statement, and the next write statement continues the same output record.

Nonadvancing I/O is specified with the **advance="no"** specifier (R912) in the read or write statement; nonadvancing can be specified only for sequential, formatted I/O, so the general forms of the read and write statements for partial-record I/O are:

read ([**unit=**] *unit* **,** [**fmt=**] *format* **, advance="no"** [**, iostat=** *ios-variable*] **)** *input-list*

write ([**unit=**] *unit* **,** [**fmt=**] *format* **, advance="no"** [**, iostat=** *ios-variable*] **)** *output-list*

Note the (syntactic) similarity of nonadvancing I/O with direct I/O, the only differences being that the format is not optional, and may not be an asterisk, and there is an **advance=** specifier rather than a **recl=** specifier. The "no" can be any scalar (default) character expression which evaluates to either "no" or "yes" (upper/lower case immaterial); "yes" represents (the default) ordinary whole-record sequential formatted I/O. An extended example of nonadvancing read and write statements is:

```
! read from a file the day-month-year, such as "24 September 1987"; year position unknown;
! write the results in a (slightly) different form to the screen, interleaving the writes with the reads
read (f, fmt="(I2)", advance="no") day          ! assume that i, j, day, year are integer
write (*, fmt="(I3,TR1)", advance="no") day     ! print day to screen, blanks on both sides
do i=1,10
  read (f, fmt="(A1)", advance="no") month(i:i)  ! assume that month, m are character
  if (i>1.and.month(i:i)==' ') exit              ! read characters through second blank
end do                                           !    (first character of month is a blank)
```

```
m = "*** January February March April May June July "     //                &
        " August September October November December "
j = index(m, month(1:i) )
write(*, fmt="(A3)", advance="no") m(j+1:j+3)   ! now write first three characters of the month
read(f, fmt="(I4)") year                        !     (or three asterisks if month "error"  in file)
write(*, fmt="(I5)") year                       ! finish with advancing read and write for year
! for the above example data the output to the screen is " 22 Sep 1987", and no more, on one line
```

This example illustrates the advance="no" option, of course, but also illustrates that non-advancing can be used with both file I/O and screen/keyboard I/O. If the interleaving of the read and write statements had been important then, because of the unknown length of the Month data, the partial-record I/O is exactly the tool needed. If the interleaving is not important (which it probably isn't in the example as shown) then whole-record read and writes could have been used, together with and internal read (see the next section). Generally speaking, nonadvancing I/O is indicated when some action must be taken after part of a record (line) has been input or output and before it is completed.

In nonadvancing I/O the iostat= specifier can be used to detect end-of-record on input as well as end-of-file. At end-of-file the iostat variable has a negative value and a different negative value at end-of-record. Unfortunately, these values are implementation dependent, although some implementations provide a module with named constant definitions that include the end-of-file (typically named EOF) and end-of-record (EOR) values. If the implementation does not provide such definitions, at least the documentation should identify what these values are for that implementation; then the programmer can provide the appropriate named constant definitions. If these values are not readily available then resort must be made to the **end=** and **eor=** specifiers, which specify the branch point label if the input encounters end-of-file or end-of-record, respectively. The **eor=** specifier is available for use only with nonadvancing input.

internal data conversion (internal files)

The file unit in a whole-record formatted sequential read or write statement may be a character variable, rather than an integer expression or an asterisk. If the variable is an array then the effect is as if the array represents a formatted sequential file, each element of the array being one record of the file; if the variable is a scalar (a scalar character variable, a substring, or an array element), the "file" is a one-record file. Such a "file" is called an *internal file*, as the file (character variable), as well as the input or output list entities, is an internal entity of the program. A read statement specifying an internal file is an *internal read* and a write statement specifying and internal file is an *internal write*. The general form of internal reads and writes are:

read ([**unit=**] *unit* **,** [**fmt=**] *format* [**, iostat=** *ios-variable*] **)** *input-list*

write ([**unit=**] *unit* **,** [**fmt=**] *format* [**, iostat=** *ios-variable*] **)** *output-list*

in which the unit in the read statement is a (default) character expression and the unit in the write statement is a (default) character variable (R901, R903).

The purpose of an internal write is to convert a set of expression values (the output list) to a (sequence of) character string(s), just as a formatted write to an external file; the purpose

of an internal read is to convert a (sequence of) formatted record(s) - any formatted record is simply a string of characters - to the proper internal values for a set of variables (the input list), exactly the role of a formatted read of an external file. Thus internal reads and writes are exactly the same as external reads and writes, except that the records are internal character objects rather the same kind of thing in an external file.

An external read “reads from” the record, converts the character values as specified by the format, and defines the variables in the input list; the record itself is not changed and therefore the internal file may in fact be any character expression - it is not limited to a variable. On the other hand, an external write converts the values in the output list to characters, according to the format, and “writes” the record; hence the internal file for an internal write must be a character variable that can be assigned a value. Any character variable, or element, section, or substring thereof, in an internal file must not appear in the format (if specified as a character expression) or the input or output list.

Internal files are most often used to convert between numerical values and character string (and vice versa). A common instance of this is when the format of an record is not known until after it has been read (into a character string); after determining the format, an internal read can convert the values to the “target” variables. For example, consider the example of the preceding section. A whole-record read, coupled with internal reads, can accomplish the same thing as the nonadvancing reads:

```
! read from a file the day-month-year, such as "24 September 1987"; year position unknown;
read (f, fmt="(A)") iFile                          ! assume that iFile isa  character variable
i = index(iFile, ' '); i = index(iFile(i+1:), ' ')  ! assume i, j , day, year are integer
                                                   ! i is location of the second blank in IFile
read (iFile(:2)), fmt'"(I2)) day                   ! internal read to convert
month = iFile(3:j)                                 ! assume month, m are character
read (iFile(i:i+4)), fmt'"(I5)) year               ! internal read

m = "*** January February March April May June July "     //                    &
        " August September October November December "
j = index(m, trim(month)//' ')
write (*, fmt="(I3,TR1,A3,I5)") day, m(j+1:j+3), year   ! ordinary advancing write to the screen
! output is the same as in preceding section, but all with whole-record and internal I/O
```

Note in this example that the contents of the entire external record are read into an ordinary character variable (**iFile**). The **index** function is then used to identify the substrings on which internal reads are used to convert the numeric data into the desired variables.

If the internal file cannot be an unallocated array or unassociated pointer; the file for an internal read must be a valid character expression (that is, all parts must have defined values). Data transfer in internal file I/O is the same as in external file I/O. Internal file I/O may be list directed (see next section), but may not be name directed (namelist), direct, unformatted, or nonadvancing. The file connection (open and close statements), inquiry (inquire statement), and positioning (backspace, rewind, and endfile) cannot be used with internal files; the **blank="null"**, **delimit="none"**, and **pad="yes"** connection specifications are assumed.

list-directed and name-directed I/O

All of the formatted I/O described above involve a programmer-supplied *format* that specifies the exact columns in which output values are to be written and from which input values are to be read. Fortran provides two forms of sequential advancing formatted I/O in which the formatting is system-supplied rather than programmer-supplied. For output, the system supplies (typically) blank-delimited values written according to some system-supplied data edit descriptors; input values are "free form", delimited by *value separators* (usually blanks or commas).

List-directed I/O syntax is:

read ([**unit=**] *unit* **,** [**fmt=**] ***** [**, iostat=** *ios-variable*] **)** *input-list*

write ([**unit=**] *unit* **,** [**fmt=**] ***** [**, iostat=** *ios-variable*] **)** *output-list*

The format is an asterisk to indicate system-supplied formatting. The unit may be either an internal or external file, or an asterisk (keyboard/screen); when the unit is an asterisk the "short form" may be used: **read * ...** and **print * ...** .

In list-directed input, the first value is extracted from the record and converted for assignment to the first variable in the input list, the second value extracted, converted, and assigned to the second variable, and so on until values have been read for all of the input list variables, or until a slash (*/*)or end-of-file is encountered. If a slash (or end-of-file) is encountered and some of the variables have not been assigned new values, those input list variables retain their current values.

A value in the input file is separated from the next value by a blank or a comma; a comma may optionally be preceded and/or followed by a blank. (Multiple consecutive blanks in the input, but not in a character value, are equivalent to a single blank, and blanks are never zeros.) Consecutive commas (possibly with intervening blanks) represent *null* input values, and the corresponding input list variable values are not changed. The end of a record is treated as a blank.

A numeric input value must be in the form of a numeric constant, assignment compatible with the input list variable to which it is to be assigned, but binary, octal, and hexadecimal constants must not be used in list-directed input and the separator characters (blank, comma, and slash) cannot be part of a logical value (they would be treated as value separators). A character value may be a character constant, assignment compatible with the corresponding input list variable, or not delimited at all; in the last case the first appearance of a separator character terminates the character value; a separator character appearing in a character constant is part of the constant, however, and not treated as a separator.

A value may be repeated, much as in the data statement (R532), by preceding it with an unsigned (default) integer constant followed by an asterisk; for example: 4*1.0 is equivalent to 1.0, 1.0, 1.0, 1.0. Such a repeated construct must contain no blanks (any blank would serve as a value separator). The repeat without a constant specifies that many consecutive nulls; for example 6*, specifies six nulls and is equivalent to , , , , , , , .

The format for list-directed output is entirely implementation dependent, except that it must be (almost) suitable for list-directed input of the same values in the same sequence. An integer value is in I format, a real value is in F or E format, the real and imaginary parts of a complex value are enclosed in parentheses and separated by a comma, a logical value is T or F, and a character value is delimited by the character specified by the delim= specifier in the open statement (note that the delim= default is “none”); if a character value is written with single (or double) quote delimiters, any single (or double) quote characters in the character values are repeated once, as per the rules for character constants. The “almost” in the first sentence of this paragraph refers to the situation where a character value containing separator characters is written with delim="none"; in this case the list-directed output does not represent the same set of values for subsequent list-directed input. A list-directed write statement may output any number of records, and the first character of each such record is a blank.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Name-directed I/O is similar in many respects with list-directed I/O, except that the values in the input or output list can be in any order. This is accomplished by making two syntax changes in the read and write statements and keywording the input (and output) values. The value keywords are modeled after the specifiers (e.g., iostat=...) of the various I/O statements - the value is preceded by a variable= construct, where variable is the identification of the variable to receive that value (input) or whose value is to be written (output); note that name-directed output works only with variables, not with arbitrary expressions.

The following forms for the name-list read and write statements illustrate the two syntax changes:

read ([**unit=**] *unit* **,** [**nml=**] *group-name* [**, iostat=** *ios-variable*] **)**

write ([**unit=**] *unit* **,** [**nml=**] *group-name* [**, iostat=** *ios-variable*] **)**

The two changes are: (1) the **fmt=*** has been replaced by **nml=***group-name*, and (2) there is no input list or output list, which are also replaced by *group-name*. The group name is defined by the **namelist** specification statement (R543-544), the simplest form of which is:

namelist / *group-name* **/** *variable-name-list*

The namelist statement associates a name (the group name) with a list of variable names; some or all of these variables, or parts of them, are assigned input values by a name-directed read statement; similarly the values of some or all of these variables, or parts of them, are output by a name-directed write statement. Each name in the variable name list of a namelist statement must be a scalar or array variable name, but cannot be the name of an array dummy argument with a nonconstant bound, an allocatable array, a variable with a nonconstant character length, a pointer or an structure containing a pointer, or any other nondummy argument local variable without the save attribute.

The variable part of a variable=value pair may specify one of the names in the variable name list, or an element or section of an array having one of those names, a component of a structure with one of those names, or a substring of a character variable with one of those

names. The value part of a variable=value pair may be empty (null value), a single scalar value (if the variable is a scalar), or if the variable is an array (or array section) the value is either a sequence of comma or blank separated scalar values or of the repeated "*" form (as in list-directed I/O); as in list-directed I/O, if there is no value following the "*" then that many null values are specified.

Name-directed input starts with an ampersand followed (immediately) by the group name, followed by a set of variable=value pairs (in any order), and ending with a slash. Each variable=value must be preceded by a comma or a blank; there may be a blank between the = and the value. If the variable and value are scalar, that value is assigned to the specified variable; if the variable is array valued then the specified sequence of scalar values are assigned to the corresponding elements of the array. Unlike list-directed input, a character value in name-directed input must be a character constant; otherwise name-directed input is pretty much the same as list-directed input. Name-directed output has an implementation-dependent form, but must be (almost) suitable for subsequent name-directed input for the same variable values; the "almost" is the same as for list-directed I/O - if delim="none" then character output is not delimited and thus likely would not represent the same values for namelist input.

Examples of list-directed and name-directed I/O are:

```
read (labData, fmt=*) name, weight(1:4)     ! #1 ! weight is an array
"white rats"     4.3, 5.1, 4.6, 4.3                              (input for #1)

print *, name, weight (3)                   ! #2 ! output the values read in #1
 white rats 4.319683                                            (output for #2)

namelist / labVars /  name, weight          ! namelist for #3 and #4, specification part
read (labData, nml=labVars)                 ! #3 ! same data read as in #1
&labVars  weight(1:4)=4.3, 5.1, 4.6, 4.3   name="white-rats" /     (input for #3)

write (*, nml=labVars)                      ! #4 ! using the same namelist as defined for #3
&labVars weight(1:4)= 4.3 5.1 4.6 4.3 name= white-rats /          (output for #4)
```

Each of the above input statement reads data from a file (labData), and prints it on the screen. Note that the **print *** statement may be used for list-directed screen output, but not for name-directed output. Also note the subtle change in the input data between cases #1 and #3 - in the first case the name is two words separated by blanks, in the second the blank is replaced with a hyphen; the output for #2 is not suitable for re-input with the #1 read statement, but the #4 output is suitable for re-input with the #3 read statement.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

As mentioned in the preface, this is an extremely concise version of Fortran's considerable I/O sublanguage. It is intended to be complete, however, with nothing left out and no need to consult other references. Nevertheless the compactness of this description may have "glossed over" some subtle details which may be better described in either of the references given in the preface or some other sources of Fortran 90 information.

7 Control Structures

Control structures allow the programmer to (a) select which statements are executed next (**if** and **select**), (b) specify repetitive execution of a group of statements (**do**), and (c) branch to another part of the program (**goto**).

if construct

The **if** construct has the basic form:

```
if ( logical-expr ) then
   executable-constructs                ! the block (R215) "guarded" by logical-expr
end if
```

If the *logical-expr* is true then the block of executable constructs is executed; if the *logical-expr* is false then the block is skipped. Note that the executable constructs can be any mix of action statements and other (nested) constructs, such as loops, other if constructs, etc. The syntax of the if construct is more fully described in syntax rule R802 and the rules it references.

Another useful form of the **if** construct is:

```
if ( logical-expr ) then
   executable-constructs                ! the block guarded by logical-expr
else
   executable-constructs                ! the alternative block, if logical-expr is false
end if
```

In this form the first block of executable constructs (between the **if** and **else**), but not the second (between the **else** and **end if**) is executed if the *logical-expr* is true, and just the reverse if it is false. This form of the **if** construct has one logical expression and two blocks of executable constructs. One can think of the logical expression as "guarding" the first block; if the "guard" is true the first block is executed and the second one isn't; if the guard is false the block it guards (the first one) is not executed and the second one is.

The most general form of the **if** construct involves n guards (logical expressions) and n+1 blocks of executable constructs (where n can be any integer value greater than zero):

```
if ( logical-expr ) then             ! the first guard,
   executable-constructs             !         and the block it guards
[ else if ( logical-expr ) then      ! any number of additional guards,
   executable-constructs ] ...       !         and the blocks they guard
[ else
   executable-constructs ]           ! the alternative block, if all guards are false
end if
```

Each guard is associated with (guards) "its own" block of executable constructs. The executable construct associated with the first guard that is true is executed and the rest of the **if** construct is skipped. If none of the guards are true then the unguarded block (the one between **else** and **end if**) is executed. Note that the optionality brackets in the above gen-

eral form show that the other two forms described above (with just one guard, with and without an unguarded block) are just special cases of this general form.

An example of an if construct with two guards is:

```
if ( temperature > BOILING ) then
  . . .                                    ! vapor phase
else if ( temperature > FREEZING ) then
  . . .                                    ! liquid phase
else
  . . .                                    ! solid phase
end if
```

Finally, Fortran has a single-line **if** statement, also called the *logical if*:

```
if ( logical-expr )  action-stmt           ! see R216 for definition of action-stmt
```

In this case the action statement is executed if (and only if) the logical expression is true. The logical if is especially handy when you want to get out fast:

```
if ( code == "DONE" )  exit                ! exit loop when processing is finished

if ( n > 20 )  return                      ! computation of routine is completed

if ( disaster )  stop "DISASTER!!"         ! anywhere disaster strikes
```

The guards in an **if** construct need not be disjoint - that is any of them can be true at the same time. But only one block of executable constructs is executed - that one guarded by the first (top most) guard that is true. In the case (**select**) construct, however, at most one guard can be true at any given time.

case construct

The **case** construct also involves n guards and n+1 blocks of executable constructs, only one (or, more precisely, at most one) of which is executed. The order of the guards in a **case** construct is immaterial (whereas the order of the guards in an **if** construct may well be critical - witness the temperature example above).

The case construct syntax is described in detail in R808 and the syntax rules it references. The general form is:

```
select case ( case-expr )                  ! the case expression to be evaluated;
[ case ( case-value-list )                 ! if the case-expr value matches one of these
  executable-constructs ] ...              !         then this block is executed;
[ case default                             ! if no match exists
  executable-constructs ]                  !         then this block is executed;
end select
```

The case expression and case values may be of type integer, type character, or type logical; all of the case values are constants, and all of the case-value lists are disjoint. The block of executable constructs corresponding to (guarded by) the case-value list that contains the value of the case expression (or the default block, if there is one and there is no value

match) is the block executed. Since the case-value lists are disjoint, the case expression value can match at most one, and therefore the order of the case blocks, including **case default**, is immaterial.

The case-value lists are comma-separated lists of constants or constant ranges, as described by R813. Examples of the **case** construct are:

```
select case ( shape )                          ! names in all caps are named constants
case ( CIRCLE );      area = d*d*PI/4
case ( SQUARE );      area = d*d
case ( TRIANGLE );    area = d*d*sqrt(3.)/2
case ( HEXAGON );     area = 6*d*d*sqrt(3.)/8
case default;         area = f_area(shape,d)
end select

select case ( age )
case ( 0:17 )         . . .                    ! youth
case ( 18:61 )        . . .                    ! adult
case ( 62: )          . . .                    ! senior
end select
```

do construct

Modern loop constructs do not involve statement labels. For compatibility with older Fortran code, which makes extensive use of the original labelled form of the **do** construct, Fortran 90 has three categories of **do** construct:

(a) a modern construct (**do - end do**) without any labels

(b) the modern construct with labels

(c) the original style

Category (b) is the same as (a) with an optional label, and is provided for those who want the modern structure but prefer to have loops with labels.

The syntax of all forms of the **do** construct is described in R816 and the syntax rules it references. The modern form, without labels (category (a)), itself comes in three flavors - *infinite*, *indexed*, and **while**:

```
do                                             ! the "infinite" form -
  executable-constructs                        ! looping stops only by explicit exit
end do                                         ! from within the loop body

do  int-variable = int-expr, int-expr  [ , int-expr ]
  executable-constructs                        ! the indexed form
end do

do while ( logical-expr )
  executable-constructs                        ! the while form
end do
```

Execution of the *infinite* **do** construct "loops forever" unless there is an **exit** statement somewhere in the block of executable constructs (loop body).

The semantics of the *indexed* **do** is best described by making the first line more specific:

```
do  i = e1, e2, e3
```

Then the semantics of the *indexed* **do** are equivalent to:

```
i = e1-e3
do; i = i+e3; if ( i > e2 )  exit
  executable-constructs
end do
```

If **e3** is omitted, a value of +1 is assumed; if **e3** is negative then the test is **i < e2** rather than **i > e2**. Of course a value of zero must not be specified for **e3**. (See R817 and related syntax rules for additional, but inconsequential, syntax details of the indexed **do** construct.)

The semantics of the **while** loop are:

```
do; if ( .not.  logical-expr )  exit          ! in the while loop the test is
  executable-constructs                       ! made at the top of the loop
end do
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

The modern form with labels (category (b)) simply replaces **do** and **end do** with labelled versions of these statements (same label on both); the *label* is described in R313; the unlabeled and labelled versions of these two statements are summarized as follows:

unlabeled	labelled
do	**do** *label*
end do	*label* **end do** *label* **continue**

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

The original style DO-loop (category (c)) required labels and did not have **end do**. Though it allowed terminating a loop on a labelled **continue** statement, it did not require it; the (labelled) last statement could be the final action statement of the loop body (R827). (Note that such an action statement could not, however, be a **goto**, **return**, or other branching statement.) Moreover, two (or more) nested loops could share the same termination statement (and label) - see R830. For example, a two-dimensional array can be initialized with the nested loops:

```
do 101,  i=1,m
  do 101,  j=1,n
101 x(i,j) = 0
```

whereas the modern version of these nested loops would be:

```
do  i=1,m
  do  i=1,m
     x(i,j) = 0
  end do
end do
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

The execution of any loop may be explicitly and immediately terminated (with an **exit** statement, R835) or advanced to the next iteration (with a **cycle** statement, R834) anywhere within the loop body. The most common use of these features is to exit a loop upon the occurrence of some condition. A common pattern, for example, is the read-test-process nature of processing file data:

```
do
  . . .                                ! read the next record
  if ( end-of-file )  exit             ! test to see if at end of file
  . . .                                ! process the data just read
end do
```

If only part of the records are to be processed, then the loop could be:

```
do
  . . .                                ! read the next record
  if ( end-of-file )  exit             ! test to see if at end of file
  if ( not-of-interest ) cycle         ! record not of interest, so go on to next
  . . .                                ! process the data just read
end do
```

In the nested loop case these simple forms of **exit** and **cycle** apply only to the inner-most loop in which they appear. To make them apply to an outer loop, that outer loop must be named with a *construct name* (R818, R825) and the **exit** (or **cycle**) statement must specify this name. For example:

```
outer_loop: do
   . . .
   do
      . . .
      if ( . . . ) cycle outer_loop
      . . .
   end do
   . . .
end do outer_loop
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

goto statements

The **if**, **case**, and **do** constructs provide the disciplined, readable, and reliable way to control the execution sequence. The **goto** statement, and variations, are primitive, but powerful, *branching* statements that allow execution to be switched to (almost) any other place in the current scope. The basic **goto** (which can also be spelled **go to**) has the form:

```
goto label                             ! see R836
```

Execution of a **goto** statement causes execution to resume with the statement having the *label* specified in the **goto** statement. Any action statement may be labelled, as well as the **if**, **select**, or **do** statement (that is, first statement) of a control construct.

A **goto** statement must not cause a branch into the body of an **if**, **select**, or **do** construct from outside that construct, but the reverse (branch out from inside) is allowed.

Though a *label* may be placed on any action statement, plus a few others, many programmers prefer to use only the **continue** statement to identify a branch point:

```
label  continue                    ! see R840; execution of continue "does nothing"
```

The *computed-goto* statement (deprecated - see chapter 5) specifies a list of labels and causes a branch to one of them, depending on the value of an integer expression:

```
goto ( label-list ) int-expr       ! see R837
```

For example:

```
goto ( 222, 333, 222 )  K/5        ! assume K is an integer variable
```

causes a branch to the statement labelled 222 if K is in the 5-9 range or 15-19 range, to the statement labelled 333 if K is in the10-14 range, and has no effect otherwise.

The *assigned-goto* statement (also deprecated) uses an integer variable as a label:

```
goto int-variable  [ ( label-list ) ]      ! see R839
```

Prior to executing an assigned-goto statement, the integer variable must have been assigned a label value with the *assign-statement* (also deprecated):

```
assign  label  to int-variable             ! see R838
```

If the optional label-list is included in the assigned-goto statement, the label value of the integer variable must match one in the list.

The *arithmetic-if* (also deprecated) causes a branch to a specified label, based upon a numeric value:

```
if ( numeric-expr )  label , label , label     ! see R840
```

If the numeric value is less than zero the branch is to the first label, if the value is zero the branch is to the second label, and if the value is greater than zero the branch is to the third label.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

The **stop** statement (R842-843) is a special kind of a branch - it terminates execution of the program; it can be placed in any execution sequence and, as an above example indicates, can be used to "get out quick" when disaster strikes. In normal use it is redundant, however, as the end statement of the main program serves to terminate execution of the program. Note that in those exceptional cases where it is useful, the stop statement can issue a relevant message, which on most systems is printed on the screen at termination.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

8 Modules

The *module* is a new program unit in Fortran 90 that provides definitions for use in other program units; modules may be placed in separate files and re-used with different programs.

A module is not executable; it contains definitions to be used by other program units; these definitions include procedures, which are individually executable, but the module itself is not executable.

Module entities are made available to other program units by the *use* statement:

```
program Seismic_Processing
 use Seismic_Trace_Definitions
 ...
end program
```

Available modules entities are said to be “use associated” within the using program. A module may contain *private* entities; such entities are not available to using program units (private entities are available only within the module itself, including any procedure definitions within the module). In addition, the *only* form of the use statement limits the module entities that are available in the using program unit.

The principal contents of a module include: type definitions, interface definitions, procedure definitions, and shared data objects (including global constants). Typical uses of modules include: procedure libraries (with explicit interfaces), encapsulated data abstractions, and shared-data units (alternative to COMMON).

module structure

The general structure of a module (R1104-1106) is as follows; additional syntax rules are listed for specific items in the following description:

```
module  module-name
   use-statements          ! modules can (optionally) use (import) other modules
   constant-definitions    ! global constants - see R538 and the parameter attribute
   variable-declarations   ! shared variables - see R501
   interface-blocks        ! explicit interfaces, defined operators, overloading - see R1201
   type-definitions        ! user-defined data types - see R422
contains
   module-subprograms      ! see R213
end module
```

module use

Other program units (main programs, functions, subroutines, and other modules) may use the definitions provided in a module by including a **use** statement (R1107-1109) immediately after the program unit heading, as in the Seismic_Processing example above:

```
use module-name [ rename-list ]        ! imports all public entities of the module

use module-name, only: only-list       ! imports only the specified public entities
```

Module entities are imported into the using program with the same name as they have in the module, unless they are renamed in the use statement; this may be necessary to avoid name conflicts, as an imported ("use associated") name does not "mask" a local entity with the same name. A *rename* (R1108) has the form:

```
local-name => use-name
```

where local-name is the new name (in the using program) and use-name is the name of the entity in the module (the new name "points to" the module entity). An *only* (R1109) can be either the name of the module entity being imported, or a rename:

```
[ local-name => ] use-name                  ! renaming is optional in an only-list
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

A module for the above Seismic_Processing program might take the form:

```
module Seismic_Trace_Definitions
 real, parameter :: PI=3.1415926           ! a global constant
 real, allocatable :: seismicWorkArray(:,:) ! a shared data work-space

 interface FFT                             ! overloading the procedure name FFT
  subroutine FFT_C (...)
    ...
  end subroutine
 end interface

 type SeismicTrace                         ! defining a "Seismic_trace" data type
  real :: trace(1000)
 end type

contains
!-----------------------------------------
 function FFT (...)                        ! definition of function FFT
    ...
 end function FFT
!-----------------------------------------
 subroutine FFT_C (...)                    ! subroutine FFT_C can also be called
    ...                                    ! with the name FFT, because of the
 end subroutine                            ! overload defined above
!-----------------------------------------
 subroutine timeDomain (...)               ! another procedure definition in this module
    ...
 end subroutine
!-----------------------------------------
end module
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

module applications

A module might contain just shared (global) constants and variables:

```
module Shared_Data
  real, parameter :: PI=3.1415926          ! a shared (global) constant
  integer :: n_rho, n_vel                  ! two shared variables
  real, allocatable :: workArray(:,:)      ! a shared array
end module
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

A module might comprise a procedure library:

```
module Procedure_Library
contains
!-----------------------------------------
 function FFT (...)
    ...
 end function FFT
!-----------------------------------------
 subroutine timeDomain (...)
    ...
 end subroutine
!-----------------------------------------
 ...
!-----------------------------------------
end module
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

A module might be used to collect a set of procedure interfaces, for use by other program units; these can include (1) explicit interfaces for external procedures, (2) generic (overloaded) names for specific procedures, and (3) operator symbol definitions:

```
module Procedure_Interfaces
!-----------------------------------------
 interface                                 ! providing an explicit interface
  subroutine T_time (A)                    ! for external procedure T_time
   real :: A(200,300)
  end subroutine T_time
 end interface
!-----------------------------------------
 interface FFT                             ! overloading FFT for use with FFT_C and FTT_R
  subroutine FFT_C (...)
     ...                                   ! FTT_C is defined external to this module
  end subroutine
  module procedure FFT_R                   ! and FTT_R is defined in this module
 end interface
!-----------------------------------------
```

```
 interface operator(.inverse.)          ! defining the operator ".inverse.";
   module procedure inverse             ! the procedure itself (inverse) is defined
 end interface                          ! in the procedure part of the module
!----------------------------------------
 interface operator(+)                  ! extending the use of the operator "+"
   function or(a, b)                    ! to use with logical operands;
     logical :: or                      ! function "or" is defined external to this module
     logical, intent(in) :: a, b
   end function
   module procedure addCharDigits       ! extending "+" to character digits as well
 end interface
!----------------------------------------
contains
!----------------------------------------
 function inverse (matrix)                  ! the definition of function "inverse"
    ...
 end function
!----------------------------------------
 function addCharDigits (d1, d2)            ! definition of adding character digits
    integer :: addCharDigits
    character :: d1, d2
       ...
 end function
!----------------------------------------
end module
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

A module might contain definitions of user-defined types (or record structures):

```
module Derived_types
!----------------------------------------
 type Point                             ! a type to similate a 3D point in two ways
   real :: x_r
   real :: y_rho
   real :: z_theta
   logical :: cartesian
 end type
!----------------------------------------
 type List                              ! a typical linked list structure node
   type (Point) :: data
   type (List), pointer :: next
   type (List), pointer :: prev
 end type
!----------------------------------------
 type Seismic_trace; private            ! a new type with "hidden" internal structure
   character(20) :: trace_ID
   real, pointer :: traceData(:)
 end type
!----------------------------------------
end module
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

A module might encapsulate a complete data abstraction:

```
module Interval_Arithmetic                  ! a data abstraction for interval arithmetic
!-----------------------------------------
 type Interval; private                     ! the basic data structure is private; an interval
  real :: lower,  upper                     ! is represented by its upper and lower bounds
 end type
!-----------------------------------------
 interface operator(+)                      ! use "+" for adding intervals
   module procedure  interval_plus_interval,                                    &
                     interval_plus_real,                                        &
                     real_plus_interval
 end interface
!-----------------------------------------
 interface operator (*)                     ! use "*" for multiplying intervals
   module procedure  interval_times_interval,                                   &
                     interval_times_real,                                       &
                     real_times_interval

 end interface
!-----------------------------------------
 interface sqrt                             ! extend "sqrt" to interval arguments
   module procedure  interval_sqrt
 end interface
!-----------------------------------------
    . . .                                   ! other interfaces ....
!-----------------------------------------
contains
!-----------------------------------------
 function interval_plus_interval (a, b)     ! one of the addition functions
   type (Interval) :: interval_plus_interval
   type (Interval), intent(in) :: a, b
      ...
 end function
!-----------------------------------------
 function interval_plus_real (a, b)         ! another addition function
   type (Interval) :: interval_plus_real
   type (Interval), intent(in) :: a
   real, intent(in) :: b
      ...
 end function interval_plus_real
!-----------------------------------------
 function real_times_interval (a. b)        ! etc....
   type (Interval) :: real_times_interval
   real, intent(in) :: a
   type (Interval), intent(in) :: b
      ...
 end function
!-----------------------------------------
```

```
 function interval_sqrt (X)                    ! definition of interval square root
   type (Interval) :: interval_sqrt
   type (Interval), intent(in) :: X
      ...
 end function interval_sqrt
!-----------------------------------------
 function interval_mid (x)                     ! to return the mid-point of an interval
   real :: interval_mid
   type (Interval), intent(in) :: x
      ...
 end function
!-----------------------------------------
    . . .                                      ! other procedure definitions....
!-----------------------------------------
end module
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

9 Procedures

Subroutines and *functions* are the two forms for Fortran 90 procedures; each may appear in the following contexts: *external* (stand-alone, separately compiled), *module* (packaged within a module definition), *internal* (packaged within another procedure or main program definition). Fortran 90 has a large number of built-in *intrinsic* procedures; these are summarized here and described in detail in chapter 10. Both subroutines and functions are used to encapsulate particular computations, and the principal difference between them is the manner in which they are used (*called*, *invoked*). A subroutine does not return a value (except possibly through its argument list) and is invoked by a separate subroutine *call* statement; a function returns a value and is invoked by using its name (and arguments) in an expression in which its returned value is used.

Procedures interact with each other (primarily) through argument lists, and these *interfaces* are either *explicit* (known) or *implicit* (unknown) to the calling procedure; only external procedures have implicit interfaces; *interface blocks* can make even these interfaces explicit. Procedures can be *generic* (multiple procedures called by the same name), and procedures may be used to define *new operators*.

subroutines

A subroutine is defined by a subroutine subprogram (R1219), which has the form

```
subroutine subroutine-name ( dummy-arg-list )! see R1221 for dummy arguments
  specification-part                          ! see R204 for specification-part
  execution-part                              ! see R208 for execution-part
  internal-subprogram-part                    ! see R210 for internal-subprogram-part
end subroutine
```

Any of the three parts in a subroutine definition may be empty, but a subroutine will normally have a specification part (e.g., argument declarations) and do some computation (execution part). If the subroutine is *recursive*, it definition starts with the keywords **recursive subroutine** rather than just the keyword **subroutine**. See R1219 for some minor syntax options (such as allowing the subroutine name to be repeated on the **end subroutine** statement).

A subroutine definition may be placed in:

(1) the *internal-subprogram-part* of a module (R1104), in which case it is a *module subroutine*,

(2) the *internal-subprogram-part* (R210) of an external subroutine definition, an external function definition, a *main-program* (R1101), or a module procedure definition, in which case it is an *internal subroutine*,

(3) its own file, or a file with other stand-alone procedure definitions, in which case it is an *external subroutine*.

An external subroutine has an implicit procedure interface (if no interface block is supplied for it) and, as it has no *host*, does not access a host data environment. (An external subroutine is also the only kind of subroutine in which **end** is an acceptable abbreviation of the **end subroutine** statement.)

A module subroutine has an explicit interface and accesses the data environment of its host module; an internal subroutine has an explicit interface and accesses the data environment of its host procedure.

In all cases, a subroutine is invoked with a **call** statement (R1210):

```
call subroutine-name ( actual-arg-list )     ! see R1213 for actual arguments
```

If the subroutine's interface is explicit the compiler can enforce consistency between the *actual-arg-list* in the call statement and the *dummy-arg-list* in the subroutine definition; otherwise interface consistency is not enforceable. (Interface inconsistency is the source of difficult-to-find errors in programs with implicit interfaces.)

(Note: subroutine argument lists may include *alternate returns* (deprecated - see chapter 5), and this is the only way that subroutine argument lists differ from function argument lists. An alternate return is an asterisk (*****) in a *dummy-arg-list*; the corresponding actual argument must be a ***label* (e.g., ***220**) specifying the alternate return point.)

functions

A function is defined by a function subprogram (R1215), which has a form similar to that of subroutines:

```
[ type-spec ] function function-name ( dummy-arg-name-list ) [ result ( result-name ) ]
  specification-part                 ! same
  execution-part                     !   as in
  internal-subprogram-part           !     subroutines
end function
```

As with a subroutine, **recursive** must be added to a recursive function (R1217), and the *result clause* is required for recursive functions (and optional for all other functions). See R1215 for other (minor) syntax options. Functions may be internal, module, or external in exactly the same way as subroutines, with the same implications in each case.

Each function returns a *result value* and therefore must have a result type; that type may be specified on the function statement (*type-spec*) or in the specification part of the function. Function results may be of any type, including derived type, and may be array valued; if the function is array valued, the array (dimension) attributes must be specified in the specification part. The result clause, if present, specifies the name to which the result is assigned in the execution part; otherwise the function name is also the result name.

A function is invoked as an operand in an expression, not as a stand-alone statement, and its result value becomes the value of that operand; such a function call has the form (R1209):

function-name **(** *actual-arg-list* **)**

Explicit interfaces and arguments work the same for functions as for subroutines, except that alternate returns are not allowed in function arguments lists. The result clause is required for recursive functions for the following reason. Consider:

```
recursive function rf(a,b) result(rf_result)
  integer :: a, b
  real :: rf_result(2,3)
    ...
    ... = rf(2,1)
    ...
end function rf
```

If the result clause had not been present, then **rf** would be used for the result value as well as for recursive calls. Because **rf** has two integer arguments and returns a two-dimensional array, the reference **rf(2,1)** could be either a recursive call or a reference to an element of the result. Though only very few combinations of argument lists and arrayness cause such ambiguity, nevertheless the Fortran 90 rule is “recursive implies result”.

host association

All procedures have the following four data access mechanisms: (1) *local* entities, (2) *argument association*, (3) *use association*, and (4) *common* association. The second of these (argument association) allows a procedure to access entities explicitly “passed” by the procedure’s caller; the third (use association) refers to module entities accessed via use statements in the procedure’s specification part; the fourth (common association) refers to entities in the common blocks listed in the specification part. These are the only data access mechanisms for external procedures.

Module procedure and internal procedures have a fifth mechanism - *host association*. Host association refers to a procedure’s access to entities declared in the procedure’s host. The host of a module procedure is the module that contains its definition; the host of an internal procedure is the procedure (or main program) that contains its definition. Any entity declared in the host is automatically accessible by that declared name in the contained procedure - even private module entities (**private** affects only use association) - unless the contained procedure has an explicit declaration of that name. In the latter case the name refers to an entity local to the procedure and not to the host entity with this name; the host entity is then available to the procedure only through argument association. The following example illustrates these concepts:

```
program P
  integer :: x
  real :: y
    ...
    call s1(x)                    ! x is passed to the procedure s1
    ...
contains
```

```
 subroutine s1(z); integer z
  real :: x                          ! x is local; y is host associated
  ...                                ! execution part of subroutine s1
 end subroutine
end program
```

In the execution part of subroutine **s1**, **x** is (always) the local real variable (not the host's integer **x**), **y** is (always) the host's real variable, and **z** is the argument-associated entity; in the shown call to **s1**, **z** becomes associated with the host's **x**, and thus for this call **z** in the procedure's execution part refers to the host's **x**.

In the preceding example the host-association rules are simple and "obvious" because all entities are explicitly declared. Fortran 90 allows implicit declaration of variables, however, which can make host association less obvious. The above rules, as stated, still apply; the one missing piece is this: a variable implicitly declared in the host is host associated as if it were explicitly declared and a variable implicitly declared in a contained procedure (possible only if it has not been explicitly or implicitly declared in the host) is local to the procedure. Recall that a variable is implicitly declared if there is no type declaration for its name in the specification part, but its name is referenced in the execution part. The following example illustrates the host-association rules involving implicit declarations:

```
program P                            ! no type declarations in the host
  ...
  call s2(x)                         ! x implicitly declared in the host
  ...
contains
 subroutine s2(z)                    ! no type declarations in s2
  ...
  print *, x, y, z                   ! x is host associated
  ...                                ! y is local to s2
 end subroutine                      ! z is argument associated (with x in this case)
end program
```

In this example subroutine **s2** accesses three variables, **x**, **y**, and **z**, all implicitly declared. **x** is implicitly declared in the host, by virtue of its reference in the call statement, and thus is host associated in **s2**. **y** is not (explicitly or implicitly) declared in the host and so is implicitly declared in **s2**, by virtue of its reference in **s2**, and is therefore local to **s2**. **z** is argument associated.

The default implicit typing rules in a main program, module, or external procedure are: all default real except variables starting with letters I-N, which are implicitly default integer. Implicit typing can be turned off (with **implicit none**, which then requires that all variables to be explicitly typed) or modified by **implicit** statements (R540) in the specification part of the host. Whatever implicit typing is in effect in the host becomes the default implicit typing in the contained procedure. This can in turn be turned off (implicit none) or modified by implicit statements in the specification part of the contained procedure. These are straightforward, consistent rules, but by far the simplest host association scenario is explicit declaration of all entities in both the host and the contained procedures.

procedure arguments and argument association

Argument association is the mechanism by which a procedure has access to entities via an argument list. The procedure definition includes a dummy argument list, which specifies the properties of the passed entities in the context of the procedure body. These properties must be consistent with the actual argument entities specified in a call to the procedure. At the beginning of execution of each call, the dummy arguments are associated with (in a sense, become aliases for) the corresponding actual arguments.

This association requires that there be exactly the same number of arguments in the dummy and actual arguments lists (except for optional dummy arguments - see below), and that (in the absence of keyworded actual arguments - see below) corresponding positional arguments become associated; that is, the nth dummy argument (left to right) becomes associated with the nth actual argument. Consistency requires that the type and kind of each actual argument exactly match the type and kind specified for its associated dummy argument. In addition, if a dummy argument has the pointer attribute, the associated actual argument must also be a pointer.

In most cases, dummy arguments of type character should be *assumed length* (R508-509), meaning they should assume (inherit) the length of the associated actual argument. Similarly, dummy arrays can be *assumed shape* (R516), which means that they have declared rank (and all associated actual arguments must have this rank) but inherit the actual argument extent in each dimension. Assumed length is specified by an asterisk for the length parameter, and assumed shape is specified by a colon for each dimension; both are illustrated by this declaration of **r**, a two-dimensional character dummy array:

```
character(*) :: r(:,:)
```

Note that actual arguments that are substrings work well with assumed-length dummy arguments. Similarly, actual arguments that are array sections work well with assumed-shape dummy arguments.

This simple set of argument association rules is all that is needed, and is the preferred way, to develop new Fortran code and to update existing code. However, assumed-shape dummy arguments require explicit procedure interfaces, and much existing code predates Fortran 90, explicit interfaces, and assumed-shape dummy arguments. Fortran 90 allows the *array element sequence association* mechanism for array arguments that Fortran provided prior to Fortran 90. In this case the dummy argument is declared as either an *explicit-shape* array (R513) or an *assumed-size* array (R518); the principal difference between the two is that the last dimension of an assumed-size array is declared as an asterisk. An actual argument array must have a size (total number of elements) at least as big as an associated explicit-shape or assumed-size dummy array (a condition that will automatically be satisfied by an assumed-size array).

Array element sequence association treats the associated actual and dummy arrays each as a *linear sequence of array elements*, with the corresponding elements of each sequence being associated; thus the nth element of the actual array sequence is associated with the nth element of the dummy array sequence. The order of such a sequence is Fortran's *array*

element order, which varies the first subscript first, the second subscript next, and so on. Thus, for a two-dimensional array, array element order is the first column followed by the second column, and so on; for the array declared as **real :: x(3,2)**, array element order is x(1,1), x(2,1), x(3,1), x(1,2), x(2,2), x(3,2).

Array element sequence association means, essentially, that the ranks of the actual and dummy arguments are immaterial, and therefore the ranks are not required to match in this form of argument association. If the actual argument is an array section (and the dummy argument is not assumed shape) the processor must generate an array element sequence for array element sequence association with the dummy argument; this may involve copying the array section into a contiguous area of memory before association with the dummy array, and then refreshing the original array section from this copy after execution of the procedure call is completed (copy-in, copy-out).

If character arrays are involved in array element sequence association there is one additional wrinkle - the sequences generated are character sequences, not array element sequences, and the association is character by corresponding character. If the character length of the actual and dummy array elements are the same (as they would be with assumed length) then this character association is equivalent to array element association. With array element sequence association both the array rank and character length can be "changed" across procedure boundaries, with the indicated consequences for argument association.

Array element sequence association is permitted in any procedure, by declaring the dummy array as an explicit-shape or assumed-size array, and is the only option for procedures with implicit interfaces - that is external procedures without interface blocks. In those cases where the interface is explicit (all module and internal procedures, and external procedures with interface blocks) both array element sequence association and assumed-shape array association are permitted (and normally assumed-shape array association is preferred).

Structure (derived type) argument association follows the same rules as for other types - namely, the actual and dummy arguments must be the same type. This means they must be derived from the same type definition. Another form of structure association, called structure sequence association, involves sequence structures (R422-423). In this case the "same type" rule for the actual and dummy arguments is relaxed in favor of *equivalent types*. Equivalent structure types are sequence types with the same type name and with components that are all public and agree in order, name, and type (or equivalent type).

A caveat about procedure arguments, regardless of the association mechanism, is that if the same entity becomes associated with two (or more) dummy arguments then, in order to prevent nondeterminism in results, the procedure must treat both dummy arguments as *intent(in)*. There are two common instances of this situation. One is when an actual argument is an array section with at least one vector subscript; in this case two or more of the actual array elements may be the same element of the section's parent array and will be associated with different dummy array elements. The second case is when the same data entity is used two (or more) times in an actual argument list: e.g., **call s3(x,y,x)**.

For procedures with explicit interfaces, the actual argument list may be keyworded (R1211-1212). When the interface is explicit, the dummy argument names are known to the procedure's caller. This information allows the actual arguments to be placed in any order, if the dummy argument names are used as *argument keywords* in the call. For example, in this call to subroutine **s4** (which has dummy arguments **d1**, **d2**, and **d3**, in that order)

```
call s4(d2=x, d3=y, d1=z)
```

the actual arguments are not in the order of their associated dummy arguments, but because the interface is explicit the processor can create the intended associations.

A dummy argument may be declared as *optional* (R503, R520). This effectively creates overloaded (generic) versions of the procedure, and calls to such procedures are resolved in accordance with the generic reference resolution rules (see below). The **present** intrinsic function is available for use within the procedure to determine if in a given call to the procedure the optional dummy argument has an associated actual argument or not. Note that either optional arguments must be the last ones in the dummy argument list or calls that omit actual arguments must be keyworded. In any event, optional arguments require explicit interfaces.

Dummy arguments are references to (aliases for) their associated actual arguments. One consequence of this is that any change to a dummy argument is reflected in the actual argument. The *intent* may be specified for a dummy argument to control such "unbridled" access in some respects. The intent attribute (R503, R511, R519) may limit the use of the dummy argument to **in**, **out**, or **inout**. Intent(in) arguments are intended for input to the procedure; the actual argument must be defined upon entrance to the procedure - it may be a variable or an expression - and the dummy argument must not be defined in the procedure. Intent(out) arguments are intended for procedure output - the dummy argument must be defined at some point during execution of the procedure (thus the actual argument must be a variable); the associated actual argument need not be defined upon entrance, so an intent(out) dummy argument must not be referenced in the procedure before it is defined. Intent(inout) specifies that the actual argument must be defined upon entrance and the dummy argument may be defined in the procedure; the associated actual argument must be a variable.

Procedures may be passed through argument lists. That is, a dummy argument may be declared as a procedure name (in an external statement, intrinsic statement, or in an interface block) or used as a procedure name (in a call statement or in a function reference). The associated actual argument must be the name (without an argument list) of a specific procedure (generic procedures cannot be passed). If the dummy argument is declared or used as a function name, the associated actual argument must be the name of a function; if the dummy argument is declared or used as a subroutine name, the actual argument must be the name of a subroutine. Except for internal procedures and statement functions, which cannot be used as actual arguments, any specific procedure, including any specific intrinsic function, may be used as an actual argument.

interface blocks

A procedure interface comprises the information needed to use that procedure correctly; explicit interfaces make this information available to the calling environment. *Interface blocks are used to provide various explicit interfaces*. Explicit interfaces include dummy argument list characteristics, alternate names for a procedure (primarily used to define procedure overloads - that is, generic procedures), and new operator and assignment definitions.

Interface blocks are not needed to make module and internal procedure interfaces explicit, as these interfaces are automatically explicit wherever such procedures are accessible. However, external procedure interfaces are not automatically explicit; interface blocks (R1201) with one or more interface bodies (R1204) may be used to make them explicit:

```
interface
  interface-body
  [ interface-body ] ...
end interface
```

Each interface body specifies an external (or dummy) procedure name, its type (if it is a function), and the order names, and types (and kinds) of all dummy arguments.

If a function has certain properties it may be given an *operator interface*, thereby creating a *defined operator*, and called using operator notation; it must have one (unary operator) or two (binary operator) intent(in) arguments. Such a function may be called with either the normal function syntax or infix operation format; in the latter form the first actual argument appears as the first operation operand and the second actual argument is the second operand (for unary operators the operand follows the operator). The operator form of the interface block is used to define a new operator and associate it with one (or more) functions:

```
interface operator ( defined-operator )  ! suppose this defines an overload of "+"; then
 [ interface-body ] ...                  ! sum_char_int(c,i)    the function form
 [ module-procedure-stmt ] ...           ! c + i                the operator form
end interface
```

An example of using an operator interface to define an overload of the **+** operator is:

```
interface operator ( + )
  integer function sum_char_int(c, i)   ! if code is a character variable and n is integer
    character :: c                      ! sum_char_int can be called in two ways:
    integer :: i                        !    sum_char_int(code,n) function form
  end function                          !    code +  n            operator form
end interface
```

A defined operator can be either a user-defined dot operator or (an overload of) an intrinsic operator (R311), as in the preceding example. If it is an intrinsic operator it must not redefine an intrinsic operation - for example, the **+** operator must not be given an operator interface for a function with two integer arguments, as that would be an attempt to redefine addition of two integer values. All operator definitions are considered to be generic proce-

dure definitions and must be consistent with the generic reference resolution rules (see below). The function(s) associated with a defined operator may be either external functions (in which case the interface contains the corresponding interface bodies, as in the above example) or accessible module functions (in which case the interface contains the corresponding module procedure statements).

If a subroutine has two arguments, the first being intent(out) or intent(inout) and the second being intent(in), then it may be given an *assignment overload*:

```
interface assignment ( = )
 [ interface-body ]...
 [ module-procedure-stmt ]...
end interface
```

An example is:

```
interface assignment ( = )
  subroutine to_char_from_int(c, i)     ! if code is a character variable and n is integer
    character :: c                      ! to_char_from_int can be called in two ways:
    integer :: i                        !  call to_char_from_int(code,n)subroutine form
  end subroutine                        !  code =  n            assignment form
end interface
```

The purpose of such a subroutine is to convert the value of the second argument to an appropriate value for the type of the first argument, and to assign this converted value to the first actual argument; the subroutine defines the conversion that takes place in such an assignment. The *assignment interface* makes it possible to use assignment syntax for this operation, as an alternative to using normal subroutine calls. In analogy with operator interfaces, assignment interfaces define assignment overloads and thus must be consistent with the generic reference resolution rules. Intrinsic assignments cannot be redefined except for intrinsic assignment of structures (that is, derived type intrinsic assignment). As with operator functions, assignment subroutines may be either external subroutines or module subroutines.

Interface blocks may be used to define overloaded (generic) procedure names. Any procedure name may be (further) overloaded, including an intrinsic procedure name:

```
interface generic-name
 [ interface-body ] ...
 [ module-procedure-stmt ] ...
end interface
```

A *generic name* may be associated with any number of external procedures and module procedures. Such a procedure may be called using either its original (specific) name or the generic name. A call to a procedure using the generic name is considered to be a generic reference; any generic reference must be resolvable to a specific procedure, in accordance with the generic reference resolution rules. Generally speaking, this means that each procedure sharing the same generic name must have a different argument “signature” (type pattern). External and module procedures may be given generic interfaces.

generic procedures

A generic procedure is one that can be called in more than one way. These include procedures with generic as well as specific names, functions with operator interfaces (function reference and operator form), and subroutines with assignment interfaces (subroutine call and assignment syntax). The only restriction on the proliferation of generic procedures is that each reference be resolvable to the appropriate underlying specific procedure.

There are two rules which allow generic procedure references to be resolved to a unique specific procedure. the first of these rules (rule (a) below) derives from the positional significance of an argument when keyworded calls are not involved; the second (rule (b)) imposes a further restriction in order to disambiguate keyworded calls. If two procedures may be called with the same generic name (or with the same operator or assignment syntax), one of the argument lists must have a nonoptional dummy argument that (a) *has a type/kind/rank pattern different from that of the dummy argument (if one) in the same position in the other argument list* and (b) *has a name/type/kind/rank pattern different from that of all the dummy arguments in the other argument list*. That is, there must be at least one argument that disambiguates two generic references on the basis of it type/kind/rank signature (and dummy argument name also, in the case of keyworded calls).

Most of Fortran's intrinsic procedures are generic, and references to these intrinsics are resolved in the same manner as described above. Intrinsic functions have an additional generic form: many of them are *elemental*. An elemental function is one defined with a scalar dummy argument and a scalar result. It may, however, be called with an array actual argument, and in this case delivers an array result with the same shape; the value of each result element is the same as if the function had been called with the corresponding actual argument array element.

return statement

The *return* statement (R1224) is a separate statement that causes return from a procedure, in the same way as the procedure's **end** statement. **return** can be used anywhere in the execution part, but is needed only in exceptional cases. See chapter 5 for alternate returns.

statement functions

A *statement function* is a "one-liner" function (R207, R1226), with scalar arguments and a scalar result, for use only in the program unit in which it is defined. Statement functions are not internal procedures, their interfaces are always implicit, they may not be used as actual arguments, and they employ only intrinsic operations. Statement function calls have the normal function call syntax and argument association rules. (See also chapter 5.)

entry statements

The *entry* statement (R1223) can be used to provide alternate entry points into a procedure; the entry statement has a name and an argument list, similar to the function or subroutine statement, and can be placed anywhere in a function or subroutine definition. The original purpose of the entry statement was for data -sharing among different procedures, a functionality now better provided by internal and module procedures.

intrinsic procedures

The rest of this chapter is devoted to summarizing and categorizing Fortran 90's 113 intrinsic procedures (108 intrinsic functions and 5 intrinsic subroutines). This summary has nine categories of procedure, each with certain similar characteristics, and ends with a concise alphabetical listing of all 113 intrinsic procedures and their arguments. Each intrinsic procedure is described more fully, in alphabetical order, in the next chapter.

numeric inquiry functions

digits	significant digits (e.g., bits) for a given integer or real kind
epsilon	a small value (small compared to 1) for a given real kind
exponent	the exponent value for a given real value
fraction	the fractional part of a given real value
huge	the largest value representable for a given real or integer kind
minexponent	the minimum exponent value for a given real kind
maxexponent	the maximum exponent value for a given real kind
nearest	the processor value nearest to a given real value, in a given direction
precision	the decimal precision of a given real or complex kind
radix	numeric base (typically binary) for a given real or integer kind
rrspacing	reciprocal of the relative spacing near a given real value
range	the decimal exponent range of a given numeric kind
scale	change the exponent of a given real value by a specified amount
set_exponent	set the exponent of a given real value to the specified amount
spacing	the absolute spacing near a given real value
tiny	the smallest positive value representable for a given real kind

array inquiry functions

allocated	true if the given array is currently allocated (false otherwise)
lbound	lower bound(s) of a given array or a given dimension of an array
shape	the number of elements in each dimension of a given array
size	the size (total number of elements) of a given array
ubound	upper bound(s) of a given array or a given dimension of an array

miscellaneous inquiry functions

associated	true if the given pointer is currently allocated (false otherwise)
bit_size	the number of bits (for bit computations) in a given integer kind
kind	the value of the kind type parameter of a given data entity
len	the number of characters in a given string value
present	true if there is an actual argument for a given optional dummy argument
selected_int_kind	the integer kind for a given integer decimal range
selected_real_kind	the real kind for a given decimal precision and range

conversion functions

achar	the character in the specified position of the ASCII character set
aimag	the imaginary part of a given complex value
aint	a given real value truncated to an integer (result is still real)

anint	a given real value rounded to the nearest integer (result is still real)
char	the character in the specified position of the processor character set
cmplx	the complex value of a given single or pair of integer or real values
conjg	the complex conjugate of a given complex value
dble	the double precision value of a given numeric value of any type
iachar	position of the specified character in the ASCII character set
ibits	the specified substring of bits of a given integer value
ichar	position of the specified character in the processor character set
int	the (truncated) integer value of a given numeric value of any type
logical	the logical value of specified kind for a given logical value
nint	the (rounded) integer value of a given real value
real	the real value of a given numeric value of any type and kind
transfer	conversion to a specified type without change in the "bit pattern"

numeric computation functions

abs	the absolute value of a given numeric value of any type
acos	the arc cosine (radians) of a given real value
asin	the arc sine (radians) of a given real value
atan	the arc tangent (radians) of a given real value
atan2	the angle (radians) of given real and imaginary components
ceiling	the smallest integer not less than a given real value
cos	the cosine of a given real or complex value (given value in radians)
cosh	the hyperbolic cosine of a given real value
dim	maximum of: zero and the difference of two real or integer values
dot_product	the dot product of two given vectors of numeric or logical type
dprod	the double precision product of two single precision real values
exp	the natural exponential function (real or complex)
floor	the greatest integer not greater than a given real value
log	the natural logarithm function (real or complex)
log10	the logarithm to the base 10 of a given real value greater than zero
matmul	matrix multiplication of two given numeric or logical matrices
max	the maximum of a set of given integer or real values
min	the minimum of a set of given integer or real values
mod	the remainder function, having the sign of the first given value
modulo	the remainder function, having the sign of the second given value
sign	apply a given sign to a given integer or real value
sin	the sine of a given real or complex value (given value in radians)
sinh	the hyperbolic sine of a given real value
sqrt	the square root of a given real or complex value greater than zero
tan	the tangent of a given real value (given value in radians)
tanh	the hyperbolic tangent of a given real value

character computation functions

adjustl	left-justify a given string value in the same-width field
adjustr	right-justify a given string value in the same-width field
index	find the location of a given substring in a given string value

len_trim	length of a given string after trailing blanks have been removed
lge	greater than or equal to ASCII comparison of two given strings
lgt	greater than ASCII comparison of two given string values
lle	less than or equal to ASCII comparison of two given string values
llt	less than ASCII comparison of two given string values
repeat	concatenate several copies of a given string value
scan	search a given string value for any of a given set of characters
trim	remove trailing blank characters from a given string value
verify	position of a character in a string that is not one of a given set

bit computation functions

btest	the bit value of a specified position in a given integer value
iand	bit-by-bit AND of two given integer values
ibclr	set to zero the bit in a specified position in a given integer value
ibset	set the bit in a specified position to the specified (0 or 1) value
ieor	bit-by-bit exclusive-OR of two given integer values
ior	bit-by-bit OR of two given integer values
ishft	end-off shift of the bits in a specified integer value
ishftc	circular shift of the bits in a specified integer value
not	bit-by-bit complement of a given integer value

array computation functions

all	true if all of the elements of a given logical array are true
any	true if any of the elements of a given logical array are true
count	the number of true elements in a given logical array
cshift	circular shift the elements of a given array of any type
eoshift	end-off shift the elements of a given array of any type
maxloc	a rank-one array locating the maximum element of a given array
maxval	the maximum element value of a given integer or real array
merge	combines (merges) two arrays under control of a mask
minloc	a rank-one array locating the minimum element of a given array
minval	the minimum element value of a given integer or real array
pack	packs elements of an array into a vector, under control of a mask
product	the product of all elements of a given numeric array of any type
reshape	reshapes a given rank-one array into the specified array shape
spread	replicates an array along a new dimension
sum	the sum of all elements of a given numeric array of any type
transpose	the matrix transpose of a given rank-two array of any type
unpack	unpacks a vector into elements of an array, under control of a mask

intrinsic subroutines

date_and_time	returns date and time information in several formats
mvbits	copies a sequence of bits between given integer values
random_number	returns one or more pseudorandom numbers
random_seed	allows setting of the random number generator seed value
system_clock	returns various data from the processor's real-time clock

alphabetical listing of intrinsic procedures

generic procedure names, with argument (keyword) names	optional arguments	specific names, arguments	specific argument types
abs (a)		abs (a) cabs (a) dabs (a) iabs (a)	default real default complex double precision real default integer
achar (i)			
acos (x)		acos (x) dacos (x)	default real double precision real
adjustl (string)			
adjustr (string)			
aimag (z)		aimag (z)	default complex
aint (a, kind)	kind	aint (a) dint (a)	default real double precision real
all (mask, dim)	dim		
allocated (array)			
anint (a, kind)	kind	anint (a) dnint (a)	default real double precision real
any (mask, dim)	dim		
asin (x)		asin (x) dsin (x)	default real double precision real
associated (pointer, target)	target		
atan (x)		atan (a) dtan (a)	default real double precision real
atan2 (y, x)		atan2 (a) dtan2 (a)	default real double precision real
bit_size (i)			
btest (i, pos)			
ceiling (a, kind)	kind		
char (i, kind)	kind		
cmplx (x, y, kind)	y, kind		
conjg (z)		conjg (x)	default complex
cos (x)		cos (x) ccos (x) dcos (x)	default real default complex double precision real
cosh (x)		cosh (x) dcosh (x)	default real double precision real
count (mask, dim)	dim		
cshift (array, shift, dim)	dim		
date_and_time (date, time, zone, values)	date, time, zone, values		
dble (a)			
digits (x)			
dim (x, y)		dim (x,y) idim (x,y)	default real default integer
dot_product (vector_a, vector_b)			
dprod (x, y)			
eoshift (array, shift, boundary, dim)	boundary, dim		
epsilon (x)			
exp (x)			

generic procedure names, with argument (keyword) names	optional arguments	specific names, arguments	specific argument types
exponent (x)			
floor (a, kind)	kind		
fraction (x)			
huge (x)			
iachar (c)			
iand (i, j)			
ibclr (i, pos)			
ibits (i, pos, len)			
ibset (i, pos)			
ichar (c)			
ieor (i, j)			
index (string, substring, back)	back	index (string, substring)	default character
int (a, kind)	kind		
ior (i, j)			
ishft (i, shift)			
ishftc (i, shift, size)	size		
kind (x)			
lbound (array, dim)	dim		
len (string)		len (string)	default character
len_trim (string)			
lge (string_a, string_b)			
lgt (string_a, string_b)			
lle (string_a, string_b)			
llt (string_a, string_b)			
log (x)		alog (x) clog (x) dlog (x)	default real default complex double precision real
log10 (x)		alog10 (x) dlog10 (x)	default real double precision real
logical (l, kind)	kind		
matmul (matrix_a, matrix_b)			
max (a1, a2, a3, ...)	a3, ...		
maxexponent (x)			
maxloc (array, dim, mask)	dim, mask		
maxval (array, dim, mask)	dim, mask		
merge (tsource, fsource, mask)			
min (a1, a2, a3, ...)	a3, ...		
minexponent (x)			
minloc (array, dim, mask)	dim, mask		
minval (array, dim, mask)	dim, mask		
mod (a, p)		mod (a, p) amod (a, p) dmod (a, p)	default integer default real double precision real
modulo (a, p)			
mvbits (from, frompos, len, to, topos)			
nearest (x, s)			
nint (a, kind)	kind	nint (a) idnint (a)	default real double precision real

generic procedure names, with argument (keyword) names	optional arguments	specific names, arguments	specific argument types
not (i)			
pack (array, mask, vector)	vector		
precision (x)			
present (a)			
product (array, dim, mask)	dim, mask		
radix (x)			
random_number (harvest)			
random_seed (size, put, get)	size, put, get		
range (x)			
real (x, kind)	kind		
repeat (string, ncopies)			
reshape (source, shape, pad, order)	pad, order		
rrspacing (x)			
scale (x, i)			
scan (string, set, back)	back		
selected_int_kind (r)			
selected_real_kind (p, r)	p, r		
set_exponent (x, i)			
shape (source)			
sign (a, b)		sign (a, b) dsign (a, b) isign (a, b)	default real double precision real default integer
sin (x)		sin (x) csin (x) dsin (x)	default real default complex double precision real
sinh (x)		sinh (x) dsinh (x)	default real double precision real
size (array, dim)	dim		
spacing (x)			
spread (source, dim, ncopies)			
sqrt (x)		sqrt (x) csqrt (x) dsqrt (x)	default real default complex double precision real
sum (array, dim, mask)	dim, mask		
system_clock (count, count_rate, count_max)	count, count_rate, count_max		
tan (x)		tan (x) dtanh (x)	default real double precision real
tanh (x)		tanh (x) dtanh (x)	default real double precision real
tiny (x)			
transfer (source, mold, size)	size		
transpose (matrix)			
trim (string)			
ubound (array, dim)	dim		
unpack (vector, mask, field)			
verify (string, set, back)	back		

10 Intrinsic Procedures

The 113 intrinsic procedures are introduced and organized in the previous chapter, and each is described in detail in this chapter. A "pseudo" interface block, without the **interface ... end interface** bracketing keywords, describes the interface of each procedure; the semantics is described in comments in the interface, often augmented by text following the interface. Constraints and other relevant information are also included, either in the interface comments or in the following text.

Most of the intrinsic procedures are generic over the various kinds of the argument type - for example, **sqrt** is generic for both single and double precision real arguments, and any other real kinds the implementation might supply. Unless explicitly mentioned otherwise, each single-argument intrinsic procedure is generic in this sense, with the result kind being the same as the argument kind. Similarly, each intrinsic function with one argument plus a **kind** argument is generic in this sense, but with the result kind as specified by the **kind** argument.

As described in the previous chapter, intrinsic procedures may be classified as either elemental or transformational (most are elemental). If an intrinsic procedure is elemental the interface starts with the keyword **elemental**; otherwise that procedure is tranformational. (In a call to an elemental function with two or more arguments, the actual arguments must be conformable; but note that a scalar is conformable with any array.) Similarly, some intrinsic functions are identified as inquiry functions with the keyword **inquiry**; actual arguments to inquiry functions need not be defined.

All intrinsic function arguments are **intent(in)** and so the intent for these arguments is not explicitly given in the interface; however, the argument intent is explicitly specified for each argument of the five intrinsic subroutines. The argument names can be used as actual argument keywords.

Many of the intrinsic procedures take array arguments of any rank. These arguments are shown as rank one **(:)** in the following interfaces, and the descriptions identify which arguments may be generic over rank and the resulting meaning of the different ranks.

abs (a)

```
elemental function abs(a)          ! the |a|
  real :: abs                      ! or integer if a is of type integer
  real :: a                        ! or type integer or type complex
end function
```

If **a** is complex with value (x,y), **abs** returns an approximation to $\sqrt{x^2 + y^2}$.

achar (i)

```
elemental function achar(i)        ! the ith ascii character
  character :: achar               ! the result kind is kind ('a')
  integer :: i
end function
```

Note that achar(iachar(x)) is x for any character x of default kind represented by the processor.

acos (x)

```
elemental function acos(x)         ! the arccosine (inverse cosine)
  real :: acos
  real :: x                        ! |x| ≤ 1
end function
```

The result has a value equal to a processor-dependent approximation to arccos(x), expressed in radians. It lies in the range $0 \le \text{acos}(x) \le \pi$.

adjustl (string)

```
elemental function adjustl(string)          ! remove leading blanks
  character(len(string)) :: adjustl         !    (the same number of trailing blanks added)
  character(*) :: string
end function
```

adjustr (string)

```
elemental function adjustr(string)          ! remove trailing blanks
  character(len(string)) :: adjustr         !    (the same number of leading blanks added)
  character(*) :: string
end function
```

aimag (z)

```
elemental function aimag(z)                 ! imaginary part of z
  real :: aimag                             ! if z = (x, y), aimag is y
  complex :: z
end function
```

aint (a, kind)

```
elemental function aint(a,kind)             ! truncate a
  real(kind) :: aint                        ! if kind is absent the result kind is kind(a)
  real :: a                                 ! may be any kind
  integer, optional :: kind                 ! if present, must be a scalar initialization expression
end function
```

If |a|<1, **aint (a)** has the value 0; if a≥1, **aint (a)** has the sign of **a** and a value equal to the integer whose magnitude is the largest integer that does not exceed the magnitude of **a.**

all (mask, dim)

```
function all(mask,dim)                      ! returns true if all of mask (along dim) is true
  logical :: all                            ! all is an array if dim is present and mask rank > 1
  logical :: mask(:)                        ! may be any kind, any rank
  integer, optional :: dim                  ! if present, 1≤dim≤n, where n is rank of mask
```

The result is scalar if **dim** is absent or **mask** has rank one; otherwise, the result is an array of rank n-1 and of shape $(d_1,d_2,...,d_{dim-1},d_{dim+1},...,d_n)$ where $(d_1,d_2,...,d_n)$ is the shape of **mask**.

allocated (array)

```
inquiry function allocated(array)           ! check allocation status of argument
  logical :: allocated                      ! true if array is currently allocated; false otherwise
  real :: array(:)                          ! array can be any type and any rank
end function
```

The actual argument for **array** must be an allocatable array with defined allocation status.

anint (a, kind)

```
elemental function anint(a,kind)            ! integer value nearest a
  real(kind) :: anint                       ! if kind is absent the result kind is kind(a)
  real :: a
  integer, optional :: kind                 ! if present, must be a scalar initialization expression
end function
```

If a>0 **anint (a)** is **aint (a + 0.5)**; if a≤0 **anint (a)** is **aint (a – 0.5).**

any (mask, dim)

```
function any(mask,dim)                      ! returns true  if any of mask (along dim) is true
  logical :: any                            ! any is an array if dim is present and mask rank > 1
  logical :: mask(:)                        ! may be any kind, any rank
  integer, optional :: dim                  ! if present, 1≤dim≤n, where n is rank of mask
end function
```

The result is scalar if **dim** is absent or **mask** has rank one; otherwise, the result is an array of rank n-1 and of shape $(d_1,d_2,...,d_{dim-1},d_{dim+1},...,d_n)$ where $(d_1,d_2,...,d_n)$ is the shape of **mask**.

asin (x)

```
elemental function asin(x)                ! the arcsine (inverse sine)
 real :: asin
 real :: x                                ! x ≤ 1
end function
```

The result has a value equal to a processor-dependent approximation to arcsin(x), expressed in radians. It lies in the range $-\pi/2 \le \text{asin}(x) \le \pi/2$.

associated (pointer, target)

```
inquiry function associated(pointer,target)   ! check association status of argument
 logical :: associated                        ! true if pointer is currently associated, else false
 real, pointer :: pointer(:)                  ! pointer can be any type, any rank
 real, optional :: target(:)                  ! target can be any type, any rank
end function
```

The actual argument for **pointer** must be a pointer with defined pointer association status. The actual argument for **target**, if present, must be either a target or a pointer with defined pointer association status. If **target** is absent, the result is true if **pointer** is currently associated with a target and false otherwise. If **target** is present and is a target, the result is true if **pointer** is currently associated with **target** and false if it is not. If **target** is present and is a pointer, the result is true if both **pointer** and **target** are currently associated with the same target, and false otherwise.

atan (x)

```
elemental function atan(x)                ! the arctangent (inverse tangent)
 real :: atan
 real :: x
end function
```

The result has a value equal to a processor-dependent approximation to arctan(x), expressed in radians, that lies in the range $-\pi/2 \le \text{atan}(x) \le \pi/2$.

atan2 (y, x)

```
elemental function atan2(x)               ! the arctangent (inverse tangent)
 real :: atan2                            !      of the nonzero complex number (x, y)
 real :: y
 real :: x
end function
```

The result has a value equal to a processor-dependent approximation to the arctan(y/x), expressed in radians that lies in the range $-\pi \le \text{atan2}(y,x) \le \pi$. If y>0, the result is positive. If y=0, the result is zero if x>0 and π if x<0. If y<0, the result is negative. If x=0, the absolute value of the result is $\pi/2$.

bit_size (i)

```
inquiry function bit_size(i)              ! number of bits in argument i
 integer :: bit_size
 integer :: i                             ! i may be an array of any rank
end function                              ! (but note extension in chapter 12)
```

btest (i, pos)

```
elemental function btest(i,pos)           ! returns true if the bit in position pos of i is1
 logical :: btest
 integer :: i
 integer :: pos                           ! 0≤pos<bit_size(i)
end function
```

ceiling (a)

```
elemental function ceiling(a)          ! least integer greater than or equal to a
  integer :: ceiling
  real :: a
end function
```

char (i, kind)

```
elemental function char(i,kind)        ! character in the ith position of the character set
  character(kind) :: char              ! default character kind if kind is absent
  integer :: I                         ! in range 0 ≤ i ≤ n-1 where n is # of characters
  integer, optional :: kind            ! if present, must be a scalar initialization expression
end function
```

Note that ichar(char(y,kind(x))) = y and char(ichar(x),kind(x)) = x.

cmplx (x, y, kind)

```
elemental function cmplx(x,y,kind)     ! complex number with real part x, imaginary part y
  complex(kind) :: cmplx               ! or default kind if kind is absent
  real :: x                            ! or integer or complex
  real, optional :: y                  ! or integer, or absent if x is complex
  integer, optional :: kind            ! if present, must be a scalar initialization expression
end function
```

If y is absent and x is not complex, then the imaginary part of the result is zero.

conjg (z)

```
elemental function conjg(z)            ! the conjugate of a complex number
  complex :: conjg
  complex :: z
end function
```

cos (x)

```
elemental function cos(x)              ! the cosine of x
  real :: cos                          ! same type and kind as x
  real :: x                            ! may be complex
end function
```

If **x** is of type real, it is regarded as a value in radians; if **x** is of type complex, its real part is regarded as a value in radians.

cosh (x)

```
elemental function cosh(x)             ! the hyperbolic cosine of x
  real :: cosh
  real :: x
end function
```

count (mask, dim)

```
function count(mask,dim)               ! count the number of true elements of mask
  integer :: count                     ! count is an array if dim is present and mask rank > 1
  logical :: mask(:)                   ! may be any kind, any rank
  integer, optional :: dim             ! if present, 1≤dim≤n, where n is rank of mask
end function
```

If **dim** is present and n is greater than 1 then the result is an array of rank n-1. For example, if **mask** is a 3x2 array and **dim** is 2, then the result is a one-dimensional array of size 3, with the count taking place along each row of **mask**.

cshift (array, shift, dim)

```
function cshift(array,shift,dim)          ! circularly shift array
 real :: cshift(:)                        ! same type, kind, and shape as array
 real :: array(:)                         ! or any type, kind, and rank (not scalar)
 integer :: shift                         ! amount to be shifted, positive for left shifts
 integer, optional :: dim                 ! if present, 1≤dim≤n, where n is rank of array
end function
```

Positive **shift** amounts are “left” shifts (e.g., cshift(i)=array(i+1)) and negative shifts are “right”; values shifted “off” one end are routed into the other end. If **dim** is absent it is assumed to be 1. If **array** has rank greater than 1 then n-1 “one-dimensional shifts” take place along the dimension specified by **dim**. For example, if **array** is a 3x2 array and **dim**=1, each of the two columns of **array** are shifted an amount specified by **shift**. **shift** is allowed to be an array of rank n-1, specifying a different shift amount for each “one-dimensional shift”. In the preceding example, **shift** could be a one-dimensional array of two elements, having values, say, of 2 and -1; in this case the first column of the array will be circularly shifted “up” two and the second column will be shifted “down” one.

date_and_time (date, time, zone, values)

```
subroutine date_and_time(date,time,zone,values)   ! returns date and time information
 character(*), optional, intent(out)) :: date     ! date in CCYYMODD format
 character(*), optional, intent(out)) :: time     ! time in HHMMSS.SSS format
 character(*), optional, intent(out)) :: zone     ! zone in ±HHMM format
 integer, optional, intent(out)) :: values(:)     ! date, time, and zone in integer form
end subroutine
```

Returned results are compatible with the representations defined in ISO 8601:1988. CC is the century, YY the year of the century, MO the month, DD the day of the month, HH the hour of the day, MM the minutes of the hour , and SS.SSS the seconds/milliseconds. For **zone**, the result is the hours and minutes from Coordinated Universal Time. For **values**, **values(1)** is the integer form of CCYY - e.g., the year, **values(2)** is MO, **values(3)** is DD, **values (4)** is the zone, **values(5)** is HH (range 0:23), **values(6)** is MM (range 0:59), **values(7)** is the seconds (range 0:60), and **values(8)** is the milliseconds.

dble (a)

```
elemental function dble(a)                ! converts a to a double precision real value
 real(DOUBLE) :: dble                     ! where DOUBLE is the double-precision kind value
 real :: a                                ! or integer or complex, of any kind
end function
```

If **a** is complex, then **dble** returns the real part of **a** in double precision form.

digits (x)

```
inquiry function digits(x)                ! returns the model value of n if x is integer,
  integer :: digits                       !                             or q if x is real (see chapter 2)
  integer :: x                            ! may also be real and/or an array
end function
```

dim (x, y)

```
elemental function dim(x,y)               ! returns max(0,x-y)
  integer :: din                          ! same type and kind as x (and y)
  integer :: x, y                         ! may also be real; y must have same type and kind as x
end function
```

dot_product (vector_a, vector_b)

```
function dot_product(vector_a,vector_b)  ! the dot-product multiplication of numeric or logical vectors
  real :: dot_product                    ! may also be logical or integer - see discussion
  real :: vector_a(:), vector_b(:)       ! one-dimensional arrays of the same size and both either
end function                             !      of type logical or of any numeric type
```

If the vectors are size zero, the result value is either zero or false; otherwise, if the vectors are of type logical the result is type logical with value and kind of **any(vector_a.and.vector_b),** if **vector_a** is of type integer or real the result value and kind are those of **sum(vector_a*vector_b),** and else the result value and kind are are those of **sum(conjg(vector_a)*vector_b)**.

dprod (x, y)

```
elemental function dprod(x,y)            ! returns double-precision product of two default real values
  real(DOUBLE) :: dprod                  ! where DOUBLE is the double-precision kind value
  real :: x, y
end function
```

eoshift (array, shift, boundary, dim)

```
function eoshift(array,shift,boundary,dim)  ! end-off shift off array
 real :: eoshift(:)                         ! same type, kind, and shape as array
 real :: array(:)                           ! or any type, kind, and rank (not scalar)
 integer :: shift                           ! amount to be shifted, positive for left shifts
 integer, optional :: boundary              ! same type and kind as array - value for vacated positions
 integer, optional :: dim                   ! if present, 1≤dim≤n, where n is rank of array
end function
```

eoshift is exactly the same as **cshift**, except that values shifted "off" one end are not routed into the vacated positions on the other end; the **boundary** value is placed in the vacated positions. If **boundary** is omitted, the default value is 0, 0.0, (0.0,0.0), false, or blanks, depending on whether **array** is type integer, real, complex, logical, or character, respectively. **boundary** is allowed to be an array of rank n-1, specifying a different fill value amount for each vacated position. For example, if **m** is the character array $\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$, then **eoshift(m,shift=1,boundary='*',dim=2)** is $\begin{bmatrix} * & A & B \\ * & D & E \\ * & G & H \end{bmatrix}$, and **eoshift(m,shift=(/–1,1,0/),boundary=(/'*','/','!'/),dim=2)** is $\begin{bmatrix} * & A & B \\ E & F & / \\ G & H & I \end{bmatrix}$.

epsilon (x)

```
inquiry function epsilon(x)              ! a positive value almost negligible compared to unity
  real :: epsilon                        ! same type and kind as x
  real :: x                              ! may be any kind; may be an array of any rank
end function
```

exp (x)

```
elemental function exp(x)                ! an approximation to e^x
  real :: exp                            ! type and kind of x
  real :: x                              ! may be complex, in which case its imaginary part
end function                             !      is regarded as a value in radians
```

exponent (x)

```
elemental function exponent(x)           ! the real model exponent part of x - see chapter 2
  integer :: exponent
  real :: x
end function
```

floor (a)

```
elemental function floor(a)          ! greatest integer less than or equal to a
  integer :: floor
  real :: a
end function
```

fraction (x)

```
elemental function fraction(x)       ! the real model fractional part of x - see chapter 2
  real :: fraction
  real :: x
end function
```

huge (x)

```
inquiry function huge(x)             ! the largest value for the type and kind of x
  real :: huge                       ! same type and kind as x
  real :: x                          ! may be any kind; may be integer; may be an array
end function
```

iachar (c)

```
elemental function iachar(c)         ! same as ichar, except the ascii collating sequence
  integer :: iachar                  !      (ISO 646:1983) is used instead of the processor's
  character :: c
end function
```

iand (i, j)

```
elemental function iand(i,j)         ! perform logical and on bits of i and j
  integer :: iand                    ! same kind as i (and j)
  integer :: i, j                    ! any kind, but both must be the same kind
end function
```

The result is the bit-by-corresponding-bit *and* of the arguments; if both argument bits are 1 the corresponding result bit is 1, otherwise the result bit is 0.

ibclr (i, pos)

```
elemental function ibclr(i,pos)
  integer :: ibclr                   ! same as i, but with the specified bit set to 0
  integer :: i                       ! may be any kind
  integer :: pos                     ! position of bit in i to set to zero; 0≤pos<bit_size(i)
end function
```

ibits (i, pos, len)

```
elemental function ibits(i,pos,len)  ! "extract" a string of bits from i
  integer :: ibits                   ! same type and kind as i
  integer :: i                       ! may be any kind
  integer :: pos                     ! position of first bit to extract; pos≥0
  integer :: len                     ! number of bits to extract; len≥0 and pos+len < bit_size(i)
end function
```

The result has the sequence of **len** bits from **i**, beginning at bit **pos**; these bits are in the rightmost bit positions of the result, with all other bits zero.

ibset (i, pos)

```
elemental function ibset(i,pos)      ! same as ibclr, except the specified bit is set to 1 instead of 0
  integer :: ibset
  integer :: i
  integer :: pos
end function
```

ichar (c)

```
elemental function ichar(c)          ! position of a character in the processor collating sequence
  integer :: ichar
  character :: c
end function                         ! same as iachar if the processor collating sequence is ascii
```

ieor (i, j)

```
elemental function ieor(i,j)         ! perform logical exclusive-or on bits of i and j
  integer :: ieor                    ! same kind as i (and j)
  integer :: i, j                    ! any kind, but both must be the same kind
end function
```

The result is the bit-by-corresponding-bit *exclusive-or* of the arguments; if one of the argument bits is 1 and the other is 0, the corresponding result bit is 1, otherwise the result bit is 0.

index (string, substring, back)

```
elemental function index(string,substring,back)   ! search string for an substring of substring
  integer :: index                                ! beginning position of substring in string (or zero)
  character(*) :: string                          ! may be any kind
  character(*) :: substring                       ! same kind as string
  logical, optional :: back                       ! if present and true, search is from back of string
end function
```

If **back** is absent or present with the value false, the result is the minimum positive value of **i** such that **string(i:i+len(substring)–1) == substring** or zero if there is no such value; if **back** is present with the value true, the result is the maximum value of **i** less than or equal to **len(string)–len(substring)+1** such that **string(i:i+len(substring)–1)==substring** or zero if there is no such value. Zero is returned if **len(string)<len(substring)** and one is returned if **len(substring)==0**.

int (a, kind)

```
elemental function int(a,kind)       ! convert a to an integer value (of specified kind)
  integer(kind) :: int               ! the converted integer value
  real :: a                          ! may be any numeric type
  integer, optional :: kind          ! if present, must be a scalar initialization expression
end function                         ! if kind is absent, the result kind is default integer
```

If **a** is of type integer, the result is this same value, but possibly of a different kind. If **a** is of type real, the real value is truncated toward zero (e.g., **int(3.7)** is 3 and **int(-3.7)** is -3). If **a** is of type complex, the result is **int(real(a))**.

ior (i, j)

```
elemental function ior(i,j)          ! perform logical inclusive-or on bits of i and j
  integer :: ior                     ! same kind as i (and j)
  integer :: i, j                    ! any kind, but both must be the same kind
end function
```

The result is the bit-by-corresponding-bit *inclusive-or* of the arguments; if either one, or both, of the argument bits is 1, the corresponding result bit is 1, otherwise the result bit is 0.

ishft (i, shift)

```
elemental function ishift(i,shift)   ! logically end-off shift the bits of i the amount shift
  integer :: ishft                   ! same kind as i
  integer :: i                       ! may be any kind
  integer :: shift                   ! amount to shift the bit pattern of i; shift must have a
end function                         !      magnitude less than or equal to bit_size(i)
```

The result is the value obtained by shifting the bits of **i** by **shift** positions. If **shift** is positive, the shift is to the left. Bits shifted out are lost; zeros are shifted in at the opposite end.

ishftc (i, shift, size)

```
elemental function ishift(i,shift, size)    ! same as ishft, but the shift is circular rather than end-off
  integer :: ishft                          ! same kind as i
  integer :: i                              ! may be any kind
  integer :: shift                          ! amount to shift the bit pattern of i; shift ≤ size
  integer, optional :: size                 ! 0 ≤ size < bit_size(i)
end function                                !     if size is omitted, bit_size(i) is assumed
```

The result has the value obtained by shifting the bits of **i** by **shift** positions. If **shift** is positive, the shift is to the left. Bits shifted out are shifted into at vacated positions at the opposite end.

kind (x)

```
inquiry function kind(x)                    ! the kind type parameter value of any object
  integer :: kind                           ! the kind value of x
  real :: x                                 ! may be any intrinsic type and any kind; may be an array
end function
```

lbound (array, dim)

```
inquiry function lbound(array,dim)          ! the lower bound(s) of array
  integer :: lbound                         ! scalar if dim is present; a rank one array otherwise
  real :: array(:)                          ! may have any type, kind, and rank
  integer, optional :: dim                  ! if present, 1 ≤ dim ≤ n, where n is the rank of array
end function
```

If **dim** is present the result is a scalar and is the lower bound of **array** along the dimension **dim**. If **dim** is absent the result is a one-dimensional array whose size is the rank of **array**, and the value of each element of the result is the lower bound of that dimension of **array**. If **array** is an array expression other than an array name (e.g., an array section), the lower bound for each dimension is 1.

len (string)

```
inquiry function len(string)                ! the length (numbers of characters) of a string
  integer :: len                            ! the length of string
  character(*) :: string                    ! may be any kind; may be an array
end function                                ! if string is an array, len is the length of each element
```

len_trim (string)

```
elemental function len_trim(string)         ! same as len(trim(string))
  integer :: len_trim                       ! the length of string with all trailing blanks removed
  character(*) :: string
end function
```

lge (string_a, string_b)

```
elemental function lge(string_a,string_b)   ! string comparison, based on ascii
  logical :: lge                            ! true if string_a >= string_b in the ascii collating sequence,
  character(*) :: string_a, string_b        !                              false otherwise
end function
```

lgt (string_a, string_b)

```
elemental function lgt(string_a,string_b)   ! string comparison, based on ascii
  logical :: lgt                            ! true if string_a > string_b in the ascii collating sequence,
  character(*) :: string_a, string_b        !                             false otherwise
end function
```

lle (string_a, string_b)

```
elemental function lle(string_a,string_b)   ! string comparison, based on ascii
  logical :: lle                            ! true if string_a <= string_b in the ascii collating sequence,
  character(*) :: string_a, string_b        !                              false otherwise
end function
```

llt (string_a, string_b)

```
elemental function llt(string_a,string_b)   ! string comparison, based on ascii
  logical :: llt                            ! true if string_a < string_b in the ascii collating sequence,
  character(*) :: string_a, string_b        !                             false otherwise
end function
```

log (x)

```
elemental function log(x)                   ! the natural logarithm, log_e x
  real :: log                               ! same type and kind as x
  real :: x                                 ! may be complex; if real, x > 0
end function                                !                  if complex, x must not be zero
```

log10 (x)

```
elemental function log10(x)                 ! base-10 logarithm, log_10 x
  real :: log10                             ! same kind as x
  real :: x                                 ! x > 0
end function
```

logical (l, kind)

```
elemental function logical(l,kind)          ! convert between logical kinds
  logical(kind) :: logical                  ! the value of l, but with kind kind
  logical :: l                              ! may be any kind
  integer, optional :: kind                 ! if present, must be a scalar initialization expression
end function                                ! if kind is absent, the result kind is default logical
```

matmul (matrix_a, matrix_b)

```
function matmul(matrix_a,matrix_b)          ! matrix multiplication of two numeric or logical matrices
  real :: matmul(:,:)                       ! type and kind determined by the arguments - see below
  real :: matrix_a(:,:), matrix_b(:,:)      ! may be any kind; may be integer, complex or logical;
end function                                ! one of the argument matrices (but not both) may be rank one
```

The two arguments must be both of type logical or of numeric (integer, real, complex) type. The size of the first (or only) dimension of **matrix_b** must equal the size of the last (or only) dimension of **matrix_a**. There are three cases: (1) matrix_a has shape (n,k) and matrix_b has shape (k,m), in which case the result has shape (n,m); (2) matrix_a has shape (k) and matrix_b has shape (k,m), in which case the result has shape (m); (3) matrix_a has shape (n,k) and matrix_b has shape (k), in which case the result has shape (n). For case (1) the (i,j) element of the result has the value and kind of **sum(matrix_a(i,:)*matrix_b(:,j))** if the arguments are of numeric type and **any(matrix_a(i,:).and.matrix_b(:,j))** otherwise. For case (2) the (i) element of the result has the value and kind of **sum(matrix_a(:)*matrix_b(:,i))** if the arguments are of numeric type and **any(matrix_a(:).and.matrix_b(:,j))** otherwise, and for case (3) the (i) element of the result has the value and kind of **sum(matrix_a(i,:)*matrix_b(:))** if the arguments are of numeric type and **any(matrix_a(i,:).and.matrix_b(:))** otherwise.

max (a1, a2, a3, ...)

```
elemental function max(a1,a2,a3,...)        ! return the maximum of the argument values
  real :: max                               ! same type and kind as the arguments
  real :: a1,  a2                           ! may be integer,
  real, optional :: a3, ...                 !      but all arguments must have the same type and kind
end function
```

maxexponent (x)

```
inquiry function maxexponent(x)          ! maximum model exponent that this kind of real can have
  integer :: maxexponent
  real :: x                              ! may be any kind; may be an array
end function
```

maxloc (array, mask)

```
function maxloc(array,mask)              ! location in array of element with the maximum value
  integer :: maxloc(:)                   ! result element values are the subscripts of the location
  real :: array(:)                       ! any kind, any rank; can be type integer
  logical, optional :: mask(size(array)) ! must be same shape as array
end function
```

The size of the result is equal to the rank of **array**. The value of the kth element of the result is the value of the kth subscript of the location of the element with the maximum value. If more than one element of **array** has this maximum value, the location of the first, in array element order, is returned. If **mask** is present, only those locations in **array** corresponding to the true values in **mask** are searched for the maximum value.

maxval (array, dim, mask)

```
function maxval(array,dim,mask)          ! the maximum value in array, or along one of its diimensions
  real :: maxval                         ! same kind and type as array
  real :: array(:)                       ! any kind, any rank; can be type integer
  integer, optional :: dim               ! dimension along which to determine maximum values
  logical, optional :: mask(size(array)) ! must be same shape as array
end function
```

The result is scalar if **dim** is omitted or **array** has rank 1 (as illustrated in the interface), in which case the value returned is the maximum element value in **array**. If array has rank n greater than 1 and **dim** is present, $1 \le dim \le n$ and **dim** specifies the dimension along which to determine the maximum values; in this case the result is an array of rank n-1 and shape $(d_1,d_2,...,d_{dim-1}, d_{dim+1},...,d_n)$ where $(d_1,d_2,...,d_n)$ is the shape of **array** and each value of the result is the maximum value along the **dim** dimension of **array**. If **mask** is present only those elements of **array** corresponding to the true values in **mask** are searched for the maximum value.

merge (tsource, fsource, mask)

```
elemental function merge(tsource,fsource,mask)  ! select one of two arguments, based on a mask
  real :: merge                                 ! same type and kind as tsource
  real :: tsource                               ! may be of any type and kind
  real :: fsource                               ! same type and kind as tsource
  logical :: mask                               ! return tsource if mask is true,
end function                                    !        fsource if mask is false
```

merge is an elemental function and is most often used to merge two arrays, based on the (conformable) **mask**.

min (a1, a2, a3, ...)

```
elemental function min(a1,a2,a3,...)     ! return the minimum of the argument values
  real :: min                            ! same type and kind as the arguments
  real :: a1,  a2                        ! may be integer,
  real, optional :: a3, ...              !      but all arguments must have the same type and kind
end function
```

minexponent (x)

```
inquiry function minexponent(x)          ! minimum model exponent that this kind of real can have
  integer :: minexponent
  real :: x                              ! may be any kind; may be an array
end function
```

minloc (array, mask)

```
function minloc(array,mask)          ! same as maxloc,
  integer :: minloc(:)               !         but with minimum value rather than maximum value
  real :: array(:)
  logical, optional :: mask(size(array))
end function
```

minval (array, dim, mask)

```
function minval(array,dim,mask)      ! same as maxval,
  real :: minval                     !         but with minimum value rather than maximum value
  real :: array(:)
  integer, optional :: dim
  logical, optional :: mask(size(array))
end function
```

mod (a, p)

```
elemental function mod(a,p)          ! the remainder function (has sign of a)
  integer :: mod                     ! same type and kind as a; value is a-int(a/p)*p
  integer :: a                       ! any kind; may be real
  integer :: p                       ! same type and kind as a; the value of p must not be zero
end function                         ! mod and modulo are the same for positive values of a and p
```

modulo (a, p)

```
elemental function modulo(a,p)       ! the modulo function (has sign of p)
  integer :: modulo                  ! same type and kind as a; value is a-floor(a/p)*p
  integer :: a                       ! any kind; may be real
  integer :: p                       ! same type and kind as a; the value of p must not be zero
end function                         ! mod and modulo are the same for positive values of a and p
```

mvbits (from, frompos, len, to, topos)

```
elemental subroutine mvbits(from,frompos,len,to,topos)     ! move bits from from to to
  integer, intent(in) :: from        ! may be any kind
  integer, intent(in) :: frompos     ! 0 ≤ frompos < bit_size(from)
  integer, intent(in) :: len         ! 0 ≤ len
  integer, intent(inout) :: to       ! same kind as from (and may be the same object)
  integer, intent(in) :: topos       ! 0 ≤ topos; (topos+len) < bitsize(to)
end subroutine
```

Copies **len** bits from object **from**, starting at bit position **frompos** in **from**, to object **to**, starting at bit position **topos** in **to**.

nearest (x, s)

```
elemental function nearest(x,s)      ! the nearest value in the specified direction
  real :: nearest                    ! same kind as x; value is that nearest x, but not x
  real :: x                          ! may be any kind
  real :: s                          ! must not be zero; s > 0 means nearest > x
end function                         !                   s < 0 means nearest < x
```

nint (a, kind)

```
elemental function nint(a,kind)      ! integer nearest to a
  integer(kind) :: nint              ! value is int(a-sign(0.5,a))
  real :: a
  integer, optional :: kind          ! if present, must be a scalar initialization expression
end function                         ! if kind is absent, the result kind is default integer
```

not (i)

```
elemental function not(i)          ! logical bit-wise complement
  integer :: not                   ! the bit complement of i
  integer :: i
end function                       ! 1 bits in i become 0 in not; 0 bits in i become 1 in not
```

pack (array, mask, vector)

```
function pack(array,mask,vector)   ! pack an array of any shape into an array of rank 1
  real :: pack(:)                  ! same type and kind as array
  real :: array(:)                 ! maybe any type, kind, and shape
  logical :: mask(size(array))     ! same shape as array
  real, optional :: vector(:)      ! if present, must have at least count(mask) elements
end function
```

If **vector** is present the size of **pack** is the size of **vector**; otherwise the size of **pack** is **count(mask)**. The elements of **array** that correspond to true values in **mask** are placed in **pack**, starting with the first element of **pack** and in array element order from **array.**

precision (x)

```
inquiry function precision(x)      ! the decimal precision of x
  integer :: precision
  real :: x                        ! may be any kind, may be complex, may be an array
end function
```

present (a)

```
inquiry function present(a)        ! determine whether an optional argument is present
  logical :: present               ! true if a is present, false otherwise
  real :: a                        !  may be any type and kind; a must be an optional argument
end function                       !          of the procedure referencing the present function
```

product (array, dim, mask)

```
function product(array,dim,mask)   ! product of the elements of array
  real :: product                  ! same type and kind as array
  real :: array(:)                 ! may be any kind, any numeric type, and any rank
  integer, optional :: dim         ! if present, 1 ≤ dim ≤ n, where n is rank of array
  logical, optional :: mask        ! if present, mask must have same shape as array
end function
```

The result is scalar if **dim** is omitted or **array** has rank 1, in which case the value returned is the product of the elements of **array**. If array has rank n greater than 1 and **dim** is present, **dim** specifies the dimension along which to compute the products; in this case the result is an array of rank n-1 and shape $(d_1,d_2,...,d_{dim-1}, d_{dim+1},...,d_n)$ where $(d_1,d_2,...,d_n)$ is the shape of **array** and each value of the result is the product of the elements along the **dim** dimension of **array**. If **mask** is present only those elements of **array** corresponding to the true values in **mask** are used in computing the product(s).

radix (x)

```
inquiry function radix(x)          ! radix (base) of the number model for x
  integer :: radix
  real :: x                        ! may be any kind, any numeric type; may be an array
end function
```

random_number (harvest)

```
subroutine randon_number(harvet)   ! generates one or more pseudorandom numbers
  real, intent(out) :: harvest     ! may be any kind, may be an array
end subroutine
```

If **harvest** is a scalar, a single pseudorandom number from the uniform distribution between 0 and 1 is generated and assigned to **harvest**; if **harvest** is an array, **size(harvest)** such numbers are generated and assigned to **harvest**.

random_seed (size, put, get)

```
subroutine random_seed(size,put,get)      ! set or retrieve the random_number seed
  integer, optional, intent(out) :: size  ! number of integers (n) used for the value of the seed
  integer, optional, intent(in) :: put(:) ! size(put) must be equal to n; set the seed to put
  integer, optional, intent(out) :: get(:)! size(get) must be equal to n; retrieve the seed into get
end subroutine                            ! a given call to random_seed has at most one argument
```

If a call to **random_seed** is made without any arguments, the seed is set to an implementation-determined value. When the argument is **put**, the seed is reinitialized to this value; when the argument is **get**, the current value of the seed is retrieved.

range (x)

```
inquiry function range(x)                 ! decimal exponent range for x
  integer :: range                        ! value is int(log10(huge(x))) - but see comment below
  real :: x                               ! may be any kind, any numeric type; may be an array
end function
```

If **x** is of type integer, **huge(x)** returns an integer, which is not legal for **log10**; the effect for **range** is, however, as if the equivalent real value had been returned for huge. If **x** is of type real the value actually returned by **range** is min(int(log10(huge(x))),-int(log10(tiny(x)))).

real (a, kind)

```
elemental function real(a,kind)           ! convert a to the equivalent real value of specified kind
  real(kind) :: real                      ! the converted real value
  real :: a                               ! may be any kind and any numeric type
  integer, optional :: kind               ! if present, must be a scalar initialization expression
end function                              ! if kind is absent, the result kind is default real
```

If **a** is of type real, the result is this same value, but possibly of a different kind. If **a** is an integer, the equivalent real value is returned. If **a** is complex, the result is the real part of **a**.

repeat (string, ncopies)

```
function repeat(string,ncopies)           ! concatenate several copies of a string
  character(len(string)*ncopies) :: repeat ! ncopies of string concatenated ; same kind as string
  character(*) :: string                  ! may be any kind
  integer :: ncopies                      ! 0 ≤ ncopies
end function
```

reshape (source, shape, pad, order)

```
function reshape(source,shape,pad,order)  ! reshape source into the array shape specified shape
  real :: reshape(:)                      ! same type and kind as source, with shape shape
  real :: source(:)                       ! may be any type, kind, and rank
  integer :: shape(:)                     ! all element values must be positive
  real, optional :: pad(:)                ! same type and kind as source; may be any rank
  integer, optional :: order(size(shape)) ! a permutation of (1, 2, 3, ..., size(shape))
end function
```

Values are copied from **source** (and then, if needed, from **pad**) to **reshape**, in array element order. If **size(source)** > **product(shape)**, the extra values of **source** are ignored. If **size(source)** < **product(shape)**, **pad** must be supplied, with **size(pad)** ≥ **product(shape)-size(source)**. If **order** is present it specifies the the array element order of the **reshape** subscripts. For example, **reshape((/1,2,3,4, 5,6/),(/2,3/))** is $\begin{bmatrix}1&3&5\\2&4&6\end{bmatrix}$, and **reshape((/1,2,3,4,5,6/),(/2,4/),(/0,0/),(/2,1/))** is $\begin{bmatrix}1&2&3&4\\5&6&0&0\end{bmatrix}$.

rrspacing (x)

```
elemental function rrspacing(x)          ! recipocal of the relative spacing of values near x
  real :: rrspacing                      ! value is x/(nearest(x,1.)-x) for x > 0
  real :: x
end function
```

scale (x, i)

```
elemental function scale(x,i)            ! scales x by a specified amount
  real :: scale                          ! same kind as x; value is x*radix(x)**i
  real :: x                              ! may be any kind
  integer :: i                           ! scaling may be up (i > 0) or down (i < 0) (or i may be zero)
end function
```

scan (string, set, back)

```
elemental function scan(string,set,back) ! search string for an occurrence of any character in set
  integer :: scan                        ! value is first such position in string
  character(*) :: string                 ! may be any kind
  character(*) :: set                    ! same kind as string
  logical, optional :: back              ! if present and true, search is from back of string instead
end function                             ! if no match, zero is returned
```

selected_int_kind (r)

```
function selected_int_kind(r)            ! determines kind value for specifed integer range
  integer :: selected_int_kind           ! the kind value, or -1 if there is no such integer type
  integer :: r                           ! specifies an integer range of at least -10**r to +10**r
end function
```

If more than one integer type meets the criteria, the kind value for the one with the smallest decimal exponent range is returned or, if there are several such, the smallest of these kind values.

selected_real_kind (p, r)

```
function selected_real_kind(p,r)         ! determines kind value for real type with specified properties
  integer :: selected_real_kind          ! the kind value, or a negative value if there is no such real type
  integer, optional :: p                 ! specifies a real type with at least p decimal digits of precision
  integer, optional :: r                 ! specifies an exponent range of at least 10**-r to 10**r
end function                             ! at least one argument must be present
```

The result is the kind type parameter of a real data type with decimal precision, as returned by the **precision** function, of at least **p** digits and a decimal exponent range, as returned by the **range** function, of at least **r**; if no such type is available on the processor, the result is –1 if the precision is not available, –2 if the exponent range is not available, and –3 if neither is available. If more than one real type meets the criteria, the kind value for the one with the smallest decimal precision is returned or, if there are several such, the smallest of these kind values.

set_exponent (x, i)

```
elemental function set_exponent(x,i)     ! a value with the fractional part of x and exponent i
  real :: set_exponent                   ! same kind as x; value is x*radix(x)**(i-exponent(x))
  real :: x                              ! may be any kind
  integer :: i                           ! i may be positive, negative, or zero
end function
```

shape (source)

```
inquiry function shape(source)           ! determine the shape of an array
  integer :: shape(:)                    ! element values are the extents of source
  real :: source(:)                      ! may be any type, kind, and rank; may be scalar
end function
```

The value of the kth element of **shape** is the size of the kth dimension of **source**. **size(shape)** is n, where n is the rank of **source**; if **source** is a scalar, n is zero. **source** must not be a dissassociated pointer array, an unallocated allocatable array, or an assumed-size array.

sign (a, b)

```
elemental function sign(a,b)          ! set the sign of a value
  real :: sign                        ! value is |a| if b ≥ 0, -|a| if b < 0; type and kind of a
  real :: a                           ! may be any kind; may be integer
  real :: b                           ! same type and kind as a
end function
```

sin (x)

```
elemental function sin(x)             ! the sine of x
  real :: sin                         ! same type and kind as x
  real :: x                           ! may be complex
end function
```

If **x** is of type real, it is regarded as a value in radians; if **x** is of type complex, its real part is regarded as a value in radians.

sinh (x)

```
elemental function sinh(x)            ! the hyperbolic sine of x;
 real :: sinh
 real :: x
end function
```

size (array, dim)

```
inquiry function size(array,dim)      ! determine number of elements in (a dimension of) an array
  integer :: size                     ! value is product(shape(array)) if dim is absent
  real :: array(:)                    ! may be any type, kind, and rank
  integer, optional :: dim            ! if present, 1 ≤ dim ≤ n, where n is the rank of array, and
end function                          !         the returned value is the extent of the dim dimension
```

spacing (x)

```
elemental function spacing(x)         ! absolute spacing near x
  real :: spacing                     ! value is nearest(x,1.)-x for x > 0
  real :: x
end function
```

spread (source, dim, ncopies)

```
function spread(source,dim,ncopies)   ! makes ncopies of source along a new dimension
  real :: spread(:,:)                 ! same type and kind as source; rank 1 greater than source
  real :: source(:)                   ! may be any  type, kind, and rank; may be scalar
  integer :: dim                      ! 1 ≤ dim ≤ n+1, where n is the rank of source
  integer :: ncopies                  ! number of copies to spread is max(0,ncopies)
end function
```

spread broadcasts several copies of **source** along a specified dimension (as in forming a book from copies of a single page) and thus forms an array of rank one greater than **source**. If **source** is a scalar then **spread** is an array of rank one and size max(0,**ncopies**) and all element have the value of **source**. If **source** is an array with shape $(d_1,d_2,...,d_n)$, **spread** has shape $(d_1,d_2,...,d_{dim-1},$ $\max(0,$**ncopies**$),d_{dim},d_{dim+1},...,d_n)$. An example: **spread((/2,3,4/),1,3)** is $\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$.

sqrt (x)

```
elemental function sqrt(x)          ! the square root of x
  real :: sqrt                      ! same type and kind as x; value is √x
  real :: x                         ! may be complex
end function                        ! if x is of type real, x > 0
```

If **x** is complex, the real part of the result is nonnegative; if the real part is zero, the imaginary part is nonnegative.

sum (array, dim, mask)

```
function sum(array,dim,mask)        ! sum of the elements of array
  real :: sum                       ! same type and kind as array
  real :: array(:)                  ! may be any kind, any numeric type, and any rank
  integer, optional :: dim          ! if present, 1 ≤ dim ≤ n, where n is rank of array
  logical, optional :: mask         ! if present, mask must have same shape as array
end function
```

The result is scalar if **dim** is omitted or **array** has rank 1, in which case the value returned is the sum of the elements of **array**. If array has rank n greater than 1 and **dim** is present, 1≤dim≤n and **dim** specifies the dimension along which to compute the sums; in this case the result is an array of rank n-1 and shape $(d_1,d_2,...,d_{dim-1}, d_{dim+1},...,d_n)$ where $(d_1,d_2,...,d_n)$ is the shape of **array** and each value of the result is the sum of the elements along the **dim** dimension of **array**. If **mask** is present only those elements of **array** corresponding to the true values in **mask** are used in computing the sum(s).

system_clock (count, count_rate, count_max)

```
subroutine system_clock(count,count_rate,count_max)   ! integer data from a real-time clock
  integer, intent(out), optional :: count             ! 0 ≤ count ≤ count_max
  integer, intent(out), optional :: count_rate        ! clock counts per second
  integer, intent(out), optional :: count_max         ! the maximum count can have
end subroutine
```

All values are processor dependent; **count_rate** indicates how many times **count** is incremented each second; when it reaches **count_max**, it resets to zero. If there is no processor clock **count** always returns -huge(0), **count_rate** returns zero, and **count_max** returns zero.

tan (x)

```
elemental function tan(x)           ! the tangent of x
  real :: tan
  real :: x                         ! value is assumed to be radians
end function
```

tanh (x)

```
elemental function tanh(x)          ! the hyperbolic tangent of x
  real :: tanh
  real :: x
end function
```

tiny (x)

```
inquiry function tiny(x)            ! the smallest positive value for the type and kind of x
  real :: tiny                      ! same kind as x
  real :: x                         ! may be any kind; may be an array
end function
```

transfer (source, mold, size)

```
function transfer(source,mold,size)     ! same bit pattern, but different type/kind
  real :: transfer                      ! bit pattern of source; type/kind of mold; may be an array
  real :: source                        ! may be any type and kind; may be scalar or array
  real :: mold                          ! may be any type and kind; may be scalar or array
  integer, optional :: size             ! specifies the shape of the result; the value must be positive
end function
```

If **mold** is scalar and **size** is absent the result is scalar. If **mold** is an array and **size** is absent the result is a rank 1 array; its size is the smallest possible to hold all of the bits of **source**. If **size** is present, the result is a rank 1 array of this size; if this makes transfer longer than source, the extra part of transfer is undefined and if it makes transfer shorter than source the extra bits of source are not transferred.

transpose (matrix)

```
function transpose(matrix)                          ! transpose matrix
  real :: transpose(size(matrix,2),size(matrix,1))  ! element (i,j) is matrix(j,i)
  real :: matrix(:,:)                               ! may be any type and kind
end function
```

trim (string)

```
function trim(string)                   ! removes trailing blanks from a string
  character(*) :: trim                  ! same as string, except all trailing blanks removed
  character(*) :: string                ! may be any kind
end function
```

ubound (array, dim)

```
inquiry function ubound(array,dim)      ! the upper bound(s) of array
  integer :: ubound                     ! scalar if dim is present; a rank one array otherwise
  real :: array(:)                      ! may have any type, kind, and rank
  integer, optional :: dim              ! if present, 1 ≤ dim ≤ n, where n is the rank of array
end function
```

If **dim** is present the result is a scalar and is the upper bound of **array** along the dimension **dim**. If **dim** is absent the result is a one-dimensional array whose size is the rank of **array**, and the value of each element of the result is the upper bound of that dimension of **array**. If **array** is an array expression other than an array name (e.g., an array section), the upper bound value is based on the lower bound value being 1.

unpack (vector, mask, field)

```
function unpack(vector,mask,field)      ! unpack a vector into an array, under control of a mask
  real :: unpack(:)                     ! same type and kind as vector; same shape as mask
  real :: vector(:)                     ! may be any type and kind
  logical :: mask(:)                    ! may be any rank
  real :: field(size(mask))             ! same type and kind as vector; same shape as mask -
end function                            !       field may be a scalar, in which case it is broadcast
```

The element of the result that corresponds to the kth true element of **mask**, in array element order, is vector(k), for all the true values in **mask**. Each other element of the result is the value of the corresponding element of **field**.

verify (string, set, back)

```
elemental function verify(string,set,back)  ! check that set contains all the characters in string
  integer :: verify                         ! value is first position in string that is not a set character
  character(*) :: string                    ! may be any kind
  character(*) :: set                       ! same kind as string
  logical, optional :: back                 ! if present and true, check is from back of string instead
end function                                ! if all characters in string are in set, zero is returned
```

11 Syntax Rules

This chapter contains the complete syntax of Fortran 90. For reference purposes the syntax rules are the same as in the Fortran 90 standard, with the same "R" (rule) numbers; however, the constraints are not included here.

Each syntax rule defines a term with the symbol **is**, optionally followed by alternative definitions introduced by the symbol **or**. Optional parts of a definition are in closed in square brackets ([ ]), and repeated parts are enclosed in square brackets followed by three dots ([ ] ...). Abbreviations are used liberally (e.g., *-stmt* for statement) and any term ending with *-list* represents a comma-separated list (e.g., *xyz-list* is an abbreviation for *xyz* [, *xyz*] ...); a term ending with *-name* is a *name* (R304). Syntactic classes (nonterminals) are in *italicized-font* and literals are in **bold**. Literal words, such as **function** are lower case, but upper-case letters are allowed. Where a syntax rule specifies more than one line (statement) of actual code syntax, the rule for each code line is on a separate syntax rule line (e.g., the *if-construct*, R802, involves multiple lines of actual code). In those few cases where a syntax rule is too long to fit on one line, the "#" is used to indicate its continuation on the next line (e.g., the syntax rule for the function statement, R1216, is too long to fit on one line).

general structure (R201-216)

R201 *executable-program* **is** *program-unit*
[*program-unit*] ...

R202 *program-unit* **is** *main-program*
or *external-subprogram*
or *module*
or *block-data*

R1101 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R203 *external-subprogram* **is** *function-subprogram*
or *subroutine-subprogram*

R1215 *function-subprogram* **is** *function-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

R1219 *subroutine-subprogram* **is** *subroutine-stmt*
[*specification-part*]
[*execution-part]*
[*internal-subprogram-part*]
end-subroutine-stmt

R1104 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

R1110 *block-data* **is** *block-data-stmt*
[*specification-part*]
end-block-data-stmt

R204 *specification-part* **is** [*use-stmt*] ...
[*implicit-part*]
[*declaration-construct*] ...

R205 *implicit-part* **is** [*implicit-part-stmt*] ...
implicit-stmt

R206 *implicit-part-stmt* **is** *implicit-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

R207 *declaration-construct* **is** *derived-type-def*
or *interface-block*
or *type-declaration-stmt*
or *specification-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*
or *stmt-function-stmt*

R208 *execution-part* **is** *executable-construct*
[*execution-part-construct*] ...

R209 *execution-part-construct* **is** *executable-construct*
or *format-stmt*
or *data-stmt*
or *entry-stmt*

R210 *internal-subprogram-part* **is** *contains-stmt*
internal-subprogram
[*internal-subprogram*] ...

R211 *internal-subprogram* **is** *function-subprogram*
or *subroutine-subprogram*

R212 *module-subprogram-part* **is** *contains-stmt*
module-subprogram
[*module-subprogram*] ...

R213 *module-subprogram* **is** *function-subprogram*
or *subroutine-subprogram*

R214 *specification-stmt* **is** *access-stmt*
or *allocatable-stmt*
or *common-stmt*
or *data-stmt*
or *dimension-stmt*

or *equivalence-stmt*
or *external-stmt*
or *intent-stmt*
or *intrinsic-stmt*
or *namelist-stmt*
or *optional-stmt*
or *pointer-stmt*
or *save-stmt*
or *target-stmt*

R215 *executable-construct* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *if-construct*
or *where-construct*

R216 *action-stmt* **is** *allocate-stmt*
or *assignment-stmt*
or *backspace-stmt*
or *call-stmt*
or *close-stmt*
or *computed-goto-stmt*
or *continue-stmt*
or *cycle-stmt*
or *deallocate-stmt*
or *endfile-stmt*
or *end-function-stmt*
or *end-program-stmt*
or *end-subroutine-stmt*
or *exit-stmt*
or *goto-stmt*
or *if-stmt*
or *inquire-stmt*
or *nullify-stmt*
or *open-stmt*
or *pointer-assignment-stmt*
or *print-stmt*
or *read-stmt*
or *return-stmt*
or *rewind-stmt*
or *stop-stmt*
or *where-stmt*
or *write-stmt*
or *arithmetic-if-stmt*
or *assign-stmt*
or *assigned-goto-stmt*
or *pause-stmt*

tokens (names, operators ...) R301-313

R301	*character*	**is**	*alphanumeric-character*
		or	*special-character*
R302	*alphanumeric-character*	**is**	*letter*
		or	*digit*
		or	*underscore*
R303	*underscore*	**is**	_
R304	*name*	**is**	*letter* [*alphanumeric-character*] ...
R305	*constant*	**is**	*literal-constant*
		or	*named-constant*
R306	*literal-constant*	**is**	*int-literal-constant*
		or	*real-literal-constant*
		or	*complex-literal-constant*
		or	*logical-literal-constant*
		or	*char-literal-constant*
		or	*boz-literal-constant*
R307	*named-constant*	**is**	*name*
R308	*int-constant*	**is**	*constant*
R309	*char-constant*	**is**	*constant*
R310	*intrinsic-operator*	**is**	*power-op*
		or	*mult-op*
		or	*add-op*
		or	*concat-op*
		or	*rel-op*
		or	*not-op*
		or	*and-op*
		or	*or-op*
		or	*equiv-op*
R708	*power-op*	**is**	****
R709	*mult-op*	**is**	*
		or	/
R710	*add-op*	**is**	+
		or	–
R712	*concat-op*	**is**	//
R714	*rel-op*	**is**	.eq.
		or	.ne.
		or	.lt.

		or	**.le.**
		or	**.gt.**
		or	**.ge.**
		or	**==**
		or	**/=**
		or	**<**
		or	**<=**
		or	**>**
		or	**>=**
R719	*not-op*	**is**	**.not.**
R720	*and-op*	**is**	**.and.**
R721	*or-op*	**is**	**.or.**
R722	*equiv-op*	**is**	**.eqv.**
		or	**.neqv.**
R311	*defined-operator*	**is**	*defined-unary-op*
		or	*defined-binary-op*
		or	*extended-intrinsic-op*
R704	*defined-unary-op*	**is**	**.** *letter* [*letter*] ... **.**
R724	*defined-binary-op*	**is**	**.** *letter* [*letter*] ... **.**
R312	*extended-intrinsic-op*	**is**	*intrinsic-operator*
R313	*label*	**is**	*digit* [*digit* [*digit* [*digit* [*digit*]]]]

data types (R401-435)

R401	*signed-digit-string*	**is**	[*sign*] *digit-string*
R402	*digit-string*	**is**	*digit* [*digit*] ...
R403	*signed-int-literal-constant*	**is**	[*sign*] *int-literal-constant*
R404	*int-literal-constant*	**is**	*digit-string* [_ *kind-param*]
R405	*kind-param*	**is**	*digit-string*
		or	*scalar-int-constant-name*
R406	*sign*	**is**	**+**
		or	**–**
R407	*boz-literal-constant*	**is**	*binary-constant*
		or	*octal-constant*
		or	*hex-constant*

R408 *binary-constant* **is** **B '** *digit* [*digit*] ... **'**
or **B "** *digit* [*digit*] ... **"**

R409 *octal-constant* **is** **O '** *digit* [*digit*] ... **'**
or **O "** *digit* [*digit*] ... **"**

R410 *hex-constant* **is** **Z '** *hex-digit* [*hex-digit*] ... **'**
or **Z "** *hex-digit* [*hex-digit]* ... **"**

R411 *hex-digit* **is** *digit*
or **A**
or **B**
or **C**
or **D**
or **E**
or **F**

R412 *signed-real-literal-constant* **is** [*sign*] *real-literal-constant*

R413 *real-literal-constant* **is** *significand* [*exponent-letter exponent*] [_ *kind-param*]
or *digit-string exponent-letter exponent* [_ *kind-param*]

R414 *significand* **is** *digit-string* **.** [*digit-string*]
or **.** *digit-string*

R415 *exponent-letter* **is** **E**
or **D**

R416 *exponent* **is** *signed-digit-string*

R417 *complex-literal-constant* **is** **(** *real-part* **,** *imag-part* **)**

R418 *real-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*

R419 *imag-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*

R420 *char-literal-constant* **is** [*kind-param* _] **'** [*rep-char*] ... **'**
or [*kind-param* _] **"** [*rep-char*] ... **"**

R421 *logical-literal-constant* **is** **.true.** [_ *kind-param*]
or **.false.** [_ *kind-param*]

R422 *derived-type-def* **is** *derived-type-stmt*
[*private-sequence-stmt*] ...
component-def-stmt
[*component-def-stmt*] ...
end-type-stmt

R423 *private-sequence-stmt* **is** **private**
or **sequence**

R424 *derived-type-stmt* **is** **type** [[**,** *access-spec*] **::**] *type-name*

R425 *end-type-stmt* **is** **end type** [*type-name*]

R426 *component-def-stmt* **is** *type-spec* [[**,** *component-attr-spec-list*] **::**] *component-decl-list*

R427 *component-attr-spec* **is** **pointer**
or **dimension (** *component-array-spec* **)**

R428 *component-array-spec* **is** *explicit-shape-spec-list*
or *deferred-shape-spec-list*

R429 *component-decl* **is** *component-name* [**(** *component-array-spec* **)**] [***** *char-length*]

R430 *structure-constructor* **is** *type-name* **(** *expr-list* **)**

R431 *array-constructor* **is** **(/** *ac-value-list* **/)**

R432 *ac-value* **is** *expr*
or *ac-implied-do*

R433 *ac-implied-do* **is** **(** *ac-value-list* **,** *ac-implied-do-control* **)**

R434 *ac-implied-do-control* **is** *ac-do-variable* **=** *scalar-int-expr* **,** *scalar-int-expr* [**,** *scalar-int-expr*]

R435 *ac-do-variable* **is** *scalar-int-variable*

declarations and attributes (R501-549)

R501 *type-declaration-stmt* **is** *type-spec* [[**,** *attr-spec*] ... **::**] *entity-decl-list*

R502 *type-spec* **is** **integer** [*kind-selector*]
or **real** [*kind-selector*]
or **double precision**
or **complex** [*kind-selector*]
or **character** [*char-selector*]
or **logical** [*kind-selector*]
or **type (** *type-name* **)**

R503 *attr-spec* **is** **parameter**
or *access-spec*
or **allocatable**
or **dimension (** *array-spec* **)**
or **external**
or **intent (** *intent-spec* **)**
or **intrinsic**
or **optional**
or **pointer**
or **save**
or **target**

R504 *entity-decl* **is** *object-name* [**(** *array-spec* **)**] [***** *char-length*] [**=** *initialization-expr*]
or *function-name* [**(** *array-spec* **)**] [***** *char-length*]

R505 *kind-selector* **is** **(** [**kind =**] *scalar-int-initialization-expr* **)**

R506 *char-selector* **is** *length-selector*
or **(len =** *type-param-value* **, kind =** *scalar-int-initialization-expr* **)**
or **(** *type-param-value* **,** [**kind =**] *scalar-int-initialization-expr* **)**
or **(kind =** *scalar-int-initialization-expr* [**, len =** *type-param-value*] **)**

R507 *length-selector* **is** **(** [**len =**] *type-param-value* **)**
or ***** *char-length* [**,**]

R508 *char-length* **is** **(** *type-param-value* **)**
or *scalar-int-literal-constant*

R509 *type-param-value* **is** *specification-expr*
or *****

R510 *access-spec* **is** **public**
or **private**

R511 *intent-spec* **is** **in**
or **out**
or **inout**

R512 *array-spec* **is** *explicit-shape-spec-list*
or *assumed-shape-spec-list*
or *deferred-shape-spec-list*
or *assumed-size-spec*

R513 *explicit-shape-spec* **is** [*lower-bound* **:**] *upper-bound*

R514 *lower-bound* **is** *specification-expr*

R515 *upper-bound* **is** *specification-expr*

R516 *assumed-shape-spec* **is** [*lower-bound*] **:**

R517 *deferred-shape-spec* **is** **:**

R518 *assumed-size-spec* **is** [*explicit-shape-spec-list* **,**] [*lower-bound* **:**] *****

R519 *intent-stmt* **is** **intent (** *intent-spec* **)** [**::**] *dummy-arg-name-list*

R520 *optional-stmt* **is** **optional** [**::**] *dummy-arg-name-list*

R521 *access-stmt* **is** *access-spec* [[**::**] *access-id-list*]

R522 *access-id* **is** *use-name*
or *generic-spec*

R523 *save-stmt* **is** **save** [[**::**] *saved-entity-list*]

R524 *saved-entity* **is** *object-name*
or **/** *common-block-name* **/**

R525 *dimension-stmt* **is** **dimension** [**::**] *array-name* **(** *array-spec* **)** #
[**,** *array-name* **(** *array-spec* **)**] ...

R526 *allocatable-stmt* **is** **allocatable** [**::**] *array-name* [**(** *deferred-shape-spec-list* **)**] #
[**,** *array-name* [**(** *deferred-shape-spec-list* **)**]] ...

R527 *pointer-stmt* **is** **pointer** [**::**] *object-name* [**(** *deferred-shape-spec-list* **)**] #
[**,** *object-name* [**(** *deferred-shape-spec-list* **)**]] ...

R528 *target-stmt* **is** **target** [**::**] *object-name* [**(** *array-spec* **)**] #
[**,** *object-name* [**(** *array-spec* **)**]] ...

R529 *data-stmt* **is** **data** *data-stmt-set* [[**,**] *data-stmt-set*] ...

R530 *data-stmt-set* **is** *data-stmt-object-list* **/** *data-stmt-value-list* **/**

R531 *data-stmt-object* **is** *variable*
or *data-implied-do*

R532 *data-stmt-value* **is** [*data-stmt-repeat* *****] *data-stmt-constant*

R533 *data-stmt-constant* **is** *scalar-constant*
or *signed-int-literal-constant*
or *signed-real-literal-constant*
or *structure-constructor*
or *boz-literal-constant*

R534 *data-stmt-repeat* **is** *scalar-int-constant*

R535 *data-implied-do* **is** **(** *data-i-do-object-list* **,** *data-i-do-variable* **=** #
scalar-int-expr **,** *scalar-int-expr* [**,** *scalar-int-expr*] **)**

R536 *data-i-do-object* **is** *array-element*
or *scalar-structure-component*
or *data-implied-do*

R537 *data-i-do-variable* **is** *scalar-int-variable*

R538 *parameter-stmt* **is** **parameter (** *named-constant-def-list* **)**

R539 *named-constant-def* **is** *named-constant* **=** *initialization-expr*

R540 *implicit-stmt* **is** **implicit** *implicit-spec-list*
or **implicit none**

R541 *implicit-spec* **is** *type-spec* **(** *letter-spec-list* **)**

R542 *letter-spec* **is** *letter* [**–** *letter*]

R543 *namelist-stmt* **is** **namelist /** *namelist-group-name* **/** *namelist-group-object-list* #
[[**,**] **/** *namelist-group-name* **/** *namelist-group-object-list*] ...

R544 *namelist-group-object* **is** *variable-name*

R545 *equivalence-stmt* **is** **equivalence** *equivalence-set-list*

R546 *equivalence-set* **is** **(** *equivalence-object* **,** *equivalence-object-list* **)**

R547 *equivalence-object* **is** *variable-name*
or *array-element*
or *substring*

R548 *common-stmt* **is** **common** [**/** [*common-block-name*] **/**] *common-block-object-list* #
[[**,**] **/** [*common-block-name*] **/** *common-block-object-list*] ...

R549 *common-block-object* **is** *variable-name* [**(** *explicit-shape-spec-list* **)**]

variables (R601-631)

R601 *variable* **is** *scalar-variable-name*
or *array-variable-name*
or *subobject*

R602 *subobject* **is** *array-element*
or *array-section*
or *structure-component*
or *substring*

R603 *logical-variable* **is** *variable*

R604 *default-logical-variable* **is** *variable*

R605 *char-variable* **is** *variable*

R606 *default-char-variable* **is** *variable*

R607 *int-variable* **is** *variable*

R608 *default-int-variable* **is** *variable*

R609 *substring* **is** *parent-string* **(** *substring-range* **)**

R610 *parent-string* **is** *scalar-variable-name*
or *array-element*
or *scalar-structure-component*
or *scalar-constant*

R611 *substring-range* **is** [*scalar-int-expr*] **:** [*scalar-int-expr*]

R612 *data-ref* **is** *part-ref* [**%** *part-ref*] ...

R613 *part-ref* **is** *part-name* [**(** *section-subscript-list* **)**]

R614 *structure-component* **is** *data-ref*

R615 *array-element* **is** *data-ref*

R616 *array-section* **is** *data-ref* [**(** *substring-range* **)**]

R617 *subscript* **is** *scalar-int-expr*

R618 *section-subscript* **is** *subscript*
or *subscript-triplet*
or *vector-subscript*

R619 *subscript-triplet* **is** [*subscript*] **:** [*subscript*] [**:** *stride*]

R620 *stride* **is** *scalar-int-expr*

R621 *vector-subscript* **is** *int-expr*

R622 *allocate-stmt* **is** **allocate (** *allocation-list* [**, stat =** *stat-variable*] **)**

R623 *stat-variable* **is** *scalar-int-variable*

R624 *allocation* **is** *allocate-object* [**(** *allocate-shape-spec-list* **)**]

R625 *allocate-object* **is** *variable-name*
or *structure-component*

R626 *allocate-shape-spec* **is** [*allocate-lower-bound* **:**] *allocate-upper-bound*

R627 *allocate-lower-bound* **is** *scalar-int-expr*

R628 *allocate-upper-bound* **is** *scalar-int-expr*

R629 *nullify-stmt* **is** **nullify (** *pointer-object-list* **)**

R630 *pointer-object* **is** *variable-name*
or *structure-component*

R631 *deallocate-stmt* **is** **deallocate (** *allocate-object-list* [**, stat =** *stat-variable*] **)**

expressions (R701-743)

R701 *primary* **is** *constant*
or *constant-subobject*
or *variable*
or *array-constructor*
or *structure-constructor*
or *function-reference*
or **(** *expr* **)**

R702 *constant-subobject* **is** *subobject*

R703 *level-1-expr* **is** [*defined-unary-op*] *primary*

R704 *defined-unary-op* **is** **.** *letter* [*letter*] ... **.**

R705 *mult-operand* **is** *level-1-expr* [*power-op mult-operand*]

R706 *add-operand* **is** [*add-operand mult-op*] *mult-operand*

R707 *level-2-expr* **is** [[*level-2-expr*] *add-op*] *add-operand*

R708 *power-op* **is** ******

R709 *mult-op* **is** *****
or **/**

R710 *add-op* **is** **+**
or **−**

R711 *level-3-expr* **is** [*level-3-expr concat-op*] *level-2-expr*

R712 *concat-op* **is** **//**

R713 *level-4-expr* **is** [*level-3-expr rel-op*] *level-3-expr*

R714 *rel-op* **is** **.eq.**
or **.ne.**
or **.lt.**
or **.le.**
or **.gt.**
or **.ge.**
or **==**
or **/=**
or **<**
or **<=**
or **>**
or **>=**

R715 *and-operand* **is** [*not-op*] *level-4-expr*

R716 *or-operand* **is** [*or-operand and-op*] *and-operand*

R717 *equiv-operand* **is** [*equiv-operand or-op*] *or-operand*

R718 *level-5-expr* **is** [*level-5-expr equiv-op*] *equiv-operand*

R719 *not-op* **is** **.not.**

R720 *and-op* **is** **.and.**

R721 *or-op* **is** **.or.**

R722 *equiv-op* **is** **.eqv.**
or **.neqv.**

R723 *expr* **is** [*expr defined-binary-op*] *level-5-expr*

R724 *defined-binary-op* **is** **.** *letter* [*letter*] ... **.**

R725 *logical-expr* **is** *expr*

R726 *char-expr* **is** *expr*

R727 *default-char-expr* **is** *expr*

R728 *int-expr* **is** *expr*

R729 *numeric-expr* **is** *expr*

R730 *initialization-expr* **is** *expr*

R731 *char-initialization-expr* **is** *char-expr*

R732 *int-initialization-expr* **is** *int-expr*

R733 *logical-initialization-expr* **is** *logical-expr*

R734 *specification-expr* **is** *scalar-int-expr*

R735 *assignment-stmt* **is** *variable* **=** *expr*

R736 *pointer-assignment-stmt* **is** *pointer-object* **=>** *target*

R737 *target* **is** *variable*
or *expr*

R738 *where-stmt* **is** **where (** *mask-expr* **)** *assignment-stmt*

R739 *where-construct* **is** *where-construct-stmt*
[*assignment-stmt*] ...
[*elsewhere-stmt*
[*assignment-stmt*] ...]
end-where-stmt

R740 *where-construct-stmt* **is** **where (** *mask-expr* **)**

R741 *mask-expr* **is** *logical-expr*

R742 *elsewhere-stmt* **is** **elsewhere**

R743 *end-where-stmt* **is** **end where**

control structures (R801-844)

R801 *block* **is** [*execution-part-construct*] ...

R802 *if-construct* **is** *if-then-stmt*
block
[*else-if-stmt*
block] ...
[*else-stmt*
block]
end-if-stmt

R803 *if-then-stmt* **is** [*if-construct-name* **:**] **if (** *scalar-logical-expr* **) then**

R804 *else-if-stmt* **is** **else if (** *scalar-logical-expr* **) then** [*if-construct-name*]

R805 *else-stmt* **is** **else** [*if-construct-name*]

R806 *end-if-stmt* **is** **end if** [*if-construct-name*]

R807 *if-stmt* **is** **if (** *scalar-logical-expr* **)** *action-stmt*

R808 *case-construct* **is** *select-case-stmt*
[*case-stmt*
block] ...
end-select-stmt

R809 *select-case-stmt* **is** [*case-construct-name* **:**] **select case (** *case-expr* **)**

R810 *case-stmt* **is** **case** *case-selector* [*case-construct-name*]

R811 *end-select-stmt* **is** **end select** [*case-construct-name*]

R812 *case-expr* **is** *scalar-int-expr*
or *scalar-char-expr*
or *scalar-logical-expr*

R813 *case-selector* **is** **(** *case-value-range-list* **)**
or **default**

R814 *case-value-range* **is** *case-value*
or *case-value* **:**
or **:** *case-value*
or *case-value* **:** *case-value*

R815 *case-value* **is** *scalar-int-initialization-expr*
or *scalar-char-initialization-expr*
or *scalar-logical-initialization-expr*

R816 *do-construct* **is** *block-do-construct*
or *nonblock-do-construct*

R817 *block-do-construct* **is** *do-stmt*
do-block
end-do

R818 *do-stmt* **is** *label-do-stmt*
or *nonlabel-do-stmt*

R819 *label-do-stmt* **is** [*do-construct-name* **:**] **do** *label* [*loop-control*]

R820 *nonlabel-do-stmt* **is** [*do-construct-name* **:**] **do** [*loop-control*]

R821 *loop-control* **is** [**,**] *do-variable* **=** *scalar-numeric-expr* **,** *scalar-numeric-expr* #
[**,** *scalar-numeric-expr*]
or [**,**] **while (** *scalar-logical-expr* **)**

R822 *do-variable* **is** *scalar-variable*

R823 *do-block* **is** *block*

R824 *end-do* **is** *end-do-stmt*
or *continue-stmt*

R825 *end-do-stmt* **is** **end do** [*do-construct-name*]

R826 *nonblock-do-construct* **is** *action-term-do-construct*
or *outer-shared-do-construct*

R827 *action-term-do-construct* **is** *label-do-stmt*
do-body
do-term-action-stmt

R828 *do-body* **is** [*execution-part-construct*] ...

R829 *do-term-action-stmt* **is** *action-stmt*

R830 *outer-shared-do-construct* **is** *label-do-stmt*
do-body
shared-term-do-construct

R831 *shared-term-do-construct* **is** *outer-shared-do-construct*
or *inner-shared-do-construct*

R832 *inner-shared-do-construct* **is** *label-do-stmt*
do-body
do-term-shared-stmt

R833 *do-term-shared-stmt* **is** *action-stmt*

R834 *cycle-stmt* **is** **cycle** [*do-construct-name*]

R835 *exit-stmt* **is** **exit** [*do-construct-name*]

R836 *goto-stmt* **is** **go to** *label*

R837 *computed-goto-stmt* **is** **go to (** *label-list* **)** [**,**] *scalar-int-expr*

R838 *assign-stmt* **is** **assign** *label* **to** *scalar-int-variable*

R839 *assigned-goto-stmt* **is** **go to** *scalar-int-variable* [[**,**] **(** *label-list* **)**]

R840 *arithmetic-if-stmt* **is** **if (** *scalar-numeric-expr* **)** *label* **,** *label* **,** *label*

R841 *continue-stmt* **is** **continue**

R842 *stop-stmt* **is** **stop** [*stop-code*]

R843 *stop-code* **is** *scalar-char-constant*
or *digit* [*digit* [*digit* [*digit* [*digit*]]]]

R844 *pause-stmt* **is** **pause** [*stop-code*]

input, output (R901-924)

R901 *io-unit* **is** *external-file-unit*
or *****
or *internal-file-unit*

R902 *external-file-unit* **is** *scalar-int-expr*

R903 *internal-file-unit* **is** *default-char-variable*

R904 *open-stmt* **is** **open (** *connect-spec-list* **)**

R905 *connect-spec* **is** [**unit =**] *external-file-unit*
or **iostat =** *scalar-default-int-variable*
or **err =** *label*
or **file =** *file-name-expr*
or **status =** *scalar-default-char-expr*
or **access =** *scalar-default-char-expr*
or **form =** *scalar-default-char-expr*
or **recl =** *scalar-int-expr*
or **blank =** *scalar-default-char-expr*
or **position =** *scalar-default-char-expr*
or **action =** *scalar-default-char-expr*
or **delim =** *scalar-default-char-expr*
or **pad =** *scalar-default-char-expr*

R906 *file-name-expr* **is** *scalar-default-char-expr*

R907 *close-stmt* **is** **close (** *close-spec-list* **)**

R908 *close-spec* **is** [**unit =**] *external-file-unit*
or **iostat =** *scalar-default-int-variable*
or **err =** *label*
or **status =** *scalar-default-char-expr*

R909 *read-stmt* **is** **read (** *io-control-spec-list* **)** [*input-item-list*]
or **read** *format* [**,** *input-item-list*]

R910 *write-stmt* **is** **write (** *io-control-spec-list* **)** [*output-item-list*]

R911 *print-stmt* **is** **print** *format* [**,** *output-item-list*]

R912 *io-control-spec* **is** [**unit =**] *io-unit*
or [**fmt =**] *format*
or [**nml =**] *namelist-group-name*
or **rec =** *scalar-int-expr*
or **iostat =** *scalar-default-int-variable*
or **err =** *label*
or **end =** *label*
or **advance =** *scalar-default-char-expr*
or **size =** *scalar-default-int-variable*
or **eor =** *label*

R913 *format* **is** *default-char-expr*
or *label*
or *****
or *scalar-default-int-variable*

R914 *input-item* **is** *variable*
or *io-implied-do*

R915 *output-item* **is** *expr*
or *io-implied-do*

R916 *io-implied-do* **is** **(** *io-implied-do-object-list* **,** *io-implied-do-control* **)**

R917 *io-implied-do-object* **is** *input-item*
or *output-item*

R918 *io-implied-do-control* **is** *do-variable* **=** *scalar-numeric-expr* **,** *scalar-numeric-expr* #
[**,** *scalar-numeric-expr*]

R919 *backspace-stmt* **is** **backspace** *external-file-unit*
or **backspace (** *position-spec-list* **)**

R920 *endfile-stmt* **is** **endfile** *external-file-unit*
or **endfile (** *position-spec-list* **)**

R921 *rewind-stmt* **is** **rewind** *external-file-unit*
or **rewind (** *position-spec-list* **)**

R922 *position-spec* **is** [**unit =**] *external-file-unit*
or **iostat =** *scalar-default-int-variable*
or **err =** *label*

R923 *inquire-stmt* **is** **inquire (** *inquire-spec-list* **)**
or **inquire (iolength =** *scalar-default-int-variable* **)** *output-item-list*

R924 *inquire-spec* **is** [**unit =**] *external-file-unit*
or **file =** *file-name-expr*
or **iostat =** *scalar-default-int-variable*
or **err =** *label*
or **exist =** *scalar-default-logical-variable*
or **opened =** *scalar-default-logical-variable*
or **number =** *scalar-default-int-variable*
or **named =** *scalar-default-logical-variable*
or **name =** *scalar-default-char-variable*
or **access =** *scalar-default-char-variable*
or **sequential =** *scalar-default-char-variable*
or **direct =** *scalar-default-char-variable*
or **form =** *scalar-default-char-variable*
or **formatted =** *scalar-default-char-variable*
or **unformatted =** *scalar-default-char-variable*
or **recl =** *scalar-default-int-variable*
or **nextrec =** *scalar-default-int-variable*
or **blank =** *scalar-default-char-variable*
or **position =** *scalar-default-char-variable*
or **action =** *scalar-default-char-variable*
or **read =** *scalar-default-char-variable*
or **write =** *scalar-default-char-variable*
or **readwrite =** *scalar-default-char-variable*
or **delim =** *scalar-default-char-variable*
or **pad =** *scalar-default-char-variable*

I/O formatting (R1001-1017)

R1001 *format-stmt* **is** **format** *format-specification*

R1002 *format-specification* **is** **(** [*format-item-list*] **)**

R1003 *format-item* **is** [*r*] *data-edit-desc*
or *control-edit-desc*
or *char-string-edit-desc*
or [*r*] **(** *format-item-list* **)**

R1004 *r* **is** *int-literal-constant*

R1005 *data-edit-desc* **is** **I** *w* [**.** *m*]
or **B** *w* [**.** *m*]
or **O** *w* [**.** *m*]
or **Z** *w* [**.** *m*]
or **F** *w* **.** *d*
or **E** *w* **.** *d* [**E** *e*]
or **EN** *w* **.** *d* [**E** *e*]
or **ES** *w* **.** *d* [**E** *e*]
or **G** *w* **.** *d* [**E** *e*]
or **L** *w*
or **A** [*w*]
or **D** *w* **.** *d*

R1006 *w* **is** *int-literal-constant*

R1007 *m* **is** *int-literal-constant*

R1008 *d* **is** *int-literal-constant*

R1009 *e* **is** *int-literal-constant*

R1010 *control-edit-desc* **is** *position-edit-desc*
or [*r*] ***/***
or **:**
or *sign-edit-desc*
or *k* **P**
or *blank-interp-edit-desc*

R1011 *k* **is** *signed-int-literal-constant*

R1012 *position-edit-desc* **is** **T** *n*
or **TL** *n*
or **TR** *n*
or *n* **X**

R1013 *n* **is** *int-literal-constant*

R1014 *sign-edit-desc* **is** **S**
or **SP**
or **SS**

R1015 *blank-interp-edit-desc* **is** **BN**
or **BZ**

R1016 *char-string-edit-desc* **is** *char-literal-constant*
or *c* **H** *rep-char* [*rep-char*] ...

R1017 *c* **is** *int-literal-constant*

program units (R1101-1112)

R1101 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R1102 *program-stmt* **is** **program** *program-name*

R1103 *end-program-stmt* **is** **end** [**program** [*program-name*]]

R1104 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

R1105 *module-stmt* **is** **module** *module-name*

R1106 *end-module-stmt* **is** **end** [**module** [*module-name*]]

R1107 *use-stmt* **is** **use** *module-name* [**,** *rename-list*]
or **use** *module-name* **, only :** [*only-list*]

R1108 *rename* **is** *local-name* **=>** *use-name*

R1109 *only* **is** *access-id*
or [*local-name* **=>**] *use-name*

R1110 *block-data* **is** *block-data-stmt*
[*specification-part*]
end-block-data-stmt

R1111 *block-data-stmt* **is** **block data** [*block-data-name*]

R1112 *end-block-data-stmt* **is** **end** [**block data** [*block-data-name*]]

procedures (R1201-1226)

R1201 *interface-block* **is** *interface-stmt*
[*interface-body*] ...
[*module-procedure-stmt*] ...
end-interface-stmt

R1202 *interface-stmt* **is** **interface** [*generic-spec*]

R1203 *end-interface-stmt* **is** **end interface**

R1204 *interface-body* **is** *function-stmt*
[*specification-part*]
end-function-stmt
or *subroutine-stmt*
[*specification-part*]
end-subroutine-stmt

R1205 *module-procedure-stmt* **is** **module procedure** *procedure-name-list*

R1206 *generic-spec* **is** *generic-name*
or **operator (** *defined-operator* **)**
or **assignment (=)**

R1207 *external-stmt* **is** **external** [**::**] *external-name-list*

R1208 *intrinsic-stmt* **is** **intrinsic** [**::**] *intrinsic-procedure-name-list*

R1209 *function-reference* **is** *function-name* **(** [*actual-arg-spec-list*] **)**

R1210 *call-stmt* **is** **call** *subroutine-name* [**(** [*actual-arg-spec-list*] **)**]

R1211 *actual-arg-spec* **is** [*keyword* **=**] *actual-arg*

R1212 *keyword* **is** *dummy-arg-name*

R1213 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or *alt-return-spec*

R1214 *alt-return-spec* **is** ***** *label*

R1215 *function-subprogram* **is** *function-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

R1216 *function-stmt* **is** [*prefix*] **function** *function-name* **(** [*dummy-arg-name-list*] **)** #
[**result (** *result-name* **)**]

R1217 *prefix* **is** *type-spec* [**recursive**]
or **recursive** [*type-spec*]

R1218 *end-function-stmt* **is** **end** [**function** [*function-name*]]

R1219 *subroutine-subprogram* **is** *subroutine-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-subroutine-stmt

R1220 *subroutine-stmt* **is** [**recursive**] **subroutine** *subroutine-name* [**(** [*dummy-arg-list*] **)**]

R1221 *dummy-arg* **is** *dummy-arg-name*
or *****

R1222 *end-subroutine-stmt* **is** **end** [**subroutine** [*subroutine-name*]]

R1223 *entry-stmt* **is** **entry** *entry-name* [**(** [*dummy-arg-list*] **)** [**result (** *result-name* **)**]]

R1224 *return-stmt* **is** **return** [*scalar-int-expr*]

R1225 *contains-stmt* **is** **contains**

R1226 *stmt-function-stmt* **is** *function-name* **(** [*dummy-arg-name-list*] **) =** *scalar-expr*

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

12 A Fortran 90 Implementation

This chapter describes vendor-specific features of the Absoft Pro Fortran™ implementation of Fortran 90, including implementation-dependent values (kind values, I/O error values, etc.), language extensions, compiler directives, and command-line compiler options. The implementation-dependent values will vary, but many of these extensions, directives, and command-line options are typical of many commercial implementations.

implementation-dependent values

kind values for all intrinsic data types

kind type parameters for	value	example kind constant declarations[a]
default integer	4	integer, parameter :: DEFAULT=kind(1)
short (2-byte) integer	2	integer, parameter :: SHORT=2
shorter (1-byte) integer	1	integer, parameter :: ONE_BYTE=1
default real (and complex)	4	integer, parameter :: SINGLE=kind(1E1)
double real (and complex)	8	integer, parameter :: DOUBLE=kind(1D1)
default logical	4	integer, parameter :: LOG_KIND=kind(.true.)
2-byte logical	2	integer, parameter :: TWO_BYTE=2
1-byte logical	1	integer, parameter :: BYTE=1
default character	1	integer, parameter :: CHAR_KIND=kind(' ')

a. Only four of these constants are really needed, one for each kind value.

iostat= variable values for EOF and EOR

iostat value	I/O condition
-1	end of file (triggers end=, if present)
-2	end of record, in nonadvancing read (triggers eor=)

iostat= variable values for various i/o error conditions (trigger err=, if present)

value	error condition	value	error condition
2	no such file or directory	10026	block= valid only for unformatted sequential files
3	resource not found	10027	unable to truncate after rewind, backspace, or endfile
5	physical I/O error	10028	formatted I/O attempted on entire structure
6	no such device or address	10029	negative unit specifiers are not permitted

value	error condition	value	error condition
7	insufficient space for return argument	10030	open specifiers do not match currently open file
9	bad file number	10031	cannot implicitly open for direct access
12	not enough space	10032	status “new” specified for existing file
13	permission denied	10033	command not allowed for unit type
17	file exists	10034	MRWE is required for that feature
19	no such device	10035	bad specification for window
20	not a directory	10036	endian specifier not “big-endian” or “little_endian”
21	is a directory	10037	cannot endian-convert entire structures
22	invalid parameter	10038	attempt to read past end of record
23	file table overflow; too many files open	10039	attempt to read past end of record in nonadvancing I/O
24	too many files open	10040	illegal specifier for advance=
28	no space left on device; volume full	10041	illegal specifier for delim=
29	illegal seek	10042	illegal specifier for pad=
30	read-only file system access	10043	size= specified with advance="yes"
31	too many links; can’t delete an open file	10044	eor= specified with advance="yes"
10000	file not open for read	10045	cannot deallocate disassociated or unallocated object
10001	file not open for write	10046	cannot deallocate a portion of an original allocation
10002	file not found	10047	an allocatable array has already been allocated
10003	record length negative or zero	10048	internal or unknown runtime library error
10004	buffer allocation failed	10049	unknown data type passed to runtime library
10005	bad io-list specifier	10050	illegal dim argument to an array intrinsic
10006	error in format string	10051	source argument to reshape smaller than shape array
10007	illegal repeat count	10052	shape array for reshape contains a negative value
10008	Hollerith count exceeds remaining format string	10053	cannot inquire about unallocated/disassociated array
10009	format string missing opening (	10054	the ncopies to repeat is negative
10010	format string has unmatched parentheses	10055	the s argument to nearest is negative
10011	format string has unmatched quotes	10056	the order argument to reshape is illegal
10012	nonrepeatable format descriptor	10057	result of transfer with no size is smaller than source
10013	attempt to read past end of file	10058	shape array for reshape is zero size
10014	bad file specification	10059	vector argument to unpack contains insufficient values
10015	format group table overflow	10060	attempt to write a record longer than specified length
10016	illegal character in numeric input	10061	advance= specified for direct or unformatted file
10017	no record specified for direct access	10062	namelist name is longer than specified record length
10018	maximum record number exceeded	10063	namelist variable name exceeds maximum length
10019	illegal file type for namelist I/O	10064	pad= specified for unformatted file
10020	illegal input for namelist I/O	10065	namelist input contains multiple strided arrays
10021	variable not present in current namelist	10066	expected & or $ as first character in namelist input
10022	variable type or size does not match edit descriptor	10067	namelist group does not match current input group
10023	illegal direct access record number	10068	pointer or allocatable array not associated or allocated
10024	illegal use of internal file	10069	namelist input contains negative array stride
10025	recl= valid only for direct access files		

language extensions

dec-style structures. A *structure type* is a data type extension that is similar to a sequence derived type, but components of objects of structure types are guaranteed to be physically stored in the order defined. Note that the terms "structure" and "structured object" refer to an object of derived-type; the terms "structure type" and "structure definition" will be used to refer to this extended type, and the terms "record" and "record object" will refer to objects of this extended type. A structure definition has the form:

structure [**/** *structure-name* **/**] [*record-list*]
abx-component-def
[*abx-component-def*]...
end structure

An *abx-component-def* is an Fortran 90 component-def-stmt (R426), a structure definition (structure definitions can contain structure definitions), a record statement, a union definition, or a **%fill** component. The structure name can be omitted in a structure definition if and only if that structure definition is an abx-component-def and has a record list. A structure-type name can be used in any context legal for a sequence derived-type name.

A **%fill** component is a component-def-stmt with a component name of **%fill**; such a component, which cannot be referenced, serves to "pad" the storage sequence the specified amount in order to achieve the desired alignment of the other components. For example,

```
structure /my_struct/
  integer(1) :: first_byte
  integer(1) :: %fill
  integer(2) :: align_second_16
end structure
```

explicitly puts a padding byte between **first_byte** and **align_second_16**.

A *record* is a scalar or array variable having the specified structure type; if it is an array, it may be dimensioned either in the record list or in a dimension statement. A record variable may be declared in the structure definition itself or in a separate record statement, specifying the structured-type name (or a sequence derived-type name):

record / *structure-name* **/** *record-list* [**, /** *structure-name* **/** *record-list*]...

A *union* defines a data area which is shared by two or more groups of *fields* and has form

```
union
map-definition
map-definition                    ! note that a union must contain at least two map definitions
[ map-definition ]...
end union
```

where a *map-definition* is

```
map
field-declaration
[ field-declaration ]...
end map
```

A *field-declaration* is a derived-type component declaration, a structure definition, a record statement, or a union. A map definition defines a storage sequence and the map definitions in a given union definition are storage associated. The storage size of a union is the size of its largest map definition. The principal uses of unions are as components in structure definitions and as map fields, but unions may also be used as components in sequence derived-type definitions. An object containing a union must not appear in an I/O list, and the name of a derived type containing a union must not be used as the name of a structure constructor.

Individual structure type components may be referenced with the % component selection operator, just as in derived type objects; in addition the dot (decimal point) may be used instead of the % (for both structure types and derived types), but in this case the component names must not be the same as the names of the intrinsic dot operators or any user-defined dot operators. For example, given the declarations

```
type h; sequence; integer t; end type
structure /x/ g; type(h) :: gt; end structure
g%gt%t          ! is a legal reference
g.gt.t          ! is not a legal reference, but would be if "gt" were spelled, say, "tg" instead
```

A principal rationale for structures and unions is to improve Fortran's interoperability with C - structures are equivalent to C structs and unions are equivalent to C unions.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

cray-style pointers. Another important, and widely-implemented, aid to interoperability is a *pointer data type*, having address values. Note that standard Fortran 90 pointers are attributes, not data types; objects may have the pointer attribute but are not considered to be pointer objects - the term "pointer object" applies to objects of this extended pointer type. Declarations of such pointer objects have the form:

pointer (*pointer-name* **,** *target* **)** [**,** **(** *pointer-name* **,** *target* **)**]...

where *pointer-name* is the name of the pointer variable being declared and *target* may be any Fortran object, including a structure component, or an external function name. When such a pointer is used as a dummy argument, the intent attribute applies to the pointer value, not the value of the pointed-to object (if any).

The intrinsic functions **loc** and **pointer** are added (see below), each of which returns the address (pointer value) of its argument.

An example serves to illustrate the use of this pointer type; note that strict type-checking of pointer arguments may be relaxed in "unambiguous" cases:

```
module Mytypes
  type Rect; sequence
    integer(2) :: top
  end type
  type Picture; sequence
    integer(2) :: picSize
  end type
end module

module Mod
  interface
  subroutine drawPicture(h_myPicture, dstRect)
    use mytypes
    type(Picture) :: myPicture
    pointer(p_myPicture, myPicture)          ! pointer to a derived-type object
    pointer(h_myPicture, p_myPicture)        ! pointer to a pointer
    value :: h_myPicture                     ! pass the pointer by value
    type(Rect) :: dstRect
  end subroutine
  end interface
end module

subroutine foobar
  use Mod
  type(Rect) :: dstRect
  pointer (p,i)
  call drawPicture(p,dstRect)                ! legal - “p” is a pointer
  call drawPicture(loc(j),dstRect)           ! legal - type of ”loc” is pointer
  call drawPicture(708089,dstRect)           ! error - an integer constant is not a pointer
end                                          !     (but may compile with a warning)
```

Pointer type matching is utilized, however, when resolving references to generic interfaces; for example:

```
interface bogus
  subroutine point_bogus(p)
    pointer (p,i)
  end subroutine
  subroutine real_bogus(z)
    real z
  end subroutine
  subroutine int_bogus(i)
    integer i
  end subroutine
end interface
call bogus(0)                                ! reference to int_bogus
call bogus(loc(z))                           ! reference to point_bogus
call bogus(1.0)                              ! reference to real_bogus
end
```

Record statements in a structure definition may specify targets of the type of the structure being defined. For example, the following is legal:

```
structure /outer/                    ! but this is illegal:
  record /outer/ pointee             !     structure /outer/
  pointer (next, pointee)            !     structure /outer/ pointee
    . . .                            !       . . .
end structure                        !     end structure
```

All of the records of such record statements (that specify the name of the structure being defined) must appear as targets in pointer declarations in that structure, and forward references to subsequent structure definitions are illegal. For example:

```
structure /outer/
 structure /inone/
  structure /intwo/
   record /outer/ junk              ! legal reference because of...
   pointer (p_outer, junk)          ! ...this pointer declaration
  end structure
  record /spaced/ nogood            ! illegal forward reference
 end structure
 record /outer/ circle              ! illegal - “circle” is not a pointer target
end structure
structure /spaced/
 integer out
end structure
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Attribute Extensions. Several attributes are added to the standard set of attributes:

attribute	effect
automatic	variable s are allocated on the stack; incompatible with static, save, and common
static	equivalent to save, but statement form must have an object list
value	applies to dummy arguments - specifies pass-by-value
volatile	assignments and references occur, even if optimizations have eliminated these objects
stdcall	standard calling sequence specified for procedures with this attribute[a]
dll_import	tags a procedure name as coming from a DLL
dll_export	tags a procedure name as an entry to be exported to a DLL

a. The stdcall attribute keyword can, alternatively, be specified on the function or subroutine statement, in the same manner as the recursive keyword; stdcall and recursive are mutually exclusive - either one or the other, or neither, appears. If stdcall appears on a subroutine statement, the parentheses for the optional dummy argument list must also appear (otherwise the statement looks like, and is interpreted as, a stdcall attribute statement rather than a subroutine statement).

These attributes can be specified either in type declarations statements, in the normal way, or in attribute statements, with syntax similar to that for the standard attributes:

automatic [[**::**] *sym-name-list*]

static [**::**] *sym-name-list*

value [**::**] *sym-name-list*

volatile [[**::**] *sym-common-name-list*]

stdcall [**::**] *procedure-name-list*

dll_import [**::**] *procedure-name-list*

dll_export [**::**] *procedure-name-list*

The name list is not required for the **automatic** and **volatile** statements, and if omitted that attribute is applied to all of the local objects in the scope. A *sym-name* is an object name and an *sym-common-name* in the **volatile** statement can be either an object name or **/** *common-name* **/**. The **stdcall** attribute can be specified only for external procedure names.

Module objects are inherently static/save, and **automatic** (or the stack directive) cannot be specified within the scope of a module. Otherwise **automatic** and **static** take precedence over save without a sym-name-list, the stack directive, and the **-ev** command-line option. (Automatic is the same as the stack directive, and the -ev option is the same as save without a sym-name-list.)

The **value** attribute is incompatible (must not be used) with these other attributes: external, intent, intrinsic, optional, parameter, pointer, private, public, save and stdcall. If the interface of a procedure having a value dummy argument is explicit, all associated actual arguments will be passed by value.

The **stdcall** attribute is incompatible with these other attributes: allocatable, intent, parameter, pointer, target, save, and value. Stdcall functions cannot be: assumed-length (len=*), variable length (len=n) character functions, array-valued functions, derived-type functions, or storage associated in any way. The **stdcall** attribute can be applied *only* to external procedure names, but not to: a function name specified in a **result** clause, a procedure name specified by an **entry** statement, or a generic name specified in an interface block. Because **stdcall** applies only to external functions, it is incompatible with: data initialization, namelist, statement functions, labels, block data, dll_import, and dll_export.

Stdcall is a platform-dependent extension specifically provided for direct communication with the Windows Win32™ API; dll_import and dll_export are intended to interface with DLLs that are not part of the Windows API. See also the compiler options -YIL, -YDLL_STDCALL, and -YDDL_NAMES for compiler settings regarding these attributes.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Intrinsic procedure Extensions.

function	purpose	P[a]	E[b]
acosd, dacosd[c]		x	x
asind, dasind[c]		x	x
atand, datand[c]		x	x
atand, datan2d[c]		x	x
bit_size(*ref***)**	returns the number of bits in the argument - see below		
bitest, bjtest	specifics for generic btest[d]	x	x
carg(*expr***)**	same as %val except for character arguments[e]; actual argument only		
cdabs, i2abs, iiabs, jiabs	specifics for generic abs	x	x
clock	equivalent to date_and_time		
cosd, dcosd[c]		x	x
cotan, dcotan[c]		x	x
cpu_time([time=] *real-variable***)**	subroutine; returns the processor time in seconds, as the argument value		
date, jdate	equivalent to date_and_time		
eof([unit=] *int-expr***)**	returns .true. if unit is connected at[f] end of file; .false. otherwise		
floati, floatj, dfloti, dflotj	specifics for generic float	x	x
i2dim, iidim, jidim	specifics for generic dim	x	x
i2max0, imax0, jmax0, imax1, jmax1, aimax0, ajmax0; purpose: specifics for generic max			x
i2min0, imin0, jmin0, imin1, jmin1, aimin0, ajmin0; purpose: specifics for generic max			x
i2mod, imod, jmod	specifics for generic mod	x	x
i2nint, inint, jnint, iidnnt, jidnnt	specifics for generic nint	x	x
i2sign,iisign, jisign	specifics for generic sign	x	x
ibchng([inta=] *int-expr*, [**intb=**] *int-expr***)**	returns value of inta with bit intb reversed		x
iiand, jiand	specifics for generic iand[d]	x	x
iibclr, jibclr	specifics for generic ibcl[d]	x	x
iibits, jibits	specifics for generic ibits[d]	x	x
iibset, jibset	specifics for generic ibset[d]	x	x
iieor, jieor	specifics for generic ieor[d]	x	x
iior, jior	specifics for generic ior[d]	x	x
iishft, jishft	specifics for generic ishft[d]	x	x
iishftc, jishftc	specifics for generic ishftc[d]	x	x

function	purpose	P[a]	E[b]
imag([**z=**] *complex-expr***)**	same as aimag; may be passed for default kind argument	x	x
inot, jnot	specifics for generic not[d]	x	x
int2, int4, iifix, iint, jint, iidint, jidint, iifix, jifix	purpose: specifics for generic int		x
irtc, rtc	equivalent to system_clock		
isha([**inta=**] *int-expr*, [**intb=**] *int-expr***)**	shift inta the amount specified by intb (positive intb shifts left)		x
ishc([**inta=**] *int-expr*, [**intb=**] *int-expr***)**	same as isha, but circular shift		x
ishl([**inta=**] *int-expr*, [**intb=**] *int-expr***)**	same as isha, but logical shift		x
izext, izext2, jzext, jzext2, jzext4	specifics for generic zext	x	x
loc, log10	extended so that can be passed[d]	x	x
lshift([**i=**] *int-expr*, [**shift=**] *int-expr***)**	same as ishft(i,shift), assuming shift is positive		
pointer(*ref* **)**	returns address of *ref* as a default integer value; otherwise same as loc		
rshift([**inta=**] *int-expr*, [**intb=**] *int-expr***)**	same as ishft(i,-shift), assuming shift is positive		x
secnds([**x=**] *real-variable***)**	subroutine; returns the seconds since midnight, as the argument value		
sind, dsind[c]		x	x
tand, dtand[c]			
zext(*int-expr***)**	returns the integer argument with no sign extension		x
%loc(*expr***)**	the address of ref is passed; applies only to actual arguments		
%val(*expr***)**	*expr* is passed by-values; applies only to actual arguments		

a. can be used as an actual argument - i.e., can be passed
b. function is elemental
c. for these functions the angle is in degrees rather than radians
d. the generic function may be passed
e. %val(%loc(char-expr // 0)) for character arguments (i.e., address of null-terminated string)
f. error if the unit is not currently connected

The **bit_size** function is extended from the standard version, which allows only integer objects as actual arguments; the extended version allows derived-type and structure-type *names* (not objects) as actual arguments as well. In the case of derived-type and structure-type names, the result is the number of bits any *object* of that type will occupy in memory at runtime. For example:

```
type point
  integer(kind=2) :: x, y, z
  integer(kind=1) :: alpha, r, g, b
end type
    . . .
print *, bit_size(Point)              ! will output 80 on a Macintosh
```

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Miscellaneous Extensions.

new open statement specifiers (last two are for the inquire statement)

specifier	effect
form="binary"	can be used only with sequential unformatted files - for stream I/O
action={**"publish"**|**"subscribe"**}	for MacOS/MRWE™
access="window[,*]**"**	for MacOS/MRWE™, the asterisk may be any string, checked at runtime
access="transparent"	same as **form="binary", blocksize=***default-integer-expr*
blocksize=*default-integer-expr*	in bytes
carriagecontrol={**"Fortran"**|**"list"**}	
filetype=*character-expr*	
creator=*character-expr*	
convert={**"big_endian"**|**"little_endian"**}	
access=*character-variable*	**"transparent"** is a valid return value in the **inquire** statement
flen=*default-integer-variable*	returns file length, in bytes, or zero if file is empty or nonexistent

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

If a character constant is appended with a **C**, as in **"now is the time"C**, then (a) a backslash character (\) in the string is interpreted as an "escape character" that converts the subsequent character(s) in accordance with the following table and (b) appends a *null character* (ascii 0) to the end of the string. The intent is to simulate C-style strings.

\a	audible alarm (BEL, ascii 07)	**\t**	horizontal tab (HT, ascii 09)
\b	backspace (BS, ascii 08)	**\v**	vertical tab (VT, ascii 11)
\f	form feed (FF, ascii 12)	**\x**h[h]	hexadecimal digit(s), up to 2
\n	newline (LF, ascii 10)	**\o**o[o[o]]	octal digit(s), up to 3
\r	carriage return (CR, ascii 13)	****	backslash

Any such escape sequence, including the backslash character, is replaced with the indicated character. If a backslash precedes any other character it is ignored (and removed). The **-YCSLASH** command-line compiler option allows these escape sequences to be used in any character constant (i.e., not just C-strings).

Subsequent use of the **len** intrinsic with a C-string value reflects the addition of the null; for example, **len("now\tis\nthe\ttime"C)** has the value 16. Octal and hex values must fall in the 0-255 (decimal) range, **'\'C** is illegal, C-strings may not appear in format statements, and the character constant must be default kind.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Expressions of type integer or pointer may be used as logical expressions in if statements and constructs. The form **if (** *expr* **)** in such a context has the meaning **if ((** *expr***)/=0)**, where *expr* is the integer or pointer expression.

Such expressions also can be used in logical assignment statements: *logical-variable* **=** *expr*, with the (same) effect: *logical-variable* **= (***expr***)/=0**.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

The keyword **recursive** may be omitted from a directly recursive procedure definition.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

The **=** *initialization-expr* in a declaration statement may be replaced by **/** *data-stmt-value* **/** if (and only if) that declaration statement does not contain the **::** separator.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

The **byte** type specifier is added and can be used anywhere the **integer** type specifier can be used; it cannot have a kind value, and is equivalent to integer(1).

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Tabs in columns 1-5 in fixed-form source are interpreted as follows:

- if the next character is a **!**, the line is a comment line,
- else if the next character is a nonzero digit (1-9) it is a continuation character,
- else the next character is the start of a statement.

If the **-f alt_fixed** compiler option is in effect, the interpretation is a bit different:

- if the next character is a letter (a-z or A-Z) it is the start of a statement,
- else the next character is in column 6, with the normal fixed-form interpretation.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

For Absoft compilers running on systems other than Apple MacOS™, symbol names may start with a leading dollar sign (**$**); for example: **subroutine $foo()**; **end**

Implicit typing of **$** is default real, and in the implicit statement, **$** is ordered after **z**; e.g.: **implicit integer (a-$)** makes everything in the program type default integer, including, for example, **$foobar**. (But note that names with leading dollar signs may not be the names of variables associated with a namelist group.)

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

compiler directives

Compiler directives are placed on (separate) lines in the Fortran source code and provide the compiler with additional information over that in the Fortran code itself. Directive lines are identified by an initial token, **!dir$**, not shown in the following table, and appear as comment lines. In fixed-form source **Cdir$** also identifies a directive line, if the **C** is in column 1. All directives, including the directive-identifying token, are case insensitive, and **ms$** and **dec$** are acceptable alternatives to **dir$** in all cases; in addition, if (and only if) the **-YMS7D** compiler option is specified, the initial token may be simply **$** in the first column for the **free**[**form**], **fixed**[**formlinesize**], **nofreeform**, and **pack** directives, but this form (**$** in column 1) should be considered deprecated.

directive	effect
attributes *attr-list* **::** *sym-list*	the possible *attr* values are: **alias, C, reference, stdcall, value, varying,**
free[**form**]	from this point on, source is free-form
nofreeform	from this point on, source is fixed-form
fixed	same as nofreeform
fixedformlinesize: {**72**|**80**|**132**}	line length for fixed-form source
name (*name***="***external-name***")**	mapping between internal (Fortran) names and external (e.g., C) names[a]
pack[**on**] [**=** {**1**|**2**|**4**|**mac68K**}]	pack[on] and packoff specify that sequenced structure fields be aligned on byte,
packoff	even byte, or word (four-byte) boundaries[b] ; default value is 1 (byte)
stack	local scalar objects are allocated on the processor stack

a. The name directive can be applied to external procedure definitions as well as to external procedure names (to put Fortran procedures into other language namespaces)..

b. The mac68k packing is for 68K-Pascal structures, which is character, integer(1), and logical(1) aligned on byte boundaries; all other objects aligned on even-byte boundaries.

The packing directives affect the current program unit being compiled (if there is one), or the next program unit (when there is no current program unit). The packing directive is reset to the default (packoff) after the end of each program unit. A packing directive affects only derived-types found below the directive in the source code.

The alignment of any derived-type object (i.e. not its components) is dependent on the highest alignment bound of any component. This holds true for packed structs, unions, and maps. A union embedded in a derived type will start on a boundary based on the most restrictive member of the union (i.e. padding may be inserted before the base of a union and all maps will start at the padded boundary).

A component with the pointer attribute has an alignment which is the same as the alignment the most restrictive of either a natural-word-sized integer or a machine address on the target machine, regardless of the object type. For example, in **type foo; character, pointer :: p_c; end type** foo has a 32-bit alignment on Pentium™ Pro and PowerPC™ 601.

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

command-line compiler options

Compiler options are placed on the command line, following the command name (**f90**).

option	effect
-c [<name>]	compile to relocatable object code
-d {**a**|**j**|**n**|**p**|**q**|**v**|**B**|**R**}	disable options (multiple options can be specified at once)[a]
-e {**a**|**j**|**n**|**p**|**q**|**v**|**B**|**R**}	enable options (multiple options can be specified at once); same note as for -d
-f <form>	source code format; <form> can be either **free**, **fixed**, or **alt_fixed**
-g	produce debugging information for use with the debugger
-I <search-path>	identify search path for include files; multiple search paths require multiple -I options
-o <name>	specify compiler output file name
-O	optimize the program for faster execution speed
-p <file>	specify module files and/or directories
-s	allocate user declared local variables with save (static) attribute
-v	verbose compilation - echo all process commands used to create the output file(s)
-V	output version number; can be used without file(s) or other options, for example: f90 -V
-w	suppress all warnings
-W <line-length>	line length for fixed form source; must be from the set {72,80,132}
-x <directive>	disable specified source code directive; possible values are **free**, **fixed**, **integer**, **name**, **stack**
-YCHARV=ICHAR	%val(char-entity) is passed as: %val(ichar(char-entity(1:1))); default is to pass it as %val(%loc(char-entity))
-YCOM_NAMES= {**UCS**|**LCS**}	specifies uppercase or lowercase for external common-block names; default is UCS
-YCOM_PFX [**=***prefix-string*]	specifies prefix (including null) for external common-block names; default is _C
-YCOM_SFX [**=***suffix-string*]	specifies suffix (including null) for external common-block names; default is null
-YCSLASH={**0**|**1**}	if 1, any character constant can contain C-string backslash escapes sequences; default is 0
-YEXT_NAMES={**ASIS**|**UCS**|**LCS**}	specifies the case of external procedure names; default is UCS
-YEXT_PFX [**=***prefix-string*]	specifies prefix (including null) for external representation of procedure names
-YEXT_SFX [**=***suffix-string*]	specifies suffix (including null) for external representation of procedure names
-YMS7D	recognize Microsoft form of source code directive, which is **$**<directive> with the **$** in column 1
-YNDFP=1	disallow use of period for component selection; default is that period can be used in place of **%**
-YPEI={**0**|**1**}	1 (the default) makes the pointer type equivalent to integer; 0 turns this off

option	effect
\/---------- MacOS/MPW™-specific options ----------\/	
-launch	launch application after successful compilation
-link <arg>	pass <arg> directly to linker
-mrwe	make an MRWE™ application (the default)
-N9	forces generated code to make very frequent checks for command period
-plainappl	make a plain application (i.e. don't link MRWE™)
-ppc	target the PowerPC™ architecture (the default)
-share	use shared versions of intrinsic libraries and I/O libraries; default is to use static linkage
-tool	make an MPW™ tool
-z <msg-level>	suppress output message by level control (errors, warnings, cautions, notes, comments)[b]
-Z <msg-number-list>	suppress the output of the specified messages (useful for turning off long warning lists)
\/---------- Windows™/PC-specific options ----------\/	
-YDLL_NAMES={ASIS|UCS|LCS}	default treatment of dll_import/dll_export names (see also -YIL=)
-YDLL_STDCALL={0|1}	0 means callee does not pop the argument frame; 1 means the frame is popped
-YIL={AC90,ACC,AC77,MSVC,MSVB,BC,BD,WINAPI}	for dll_import/dll_export; see below
\/---------- Unix™-specific options ----------\/	
-l <library>	specify library names to linker
-L <path>	search path for library names
-m <msg-level>	same as -z <msg-level> in the Mac-specific options
-M<msg-number-list>	same as -Z <msg-number-list> in the Mac-specific options
-r	leave relocation information in file
-S	produce an assembly source listing
-u <sym>	force load of specified library name
-YCFRL={0|1}	location of character length in argument list; 0 (default) at end of list, 1 after character value

a. The meanings of the -e and -d options are:
   a - if enabled compilation will halt after one error is encountered
   j - if enabled causes **do** loops to execute at least once
   n - ANSI warnings generated for nonstandard code
   p - if disabled then all **double precision** is internally treated as real with default kind
   q - if disabled the compiler will continue parsing code after 100 errors (the default is to stop compilation when the error count reaches 100)
   v - specify **save** for all local objects in all program units
   R - give all functions and subroutines the **recursive** attribute
   B - disable to run front end only, to check for errors (no object code written)

b. The possible values in the -m<msg-level> and -z<msg-level> options are:
   0 - compiler issues errors, warnings, cautions, notes, and comments
   1 - compiler issues errors, warnings, cautions, and notes
   2 - compiler issues errors, warnings, and cautions
   3 - compiler issues errors and warnings
   4 - compiler issues errors
   The default value is 3.

The -YIL= Windows option controls the calling mechanism and name mangling used in the machine code when creating LIB and DLL files. The following table summarizes the effect of the various -YIL= option values:

value	call mechanism	name mangle[a]
AC90[b]	default	uppercase
ACC	default	asis
AC77	default	asis
MSVC	stdcall (callee pop)	asis & @argsize
MSVB	stdcall	asis
BC	stdcall	asis
BD	stdcall	asis
WINAPI	stdcall	asis & @argsize

a. **!dir$ name** directive takes precedence

b. AC90 is the default, if the -YIL= option not present

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Trademark acknowledgments:

◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊

Appendix A

Fortran 95 Features

The Fortran 95 language features of Absoft ProFortran are described in this appendix. Fortran 95 extends the Fortran 90 language with extensions to existing language features and the addition of certain new features. These extensions and additions include the `WHERE` statement and construct; the `FORALL` statement and construct; and a number of intrinsic functions including `NULL`, `CPU_TIME`, `CEILING`, `FLOOR`, `MAXLOC`, and `MINLOC`.

WHERE

The `WHERE` keyword can be used both as a statement and a construct, similar to the `IF` keyword. `WHERE` is used to perform masked array assignments, applying a logical test to each element of an array. The syntax of the `WHERE` statement is:

```
WHERE ( mask_expr )assign_stmt
```

where: *mask_expr* is a logical array expression

assign_stmt is an array assignment statement. The shape of the array must be the same as the shape of the array used in the *mask_expr*

In the following example, the arcsine function will only be evaluated if the absolute value of the element of the array `a` is less than or equal to 1.0.

```
REAL a(100), b(100)
.
.
.
WHERE (ABS(a) <= 1.0) b = ASIN(a)
```

The syntax of the `WHERE` construct is:

```
[name:] WHERE ( mask_expr )
     [where_body_construct]
[ELSEWHERE ( mask_expr ) [name]
     where_body_construct]
[ELSEWHERE [name]
     where_body_construct]
END WHERE [name]
```

where: *mask_expr* is a logical array expression

where_body_construct is an array assignment statement or a WHERE statement or construct. The shape of all arrays must be the same as the shape of the array used in the *mask_expr*

FORALL

The FORALL keyword can be used both as a statement and a structure. It is similar to the masked array assignment WHERE, but is more general, allowing more array shapes to be assigned. It is used to perform array assignments, possibly masked, on an element by element basis. The syntax of the FORALL statement is:

```
FORALL (triplet_spec [,triplet_spec]… [,mask_expr] )assign_stmt
```

where: *triplet_spec* is a triplet specification of an index variable normally used as an array element index. It has the following form:

```
index = subscript : subscript [: stride]
```

where: *index* is a scalar integer variable. It is valid only with the scope of the FORALL statement

subscript is a scalar integer expression and may not contain a reference to any *index* in the *triplet_spec* in which it appears

stride is a scalar integer expression and may not be zero. If omitted, a default value of 1 is supplied. It. may not contain a reference to any *index* in the *triplet_spec* in which it appears

mask_expr is any logical scalar expression, including one which references an *index* of a *triplet_spec*.

assign_stmt is an assignment statement or a pointer assignment statement.

In the following example, every element of the array a is assigned the value 1.0.

```
REAL a(100, 100)
.
.
.
FORALL (i=1:100, j=1,100) a(i,j) = 1.0
```

The syntax of the FORALL construct is:

```
[name:] FORALL (triplet_spec [,triplet_spec]… [,mask_expr] )assign_stmt
     forall_body_construct
END FORALL [name]
```

where: *triplet_spec* is a triplet specification of an index variable normally used as an array element index. It has the following form:

```
index = subscript : subscript [: stride]
```

where: *index* is a scalar integer variable. It is valid only with the scope of the FORALL statement

subscript is a scalar integer expression and may not contain a reference to any *index* in the *triplet_spec* in which it appears

stride is a scalar integer expression and may not be zero. If omitted, a default value of 1 is supplied. It. may not contain a reference to any *index* in the *triplet_spec* in which it appears

mask_expr is any logical scalar expression, including one which references an *index* of a *triplet_spec*.

forall_body_construct is an assignment statement, pointer assignment statement, WHERE statement or construct, or FORALL statement or construct.

CPU_TIME

CPU_TIME is subroutine that returns the processor time. The calling sequence is:

```
CALL CPU_TIME(time)
```

where: *time* is a scalar real variable. It an INTENT(OUT) argument that is assigned a processor-dependent approximation of the processor time in seconds

NULL

Null is a transformational intrinsic function that returns a disassociated pointer. The referencing sequence is:

```
NULL ([mold])
```

where: *mold* is a pointer of any type. Its association status can be undefined, disassociated, or associated. If its status is associated, the target does not have to be defined. If *mold* is present the result type is the same as *mold*; otherwise the result type is determined by the context

CEILING

CEILING is an elemental intrinsic function that returns the smallest integer greater than or equal to its argument. The referencing sequence is:

```
CEILING (a [,kind])
```

where: *a* is of type real

kind is a scalar integer initialization expression

FLOOR

FLOOR is an elemental intrinsic function that returns the greatest integer less than or equal to its argument. The referencing sequence is:

```
FLOOR (a [,kind])
```

where: *a* is of type real

kind is a scalar integer initialization expression

MAXLOC

MAXLOC is a transformational intrinsic function that returns the maximum value of the elements in an array, a set of the array elements, or along a specified array dimension. The referencing sequence is:

```
MAXLOC (array [,dim] [,mask])
```

where: *array* is an array of type integer or real

dim is a scalar integer that must be less than or equal to the rank of the array

mask is a logical array and must be conformable with *array*

MINLOC

MINLOC is a transformational intrinsic function that returns the value of the elements in an array, a set of the array elements, or along a specified array dimension. The referencing sequence is:

```
MAXLOC (array [,dim] [,mask])
```

where: *array* is an array of type integer or real

dim is a scalar integer that must be less than or equal to the rank of the array

mask is a logical array and must be conformable with *array*

Index

syntax terms